P2P based RDF Querying and Reasoning for Grid Resource Description and Matching

Dissertation

von

Felix Heine

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Paderborn, Juni 2006

Datum der mündlichen Prüfung:
25. August 2006

Gutachter:
Prof. Dr. Odej Kao, Universität Paderborn
Prof. Dr. Harald Kosch, Universität Passau
Prof. Dr. Friedhelm Meyer auf der Heide, Universität Paderborn

Acknowledgment

To write a thesis like this, a lot of support is needed from various sides. Although I am not able to mention everybody who supported me during the past years, I want to pick some of them who helped much in making this thesis reality.

First of all, I want to thank my advisor Odej Kao. He always supported my work and helped me with numerous fruitful discussions. Under his lead, PC^2 is a perfect place to do research, providing a good climate and much support. Further, I am grateful to Harald Kosch and Friedhelm Meyer auf der Heide who agreed to review this thesis.

My colleagues always were willing to listen to my thoughts and problems. I am especially grateful to my office mate Matthias Hovestadt, who contributed a lot to the great atmosphere at work. The technical staff at PC^2 helped to resolve problems during my experiments on both the older pscLine and the newer ARMINIUS cluster. Special thanks to Axel Keller for his support during the final experiments.

Dominic Battré joined PC^2 at the end of 2005, and immediately dived into the topic of p2p based RDF engines. The numerous in-depth discussions with him helped me a lot. Many thanks to him also for proof-reading the thesis. André Höing supported me in the development of the prototype.

The European Commission supported this work partially through the DELIS project. The discussions and suggestions during the project meetings and mails were very stimulating and helpful for my work.

Finally, I would like to especially thank Friederike for her support and understanding. She provided me with a perfect backing. Also many thanks to little Emma who helped me a lot to forget about work for a while.

Abstract

In this thesis, we look at the problem how resources in large, heterogeneous Grids can be discovered. In world-sized Grids, two aspects of the **resource discovery** problem become especially important: **heterogeneity** and **size**.

Heterogeneity means that the types of resources included in the Grid are highly diverse. Additionally to traditional resources like high performance clusters and storage devices, any kind of service including arbitrary applications and expensive physical instruments are treated as resources. No single standard can encompass any resource to be described. As soon as there are multiple standards, additional knowledge is needed to mediate between these standards.

Size means that a scalable solution to the resource discovery problem is needed. Although there are numerous reasoning systems, they typically assume that all knowledge is collected at a single system, which is infeasible for arbitrary large collections of information.

We present a system that contributes the initial steps to the solution of the described problem. Although Grid computing is the motivating application, the scope of this thesis is larger. Its goal is to provide **a scalable approach to combine and query large-scale collections of machine-readable information**.

The main elements of the thesis are the description of the system architecture based on a structured p2p network, a dissemination algorithm that places information on well-defined nodes, a reasoning mechanism that derives new knowledge from the existing, combining information which originates from different nodes, and two query evaluation strategies.

The first evaluation strategy aims to extract all matches for a given query. We describe various strategies to minimize the network load. However, in case of queries with large result sets, an exhaustive evaluation is infeasible. Thus we present a second strategy targeting queries with a huge number of results that retrieves only a restricted number of results according to some sorting criterion. We use caching and look ahead strategies to make the algorithm efficient.

We have implemented the system prototypically. Using this implementation, we perform various experiments both on a simulation base and using real test runs to show the efficiency of the system.

Contents

1 Introduction

The idea behind Grid computing [41] is to deliver services to the end-user in a completely transparent manner. The term "Grid computing" stems from an analogy with the power grid. When using electrical power, it is neither important to know which plant generates the energy, nor to care about the power lines used to transfer the electricity. Users simply connect their devices to the power grid without further configuration. The vision of Grid computing is to enable users to utilize computational services in the same simple way.

Historically, Grids have started as a new way to share resources like high performance computing systems or large storage capacities. They were deployed for a restricted community in order to share the resources of this community. The main focus was to enable better utilization of resources.

However, the vision of Grids goes far beyond this scenario. An expert group convened by the European Union wrote three reports about the so-called "Next Generation Grid" [9, 10, 12]. Within these reports, the authors describe a vision of a ubiquitous Grid which can be used at any time for generic problems: "The end-user should interact with a Grids environment using whatever mode and device is most convenient at any time. The Grids environment should accept the user request, [...] execute the response to the request. [...] thus many of the facilities required by the end-user are provided in the Grids Foundations Middleware layer compensating for the inadequacies of operating systems.".

