Efficient Query Processing in DHT-based RDF Stores

vorgelegt von
Dominic Battré, MSc
aus Berlin

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
- Dr. rer. nat. -

genehmigte Dissertation

Promotionsausschuss:
Vorsitzender: Prof. Dr.-Ing. Sahin Albayrak
Gutachter: Prof. Dr. Odej Kao
Prof. Dr. Volker Markl
Dr. Karim Djemame

Tag der wissenschaftlichen Aussprache: 18. Dezember 2008

Berlin 2008

D 83
Acknowledgement

I would like to take the chance of expressing my appreciation to the many people who helped me during the course of my studies and my academic career and making this work possible. First and foremost I would like to thank my advisor Odej Kao for his support and guidance during the entire time. Also, I would like to express my appreciation to Volker Markl and Karim Djemame for agreeing to review this thesis.

This work would not have been possible without the great discussions and the fruitful collaboration with my colleagues Felix Heine and André Höing and our students Björn Lohrmann, Philipp Richter, and (now colleague) Martin Raack. Special thanks go to Matthias Hovestadt for spending countless days and nights helping with AssessGrid when deliverables and reviews were approaching fast and little was working as expected, and for writing many great papers with me. The same goes to Kerstin Voß. Without her efforts and talent for organization, our project could not have been as successful. Special thanks also go to Kerstin, Philipp Berndt, André Höing, Axel Keller, and Martin Raack for proofreading my thesis. Furthermore, I shall not forget the Atlanta based soft drink company for their invention of caffeinated soft drinks. You all made this thesis become reality.

Thanks go also to my entire working groups in Berlin and Paderborn for making my life and research at PC² and TU-Berlin so incredibly enjoyable and satisfying!

Furthermore, I would like to say thank you to my friends from #phreax. You guys are a more comprehensive encyclopedia of computer science, \LaTeX, and geek life than could ever be written. Special thanks go to Tobias Thierer, who was one of the reasons for me getting into the Bundeswettbewerb Informatik, which has played an important role in my life and has opened many doors for me. Also, I would like to say thank you to the Studienstiftung des Deutschen Volkes, the Fulbright Commission, and the European Commission for their support during my academic career.

Finally, I would like to thank my dear family for their love and support.
Abstract

The growth of information in modern society makes it more and more difficult to find relevant information that supports people at their tasks. Therefore, we strive to give information a well-defined meaning for better enabling computers and people to work in cooperation. The prevalent representation for semantic information is the Resource Description Format (RDF). RDF describes information as sets of triples or equivalently as directed graphs. It provides the foundation for defining ontologies, annotating objects and concepts, information integration, and knowledge inference. Its applications are found in the areas of Semantic Web, Semantic Desktop, Grid computing, e-Business, social networks, natural sciences, and many others.

This thesis presents an indexing schema for RDF triples that is based on a distributed hash table (DHT), a structured peer-to-peer overlay network, in order to improve scalability and reliability. Scalability is improved in comparison to centralized triple stores as the peers participating in the DHT share the storage and the query load. Reliability is achieved by replication and fault tolerant routing mechanisms.

The core question addressed by this thesis is how queries can be processed efficiently given the fact that data relevant for the query is spread over the network. The thesis presents and compares novel strategies to estimate the effort of retrieving remote data and mechanisms to reduce the amount of data that needs to be transferred. It presents a caching mechanism that strives to reuse intermediate results of previous queries. Finally, it discusses strategies to balance the query and storage load of peers. The success of the strategies is demonstrated by benchmarks with the Lehigh University Benchmark.
Zusammenfassung


Diese Dissertation beschreibt ein Indizierungsschema für RDF Tripel auf der Basis einer verteilten Hashtabelle (DHT), einem strukturierten Peer-to-Peer Overlay-Netzwerk, um Skalierbarkeit und Verlässlichkeit zu verbessern. Die Skalierbarkeit wird im Vergleich zu konventionellen zentralen Tripel-Speichern verbessert, indem die Knoten der DHT die Speicher- und Anfragelast untereinander teilen. Verlässlichkeit wird durch Replikation und fehlertolerante Routingmechanismen erreicht.

Contents

1 Introduction ............................................. 1
   1.1 Problem Definition .................................. 4
   1.2 Contribution .......................................... 6
   1.3 Outline of the Thesis .................................. 9

2 Semantic Web Foundations .................................. 11
   2.1 Data Representation .................................. 11
   2.2 RDF .................................................. 13
   2.3 Ontologies ............................................. 19
   2.4 RDF Schema ........................................... 22
   2.5 OWL ................................................... 25
   2.6 SPARQL ............................................... 26
   2.7 Applications .......................................... 28
   2.8 Appraisal of Semantic Web Technologies ................. 30

3 Storing RDF triples in DHTs ................................ 33
   3.1 Peer-to-Peer Networks ................................ 33
      3.1.1 Classes of Peer-to-Peer Networks ................. 34
      3.1.2 Pastry ............................................. 36
   3.2 Triple Storage ......................................... 38
      3.2.1 Life-Cycle of Triples ............................. 41
      3.2.2 Node Departure .................................... 42
      3.2.3 Node Arrival ....................................... 42
      3.2.4 RDFS Rules ......................................... 43
      3.2.5 Local Storage ..................................... 44
      3.2.6 Triple Lookup ..................................... 44
      3.2.7 Bloom Filters ...................................... 45
   3.3 Related Work .......................................... 48
      3.3.1 Centralized Triple Stores ......................... 48
      3.3.2 Distributed Triple Stores ......................... 51
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.1</td>
<td>Foundation</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>4.2</td>
<td>Outline of Query Processing</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>4.3</td>
<td>Query Evaluation Agents</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>4.4</td>
<td>Selection of Lookups</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>4.4.1</td>
<td>Local Heuristics</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>4.4.2</td>
<td>Network Heuristics</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>4.4.3</td>
<td>Caches</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>4.4.4</td>
<td>Network Heuristics (cont.)</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>4.4.5</td>
<td>Wrappers</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>4.5</td>
<td>Migration</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>4.6</td>
<td>Processing Triple Patterns</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td>4.7</td>
<td>Evaluation</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>4.7.1</td>
<td>Local Heuristics</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>4.7.2</td>
<td>Network Heuristics</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>4.7.3</td>
<td>Estimation Accuracy</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>4.7.4</td>
<td>Wrappers and Variations</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>4.8</td>
<td>Related work</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>4.9</td>
<td>Conclusion</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>4.10</td>
<td>Outlook</td>
<td>108</td>
</tr>
<tr>
<td>5</td>
<td>5.1</td>
<td>Hashing Intermediate Results</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>Containment of Subqueries</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>Bottom Up Search for Intermediate Results</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>Improved Bottom Up Search for Intermediate Results</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>5.5</td>
<td>Updates and Replacement</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>5.6</td>
<td>Evaluation</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td>5.7</td>
<td>Related Work</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>5.8</td>
<td>Conclusion</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>5.9</td>
<td>Outlook</td>
<td>124</td>
</tr>
<tr>
<td>6</td>
<td>6.1</td>
<td>Related Work</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>Strategies</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>6.2.1</td>
<td>Replication</td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>6.2.2</td>
<td>Relocation</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Overload Detection</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>Benchmark and Analysis</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>6.4.1</td>
<td>Experimental Setup</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>6.4.2</td>
<td>Simulation</td>
<td>137</td>
</tr>
</tbody>
</table>
6.5 Impact on Triple Management ........................................ 141
  6.5.1 Overlay Tree .................................................. 141
  6.5.2 Compensation of Node Failures ................................. 143
  6.5.3 Soft-State Updates ............................................ 145
  6.5.4 RDFS Rules .................................................... 145
6.6 Conclusion .......................................................... 146
6.7 Outlook .............................................................. 146

7 Conclusion .............................................................. 151
1 Introduction

Contents

1.1 Problem Definition ........................................ 4
1.2 Contribution ................................................ 6
1.3 Outline of the Thesis ........................................ 9

The growth of information in modern society makes it more and more difficult to find relevant information that supports people at their tasks. Computer science has been dealing with this issue over the past decades. Most prominently, the World Wide Web has developed into an omnipresent information source that has become a crucial tool in our everyday life. Search engines help human users master the vast amounts of information that keep growing at an increasing pace. However, the World Wide Web as we know it today has several issues that inspired the idea of the Semantic Web as described by Tim Berners-Lee’s visionary article [32] in the Scientific American.

The amount of information available on the World Wide Web has grown to a level where almost any query can be answered with the knowledge that is stored somewhere on the Web. Search engines crawl and index huge parts of the Web but support users only partially at their task of finding answers to their questions. Instead of answering questions, search engines provide keyword based searching that points to websites and delivers mediocre results. A serious problem is that search engines often deliver result sets of high recall but low precision. This means that the results to a query contain a huge part of the relevant information available on the Web (high recall) but are mixed up with a lot of irrelevant information (low precision). This forces the user to manually browse through the results in order to identify the really relevant information. Conversely to this, other queries suffer from very little or no recall. Time tables of public transportation can be considered as an example. Search engines usually cannot usurp the information that is buried in databases and can only be queried using specific APIs. But even if time tables are openly accessible as web documents, search engines see nothing but large sets of numbers without any meaning attached.

A major drawback of the current World Wide Web originates from the chosen means of data representation: Information is represented in a layout-focused format to
1 Introduction

provide human-readable content. While this way of representing information makes websites pleasant to read and navigate for the human reader, it makes it difficult for machines to process the information and give concise answers to users’ queries.

Tim Berners-Lee defines his vision of the Semantic Web in [32] as follows:

“The Semantic Web is [...] an extension of the current [web], in which information is given well-defined meaning, better enabling computers and people to work in cooperation.”

The following definition is attributed to the W3C [170]:

“The Semantic Web is a vision: the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications. In order to make this vision a reality for the Web, supporting standards, technologies and policies must be designed to enable machines to make more sense of the Web, with the result of making the Web more useful for humans. Facilities and technologies to put machine-understandable data on the Web are rapidly becoming a high priority for many communities. For the Web to scale, programs must be able to share and process data even when these programs have been designed totally independently. The Web can reach its full potential only if it becomes a place where data can be shared and processed by automated tools as well as by people.”

Thomas B. Passin presents a wide spectrum of further expectations and ideas of the Semantic Web from various points of view in his book [136].

If information becomes increasingly represented in a machine processable format, search engines can take the next step and respond to a user’s query with answers instead of addresses of websites. Searching for the European country with the greatest or lowest gross domestic product (GDP) is for example very challenging nowadays—unless some website gives this information explicitly with a similar phrasing so that the user can guess suitable key words. The Powerset search engine, which is praised for its natural language processing capabilities and delivering direct answers instead of website addresses, answers the question “What is the European country with the lowest GDP?” with “Europe: Area 10,180,000 Square kilometers”. The first match in the result set describes the economy of Mexico, the second match shows Portugal having “the lowest GDP per capita of Western European countries” (emphasis added). Search engine leader Google finds a list of European countries sorted by GDP but does not answer the question directly.
The Semantic Web provides the means so that search engines can some day answer queries as the one given above. A country can be annotated to be member of the European Union, and numbers representing the gross domestic product, gross income per capita, etc., can be labeled as such. A search engine for the World Wide Web requires the occurrence of the terms “European country” and “gross domestic product” on the same website in order to consider this site a match. It cannot use the information “Germany has a GDP of ...” because it does not “know” that Germany is a member of the European Union and GDP is an abbreviation for gross domestic product (though the latter is increasingly supported by thesauri of search engines). The task of finding the European country with the greatest or lowest GDP is far beyond the capabilities of current search engines because it requires two non-trivial steps: First identifying the meaning of numbers (if the terms “gross income per capita” and “gross domestic product” appear in close proximity to some numbers it is difficult to match these), and, once all but only relevant data is collected, aggregation for finding the maximum or minimum.

The probably biggest advantage of the Semantic Web can be summarized as enabling the use of data in ways that have not been foreseen when the data was entered. Topological information like the membership of Germany in the European Union might be stored on a web server of the European Union. Romania’s GDP might be stored on the government’s web server. Germany’s GDP might be stored on a website comparing the G8’s environmental pollution in relation to their GDP. The vision of the Semantic Web is that these heterogeneous information sources can be combined in order to find the answer to the query stated above. Further advancements of the Semantic Web, such as the introduction of globally unique identifiers and knowledge inference, will be described in the following chapters.

The Resource Description Framework (RDF) [119] and RDF Schema [39] have become de facto standards for storing ontologies (see chapter 2 for a definition) and semantically rich data in the Semantic Web. With major movements like the Semantic Web [32] and the Semantic Grid [64] gaining momentum, the amount of semantically annotated data available in the future will likely exceed the capabilities of current popular RDF data stores and reasoning engines, such as Sesame [41] or Jena [50, 172], as we have explained in [95, 21], and managing such amounts of information will become a real challenge. Because of scalability issues we and several other research groups investigate the use of peer-to-peer based data stores for managing RDF data. Examples for such peer-to-peer based approaches include our BabelPeers [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 94, 95] implementation, Atlas [99, 103, 111, 113, 114], RDFPeers [47], GridVine [5, 60], and several others.

Felix Heine was among the first to do research on DHT-based RDF stores with his work “P2P based RDF Querying and Reasoning for Grid Resource Description and
1 Introduction

Matching” [94]. His work presents several strategies for RDF query processing in DHTs and is continued in this dissertation.

1.1 Problem Definition

Many peer-to-peer based RDF stores presented in the past follow a common strategy, where RDF triples, consisting of the three components subject, predicate, and object, are stored in a distributed hash table (DHT) three times, once hashed by each component. This approach has several advantages over centralized repositories. Data is distributed between many nodes that can process queries concurrently. This increases not only the available storage capacity for triples but also the available CPU power for query processing. While individual peers may be poorly connected the total bandwidth of a large peer-to-peer network can be extremely large. Furthermore, a global repository established in a DHT simplifies the integration of data from many distributed information sources compared to approaches based on many smaller independent repositories as suggested by Stuckenschmidt et al. [167] and Nejdl et al. [128, 130].

Several topics are important for the performance of DHT-based RDF stores. These include issues such as reasoning, local data storage, query processing, data updates, load balancing, caching, and others. Felix Heine’s dissertation [94] opened up the field and touched on many of these topics. While his work showed the feasibility and potential of DHT-based RDF stores to store and handle large quantities of RDF data, several areas remained open for further research. This dissertation focuses on the aspects of query processing in DHT-based RDF stores. The problem addressed by this thesis can therefore be summarized as

Given a Chord-like DHT that stores RDF triples indexed by subject, predicate, and object, how can one efficiently process RDF queries?

In the context of this thesis, RDF queries are expressed as base graph patterns (defined formally in the following chapters), which form a fundamental basis for the SPARQL query language [143].

The term efficiency covers a broad spectrum of aspects. The principal idea of DHT-based RDF stores is to disseminate all RDF triples from many sources into a structured overlay network and to collect triples again during query processing. In this step it is important to collect only those pieces of information that are relevant to the specific query in order to limit network traffic. Experiments by two independent parties have shown that even in a well connected cluster the network was a bottle neck of the query evaluation [18, 89].
1.1 Problem Definition

(a) Network load due to many concurrent queries.

(b) Network load due to many concurrent answers.

Figure 1.1: Network traffic as a bottle neck in query processing.

In a DHT, efficiency with regard to network traffic can be expressed in two metrics: in number of bytes sent and in number of messages sent. These metrics are relevant for several reasons. The network is in particular a bottle neck if the peers are not connected by high bandwidth interconnects as they can be found in special purpose cluster systems. The advent of large multi-core CPUs will increase the severity of this issue even further. In case many peers submit queries concurrently to a node, this node’s outbound bandwidth becomes the limiting factor when responding (c. f. figure 1.1a). If many peers respond to a node’s request concurrently, these peers can easily fill the receiving node’s inbound bandwidth (c. f. figure 1.1b). The second reason is that large messages need a long time to be completely delivered. As the Bloom filters [36] used in the BabelPeers implementation [20] imply alternating phases of CPU and network activity, long transfer times slow down the process. Figure 1.2a shows the alternating phases of network communication and query processing and how these sum up to the total query processing time.

A large number of messages have detrimental effects as well. Many small messages put high load on the DHT routing layer and increase the probability that any of the responding nodes is suffering from high load and that because of this a message is buffered in an input-queue for some time before being processed. If answers to all messages are required for proceeding further, this delays the entire progress of a query as well, as shown in figure 1.2b.

Another problem arises from load-imbalances. If a node receives too many requests or answer messages are too large, this node becomes a bottle neck and slows the query processing down. This needs to be addressed as well.
1 Introduction

Figure 1.2: Delay of processing due to communication.

For these reasons, the goal of the thesis can be further refined as:

- to find new strategies that reduce the network overhead during query processing and mitigate the problem of load imbalances.

During all this, it needs to be kept in mind that peer-to-peer networks do not provide global knowledge nor centralized services.

1.2 Contribution

The thesis is based on a triple indexing schema in distributed hash tables that has been employed by Heine [94], Cai et al. [45], Aberer et al. [5], Liarou et al. [113] and others.

The contributions of this thesis can be found in two major areas, reduction of network traffic during query processing and increasing query throughput due to a reduction of bottlenecks by load balancing.

The network traffic consumed during query processing is reduced by several novel strategies. The thesis presents a framework and heuristics to accurately estimate the selectivity of triple patterns during the course of query evaluation at low cost. This estimation allows for better query planning and the construction of Bloom filters that cut the network traffic dramatically. Strategies to exploit data locality contribute further towards this goal. Finally, strategies are described that allow
1.2 Contribution

caching and reuse of intermediate results of previously evaluated RDF queries to prevent repetitive efforts.

The reduction of bottlenecks is addressed by a load-detection and balancing schema and an outlook to a second indexing and load balancing schema that have not been used for RDF data before.

Parts of this thesis have been published in the following publications:

Journals

1. Dominic Battré, Odej Kao
   Query Processing in DHT based RDF stores
   In: Journal of Internet Technology, accepted for publication, 2009

2. Dominic Battré
   Caching of Intermediate Results in DHT-based RDF Stores
   International Journal of Metadata, Semantics and Ontologies (IJMSO), 3(1), pp. 84-93, 2008

Book chapters

3. Dominic Battré, Felix Heine, André Höing, Odej Kao
   BabelPeers: P2P based Semantic Grid Resource Discovery

Proceedings

4. Dominic Battré
   Query Planning in DHT based RDF stores

5. Dominic Battré, André Höing, Matthias Hovestadt, Odej Kao
   Dynamic knowledge in DHT based RDF stores
1 Introduction

6. Dominic Battré, André Höing, Matthias Hovestadt, Odej Kao, Ulf Rerrer-Brusch
   Adopting DataGuides to RDF Databases

7. Dominic Battré, Giovanni Cortese, Felix Heine, André Höing
   Accessing XML Documents using Semantic Meta Data in a P2P Environment

8. Dominic Battré, Felix Heine, André Höing, Odej Kao
   On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF stores

9. Dominic Battré, Felix Heine, André Höing, Odej Kao
   Load-balancing in P2P based RDF stores

10. Dominic Battré, Felix Heine, Odej Kao
    Top k RDF Query Evaluation in Structured P2P Networks

Technical Reports

11. Dominic Battré, Felix Heine, André Höing, Odej Kao
    Scalable Semantic Query Evaluation by Query Rewriting

12. Giovanni Cortese, Luca Lorenz, Dominic Battré, Felix Heine, André Höing, Odej Kao
    Peermarks: P2P based Social Bookmarking
1.3 Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2: Semantic Web Foundations

Chapter 2 gives background information and definitions that help understanding the algorithms proposed in this thesis. This comprises a discussion of RDF, the information model used in this thesis; ontologies; SPARQL, the query language; and a brief overview of application domains that can benefit from Semantic Web technologies.

Chapter 3: Storing RDF triples in DHTs

Chapter 3 presents the foundation how RDF data can be managed in a distributed hash table. This includes information dissemination, expiration, reasoning and other background information relevant for chapters 4 to 6.

Chapter 4: Query Evaluation

Chapter 4 constitutes the first main chapter of the thesis and presents the query processing developed. This covers important contributions such as heuristics for query planning, exploitation of data locality, determination of Bloom filter parameters to reduce the amount of data that needs to be transferred, and many others.

Chapter 5: Caching

Chapter 5 constitutes the second main chapter of the thesis and presents strategies for caching and reusing intermediate results of preceding query evaluations.

Chapter 6: Load balancing

Finally, chapter 6 constitutes the last main chapter of the thesis and presents strategies for load balancing in DHT-based RDF stores. These address the problem of hot-spots in the load distribution at discrete positions in the DHT ID space.

Chapter 7: Conclusion

Chapter 7 concludes the thesis with a summary of the contributions.
1 Introduction
2 Semantic Web Foundations

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data Representation</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>RDF</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Ontologies</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>RDF Schema</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>OWL</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>SPARQL</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>Applications</td>
<td>28</td>
</tr>
<tr>
<td>2.8</td>
<td>Appraisal of Semantic Web Technologies</td>
<td>30</td>
</tr>
</tbody>
</table>

In order to establish the foundation for DHT-based RDF stores and query processing, the thesis begins with a presentation and discussion of relevant Semantic Web technologies. We will first discuss the topic of information representation using semi-structured data and see how RDF is a suitable data format for this. We will then expand the discussion onto ontologies and their representation in order to illustrate the advantages of such a semi-structured data format with respect to information integration. After a brief introduction of SPARQL, the most prominent query language for RDF data, several application scenarios are presented that illustrate the potential of RDF.

2.1 Data Representation

A central problem in information processing is the representation of data. Several data representations have been devised throughout the history of computer science, which allow to store and load information, exchange information between different parties, and query information.

Information can be represented as unstructured (raw), semi-structured, and highly structured data [6]. The most prominent example of unstructured data is continuous text that contains information, which is difficult to grasp and process by computers. Data mining, and text mining in particular, is used to extract useful information from unstructured data. The diametral data type is highly structured data as it can
be found in relational databases. Data is stored as records that share a common format (schema).

Semi-structured data often arises from the integration of different information sources with heterogeneous representations of similar concepts. Several parties might represent data differently or describe different levels of detail (e.g., one party might add additional annotations). Abiteboul summarizes this in [6] under the aspect of structure irregularity. A second important aspect is the implicitness of structure. While a grammar in form of a DTD [38] or XML Schema [69, 72, 139] may exist, the actual structure of data is implicitly encoded and beyond the capabilities of a relational database. While a relational database describes a rigid structure of records in a schema, the data guide plays a more important role for semi-structured data to represent the structure of the actual instance data.

ASN.1 [67], XML [38], and RDF [108] are among the most popular and relevant general purpose languages (meta languages) to describe semi-structured data, each having different characteristics and advantages. We will touch very briefly on each of these data representations and then focus on RDF with its particularities.

ASN.1 is a mature standard published as the X.680-683 and X.690-694 family by the International Telecommunication Union (ITU-U) and as ISO/IEC 8824-* and 8825-* by the ISO/IEC. It allows a compact and platform independent representation of data for exchange between independent systems and provides several atomic data types and the mechanisms to define new complex data types. Objects can have identifiers that can be referenced to create complex networks of related objects. Owing to the focus of compactness and platform independence, ASN.1 is very popular in the telecommunication sector.

XML is a semi-structured tree-like data representation, among whose outstanding features is the human readable serialization. The simplification of data exchange due to standard parsers contributed to the popularity of XML in the Internet world. In particular the simplicity increased the popularity of XML in comparison to SGML [100], a superset of XML. Schema languages as DTD and XML Schema allow describing the structure of XML documents and the validation of XML documents accordingly. Query languages as XPath [56], XQuery [37], and XSL Transformations [55] allow for powerful access and transformation of data. XML provides platform independence but is not as compact as ASN.1. A crucial distinction between these two data representations is the underlying data model. While ASN.1 represents structures and objects, XML is based on a tree model. This has strong impact on the way data is represented and queried.

RDF cannot yet claim having the importance and popularity of ASN.1 and XML.
but possesses several outstanding features that will be highlighted in the following sections.

2.2 RDF

The Resource Description Framework comprises a whole family of specifications:

- RDF Primer [119]
- RDF Vocabulary Description Language 1.0: RDF Schema [39]
- RDF Semantics [92]
- RDF Test Cases [77]

In the following, the focus is on the data model of RDF and RDF Schema.

Tim Berners-Lee motivated in [32] the need for representing data in a semantically annotated way so that machines can process these data. The RDF standard is an outcome of this vision. Shadbolt et al. review the development from a current perspective [156]. They point out that the use of URIs (Universal Resource Identifiers) as a global naming convention is crucial to the Semantic Web: Everything has a globally scoped name (represented by the URI) which allows expressing knowledge about these “things” and relationships between them. Everyone can link to them, refer to them, or retrieve a representation of them. This shifts the focus from document centralized data representation towards data or object centralized representation. Data that is locked in relational databases can be unlocked by becoming globally addressable.

RDF is a data model that is flexible and suitable to model data from various domains. The underlying principle is very similar to the Object Exchange Model presented by Papakonstantinou et al. in [134]. Objects (called “resources” in the RDF context) with identifiers can have attributes and are put into relation to each other. No rigid structure is imposed and only very limited restrictions are made a priori. In particular, it is not required that data is modeled as a tree as required by XML. Thereby, it provides a minimalist knowledge representation for the Web [156]. We will introduce RDF and its advantages by giving an example.

RDF introduction by example  Consider trading platforms like PriceGrabber.com, DealTime.co.uk, idealo.de, et cetera. These platforms present various products, including technical specifications, reviews, and user comments. For each product they show vendors selling the product, including the price, shipping and handling, vendor
ratings, et cetera. This information model is inherently distributed (information
originates from many distributed sources) and semi-structured (vendors present
information at various levels of detail). The benefit of global names becomes
immediately apparent when one considers the true source of information. Technical
specifications originate from the manufacturers; pricing information comes from
vendors; reviews from blogs, magazines, or users. If a global naming schema were in
place in the WWW, it would become instantly visible what information is available
about which product.

In order to fit the data into relational databases, product descriptions are often
encoded as unstructured plain-text. While important features, such as the hard disk
capacity of a laptop, are sometimes extracted into separate attributes and therefore
searchable, the vast majority of information is usually buried in text. Searching
for laptops with graphics cards that have driver support for the Linux operating
system is currently not possible. This is where semi-structured data provides huge
advantages.

As mentioned above, RDF is a language to represent knowledge around named
concepts and identities that are put into relation with each other. This knowl-
dge can be represented in several equivalent forms. An intuitive representation is
that of triples. Consider the need to express that a digital camera model “IXUS
860 SD” is produced by a specific manufacturer “Canon”. This can be modeled
by the triple (ns:IXUS860SD, ns:manufacturer, ns:Canon). The three compo-
nents of the triple are named subject, predicate, and object because a triple can
usually be read as a sentence that makes a statement about entities and their
relationship. The term ns:IXUS860SD in this example is just a name for a con-
cept. This concept can be described further by statements like (ns:IXUS860SD,
ns:resolution, 8MP), (ns:IXUS860SD, ns:supportedMemoryChip, ns:MMC), (ns:
IXUS860SD, ns:supportedMemoryChip, ns:SDHC). When we add more informa-
tion like (ns:SanDiskExtremeIII, rdf:type, ns:SDHC), the model becomes more
complex and we see that a graph structure arises as depicted in figure 2.1.

We see that figure 2.1 uses color codes to distinguish two types of nodes. On the
left hand side, we see in red concrete instances and their relations. On the right
hand side, we see in blue classes and relations of classes. Some nodes are somewhere
in-between. A ns:SanDiskSDCardIII is a concrete instance of the class of SDHC
memory chips produced by a concrete manufacturer, but at the same time, many
instances of this type exist. Therefore, a ns:SanDiskSDCardIII can be considered
an instance and a class at the same time. The nodes :1 and :2 are blank nodes,
i.e. nodes without a global name.

\[^{1}\text{The notation ns:IXUS860SD is based on URIs (Uniform Resource Identifiers), where ns constitutes a representation of a namespace, and IXUS860SD represents an identifier within this namespace.}\]
Figure 2.1: Example of an RDF graph describing a digital camera with fitting flash memory (namespaces omitted for simplicity).
RDF building blocks

After this informal presentation of RDF we will now become more formal. RDF distinguishes between URI references (usually abbreviated as URIs), literals, and blank nodes.

URI references: URI references are globally scoped identifiers referring to so-called “things”, “entities”, or “resources” about whose nature no assumptions are made a priori (see [92]). URI references are expressed as absolute URIs (per RFC2396) with optional fragment identifier. An example of a URI is http://www.w3.org/2000/01/rdf-schema#Class. As many URIs share common prefixes, the QName notation is often used for brevity and readability. In this particular example the prefix rdfs: can be bound to the namespace URI http://www.w3.org/2000/01/rdf-schema#. This allows abbreviating the entire URI from above to rdfs:Class. The original URI described by a QName can be resolved by concatenating the namespace URI and the local name. The set of all possible URIs is denoted with \( U \) in the following.

literals: Literals complement URI references by concepts such as numbers or strings. The difference from a model theoretic perspective is that literals are interpreted just as their values. RDF distinguishes plain literals, typed literals, and XML literals. Plain literals consist of a Unicode string, optionally with a language tag (e.g. "Technische Universität Berlin" or "Technische Universität Berlin"@DE). Typed literals consist of a Unicode string with a datatype URI such as "1"^^xsd:integer. XML literals represent valid XML fragments. The set of all literals shall be denoted herein as \( L \).

blank nodes: Blank nodes (or bnodes) represent concepts whose globally scoped label is either not known or not specified. While blank nodes are similar to URI references, the set of blank nodes, URI references, and literals are pairwise disjoint. The RDF Semantics document [92] describes how blank nodes can be considered as existential variables, “indicating the existence of a thing, without using, or saying anything about, the name of that thing.” The identifier scope of blank nodes is local to an RDF document. The set of blank nodes is denoted with \( B \).

With these building blocks, an RDF graph can be described. As both, the actual data as well as queries, can be modeled as graphs, one distinguishes between the model graph \( T_M \) and the query graph \( T_Q \). A triple \( t \) of the model graph \( T_M \) consists of the values subject, predicate, and object denoted by \( t = (s, p, o) \in T_M \subseteq (U \cup B) \times U \times (U \cup B \cup L) \), where \( U \), \( B \), and \( L \) denote the sets of URIs, blank nodes, and literals respectively. Note that literals can only occur as a triple’s object, never
2.2 RDF

Figure 2.2: Graph interpretation of triple (ex:a, ex:b, ex:c).

as subject or predicate. Blank nodes may only appear as subject or object. As arbitrary URIs and blank nodes may appear as subjects and objects, RDF does not prescribe a tree structure as XML.

Besides the interpretation as triple sets, RDF graphs can be interpreted as labeled, directed, and potentially cyclic graphs. Each triple \( t = (s, p, o) \in T_M \) is considered a directed labeled edge in the graph. The edge links from a node \( s \) to a node \( o \) and is labeled with \( p \), see figure 2.2. If triples share common URIs, blank nodes, or literals in their subject or object position, these shall be interpreted as common nodes in the graph.

Besides the modeling as triple sets or graphs, there are many different ways of RDF serialization, which is required for data exchange. Among the most common formats are N-Triples, N3, and RDF/XML.

N-Triples is a subset of N3, originally proposed by Beckett, where each line contains exactly one triple consisting of a sequence of subject, predicate, object, and a trailing period. Its grammar is defined in [77]. N-Triples files often have the extension ‘.nt’.

N3 (Notion 3) by Berners-Lee [31, 30] extends N-Triples by several abbreviations. Sequences of triples can share common subjects, or subjects and predicates. Special constructs allow representing common prefixes, collections, and literals.

RDF/XML is a complex XML serialization of RDF described in [25]. The RDF graph is encoded by nested XML tags. Nodes of the RDF graph are represented by either rdf:Description XML-tags or literals. These XML-tags use the attributes rdf:about, rdf:resource, and rdf:nodeID to name and refer to each other. This compensates for the tree structure prevailing in XML. Predicates or edges in the RDF graph are represented by nested XML tags. The name of the predicate matches the XML tag used. For a more in-depth explanation of RDF/XML we refer to the specification. RDF/XML has been criticized in [52] as overly complex. The authors discuss several issues of RDF/XML and review several proposed alternatives. Beckett adds in [29] to the list of criticism. Despite strong criticism RDF/XML has evolved into the most prevailing RDF serializations nowadays.
RDFa (Resource Description Framework in attributes) [7, 8] is an extension to XHTML that allows annotating the XHTML markup intended for visual data representation with machine-readable hints [7]. Figure 2.3 illustrates the gap between human and machine interpretation of XHTML documents. RDFa intends to bridge this gap by adding semantic annotations to links and structural elements in XHTML.

After this brief overview of RDF serializations we continue with some of the remaining concepts of the RDF that have not been addressed, yet.

Reification is a process to make statements about triples. It can be used for provenance, i.e. expressing who said something, when, and where some statement was found. Figure 2.4 shows the reified triple depicted in figure 2.2. Three statements describe the subject, predicate, and object of a triple. A fourth statement is used to annotate the construct as an RDF statement. It is important to note that the reified statement depicted in figure 2.4 does not imply the existence of the statement depicted in figure 2.2. RDF Reification allows expressing something about a triple but does not give the assertion that the triple described is true.

While provenance is a very important issue in a Semantic Web that deals with information integration, reification did not become very popular due to the huge overhead induced. A single RDF statement is expanded into four statements by reification.

We mentioned that an important feature of RDF is to contribute distributed information into one global information model by using global identifiers. If several
information sources need to be combined, one has to distinguish between merging graphs and building the union of graphs. The latter is generated by concatenating the N-Triples but is usually not desired. In order to merge RDF graphs, one has to first standardize blank nodes apart, i.e. rename blank nodes such that no two blank nodes of the documents to be merged share a common name. After that, the union of the N-Triples is calculated. This ensures that blank nodes pertain to only their local namespace.

Finally we address data structures defined in RDF and RDFS. Details can be found in [39]. RDF distinguishes between Containers and Collections. This distinction is mainly relevant under the open world assumption that is discussed in section 2.5. Containers represent open sets, i.e. constructs that specify which elements are set-members, but leave it open whether non-specified elements are members of the sets or not. Collections on the other hand represent closed sets. Elements that are not specified to be members of a collection are not contained. RDF vocabulary provides three different kinds of RDF containers. Bags represent unsorted lists; Sequences represent sorted lists; Alternatives represent just that. Only one Collection type is defined that is represented as a recursively linked list.

RDF contains an rdf:type element but is so far oblivious to classes and class hierarchies. These are defined in RDF Schema, which will be discussed below.

2.3 Ontologies

RDF itself is a barebone framework to put “objects” or “things” into relations with each other. Ontologies are a higher level construct that often build on RDF. An often cited definition for ontologies is given by Tom Gruber [81] based on [75]:

An ontology is an explicit specification of a conceptualization [...] objects, concepts, and other entities that are assumed to exist in some area of interest and the relationships that hold among them.
He refines this in [82] with:

[The] essential points of this definition of ontology are

- An ontology defines (specifies) the concepts, relationships, and other distinctions that are relevant for modeling a domain.
- The specification takes the form of the definitions of representational vocabulary (classes, relations, and so forth), which provide meanings for the vocabulary and formal constraints on its coherent use.

An ontology defines the way in which information is *structured* in a very powerful way. If parties agree on common ontologies, this enables *exchange and integration* of data. Furthermore, ontologies can be used to *automatically derive information* that is implicitly contained in data. An *inference process* generates this implicit knowledge as will be described later.

Ontologies can be contrasted against taxonomies. Ontologies are a superset of *taxonomies*. They allow specifying a network of information with logic relations, while taxonomies describe strictly hierarchical structures. Taxonomies are used for example in biology to categorize forms of life and to describe the tree of life, the phylogenetic development of species. Taxonomies are constrained in two ways. First, they represent tree-like hierarchical structures, and second, they are usually restricted to only one kind of relationship such as “descended from” or “part of”. Considering the example of the tree of life, this means that gene transfer, where a species has more than one ancestor species, cannot be represented because of the first constraint. Furthermore, one cannot create one global model that integrates many different relations such as descendence, information which species is parasite or pray to which other species, and annotations on the habitat of species, to name a few.

Ontologies generalize this concept and consist in the sense of the Semantic Web of the following building blocks that will be explained in the following:

- Concepts
- Instances
- Relations
- Inheritance
- Axioms
- Rules

The terms *concepts* and *classes* are synonymous and comparable to classes in the object oriented paradigm. Examples for concepts in the semantic sense may be vertebrates, mammals, and humans. Each concept can have many *instances* such as individual people and animals. Concepts and instances may have *relations* between
2.3 Ontologies

Each other. For example, individual people may have a unidirectional “knows” relation or a bidirectional “is-sibling-of” relation. A special kind of relation between classes is inheritance. A class can be a subclass of another class inheriting all its properties. Likewise, a relation can also be a subrelation of another relation. Axioms represent commonly accepted facts, and rules describe how information can be derived. The following two examples show how implicit information can be made explicit by inference: If an individual is asserted to be a human and the class of humans is asserted to be a subclass of mammals, we can derive that this individual is also a mammal. If individual $A$ is a sibling of individual $B$ and the sibling relation is symmetric, we can derive that $B$ is a sibling of $A$.

Shadbolt et al. describe in [156] that ontologies occur in various shapes. Deep ontologies are often used in science and engineering, e.g., in proteomics and medicine, where a lot of information is contained in the concepts, while instances play a less important role. Shallow ontologies typically contain few and unchanging concepts but are used to organize very large amounts of data (instances such as customers, account numbers, etc.).

The distinction between concepts and instances is represented in the often-encountered terminology of T-Box and A-Box. The T-Box comprises taxonomical knowledge, i.e., information about the concepts like subclass relations and the structural composition of classes. The A-Box comprises assertional knowledge, i.e., assertions about instances like their membership in classes. The T-Box and A-Box can often be handled separately during the inference process. This helps managing larger amounts of data.

Ontologies can be described in many different logics and languages. Important examples are RDF-Schema [39], OWL [51, 65, 93, 122, 137, 161], DAML+OIL [58, DL-Lite [48], EL+ [12], and F-Logic [107]. These languages differ in their expressivity and computational complexity. Lightweight ontology languages employ simple constructs while heavyweight ontology languages provide complex constructs and may even be undecidable. Examples for such constructs are subclassing, transitivity, symmetry, reflexivity, functional relations, negation, cardinality restrictions, and many others. While each of these constructs individually may be computationally tractable, combinations of them result in various complexity classes. A huge body on research exists on the complexity of ontology languages (see e.g., [11, 150, 162]).

The reasoning complexities of OWL-Lite and OWL-DL are for example ExpTime-complete and NExpTime-complete, meaning that the decision problem can be solved by a (non-)deterministic Turing machine in time $O(2^{p(n)})$ for some polynomial $p(n)$. Simpler languages as DL-Lite achieve a data complexity of PTIME and combined data and query complexity of NP-complete [48]. The computational complexity of RDF Schema reasoning has been analyzed by Brujin and Heymans in
Depending on the constructs used by an ontology language, different approaches may be necessary for the inference process. These can be first order logic theorem provers, DL reasoners, rule engines, logic programming languages (like Prolog), or others.

A non-exhaustive list of important ontologies comprises the Dublin Core [101], a metadata element set, which deals with cross-domain information resource description; FOAF [40], an ontology for expression of social networks; and the Gene Ontology [10], a vocabulary to describe biological processes, molecular functions, and cellular components. A survey of ontologies can be found in [171].

The following sections will now look into more detail of two important ontology languages, RDF Schema and OWL. We will begin with RDF Schema, which sets the foundation for OWL.

### 2.4 RDF Schema

The purpose of RDF Schema (RDFS) is to serve as a meta language for defining an ontology with concepts, called classes in RDFS terminology, and instances, called resources. RDF Schema is defined in [39] with semantic interpretations in [92]. It has a very limited set of constructs and axioms that will be described briefly in the following.

A central construct for RDF Schema is the `rdf:type` predicate that describes the membership of a resource in a class. This allows defining a class with a statement like

\[ C \text{ rdf:type rdfs:Class}. \] (2.1)

where \( C \) would be filled with any URI that globally identifies the class. Similarly, it is possible to assert that a resource \( U \) is member of this class \( C \) using the statement

\[ U \text{ rdf:type } C. \] (2.2)

Classes allow partitioning sets of resources into subsets, as each resource may be member of zero, one, or many classes. In fact, every resource is axiomatically defined to be member of the class `rdfs:Resource`, meaning that each resource is always member of at least one class.

RDF Schema provides the transitive `rdfs:subClassOf` relation that allows defining one class to be a subclass of another class as shown in the example below:

\[ C_1 \text{ rdfs:subClassOf } C_2. \] (2.3)
2.4 RDF Schema

With this statement, all instances of $C_1$ become automatically instances of $C_2$ as well. This is illustrated in the following example. Given a set of RDF triples

\[
\begin{align*}
\text{ns:Animal} & \quad \text{rdf:type} & \quad \text{rdfs:Class}. \\
\text{ns:Dog} & \quad \text{rdf:type} & \quad \text{rdfs:Class}. \\
\text{ns:Dog} & \quad \text{rdfs:subClassOf} & \quad \text{ns:Animal}. \\
\text{ns:Felix} & \quad \text{rdf:type} & \quad \text{ns:Dog}.
\end{align*}
\]

one can entail that

\[
\begin{align*}
\text{ns:Felix} & \quad \text{rdf:type} & \quad \text{ns:Animal}.
\end{align*}
\]

is true.

This is possible due to the RDF Schema Entailment rule “rdfs9” [92], which states that

\[
\begin{align*}
\text{If} & \quad ?u \quad \text{rdfs:subClassOf} & \quad ?x. \\
\text{and} & \quad ?v \quad \text{rdf:type} & \quad ?u. \\
\text{this entails that} & \quad ?v \quad \text{rdf:type} & \quad ?x.
\end{align*}
\]

This rule matches to the RDF graph defined by statements 2.4–2.7 by assigning ?u ← ns:Dog, ?x ← ns:Animal, and ?v ← ns:Felix. Replacing the variables in (?v, rdf:type, ?x) with these assignments gives the inferred triple (ns:Felix, rdf:type, ns:Animal). A more comprehensive list of RDFS inference rules will be given later.

The rdfs:subClassOf relation becomes transitive by RDF Schema Entailment rule “rdfs11”:
Besides the definition of classes and resources that are members of these classes, RDF schema allows one to define relationships between resources or classes. So far we have shown the two relations \( \text{rdf:type} \) and \( \text{rdfs:subClassOf} \). It is possible to define a new relation type \( P \) by asserting that \( P \) is of type \( \text{rdf:Property} \):

\[
P \quad \text{rdf:type} \quad \text{rdf:Property}.
\]

(2.15)

Relations are, like classes and resources, identified by URIs. The relation \( P \) allows connecting two resources \( U_1 \) and \( U_2 \) with a statement like

\[
U_1 \quad P \quad U_2.
\]

(2.16)

RDF Schema provides the equivalent to \( \text{rdfs:subClassOf} \) for relations with the transitive \( \text{rdfs:subPropertyOf} \) relation. Similar rules exist to derive the transitive closure of sub-relations and entail all valid relations.

The \( \text{rdfs:domain} \) and \( \text{rdfs:range} \) relations allow defining the domain (subject) and range (object) of instances that occur in a specific relation. An example of this can be found in the RDF Schema axioms:

\[
\text{rdfs:subClassOf} \quad \text{rdfs:range} \quad \text{rdfs:Class}.
\]

(2.17)

\[
\text{rdfs:subClassOf} \quad \text{rdfs:domain} \quad \text{rdfs:Class}.
\]

(2.18)

These axioms assert that every instance occurring in the object of a \( \text{rdfs:subClassOf} \) relation is of type \( \text{rdfs:Class} \) (2.17) and that every instance occurring in the subject of a \( \text{rdfs:subClassOf} \) relation is also of type \( \text{rdfs:Class} \) (2.18). Therefore, these axioms make triples 2.4 and 2.5 redundant as they can be entailed from 2.6.

Table 2.1 shows a subset of the RDF Schema Entailment rules defined in [92]. The table contains a set of preconditions that trigger the generation of the triples given in the conclusion. This kind of logical argument, where a new proposition (conclusion)
### 2.5 OWL

In line with the previous sections, the goal of this section is to give a very brief overview of OWL. The complexity and scope of OWL is several orders of magnitude larger than that of RDF and RDF Schema. Therefore, we refer to the original specifications for details.

The Web Ontology Language (OWL) is defined in a family of specifications [51, 65, 93, 122, 137, 161] and provides a larger set of constructs to define ontologies than RDF Schema. Examples of such features are the definition of equivalence

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Precondition</th>
<th>Conclusion</th>
</tr>
</thead>
</table>

Table 2.1: Selected RDF Schema Entailment rules (see [92]).

Looking back at this introduction, we summarize that RDF Schema provides a class and property hierarchy as well as means to entail the class membership of instances based on domains and ranges of properties. OWL adds additional capabilities with regard to reasoning.
of classes, properties and instances; inverse, transitive, symmetric, functional and inverse functional relations; and many others.

With OWL Lite, OWL DL and OWL Full, three different versions of OWL were developed with different feature subsets and restrictions. OWL Lite comprises the smallest set of features. It was devised to enable a simpler development of tools that support just this subset of constructs [122]. OWL DL provides greater expressiveness while still remaining computable. OWL DL provides a much greater expressiveness than RDF Schema with regard to its axioms about classes, properties and individuals. Yet, it is no superset of RDF Schema because OWL DL does not include the metamodeling of RDF Schema (mixing classes and instances) [133]. OWL Full provides even greater expressiveness than OWL DL, e.g. by removing this distinction between classes and individuals. It loses at the same time however the property of being computationally tractable because its inferencing is undecidable.

Even OWL Lite is significantly more difficult to handle than RDF Schema because it assumes the Open World Assumption and contains negation. While query languages such as SQL and Prolog assume that statements, which cannot be proven, are false, the Open World Assumption does not allow drawing this conclusion. The reason for this is that ontologies shall not be considered as closed but can be extended in other ontologies. This can have several unexpected consequences such that one cannot prove that two instances or classes are different unless this is explicitly stated.

We do not go any deeper into OWL as the focus of this thesis is put on RDF Schema, which is less expressive but also highly relevant [171]. Jena2 [172, 148] demonstrates how a forward chaining rule system can be used to extend the reasoning system presented in chapter 3 to support also OWL Lite.

### 2.6 SPARQL

Once the modeling and representation of (semi-)structured data are agreed upon, the question arises how this data can be queried for information retrieval. Several languages and paradigms to query RDF data have been proposed in the past. Furche et al. surveyed RDF query languages in [70] and distinguish the following categories of query languages:

- query languages with navigational access
- pattern-based RDF query languages
- “relational” RDF query languages
- reactive RDF query languages
Query languages with navigational access like Versa [131] are strongly influenced by XPath [56] and emphasize the traversal of the RDF graph based on labeled edges. A Versa query operates on sets of nodes. The query can start for example with a set of all instances of a certain class. This set can then be replaced by all nodes that can be reached with edges that carry a certain predicate or be restricted to those nodes that have incident edges with a certain predicate. “Relational” RDF Query Languages such as RQL [106] and SPARQL [57] share their character with SQL with regard to having SELECT–FROM–WHERE statements with semantics known from relational query algebra. They will be discussed in more detail at the example of SPARQL. Pattern-based RDF query languages like TRIPLE [158] and Xcerpt [152] include pattern matching and rule evaluation into the query language. This provides for example independence from RDF Schema entailment in the triple store. Finally reactive RDF query languages like Algae [142] are based on event-condition-action statements. Such actions (or reactions to events and conditions) can include querying the RDF graph, inserting new statements, or adding new event-condition-action statements.

After this brief survey, the following sections concentrate on the SPARQL Protocol and RDF Query Language, abbreviated as SPARQL. SPARQL is a W3C Recommendation [57, 27] that provides means for querying RDF databases. Given its state as a W3C Recommendation, SPARQL plays a very important role in the Semantic Web community and is thus focus of this thesis. It comprises the definition of a query language as well as a message protocol that defines the communication between the party who sends the query and the party who answers it.

SPARQL is a RDF Query Language similar to the relational SQL and originated in a W3C standardization process from its predecessors RDQL [155] and SquishQL [124]. A central concept of SPARQL is the matching of graph patterns, so called basic graph patterns (BGPs). A basic graph pattern is a set of triples (in Turtle notation [26]) that become patterns by the introduction of variables.

Recall the definition of the model graph $T_M$ with triples $t$ consisting of the values subject, predicate, and object denoted by $t = (s, p, o) \in T_M \subseteq (U \cup B) \times U \times (U \cup B \cup L)$, where $U$, $B$ and $L$ denote the sets of URIs, blank nodes, and literals respectively. A basic graph pattern, or query graph $T_Q$, extends this and consists of a set of triples $t = (s, p, o) \in T_Q \subseteq (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$, where $V$ denotes a set of variable identifiers. A result of such a query is an assignment of values to variables, called valuation, such that replacing the variables in the query graph with the assigned values makes the query graph a subgraph of the model graph.

Basic graph patterns allow formulating SELECT–WHERE statements such as the following example, which queries all digital cameras that support MMC memory chips:

$$\text{SELECT} \quad \text{WHERE} \quad \text{FILTER} (\text{mmc} \neq \text{null})$$
SELECT ?camera
WHERE {
  ?camera rdf:type ns:DigiCam .
  ?memory rdf:type ns:MMC .
}

The SELECT clause has similar semantics as in SQL, representing a conjunction of conditions that need to be fulfilled for each result. It contains variables just like SQL.

SPARQL supports four kinds of queries. SELECT queries return a set of binding of variables in the form of result sets as known from SQL. CONSTRUCT queries create and return RDF graphs, similarly to XQuery [37], which returns a constructed XML tree. DESCRIBE queries return an RDF graph that describes the resources found. The structure of this graph is not defined in the SPARQL specification, however. Finally, ASK queries check whether a query pattern has matches in the model graph or not and return a boolean answer.

The SELECT statement allows for the solution modifiers DISTINCT, ORDER BY, LIMIT <integer>, and OFFSET <integer>, which carry the same meaning as in SQL. The Basic Graph Patterns in the WHERE clause can be combined with UNIONS and extended with OPTIONAL parts. FILTERs allow restricting the valuations (e.g. with regular expressions or inequalities).

A more in-depth introduction to the constructs of SPARQL and their meaning can be found in [57, 70, 135].

2.7 Applications

Many applications of Semantic Web technologies can be found in various domains. The following paragraphs sketch a sample of such application scenarios.

Bizer et al. show in [35] how Semantic Web technologies can be used in the job recruitment process and present taxonomies that can be used to describe capabilities of potential employees and job requirements. In this scenario the focus of searching shifts towards near matches, which is currently not supported by SPARQL.

The GLUE 2 [9] information schema serves as a model for entities in the Grid world such as computing and storage resources. Heine has presented in [96, 94] the idea to translate this model to RDF for semantic resource discovery. The use of Semantic
Web technologies for the Grid is further discussed in the scope of the Semantic Grid movement, see e.g. [64].

Extensive research has been done on integrating semantic information in Wikis [112, 153, 154]. The chief idea is to annotate links. An article about London would not just link to the United Kingdom but the link could carry the annotation “capital of”. This allows, among others, automatic data extraction (e.g. for a table of all capitals) and improves consistency because information can be queried from page contents instead of being stored redundantly in summary tables. The information stored in Wikipedia can be used as well to generate global identifiers for concepts (e.g. the URI http://en.wikipedia.org/wiki/London may be used to represent London) and build ontologies, see for example [97]. Semantic information can be embedded likewise in conventional HTML pages using RDFa [7, 8].

The Nepumuk project [80] aims at the integration of Semantic Web technologies into the desktop. This includes tagging files, tracking the origin of files, access management, and social collaboration. The project shows various innovations possible with Semantic Web technologies. One of its foundation layers is GridVine [60], a Peer-to-Peer based RDF datastore.

The news agency Reuters recently published OpenCalais [132], a service that reads unstructured texts and extracts named entities (people, companies, organizations, geographies, books, albums, authors, etc.), facts (positions, alliances, etc.) and events (sporting, management change, labor action, etc.) that allow interlinking documents and improving search and navigation. This is an example how semantic technologies are employed in industry today already. Other attempts have been made to provide publication data as RDF [34]. This allows finding co-authors, other papers of the same author, and much more. It also solves the problem of several authors sharing the same name.

Figure 2.5 presents five parties that contribute and use information in a B2C sales environment. Instead of presenting product specifications in pure plain text, features can be presented by manufacturers in a structured way. This allows customers to filter products and compare them in a much better way than manually comparing a set of unstructured textual descriptions. At the same time, it reduces the effort of vendors to transfer such information into their local sales system. The domain of digital cameras is just one example that demonstrates the problem of consumer product compatibility. Lenses need to fit on camera bodies, cases need to fit to bodies, lens hoods to lenses, memory chips to cameras. This kind of information can be published globally by manufacturers of products. Having this information represented in a formal language allows for easy recommendation systems by the vendors. These need to describe the products sold in their online shops and benefit from being able to reuse information from manufacturers. At the same time they
Figure 2.5: Information integration in a B2C Platform.

contribute information themselves as well. This information comprises the availability and prices of products, shipping and handling fees, and the contact points of the vendors. This information can then be used by customers, search engines (e.g. to locate geographically near vendors), and sales platforms that compare prices. While buyers benefit from the availability of information contributed by the previously described parties, they can contribute useful information as well, such as reviews and feedback about products (quality) and vendors (reliability, shipping speed, etc.). Detailed reviews of products are also contributed by magazines and journals. All this information can be used by B2C platforms (shopping portals) that guide buyers and arrange sales.

The B2C platform scenario demonstrates how information can be inherently distributed throughout the Internet. This illustrates the requirement to integrate information and refer to information provided by other parties. These are just some of the strengths of Semantic Web technologies.

2.8 Appraisal of Semantic Web Technologies

The preceding paragraphs sketched applications of Semantic Web technologies in various problem domains. Each problem domain reaps the benefits from semantic technologies in different ways but several aspects pervade many examples and shall be summarized in this section.
2.8 Appraisal of Semantic Web Technologies

Global identifiers The introduction of globally unique and agreed upon identifiers for concepts (not documents) is among the most fundamental ideas of RDF and a feature that is not seen anywhere else in the Internet domain. Huge amounts of data and information are locked in closed databases and cannot be reused by applications different from the ones for which the databases were designed in the beginning. Naming services such as DNS provide global identifiers but their scope is limited to servers on the Internet. URLs allow one to address documents on the Internet and link to these documents but serve a pure navigational purpose. URLs carry no semantic meaning. The URIs employed by RDF on the other hand allow for example defining and processing synonyms and homonyms.

Annotation of objects Content creation by users has become a hot topic in the stream of Web 2.0. Compared to the possibilities offered by Semantic Web technologies, the current annotation possibilities of Web 2.0 have to be considered rather in their infancy. In particular the lack of global identifiers is a huge barrier. With the existence of global identifiers, everybody can refer to these for annotating objects. It is not necessary that the annotation is performed and stored at the same place where the objects are located. Due to the global naming schema it will still be possible to identify the annotations to be associated with the objects.

Integration of data from various sources Collecting information about a topic such as all reviews of a specific camera is currently rather difficult. A search is often incomplete (reviews cannot be found because certain keywords were used instead of others, e.g. the word “test” instead of “review”) or pages appear that do not contain actual reviews but just link to pages with reviews. This leads to poor results with regard to precision and recall. Semantic Web technologies mitigate this issue.

Automatic processing Structured information can be processed much better than plain text for example. It can be queried to generate reports, updated automatically, and may be normalized such that contradictions (due to keeping information redundantly but not updating all copies) can be reduced.

Enhanced searching and interlinking In the World Wide Web, links are used mainly for two purposes. First, for navigation by human users, and second, as an indication of page popularity by ranking algorithms of search engines. Links provide, however, little information about the content they lead to. This becomes possible with semantic technologies (c.f. [7, 8]) and allows for automatic information extraction (e.g. generation of a report of all capitals in the world) as well as user support during navigation (see also [68]).
Knowledge inference RDF Schema and OWL provide tools for knowledge inference. The means we consider most important from a practical perspective are the ability to describe sub-classes and -properties as well as equivalence of classes and instances.

Exchange format Finally, RDF can be used as a simple exchange format (e.g., as in RSS) that allows for simple parsers and applications that extract just the information they understand.

These advantages are countered by critics as Shirky [157] and Zambonini [174], who argue that syllogism (conclusions from two premises) is not useful in the real world because “most of the data we use is not amenable to such effortless recombination” [157], that Semantic Web technologies are too difficult to learn, and that they do not solve any real problem [174].

With enthusiasts and critics taking up their stance, only the future will be able to tell who is right. The author considers the chances offered by Semantic Web technologies weightier than the problems adduced by critics.
3 Storing RDF triples in DHTs

Contents

3.1 Peer-to-Peer Networks ........................................... 33
  3.1.1 Classes of Peer-to-Peer Networks .......................... 34
  3.1.2 Pastry ......................................................... 36
3.2 Triple Storage ..................................................... 38
  3.2.1 Life-Cycle of Triples ......................................... 41
  3.2.2 Node Departure ................................................ 42
  3.2.3 Node Arrival .................................................. 42
  3.2.4 RDFS Rules ................................................... 43
  3.2.5 Local Storage .................................................. 44
  3.2.6 Triple Lookup .................................................. 44
  3.2.7 Bloom Filters .................................................. 45
3.3 Related Work ..................................................... 48
  3.3.1 Centralized Triple Stores ..................................... 48
  3.3.2 Distributed Triple Stores ..................................... 51

After the presentation of a Semantic Web foundation, chapter 3 deals with distributed hash tables (DHTs) and how they can be employed to store RDF triples. In section 3.1 we present an introduction to distributed hash tables and explain in section 3.2 how these are used in BabelPeers to store RDF triples. These strategies are then contrasted with other RDF triple stores in section 3.3. Parts of this chapter have been published in [18] and [20].

3.1 Peer-to-Peer Networks

Peer-to-peer overlay networks have been developed to harness the storage, computing, and network capacity of many computers. In contrast to client/server architectures, with a distinguished server serving many clients, peer-to-peer networks consider the peers as equals. Each peer can act as a server and a client simultaneously. This distributes the storage, computing, and network load between many peers and improves fault tolerance by removing the server as a single point of failure. Scalability and fault tolerance are crucial advantages of peer-to-peer networks over client/server architectures.
3 Storing RDF triples in DHTs

Yang and Garcia-Molina define peer-to-peer networks as follows [173]:

“In a peer-to-peer system (P2P), distributed computing nodes of equal roles or capabilities exchange information and services directly with each other.”

Gribble et al. [78] define peer-to-peer networks as:

“A peer-to-peer (P2P) distributed system is one in which participants rely on one another for service, rather than solely relying on dedicated and often centralized infrastructure. Instead of strictly decomposing the system into clients (which consume services) and servers (which provide them), peers in the system can elect to provide services as well as consume them. The membership of a P2P system is relatively unpredictable: service is provided by the peers that happen to be participating at any given time.”

Peer-to-peer networks add a new layer onto the network stack. Peers, also called nodes, are connected by purely virtual connections, which express that a peer A knows another peer B (B does not necessarily need to be aware of A). Such connections are often expressed by a node identifier and a socket (end-point of the IP networking protocol), where the remote peer listens for messages. Virtual connections do not need to represent physical proximity although this is desirable for performance reasons. Messages are solely sent along these virtual connections connecting two peers, with the IP protocol being used for routing the messages between them. A routing protocol of the peer-to-peer overlay network is in charge of routing messages to peers that are not directly connected to the sender.

3.1.1 Classes of Peer-to-Peer Networks

Peer-to-peer networks are commonly grouped into two different classes based on the way messages are routed and objects are located in the network. These groups are unstructured and structured overlay networks. Unstructured overlay networks rely on broadcasting or external index servers to locate information. Structured overlay networks, on the other hand, maintain an internal address space that is used for routing as will be discussed later.

The first peer-to-peer networks, such as Napster by Shawn Fanning, relied on a central index server to locate data while peers served only as providers for the data and had no message routing task. Because of the existence of such a distinguished index server, this generation of networks is considered semi-centralized and usually not regarded as true peer-to-peer networks. As a central server has to be considered
3.1 Peer-to-Peer Networks

a single point of failure of the network, this motivated the development of the first
generation of unstructured peer-to-peer networks. Prominent representatives of this
class of networks are Gnutella, originally developed by Justin Frankel and Tom
Pepper, and the super-peer network FastTrack by Niklas Zennström, Janus Friis
and Jaan Tallinn. In these networks, nodes perform data searches by broadcasting
a search request to their neighbors who will then recursively forward the request
until a certain hop-count has been reached. Each peer that receives a request and
stores data relevant to the request notifies the searching node. After that the data
requester and bearer communicate by a “direct” IP connection.

The second generation of peer-to-peer networks represents the structured peer-to-peer
networks that have seen a lot of attention from academia. Famous representatives
are CAN by Ratnasamy et al. [147] and several networks that are based on Plaxton’s
algorithm [140] such as Chord by Stoica et al. [165], Pastry by Rowstron and
Druschel [151], and Tapestry by Zhao et al. [175]. Recent developments include
P-Grid by Aberer et al. [4], SkipNet by Harvey et al. [91], Kooerd by Kaashoek and
Karger [102] and Kademia by Maymounkov and Mazières [121]. A survey about
various peer-to-peer networks can be found in Lua et al. [116].

Structured peer-to-peer overlay networks make use of an ID space for message
routing. Each peer in the network is assigned a unique position in the ID space and
knows a (small) set of other peers in the network as well as their position in the ID
space. In order to send a message to a certain position in the ID space, a distance
function allows comparing the distance between positions of known peers and the
target and allows forwarding the message to the peer that is closest to the target.
Overlay networks differ from each other by the structure of the ID space, selection
and number of known “neighbor peers”, and the routing strategy. Nevertheless, they
share sufficient commonality to justify abstraction layers such as the Rice Common
API [62] that allows interchanging concrete structured overlay network protocols.

Distributed hash tables (DHTs) are a specialization of structured peer-to-peer
networks and can be used to store (key, value) pairs by hashing the key into the
ID space of the overlay network and storing the value at that position just as in
the case of common hash tables. The central idea is to use consistent hashing (see
Karger et al. [104]), where the ID space is chosen to be very large compared to the
number of peers in the network and partitioned among them in such a way that
the arrival or departure of a node has only limited and only local influence on the
assignment of fractions of the ID space to peers.
3 Storing RDF triples in DHTs

3.1.2 Pastry

The Pastry network [151] is a well studied structured overlay network that provides DHT functionality using prefix based routing. As furthermore a stable and free implementation (FreePastry) is available, it has been chosen as the foundation for BabelPeers and shall be illustrated very briefly.

Pastry uses a one-dimensional, cyclic ID space of numbers from \( \{0, \ldots, 2^{128} - 1\} \). Each peer and each value stored in the DHT has a specific location in this ID space. The IDs of peers can be generated using a (pseudo) random number generator or by hashing the peer’s IP address and join time using a cryptographic hash function. The location, where values shall be stored, is determined by hashing the key of the \((key, value)\) pair with a cryptographic hash function. IDs are interpreted as numbers to the base \(2^b\) with \(b \in \mathbb{N}\). Here \(b\) is often chosen to be 4 so that the IDs resemble hexadecimal numbers.