The key component to achieve transparency in resource access is the resource broker. It receives a request from the user and identifies candidate resources which are suitable to fulfil the request. The user or an automated process then selects target resources from this candidate set. Finally, the request is sent to the resources and the results are delivered back to the user though the Grid middleware.

In this thesis, we focus on the first step where suitable resources are identified. It is called **resource discovery** or synonymously **resource matching**. Identifying suitable resources in modern Grid systems is a complex and challenging task, which is the more difficult the larger the Grid grows.

Two aspects of the problem become most apparent: **size** and **heterogeneity**. First of all, the amount of data to process is enormous. Thus, good query evaluation strategies are needed to be able to answer the user's request in reasonable time. Second, there is a large variation in the kind of resources which are provided within the Grid. Thus there is a need to have a very flexible way to describe the Grid's resources.

However, that's not all. Grids are **highly dynamic systems**. On the one hand, the resources within the Grid have dynamic attributes like current load, available memory, etc. On the other hand, as Grids grow larger and larger, also the Grids themselves are no more static. Providers will appear and disappear at any time, and resources will be added and removed continuously. Thus the data basis containing all resource descriptions cannot be considered static. Permanent updates, insertions and deletions must be handled.

Grid resource discovery can be viewed as an instance of a broader problem, which arises in many contexts. The problem is **how to manage and query large, dynamic, and semantically heterogeneous information sources**. This problem arises e. g. in the fields of distributed databases [42], information retrieval [8], and data warehousing [66]. One of the recent applications is the so-called Semantic Web, which is "an extension of the current [Web], in which information is given well-defined meaning" (Tim Berners-Lee, [16]). The question how to **query heterogenous information** on the Semantic Web **in a scalable way** is still unresolved.

Although this thesis focusses on Grid resource matching, we keep the broader problem in mind. We argue that the main methods presented in this thesis are applicable to other use cases, while only a few details are specific to resource matching.

1.1 Problem Definition

The key problem we are going to tackle is **resource matching within large, heterogenous Grids**. Within this section, we define this problem and its boundary conditions precisely.

Resource matching is the process of comparing advertisements from providers with requests from users. The terms **providers** and **users** should be understood in a very generic manner. A provider is anyone offering a resource to the users of the Grid. Thus it does not need to be a large institution or company. Providers may have thousands of resources or just a single service or machine. The term users means anyone who wants to use a Grid's resource. This might be the human end-user. However, this might also be a program executing a large workflow seeking for a resource to execute the next item of the workflow. It might also be a resource management system which needs a suitable resource to migrate a job whose original resource has failed [58, 61, 62, 25, 26].

The term **resource** should also be understood in a generic manner. Resources of modern Grid systems can be anything. They are no more restricted to high performance computing (HPC) systems or storage capacities. A Grid resource might as well be a specialized input device for physicists as a set of sensors delivering weather data. This fact has to be reflected by the formalism we use to describe the resources.

Different users use different levels of detail to describe resources. A human end-user might need a powerful symmetric multi processor (SMP) machine to run her self-written programm. Thus the requirements are rather broad. She will need, for example, a C++ compiler compatible to a certain standard, and a certain amount of main memory and hard-disk space. On the other hand, when spare resources are searched for the migration of a job, the constraints for matching resources are much tighter. A job gets migrated by transferring an image of the process. Therefore binary compatibility with the original system is mandatory in order to be able to restart the process properly. This includes compatibility of the hardware architecture, availability of shared libraries in the correct version, etc.

In order to enable a good matching procedure fulfilling the needs of different kinds of users, we need further **background knowledge** about resources. By this, we mean domain-specific knowledge about each kind of resource used to improve the matching process. In the domain of HPC systems, this is e. g. knowledge about the compatibility of certain resources, or knowledge of hierarchies which allows to describe resources very specific and to query them on a more general level.

Thus we need a **formalism** which is generic enough to describe resources, schema knowledge about resources, and background knowledge. However, the formalism should be computationally efficient in a sense that the evaluation of large amounts of resource descriptions with respect to different schemas can be done in reasonable time. We have to decide which constructs in the logic used to describe schema knowledge are necessary for our application. However, we have to be very careful, as even simple logical formalisms tend to be computationally expensive.