Figure 3.1 shows a Pastry network with 11 nodes in an ID space of base \(2^2\) ranging from 000 to 333. The top most point of the circle represents the address 000, the right most point represents the address 100, the bottom most point represents the address 200, and the left most point represents the address 300. All addresses in the

![Pastry Network Diagram](image_url)
3.1 Peer-to-Peer Networks

The circle illustrates how Pastry assigns fractions of the ID space to peers. Each peer is responsible for those IDs that are numerically closer to it than to any other peer. This means that peers in densely populated areas of the ID space are responsible for a smaller fraction of the ID space than peers in a sparsely populated area. If node IDs are generated uniformly random, it is guaranteed, however, that the size of the largest fraction is with high probability only by a factor of $O(\log N)$ larger than the expected value of $2^m N$, where $2^m$ represents the size of the ID space and $N$ represents the number of participating nodes in the network. A proof of this can be found for the similarly structured Chord network in [118].

Pastry uses prefix routing to deliver a message to the node that is responsible for the target ID. Each node has a routing table that points to nodes which share prefixes of different lengths with the node maintaining the routing table. Such a routing table for node F of figure 3.2a is illustrated in figure 3.2b. F is located at position 113. The first row of the routing table stores pointers to nodes that share

\[\begin{array}{cccc}
\text{Prefix length} & \text{Next Digit} \\
0 & B & * & I \ K \\
1 & D & * & ? \ G \\
2 & ? & E & ? \ * \\
\end{array}\]

(a) 3.2: Routing table for node F.
no common prefix with the current node F. These are pointers to arbitrary nodes in the quadrants shown in figure 3.2a. If peer F needs to deliver a message to any address starting with prefix 2, it would see that its ID (113) has no common prefix with this target address and forward the message to the node pointed to in the routing table for prefix length 0 and next digit 2 which is node I. This node has an ID whose common prefix length with the target ID is at least one. The node can then use its own routing table to forward the message to a node with common prefix length two or greater. The routing table is used to bridge long distances very quickly and get closer to the target at each step. Eventually the routing uses a node’s leaf-set, i.e. a list of nodes that are closest to this node on the ID ring. This allows routing the message to the node that is numerically closest to the target position. On average only $O(\log N)$ routing steps are necessary to route a message to any target. The routing table shows several cells with values * and ?. The * entries represent the cases where the node who owns the routing table would be the matching entry for this cell. The ? represents empty entries in the table.

For a description of the exact routing protocol, join protocol, and repair protocol of the routing tables, we refer to the original publication by Rowstron and Druschel [151]. An interesting result from experiments presented in this paper is that on networks of 1,000 to 1,000,000 nodes, strategies to improve locality allowed the authors to achieve performances, where routing over the Pastry layer was only 30% to 40% slower than sending the message directly (assuming each node knew each other node).

### 3.2 Triple Storage

After the introduction of distributed hash tables, this section discusses the question how triples can be stored in such networks. We begin this with recapitulating some important notations given in chapter 2 and generalizing the concepts of Pastry to generic DHTs.

In RDF, information is modeled as directed labeled graphs or, equivalent, as triples. A triple $t$ of the model graph $T_M$ consists of the values subject, predicate, and object denoted by $t = (s, p, o) \in T_M \subseteq (U \cup B) \times U \times (U \cup B \cup L)$, where $U$, $B$ and $L$ denote the sets of URIs, blank nodes, and literals respectively. Blank nodes of the model graph can be treated as scoped unique identifiers and are therefore handled like URIs. After normalization, literals are treated as a third kind of identifiers. Therefore, we consider triples to be of the form $(s, p, o) \in T_M \subseteq X \times X \times X$, where $X$ stands for the set of internal representations of URIs, blank nodes, and literals.
3.2 Triple Storage

A distributed hash table is comparable to a regular hash table with efficient insert and lookup operations, except that the ID space of a DHT is distributed between the nodes of a peer-to-peer network. This network is able to route insert and lookup requests to those nodes which are responsible for the respective fractions of the ID space in $O(\log N)$ routing steps with high probability, where $N$ is the number of nodes in the network.

For storing RDF triples in the DHT, we use a cryptographic hash function $h : \mathcal{X} \rightarrow D$ to map the components $s, p, o$ of a triple to three positions in the ID space $D$ of the distributed hash table and store copies of the triples at the nodes responsible for the respective hash values $h(s), h(p), \text{and } h(o)$. An optional extension is to map combinations of subject, predicate, and object to IDs ($h : \mathcal{X} \times \mathcal{X} \rightarrow D$) to achieve better load balancing in storage and query load. The function $n : D \rightarrow N$ is used to denote the node which is responsible for a certain ID in the ID space. This function abstracts the routing functionality of the peer-to-peer layer.

Each node hosts multiple RDF databases that serve different purposes. The first of these RDF databases to be mentioned is the local triples database. It stores those RDF triples that originate from the particular node and is therefore comparable to the local databases of Edutella [128].

All local triples are regularly disseminated to the nodes in the network by calculating the hash value of the subjects, predicates, and objects and sending the triples to the nodes responsible for the corresponding parts of the DHT space. These nodes store the triples in their received triples set. Triple owners may occasionally be the responsible target nodes for some of their triples, so perhaps some of the triples in the local triples set are duplicated in the received triples set. However, the larger the network grows, the less likely is this to happen for a particular triple. Thus, we do not take special care of duplicates and neglect the memory consumption caused by a triple stored twice on a single node.

Unless a triple has two or even three identical components, it is disseminated to three distinct positions in the DHT. The larger the network grows, the higher is the probability that these positions are actually located on different nodes. Thus, for large networks our distribution procedure results in a tripling of the number of triples.

A third kind of triple store is responsible for a replica database. Pastry provides means to store replica of data on the $k$ nodes whose IDs are nearest to the target hash value determined by the hash function. The purpose of these replica is to support the network when nodes depart or crash. In Pastry it is likely that a lookup message is routed through a replica node before it arrives at the node responsible for the target position determined by the hash function. As replica nodes are often
3 Storing RDF triples in DHTs

geographically closer to the query issuer than the actual target of the query, they can intercept the message and answer the query. This exploits locality information and reduces latencies.

The node with the ID closest to the hash value of a triple becomes root node of the replica set. This node is responsible for sending all triples in its received database to the replica nodes. In Pastry these nodes are simply defined by the leaf-set. The replica, ordered by their distance to the target hash-value, are said to have rank 1 through k.

Finally, each node hosts a database for generated triples that originate from forward chaining. The forward chaining rules are applied to the received triples databases and all generated triples are stored into the generated triples database. They are then disseminated like local triples to the respective nodes and replicated from there on.

Figure 3.3 illustrates the process of inserting an RDF triple \((s, p, o)\) by a node A. In this process it does not matter whether A possesses the triple because some third party that is not member of the DHT asked A to store it (step 1) or whether A is the immediate source of this triple. The life-time of the triple is determined by A’s membership in the DHT. Node A calculates the hash value of the triple’s subject and routes an insert request to the node responsible for this location in the ID space in the DHT (step 2). This is node B. Node B replicates the triple to nodes in its vicinity (step 3), often called the leaf-set. In case node B crashes or leaves the network, these neighbor nodes assume the ID space previously covered by B and ensure that the triple remains in the network. Steps 2 and 3 are then repeated for the triple’s predicate and object.

Figure 3.4 illustrates the triple databases hosted by peers and the flow of triples. The following sections elaborate further on this topic.
3.2 Triple Storage

3.2.1 Life-Cycle of Triples

The life-cycle of triples is influenced by various events. As each node contributes its local triples to the network, nodes joining and departing from the network change the knowledge base. Furthermore, nodes can of course acquire new information that needs to be distributed within the network, and delete or update triples.

Owing to the very dynamic nature of peer-to-peer networks and the autonomy of peers, it is not possible to guarantee that nodes depart gracefully and unregister all triples contributed by them to the network. In order to delete these obsolete triples from the received triples databases eventually, it is either necessary for nodes to poll the triple sources whether their triples are still up to date, or to add an expiration date to triples. We have decided to follow the latter approach as polling does not allow a node to learn about new triples that are hashed into its ID space when a new node joins the network and contributes these new triples. Each triple has an expiration date and the owner is in charge of continuously sending update messages. This approach is known as “soft-state” in literature. The information source has to decide whether it considers itself as stable, such that triples have rather long life-times and few refreshes, or whether it considers itself as volatile. If more than one source inserts the same triple, the life-time of this triple in the received triples database is defined as the maximum life-time of all copies in the network.

The life-time of triples has a high influence on the traffic in the network. Too short life-times require frequent re-disseminations of triples and create high load. Too long life-times prevent their timely deletion when the owner parts the network. As
3 Storing RDF triples in DHTs

the life-time can be defined individually for each triple depending on the specific domain and circumstances, no recommendation is given at this place.

The life-time of triples that were inferred in the reasoning process is defined to be the minimum remaining life-time of the triples that fulfilled the precondition of the RDFS rule (see section 3.2.4). An update of these triples in the precondition triggers an update of the inferred triples as well.

3.2.2 Node Departure

A node departing or crashing does not only cause the expiration of triples but makes it also necessary that other nodes cover the area and data of the DHT that was previously occupied by the departing node. The DHT layer repairs the routing tables of the peers and thereby ensures that the peer-to-peer network remains connected. But besides correct message routing it is also important that one or more nodes take over the responsibilities of the departing node. As the node storing the first replica of a triple is located in the immediate vicinity of the departing node in the DHT space (closest to the respective hash value of the triple), this node receives all following queries for the respective triple. As it stores the relevant data due to being rank 1 replica, we do not have to take special provision for failing nodes. Owing to the DHT schema used, the data from replication is automatically available to queries.

When a replica node becomes root node for a triple, the number of replica has decreased. The new root node is notified by the DHT layer that its leaf-set (set of nodes in the vicinity of the node regarding DHT space) has changed and sends triples to a new replica node. It is important to start this update procedure immediately as more nodes can fail until the next update of triples is issued.

3.2.3 Node Arrival

A new node disseminates all its local triples. But while the departure of nodes is handled almost automatically, the arrival of a new node is more complicated. At the moment a node joins the DHT, it receives queries for the covered ID space but lacks the data to answer these queries correctly. As the owner of a triple (the node having the triple in the local triples database) cannot notice the arrival of the new node and because it sends update messages at a low frequency, it is task of the replica nodes of the new node to provide it with all necessary data. These observe the new node as a new member in their leaf-set and instantly send all triples to the new node for which it is to be considered a root node.
3.2 Triple Storage

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Precondition</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdfs2</td>
<td>(?a \text{rdfs:domain} ?x . ?u \text{rdfs:type} ?x .)</td>
<td>(?u \text{rdfs:type} ?x .)</td>
</tr>
<tr>
<td>rdfs3</td>
<td>(?a \text{rdfs:range} ?x . ?v \text{rdfs:type} ?x .)</td>
<td>(?v \text{rdfs:type} ?x .)</td>
</tr>
<tr>
<td>rdfs5</td>
<td>(?u \text{rdfs:subPropertyOf} ?u . ?v \text{rdfs:subPropertyOf} ?x .)</td>
<td>(?u \text{rdfs:subPropertyOf} ?x .)</td>
</tr>
<tr>
<td>rdfs7</td>
<td>(?a \text{rdfs:subPropertyOf} ?b . ?u ?b ?y .)</td>
<td>(?u ?y .)</td>
</tr>
<tr>
<td>rdfs9</td>
<td>(?u \text{rdfs:subClassOf} ?x . ?v \text{rdfs:type} ?y .)</td>
<td>(?v \text{rdfs:type} ?y .)</td>
</tr>
<tr>
<td>rdfs11</td>
<td>(?u \text{rdfs:subClassOf} ?x . ?v \text{rdfs:subClassOf} ?x .)</td>
<td>(?u \text{rdfs:subClassOf} ?x .)</td>
</tr>
</tbody>
</table>

Table 3.1: Selected RDF Schema Entailment rules (see [92]).

This strategy reduces the period of time enormously during which a new node sends empty results. If the amount of data that needs to be transferred from replica nodes to the new root node is large, however, the transfer creates a rather long period during which the new node sends incomplete results. Because of that, the new node forwards queries to one of the replica picked at random during the transition period. The transition period ends when the new node does not receive any new triples from its replica for a certain time.

3.2.4 RDFS Rules

The RDF semantics document [92] describes how RDFS entailment can be seen as a set of rules which generate new RDF triples from existing ones. Several rules map a single triple to the creation of a new triple. This is an easy process that can be executed on the received triples database. An example for such a rule is rdfs4a where a triple (\(?u, \?a, \?x\)) creates a new triple (\(?u, \text{rdfs:type, rdfs:Resource}\)). These rules are easy to evaluate with the local knowledge of peers.

Table 3.1 contains a list of those RDF Schema rules that contain two triples in the precondition. The underscored variables show that all 6 rules can be evaluated on nodes with only local knowledge. As triples are hashed by subject, predicate, and object, all triples contributing to the precondition of a rule can be found on the root node. At the example of rdfs7, we see that all triples with common subject ?a
3 Storing RDF triples in DHTs

<table>
<thead>
<tr>
<th>Filter</th>
<th>Tree</th>
<th>Filter</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*, *, *)</td>
<td>SPO</td>
<td>(*, P, O)</td>
<td>POS</td>
</tr>
<tr>
<td>(*, *, O)</td>
<td>OSP</td>
<td>(S, *, O)</td>
<td>OSP</td>
</tr>
<tr>
<td>(*, P, *)</td>
<td>POS</td>
<td>(S, P, *)</td>
<td>SPO</td>
</tr>
<tr>
<td>(S, *, *)</td>
<td>SPO</td>
<td>(S, P, O)</td>
<td>SPO</td>
</tr>
</tbody>
</table>

Table 3.2: Mapping of selection patterns to in-memory trees.

and predicate ?a are mapped to the same ID in the DHT space and are, therefore, stored at the same node.

3.2.5 Local Storage

The received triples database stores triples in memory for fast response times. It comprises three AVL trees that store the triples in lexicographic order, sorted by SPO (subject, predicate, object), POS, and OSP order. This allows for example selecting all triples containing a specified subject with the SPO tree or selecting all triples containing a specified predicate and object with the POS tree. In fact, each subset of triples specified by values on positions from the set \{S, P, O\} can be selected in time $O(\log n + m)$, where $n$ represents the number of triples stored in the respective tree and $m$ represents the number of matches, by simple traversal of the tree. Table 3.2 illustrates how all 8 possible filter sets can be mapped to one of the three AVL trees for efficient evaluation.

All strings representing URIs and literals are internalized meaning that no two objects exist in memory that represent equal strings. This enables that resources are shared between triples and reduces the memory consumption tremendously.

3.2.6 Triple Lookup

So far section 3.2.5 discussed local lookups on the received triple set that do not require any network communication. From the way RDF triples are stored in the network, it is obvious that lookups for all triples with a defined subject, predicate, or object are possible by routing a lookup message to the nodes that are responsible for the respective hash values. However, specifying only one fixed value and returning all triples containing this value at the specified position creates huge and often unnecessary traffic. Therefore, we allow restricting the possible values at positions that are not used for the DHT lookup. These restrictions are either filters for fixed values (i.e. a URI, literal, or blank node) or filters for sets of possible values (sets of
3.2 Triple Storage

URIs, literals, or blank nodes. Set filters are expressed as Bloom filters (see [36]) that will be discussed in the next section.

A lookup is modeled as

\[
\text{lookup} : X \times P_s \times P_p \times P_o \rightarrow (X \times X \times X)^* \quad (3.1)
\]

The \(X\) on the left hand side indicates the value (URI, literal, blank node) that is used for the hash function to determine the target node. It is followed by three predicates (in the sense of logic) that restrict the subject, predicate (in the sense of RDF) and object of matching triples. These predicates can test for equality to URIs, blank nodes, literals, containment in Bloom filters, or accept anything (wildcards denoted with \(*\)). The result of a lookup is a set of matching triples. In a lookup for triples matching the pattern \((U_1, *, *)\) we would call \(U_1\) the lookup key and the subject the (lookup) position, i.e. the position where the lookup key is located in the triple. In this example, \(P_s(x)\) evaluates to true iff \(x = U_1\); \(P_p(x)\) and \(P_o(x)\) evaluate to true for all values of \(x\).

We define an analogous count-operation that returns the number of matches instead of the matches themselves.

If triples are indexed not only by subject, predicate, and object, but also by combinations of these, the lookup can be generalized to

\[
\text{lookup} : (X \cup \{\ast\})^3 \times P_s \times P_p \times P_o \rightarrow (X \times X \times X)^* \quad (3.2)
\]

where \((X \cup \{\ast\})^3\) defines the lookup key used for routing the lookup message. This key would be used for storing and retrieving triples.

3.2.7 Bloom Filters

Bloom filters [36] are a means to represent the membership of elements of a large set in a compact form. They are used in BabelPeers to encode a set of valuation candidates (for the predicates discussed above) that needs to be transferred over the network. Transferring all URIs would create a huge amount of traffic and can be circumvented.

A Bloom filter consists of a bit vector \(b = (b_1, b_2, \ldots, b_m)\) and a family of hash functions \(h_1(x), \ldots, h_k(x)\). In order to assert the membership of an element \(e\) in the Bloom filter, the bits \(h_1(e), \ldots, h_k(e)\) are set to 1 in the bit vector. For checking whether an element \(e\) is member of the Bloom filter, it is tested whether \(h_1(e) = \ldots = h_k(e) = 1\).
3 Storing RDF triples in DHTs

Figure 3.5: Bloom filter of \( m = 11 \) bits with \( k = 3 \) hash functions \( h_i(e) = 2 + c_i \cdot e \mod m \), with \( c_1 = 3 \), \( c_2 = 5 \), and \( c_3 = 7 \).

Figure 3.5 illustrates this principle for a bit vector of \( m = 11 \) bits with a family of \( k = 3 \) hash functions, which are defined as \( h_i(e) = 2 + c_i \cdot e \mod m \), with \( c_1 = 3 \), \( c_2 = 5 \), and \( c_3 = 7 \). The upper half of figure 3.5 shows how the elements 3, 4, and 5 are inserted into the Bloom filter. We see that \( h_1(3) \) and \( h_2(4) \) collide at bit 0. After inserting element 3, this bit is already set to 1 and remains 1 when element 4 is inserted. The lower half of figure 3.5 illustrates how the same hash functions are used to check whether the elements 1, 2, 3 are members of the Bloom filter. This check rejects element 1, as \( h_2(1) = 7 \) and bit 7 is 0. Therefore, it returns the correct answer “negative”. The check for element 2 returns positive even though this element was never asserted to be member of the Bloom filter. The reason for this is that all relevant bits are set to 1 by other inserted elements. This phenomenon is called a false positive and shall be kept rare by finding a good combination of Bloom filter size and number of hash functions as discussed below. Finally the element 3 is correctly determined as a member of the Bloom filter.

Bloom filters can return true positives and negatives (correct answers that elements are members of the Bloom filters or not) but also false positives (answers that elements are members of the Bloom filters even though they are not). False negatives are impossible. The rate of false positives is determined by three characteristic constants of the Bloom filter:

- \( n \), number of elements inserted into the Bloom filter,
- \( m \), number of bits in the bit array, and
- \( k \), number of hash functions.

After inserting all \( n \) elements into the Bloom filter, the probability that a specific
3.2 Triple Storage

bit is still 0 is

\[ p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}. \]  

(3.3)

The reason for this is that one hash function for a specific element does not set the bit to 1 with probability \(1 - \frac{1}{m}\) and this process is repeated for \(k\) hash functions and \(n\) elements. The approximation follows from \(e = \lim_{x \to 0} (1 + x)^\frac{1}{x}\) and replacing \(m = -\frac{1}{x}\):

\[ p = \left(1 - \frac{1}{m}\right)^{kn} = (1 + x)^{kn} = (1 + x)^{\frac{1}{x} \cdot kn} = \left((1 + x)^{\frac{1}{x}}\right)^{kn} = \left((1 + x)^{\frac{1}{x}}\right)^{-\frac{kn}{m}} \]  

(3.4)

For large \(m\), \(x = -\frac{1}{m}\) converges to 0 and the approximation from 3.3 holds (see also [125]).

Given this, the probability of a false positive, i.e. the indication that a random element is contained in the Bloom filter even though it is not, becomes

\[ f = (1 - p)^k = \left(1 - e^{-\frac{kn}{m}}\right)^k = \exp\left(k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right)\right). \]  

(3.5)

We are now looking for an optimal \(k\) that minimizes \(f\) by finding a solution for \(\frac{df}{dk} = 0\). As the exp function is strictly monotonous, we can also search for

\[ \frac{d}{dk} \left(k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right)\right) = 0. \]  

(3.6)

With

\[ \frac{d}{dk} \left(k \cdot \ln\left(1 - e^{-\frac{kn}{m}}\right)\right) = \ln\left(1 - e^{-\frac{kn}{m}}\right) + \frac{kn}{m} \cdot \frac{e^{-\frac{kn}{m}}}{1 - e^{-\frac{kn}{m}}} \]  

(3.7)

it is easy to verify that \(k = (\ln 2) \frac{m}{n}\) gives \(\frac{df}{dk} = 0\) and it can be shown that \(k = (\ln 2) \frac{m}{n}\).
3 Storing RDF triples in DHTs

is a global minimum [125]. By using this valuation of $k$, the error rate $f$ becomes

$$f = \exp \left( k \cdot \ln \left( 1 - e^{-k \frac{m}{n}} \right) \right)$$

$$= \exp \left( \ln 2 \frac{m}{n} \cdot \ln \left( 1 - e^{-\ln 2 \frac{m}{n}} \right) \right)$$

$$= \exp \left( \ln 2 \frac{m}{n} \cdot \ln \frac{1}{2} \right)$$

$$= \exp \left( -(\ln 2)^2 \frac{m}{n} \right)$$

$$= \exp \left( -(\ln 2)^2 \frac{m}{n} \right)$$

$$\approx 0.6185 \frac{m}{n}.$$  

We summarize that we can achieve the lowest false positive rate for Bloom filters with $k = (\ln 2) \frac{m}{n}$ hash functions, which makes the false positive rate $f \approx 0.6185 \frac{m}{n}$. Conversely, for a candidate set of $n$ elements, we can pick a bit vector size $m$ such that, given an optimal $k$ for this combination of $n$ and $m$, the false positive rate meets a certain desired limit. These results will be reused later.

3.3 Related Work

RDF triple stores have been subject of active research during the past years. The following sections shall give an overview over strategies developed to store RDF data.

3.3.1 Centralized Triple Stores

Abadi et al. review in [2] how relational databases can be used for RDF databases, as exercised by Jena 2 [172, 50], Oracle [54], Sesame [41], and 3store [86]. As RDF does not prescribe a physical storage organization, different relational representations have been developed in the past. Abadi et al. list four different ways of storing RDF data in relational databases. Most representations assume that concrete URIs and literals are stored in a separate dictionary and linked to by short identifiers (e.g. 4-byte integers). This reduces space requirements because long URIs do not need to be stored in the database several times any more. At the same time this normalization step makes query processing more efficient as the identifiers require less memory and can be compared much faster for equality.
3.3 Related Work

The simplest representation of RDF in a relational database is a *three-column schema*, where all triples are stored in a single table with the columns subject, predicate, and object. This data representation is considered rather inefficient by Abadi et al. [2] because most queries require several self-joins over this table, which creates a lot of I/O traffic as it is impossible to cache the entire table in memory.

*Property tables* (predicate tables) are an alternative to the three-column schema. The idea of this technique is to denormalize the table of a three-column schema to save the costs of many joins. If a subject occurs frequently with a common set of attributes (e.g. “title”, “author”, and “isbn” are common attributes of books), one can create a table with the subject as a key and the other attributes as further columns. No joins are necessary on such a table if all attributes of a query are covered by a single table.

This technique has several issues:

- RDF is a semi-structured data representation; not all subjects share a common set of attributes. If this data representation was used for example for an address book, the telefax column would have *many NULL entries*. These consume space and need to be read into memory when records are loaded. Depending on the selection of attributes that are stored denormalized, one can have plenty of NULL values, which is inefficient, or have dense data but many property tables that need to be joined. It is difficult to find a balance here.
- A second issue is *multi-valued attributes*. A book might have any number of authors, which is difficult to represent in a property table.
- Finally, if queries span several property tables, complex joins or unions between subqueries need to be calculated.

Two different kinds of property tables have been suggested. In *clustered property tables*, a clustering algorithm looks for properties that often occur together and builds property tables from these. Several property tables for different clusters may be created but each property may occur in only one property table. In *property-class tables*, one property table is defined for each RDF Schema class. This assumes that all elements of one class have similar descriptions. It works particularly well for reified statements where the class is `rdf:Statement` and each reified triple has three properties (subject, predicate, and object).

Finally, Abadi et al. propose *vertically partitioned databases* with one two-column table for each unique property (predicate). The first column contains the subjects and the second column contains the objects of all triples that share one predicate. Although many joins are required to process queries over multiple properties, fast
3 Storing RDF triples in DHTs

(linear) merge joins can often be used, if each table is sorted by subject. Column-oriented databases are optimized for this kind of data representation and give very promising results.

The advantages of the vertically partitioned databases are the native support for multi-valued attributes and heterogeneous records (see NULL entries above), only properties accessed by a query are read, no clustering algorithms are needed, and joins are often very efficient, because the tables are small and linear merge joins can be used.

Two or more sorted relations can be efficiently joined by their subject (subject-subject joins) with a merge join, which is linear in the number of elements in the relations. Joining two triples on the object of the first triple and the subject of the second triple (subject-object joins) is less efficient, because the first relation is not sorted by object. Abadi et al. propose to add additional indices or to materialize frequent path expressions. The query (?book, ex:author, ?author), (?author, ex:wasBorn, ?year) is an example for requiring a subject-object join. This join can be materialized in a two-column table “author:wasBorn” to prevent recalculating it every time the values are needed.

Abadi et al. found that property tables and vertical partitioning performed a factor of 2-3 faster than the three-column schema approach. Using C-Store [1] a column based database in combination with property tables, gained another speedup of factor 10 over a set of 7 queries compared to regular property tables and vertical partitioning [2].

The following sections give a very brief introduction to popular centralized RDF stores and match them with the data representations presented above.

**Jena2**

Jena2 is presented in [172] and [50]. Jena2 uses relational databases in the back-end based on a denormalized three-column schema in combination with special property tables. It supports OWL reasoning by a rule system and plugins for external reasoners such as Racer [84] and FaCT [98]. Query processing is mostly pushed down to the relational database layer.

**Sesame**

Sesame [41] is a very modular framework for storing and querying RDF data. The persistence layer is encapsulated in a so called Storage And Inference Layer (SAIL).
3.3 Related Work

This allows implementing Sesame on top of various RDF repositories. The SAIL provides a very lean interface to add RDF statements, delete RDF statements, and lookup RDF statements that match a pattern of the form \((s, p, o)\) where each element is either a constant or a “don’t care” (wildcard) symbol. Furthermore, the SAIL supports a call-back interface that is notified if statements are added or deleted to trigger the inference process. The authors experimented with 3-column schemata and vertical partitioning.

3Store

3Store by Harris and Shadbolt [86, 87] is a centralized triple store that focuses on large quantities of data [156]. It maps literals and URIs to 64 bit numbers (by using parts of an MD5 hash) and stores triples in one huge relational database table with four columns for a hash that identifies the model graph a triple originates from, a hash for the subject, a hash for the predicate, and a hash for the object. Collisions in the hash values are detected and reported but not handled specifically. This is justified by the calculation that the probability of a collision in \(10^9\) different URIs is only \(10^{-10}\). SPARQL queries are translated into relational SQL queries and submitted to the underlying RDBMS. Harris and Shadbolt do not mention any reasoning capabilities.

Redland

Redland by Beckett [28] is one of the very first RDF stores. Its focus is on providing a library for other applications to build on. One part of this library is responsible for storing RDF triples. Redland uses three hash tables that map subject-predicate, predicate-object and subject-object combinations to URIs and literals that occur in the missing third position of triples in the model graph. Triple patterns with just one known value require a full table scan. Beckett does not mention any reasoning capabilities.

3.3.2 Distributed Triple Stores

Besides centralized triple stores, a large body of research deals with distributed triple stores.
3 Storing RDF triples in DHTs

**Triple stores using unstructured Peer-to-Peer networks**

Edutella by Nejdl et al. [128, 127, 129, 43] was designed to provide an RDF-based metadata infrastructure for P2P applications, building on the JXTA framework, in order to enable interoperability between heterogeneous JXTA applications [128]. JXTA is a set of XML based protocols to cover typical functionality of unstructured super-peer networks. The principal idea of Edutella is that peers register the queries they may be asked (e.g. by specifying supported metadata schemas or by specifying which predicates occur in their database) at super-peers so that queries can be routed to peers who may provide data of interest [128]. Special services have been developed to enable data replication, workload balancing, and schema mapping. Another example of this class of distributed triple stores, where peers advertise the content they provide and queries are routed to relevant peers, is SQPeer by Kokkinidis et al. [110].

Advantages of this kind of triple stores are that it is relatively easy to perform rights management, as each peer can decide individually whether it wants to answer a query or not, and that many queries can be answered sufficiently with locally stored knowledge without joining information that is spread across several peers. Also data updates are fairly inexpensive. A disadvantage is, however, that queries may need to be flooded to many nodes. Ranger and Cloutier criticize in [145] that Edutella relies on a high workload on super-peers and that SQPeer makes the invalid assumption that peers knew all about the subjects they publish.

RDFGrowth by Tummarello [169] allows users to browse on a local database of triples. The information stored in this database grows by iteratively communicating and exchanging knowledge of interest with other peers in a peer-to-peer group.

**DHT-based RDF stores**

Cai et al. [45, 47] were among the first to propose RDF triple stores based on DHTs (in their case a MAAN [46] based on Chord) for their RDFPeers implementation. The focus of their work is on storing triples in the network and performing lookups of single triple patterns. The indexing schema is identical to BabelPeers based on hashing and inserting each triple by subject, predicate and object.

RDFCube by Matono et al. [120] extends RDFPeers with a three dimensional bitmap cube where nodes store slices of this cube. The intention of this is to speed up triple joins by calculating the bitwise AND of slices and transferring only relevant triples. This is a simpler alternative to the Bloom filters employed in BabelPeers.
3.3 Related Work

The Atlas project by Koubarakis et al. [111, 113, 114, 103] uses the Bamboo overlay network [149], which is similar to other DHTs but optimized for high churn, and indexes triple patterns similarly to BabelPeers, based on subject, predicate, object, and combinations thereof. The query processing [113, 114] of Atlas will be discussed later. Atlas provides reasoning based on forward chaining [103] that is implemented as in BabelPeers [18] and also backward chaining.

The GridVine project by Aberer et al. [5, 60] is based on the P-Grid [4] network. It employs the same idea of hashing subject, predicate, and object of a triple. GridVine supports “Semantic Gossiping” as a form to mediate between different schemata.

YARS2 by Harth et al. [89, 88] falls somewhere in-between centralized RDF stores and DHT-based RDF stores. It implements a B-tree that stores RDF quadruples in 6 permutations and can be executed on a constant number of nodes of a cluster. Several operations require flooding the network.

In conclusion we see that many DHT-based triple stores use very similar indexing strategies for RDF triples as BabelPeers. A greater degree of differentiation can be found in the way queries are processed, which is topic of the next chapters.
3 Storing RDF triples in DHTs
4 Query Evaluation

Contents

4.1 Foundation ........................................... 56
4.2 Outline of Query Processing ..................... 56
4.3 Query Evaluation Agents ............................. 58
4.4 Selection of Lookups ................................ 59
  4.4.1 Local Heuristics ................................ 63
  4.4.2 Network Heuristics .............................. 66
  4.4.3 Caches ........................................ 71
  4.4.4 Network Heuristics (cont.) ..................... 74
  4.4.5 Wrappers ...................................... 76
4.5 Migration ............................................. 81
4.6 Processing Triple Patterns ......................... 82
4.7 Evaluation ........................................... 87
  4.7.1 Local Heuristics ................................ 89
  4.7.2 Network Heuristics .............................. 93
  4.7.3 Estimation Accuracy ............................ 99
  4.7.4 Wrappers and Variations ....................... 101
4.8 Related work ......................................... 105
4.9 Conclusion ........................................... 107
4.10 Outlook ............................................. 108

The query processing strategies for RDF queries provide huge degrees of freedom. Queries can be processed sequentially or in parallel, clocked or streamed, based on a relational model or based on sub-graph isomorphism, and various strategies can be applied to determine the order in which triple patterns are processed.