It is important to stress that the background knowledge can originate from any provider in the network. Consider a query like "I'm looking for an 8-way SMP machine with 64 bit processors running a Unix operating system". A resource provider might have advertised its 8-way machine to have "Itanium2" processors running the "Debian" operating system. Thus we need three pieces of background knowledge encoding that Itanium2 are 64 bit processors and that Debian is a kind of Linux and that Linux is a kind of Unix. These pieces of knowledge might be stored on arbitrary nodes different from the provider. We aim to find and integrate every relevant piece of knowledge while answering a query.

A closely related problem is **schema matching**. If a provider uses a schema different from the user's schema, a translation has to be applied. This can be done either by defining complete translations from one schema A to another schema B, or by giving individual translations, also called **mediators**, between schema elements of A and B. We believe that individual mediators, defined using a logical framework, are much more powerful, as arbitrary combinations can be used to build the needed translation. We treat these mediators as background knowledge and aim to identify necessary mediators to answer a given query.

The stored knowledge is not static. We have to consider different types of **dynamics**. The description of resources from providers changes over time. Some attributes are highly dynamic in nature like current queue length, load, free storage space, etc. Some parts of the description change as resources are reconfigured, updated, as new software is installed, etc. Another type of dynamics to be considered is the appearance and disappearance of providers themselves. The set of providers in large Grids is unlikely to be stable. These facts have to respected.

The larger the Grid grows, the more resources are described. Therefore, there might be huge result sets for queries which are rather generic. This makes it infeasible to deliver all results of a query to the user. In these circumstances, we expect also that the user is typically not interested in every result.

Users might just seek for *any* resource matching their requirements. In this case, simply delivering the first result which have been found is enough. In other cases, the user has preferences according to some attributes of the resource. Thus there needs to be a way for the user to express a desired ranking of the result set. Typical criteria might be the price for resource usage, the network distance from the user, either in terms of latency or bandwidth, or the geographic distance. Other criteria include quality of service aspects as e. g. security guarantees or performance guarantees.

This use case can be compared with the typical operation of Google. As average queries return hundred thousands or millions of matching web sites, no one is able to look at all of them in detail. Users rely on Google's ranking capability and look at the first few pages of the result.

A major aspect of this problem is the **issue of scale**. Resource descriptions in the Grids grow on the one hand because more resources and types of resources are integrated, and on the other hand because the increasing heterogeneity demands for a very detailed resource description. Also describing the very details as e. g. needed for process migration yields huge entries for individual resources. Thus the system architecture and the query strategies must cope with these amounts of data.

As a **summary**, we consider the following setting within this thesis: There is a large set of providers. Each of them uses a schema described with a logical formalism to store knowledge about the local resources and background knowledge about resources in general. This background knowledge contains links between different schemas, enabling mediation between these schemas. The overall amount of information stored in the network is large. Both the set of providers and the knowledge each provider stores are highly dynamic. A user can formulate a query using one or multiple schemas from some of the providers and receives a list of matching resources from all providers in response, incorporating also knowledge which can be derived by using background knowledge and mediating knowledge from all providers. If the user specifies sorting criteria, the result set is a top k list.

1.2 Contribution

Within this thesis, we propose a solution to the above described problem. It consists of a **distributed hash table based peer-2-peer network** (in short, DHT based p2p network). Each provider acts as a peer within the network and publishes its knowledge to the network. Via the DHT approach data items can be assigned to well defined peers, so that a query algorithm can locate the peers storing the pieces of information needed to answer the current query. The W3C **Resource Description Framework** (RDF) together with **RDF Schema** are used to describe and query resources and schema knowledge. This combination allows basic reasoning mechanisms and schema mediation needed for Grid resource discovery while maintaining efficiency of query evaluation.

The query evaluation is prepared through publication of local knowledge from the individual providers to well-defined nodes. During the publication process, additional knowledge which can be derived from the background information is generated and stored. We describe a novel **dissemination algorithm** which takes care of the following issues: First, it ensures that the distribution of knowledge is reasonably fast and does neither consume too much bandwidth nor too much CPU time from the peers. Second, the algorithm prepares for efficient query evaluation by **forward-chaining rules to derive new knowledge**. Third, the distributed knowledge is replicated to compensate for crashed peers which leave the network without notice. Fourth, the algorithm updates and removes knowledge which has been changed or removed at the providers.