This chapter presents strategies developed for BabelPeers and contrasts them against strategies by other researchers. The chapter is structured as follows. Section 4.1 presents the foundation for query processing and defines the type of queries addressed. Based on this, section 4.2 gives a high-level picture of the algorithm and steps used to process queries that is detailed in sections 4.3 through 4.6. Section 4.7 evaluates the strategies, section 4.8 presents related work, and section 4.9 concludes the query evaluation.

Parts of this chapter have been published in [15], [20] and [24].
4 Query Evaluation

4.1 Foundation

Subject of this chapter is the evaluation of basic graph patterns, a substantial subset of SPARQL. For this we assume the notations introduced in section 3.2.

Following [113], we define a conjunctive query \( Q \) as a formula

\[
\forall x_1, \ldots, x_n : (s_1, p_1, o_1) \land (s_2, p_2, o_2) \land \cdots \land (s_m, p_m, o_m)
\]

where \( x_1, \ldots, x_n \) are variables and each \((s_i, p_i, o_i)\) is a triple pattern, consisting of URIs, blank nodes, literals, and variables. Each variable \( x_k \) of the query head \( H(Q) \) appears in at least one triple pattern. The triple patterns \((s_i, p_i, o_i)\) of the query make up the set \( T(Q) \subseteq (U \cup V) \times (U \cup V) \times (U \cup L \cup V) \), also called basic graph pattern or query graph, where \( V \) denotes the set of variables.

The goal of the query processing is to find all assignments of variables to URIs, blank nodes, and literals such that all triples of the query can be found in the model graph, or formally:

Find all valuations (assignments of values to variables) \( v: V \mapsto X \) such that \( T(Q)[x_1/v(x_1), \ldots, x_n/v(x_n)] \subseteq T_M \).

The \([a/b]\) notation represents substitution (\( a \) is substituted by \( b \)).

4.2 Outline of Query Processing

If the triples in the query graph \( T(Q) \) are denoted as \( t_i = (s_i, p_i, o_i) \), the query can be expressed as

\[
\pi_{H(Q)}(\sigma_{t_1}(T_M) \Join \sigma_{t_2}(T_M) \Join \cdots \Join \sigma_{t_m}(T_M))
\]

using relational algebra. Formula 4.2 shows the selection of all triples matching the triple patterns, calculating the natural join, and projecting the resulting relation to the relevant variables. Note that each selection usually involves a costly network transfer. As the natural join is associative and commutative, it is task of the query processor to find a sensible order in which triple patterns are evaluated and joined.

Figure 4.1 illustrates an example query that searches for all publications and their authors of a specific department in a university. The nested shapes present the order in which triples matching the patterns of the query were retrieved and joined.
4.2 Outline of Query Processing

Figure 4.1: Query processing orders.

to the previous intermediate results. We see in the figure how two different orders
where chosen by different query processing strategies.

The graphically represented query of figure 4.1 can be stated as follows using the
formalisms stated above:

\[
?\text{pubName}, ?\text{authName} : - (\?\text{pub}, \text{type}, \text{Publication}) \land \\
(\?\text{pub}, \text{name}, ?\text{pubName}) \land \\
(\?\text{pub}, \text{pubAuth}, ?\text{auth}) \land \\
(\?\text{auth}, \text{name}, ?\text{authName}) \land \\
(\?\text{auth}, \text{worksFor}, \text{Department13}).
\]

Owing to a possibly large number of triple patterns we consider it reasonable to take
the decision which triple pattern to process next iteratively, each time after a triple
was processed, instead of planning the entire query in advance. This motivates the
following high-level algorithm:

```
while not all triples processed do
    select next triple pattern, lookup key and position;
    (optionally) migrate query evaluation;
    fetch triples;
    join;
```

The first step in the algorithm selects the triple pattern that shall be processed next.
As three copies of each triple are stored in the DHT, it determines furthermore
whether subject, predicate, or object shall be used for the lookup. The goal of this
first step is to find an order in which triple patterns are processed that generates an
overall low effort. Heuristics to implement this are shown in the following sections.
4 Query Evaluation

Note that we do not limit the algorithm to left-deep join trees. Imagine a query with the triple patterns (\(?v_1, U_1, ?v_2\)), (\(?v_2, U_2, ?v_3\)), (\(?v_3, U_3, ?v_4\)). If the first and third pattern have much fewer matches than the second, we allow fetching and processing these first, which creates two relations (\(?v_1, ?v_2\)) and (\(?v_3, ?v_4\)) and valuation candidate sets (possible valuations; each additional triple pattern can only reduce these valuations) for the four variables. Only then we fetch the second triple pattern and exploit the known candidate sets for Bloom filters to reduce the network traffic. Details are shown later.

As the triples of the model graph are distributed throughout the network, we want to exploit locality of information. This comprises the location of triples in the network as well as the location where a query is being processed. Depending on the situation it may be advantageous to fetch information from remote nodes or to migrate the query evaluation, including the current intermediate result, to the location where remote triples are located. This motivates modeling the query processing as an agent with a goal, state, and knowledge that can migrate in the network.

In a third step, triples need to be transferred over the network (unless the algorithm decided to transfer the evaluation to the location of the triples). In this state it is important to use strategies that prevent the transfer of many irrelevant triples. Finally, the triples matching a lookup are joined with previous intermediate results.

The following paragraphs present and discuss strategies to perform the steps shown above in an efficient manner, where in the context of DHT-based RDF triple stores the term “efficient” means that few bytes and few messages need to be sent through the network. It is important to realize that the total query processing costs arise from determining a query order and performing the evaluation according to this. In a DHT network, each message has a noticeable latency. Therefore, a query order strategy that gives good results but relies on too many or too big messages may eat up its benefits of ordering the triple patterns. Owing to the distributed nature, we cannot assume global knowledge about the data nor extensive preprocessing.

4.3 Query Evaluation Agents

As mentioned above, the process of evaluating a query is encapsulated in a movable agent. An agent in this case is composed of a position, state, task and knowledge. The position refers to the node that executes the agent’s instructions. The state is encoded in a simple automaton which resembles the algorithm shown above. The states of the automaton comprise “find next triple to process”, “consider migration”, “migrate”, and “fetch triples and join”. A more complex automaton is used in case intermediate results are cached, which is described later (see chapter 5 for details).
4.4 Selection of Lookups

Selecting the next triple pattern and lookup position (whether subject, predicate, or object will be used for calculating the hash key that determines where a lookup is send to) is performed in several phases as depicted in figure 4.2. The overall goal is to score triple patterns and lookup positions and select the combination with the highest score for being processed next.

The algorithm begins with generating all possible combinations of not yet processed triple patterns and lookup positions and then sorts these by the following order (assuming triples are indexed in the DHT by subject, predicate, and object individually):

- If the lookup position of a triple pattern contains a fixed value (URI or literal), this lookup is preferred over a lookup by a variable. As many lookups can provide a fixed value as lookup key, the following tiebreaker is used.

- Lookups by subject are preferred over lookups by predicate and object; and lookups by object are preferred over lookups by predicates. This preference of positions is a heuristic based on expected selectivity and was also proposed by Liarou et al. [113].

- The lexicographical order of triple patterns is used as a final tiebreaker.

As the following methods score and order only triple patterns as such and do not incorporate the lookup position, this presorting is very helpful to prefer sensible lookup positions.

Consider an example query with two triple patterns (?person, type, Student) and (?person, name, ?name). This gives the following possible lookups:
4 Query Evaluation

<table>
<thead>
<tr>
<th>Triple pattern</th>
<th>Lookup position</th>
</tr>
</thead>
<tbody>
<tr>
<td>(?person, type, Student)</td>
<td>Subject</td>
</tr>
<tr>
<td>(?person, type, Student)</td>
<td>Predicate</td>
</tr>
<tr>
<td>(?person, name, ?name)</td>
<td>Subject</td>
</tr>
<tr>
<td>(?person, name, ?name)</td>
<td>Predicate</td>
</tr>
<tr>
<td>(?person, name, ?name)</td>
<td>Object</td>
</tr>
</tbody>
</table>

The first sorting criterion creates the following order (each block represents one equivalence class), by distinguishing between constant values and variables:

| (?person, type, Student)    | Predicate       |
| (?person, name, ?name)      | Subject         |
| (?person, type, Student)    | Subject         |
| (?person, name, ?name)      | Object          |

The second sorting criterion considers the lookup positions and creates:

| (?person, type, Student)    | Object         |
| (?person, type, Student)    | Predicate      |
| (?person, name, ?name)      | Predicate      |
| (?person, name, ?name)      | Subject        |
| (?person, type, Student)    | Subject        |
| (?person, name, ?name)      | Object         |

Finally, the last sorting criterion uses the alphabetic order as a final tie-breaker:

| (?person, type, Student)    | Object         |
| (?person, name, ?name)      | Predicate      |
| (?person, type, Student)    | Predicate      |
| (?person, name, ?name)      | Subject        |

If triples are indexed not only by subject, or predicate, or object, but also by combinations of these, we suggest a slightly different proceeding. If a triple pattern contains two or three fixed values (URIs or literals) we do not allow lookups by each one individually but only by the combination of both. This gives the following possibilities and ranking of lookups:

1. \((S, P, O)\), lookup by only fixed values
4.4 Selection of Lookups

2. \((S,*,O)\), lookup by only fixed values
3. \((S,P,*)\), lookup by only fixed values
4. \((*,P,O)\), lookup by only fixed values
5. \((S,*,*)\), lookup by fixed value
6. \((*,*,O)\), lookup by fixed value
7. \((*,P,*)\), lookup by fixed value
8. \((S,*,*)\), lookup by variable
9. \((*,*,O)\), lookup by variable
10. \((*,P,*)\), lookup by variable

After sorting the possible lookups by this order, we score them in the following and use a stable sorting according to the calculated score to find the best triple pattern. The score of lookups (triple patterns + positions) is called the retrieval performance. Using stable sorting guarantees that triples with identical retrieval performance keep their relative order.

The order defined in the pre-sorting phase serves several purposes:

- If two combinations of triple pattern and lookup position get the same score, the stable sorting guarantees that the combination is preferred which ranked higher in the pre-sorting. This favors constant values and discriminative positions.

- Determining the score of a triple pattern with a variable in the lookup key may be expensive. Scores are determined in the order defined in the first phase. This allows caching mechanisms to prevent such expensive score calculations, which will become clearer later.

- Finally, the pre-sorting prevents that an overly large fraction of triple patterns is processed by their predicates if possible. This is detrimental for load balancing because the number of different predicates is rather small.

The idea of scoring lookups by what is called a retrieval performance (denoted with \(\mathcal{RP}\)) has been introduced already. Values of the retrieval performance do not need to be natural or real numbers but can be taken from any totally ordered set (i.e. a set of values with a antisymmetric, transitive, and total comparison predicate \(\leq\)). This total order is the most fundamental property of the retrieval performance. The set of natural numbers with the natural \(\leq\) or \(\geq\) comparison may be used to represent retrieval performances. We will see later that different strategies use more advanced retrieval performance definitions that estimate the number of triples matching a triple pattern.
Each retrieval performance definition contains three distinguished elements:

- **bestPossible** represents the best possible value of a retrieval performance definition.
- **worstDefined** represents the worst possible value of a retrieval performance definition that is still defined.
- **undefined** represents the value of a retrieval performance definition indicating that a triple pattern lookup is undefined and impossible.

We assume the following order \( \text{bestPossible} \leq x \prec \text{worstDefined} \prec \text{undefined} \) for any \( x \in \mathcal{RP} \) with \( x \neq \text{worstDefined}, \text{undefined} \). The motivation of these distinguished elements will become clear later.

We will now present a framework of building blocks for determining the retrieval performance. This framework assumes a set of **base modules** that map triple patterns and positions to retrieval performances (scores) and a set of **wrappers** around these base modules that intercept scoring requests to prevent their execution or to let them pass but modify the returned values. The purpose of these wrappers is mainly caching and pruning to save network traffic. Figure 4.3 illustrates this idea. The **\( \mathcal{RP} \) Calculator** is such a base module. **\( \mathcal{RP} \) Pruning** and **\( \mathcal{RP} \) Cache** are wrappers. They provide the same interface as the **\( \mathcal{RP} \) Calculator** and can be mixed and matched arbitrarily.

All base modules and wrappers share a common signature to calculate a retrieval performance

\[
\text{calcRP} : \mathcal{X}^{3}_V \times \text{Pow} \left( \{S, P, O\} \right) \times \text{Pow}(\mathcal{X})^3 \rightarrow \mathcal{RP}
\]  

(4.3)

The first component, \( \mathcal{X}^{3}_V \) (\( \mathcal{X}_V = \mathcal{X} \cup \mathcal{V} \)), represents the triple pattern for which the retrieval performance shall be calculated. The second component, \( \text{Pow}(\{S, P, O\}) \),
4.4 Selection of Lookups

determines to which peer the lookup request shall be sent. The third component of
the retrieval performance calculation, $\text{Pow}(\mathcal{X})^3$, describes possible valuations for the
three positions of the triple. For URIs and literals, these valuation candidate sets
contain exactly the one specific value. For variables, the valuation sets can contain
sets of known valuation candidates, or they can be defined as $\Delta$ representing all
possible values. The latter is used in case no valuation candidates are known. We
define the cardinality of $\Delta$ with $|\Delta| = \infty$. Finally the range of the $\text{calcRP}$ function
represents a retrieval performance as defined above.

Note that the retrieval performance is not defined for a specific triple pattern but
for a combination of triple pattern, lookup position, and valuation candidates.

**Example:** Consider the triple pattern $(U_1, ?v_1, ?v_2)$ with $U_1$ representing a URI,
$?v_1$ representing a variable with a candidate set $\text{cand}(?v_1)$ and $?v_2$ representing a variable
without a known candidate set. In this case, we would calculate
$\text{calcRP}(U_1, ?v_1, ?v_2, \{S\}, \{U_1\}, \text{cand}(?v_1), \Delta)$.

This definition of a retrieval performance and calculation leads to the following
interfaces:

```java
interface RP
    lessOrEqual(RP that) : boolean;
```

**Listing 4.1:** Definition of the retrieval performance interface.

```java
interface RPCalculator
    calcRP(ArrayList t, Pow({S, P, O}) pos, Pow(\mathcal{X})^3 cand) : RP;
```

**Listing 4.2:** Definition of the retrieval performance calculator interface.

### 4.4.1 Local Heuristics

We begin the discussion of retrieval performance calculators with *local heuristics*—
that is heuristics which do not require any network communication to rank triple
patterns. While these heuristics do not generate any network traffic for determining
the query plan, they have very little knowledge about the costs that are incurred by
the individual query processing steps. Therefore, they are cheap but inaccurate.
These heuristics use a very simple retrieval performance definition called \emph{IntegerRP}.

\begin{Verbatim}
class IntegerRP implements RP
   int value;
   lessOrEqual(RP that) : boolean begin
      return this.value ≥ that.value;
   end
\end{Verbatim}

\textbf{Listing 4.3}: Definition of the integer retrieval performance.

Note that the greatest and therefore best retrieval performance is the one with the \textit{smallest} value. In other words, the value should be interpreted as some kind of cost or penalty.

An \emph{IntegerRP} with value \texttt{MAX\_VALUE} represents the distinguished value \texttt{undefined}. An \emph{IntegerRP} with value \texttt{MAX\_VALUE} − 1 represents the distinguished value \texttt{worstDefined}.

\section*{Query order}

As query planning has not been addressed by other DHT-based RDF triple stores like [5, 45, 113, 120], we begin with a base module called \textit{query order}, which ranks triple patterns according to the order in which they appear in the query. This can be used as a base line for benchmarking the following procedures. Listing 4.4 gives pseudocode for this strategy.

\begin{Verbatim}
class QueryOrderRPCalculator implements RPCalculator
   List<XV[3]> triplesToProcess;
      return IntegerRP(Position of t in triplesToProcess);
   end
\end{Verbatim}

\textbf{Listing 4.4}: Retrieval performance calculator using query order of triples.

\section*{Variable counting}

Jena’s [172] \textit{variable counting} heuristic assigns costs of $\frac{1}{2}$, $\frac{1}{8}$, and $\frac{1}{4}$ for the occurrence of variables in the subject, predicate, and object of the triple pattern. The cost of a triple pattern equals the sum of the variable costs (plus $\frac{1}{8}$). This strategy penalizes
variables in particular in subject and object where fixed values are assumed to have a low selectivity (i.e., they are discriminative and can filter many triples). Listing 4.5 illustrates this using the cost measures given above multiplied by 8 in order to use integer numbers.

```
class VariableCountingRPCalculator implements RPCalculator
    calcRP(\text{\(X\{S, P, O\}\)} t, Pow(\{S, P, O\}) pos, Pow(\text{\(X\)}\{3\} cand) : \text{\(RP\)} begin
        value = 1;
        if \(t[S]\) is variable then value = value + 4;
        if \(t[P]\) is variable then value = value + 1;
        if \(t[O]\) is variable then value = value + 2;
        return IntegerRP(value);
    end

Listing 4.5: Retrieval performance calculator using Jena’s variable counting.
```

**Message counting**

The message counting strategy originates from Heine’s SG1 method in BabelPeers [94]. It is based on the idea to simply count the number of messages necessary for a lookup. This number is 1 for a lookup by URI or literal; it corresponds to the number of valuation candidates for variables with known candidate sets and is \(\infty\) for variables without known valuation candidates. Pseudo-code is given in listing 4.6. Note that our definition of non-existing candidate sets implies a cardinality of \(\infty\) whose \(\text{\(IntegerRP\)}\) is undefined.

```
class MessageCountingRPCalculator implements RPCalculator
    calcRP(\text{\(X\{3\}\)} t, Pow(\{S, P, O\}) pos, Pow(\text{\(X\)}\{3\} cand) : \text{\(RP\)} begin
        if |pos| \neq 1 then return undefined;
        return IntegerRP(|cand[pos]|);
    end

Listing 4.6: Retrieval performance calculator using message counting.
```

**Smallest candidate set**

The smallest candidate set retrieval performance calculator counts the number of candidates for each position and returns the minimum of these values, as it is considered most restricting. This is shown in listing 4.7.
4 Query Evaluation

```java
class SmallestCandSetRPCalculator implements RPCalculator {
    end
}
```

Listing 4.7: Retrieval performance calculator preferring the smallest candidate set.

**Smallest candidate set ignoring fixed values**

The smallest candidate set ignoring fixed values strategy extends “smallest candidate set” by ignoring any URIs and literals in the triple pattern because these have a candidate set of size 1. If each triple pattern has at least one fixed value, they are all scored equally by “smallest candidate set”.

```java
class SmallestCandSetIgnoringFixedRPCalculator implements RPCalculator {
        value = ∞;
        if t[S] is variable then value = min(value, |cand[S]|);
        if t[P] is variable then value = min(value, |cand[P]|);
        if t[O] is variable then value = min(value, |cand[O]|);
        if value = ∞ then return worstDefined;
        return IntegerRP(value);
    end
}
```

Listing 4.8: Retrieval performance calculator preferring the smallest candidate set while ignoring fixed values (URIs and literals).

4.4.2 Network Heuristics

We can distinguish between heuristics which do not require any network communication to rank the triple patterns and heuristics that query the network for the ranking. Heuristics of the first category comprise “query order”, Jena’s “variable counting”, Heine’s “message counting” and the two “smallest candidate set” heuristics. Our findings are that these heuristics have very little information at hand for taking their decisions and produce significantly higher traffic during query processing. For the sake of comparison, we have presented these strategies first and now focus on strategies that query information—the number of triples that would be returned by a lookup—from peers in the network. This introduces a delay due to the network routing and calculating an answer, but provides better estimates of
the costs of processing triples and generates overall considerably lower traffic for query processing.

We introduce a new type of retrieval performance for network heuristics called \textit{MatchCountRP}. As the name suggests it represents the number of triples matching a certain pattern. Less matches suggest a higher score.

```java
class MatchCountRP implements RP
    int matchCount;

    lessOrEqual(RP that) : boolean begin
        return this.matchCount ≥ that.matchCount;
    end

    getMatchCount() : integer begin
        return matchCount;
    end

Listing 4.9: Definition of the match count retrieval performance.
```

\textbf{Graph statistics}

Jena’s \textit{graph statistics} heuristic counts the number of triples that contain the subject, predicate, or object (one at a time) of a triple pattern, divides these numbers by the number of triples in the database, and returns the product of these three numbers as described in the following equation:

\[
GraphStatisticsRP'(s, p, o) = \frac{|(s, *, *)|}{|(*, *, *)|} \cdot \frac{|(*, p, *)|}{|(*, *, *)|} \cdot \frac{|(*, *, o)|}{|(*, *, *)|} = 4.4
\]

If the subject contains a variable, the first factor becomes \(\frac{|(*, *, *)|}{|(*, *, *)|} = 1\). Variables in the predicate and object position are handled analogously.

The equation gives a selectivity estimation between 0 and 1. In order to return the estimated number of matching triples, we multiply the result of equation 4.4 with \(|(*, *, *)|\) and get

\[
GraphStatisticsRP(s, p, o) = \frac{|(s, *, *)|}{|(*, *, *)|} \cdot \frac{|(*, p, *)|}{|(*, *, *)|} \cdot \frac{|(*, *, o)|}{|(*, *, *)|} \cdot |(*, *, *)| = 4.5
\]

This strategy cannot be implemented straightforward in a DHT because the total number of triples stored cannot be determined easily. The number can be extrapolated, however, by counting the number of triples stored on a node and
4 Query Evaluation

dividing this number by the fraction of the ID space occupied by this node. This needs to be normalized by the replication factor of triples in the network. As hash-values of URIs and literals are not evenly distributed (the number of occurrences of different values is not uniform), this estimation can be improved significantly by hashing triples not only by subject, predicate, and object separately but also by the concatenation of all three components and using only these hash values for estimating the number of triples in the network. Using a cryptographic hash function with the concatenation of all three components of the triple guarantees that the load-imbalance is bound by $O(\log N)$ [146, 168, 176], where $N$ is the number of nodes in the network. This load imbalance propagates of course onto the estimation of the number of triples in the network.

Listing 4.10 gives pseudo-code for this heuristic. The variable remoteTS refers to a remote triple set, an abstraction of the DHT that sends requests, waits for the answers, and returns these synchronously.

```java
class GraphStatisticsRPCalculator implements RPCalculator {
    calcRP($\mathcal{X}$[3] t, Pow({S, P, O}) pos, Pow($\mathcal{X}$)[3] cand) : RP begin
        product = 1;
        if t[S] is not variable then
            product = product \cdot remoteTS.matchCount(t[S],*,*)
            remoteTS.estimateNrTriples();
        if t[P] is not variable then
            product = product \cdot remoteTS.matchCount(*,t[P],*)
            remoteTS.estimateNrTriples();
        if t[O] is not variable then
            product = product \cdot remoteTS.matchCount(*,*,t[O])
            remoteTS.estimateNrTriples();
        product = product \cdot remoteTS.estimateNrTriples();
        return MatchCountRP(max(product, 1));
    end
}
```

Listing 4.10: Retrieval performance calculator using graph statistics.

Note that the cost of periodically hashing all triples by a concatenation of subject, predicate, and object has not been considered in the following analyses as it is independent of the number of queries evaluated.

Simple match count

The strategy simple match count (listing 4.11) simulates a triple pattern lookup request with the exception that not the actual matches are returned but the number
of matches. If a triple pattern contains variables, these are replaced with wildcards that match anything (like unbound variables). This reduces the precision compared to the Bloom filter based approach presented next but creates smaller messages.

```java
class SimpleMatchCountRPCalculator implements RPCalculator
{
    begin
        bloomfilters = {bloom(Δ), bloom(Δ), bloom(Δ)};
        mc = remoteTS.matchCount(t, pos, bloomfilters);
        return MatchCountRP(mc);
    end
}
```

Listing 4.11: Retrieval performance calculator using simple match count.

**Match count with Bloom filters**

A more precise version of this strategy is **match count with Bloom filters**, where variables are not just replaced by wildcards. All variables that occurred in triple patterns that were processed before have valuation candidate sets that are (not strict) supersets of the final valuations. We encode the possible valuations in Bloom filters and submit these along the lookup. This allows for more precise results. See listing 4.12.

```java
class MatchCountBloomFiltersRPCalculator implements RPCalculator
{
    begin
        bloomfilters = {bloom(cand[S]), bloom(cand[P]), bloom(cand[O])};
        mc = remoteTS.matchCount(t, pos, bloomfilters);
        return MatchCountRP(mc);
    end
}
```


Bloom filters can be constructed in any size. Too small Bloom filters produce high false positive rates; too big Bloom filters are expensive to transfer. Section 3.2.7 showed that the false positive rate of a Bloom filter can be estimated as $f = 0.6185 \frac{m}{n}$ where $\frac{m}{n}$ represents the number of bits per element reserved in the Bloom filter ($n$ is the number of valuation candidates, $m$ is the number of bits in the Bloom filter). As a basis, we use a relatively small ratio of $\frac{m}{n} = 3$ so that the number of hash functions is $k = (\ln 2) \frac{m}{n} \approx 2.08 \approx 2$. This gives an estimated false positive rate of $f \approx 23.7\%$. If the number of triples that are tested against this Bloom filter is
small compared to \( n \), the absolute number of false positives remains reasonable. If the number of valuation candidates is large, however, the number of false positives increases.

This heuristic can be improved as follows: At the beginning of the query evaluation, no valuation candidates are known for any variables. Therefore, when looking for the first triple pattern to process, each occurrence of a variable in a triple pattern is replaced with a wildcard and the number of matches to these triple patterns is determined and memorized. This number is an upper bound for the future number of valuation candidates, but more importantly, it tells how many triple patterns will be tested against the Bloom filters. Consider the following query:

```
SELECT ?phone
WHERE {
  ?person ns:name "Dominic Battre" .
}
```

The query processor will learn that \((*, \text{ns:name}, \text{Dominic Battre})\) has only one match while \((*, \text{ns:telephone}, *)\) has thousands of matches. After fetching all matches to \((*, \text{ns:name}, \text{Dominic Battre})\) the valuation candidates for \(?person\) are known. We assume that this is just one valuation candidate. That can be encoded in a Bloom filter. But even if this Bloom filter has a false positive rate of only \( f = 0.1\% \), testing 100,000 non-matching telephone number triples against this Bloom filter leads to 100 false positive matches, which may be off by a factor of 50. A higher precision has been reached by the following strategy.

We choose the number of bits in the Bloom filter as \( m = 3 \cdot \max(n, 1/10 \cdot n_t) \), where \( n_t \) represents the number of matches to the triple pattern while ignoring valuation candidate sets, and use \( k = 2 \) hash functions. If the number of local valuation candidates is of the same order as the number of remote matching triples (ignoring valuation candidate sets), the Bloom filter works as described above. If the number of remote matches exceeds the number of valuation candidates by a factor of 10, it becomes the dominant factor determining the Bloom filter size. The Bloom filter grows proportionally and becomes sparsely populated. This reduces the number of false positives. As a sparsely populated Bloom filter can be compressed well, the message size stays reasonable. The constants 3, \( 1/10 \) and 2 have been selected empirically. We will later see an improved strategy that works without these magic numbers.

A different strategy is to target a fixed number of false positives assuming that all remote triples are mismatches. For targeting a false positive rate that leads to only
one false positive in total, we would solve \( f \cdot n_t = 1 \), i.e. \( f = \frac{1}{n_t} = 0.6185 \frac{m}{n} \), and get \( m = n \cdot \log(\frac{1}{n_t})/\log(0.6185) \). In practice this created too large Bloom filters or too little precision, depending on the total target number of false positives. Therefore, the previous approach has been chosen.

### Match count with Bloom filters and mismatch count

The “match count with Bloom filters” is extended in *match count with Bloom filters and mismatch count*. The lookup messages and the construction of Bloom filters are exactly as before. The only difference is that the answer message to the request to count the number of matches contains an additional field, representing the number of triples that were filtered due to the Bloom filters. This number of mismatches drives the number of false positives during the final triple fetching. Section 4.6 discusses how this additional information can be used when retrieving remote triples.

In order to store the number of triples that were filtered by the Bloom filters, we define a slightly extended retrieval performance measure.

```java
class MatchAndMismatchCountRP implements RP {
    int matchCount;
    int mismatchCount;

    lessOrEqual(RP that) : boolean begin
        return this.matchCount >= that.matchCount;
    end

    getMatchCount() : integer begin
        return matchCount;
    end

    getMismatchCount() : integer begin
        return mismatchCount;
    end
}
```

**Listing 4.13**: Definition of the match count with mismatches retrieval performance.

### 4.4.3 Caches

Several types of *caches* can be used to prevent repetitive lookups. The match count for a triple pattern \( U_1, U_2, ?v_1 \) (two URIs and one variable) is for example the same regardless whether it is determined by asking the node responsible for \( h(U_1) \) or
4 Query Evaluation

\( h(U_2) \). As the retrieval performance of all possible lookups is required each time the next lookup needs to be determined, there is great potential for reusing previous results. A cache wraps the retrieval performance calculator. Lookups that occur for the first time are passed to the underlying base module and the result is written to the cache. Succeeding identical lookups are intercepted and the cached value is returned. Several criteria can be used to decide whether two lookups are identical. These criteria are usually reflected in differently chosen hash functions as shown in listing 4.14. The code shows an abstract base class that serves as a basis for several cache implementations.

All caches presented have a life-time of a single query and are not stored beyond that or used for other queries. Therefore, we do not discuss replacement and update strategies.

```java
class AbstractRPCache implements RPCalculator
    RPCalculator parent;

    calcRP(\( \mathcal{X}[3] \), Pow(\( \{S, P, O\} \)) pos, Pow(\( \mathcal{X}[3] \)) cand) : RP begin
        key = generateKey(t, pos, cand);
        if cache contains key then
            return cache[key];
        value = parent.calcRP(t, pos, cand);
        cache[key] = value;
        return value;
    end

```

Cache by pattern

The cache by pattern (listing 4.15) calculates a hash value from subject, predicate, and object of a triple pattern while substituting variables with a unified identifier. Therefore, this approach considers \( U_1, U_2, U_3 \) different from \( U_3, U_2, U_1 \) but \( U_1, U_2, ?v_1 \) and \( U_1, U_2, ?v_2 \) identical. As this approach is oblivious to valuation candidate sets, it is used in connection with the “simple match count” strategy.

Using this cache with “match count with Bloom filters”, would render the “match count with Bloom filters” base module equal to “simple match count”: At the very beginning no valuation candidates are known, so the Bloom filters would accept anything as matches. Lookups that follow when valuation candidates are known would be intercepted and answered by the cache. Therefore, no Bloom filter except \( \text{bloom}(\Delta) \) would be sent over the network.
4.4 Selection of Lookups

class CacheRPByPattern extends AbstractRPCache
generateKey(\mathcal{X}^3 t, \text{Pow}(\{S, P, O\}) \ pos, \ \text{Pow}(\mathcal{X}^3 \ candid) : \text{String} \ \begin{align*}
\text{key} &= \text{"}\}; \\
\text{key} &= \text{key} \circ \text{"} \circ (t[S] \text{ is variable } ? \square : t[S]); \\
\text{key} &= \text{key} \circ \text{"} \circ (t[P] \text{ is variable } ? \square : t[P]); \\
\text{key} &= \text{key} \circ \text{"} \circ (t[O] \text{ is variable } ? \square : t[O]); \\
\text{return} \ \text{key};
\end{align*} 
\ \end{align*}
end

Listing 4.15: Cache by pattern.