As we model knowledge with RDF and RDFS, knowledge is stored as graphs. Each provider stores a graph containing all its local knowledge and schema information. The union of these graphs, plus all knowledge which is inferred by forward-chaining from these graphs, is called the *model graph*.

The query evaluation algorithm takes a query and searches for matches within the model graph. It identifies peers which hold information relevant for the query, asks them to send this information, combines it, and delivers the results to the user. The user can define preferences which are used to rank the results. The peer, in turn, generates the results one by one and sends them back to the user.

In order to evaluate the query, we present two different strategies. The **exhaustive strategy** determines every result for a query, which can be potentially huge amounts. It works in two phases. The first phase breaks down the query to small subqueries and collects as much information from the network nodes as needed for the second phase to answer the query without further network interaction. The second strategy, the **top** k **strategy**, aims to deliver a small result set which is "good" with respect to a certain user-defined attribute. It combines the two phases of the exhaustive strategy into a single evaluation which retrieves further information from the network as needed.

In a p2p setting, the dominating factor for performance is the amount of network traffic generated. Thus we tune our methods to minimize network usage. We employ various optimizations including caching techniques, look ahead, and Bloom filters to further reduce consumed bandwidth and message counts.

Finally, we **evaluate our system both experimentally and by simulation**. The simulations yield values for a number of performance-critical characteristics, even for large numbers of nodes in the network. However, in many circumstances the simulations show a trade-off between different characteristics. Thus, we have implemented a prototype of the system and deployed it on up to 128 nodes using the computational resources of PC^2. This allows to compare the simulation results with the performance of the real system.

1.3 Outline of the Thesis

Within this thesis, we use several basic techniques as building blocks for the system for resource discovery. In the following chapter 2 we explain these foundations. These include formalisms for ontologies, the RDF / RDF Schema recommendations and existing standards for resource description in the Grid. We describe the state of the art in p2p and DHT technology, followed by a section on the subgraph isomorphism problem. We also describe Bloom filters which we use in our query processing algorithm. The next chapter 3 describes how we use RDF / RDF Schema to describe resources and background knowledge. The chapter also discusses how we organize data. Chapter 4 is devoted to the methods we have developed to distribute, mediate, and query heterogeneous RDF-based information in a scalable way. The methods are evaluated and compared in chapter 5. In chapter 6 other systems and literature relevant for our topic are described and compared with our work. As already stated, our approach is generic and has more applications than Grid resource discovery. In chapter 7, we thus we give an outlook explaining other applications which can be based upon our work. We further describe what we have left out by giving an overview of possible future work and finally summarize the thesis.

2 Foundations

Within this chapter, we explain the ideas, technologies, architectures, and algorithms which are used within this work to achieve our goals. Some of this material is applied as described in literature, much of it had to be improved and further developed to be useful in our scenario. Here, we only describe the results known a priori without our own work.

The chapter is organized as follows. In the first section, we look at the modeling perspective, answering the questions "how are resources described?", "how are queries described?", and we explore the underlying semantics which define valid matches. After a general discussion of the idea of ontologies, we describe more specifically the Resource Description Framework (RDF) and its accompanying specifications, which are the formal basis for our work. We further look at SPARQL, a query language for RDF. As an example of models for resource description in the Grid, we explain the GLUE information model.

The following section is concerned with the perspective of distributed systems. We describe the idea of p2p networks, and their evolution towards structured p2p networks. We explore the idea of achieving guarantees out of a collection of unreliable components. As main representatives of this class of systems, we explain the DHT networks Chord and Pastry. We further explain the common API for p2p networks, which will serve as our interface to the p2p routing layer.

In section 2.3, Ullmann's algorithm for detecting subgraph isomorphisms is explained. Finally, we describe Bloom filters which are important to reduce network load during query processing.

2.1 Modeling

Traditional modeling of resources is based on sets of attributes which are assigned values. Thus a certain standard defines a number of known attributes. To describe a resource, we assign values to these attributes. Any aspect of the resource which is not contained in the standard cannot be described.

Further, attribute / value based resource descriptions are not capable of integrating general statements about the resources, called *background knowledge* in this thesis.

A similar problem exists in many areas where domains and their entities have to be modeled. The concept of *ontologies* has proven to be an advanced way to model domains. We now look at this concept in more detail.

2.1.1 Ontologies

The term "ontology" is used in the literature in many different circumstances. There are also numerous different types of ontologies, and the range of underlying formalisms is broad. Books containing introductions to the topic of ontologies include Gómez-Pérez, Fernández-López, and Corcho: "Ontological Engineering" [43] and Fensel: "Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce" [39].

A classical definition of an ontology is "An ontology is an explicit specification of a conceptualization." (Tom Gruber, [46]). As reported in [47], Tom Gruber later published a revised definition on a mailing list: "Ontologies are agreements about shared conceptualizations".

What does this mean? First, an ontology is a **conceptualization**, i. e. a model of the world or a part of the world. Second, this model is specified **explicitly**. It is encoded in some logical formalism, which enables automatic processing of ontologies. Third, it is **shared**, which means that multiple parties have agreed to use it.

Many formalisms can be used to write ontologies. Among them are UML diagrams [84], ER diagrams [68], first order logic [38], and various types of description logics [7].

Typically, an ontology has two levels, the instance level and the terminological level, also called the assertional box (A-Box) and the taxonomical box (T-Box) in description logics. This separation is well-known from object oriented modeling, where we differentiate classes from objects. The instances in the A-Box model the concrete individuals in the domain of discourse, while the terminological level defines the abstract concepts and relationships between these concepts.

A standard example for this separation is an ontology about families. In the T-Box, concepts like "person", "male", "female", "father", "mother", "grandfather", etc. are defined. Additionally, relationships between these concepts can be defined like "isChildOf", "isParentOf", etc. In the A-Box, the individuals and their concrete relationships are defined, e. g. "Peter isChildOf Paul" and "Peter is male" etc.

An important part of ontologies is the possibility to encode generic facts about the domain which is modeled. For the family ontology, these can be facts like "if A isChildOf B, then B isParentOf A", or "if A is a brother of B's mother or father, then A is an uncle of B", or "a mother is always female".

So called **reasoners** or **inference engines** are tools that evaluate generic facts together with information about individuals and draw conclusions from these facts. This allows to make information explicit which is only contained implicitly in the ontology. As there are numerous types of formalisms, there are also numerous reasoners and algorithms to do the reasoning. Examples are first order logic theorem provers, DL reasoners, rule engines, etc. [94].

According to the type of formalism, the options to encode such facts are larger or smaller. Very generic and flexible formalisms like first order logic allow to express complex relationships between the concepts and roles in the domain. Ontologies using such formalisms are called **heavyweight ontologies** in contrast to **lightweight**

ontologies which use simpler formalisms with fewer possibilities to express complex relationships.

However, it is computationally demanding to evaluate heavyweight ontologies. Both first order logic and description logics, and even simple types of description logics cannot be used to process world-scale collections of resource descriptions. Although there are some initial ideas of how to distribute processing of knowledge encoded in description logics [100], there is currently no algorithm known to distribute the processing load of description logics.

2.1.2 Semantic Web

The idea of ontologies originates from AI research. However, the Semantic Web initiative has revived this idea and gave new impulses to further develop the concepts, methods, and tools.

As mentioned in the introduction, the Semantic Web is "an extension of the current [Web], in which information is given well-defined meaning" [16]. Thus everything in the Web has to be described using some formalism. This will allow to mine the wealth of information contained in the Web through programs rather than humans. Consider as an example the current way a trip booking is done on the Web. Information is scattered over numerous Web pages that contain e. g. fight routes, timetables, and availability, hotel information, car rental information, information about places of interest, etc. Currently, human beings are needed to put all this together and to develop a travel plan. The vision of the Semantic Web is to enable programs to do the same automatically upon the user's requests. The user specifies her preferences and constraints, and the rest is done by agents.

For that reason, ontologies that model the domains of interest are an important cornerstone of the Semantic Web. In this context, a set of specifications has been developed that is now being standardized through the World Wide Web Consortium (W3C) [111]. It contains multiple ontology formalisms which range from lightweight to heavyweight. The lightweight formalism is the **Resource Description Framework** (RDF) together with its schema language **RDF Schema**. The **Web Ontology Language** (OWL) family of specifications extends RDF[1] and contains three flavors of increasing complexity: OWL-Lite, OWL-DL, and OWL-Full.