Cache by pattern and candidate set size

The cache by pattern and candidate set size (listing 4.16) works similarly as “cache by pattern”, except that it does not replace variables with a unified identifier but with the current number of valuation candidates for this variable (or $\infty$ if no candidates are known). This pays respect to the fact that the number of candidates decreases for variables when additional triple patterns are joined against intermediate results and makes the cache suitable for “match count with Bloom filters”. As soon as the valuation candidate sets of variables change, requests for “match count with Bloom filters” containing affected variables pass the cache and answers reflect the change.

class CacheRPByPatternAndCandidateSetSize extends AbstractRPCache
generateKey(\mathcal{X}^3 t, \text{Pow}(\{S, P, O\}) \ pos, \ \text{Pow}(\mathcal{X}^3 \ candid) : \text{String} \ \begin{align*}
\text{key} &= \text{"}\}; \\
\text{key} &= \text{key} \circ \text{"} \circ (t[S] \text{ is variable } ? |cand[S]| : t[S]); \\
\text{key} &= \text{key} \circ \text{"} \circ (t[P] \text{ is variable } ? |cand[P]| : t[P]); \\
\text{key} &= \text{key} \circ \text{"} \circ (t[O] \text{ is variable } ? |cand[O]| : t[O]); \\
\text{return} \ \text{key};
\end{align*} 
\ \end{align*}
end

Listing 4.16: Cache by pattern and candidate set size.

Local relative match count cache

The local relative match count cache (listing 4.17) is an extension of “match count with Bloom filters”. Instead of caching the number of matches for a triple pattern, it caches the number of matches divided by the size of the largest candidate set of variables occurring in a triple pattern. Imagine a valuation candidate set for
a variable \(?\text{person}\) is known. In a country where everybody owns a landline and a mobile phone, the number of matches for the pattern \((\text{?}\text{person}, \text{ns:telephone}, \text{?}\text{phone})\) will correspond to about twice the number of candidates for \(?\text{person}\). If further processing of triple patterns reduces the candidate set for \(?\text{person}\) to half of its previous size, we expect that the number of matches to the triple pattern above will be reduced to half of its previous size as well. Therefore, no second lookup is needed to determine the retrieval performance if the candidate set changes its size. This reduces the effort to determine the next triple pattern when the query graph contains many nodes with degree greater or equal three.

This concludes the list of caches used in BabelPeers.

4.4.4 Network Heuristics (cont.)

Relative match count

Finally, we present the strategy relative match count that combines ideas of several strategies presented above. We maintain the scenario of people with telephone
4.4 Selection of Lookups

numbers described before. Once we knew the factor 2, we were able to estimate the number of matches without transferring any further Bloom filters. The strategy “relative match count” targets exactly this at much lower costs than the “local relative match count cache” with “match count with Bloom filters”. In this case, a node that processes a query sends requests that are identical to the requests of the “simple match count” heuristic (variables are substituted by wildcards) but the answers are slightly extended: Besides the number of matches to the triple pattern with wildcards, they contain up to six ratios like the one described above.

A triple pattern can contain one or two variables with known candidate sets in six different combinations of positions (S, P, O, SP, SO, OP). Consequently the ratio of the number of matching triples divided by the size of the largest candidate set can be computed for up to six different combinations and each of these six ratios is transferred in the answer message if available. These ratios can also be unknown or undefined (e.g. because a certain position carries a fixed value in the triple pattern).

If somebody wants to estimate the number of matches for the triple pattern (?person, ns:telephone, ?phone) by a lookup of (*, ns:telephone, *) the answer might be: “there are 60,000 triples matching this pattern; the known ratios are: (a) if just a Bloom filter for the subject is defined: 2, (b) if a Bloom filter for subject and object is defined: 0.01”. The node who sent the lookup, would now count the number of known valuation candidates for ?person and multiply this number with ratio (a).

This leads to the question what happens if a node has a candidate set for ?phone but not for ?person and no such ratio is known. In this case, the node would send a second lookup request that contains a Bloom filter for the valuation candidates of ?phone as in “match count with Bloom filters”; the remote peer would update its statistics and return the newly calculated ratio. This has to be done for the triple pattern only once, as the calculated ratio can be used in the future.

Match ratios are updated permanently. Every time a node decides to fetch all triples matching a triple pattern, it sends Bloom filters to reduce the number of returned matches. This gives the remote node the chance to calculate the ratio of the number of matches (respecting the Bloom filter) divided by the number of elements written into the Bloom filter. This new ratio can then be incorporated into a mean match ratio that is returned in future requests.

Besides updating the cached ratios when triple patterns are selected for actual triple lookups, we perform lookups as in “match count with Bloom filters” at a rate of 5% per chance even if a match ratio is known. This reduces the problem of lock-in effects in case a first estimation was extremely unfavorable.
4 Query Evaluation

In summary, a client requests the match count for a triple pattern \((s,p,o)\) with \(s,p,o \in \mathcal{U} \cup \mathcal{L} \cup \mathcal{B} \cup \{\ast\}\) and receives a tuple \((m,r_s,r_p,r_o,r_{sp},r_{so},r_{po}) \in \mathbb{N} \times \{\mathbb{R} \cup \{\text{unknown}\}\}\)\(^6\), where \(m\) represents the number of triples matching the pattern if candidate sets are ignored, and the \(r\) values represent the match ratios if Bloom filters are known at the positions specified in the index. If no ratio of matches is known for the present combination of valuation candidates, it is generated with the “match count with Bloom filters” method. The client will then multiply the \(r_?\) value with the number of valuation candidates and use the result as an estimation of the number of matches.

4.4.5 Wrappers

Besides caches, several wrappers can be used for pruning when it is clear that a triple pattern cannot be better than the best pattern found so far or when determining the triple pattern’s retrieval performance is too expensive.

Block lookups of unbound triples

The first pruning strategy is to block lookups of unbound triples (see listing 4.18). A triple pattern is considered bound if it does not contain any variables or if it contains at least one variable that has occurred in a previously processed triple pattern. This rule is activated when the first triple pattern with a variable has been processed. The effect of this strategy is that the query processing is free to start with any triple pattern of the query but then only allowed to process adjacent triple patterns. Therefore strategies using this wrapper are denoted with the attribute connected. The consequences of this wrapper will be discussed in section 4.7. This strategy is implicitly used in some methods described in [164] as well.

Block variables in the lookup key

Another pruning strategy is to block variables in the lookup key. While it is possible to lookup a triple pattern by a variable if this variable has a known candidate set, this is a very expensive operation. A variable with 1000 valuation candidates generates 2000 messages for determining the match count (see next wrapper below) and therefore a huge overhead. This holds for both, determining the match count and performing the lookup. For that reason, we allow blocking this operation completely as long as other lookups by URIs or literals are possible (see listing 4.19).
4.4 Selection of Lookups

```java
class BlockLookupsOfUnboundTriplesRPCalculator implements RPCalculator {
    RPCalculator parent;
    hasBoundVariable(AX[3] t, Pow(AX)[3] cand) : boolean begin
        return \exists p \in \{S, P, O\} : t[p] is variable \land cand[t[p]] \neq \Delta;
    end

    /* The rule is only activated if at least one triple pattern contains a variable with known candidate set */
    active = \exists t \in nonProcessedQueryTriples : hasBoundVariable(t, cand);
    if active \land \neg hasBoundVariable(t) then
        return worstDefined;
    return parent.calcRP(t, pos, cand);
end
```

Listing 4.18: Block lookups of unbound triples.

```java
class BlockVariablesInLookupKeyRPCalculator implements RPCalculator {
    RPCalculator parent;
    hasConstant(AX[3] t) : boolean begin
        return \exists p \in \{S, P, O\} : t[p] is no variable;
    end

    /* The rule is only activated if at least one triple pattern has a constant value */
    active = \exists t \in nonProcessedQueryTriples : hasConstant(t);
    if active \land t[pos] contains variable then
        return worstDefined;
    return parent.calcRP(t, pos, cand);
end
```

Listing 4.19: Block variables in the lookup key.
4 Query Evaluation

Resolve variables in the lookup key

The exact opposite of the previous wrapper is resolve variables in the lookup key. Instead of denying a lookup by a variable, the variable is resolved to a set of lookups (one new lookup for each valuation candidate). The final match count is then calculated by summing up the match counts of the individual lookups. Listing 4.20 illustrates this. This method creates a huge overhead but allows for good load balancing.

```java
class ResolveVariablesInLookupKeyRPCalculator implements RPCalculator

RPCalculator parent;

    if \exists p \in pos : cand[p] = \Delta then
        return worstDefined;
    c[S] = (cand[S] \neq \Delta \land S \in pos ? cand[S] : \{t[S]\});
    c[P] = (cand[P] \neq \Delta \land P \in pos ? cand[P] : \{t[P]\});
    c[O] = (cand[O] \neq \Delta \land O \in pos ? cand[O] : \{t[O]\});
    return \sum_{t' \in c[S] \times c[P] \times c[O]} parent.calcRP(t', pos, cand);
end
```

Listing 4.20: Resolve variables in lookup key.

Block lookups of rdf:type

A very simple technique to reduce load hot-spots is to block lookups of rdf:type as these may occur very frequently in RDF graphs and create high load on the peer responsible for this key. Instead of disallowing such lookups categorically, they are scored with the retrieval performance worstDefined and therefore discouraged. This ensures that queries can still be answered if the only possible next step is a lookup by rdf:type. Pseudo-code is given in listing 4.21.

Prevent large Bloom filters

As the Bloom filters that are used for determining the order in which triple patterns are processed can make up a substantial part of the network traffic we have developed the prevent large Bloom filters wrapper (listing 4.22). If a candidate set of a variable is very large, the variable name is substituted with a dummy name for which no candidate set exists. The Bloom filter for \Delta is very small and creates little traffic.
### 4.4 Selection of Lookups

**class** BlockRDFType\(\text{RP}\) Calculator **implements** \(\text{RP}\) Calculator

\(\text{RP}\) Calculator parent;

```
calcRP(AX[3] t, Pow\{S, P, O\} pos, Pow(AX)[3] cand) : \(\text{RP}\) begin
  if t[P] = \text{rdf:type} \land pos = \{P\} then
    return worstDefined;
  return parent.calcRP(t, pos, cand);
end
```

**Listing 4.21**: Block lookups of \text{rdf:type}.

This strategy has no effect unless a triple pattern contains two variables and each one has a known candidate set. This occurs in case the query graph contains loops or triple patterns are processed in an unconnected order. The “prevent large Bloom filters” wrapper is parameterized by a factor constant that defines how much a candidate set needs to be larger than the smallest candidate set of not processed triple patterns in order for not being encoded in a Bloom filter during a match count determination.

```
class PreventLargeBloomFilters\(\text{RP}\) Calculator **implements** \(\text{RP}\) Calculator

\(\text{RP}\) Calculator parent;

```
calcRP(AX[3] t, Pow\{S, P, O\} pos, Pow(AX)[3] cand) : \(\text{RP}\) begin
  smallestCandSetSize = \min_{p \in \text{variables}(T_Q)} (|cand[p]|);
  t' = t;
  if |cand[S]| > factor \cdot smallestCandSetSize then
    t'[S] = ?dummyS, cand[S] = \Delta;
  if |cand[P]| > factor \cdot smallestCandSetSize then
  if |cand[O]| > factor \cdot smallestCandSetSize then
    t'[O] = ?dummyO, cand[O] = \Delta;
  return parent.calcRP(t', pos, cand);
end
```

**Listing 4.22**: Prevent large Bloom filters.

**Favor local lookups**

In case query processing agents are allowed to migrate within the network, it may pay off to process triple patterns out of order: When an agent has to decide which
4 Query Evaluation

triple pattern to process next, it may prefer to choose triple patterns that happen
to be stored on the node that hosts the agent at that time. As a consequence, the
lookup creates no network traffic at that time. On the other hand, it limits the
capability of the agent to migrate to other nodes in the future as it increases the
payload that needs to be migrated. This strategy is encoded in favor local lookups
as shown in listing 4.23.

```java
class FavorLocalLookupsRPCalculator implements RPCalculator
    RPCalculator parent;
        if key for (t, pos) is local then
            return bestPossible;
        return parent.calcRP(t, pos, cand);
    end
```

Listing 4.23: Favor local lookups.

Favor local lookups if connected

The previous wrapper can be restricted to only favor local lookups if their triple
patterns are connected by variables to previously processed triple patterns. This
mitigates the potential problem that intermediate results become extremely large,
which results in potentially large Bloom filters and makes migration very costly.

```java
class FavorLocalLookupsIfConnectedRPCalculator implements RPCalculator
    RPCalculator parent;
        return \( \exists p \in \{S, P, O\} : t[p] \text{ is variable} \land \text{cand}[t[p]] \neq \Delta; \)
    end
        if key for (t, pos) is local \& hasBoundVariable(t, cand) then
            return bestPossible;
        return parent.calcRP(t, pos, cand);
    end
```

Listing 4.24: Favor local lookups if connected.
4.5 Migration

After selecting the next triple pattern and lookup key, the agent needs to decide whether to retrieve matching triples from the node storing these or whether to migrate the query processing to that node. This decision is based on two key factors: feasibility and benefit. A migration is feasible if the lookup key represents a fixed value, whose hash value can be determined, or a variable with only one valuation candidate.

If the agent possesses accurate estimations of the number of triples that are returned as a result of the lookup and therefore need to be transferred over the network, it can compare this effort to the effort of migrating to the target location. The effort is measured in the unit “resources”. Each triple that needs to be transferred for a lookup and each triple that is contained in the query graph accounts for three resources. Before estimating the size (resource count) of intermediate results, we need to illustrate how these are represented.

The result of a query evaluation is a set of valuations, i.e. assignments of values to variables. Such sets of valuations can be stored in relations (tables) with columns representing variables and each row representing a valuation for all variables.

Formula 4.2 on page 56 indicated that triple patterns can be processed in any order due to the join operation being associative and commutative. As joining two relations without common variables degenerates to a cross product, we defer such joins as long as possible. During the query evaluation we maintain one or more relations. Each variable of the query is contained in at most one of these relations. When a new triple pattern is processed, we distinguish between two cases: If no variable of the new triple pattern occurs in any of the previously created relations, a new relation is created from the variables and matching triples of the newly processed triple pattern. If variables of the new triple pattern occur in one or more previously generated relations, we join the matches of the triple pattern with all these relations. This is illustrated in figure 4.4 and listing 4.25.

Query: (\(?v_1, U_1, ?v_2\), (\(?v_2, U_2, ?v_3\), (\(?v_3, U_3, ?v_4\))

Variables: \(?v_1, ?v_2, ?v_3, ?v_4\)

Processed: (\(?v_1, U_1, ?v_2\), (\(?v_3, U_3, ?v_4\)

becomes

Processed: (\(?v_1, ?v_2\), (\(?v_2, U_3, ?v_3\), (\(?v_3, U_3, ?v_4\)

Variables: \(?v_1, ?v_2, ?v_3, ?v_4\)

Figure 4.4: Joining triple pattern (\(?v_2, U_2, ?v_3\)) with patterns that were processed before.
4 Query Evaluation

An intermediate result is represented by a set of relations. The resource count of an intermediate result is therefore defined by the sum of the resource counts of all relations, and the resource count of a relation is defined by the product of the number of columns and the number of rows.

When using a heuristic that provides an estimation of the match count (number of matching triples on the remote site), the agent can estimate whether it is cheaper to fetch matching triples over the network or to migrate the current intermediate result to the node responsible for the lookup key. In the latter case a local join on the remote node follows the query migration.

4.6 Processing Triple Patterns

Processing a triple pattern means collecting all matching triples, creating a relation from the matches, joining this relation with previous intermediate result relations, and storing the result as new intermediate result relation as described in the previous section.

The process of collecting matching triples creates a lot of network traffic that shall be reduced as it has been found to be the bottleneck in two distributed RDF triple stores [18, 89].
4.6 Processing Triple Patterns

Bloom filters have been identified as a suitable means to reduce network traffic by compactly encoding valuation candidates for variables. The peer answering a triple pattern lookup can then check whether triples contradict the valuation candidate set and return only those that do not contradict as illustrated in listing 4.26. The algorithm iterates over all triples that match the triple pattern while ignoring variables (see section 3.2.5 for a discussion how this can be performed efficiently) and checks for each component whether it is contained in the Bloom filter.