Hand in hand with the increasing set of features of the W3C formalisms comes an increase in the complexity of reasoning. RDF Schema inference for instance can be evaluated by forward chaining a set of rules. This means that every piece of knowledge which can be derived is generated. The evaluation of the rules will eventually come to an end, as the set of inferable knowledge is finite. As the name indicates, OWL-DL builds upon description logics. Thus the typical way to infer knowledge from an OWL-DL knowledge base is to use an refutation proof. When a reasoner tries to evaluate the

[1]To be precise we have to mention that OWL also restricts RDF because the separation between the A-Box and the T-Box is more strict than in RDF

truth of some fact, it combines the negation of the fact with the ontology and tries to derive a contradiction. If no such contradiction emerges, the fact is considered to be true. With this approach, it is not possible to generate a priori every statement which follows logically from an OWL-DL ontology. It only allows to give answers to concrete questions. The most advanced version OWL-Full is even undecidable [5].

In the following, we concentrate on lightweight ontologies using RDF and RDF Schema. This formalism is the basis for our work. However, in the related work chapter on page 118 we will also mention the basics of OWL, especially OWL-Lite, as this is interesting for the outlook to further work following this thesis.

2.1.3 RDF and RDFS

We use RDF in combination with RDF Schema (RDFS), abbreviated together as RDF/S, to represent knowledge about resources. RDF/S is defined in a couple of W3C recommendations including an informal introduction [80], a definition of the concepts and abstract syntax [69], an explanation of the XML syntax for RDF [14], formal semantics [55], and the RDF Schema recommendation [18].

Our system does not support every aspect of RDF/S. We focus on the rules related to the class hierarchy and the property hierarchy. Because of that, the overview in this section concentrates on the supported portions of RDF/S. However, we also mention further aspects in order to give the reader an idea what we have left out. Details about the implemented parts can be found in chapter 4.

RDF has been designed in the light of the Semantic Web effort as a way to describe **resources**. The term resource within the RDF context has a slightly different meaning compared to our usage of the term resource meaning a resource in a Grid. It denotes anything which should be described in the Semantic Web. It can be a real-world object, a human being, a concept, a verb, or anything else we can think of. Within RDF, each resource is assigned a world-wide unique URI reference [15]. This is important, as it avoids name clashes through the introduction of namespaces.

The basic concept of RDF is to describe resources by triples consisting of a **subject**, a **predicate**, and an **object**. A subject and a predicate is given by a **URI reference** (URIref in short), while an object can be either a URIref or a literal like a string or a floating point number. A URIref which is the subject of some triple might be also the predicate or the object of some other triple. However, literals can only be used as objects. The intended meaning of a triple is to assert the existence of some relationship between the subject and object. The kind of relationship is given by the predicate. Thus, an RDF triple should informally be read and understood as a natural language sentence.

Now consider a set of triples describing some knowledge, called an **RDF graph**. The intended meaning is to assert all of the relationships described by the individual triples. Hence its meaning is a logical "and". It can be viewed as a directed graph with vertices containing the subjects and objects, and directed edges containing the predicates. Some vertices just serve a structural purpose, providing a link between different resources. These vertices do not need to be labeled, they are called **blank nodes** in the RDF terminology.

Identifying subjects and objects with identical URIrefs yields a directed graph which describes the relationships between different resources. The URIrefs of subjects and objects are labels for the vertices of this graph, while the predicate URIrefs are labels of the edges. Note that there might be multiple edges between two nodes, as a single pair of subject and object may be connected via different predicates. Another way to model this is to consider edge labels to be sets of URIrefs.

RDF can be serialized using different syntaxes. A basic syntax is called **N-Triples** [45]. Here, each triple is written as a single statement, followed by a period. URIrefs are written in angle brackets, and literals are enclosed in double quotes. Blank nodes are written using the notation `_:` followed by a name for the blank node which identifies it within the document. Another serialization is **RDF/XML** which defines a XML conform way to serialize triples. It is described in the RDF Primer [80].

RDF defines a special set of URIrefs that have a pre-defined meaning. This set is called the **RDF vocabulary**. Each URIref in this vocabulary uses the namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, commonly abbreviated with `rdf:`. The most important element of the RDF vocabulary is `rdf:type`. It is used to assert that an individual belongs to a certain type. Further predefined URIrefs include a predefined class that contains all properties `rdf:Property`, vocabulary to define sets of objects, and reification vocabulary.