```
class ReceivedTripleDatabase
  lookup(\{V\}_t, \text{Pow}(\{S, P, O\}) \text{ pos}, \text{BloomFilter}[3] \text{ bloom}) : \text{Integer} \text{ begin}
    \text{result} = \{\};
    \text{for } t' \in (\text{received triples matching } t) \text{ do}
      \text{if } t'[S] \in \text{bloom}[S] \land t'[P] \in \text{bloom}[P] \land t'[O] \in \text{bloom}[O] \text{ then}
        \text{result} = \text{result} \cup \{t'\};
    \text{return } \text{result};
  \text{ end}
\text{Listing 4.26: Triple pattern lookup implementation on a remote peer.}
```

The concept of Bloom filters has been introduced in section 3.2.7. A Bloom filter is characterized by the following constants:

- \( n \), number of elements inserted into the Bloom filter,
- \( m \), number of bits in the bit array, and
- \( k \), number of hash functions.

Section 3.2.7 showed that the expected false positive rate of Bloom filters is

\[
\frac{f}{\text{false positive rate}} = \frac{(\ln 2)^k m}{n}
\]

if \( k = (\ln 2)^{\frac{m}{n}} \) hash functions are used. The question arises how this information is used to create Bloom filters for triple pattern lookups.

We extend the notation as follows:

- \( n \) represents the number of valuation candidates for a variable (this is just another interpretation of “number of elements inserted into the Bloom filter”).
- \( n_t \) represents the number of triples that match the triple pattern \( t \) when ignoring known valuation candidates.
- \( n_{tc} \) represents the number of triples that match the triple pattern \( t \) when respecting known valuation candidates.

Depending on the heuristic used to determine the retrieval performance, three different levels of information are available for the construction of Bloom filters.
4 Query Evaluation

1. The number of valuation candidates for variables is known but no estimations of \( n_t \) or \( n_{t|c} \) are available. This is the case for purely local retrieval performance metrics.

2. The number of valuation candidates for variables is known and the number \( n_t \) of matching triples, while ignoring valuation candidates, is known or reasonably estimated.

3. The number of valuation candidates for variables and the numbers of matching triples \( n_t \) and \( n_{t|c} \) are known or reasonably estimated.

The number of triples passing the Bloom filters as false positives depends on two variables, the false positive rate \( f \) and the number of true negatives \( n_t - n_{t|c} \) because each true negative is tested against the Bloom filter and is rejected with probability \( 1 - f \).

In the first scenario, where only the number of valuation candidates is known but no estimations for \( n_t \) or \( n_{t|c} \), there is little choice but choosing a fixed \( \frac{m}{n} \) ratio, i.e. a fixed targeted false positive rate. We have chosen a target rate of 1%. With \( f = 0.01 = 0.6185 \frac{m}{n} \) this gives \( \frac{m}{n} = 9.58 \approx 10 \) bits in the Bloom filter for each valuation candidate. If the number of valuation candidates \( n \) is much larger than the number of matching triples \( n_t \), this creates unnecessary overhead. This is possible because heuristics that use only local information to decide the order in which triple patterns are processed cannot guarantee to start with the least selective triple pattern.

In the second scenario, the number of valuation candidates \( n \) and the number \( n_t \) of triples matching a triple pattern, while ignoring valuation candidates, is known. This reflects the “graph statistics” and “simple match count” heuristics presented in section 4.4.2. In this case the number of true negatives is still unknown, but the false positive rate can be adapted to be lower if a large number of triples are tested against small candidate sets. As discussed before, we are using \( m = c_1 \cdot \max(n, n_{t|c}) \) to incorporate both the number of local valuation candidates as well as the number of remote elements to be tested against the Bloom filter, while using \( k = (\ln 2) \frac{m}{n} \) hash functions. Values of \( c_1 = 5 \) and \( c_2 = 2 \) have given good results in experiments.

Finally, if \( n, n_t \) and \( n_{t|c} \) are known, we can take an analytic decision. The number of true negatives shall be denoted with \( n_{t|c}' = n_t - n_{t|c} \). The traffic (in bytes) for sending a lookup with \( m \) bits in Bloom filters is estimated using a linear model

\[
\text{effort}_{\text{bloom}}(m) = \alpha_b \cdot m + \beta_b
\]

and the traffic of sending \( k \) triples is estimated using

\[
\text{effort}_{\text{triples}}(k) = \alpha_t \cdot k + \beta_t.
\]
Using this notation, the total effort for fetching triples is

\[
\text{effort} = \text{effort}_{\text{bloom}}(m) + \text{effort}_{\text{triples}}(n_{t|c} + n'_{t|c} \cdot f)
\]

\[
= \alpha_b \cdot m + \alpha_t \cdot \left( n_{t|c} + n'_{t|c} \cdot f \right) + \beta_b + \beta_t
\]

\[
= \alpha_b \cdot m + \alpha_t \cdot \left( n_{t|c} + n'_{t|c} \cdot 0.6185 \cdot m \right) + \beta_b + \beta_t.
\]

An optimal Bloom filter size \( m \) can be determined by minimizing \( \text{effort} \) with \( \frac{d\text{effort}}{dm} = 0 \).

\[
\frac{d\text{effort}}{dm} = \alpha_b - 0.4805 \cdot \frac{1}{n} \cdot n'_{t|c} \cdot \alpha_t \cdot 0.6185 \cdot m = 0.
\]

This gives

\[
m = 2.0813 \cdot n \cdot \ln \left( \frac{1}{2.0813} \cdot \frac{\alpha_t \cdot n'_{t|c}}{\alpha_b \cdot n} \right).
\]

Based on empirical data from the Lehigh University Benchmark data set [83], linear models have been fitted for the traffic of transferring Bloom filters and triples over the network using least squares linear regression.

The cost of transferring triples has been fitted to the model

\[
\text{effort}_{\text{in bytes}} = \alpha_t + \beta_t \cdot \text{triples transferred}.
\]

As the measured data had a few outliers on the predictor variable at around 8000 and 14000 triples (see figure 4.5a) that had a very high leverage, the regression was reduced to lookups returning less than 2000 triples (see figure 4.5b).

The regression model estimates

\[
\text{effort}_{\text{in bytes}} = 628 \text{ bytes} + 14.4 \frac{\text{bytes}}{\text{triple}} \cdot \text{triples transferred}.
\]

Both coefficients are highly significant and the residual errors are summarized with

<table>
<thead>
<tr>
<th></th>
<th>Minimum</th>
<th>1. Quartile</th>
<th>Median</th>
<th>3. Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-2722.16</td>
<td>-79.67</td>
<td>15.12</td>
<td>181.32</td>
<td>2418.65</td>
</tr>
</tbody>
</table>

The coefficient of \( \beta_t = 14.4 \) is quite low, considering the implication that transferring an additional triple consumes only 14.4 bytes of network traffic. The reason for this
is that the transferred sets of triples share common values at all locations where the triple pattern had URIs or literals. Furthermore, many URIs share the same prefix, which allows for good compression as well. Nevertheless, the compression of triples has not been fully optimized and could be improved further.

Similarly to the previous regression, the network traffic for submitting a request including Bloom filters has been fitted to

\[
effort\text{ in bytes} = \alpha_b \cdot \text{bloom filter size in bits} + \beta_b \cdot \text{bloom filter size in bits}.\]

(4.13)

The coefficients found are \( \alpha_b = 0.1304 \) and \( \beta_b = 907 \) leaving the following residual errors:

<table>
<thead>
<tr>
<th>Minimum</th>
<th>1. Quartile</th>
<th>Median</th>
<th>3. Quartile</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-68.807</td>
<td>-13.489</td>
<td>-7.489</td>
<td>15.676</td>
<td>94.154</td>
</tr>
</tbody>
</table>

The deviation of \( \alpha_b \) from the expected value of \( \frac{1}{8} \) and the high value of \( \beta_b \) must be mostly explained with Java’s serialization of BitSets. All messages in BabelPeers are (gzip) compressed, but while compressing a bit array with high entropy expands its size, this expansion is much smaller than the observed divergence of the expected value for \( \alpha_b \). The hash functions used in the Bloom filter are encoded by a set of seed numbers that need to be transferred along the bit array. The number of hash
functions grows with the size of the Bloom filter and therefore also the number of seeds. For large values of $n$ this becomes irrelevant, however.

Based on the presented regressions and values for $\alpha_t$, $\beta_t$, $\alpha_b$, and $\beta_b$ the size of Bloom filters is chosen as

$$m = 2.0813 \cdot n \cdot \ln \left( 53.058 \cdot \frac{n_t - n_t|c}{n} \right) \quad (4.14)$$

in this third scenario where $n$, $n_t$ and $n_t|c$ are known.

One further refinement is performed on this and the previous calculations of the Bloom filter size $m$: Each resource that shall be added into the Bloom filter is hashed onto $k$ values from $0, \ldots, 2^{32} - 1$. These hash values then need to be reduced onto the range $0, \ldots, m - 1$, which is achieved by using the modulo operation. We follow Knuth’s suggestion [109] to pick $m$ primal and therefore increment $m$ to the next bigger prime number to get a better spread of the numbers. The prime number theorem states that the distance of a prime number $N$ to the next prime number is on average $\ln N$. This gives an indication of the expansion of $m$.

### 4.7 Evaluation

The preceding sections presented various strategies for determining the order in which triple patterns shall be processed, for deciding whether to migrate the query evaluation, and for calculating Bloom filter sizes. These strategies will be evaluated in the following. As indicated before, we are mainly interested in two performance measures: network traffic in terms of bytes transferred and network traffic in terms of number of messages sent. These are motivated in section 1.1.

The benchmarks follow [164] with using the Lehigh University Benchmark (LUBM) by Guo et al. [83]. The Lehigh University Benchmark provides an ontology that describes a university environment with professors, departments, students, classes, and other concepts that can be found in a university. It contains a data generator that creates concrete instances of these concepts.

We have generated a set of 16 queries that were parameterized by one variable each (e.g., by a concrete professor or university department). This extended the query set to a total of $16 \cdot 15$ queries. Figure 4.1 on page 57 gives an example of a query whose structure is representative for 14 further queries. These 14 queries can be derived by replacing Department13 with different concrete departments. These modifications create a larger sample set that is slightly less affected by chance.
Based on this data and query set, we analyze several aspects, such as:

- the effect of local retrieval performance heuristics,
- the effect of network retrieval performance heuristics,
- the estimation accuracies of network retrieval performance heuristics,
- the effect of several wrappers, and
- the effect of allowing query migration.

Analyses are conducted in several dimensions: The first dimension is the setup of base modules and wrappers used to determine retrieval performance. Each setup is tested in two variations. In the first case, the algorithm is restricted to process triple patterns in a connected fashion. This restriction is indicated with a suffix “c” (for connected) in the name of a setup. In the second case, this restriction is dropped.

The second dimension regards the migration strategy. As local strategies have no information to decide whether to migrate or not, out of the three migration strategies (no migration, migration if it appears useful, unconditional migration) only the first and third were evaluated for local strategies. For network heuristics, all three migration strategies are analyzed.

For each combination of retrieval performance heuristic, connection restriction, and migration strategy, the effort was recorded to process the set of 240 queries. The queries were evaluated on the LUBM-3 dataset stored on a simulated Pastry network of 64 nodes. The effort is recorded as the number of messages sent to process the queries and as the number of bytes sent through the network in message payloads.

We distinguish messages in three different categories; messages that are sent for determining the order in which triple patterns are processed, messages that are sent for fetching remote triples matching a patterns, and messages that are used for migrating the query evaluation to different nodes.

The network traffic in bytes is aggregated in two different ways; once by summing up the total effort for all queries, and once by calculating the geometric mean of the number of bytes sent for the individual queries.

The geometric mean is defined as

$$\left( \prod_{i=1}^{n} a_i \right)^{\frac{1}{n}} = \sqrt[n]{a_1 \cdot a_2 \cdots a_n} = \exp \left( \frac{1}{n} \sum_{i=1}^{n} \ln a_i \right).$$

(4.15)

In contrast to the sum or arithmetic mean, it compensates for the fact that the 16 query patterns have different degrees of difficulty. If $a_i$ represents the effort for processing a query $i$ using a heuristic $a$, and $b_i$ represents the effort for processing the same query $i$ using heuristic $b$, the geometric mean determines $a$ and $b$ as equally
4.7 Evaluation

good in case \( a_1 = 10 \cdot b_1 \) and \( a_2 = \frac{1}{10} b_2 \). This holds regardless whether query 1 and
2 share the same difficulty (in terms of effort) or whether query 1 is a very simple
query that can be processed with little effort and query 2 is a very difficult query.
This is not the case for the arithmetic mean.

The ratio of the geometric means of all \( a_i \) and \( b_i \) is the same as the geometric mean
of the all ratios \( \frac{a_i}{b_i} \):

\[
\frac{\text{geom. mean}(a_i)}{\text{geom. mean}(b_i)} = \text{geom. mean} \left( \frac{a_i}{b_i} \right)
\]  (4.16)

Owing to this, the ratio of two geometric means can be used to analyze the pairwise
relative effort for a set of queries.

The geometric mean solves the problem that some queries create more effort across
all heuristics than other queries, e.g. because the result set is very large. Consider
the following example. Using migration, heuristic 2 c (a definition will follow later)
consumed a total of 47 MB of traffic. Using no migration, the same heuristic
consumed 54 MB of traffic. These figures are quite close together. The geometric
mean was approximately 50% lower in the case of migration, however. Figure 4.6
shows that the migration was over 50% more efficient in the vast majority of queries
but this majority accounted for only about 20% of the total traffic. For that reason
the arithmetic mean is driven by the queries in query sets 4, 10, and 12, while the
geometric mean prevents a higher weighting for these queries. This does not make
the geometric mean a superior measure. A different but also valid point of view
is that particularly the expensive queries with a lot of network traffic need to be
improved. This is better expressed in the arithmetic mean or the sum.

4.7.1 Local Heuristics

Section 4.4.1 introduced five different local heuristics to determine retrieval performance measures:

• Query order
• Variable counting
• Message counting
• Smallest candidate set
• Smallest candidate set ignoring fixed values

As neither of these approaches is capable of estimating the number of triples matching
a triple pattern on a remote peer, the Bloom filters are constructed to reach a constant
false positive rate of \( f = 1\% \), i.e. \( \frac{m}{n} = \frac{\ln 0.01}{\ln 0.6185} \). Multiplying this ratio by the number
of local valuation candidates determines the intended Bloom filter size. In order to
make the hashing more effective, this size is rounded up to the next bigger prime
number. Therefore, \( m = \text{nextPrime}(\lceil \ln 0.01 \cdot n \rceil) = \text{nextPrime}(\lceil 9.585 \cdot n \rceil) \) and
\( k = \text{round}(\ln 2 \cdot \frac{m}{n}) \).

Each of the heuristics is benchmarked with the restriction to process triple patterns
in a connected fashion and without this restriction. The following chains of wrappers
and base modules were used:

1. **Query order (connected)** block lookups of rdf:type, block variables in lookup
   key, (block lookups of unbound triples,) query order
2. **Variable counting (connected)** block lookups of rdf:type, block variables in
   lookup key, (block lookups of unbound triples,) variable counting
3. **Message counting (connected)** block lookups of rdf:type, block variables in
   lookup key, (block lookups of unbound triples,) message counting
4. **Smallest candidate set (connected)** block lookups of rdf:type, block variables
   in lookup key, (block lookups of unbound triples,) smallest candidate set
5. **Smallest candidate set ignoring fixed values (connected)** block lookups of rdf:type, block variables in lookup key, (block lookups of unbound triples,) smallest candidate set ignoring fixed values

The results of these benchmarks are shown in table 4.1.

Note that the total number of messages varies slightly between the configurations as 24 of the 240 queries returned no results and the strategies came to this conclusion at different points in time.

The first observation is that the “query order” strategy apparently performed worst. This is surprising as one might think that queries are intuitively written in some structured and connected order. Analysis of the orders in which triple patterns were processed confirmed this but shows that the queries tended to be constructed in a top down order. The candidates for a variable ?x were for example first required to be of type Student (a very broad class giving many hits) and were only then restricted e.g. by the attendance of some class. As broader concepts appeared first, this caused a lot of network traffic in total but also in the sense of the geometric mean. Unless a query author designs the queries specifically with query optimization in mind, the intuitive order seems to result in bad performance.

The “variable counting” strategy illustrates the importance of processing triple patterns in a connected order if little information is known about the number of matches. Without this restriction the Bloom filters are not used effectively. Triple patterns that are not connected with previously processed triple patterns carry Bloom filters encoding the set ∆ for variables. Therefore, the benefit of discarding triples before transferring them over the network vanishes and the network traffic increases tremendously.

“Variable counting”, “message counting”, and the “smallest candidate set” heuristics showed very similar behavior.

The “variable counting” heuristic is very similar to the pre-sorting: It prefers triple patterns with fixed values in the subject or otherwise the object. Therefore, the “variable counting” has the same effect as preferring a lookup by subject or object. Only the tie-breaker differs slightly from the pre-sorting.

“Message counting” and “smallest candidate set” were effectively equivalent because each triple pattern in the query set contained at least one fixed value. This renders at least one lookup of all triple patterns equal and optimal with regard to their retrieval performance (number of messages is 1 or size of the smallest candidate set is 1). Therefore, the processing order was solely determined by the pre-sorting. In the reverse conclusion (the effort of “variable counting” was about the same as
Table 4.1: Effort for processing a benchmark query set on the LUBM-3 data set. Methods with suffix “c” require that triples are processed in a connected order.

<table>
<thead>
<tr>
<th>Method</th>
<th>Index</th>
<th>Always</th>
<th>Never</th>
<th>Number of Messages</th>
<th>Bytes of Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>1 (query order)</td>
<td>never</td>
<td>0</td>
<td>2,692</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2 (var cnt)</td>
<td>never</td>
<td>0</td>
<td>2,722</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3 (msg cnt)</td>
<td>never</td>
<td>0</td>
<td>2,722</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4 (sm cand set)</td>
<td>never</td>
<td>0</td>
<td>2,720</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5 (... ign fixed)</td>
<td>never</td>
<td>0</td>
<td>2,848</td>
<td>0</td>
</tr>
</tbody>
</table>

Methods with suffix “c” require that triples are processed in a connected order.
“message counting” and “smallest candidate set”) we see that “variable counting” had little effect compared to the pre-sorting as well.

The performance of “smallest candidate set ignoring fixed values” without the restriction to process triple patterns in connected order is interesting. While this restriction was not imposed by a wrapper, it was enforced by the fact that connected triple patterns had small candidate sets ($n \ll |\Delta| = \infty$). This effectively activated the “connected order” restriction.

It is furthermore interesting to see that an unconditional migration of the query processing resulted in a $\approx 50\%$ lower geometric mean of the traffic compared to no migration, which has been already illustrated in figure 4.6 on page 90. This is an indication of badly sized Bloom filters, which are only used in case of no migration. If the number of valuation candidates was much smaller than the number of remote triples matching a triple pattern, the Bloom filters were not effective enough, allowing for too many false positives. If the number of valuation candidates was much larger than the number of remote triples matching a triple pattern, the Bloom filters consumed unnecessary space. Furthermore we see the disparity between sum (or arithmetic mean) and geometric mean. This indicates that the effect of badly sized Bloom filters had a particularly strong impact on the many queries with little effort, less so on the queries with larger effort.

4.7.2 Network Heuristics

This section continues the effort of benchmarking retrieval performance metrics and shifts the focus onto network heuristics. These heuristics differ from purely local heuristics by the fact that the network is queried to retrieve an estimation of the cost of processing various triple patterns. This gives a better foundation for query planning. Furthermore, it provides information for sizing Bloom filters and deciding whether to migrate the query evaluation. At the same time, however, querying the network creates effort that needs to pay off by the savings.

The following list presents chains of wrappers and base modules that were benchmarked. The abbreviation “std. prefix” is used to represent “block lookups of rdf:type, block variables in lookup key”.

6. *Graph statistics (connected)* std. prefix, (block lookups of unbound triples,) graph statistics

7. *Simple match count (connected)* std. prefix, (block lookups of unbound triples,) cache by pattern, simple match count

8. *Bloom match count (connected)* std. prefix, (block lookups of unbound triples,) cache by pattern and candidate set size, Bloom match count
4 Query Evaluation

9. *Relative match count (connected)* std. prefix, (block lookups of unbound triples,) relative match count (contains caches)

Though not stated explicitly here, the “graph statistics” approach uses the “cache by pattern” cache internally to prevent repetitively determining the match count of triple patterns.

Table 4.2 shows the effort for processing the set of 240 queries on a LUBM-3 data set. As the network heuristics deliver estimations for the number of triples matching a triple pattern, the table lists three different migration strategies: no migration (never), migration if it appears useful (may), and unconditional migration (always).

We can draw several conclusions from these measurements. First of all, we observe that the “graph statistics” (6, 6 c) and “simple match count” approaches (7, 7 c) perform extremely badly without the requirement to process triple patterns in a connected order. The reason for this is that valuation candidates encoded in Bloom filters create strong filters for matching triples during lookups. These are not recognized by the two approaches. Both approaches (in particular “simple match count”) are fairly reliable to select a first triple pattern with low selectivity. They do not recognize, however, that the low selectivity propagates to adjacent triple patterns due to the Bloom filters. This flaw is largely compensated by the restriction to process triple patterns in a connected order.

The variations 6 c and 7 c deliver overall fairly good results. One reason for this is their low overhead for determining the order in which triple patterns are processed (see “Order” columns in the table). This overhead is considerably smaller than the overhead of the “match count with Bloom filters” approach (8) in several dimensions. First of all, fewer messages are sent through the network, because retrieval performances are not re-evaluated when valuation candidate sets change. Second, the messages are smaller because they do not contain Bloom filters. Finally, as valuation candidates are required for when determining the query processing order, all messages for counting the number of matching triples can be sent concurrently into the network before the real query evaluation commences. This allows for a high degree of concurrency and low wall clock time overhead. One restriction has to be made, however: The graph statistics approach requires an estimation of the number of triples in the network. Such an estimation requires hashing all triples with a uniform hash function (not just by subject, predicate, and object but by a combination of the three). This creates additional overhead that is not shown in the table.

If we compare the effort (in bytes) to fetch triples of 6 c, 7 c, 8, and 9 in the variation without migration (this variation clearly distinguishes between the effort to fetch
<table>
<thead>
<tr>
<th>Method</th>
<th>Migration</th>
<th>Number of Messages</th>
<th></th>
<th></th>
<th></th>
<th>Bytes of Payload</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Order Fetch Transfer Sum</td>
<td>Order Fetch Transfer Sum</td>
<td></td>
<td>Geom. Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 (graph stat)</td>
<td>never</td>
<td>3,150 2,658 0</td>
<td>5,808</td>
<td>1,101,809</td>
<td>67,736,614</td>
<td>0</td>
<td>67,873,444</td>
<td>153,013</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>may</td>
<td>3,150 658 1,852 5,660</td>
<td>1,101,809</td>
<td>1,609,134</td>
<td>37,831,827</td>
<td>40,542,770</td>
<td>104,317</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>always</td>
<td>3,150 0 2,448 5,598</td>
<td>1,101,809</td>
<td>0</td>
<td>47,884,003</td>
<td>48,935,812</td>
<td>126,439</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 c</td>
<td>never</td>
<td>3,150 2,652 0</td>
<td>5,802</td>
<td>1,101,809</td>
<td>5,736,614</td>
<td>0</td>
<td>6,802,633</td>
<td>23,901</td>
<td></td>
</tr>
<tr>
<td>6 c</td>
<td>may</td>
<td>3,150 648 2,094 5,892</td>
<td>1,101,809</td>
<td>1,014,984</td>
<td>4,179,898</td>
<td>6,296,691</td>
<td>23,064</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 c</td>
<td>always</td>
<td>3,150 0 2,710 5,860</td>
<td>1,101,809</td>
<td>0</td>
<td>5,700,824</td>
<td>6,802,633</td>
<td>24,355</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 (simple mc)</td>
<td>never</td>
<td>2,340 2,658 0</td>
<td>4,998</td>
<td>967,369</td>
<td>66,760,289</td>
<td>0</td>
<td>67,727,658</td>
<td>151,871</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>may</td>
<td>2,340 214 2,266 4,820</td>
<td>967,392</td>
<td>605,588</td>
<td>38,356,888</td>
<td>39,929,868</td>
<td>101,954</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>always</td>
<td>2,340 0 2,448 4,788</td>
<td>967,416</td>
<td>0</td>
<td>47,783,756</td>
<td>48,751,172</td>
<td>125,347</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 c</td>
<td>never</td>
<td>2,340 2,652 0</td>
<td>4,992</td>
<td>968,721</td>
<td>12,500,865</td>
<td>0</td>
<td>13,469,588</td>
<td>26,795</td>
<td></td>
</tr>
<tr>
<td>7 c</td>
<td>may</td>
<td>2,340 280 2,462 5,082</td>
<td>968,709</td>
<td>325,860</td>
<td>6,684,282</td>
<td>7,978,851</td>
<td>23,882</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 c</td>
<td>always</td>
<td>2,340 0 2,710 5,050</td>
<td>968,721</td>
<td>0</td>
<td>10,408,081</td>
<td>11,376,802</td>
<td>26,707</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 (bloom mc)</td>
<td>never</td>
<td>4,934 2,610 0</td>
<td>7,544</td>
<td>2,732,229</td>
<td>5,621,011</td>
<td>0</td>
<td>8,353,240</td>
<td>29,942</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>may</td>
<td>4,722 1,390 1,205 7,317</td>
<td>2,565,134</td>
<td>2,365,825</td>
<td>1,825,675</td>
<td>6,756,634</td>
<td>25,432</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>always</td>
<td>4,802 0 2,728 7,530</td>
<td>2,596,718</td>
<td>0</td>
<td>5,408,394</td>
<td>8,005,112</td>
<td>29,519</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 c</td>
<td>never</td>
<td>4,904 2,670 0</td>
<td>7,574</td>
<td>2,789,481</td>
<td>14,453,577</td>
<td>0</td>
<td>17,243,058</td>
<td>35,118</td>
<td></td>
</tr>
<tr>
<td>8 c</td>
<td>may</td>
<td>4,662 1,508 1,147 7,317</td>
<td>2,590,963</td>
<td>5,863,626</td>
<td>1,383,869</td>
<td>9,838,450</td>
<td>27,995</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 c</td>
<td>always</td>
<td>4,772 0 2,878 7,650</td>
<td>2,656,334</td>
<td>0</td>
<td>12,855,457</td>
<td>15,511,791</td>
<td>35,591</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 (relative mc)</td>
<td>never</td>
<td>2,584 2,660 0</td>
<td>5,244</td>
<td>1,136,544</td>
<td>5,451,460</td>
<td>0</td>
<td>6,588,004</td>
<td>23,119</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>may</td>
<td>2,592 1,588 1,061 5,241</td>
<td>1,138,088</td>
<td>2,445,131</td>
<td>1,586,234</td>
<td>5,169,453</td>
<td>19,604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>always</td>
<td>2,588 0 2,782 5,370</td>
<td>1,131,425</td>
<td>0</td>
<td>6,067,745</td>
<td>7,199,170</td>
<td>25,439</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 c</td>
<td>never</td>
<td>2,622 2,682 0</td>
<td>5,304</td>
<td>1,190,178</td>
<td>14,045,121</td>
<td>0</td>
<td>15,235,299</td>
<td>27,448</td>
<td></td>
</tr>
<tr>
<td>9 c</td>
<td>may</td>
<td>2,564 1,702 989 5,255</td>
<td>1,132,216</td>
<td>5,820,671</td>
<td>1,190,116</td>
<td>8,152,003</td>
<td>21,685</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 c</td>
<td>always</td>
<td>2,594 0 2,866 5,460</td>
<td>1,158,748</td>
<td>0</td>
<td>12,962,142</td>
<td>14,120,890</td>
<td>29,879</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Effort for query processing of 240 queries on LUBM-3. Methods with suffix "c" require that triples are processed in a connected order.
triples and to plan the query processing order), we see that approaches 8 and 9 seem to generally produce better query processing orders. This is due to the fact that their estimation of the numbers of matches is much more accurate. On the other hand, the savings by better processing orders do not completely outweigh the additional costs incurred by the Bloom filters in this benchmark for strategy 8.

There has to be a word of caution for heuristics 6 and 7, however. These heuristics achieved their good results only if triple patterns were processed in a connected order. Figure 4.7 shows the structure of queries that suffered from the restriction to process triple patterns only in connected order in the benchmark.

The approaches 7 c, 8, and 9 selected the triple pattern (\(?author2, rdf:type, ub:FullProfessor\)) as the pattern to start the query evaluation with. This is based on the fact that this triple pattern truly has the fewest matches (427) of all.

After processing the first triple pattern, method 7 c had no choice but to continue the evaluation with collecting all 7485 publications by the full professors, i.e. continue with pattern (\(?pub, ub:publicationAuthor, ?author2\)). From there on all remaining operations were rather expensive because of the large candidate set.

Methods 8 and 9 did not have the restriction to process triple patterns in a connected order. They started with (\(?author2, rdf:type, ub:FullProfessor\)) as well but continued with looking up values for \(?author1\) based on the department (658 matches for one of the 15 queries), i.e. with (\(?author1, ub:memberOf, DEPARTMENT\)). The 658 candidates for \(?author1\) could be pruned even further by checking whether they were graduate students and finally restricted the candidates for publications much stronger than method 7 c, which found 7485 publications.

Method 6 c would have suffered equally from the restriction to process triple patterns in a connected order if the inaccuracy of estimating the number of triples had not created the false testimony that the triple pattern (\(?author1, ub:memberOf, \)
DEPARTMENT) has less matches (estimated with 7 matches) than (?author2, rdf:type, ub:FullProfessor) (estimated with 361 matches). Therefore, only a coincidence saved method 6 c from worse performance.

Figure 4.8 shows the overall traffic for each of the 240 queries split by the four strategies, 6 c, 7 c, 8, and 9. The effort for the query pattern discussed above can be found in block 8 of the figure. The methods 6 c and 7 c suffered from the same issue in queries of block 9 as well.

Table 4.2 and figure 4.8 show method 9, the “relative match count”, as clear winner of the strategies. It combines the advantages of the “simple match count” (low overhead) with the advantages of the “match count with Bloom filter” approach (high accuracy):

Table 4.2 shows that the number of messages required for determining the query order is second only to the “simple match count”. Once the relative match count values were known on the remote peers, the number of messages required by “simple match count” and “relative match count” are identical. Only during the initialization phase a few messages with Bloom filters are required to find initial match ratios. These are later updated along with the messages to fetch triples matching certain patterns.

The table shows furthermore that the messages used in “relative match count” are relatively small (the average message size for determining the retrieval performance were: 345, 414, 553, and 443 bytes for methods 6 and 6 c, 7 (+c), 8 (+c), and 9 (+c) respectively). The total number of bytes transferred for determining the retrieval performance was only slightly higher than for “graph statistics” and “simple match count”.

An important property of “graph statistics”, “simple match count”, and (with few exceptions) “relative match count” is that the information requested from the network does not depend on actual valuation candidate sets. This allows sending all requests necessary for determining the query processing order concurrently at the beginning of the query evaluation, therefore allowing for a high degree of parallelism.

Despite the low overhead, “relative match count” achieved a high precision that allows and encourages refraining from the constraint to process triples only in connected order. This reduces the danger of extremely bad performance for individual queries. We will now investigate the estimation accuracy of strategies 6 to 9.
Figure 4.8: Communication effort for processing individual queries. The figure shows 16 different blocks of 15 queries. Several values for strategy 7 c in block 8 were cropped to 80,000 bytes.
4.7 Evaluation

4.7.3 Estimation Accuracy

During the query processing, the query agent frequently needs to fetch triples matching a certain pattern from a remote (or local) peer. The match count, the retrieval performance measure of all network heuristics, estimates this number of matching triples and allows taking decisions about the query processing order, migration, and Bloom filter configurations. Precise estimations are advantageous. Figure 4.9 compares the estimated and the real number of triples matching a triple pattern and the valuation candidates. The real match count is calculated with a semijoin of the fetched triples with the newly generated intermediate result.

The left hand side of figure 4.9 shows a histogram of the ratios of the estimated match counts and the real match counts on a logarithmic scale. A rug beneath the histograms shows all occurring values as well. It helps seeing occurrences of ratios whose frequency did not add up to a visible peak in the histogram.

Correct estimations appear as peeks at position $2^0$ in the histograms (the ratio is 1 for correct estimations). The effect of the logarithmic scale is that overestimations by a factor $c$ create a peak at the same distance from $2^0$ to the right, as underestimations by a factor $\frac{1}{c}$ do to the left.

The figure shows that methods 6 c and 7 c deliver extremely bad estimations of the match count. Method 7 c achieves a certain percentage of correct estimations that stem from the first triple pattern of each query, where no valuation candidates exist. Method 6 c does not even find these match counts. Both methods produce some overestimations by a factor of more than $2^{15} = 32,768$. Sometimes method 6 c also underestimates the real match count severely. The figure shows that methods 8 and 9 provide much better estimations. A large fraction of estimates is very accurate and the margin of error of the remaining elements is much smaller.

The right hand side of figure 4.9 shows the absolute error and indicates how large these are for methods 6 c and 7 c.

Figure 4.10 shows the real and estimated match count as a xy-plot. Each dot corresponds to one triple pattern lookup. The x-coordinate describes the real number of matching triples; the y-coordinate shows the estimated number of matching triples. The figure shows that a large fraction of points lies close to the ideal line. The absolute error is slightly larger for method 9 ("relative match count").
Figure 4.9: Comparison of the estimated and real number of triples matching a triple pattern.
4.7 Evaluation

Figure 4.10: xy-plot of real and estimated match count.

4.7.4 Wrappers and Variations

So far we have compared the four major heuristics to determine the query processing order. In the following we will investigate the effect of several wrappers and “match count with Bloom filters and mismatch count”, as a variation to “match count with Bloom filters”.

Sections 4.4.3 and 4.4.5 introduced the following caches and wrappers:

- local relative match count cache
- block lookups of unbound triples
- block variables in the lookup key
- resolve variables in the lookup key
- block lookups of rdf:type
- prevent large Bloom filters
- favor local lookups
- favor local lookups if connected

As described before, the blocking and resolving of variables has no effect on the query set; neither has the blocking of lookups of rdf:type due to the pre-sorting. As the blocking of unbound triple patterns has been analyzed for all methods before
and resulted in worse performance for Bloom filter performances, this option is dropped in the following as well.

Therefore, the following chains of wrappers and base modules remain for benchmarks:

10. *Bloom match count + mismatch count* std. prefix, cache by pattern and candidate set size, match count with Bloom mismatches
11. *Local relative cache + bloom match count* std. prefix, local relative match count, cache by pattern and candidate set size, Bloom match count
12. *Bloom match count + prevent large bloom filters* std. prefix, prevent large Bloom filters, cache by pattern and candidate set size, Bloom match count
13. *Relative match count + prevent large bloom filters* std. prefix, prevent large Bloom filters, relative match count (contains caches)
14. *Relative match count + favor local lookups* std. prefix, favor local lookups, relative match count (contains caches)
15. *Relative match count + favor local lookups if connected* std. prefix, favor local lookups if connected, relative match count (contains caches)

The results are shown in table 4.3.

First we compare the different versions of Bloom filter based approaches (10-12). The “Bloom match count + mismatch count” (10) extends “Bloom match count” (8) with the capability to estimate the number of triples that are filtered by Bloom filters during the retrieval performance determination. If these triples are considered true negatives, it allows using the advanced determination of Bloom filter sizes summarized in equation 4.14 on page 87. This reduced the effort of fetching triples by approximately 5.5% (comparison of method 8 and 10 without migration) on the test data set.

The strategy “local relative cache + Bloom match count” addresses the issue that modifications in the valuation candidate sets creates additional effort because the match counts need to be reevaluated. The local relative caches become effective in case more than two triple patterns share a common variable. Table 4.3 shows that 14.3% of messages used for determining the query order were saved, which reduced 18.4% of the traffic in this category without noticeable increase in the traffic for fetching triples.

Discarding the Bloom filters for retrieval performance estimations if the candidate set of a variable exceeded the size of the smallest candidate set occurring in any non-processed triple pattern by a factor of 5 had little effect.

larger of two Bloom filters in lookups for determining the retrieval performance reduced the effort for ordering the triple patterns slightly, but had a bad impact on
<table>
<thead>
<tr>
<th>Method</th>
<th>Migration</th>
<th>Number of Messages</th>
<th></th>
<th>Bytes of Payload</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Order</td>
<td>Fetch</td>
<td>Transfer</td>
<td>Sum</td>
</tr>
<tr>
<td>10 (bloom mc)</td>
<td>never</td>
<td>4.934</td>
<td>2,610</td>
<td>0</td>
<td>7,544</td>
</tr>
<tr>
<td>10 + mismatch</td>
<td>may</td>
<td>4.722</td>
<td>1,390</td>
<td>1,205</td>
<td>7,317</td>
</tr>
<tr>
<td>10</td>
<td>always</td>
<td>4.802</td>
<td>0</td>
<td>2,728</td>
<td>7,530</td>
</tr>
<tr>
<td>11 (local relativ)</td>
<td>never</td>
<td>4.228</td>
<td>2,610</td>
<td>0</td>
<td>6,838</td>
</tr>
<tr>
<td>11</td>
<td>may</td>
<td>4.102</td>
<td>1,384</td>
<td>1,190</td>
<td>6,676</td>
</tr>
<tr>
<td>11</td>
<td>always</td>
<td>4.098</td>
<td>0</td>
<td>2,728</td>
<td>6,826</td>
</tr>
<tr>
<td>11 (bloom mc, prev. large)</td>
<td>never</td>
<td>4.906</td>
<td>2,612</td>
<td>0</td>
<td>7,518</td>
</tr>
<tr>
<td>11</td>
<td>may</td>
<td>4.722</td>
<td>1,390</td>
<td>1,205</td>
<td>7,317</td>
</tr>
<tr>
<td>11</td>
<td>always</td>
<td>4.746</td>
<td>0</td>
<td>2,644</td>
<td>7,390</td>
</tr>
<tr>
<td>12 (bloom mc, prev. large)</td>
<td>never</td>
<td>2.610</td>
<td>2,662</td>
<td>0</td>
<td>5,272</td>
</tr>
<tr>
<td>12</td>
<td>may</td>
<td>2.584</td>
<td>1,584</td>
<td>1,061</td>
<td>5,229</td>
</tr>
<tr>
<td>12</td>
<td>always</td>
<td>2.592</td>
<td>0</td>
<td>2,694</td>
<td>5,286</td>
</tr>
<tr>
<td>12 (relative mc, prev. large)</td>
<td>never</td>
<td>2.618</td>
<td>2,664</td>
<td>0</td>
<td>5,282</td>
</tr>
<tr>
<td>12</td>
<td>may</td>
<td>2.622</td>
<td>1,504</td>
<td>1,078</td>
<td>5,204</td>
</tr>
<tr>
<td>12</td>
<td>always</td>
<td>2.592</td>
<td>0</td>
<td>2,406</td>
<td>4,998</td>
</tr>
<tr>
<td>15 (relative mc, favor local)</td>
<td>never</td>
<td>2.604</td>
<td>2,664</td>
<td>0</td>
<td>5,268</td>
</tr>
<tr>
<td>15</td>
<td>may</td>
<td>2.626</td>
<td>1,488</td>
<td>1,146</td>
<td>5,260</td>
</tr>
<tr>
<td>15</td>
<td>always</td>
<td>2.610</td>
<td>0</td>
<td>2,454</td>
<td>5,064</td>
</tr>
</tbody>
</table>

Table 4.3: Effort for query processing of 240 queries on LUBM-3.
the accuracy of estimated match counts. Overall it made the performance worse (see method 12 and 13).

Finally we have analyzed the impact of favoring lookups of triple patterns that contain a component which is hashed to the node executing the query agent. This can be used for example if several triple patterns of a query share common URIs or literals. We see that always favoring local lookups increased the overall costs of processing a query. The agent collects too much information at an early stage that needs to be carried during the further process of evaluating the query. A slight variation, where local lookups are only preferred if the local lookup belongs to a triple pattern that is connected to previously processed triple patterns, on the other hand, improved the overall effort. Method 14 experienced another reduction of network traffic of 3.5% compared to method 9 (migration is allowed).

Heine’s SG4 method [94] consumed a total of 290,576,904 bytes in 8618 messages to process the set of 240 queries on LUBM-3. With all improvements implemented, the effort has been reduced to 4,988,882 bytes in 5260 messages. This represents savings of factor 58.2 or 98.2% in network traffic and 38.9% in the number of messages. Furthermore most messages required for determining the processing order of triple patterns can now be sent in parallel and require only one lookup in a hash table instead of selecting all relevant triples and checking them against a Bloom filter.

The improvements that lead to these saving are:

- gzipping messages
- sizing Bloom filters for lookups based on estimations of the number of true negatives
- sizing Bloom filters for determining the match count incorporates the number of matches of a triple pattern while ignoring valuation candidates
- remotely caching of match count ratios instead of transferring Bloom filters
- use of better hash functions in Bloom filters
- migrating query agents
- pre-sorting with preference for lookups by subject and object
- several wrappers
4.8 Related work

Several distributed RDF stores have been developed in the past; their query processing strategies shall be contrasted herein briefly. All distributed RDF stores described below process the same type of conjunctive RDF queries that has been described in this chapter unless explicitly stated otherwise.

Harth et al. describe in [89] YARS2, a cluster based distributed RDF store. It is designed for small scale clusters, as several operations require flooding requests to all nodes, and has been benchmarked on a cluster of 16 nodes in [89]. YARS2 evaluates conjunctive RDF queries using index nested loop joins. Lookups are flooded to all nodes of the cluster and matches are reported back to a join node, which in turn floods the next set of lookups. Unfortunately, the authors do not go into much detail. Queries of more than one triple pattern are said to be planned with dynamic programming but no details are published about this either.

Heine was among the first to discuss query planning of conjunctive RDF queries in DHT-based RDF triple stores [95, 94]. The methods proposed in these publications inspired some of the approaches discussed above but have been largely improved. The idea of Heine’s query processing is based on collecting a set of RDF triples that are potentially relevant for the query and performing a search for subgraph isomorphism on the query and the extracted triple set. A set of rules is used to discard collected triples early in order to accelerate the subgraph isomorphism testing.

Liarou et al. propose in [113] to index RDF triples in the Bamboo overlay network [149] by hashing subject, predicate, object, and combinations thereof. They consider the question in which order triple patterns are processed as future work and assume an a priori given order. Two different strategies are proposed to process queries. The first strategy resembles the unconditional migration described above. The authors assume that each triple pattern contains at least one constant value. Based on an ordered sequence of triple patterns (t_i) they select a sequence of nodes (n_i) where n_i is the node that is responsible for the hash value of the constant parts of triple t_i. The first node selects all triples matching t_1, creates a relation from the variables in the triple pattern and the matching triples, and forwards the relation to n_2. The following nodes n_i receive a relation from n_{i-1}, join it with matches to t_i, and forward the result to n_{i+1}. The second strategy presented in [113] proposes splitting the query into many subqueries under the light of load balancing. If a node i determines a set of valuations v_1, . . . , v_k for a variable ?x, it forks k new queries. In each of these queries, all occurrences of ?x in t_{i+1}, t_{i+2}, . . . are substituted by one concrete valuation. This strategy improves load balancing but the authors find that it increases network traffic by two orders of magnitude on their query set. For
queries with small (intermediate) result sets, splitting queries has little effect, but for queries with large (intermediate) result sets, millions of messages may be created.

Matono et al. propose in [120] the introduction of a bit-cube of size $2^c \times 2^c \times 2^c$. Each of the $3 \cdot 2^c$ slices of dimension $1 \times 2^c \times 2^c$, $2^c \times 1 \times 2^c$, or $2^c \times 2^c \times 1$ is stored in the DHT by hashing the location of the slice in the dimension of size 1. Each triple $(s, p, o)$ is hashed to a position in $2^c \times 2^c \times 2^c$, which is contained in exactly three slices. When inserting the triple, one bit indicating its position is activated in each of the three slices. When joining two or more triple patterns, the corresponding bit matrices are AND-ed and only those triples are transferred over the network, where the AND resulted in a 1. The authors do not elaborate on the choice of $c$.

Stocker et al. present in [164, 33] query optimization of conjunctive RDF queries in the scope of Jena. The authors present several approaches, such as “variable counting”, “graph statistics”, and a more advanced “probabilistic framework”. Several of the strategies discussed in [164] have extremely bad performance because the authors allow the joining of triple patterns without common variables, which of course degenerates to cross products. The “probabilistic framework” method generates very good performance but relies on extensive precomputed statistics, which are difficult to generate in a DHT based setting. Query plans generated by the “probabilistic framework” are always left-deep joining triple patterns in a connected order. They are generated iteratively before query evaluation time.

Query optimization has been addressed by Hartig and Heese in [90] on a very different level. Hartig and Heese operate on a SPARQL query graph model (SQGM) proposed in their paper. Such a SQGM represents a tree of operators that can be used to evaluate a SPARQL query. The authors present several rules to rewrite this operator tree. The evaluation of conjunctive RDF queries addressed in this thesis, however, is just one atomic operator in the SQGM. Therefore, the work is orthogonal.

Quilitz and Leser describe in [144] how federated RDF triple stores can be queried with SPARQL using their DARQ query engine. The way RDF triples are stored is comparable to Edutella [128] but very different from BabelPeers. Independent, self-contained RDF triple stores (or other data sources with a SPARQL endpoint) are assumed to publish the type of content they provide. The DARQ query engine selects relevant data sources and processes a SPARQL query across these sources. The algorithm focuses on lookups by predicate, which is explicitly discouraged in BabelPeers.

Ranger and Cloutier describe in [145] a P2P based infrastructure for RDF query evaluation based on subscription in a Pastry/Scribe network. Similar efforts have been made by Liarou et al. [114]. Groppe et al. present in [79] how a set of RDF
triples can be streamed into a network of pattern matchers and event generators to evaluate SPARQL queries. This approach delivers first results within very little time.

Several papers discuss the use of relational databases for processing SPARQL queries [61, 138, 54, 117]. The scalability and performance of these approaches when exposed to a high number of concurrent queries depends on the underlying database and whether it is capable of exploiting the particularities of RDF. DeWitt and Gray [66] and Graefe [76] survey parallel databases. A major difference to DHT-based RDF stores is that these databases can assume global knowledge, centralized services for query coordination, and extensive precalculations. These are not given in a peer-to-peer environment. Nevertheless, research on parallel databases will have important impact on DHT-based RDF stores in the future.

4.9 Conclusion

This chapter has presented and analyzed several strategies for query processing in the context of DHT-based RDF triple stores. The findings are that the additional effort for querying the network to determine a query processing order pays well off compared to strategies that do not query the network. A purely local strategy can perform arbitrarily bad if the amount of data grows.

The favored strategy “relative match count” combines the advantages of several strategies. It requires few messages for planning the query and almost all of them can be sent in parallel after a short warm-up. As the queries can be answered by simple lookups in a hash table, the overhead for this first planning phase is very small. Yet, the method reaches a high precision in forecasting the number of triples matching a triple pattern. This enables the use of agents that decide at runtime whether it is more economic to fetch triples matching a pattern or migrate the evaluation including all intermediate results to the location where these triples are stored. Furthermore it delivers information that can be used to determine the parameters for Bloom filters used for triple pattern lookups.

The approaches presented above do not require global knowledge nor excessive precalculations. They are not strictly bound to the query processing schema presented in this thesis but can be used by the algorithms proposed by Heine [94] or Liarou et al. [113] as well.
4 Query Evaluation

4.10 Outlook

Given the fact that a DHT consists of many nodes with compute, network, and storage capacity, it is of course desirable to increase the degree of parallelism during the query evaluation. First experiments that exploit relative match count factors as described above and join tree rewriting as discussed in [53] look very promising. An initial left-deep join tree is balanced into a bushy tree based on cost estimations with the goal to deliver the final result of the last join operation as soon as possible.

The time by which a join operation can deliver its result is defined recursively:

\[
\text{finalized}(R \bowtie S) = \text{join-time}(R \bowtie S) + \max(\text{available}(R), \text{available}(S))
\]

\[
= (|R| \log |R| + |S| \log |S| + |R \bowtie S|) \cdot \text{CPUspeed}
\]

\[
+ \max \left( \frac{|R|}{\text{bandwidth}} + \text{finalized}(R), \frac{|S|}{\text{bandwidth}} + \text{finalized}(S) \right)
\]

The time to calculate \( R \bowtie S \) depends on the time to join the two relations and the time until they are both available on the node performing the join. The time to join two relations is assumed to be driven by the sort operation in case of a merge join, or the size of the result in case it is expanded by the join. An input relation is available for the join when it has been completely generated and transferred over the network.

This leads to the question how the size of a relation \(|R|\) can be estimated. The leaves of the join trees are selections of triple patterns without valuation candidates. The number of matches can be queried using the match count presented before. More difficult is estimating the size of the result relation of a join \(|R \bowtie S|\). Three different cases may occur:

1. \( R \) and \( S \) are triple patterns: In this case \(|R \bowtie S|\) can be estimated as \(|R| \cdot r_S\), where \( r_S \) is a relative selectivity factor that is determined as in “relative match count”. If no information is available, a Bloom filter is used to determine a first value for \( r_S \). Later it is updated with experienced values from the past.

2. \( R \) is a relation and \( S \) is a triple pattern: In this case we can proceed similarly as above using \(|R| \cdot r_S\) but would use \( r_S = 1 \) if no value is known from the past, as no Bloom filter for \( R \) can be constructed at query planning time.

3. \( R \) and \( S \) are relations: In this case we can transform the join tree with children \( R \) and \( S \) into a left deep join tree which delivers the same number of results as \( R \bowtie S \). For the left deep join tree, we can estimate the size of the relation
using the two cases described above ($R$ and $S$ triple patterns, and $R$ relation and $S$ triple pattern).

The “relative match count” strategy provides an estimate of the number of triples matching a certain pattern. This can be extended for triple patterns with two variables but only one candidate set to estimate the ratio of unique valuations of the variable with candidate set and the total number of matching triples. This allows a better estimation of the size of the result relation $|R \bowtie S|$ after fetching and joining triple patterns, whereas the presented strategies focus solely on the network traffic.

First results indicate good speedups in particular for relations with large result sets but the work on this topic has not been finalized for publication.
4 Query Evaluation
5 Caching

During query processing the query evaluation agent has to fetch and transport triples matching a triple pattern over the network or transport intermediate results to nodes that store relevant triples. As each of these operations creates significant network traffic, it is desirable to cache and reuse previous results, and thereby limit the use of resources for query processing. This chapter presents several caching strategies that address this issue. They have been published in [14].

As described in the previous chapters, a query can be formalized as \( H(Q) \) :- \( T(Q) \), where \( H(Q) \) denotes the head of the query, i.e. a set of variables, and \( T(Q) \) denotes a set of triple patterns, the query graph.

A central concept of this chapter is the intermediate result. It consists of two components, the set of triple patterns already processed, denoted with \( T(R) \), and a relation \( R \) that stores valuations for the variables (triple patterns are assumed to be processed in a connected order). These valuations are valid according to the triple patterns already processed. Analogously to the head of a query, we use \( H(R) \) to denote the variables occurring in the relation. Finally, we denote cached intermediate results with \( \tilde{R} \) and \( T(\tilde{R}) \).

A first intermediate goal of this chapter is to introduce the caching of results of entire queries as this will present the foundation for caching of intermediate results.

Suppose node C in figure 5.1 has calculated the result for a query \( H(Q) \) :- \( T(Q) \). The most basic form of caching final results is to store this result in a local cache and to calculate a hash value \( h(H(Q), T(Q)) \) that denotes a position in the distributed
5 Caching

hash table. Instead of copying the result of the query to node \( n(h(H(Q), T(Q))) \), we keep the result at C and store just a reference to C at the location determined by the hash function. This saves network traffic.

If the query \( H(Q) :- T(Q) \) needs to be processed by another node D, this node checks first whether the result of the query is already referenced at \( n(h(H(Q), T(Q))) \). In this case, D finds the reference at A, fetches the result from C, and delivers it to the query issuer. If no reference to the query can be found, D processes the query as before and stores it with some life-time in a local cache as well as a reference in the DHT.

This approach can be improved in many directions. In order to limit the number of results that are kept in caches, we record the number of cache lookups for each query in a reference that does not point to any results initially as indicated for \( H(Q') :- T(Q') \). Only if it has been requested more frequently in a certain period of time than a given threshold, we store the results. Furthermore, we could record the effort of calculating the result and compare it to the size of the result. A result would only be stored in the DHT if recalculating the result is more expensive than storing it. For this some cost function needs to be defined, which is considered out of scope at this place.

The approach of storing results in the DHT works well in case clients submit exactly identical queries frequently. A first step to make the caching more tolerant to variations, is to reduce the impact of differently named variables. The queries \( q_1 = ?a :- (?a, U_1, U_2) \) and \( q_2 = ?b :- (?b, U_1, U_2) \) should be considered equivalent. For this, we normalize the variable names in a way such that both queries would be mapped to \( ?v_1 :- (?v_1, U_1, U_2) \). We follow Carroll’s approach [49] for the normalization, which is described below.

In the first step, we iterate over the triple patterns of the query and replace all variables with place holders. Every time we encounter a variable \( v \) for the first time,
we introduce a new named place holder □_k. All occurrences of v are substituted by □_k.

A central concept for the following normalization algorithm is the comparison of triple patterns. For URIs, literals, and variable names we use a simple lexicographical comparison ≺. For triples t_1 = (s_1, p_1, o_1) and t_2 = (s_2, p_2, o_2) we say that t_1 ≺ t_2 if s_1 ≺ s_2, or s_1 = s_2 ∧ p_1 ≺ p_2, or s_1 = s_2 ∧ p_1 = p_2 ∧ o_1 ≺ o_2. Place holders are considered lexicographically smaller than anything else. Two different place holders □_k, □_l are considered lexicographically equal.

We normalize a query now by calculating a mapping of place holders to newly generated variable names. The following loop is executed until all place holders are substituted: Sort all triple patterns according to ≺. Iterate over the sorted sequence of triple patterns. If a pattern is different from the preceding and following triple pattern (all place holders are considered equal) and contains a place holder □_k for which no replacement rule is determined, add a new rule that replaces this and all following occurrences of □_k by a new variable ?v_i. After iterating over the triple patterns, apply the replacement rules. If no replacement was possible but the triple patterns still contain place holders □_k, □_l are considered lexicographically equal.

The normalization algorithm returns a bijective function τ that translates variable names of the original query to normalized variable names. We use h(τ(H(Q)), τ(T(Q))) to store and lookup queries in the hash map. If τ(H(Q)) = τ(H(˜R)) and τ(T(Q)) = τ(T(˜R)) for a query Q and a cached intermediate result ˜R, we have found a match and can return the stored result of ˜R.

Note that the normalization algorithm does not guarantee to produce identical results for isomorphic graphs. No polynomial algorithm is known to decide the problem of graph isomorphism as of today (see e.g. [159]). In most cases, the algorithm works deterministically and very well, however (see [49]).

The caching techniques described above work well, in case queries are sent repeatedly and they are identical except for their variable naming. This, however, is unlikely to occur in the real world. We consider it more relevant that queries are largely prestructured (like prepared statements in many RDBMS interfaces) with placeholders that are filled by input given from the user via a user interface for example. Hence, query graphs will likely look alike in large parts but differ in one or two positions. This leads us to the problem of hashing intermediate results.
5 Caching

<table>
<thead>
<tr>
<th>Schema:</th>
<th>?auth</th>
<th>?authName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple 1:</td>
<td>ns:OK</td>
<td>“Odej Kao”</td>
</tr>
<tr>
<td>Tuple 2:</td>
<td>ns:DB</td>
<td>“Dominic Battre”</td>
</tr>
</tbody>
</table>

Table 5.1: Intermediate result storing people working for a specific department as well as their names.

![Usable subresult](image)
(a) Usable subresult

![Not usable subresult](image)
(b) Not usable subresult

![Not usable subresult](image)
(c) Not usable subresult

Figure 5.2: Containment of subresults.

5.1 Hashing Intermediate Results

As presented above, the result relation is calculated iteratively. The relation storing the current state of the evaluation consists of a set of tuples and a schema. The schema assigns variable names to the elements of the tuples. An intermediate result for the example query in figure 4.1 on page 57 might look like depicted in table 5.1. We see two tuples with valuations for the variables ?auth and ?authName.

This intermediate result might be interesting for a second query that wants to generate a phone book of all people working for a specific department.

For the example of table 5.1 we would calculate the hash value \( h(\tau(H(R)), \tau(T(R))) \), where \( H(R) = \{ ?auth, ?authName \} \) and \( T(R) = \{ (?auth, worksFor, Department13), (?auth, name, ?authName) \} \).

5.2 Containment of Subqueries

Suppose now we have processed queries for some time and stored intermediate results in the DHT-based cache in case they occurred with a certain frequency. The question is now, how we can utilize these cached values and which ones can be utilized?
5.2 Containment of Subqueries

Suppose we look for the results of a query \( Q \) depicted in figure 5.2a. All variables \( (?v_1, \ldots, ?v_5) \) are distinct variables, i.e. appear in the head of \( Q \). Assume that the results of \( \bar{R} \) are stored in the cache and the gray nodes appear in the query head of \( \bar{R} \). In this case, we do not need to reevaluate the triples of \( T(\bar{R}) \) but can take the results and continue the evaluation for the remaining triples \( T(Q) \setminus T(\bar{R}) \).

Now, suppose that not the values of \( \bar{R} \) but of \( \bar{R}' \) are stored in the cache (see figure 5.2b). We cannot use these because the additional triple pattern \( (?v_3, \cdot, U_0) \) may have restricted the possible valuations for \( ?v_3 \) in \( \bar{R}' \) stronger than allowed for \( Q \). Hence, we cannot guarantee to receive all results if we use the values of \( \bar{R}' \).

We have seen that the set of triples processed for a subresult has to be a subset of the query triples, as additional triples impose additional restrictions on the valuations.

Now consider figure 5.2c in which the relation \( \bar{R}'' \) stores only valuations of variable \( ?v_2 \). The valuations for \( ?v_3 \) and \( ?v_4 \) have been discarded. This subresult is not usable either because we cannot join it with the triples matching the triple pattern \( (?v_5, \cdot, ?v_4) \) correctly. Hence, we see that each variable that occurs in the triple patterns processed for the subresult and in the query triples needs to occur in the schema of the subresult. A second argument for not being able to use \( \bar{R}'' \) is that valuations for \( ?v_3 \) and \( ?v_4 \) have to be returned for query \( Q \), but these are not available in \( \bar{R}'' \).

Formally, we can say that we can use a cached intermediate result \( \bar{R} \) if

\[
T(\bar{R}) \subseteq T(Q)
\]\[
\land \forall v \in \left( \text{var} \left( T(Q) \setminus T(\bar{R}) \right) \cap \text{var} \left( T(\bar{R}) \right) \right) : v \in H(\bar{R})
\]\[
\land \forall v \in H(Q) : v \in \text{var} \left( T(\bar{R}) \right) \rightarrow v \in H(\bar{R}).
\]

If we add the denormalization of variable names, we get

\[
\exists \tau : \tau \left( T(\bar{R}) \right) \subseteq T(Q)
\]\[
\land \forall v \in \left( \text{var} \left( T(Q) \setminus \tau \left( T(\bar{R}) \right) \right) \cap \text{var} \left( \tau \left( T(\bar{R}) \right) \right) \right) : v \in \tau \left( H(\bar{R}) \right)
\]\[
\land \forall v \in H(Q) : v \in \tau \left( \text{var} \left( T(\bar{R}) \right) \right) \rightarrow v \in \tau \left( H(\bar{R}) \right).
\]

Here, \( \tau \) is a bijective function that maps the variable names of \( \bar{R} \) to \( Q \). It is determined by recursively searching an embedding of \( \bar{R} \) in \( Q \).
5 Caching

The condition $T(\tilde{R}) \subseteq T(Q)$ is not necessary (other algorithms may cope with a violation of this assertion), but if queries are connected graphs and processed in a connected order we consider it a reasonable restriction.

5.3 Bottom Up Search for Intermediate Results

Given this formal description whether intermediate results can be used for a query, we can now ask how we locate relevant intermediate results in the cache. Formally: given an entire query $Q$, how can we partition $Q$ and determine one or more intermediate results $\tilde{R}_i$ which are cached and can be combined and used. All subsets of $Q$’s triples might be possible partitions of $Q$, but as these are too many combinations to check, we cannot perform an exhaustive search over all partitions but need some kind of strategy such as the following bottom up strategy.

In the preceding chapters, we have shown query planning strategies for DHT-based RDF stores. These strategies determine for each triple pattern of a query the estimated number of matching triples. The query processing agent evaluates then in each step the most specific triple pattern not processed, yet. As the most specific triple pattern is sometimes the most distinctive feature of a query and not common in any other query, this approach limits the reusability of cached intermediate results. Therefore, we consider the popularity of triple patterns to determine the order in which a query is processed.

For each triple pattern $(s, p, o)$ we determine a hash value $h_1(h_2(s), h_2(p), h_2(o))$ at which we count in the DHT how often that particular triple pattern occurred in queries. $h_1$ is just any hash function that produces reasonable hash numbers from three integers. $h_2$ considers different cases: In case its parameter is a variable that has not been processed, $h_2$ returns null. In case its parameter is a variable with valuation candidates, it calculates a hash value of the possible valuation candidates. In all other cases it calculates a hash value from the textual representation of the URI or literal (blank nodes in queries are treated as variables). The number of queries that contained a specific triple pattern (including possible valuations for variables) is defined as the popularity of that triple pattern.

Queries are processed starting with the most popular triple pattern, then following the triple patterns in order of descending popularity. During this process we use the restriction to process only connected triple patterns: We label the variables that occurred in triple patterns that have been processed. As long as non-processed triple patterns exist that contain labeled variables, only these may be considered for the choice of the next lookup.
After determining the first triple pattern to process, we do a lookup for an intermediate result for just this triple pattern and, in case we find one, migrate the processing of the query to the node that stores this intermediate result. Now we proceed by determining the next triple pattern that shall be evaluated and check whether there exists a stored intermediate result for both triple patterns. In the best case, where all necessary intermediate results are cached, we do not need to fetch any triples from the network but can only migrate the query evaluation until we have migrated it to the location that stores the result of the complete query. During this migration, no intermediate results need to be carried, as they are available at the destination of the migration already. In the case that the result of the entire query is not cached, it is still possible that we are able to reuse several intermediate results during the evaluation of the first patterns of the query. Query processing always starts at the triple pattern that was requested most often. Hence, in case of two similar but different queries, it is likely that the first triple patterns of both queries coincide.

As mentioned already, this approach reduces the query evaluation in the best case to determining the popularity of triple patterns and migrating to the nodes storing previously computed intermediate results. Determining the popularity is still an expensive operation because many messages are sent. The following idea reduces this.

5.4 Improved Bottom Up Search for Intermediate Results

Suppose we process a query and store the intermediate results as before. Now we store for each intermediate result a link to the intermediate result that was processed next. In figure 5.2a for example, we might store without any additional network communication at the cached intermediate result $(\tau(H(\hat{R})), \tau(T(\hat{R})))$ that $T(\hat{R}) \cup \{(?v_5, ?, ?v_4)\}$ was the subquery that was processed next. Keeping such traces of past query evaluations has several advantages. First, we save the cost of determining the next triple pattern to be processed by simply taking the one that has been chosen in the past. This may be not the most popular triple pattern but by migrating the evaluation to the node storing the intermediate result we can process it with very little costs. Before migrating, we can discard the current intermediate result, as we get it from the next node. Hence, the traffic cost is very little.

A second advantage of this approach is the better utilization of caches. This approach tends to follow recent traces for several triple patterns in a row without regarding the popularity of triples. This allows processing several triple patterns in a row almost for free. The previously described approach in contrast has to rely on the
5 Caching

Figure 5.3: Query processing with and without caching of intermediate results.

popularity of triple patterns to follow the traces. A single popular triple pattern can make the algorithm depart from the trace and render the cached trace useless.

Figure 5.3 summarizes the overall process of query evaluation with and without caching. In case caching is deactivated, the agent determines the next triple pattern to process based on selectivity estimations, considers whether migration is more economic than fetching triples with Bloom filters and, based on this decision, migrates to the location where the triples are stored or remains where it is. After that, the triples are either fetched locally or using Bloom filters from a remote peer.

In case caching is activated, the agent first checks whether its local cache knows any cached subquery that was derived from the current intermediate result in the past. If this is the case, it migrates to the location of this cached subquery if necessary and takes it as the new intermediate result. In case no previous traces are available, the agent will determine the triple pattern that shall be processed next. This defines a new intermediate result whose existence is checked and that is adopted in case it is cached anywhere. Otherwise, the query is processed as before. After each triple pattern the new intermediate result is cached in case the threshold discussed before is met.
5.5 Updates and Replacement

In order to limit the storage of intermediate results in the DHT, replacement strategies need to discard previously cached intermediate results that are not used any more, e.g. because a certain topic lost interest. A vast body of research on replacement strategies such as LRU (least recently used), MRU (most recently used), LFU (least frequently used), and others exists, see e.g. [163, 123, 141]. The expiration of an intermediate result close to the beginning of a cached trace will propagate further along the cached trace unless the intermediate results found there are shared by other cached traces. Caches can be kept up to date based on subscription of triple patterns and hash joins, see e.g. [114, 74].

5.6 Evaluation

For the evaluation we have used the Lehigh University Benchmark [83] with one university of fifteen departments. After OWL inferencing with Jena this results in approximately 300,000 triples. The triples were distributed to a simulated Pastry ring of 64 nodes. We have generated 11 query sets that sum up to 270 unique queries in total. Each set shares queries of a common structure with one distinctive parameter that states a concrete professor or a concrete department. The first query set comprises for example queries of the following type:

```
SELECT *
WHERE {
  ?X rdf:type ub:Student .
  ?Y rdf:type ub:Course .
  ?X ub:takesCourse ?Y .
}
```

The string CONCRETE_PROFESSOR is replaced by different instances to generate 50 different queries from this pattern. We believe that this type of queries becomes important when interactive websites provide forms to enter such distinctive parameters to prestructured queries.

Figure 5.4 depicts the required network communication (the factor that largely influences the query throughput) to process each of the queries, if they are executed in order. The left hand side shows the match count with Bloom filter approach described in chapter 4 in slight variation (adaptive Bloom filter sizes and message
Figure 5.4: Effort to process queries.
5.6 Evaluation

<table>
<thead>
<tr>
<th>Caching Strategy</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caching</td>
<td>0</td>
<td>7,136</td>
<td>4,233</td>
<td>5,696</td>
<td>2,982</td>
</tr>
<tr>
<td>Migration</td>
<td>11,455</td>
<td>3,343</td>
<td>3,516</td>
<td>4,200</td>
<td>4,223</td>
</tr>
<tr>
<td>Triple Fetching</td>
<td>38,580</td>
<td>25,253</td>
<td>23,079</td>
<td>43,619</td>
<td>40,364</td>
</tr>
<tr>
<td>Query Plan</td>
<td>6,945</td>
<td>12,575</td>
<td>4,458</td>
<td>12,575</td>
<td>4,692</td>
</tr>
<tr>
<td>Sum</td>
<td>56,979</td>
<td>48,307</td>
<td>35,286</td>
<td>66,090</td>
<td>52,260</td>
</tr>
</tbody>
</table>

Table 5.2: Aggregated network communication (in kB) for different caching strategies for intermediate results.

Compression where not developed at the time [14] was published; this explains why the overall traffic is higher in these benchmarks than in chapter 4; the right hand side shows the effort for the improved bottom up caching using the triple popularity for ordering the patterns in the query and creating cache entries at the first time they are requested and calculated.

The stripes show (from top to bottom) the effort for storing and looking up cache references, the effort for migration, for fetching triples from one node to another, for determining the order in which triple patterns are processed, and finally (drawn as negative numbers as this effort is identical on both sides and does not depend on the processing strategy) the effort for sending the result back to the client. The vertical segments show the development of costs over the queries of one set.

We see that in the case of caching the first queries are very expensive but the following queries require much less effort. On the long-run this pays off as shown in Table 5.2. The table shows the total communication effort per message category for different caching strategies. The caching strategies are (labeled 1 to 5):

1. No caching
2. Bottom up according to section 5.3, where intermediate results are cached the first time they are requested.
3. Improved bottom up according to section 5.4, where intermediate results are cached the first time they are requested.
4. Bottom up according to section 5.3, where intermediate results are cached if they have been requested twice.
5. Improved bottom up according to section 5.4, where intermediate results are cached if they have been requested twice.

None of these measurements used any replacement strategies for the caches as the caches remained quite small. The total amount of triples stored in the network measured to 208.7 MB. The caches consumed 20.6 MB in case of strategy 2 and 3, and 17.3 MB in case of strategy 4 and 5.
Strategy 2 saved 15% of the network traffic, strategy 3 saved 38%. Strategy 4 was even more expensive than the one without caching (1). The savings depend largely on the number of similarly structured queries. If we consider only the steady state (neglect the first 4 queries in each query set), the average query processing cost of strategy 1 is 194.6 kB, while strategy 3 requires just 52.5 kB on average per query, a saving of 73%.

In case the order of the queries was randomly permuted, strategies 2 to 5 required 3,007 kB, 1,217 kB, 7,053 kB, and 2,717 kB of additional network traffic compared to processing the queries in order.

Figure 5.5 shows the accumulated effort for processing the queries in a random order. We see that the caching creates a high effort when queries are issued for the first time because the popularity measure is worse than the selectivity measure for query planning, when we look at a single query. This effort pays off, when intermediate results can be reused. The more similar queries are processed they higher is the payoff. Furthermore, we see how the improved caching version can exploit the trails of intermediate results better than the other bottom-up caching version. The curves show the instant caching of intermediate results. If intermediate results are cached
only after they have been requested and calculated twice, the curves bend later but reach approximately the same slope as depicted in the figure.

The statistics presented in this chapter represent of course only the specific data set and queries analyzed. The problem domain has a large influence on the performance of the caching strategies. Important questions are how the model graph, ontology and queries look like and how much correlation can be found within the queries. We have seen that queries which are very dissimilar to previously processed queries require a large effort. If most queries are ad hoc queries (in contrast to prestructured queries), the methods presented in chapter 4 give better performance because they respond better to the selectivity of triple patterns.

5.7 Related Work

Stuckenschmidt [166] proposes a technique for similarity-based query caching. A query gets processed on the results of a previous query instead of the entire model graph if it is subsumed by the results of the previous query. This regards semantic subsumption with subclass and subproperty relations.

Brunkhorst and Dhraief [42] look into semantic caching in Edutella [128], which employs a super-peer network. In their case, caches are kept at super-peers, which are responsible for query processing. From the perspective of query caching, the idea is comparable to a centralized RDF store.

Both papers, [166] and [42], address the issue of query caching in a different context than this chapter. They assume that a central server/super-peer stores recently evaluated queries and that this central instance checks whether a new query is subsumed by one of the recently evaluated queries. In our model, there exists no such central instance, and hence it makes less sense to check whether a new query is subsumed by a recent query. Of course it is possible to implement this strategy on each node that can be a starting point for query evaluation but that approach does not benefit from caches on different nodes and intermediate results of previous queries.

Skobeltsyn and Aberer address the issue of multi-term queries in P2P networks [160]. Coming from the area of information retrieval, they try to find documents containing a set of key words using a combination of broadcasts and result caching. Their view of query subsumption is similar to ours but cannot be applied because RDF queries containing variables are much more difficult to handle than simple keywords. Garcés-Erice et al. [73] present an indexing approach for XML documents in DHTs. Their work cannot be transferred directly because RDF graphs lack the
5 Caching

tree-structure of XML trees, in which it is possible to cache subtrees or root nodes of subtrees with certain properties. Furthermore, Garcés-Erice et al. assume a manual creation of indexes.

Halevy surveys query answering in RDBMSs using views in [85] and gives pointers to research from the database community. Here it is however generally assumed that the number of materialized views (similar to our cache entries) is roughly comparable to the size of the database schema. In our case it is very difficult to enumerate the available cache entries due to their number and their distribution in a peer-to-peer network. View selection becomes a very different issue because of the lack of global knowledge. Furthermore, the RDF data is stored in a single large relation of 3-tuples.

5.8 Conclusion

We have presented a novel strategy for caching intermediate results of the query evaluation of conjunctive triple pattern queries in DHT-based RDF stores. By recording traces of previous query evaluations, we are able to exploit structural similarity between queries and to guide the evaluation of a new query along those traces. With this, we save network communication when determining the order in which triple patterns are evaluated and when collecting triples that match a triple pattern for the evaluation. Instead of fetching triples, we can often migrate the query evaluation (almost for free) to the location where an intermediate result is cached.

This strategy has shown to decrease the network traffic significantly even though we have never posed the same query twice. In an environment, where queries are prestructured with some place holders that are filled with user input at runtime, we believe that this approach reduces the query processing times and thus opens up new fields where DHT-based RDF stores can be employed.

5.9 Outlook

The caching described in this chapter was motivated by the idea to reuse intermediate results of other queries under the light that certain subgraphs will be shared between several queries. A major difficulty is the issue of finding common subsets in queries which was addressed by counting the popularity of triple patterns and lead to sometimes unfortunate orders in which triple patterns were processed. The following
strategies show how an evaluation of prepared statements can be used to mitigate this problem.

A conjunctive query was defined as

\[
\begin{align*}
H(Q) := (s_1, p_1, o_1) & \land (s_2, p_2, o_2) \land \cdots \land (s_m, p_m, o_m) \\
T(Q) &
\end{align*}
\]

(5.1)

where \( ?x_1, \ldots, ?x_n \) are variables and each \((s_i, p_i, o_i)\) is a triple pattern, consisting of URIs, blank nodes, literals, and variables. We explained that if \( t_i = (s_i, p_i, o_i)\) denotes a triple pattern of the query, this query can be evaluated in relational algebra as

\[
\pi_H(Q) (\sigma_{t_1}(T_M) \bowtie \sigma_{t_2}(T_M) \bowtie \cdots \bowtie \sigma_{t_m}(T_M)).
\]

(5.2)

If \( t'_i \) denotes an alternation of \( t_i \) where any part (subject, predicate, or object) is replaced with a wildcard it holds that \( \sigma_{t_i}(T_M) \bowtie \sigma_{t'_i}(T_M) = \sigma_{t_i}(T_M) \). Therefore, equation 5.2 is equivalent to

\[
\pi_H(Q) (\sigma_{t_1}(T_M) \bowtie \sigma_{t_2}(T_M) \bowtie \cdots \bowtie \sigma_{t'_i}(T_M) \bowtie \cdots \bowtie \sigma_{t_m}(T_M))
\]

(5.3)

which is equivalent to

\[
\pi_H(Q) \left( (\sigma_{t_1}(T_M) \bowtie \sigma_{t_2}(T_M) \bowtie \cdots \bowtie \sigma_{t'_i}(T_M) \bowtie \cdots \bowtie \sigma_{t_m}(T_M)) \bowtie \sigma_{t_i}(T_M) \right)
\]

(5.4)

due to commutativity.

Suppose a website submits many queries that are largely prestructured except for some place holders that are substituted by input given by a user. In this case the query can be evaluated first using the place holders but no user input. Only later triple patterns containing the user input are joined against the result of the previous query evaluation. This resembles the idea of materialized views in databases.

This approach has several advantages: Common patterns among many queries do not need to be found automatically but are given explicitly by a user. Therefore, the order in which triple patterns of subqueries are processed can be based on selectivity instead of popularity of triple patterns. This reduces the costs for generating the intermediate result. As the common patterns are less likely to contain triple patterns of extremely low selectivity (such as a person’s name or telephone number), evaluating the patterns based on selectivity still has a good chance of finding common traces. Most of the ideas presented above remain in place, such as the normalization of variable names. Subquery caches can be still kept up to date by subscriptions of triple patterns.
5 Caching

These ideas can be easily embedded in SPARQL. Consider the following query:

```
SELECT ?pubName ?name
WHERE {
  { 
    ?pub type Publication .
    ?pub name ?pubName .
    ?pub pubAuth ?person .
  } 
  { 
    ?person type Person .
    ?person name ?name .
    ?institution name ?instName .
  } 
  ?institution name "TU-Berlin" .
}
```

The query shows two patterns describing publications and people that can be kept in caches and used by many queries. This is somewhat similar to the property tables described by several authors except that they are created implicitly by programmers writing query statements.
6 Load balancing

Contents

6.1 Related Work ............................................................................. 128
6.2 Strategies ................................................................................. 130
   6.2.1 Replication .................................................................... 130
   6.2.2 Relocation .................................................................... 132
6.3 Overload Detection ................................................................. 133
6.4 Benchmark and Analysis ......................................................... 135
   6.4.1 Experimental Setup ....................................................... 135
   6.4.2 Simulation ................................................................ 137
6.5 Impact on Triple Management .................................................. 141
   6.5.1 Overlay Tree ................................................................ 141
   6.5.2 Compensation of Node Failures .................................... 143
   6.5.3 Soft-State Updates ....................................................... 145
   6.5.4 RDFS Rules ................................................................ 145
6.6 Conclusion ................................................................................. 146
6.7 Outlook ..................................................................................... 146

The contents of this chapter are published in [18] and [17].

The methods presented in the previous chapters present a storage and retrieval system for RDF triples. However, a major limitation is due to the problem of load imbalance. DHTs work on the assumption that a universal hash function creates an even distribution of data over the ID space. Even if a cryptic hash function generates a very good spread of values over the ID space, the skew in URIs poses a problem. RDF suffers here from the fact that several URIs are inherently more frequent than others such as rdf:type, which assigns a class to each instance. The hash function maps all triples containing the rdf:type predicate to the same ID and therefore to the same peer in the network. This creates a tremendous storage load for this peer.

But not only predicates suffer from the problem of collisions. All transitive properties, such as rdfs:subClassOf and rdfs:subPropertyOf, foster collisions for the objects of triples if the peer-to-peer network uses forward chaining for reasoning. The idea of forward chaining is, as presented in section 3.2.4, to generate all knowledge derivable from the original data such that queries can be processed rather easily on the union
of generated and original data. If \(B\) is sub-class of \(A\), an individual of class \(B\) is automatically an individual of class \(A\) as well. In a deep hierarchy of classes or properties, the generated data will have many triples with common objects (in this case there will be many triples of type \((*, \text{rdf:type}, A)\) that collide at \(h(\text{rdf:type})\) and at \(h(A)\)). Even suggestions not to hash triples by predicate \(\text{rdf:type}\) mitigate but do not solve this problem.

A second issue creating overloaded regions (called hot-spots) is the unequal popularity of data. Most query processors try to order triple patterns in the query evaluation such that the triple patterns whose lookup returns the smallest number of triples are fetched first. As each peer follows the same strategy, this can create a prevalence of data requests issued to a particular peer which happens to store a small set of triples. The number of queries submitted to this peer can therefore establish another hot-spot.

Finally, the heterogeneous hardware in a peer-to-peer network, ranging from CPU capacity over storage capacity to network connectivity, is another serious issue for overloaded peers.

Benchmarks show that load imbalance creates a serious scalability problem. We have created DHT rings with 2, 4, 8, 16, 32, and 64 peers. Each peer served as an RDF store and continuously processed a set of 100 queries, which of course required the consultation of other peers. The processing of a new query was started instantly after the previous query was completed.

Figure 6.1 shows the performance decline if the peer-to-peer network does not employ any load balancing. Conversely, it demonstrates the positive impact of two simple load balancing strategies that are shown in this chapter.

### 6.1 Related Work

Zhu and Hu [176] give a very good overview about recent research on load balancing in DHTs based on a paper by Rao et al. [146]. It shows three principal approaches being investigated in the past few years.

The authors of Chord [165] propose to run up to \(\log N\) “virtual nodes” on each actual node, where \(N\) is the number of actual nodes in the network. These nodes have independent hash identifiers to achieve an average bound on the load imbalance. Rao et al. [146] build on this idea and allow virtual nodes with high utilization to migrate to less utilized peers in order to achieve better load balance. Zhu and Hu incorporate proximity information in this approach.
6.1 Related Work

Karger and Ruhl [105] migrate under-utilized nodes to ranges of the DHT that are highly populated. This helps if order preserving hash functions map many objects onto a small range of the DHT.

Neither of these approaches is capable of handling the discrete hot-spots encountered in DHT-based RDF stores as illustrated in figure 6.2. Owing to the scarcity of different URIs in an ontology (represented by URIs $U_1$ to $U_{10}$) one observes discrete peaks of many collisions in the DHT. Approaches as described above are incapable of splitting this load to different nodes. Approaches that hash by combinations of subject, predicate, and object, can achieve a more even load distribution but do not remove any peaks in the storage load.

Byers et al. [44] propose the “power of two” principle in which each element can be inserted into a DHT according to two different hash functions. Each insert operation is preceded by asking the peers responsible for the respective ranges of the DHT for their load. An object is then inserted at the location covered by the less utilized peer while the location of the higher utilized peer references to the other one. This approach is very close to what we will describe as static replication below. We will argue below why this approach fails to do proper load balancing in the case of DHT-based RDF stores.
6 Load balancing

Cai and Frank [45] address the issue of load balancing in RDFPeers by not storing triples at overly popular positions in ID space. This comes of course at the cost of possibly losing the correctness of the algorithm.

6.2 Strategies

There are two major categories of load balancing strategies that we consider in this chapter, namely the replication of data and the relocation of data. The following sections discuss four strategies that fall into these categories.

6.2.1 Replication

**Static replication** The simplest approach of load balancing by replication defines \( a \ priori \) a static number \( R \) of replica for each triple. Replication can be achieved for example by using not only one hash function for inserting triples but a whole family of \( R \) hash functions. Each insert operation is repeated once with each hash function; a lookup uses a randomly chosen hash function.

This strategy can be implemented easily and is often offered by peer-to-peer middlewares natively (e.g. by FreePastry). The disadvantage of this approach is the bad scalability. The load of the most occupied nodes can be reduced only by a factor of \( \frac{1}{R} \), and if only \( p\% \) of the peers of a network are responsible for a significant number of RDF triples at all, this fraction of peers can be augmented to at most \( R \cdot p\% \). These can be still few peers compared to the total network size. Further,
this kind of replication balances the load in terms of query pattern processing, but peers responsible for frequent URIs still have to store a large number of triples.

It is difficult to find an appropriate value for \( R \) in advance and, as \( R \) is global knowledge, it is difficult to change this value at runtime. Even though bigger values for \( R \) improve the load balancing, it cannot be chosen arbitrarily high. All replicas need storage capacity and create network load because of update messages due to soft state implementations. The idea of soft state is that triples have a life-time and expire unless they are refreshed by their originators. This creates a permanent flow of update messages proportional to the number of triples stored in the DHT, which increases in case of many replica.

As most DHT-based peer-to-peer networks require replication anyway, it makes sense to exploit this replication with a small constant \( R \) for load balancing. The native replication of DHTs is usually not based on a family of hash functions. Instead elements are replicated in the leaf set of a node. The first replica that receives a request intercepts and answers it. From the argument above, however, we conclude that static replication with a constant factor \( R \) is insufficient because it does not scale up to arbitrary large networks.

**Dynamic replication** As the load of peers differs greatly, it is desirable to replicate only triples of those nodes that suffer from very high loads. This provides load balancing while keeping storage requirements and the overhead of update messages within limits.

In order to initiate the replication of data due to load, we need to define a reasonable load measure that can be determined at run-time. Examples for such load measures are the number of messages received by a peer within a certain time, the total number of bytes sent in result sets by a peer within a certain time, or the average wait time of queries in a queue. Each of these measures is easy to determine locally for a single node but difficult to determine for the global network. The global values (or approximations) are necessary, however, to evaluate the relative performance of a peer. Section 6.3 discusses sampling techniques to detect whether a node suffers from high load.

Dynamic replication follows a simple pattern once a load imbalance was detected. From the sampling technique to be discussed, a node knows the load of several peers. This allows copying its content to the two least occupied nodes known and forwarding subsequent queries to either of these nodes picked at random each time. Recursively applying this procedure creates an overlay-tree for load balancing in which the leaf nodes do the actual query processing.
6 Load balancing

6.2.2 Relocation

As replication increases the number of triples in the network that need to be stored and further need to be updated in order to prevent their soft-state expiration, relocating triples becomes an interesting alternative to replication. We propose two simple relocation strategies based on the idea of constructing a binary overlay tree.

**Binary splitting with query broadcasting**  It is possible to split the data of an overloaded node arbitrarily (but deterministically for soft-state updates) into two or more equal sized parts which are disseminated to those peers that appeared to be least busy during the load detection phase. These nodes become children in the overlay tree. The union of the children’s result sets equals the result of the parent peer. A part of the data is kept local so that the parent peer does not become empty. Therefore, an overloaded peer becomes child of itself in the overlay tree. As a split peer has to broadcast incoming requests to all of its children and because the peer issuing the original request has to collect all sub-result sets, the total number of messages increases in the network. As triple patterns in the queries are much smaller than result sets, we expect the increase in the number of messages to be of little influence.

**Binary splitting with query unicasting**  If components of triples (subjects, predicates, objects) have a total order, for example by using lexicographical comparison, it is easy to define a total order on full triples as well. That way, it is possible for a peer in the overlay tree to memorize a split-element, such that all triples smaller or equal the split element are located at the left child and all remaining triples are located at the right child. Queries can sometimes be routed to only one of the children instead of being broadcasted to them. This is particularly important for Top \(k\) searches (see [21]) where requests have the form “give me up to \(k\) triples matching to the triple pattern \((A, \{A,B,C\}, ?v1)\) that appear after triple \((A,B,A)\).”

The approaches for replication and relocation based on dynamic criteria use binary or \(n\)-ary overlay trees. Routing messages to a destination in an overlay tree requires \(O(\log N + d)\) routing steps, where \(N\) is the number of peers in the DHT and \(d\) is the depth of the tree. The \(O(\log N)\) part originates from the DHT routing of a message to the root of an overlay tree. All messages within the tree can be sent directly from a parent node to its children without DHT message routing. Therefore, we do not expect this to have a serious impact on the performance.

The following section addresses the question how overloaded peers can be detected in the network without global knowledge.
6.3 Overload Detection

Figure 6.3 shows a histogram of the distribution of the number of bytes sent by peers to answer query pattern requests in a simulation of 1024 peers with no load balancing. The histogram shows a distribution very typical of various load indicators: The vast majority of peers has values close to zero while few peers have very high values. As the slow nodes of a DHT determine the number of queries that can be processed per time unit, the goal is to shift the load of these slow nodes to others.

Note the rug below the bars of the histogram in figure 6.3 that indicates encountered loads of peers that did not add up to visible bars. Without defining a crisp border, it appears reasonable to consider peers with loads to the right of the visible bars (>0.1e+05) as outliers. The central problem is that a peer needs to detect whether it shall be considered an outlier without knowing the entire distribution.

For that reason we use a sampling technique, where each peer sends its statistics to 25 (this number is called sample size, SS) randomly generated IDs after a certain period of time. At this point it compares its computed load to the 25 most recently received values from other peers. In case the local load exceeds the maximum of the remote load values by a certain sample factor (SF), it initiates a replication or split of the local data.

This approach shows some very nice features that we can observe in the lower part of figure 6.3. The graph shows the probability (y-axis) that a peer with a certain load (x-axis) collects a sample of remote load statistics that requires the local peer to be split. We see that this probability is almost zero for the majority of nodes and then increases the larger the load of a local peer is. Therefore, only nodes with very high load are split.

When considering loads >0.1e+05 as outliers, the algorithm did not produce any false positives (nodes that are split even though their load is smaller than the bound). This has proven to be important because too many splits create a tremendous network load that throttles the overall performance. On the other hand, the algorithm produces several false negatives. This, however, is no problem because peers with high load that are not detected to be outliers will be reevaluated in a later iteration. The probability of a high load peer not to be detected \(k\) times is \(1 - (1 - p)^k\), where \(p\) is the probability of a split or replication in one iteration. As this tends to 0 for large \(k\), we expect high load peers to be discovered eventually.

Another positive feature of this load detection strategy is its convergence. The splitting of high load peers has hardly any impact on the lower bound where the splitting probability shown in the lower part of figure 6.3 grows > 0 + \(\epsilon\).
Figure 6.3: Overload detection.
6.4 Benchmark and Analysis

Owing to the lack of access to resources with many nodes, our analysis of load balancing strategies falls back to simulations. We will first describe the experimental setup, then argue why we expect the simulation to deliver representative data, and finally analyze the results of the simulation.

6.4.1 Experimental Setup

RDF and RDFS data can be characterized by many factors such as the number of classes, properties, and individuals (instances of classes). Also the degree of inheritance, i.e. many individual classes, a very deep hierarchy, classes with few or many ancestors, and so on, has a high impact on performance benchmarks. The following benchmark data has been designed by Heine (see [94] for details).

The test data is generated in multiple steps. The algorithm starts with generating a class hierarchy and a property hierarchy and then proceeds to generating triples from these hierarchies.

The hierarchy generation is driven by three parameters, viz. the number of classes/properties, the depth, and the maximum number of inheritances per class. Figure 6.4 gives an example of how 9 different classes are distributed over 3 levels. Each class has up to two classes in higher levels it is derived from.

Splitting based on the median of a sample turned out not to work well. Medians of samples were often found to be 0 such that any node initiated a split, even if it stored only few triples. Using the mean and a large factor might work but it is difficult to justify any chosen sample factor.

Figure 6.4: Class/Property hierarchy for test data.
6 Load balancing

Figure 6.5: Lorenz curves of load imbalance.

for the classes and the properties are generated as described, many instances are created from these classes. Finally, randomly chosen instances are connected by random properties. These connections produce the triples.

For the following benchmarks we have created an RDF graph of 60,000 triples that describe 10,000 individuals from 200 classes connected by 50,000 triples from 200 different types of properties. The classes and properties hierarchies were generated as described above with 20 and 10 levels respectively in the hierarchy and at most 3 ancestors per node. Reasoning (forward chaining) augmented this RDF graph by approximately 135,000 additional rdfs:type triples and 330,000 additional connections between URIs.

The queries were generated starting with a random subgraph of the RDF graph. In this subgraph, we have then randomly replaced URIs with variables. Thus each query had at least one answer and up to over 1,000,000 answers in some cases.

In order to quantify the load balance, we measure the average time a triple pattern remains in the in-queue of a peer before this peer starts processing the query. If these wait-times are long for one node, this peer is overloaded and slows down the query-processing of other peers.

Figure 6.5 shows Lorenz-curves [115] of the load imbalance as defined above. For each node we sum the average queuing times of all triple pattern lookups and sort the nodes in ascending order. The figure then shows for how many percent of the

136
summed average queuing times the first \( p\% \) of peers account. It shows for example for 8 nodes that the 75\% percent of the nodes with the least average queuing times amount together for only 25\% of the sum of all average queuing times in the network. The Lorenz-curves show that the load imbalance increases with the number of peers in the network. As idle peers do not contribute to the performance of the network, we consider load imbalance the major factor for a decline in query performance as shown in figure 6.1.

### 6.4.2 Simulation

The strategies described above leave a large degree of freedom concerning the concrete implementation and choice of parameters. First, we have to choose whether to optimize the storage load or the query throughput. This drives the choice of the key indicator used in overload detection and the balancing strategy. In this context, we investigate whether it is possible to balance storage load and query performance simultaneously. After picking a load indicator, we have to choose a sampling strategy and its parameters. We close with an analysis of the overhead produced by our strategies compared to their effect.

When choosing to balance storage load, the number of triples stored at each node is used as key indicator during overload detection. However, if we choose to maximize query throughput, we have multiple indicators at hand. In real-life scenarios, we can use the queue wait times but in the simulation, we do not have this indicator. We can either use the number of triple pattern lookups or the accumulated number of bytes sent in response to these lookups. Therefore, we have analyzed in small-scale (not simulated) benchmarks whether these indicators can be used as indicators for the average queuing time.

Figure 6.6 shows the correlation between the two factors and the queuing time of triple patterns. Figures 6.6c and 6.6d show the plot after removing the two outliers shown in figures 6.6a and 6.6b. We can see a strong correlation between the number of bytes a peer sends and the average waiting time of a peer. On the other hand, we do not find a strong correlation that supports our initial hypothesis that hot-spots are created by a large number of requests even though the results are small. For that reason, we consider approaches successful if they balance the load regarding the number of bytes sent per peer. In heterogeneous environments this assumption does not hold of course.

In order to better quantify the load imbalance we use the Gini index. It expresses the ratio of the area between the 45 degree line and the Lorenz curve and the area
6 Load balancing

Figure 6.6: Effect of the factors “total bytes sent by peer” and “number of query-patterns processed by peer” on the average waiting time of query-patterns to be processed.
under the 45 degree line, which is 0.5. It ranges from 1 to 0; lower values indicate smaller imbalances between the data.

Figure 6.7 shows the development of various Gini indices over time. We have executed 30 turns of processing 300 queries on a DHT of 1024 nodes. After each turn, the nodes used the sampling based load balancing strategies described above. We have used recursive replication and split strategies whose criterion for detecting load imbalance were the number of bytes sent and the number of triples stored. We did not use the replication strategy to balance the number of stored triples, because this creates many peers with many triples that replicate recursively in each iteration.

The two diagrams of figure 6.7 show the effect of the strategies on the load imbalances regarding the number of bytes sent and the number of triples stored. The replication by number of bytes sent did not complete all 30 turns, because the growth in data exceeded the machine’s main memory.

First we look at the left diagram, which depicts the Gini indices of the number of bytes sent by peers in a iteration of 300 queries. The strategy “split by bytes sent” reaches the lowest value. In the first iterations the Gini index drops from approximately 0.85 down to 0.6. The replication strategy is not as successful in
6 Load balancing

The right diagram shows the development of storage imbalances when using the same strategies. Here, the “split by #triples stored” strategy wins. However, we see that balancing the number of bytes sent either by splitting or by replication makes the storage imbalance worse. The splitting strategy, however, is again superior to the replication.

To summarize, we have two main choices. Either we maximize query performance using the splitting strategy, which increases the storage imbalance slightly, or we choose to balance the storage load, which leads to persisting performance problems during query processing.

Figure 6.8 compares how the parameters of the sample size (SS) and sample factor (SF) influence the development of the Gini index and how much maintenance overhead is needed, expressed in number of splits initiated. We can observe that the effect on the Gini index correlates with the number of splits. SS=25, SF=1.5 has a large impact on the Gini index, while using the highest number of splits. We can further observe that smaller values for both SS and SF lead to a better load balancing. All versions tend to stabilize in the long run, after an initial number of iterations where numerous splits are performed.
6.5 Impact on Triple Management

The preceding sections have shown the principle ideas of load balancing using replication and splitting of data sets. These ideas shall be made more concrete and their implications for the overall system shall be shown in the following.

6.5.1 Overlay Tree

Load balancing can be implemented as seen before by constructing an overlay tree over the DHT when nodes detect that they are overloaded. As the DHT range assigned to a node can change at runtime (if nodes join or depart the neighborhood of a node) it is very difficult to store and maintain this range in an overlay tree. Instead, we build overlay trees only for discrete DHT positions like, for example, the hash position \( h(\text{rdf:type}) \), which stores triples with \( \text{rdf:type} \) predicate. In case a node detects that it is overloaded, it initiates the split of its most frequent hash-value.

While all triples colliding at a DHT position have the same hash value they can still be put into a total order by comparing the components that were not involved in the calculation of the hash value. This allows determining a split element that divides the triples of the peak into those lexicographically smaller and those larger than the split element. One half of the triples is relocated to another node which has had little load during the overload detection phase. The other half remains on the current node. Subsequent queries are either processed by both nodes or by one, in case the split element allows determining that a query regards only the data on one side of it.

To understand the details of this load balancing, one should first note that the replica database is now merged into the received triples database because their entries can be distinguished by the DHT at runtime. At the same time, we introduce a new remote triples database type which is responsible for storing triples in the overlay tree structure. I.e. these databases store triples they are not necessarily responsible for according to their position in the DHT. Each node can have several of these remote triples databases as it can offer its capacity to several overloaded nodes.

Figure 6.9 illustrates a possible overlay tree. Let the four pillars represent peers 1 through 4, then the setup could be established by this history for example: First, node 1 recognized a load imbalance and replaced a frequent collision from its “normal” received triples database with a link to a newly created remote triples database. Half of the triples remained in the local part of the remote triples database and half of them were moved to node 2. The external database of node 1 was linked to
Figure 6.9: Load balancing with remote triples databases.
6.5 Impact on Triple Management

The respective new remote triples database on node 2. At this point, node 2 had a remote triples database with a local part and no external part. Node 2 recognized that is was overloaded as well, and recursively split its data and moved half of the triples to node 3. At this point, node 1 recognized that it was still overloaded. Therefore, it split the local part of the received triples by creating a remote triples database that stores only half of the triples and links to another remote triples database on another node that stores the remaining triples. This effectively reduced the load of node 1 to a fourth of the initial load.

If node 1 receives a query for the ID that is stored in the remote triple database, it is possible that this query can be routed directly into either of the two branches. Otherwise, the query is forwarded into all branches and executed on each local part on the path to the leaves. The query module on the node who submitted the initial query assembles the results such that it is handled completely transparently for the upper layers. For this to work, each local part of a remote triples database that processes a query sets a field in the result set that indicates whether the query has been forwarded to an external remote triples database and if so to which one. This allows the initial query submitter to delay further processing until all result sets have arrived.

This approach reduces the storage and query load of nodes by shifting some of the triples to other nodes and thereby distributing the load. As references can be stored as IP addresses of the target nodes, forwarding queries within the tree does not consume expensive DHT routing but can be done via direct communication.

6.5.2 Compensation of Node Failures

If a node in the overlay tree crashes, a whole branch of the tree vanishes. This makes load balancing with the kind of overlay tree described rather fragile. For that reason, we need to do further modifications to the replication strategy and introduce a replicated overlay tree as depicted in figure 6.10. All nodes of the tree are of course members of the DHT ring but are connected at the same time to form a virtual tree.

We see that each node (represented by a white circle) in the depicted overlay tree is replicated twice (represented by gray circles). A parent knows its children and their replica, as well as its replica. Hence, if a node fails, a replica node can take its position and replicate data for a new backup node. The node can be determined by quorum.

A difficult question is to decide whether a split of a node shall be initiated because this decision involves the current load of a root node and its replica if either of them
being overloaded can create a bottle neck. For this reason we change the query routing strategy such that the first replica that receives a query does not intercept and process the query any more. Instead, the query is routed to the root node which then performs load balancing among itself and the replica. Only if all of them are overloaded, a split is initiated. Figure 6.11 depicts this query routing with load balancing. The very first root node decides not to process the query itself but to forward it to its rank 2 replica. This realizes that both of its children cover triples that are important to the result of the query and sends each of them a copy of the request message (solid arrow). At the same time it processes the query on local data and sends a result message to the node issuing the query (dashed arrow). The same happens recursively in the children.

As replica have to contain identical remote triples databases they share common split elements. If a split is initiated by a root node (whether it is the one at the root or at another position in the tree), it propagates a split element to the replica. Root node and replica look for possible children and communicate the children’s addresses to each other. Then, they move triples to the children. Triples are not deleted until their arrival has been confirmed. That guarantees that queries can always be answered correctly.
6.5 Impact on Triple Management

6.5.3 Soft-State Updates

This overlay tree requires little modifications to soft-state updates and RDFS reasoning. Soft-state updates are rather easy to implement. An update message can be compared to the split element of a remote triples database and executed either on its local part or forwarded to the external part. If an update message addresses the local part it gets propagated to the replica nodes.

6.5.4 RDFS Rules

The necessary modifications for RDFS reasoning are not that obvious. While RDFS rules based on a single triple in the precondition remain trivial to execute, we have to pay attention to those rules with two precondition triples because the current strategy does not guarantee that two triples with common value (e.g. ?u in rdfs9) are located on the same node in the overlay tree. We see however, that each rule of table 3.1 on page 43 contains at least one triple pattern with an RDFS URI in the predicate. These triples describe schema knowledge. As we anticipate that nodes
store much less schema knowledge than actual data, it is possible to propagate the schema knowledge to all nodes of the overlay tree (i.e. flood the tree with the schema knowledge), while each actual data triple remains to be stored on exactly one node (and its replica). Note that the propagated schema knowledge does not comprise the schema knowledge of the entire network but only that fraction which has a subject or object whose hash-value falls into the range of the root node of the overlay tree. With the strategy described, it is possible to combine correct RDFS reasoning, on the one hand, with the necessary load balancing, on the other hand.

6.6 Conclusion

We have presented and evaluated load balancing strategies that distribute the load of individual overloaded peers while not distributing the load of low loaded peers. The load balancing strategies are based on recursively extending an overlay tree where nodes either split their local triple stores and assign them to their children in the overlay tree and/or replicate their content. We have further introduced a sampling technique to determine whether a peer suffers from high load and therefore throttles the total query-processing throughput in the network.

These load balancing strategies have proven to be successful in benchmarks of up to 64 nodes (see figure 6.1). In simulations we have shown that the strategies are capable of improving the load of overloaded peers even for larger networks.

In order to address RDF Schema reasoning, which we consider necessary to exploit the real power of RDF, we have presented a data management strategy for forward-chaining. We furthermore addressed the problem of robustness to node failure and churn.

6.7 Outlook

A core problem of DHT-based RDF stores is the number of collisions at discrete ID positions due to an extreme skew in the frequency in occurrences of URIs and literals. This chapter has presented several strategies to address this issue but there is another very elegant way of working around it, which is work in progress. This is based on a data structure that does not require hashing for the storage and lookup of triple patterns but a definition of a total order of elements. Such a data structure is SkipNet [91].
SkipNet is an overlay network that has been introduced for document retrieval with locality in a DNS like namespace. This boils down to an architecture that allows routing on keys that possess a total order. Each node in the overlay network possesses a name and is responsible for elements that are greater or equal to this name but less than the name of the node’s successor. We can assign triples as names of nodes and thereby define their responsibilities.

Suppose we have a comparison predicate \( \prec \) that can compare URIs and literals. This predicate would assume \( u \prec b \prec n \prec s \) for URIs \( u \in U \), blank nodes \( b \in B \), numbers \( n \in \mathbb{R} \), and strings literals \( s \in S \). For comparing two URIs, blank nodes, or literals, a natural comparison (lexicographical or numerical comparison) is used. Given this component-wise comparison, we can define a comparison on triples. We define \( t_1 \prec t_2 \) with \( t_1 = (s_1, p_1, o_1) \) and \( t_2 = (s_2, p_2, o_2) \) iff \( s_1 \prec s_2 \) or \( s_1 = s_2 \land p_1 \prec p_2 \) or \( s_1 = s_2 \land p_1 = p_2 \land o_1 \prec o_2 \). This defines a total order on all triples.

A lookup for all triples with a certain subject \( U_1 \) and any predicate and object \((U_1, *, *)\) can be represented as a range query from \((U_1, \bot, \bot)\) until \((U_1, \top, \top)\) where \( \bot \) represents an artificial element that is considered smaller than all possible URIs, blank nodes, and literals and \( \top \) represents an artificial element that is greater than all these elements. As before, we store three copies of each triple. This time the three copies differ by the order of their elements. The first copy would represent the subject, predicate, object order \((s, p, o)\). The second copy represents the order \((p, o, s)\) and the third copy represents the order \((o, s, p)\). All copies follow the comparison by first position, second position, and third position as defined above. The type of the permutation becomes the first comparison criterion in the total order of triples.

We can define a total of \(2^3\) lookups for triples with a pattern \((s, p, o)\) where each component is either a fixed value or a wild card. These lookups are:

- \((*, *, *)\)
- \((s, *, *)\)
- \((*, p, *)\)
- \((*, *, o)\)
- \((s, p, *)\)
- \((s, *, o)\)
- \((*, p, o)\)
- \((s, p, o)\)

We see that each lookup can be described as a range query on the set of copies in a particular order. For example a lookup \((s, *, o)\) can be put as \((o, s, \bot)\)–\((o, s, \top)\).

The SkipNet architecture allows routing messages to nodes that are responsible for certain ID positions, just like DHTs. The difference is that these positions are
not represented numerically (this is allowed as well but disregarded here) but by triple permutations that possess a total order. A lookup is performed by routing a message to the lower end of the lookup range and delivering all matches until the upper end of the lookup range is reached. This may involve forwarding the lookup message to successor nodes.

Note how this solves the problem of DHTs where all triples with `rdf:type` predicate were mapped to the same indivisible and discrete location. Using the total order on triples, we can put the borders of nodes’ responsibilities at any position.

The challenge is now to assign ranges of responsibilities to nodes that provide an even load distribution. Ganesan et al. describe in [71] a load balancing approach based on two primitive operations that can be executed in a SkipNet network. The first operation (called NeighborAdjust) moves that borders of responsibility between two nodes such that both carry the same load. The second operation (Reorder) makes a node push its load to one of its neighbors and change its position in the ID space to take half the load of an overloaded node. These two operations guarantee a ratio between the highest and lowest loaded node of a constant factor. The amortized overhead is constant and close to 1.

Figure 6.12 shows on the left hand side the costs for inserting three permutations of approximately 107,500 triples in a network of 1024 nodes. The triples are permuted and inserted in the order they are read from the generated LUBM input file. The cost is expressed in the number of triples moved in the network. Inserting a triple without
load balancing accounts for a cost of 1. If load balancing requires furthermore moving 10 triple from one node to another, this accounts for a total cost of $1 + 10 = 11$. The figure shows that some load balancing operations needed to move approximately 650 triples.

The right hand side of figure 6.12 shows a histogram of the frequency with which certain costs were created. This shows that load balancing operations were extremely rare. A single peak at cost 1 is visible in the histogram. The rug indicates frequencies that did not add up to a visible peak. We see that the vast majority of triple insertions did not require any load balancing. In total triples were moves 550,395 times for inserting 322,468 permutations of triples. During these insertions a total of 4,215 load balances were executed, many of which moved hardly any triples. If triples are inserted in a random permutation, the load balancing effort is even lower.

Figure 6.13 shows the Lorenz curves of the storage load distribution after inserting the LUBM triple set into a DHT without load balancing and into a SkipNet using load balancing.
6 Load balancing

The combination of SkipNet with load balancing by Ganesan et al. provides major advantages. First, using a total order on triples instead of hashing components prevents indivisible peaks in the load. Second Ganesan’s algorithm allows for a well balanced storage load distribution between nodes at tiny costs. And finally, lookups for e.g. all instances of a certain class can be routed to a very specific position in the overlay network. Even though each triple is inserted only three times in the network, lookups for \texttt{rdf:type} and a specific class can still be routed to only the relevant peers. A collision at \texttt{rdf:type} does not exist.

Despite these advantages, implementing an RDF store on SkipNet and load balancing by Ganesan et al. provides several major challenges. The load balancing operations require a node to change its name. This requires either telling all nodes that link to the modified node of a change or a lazy update protocol that allows correct routing while routing tables are inconsistent. The second major challenge is atomicity in changing a name. Lookups and inserts may arrive while a node changes its position. This needs to be handled gracefully so that complete answers are returned at any time. A third challenge can be found in reasoning. If for example the set of triples with predicate \texttt{rdfs:subClassOf} is not stored entirely on one node, calculating the transitive closure becomes difficult. This issue can be addressed by using backward chaining during query evaluation or a strong separation of A-Box and T-Box with the assumption that the T-Box is small compared to the A-Box and can be replicated on all nodes of the network.

With these challenges, RDF stores based on SkipNet seem as an auspicious alternative but one has to realize that several major issues need to be dealt with and overcome.
7 Conclusion

The Resource Description Framework (RDF) constitutes a novel and extremely useful information model that provides new capabilities that cannot be found in common data representation formats and models such as XML, HTML, or relational databases. Among the advantages of RDF are the possibility to flexibly express information in a semi-structured way; to use globally scoped identifiers that allow for global annotation of objects and things as well as for information integration; and to infer implicit knowledge by exploiting ontologies to structure information.

If information is expressed as RDF on a global scale, we are faced with scalability problems when querying and processing this information. Several research groups have addressed this issue by storing RDF data in distributed hash tables (DHTs). This enables many nodes in the network to share the storage and query load while providing replication and fault tolerant routing for reliability.

This thesis contributes to the research on DHT-based RDF stores in two important areas, query processing and load balancing. The thesis presents a query evaluation strategy based on relational algebra. An agent, who is commissioned with the task to evaluate an RDF query, may fetch information from peers in the network and migrate with its current intermediate results to locations that store relevant information. A major task is to determine an order in which triple patterns of the query are processed such that the overall effort of the query evaluation is little. The thesis presents and analyzes heuristics to estimate the selectivity of triple patterns and to determine an order by which they are processed. The information gathered from this step allows furthermore determining parameters for Bloom filters that limit the amount of RDF data that needs to be transferred over the network. A combination of several strategies leads to a 98% reduction in network traffic compared to previous results presented in [94]. Furthermore, query evaluation is accelerated because almost all messages required for query planning can be sent into the network concurrently which exploits the parallel nature of the network and because their evaluation requires less effort on the remote peers.

Furthermore, the thesis presents and evaluates strategies to reuse intermediate results from previous queries. As DHT-based RDF stores are designed to allow high throughput query processing by many clients, the caching strategies attempt to find synergies between queries such that frequent subqueries do not need to be
7 Conclusion

evaluated over and over again. This addresses the issue that RDF data needs many
join operations during query evaluation.

Finally, the thesis presents and evaluates strategies to improve balancing of storage
and query load of the peers in the network. DHT-based RDF triple stores suffer from
a high skew in the distribution of data elements. Frequent elements are hashed to
the same position and create load hot-spots. The thesis presents strategies to detect
these hot-spots without global knowledge and to move data in order to mitigate the
issue. Benchmarks show that this improves the overall scalability.

During the research, several areas have been discovered, where first investigations
indicate promising results. These have been pointed out as future research in the
respective chapters.

Overall, the thesis presents a set of important contributions to the field of DHT-based
RDF stores and another step for the difficult quest of managing the tremendous
growth of information in modern society.
Bibliography


Bibliography


Bibliography


Bibliography


[62] Frank Dabek, Ben Y. Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common api for structured peer-to-peer overlays. In M. Frans

[63] Jos de Bruijn and Stijn Heymans. Logical foundations of (e)rdf(s): Complexity and reasoning. In Aberer et al. [3], pages 86–99.


Bibliography


[89] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In Aberer et al. [3], pages 211–224.


Bibliography


Bibliography


Bibliography


Bibliography