Collections of objects are described in RDF using the URIrefs `rdf:Seq`, `rdf:Bag`, or `rdf:Alt`. Each collection is itself an object and is asserted have one of these types. Each element is connected to the collection using predefined predicates `rdf:_1`, `rdf:_2`, etc. Both `rdf:Seq` and `rdf:Bag` may contain duplicate members. The difference is that for `rdf:Bag`, the order of the elements is irrelevant, while it is relevant for `rdf:Seq`. The indented semantics of the third collection type `rdf:Alt` is to provide alternative values for some property.

As an example how these URIrefs are used, consider the set of triples in figure 2.1. It describes the set of clusters installed at PC^2. As we are not interested in their order, we use `rdf:Bag`.

Subject	Predicate	Object
`pc2:Clusters`	`rdf:type`	`rdf:Bag`
`pc2:Clusters`	`rdf:_1`	`pc2:SFB`
`pc2:Clusters`	`rdf:_2`	`pc2:ARMINIUS`
`pc2:Clusters`	`rdf:_3`	`pc2:PSC2`

Figure 2.1: Example for RDF Collection Type `rdf:Bag`.

Reification means to express statements about statements. It is an example of the power of RDF which arises from using a single, very generic concept (triple), which can be combined and used in numerous ways. The vocabulary used for reification is `rdf:Statement`, `rdf:subject`, `rdf:predicate` `rdf:object`. By using `rdf:Statement` together with `rdf:type`, an object can be declared to be an RDF statement. The subject, predicate, and object are attached to the object using the respective properties from the reification vocabulary. The statement can be classified further using additional predicates. An example is given in figure 2.2. It contains a reification of the statement ⟨`pc2:User1`, `pc2:hasExecuteRight`, `rdf:PSC2`⟩. The statement is further annotated expressing the fact that `pc2:Admin1` has assigned this right.

Subject	Predicate	Object
`pc2:AccessRight1234`	`rdf:type`	`rdf:Statement`
`pc2:AccessRight1234`	`rdf:subject`	`pc2:User1`
`pc2:AccessRight1234`	`rdf:predicate`	`pc2:hasExecuteRight`
`pc2:AccessRight1234`	`rdf:object`	`pc2:PSC2`
`pc2:AccessRight1234`	`pc2:assignedBy`	`pc2:Admin1`

Figure 2.2: Example for RDF Reification.

There are a few more elements in the RDF vocabulary, which can be found in the RDF primer [80]. They include another way to model lists of objects.

RDF Schema defines further vocabulary for RDF. It is used to model the T-Box layer, i. e. classes, roles, and their relationships. The main elements of RDF Schema are predicates to define hierarchies of classes and properties. Further elements are used to define the domain and range of properties, and special classes are introduced. The RDFS vocabulary uses the namespace `http://www.w3.org/2000/01/rdf-schema#`, abbreviated as `rdfs:`.

A **class hierarchy** is defined using the `rdfs:subClassOf` property. There are no restrictions on the shape of the class hierarchy. Multiple inheritance is allowed. The intended meaning of the `rdfs:subClassOf` predicate is to say that all instances of the sub class are also instances of the super class. The `rdfs:subClassOf` predicate itself is a transitive property.

Property hierarchies are defined in the same way using the `rdfs:subPropertyOf` predicate. Here the meaning is that any triple which asserts some relationship using the sub property also holds for the super property. As `rdfs:subClassOf`, also the `rdfs:subPropertyOf` property is transitive.

There is no special property to assert that two classes or properties are equivalent. However, specifying the sub class or sub property relationship between two elements in both directions has the same effect. Both, the property and the class hierarchy, form a directed graph, which is usually acyclic. If the graph contains cycles, all elements on the cycle are equivalent. The cycle could by replaced by a single element, rendering an acyclic graph.

The `rdfs:domain` and `rdfs:range` properties are used to assert the **domain** and the **range** of another property. To better understand the meaning of these properties consider the example in figure 2.3. It looks like a contradiction, because `pc2:ARMINUS` is of

Subject	Predicate	Object
`pc2:ARMINIUS`	`pc2:hasInterconnect`	`pc2:Infiniband`
`pc2:hasInterconnect`	`rdfs:domain`	`pc2:Cluster`
`pc2:ARMINIUS`	`rdf:type`	`pc2:ComputeResource`

Figure 2.3: Example for `rdfs:domain`.

type `pc2:ComputeResource`, however it should be of type `pc2:Cluster` according to the domain of property `pc2:hasInterconnect`. In RDF, however, no contradiction arises because each individual can be asserted to any number of classes, even if they are not related in the class hierarchy. The semantics of `rdfs:domain` imply that `pc2:ARMINIUS` *is also* of type `pc2:Cluster`. By that, the `rdfs:domain` and `rdfs:range` constructs *cannot* be used to restrict the possible subjects and objects of a property. This is a fundamental difference e. g. to database modeling where integrity constraints can be defined.

Owing to this fact and because RDF/S does not contain any kind of negation, it is **impossible to generate contradictions** in a triple set. It is important to keep this in mind when joining multiple RDF knowledge bases – there will never be any inconsistencies. However, the effect might be surprising e. g. because individuals belong to unexpected classes. A cycle in the class hierarchy implies that all classes on the cycle are equivalent.

In the previous paragraphs, we have used multiple times the expression "intended meaning", which has so far not been explained in detail. This "intended meaning" of RDF triples is defined in the **RDF Semantics** document [55]. It defines which knowledge is implicitly contained in a set of RDF triples, by defining the set of triples that can be derived by a reasoner from a given set of triples.

Formally, the semantics are defined through a *model theory*. A model theory is a formal means to specify in a declarative way which conclusions are valid and which are not. In the RDF Semantics document a **set of rules** is given which renders equivalent semantics compared to the model theory. Thus we can stick to these rules. To generate a complete set of triples which is derivable from a given knowledge base these rules have to be iteratively re-applied to the resulting triple set until no new triples are generated. We call this process *forward chaining* of the rules.

Selected rules are listed in figure 2.4. The rules must be interpreted as follows. If the knowledge base contains some triples that match the pattern given in the column "Precondition", then add the triple given in the column "Generated Triple". The rule name is according to [55].

Additionally, there is a number of RDF triples which are considered to be always true. They are called **axiomatic triples**. Examples are listed in figure 2.5. The axiomatic triples also have to be considered when looking for the preconditions of the rules.

Rule Name	Precondition	Generated Triple
rdfs2	a, `rdfs:domain`, x u, a, v	u, `rdf:type`, x
rdfs3	a, `rdfs:range`, x u, a, v	v, `rdf:type`, x
rdfs5	u, `rdfs:subPropertyOf`, v v, `rdfs:subPropertyOf`, x	u, `rdfs:subPropertyOf`, x
rdfs7	a, `rdfs:subPropertyOf`, b u, a, y	u, b, y
rdfs9	u, `rdfs:subClassOf`, x v, `rdf:type`, u	v, `rdf:type`, x
rdfs11	u, `rdfs:subClassOf`, v v, `rdfs:subClassOf`, x	u, `rdfs:subClassOf`, x

Figure 2.4: RDF Schema Rules.

Subject	Predicate	Object
`rdf:type`	`rdf:type`	`rdf:Property`
`rdf:subject`	`rdf:type`	`rdf:Property`
`rdf:type`	`rdfs:domain`	`rdfs:Resource`

Figure 2.5: Example for Axiomatic Triples.

2.1.4 SPARQL

To take advantage of RDF knowledge bases, we need a query language. During the past years, numerous different languages have been proposed for querying RDF based data like RQL, SeRQL, and RDQL [50]. In order to sort out the various features and standardize them, the W3C started a new language which integrates most of the previously known features. It is called SPARQL [86], which is a recursive acronym standing for "SPARQL Protocol and RDF Query Language". Since April 2006 it has the status of a W3C recommendation.

From the syntax, SPARQL looks a bit like SQL [68]. Figure 2.6 shows a typical query. It asks for objects of type `pc2:ComputeResource` running a Unix operating system on an Intel CPU. The returned resources are ordered by CPU frequency, largest values first. Only the 10 best hits according to this sorting are returned.

The example introduces the main elements of SPARQL queries. The **`PREFIX`** clause is used to enable abbreviation of namespace prefixes. The **`SELECT`** clause defines which of the query's variables are returned to the user. The main part of the query is the **`WHERE`** clause where a pattern is given. This pattern defines an RDF graph written as a set of triples where some URIrefs are replaced with variables. The variables are

```
PREFIX pc2: <http://www.upb.de/pc2/>
SELECT ?res
WHERE {
    ?res rdf:type pc2:ComputeResource .
    ?res pc2:hasOS ?os .
    ?os rdf:type pc2:UNIX .
    ?res pc2:hasCPU ?cpu .
    ?cpu rdf:type pc2:Intel .
    ?cpu pc2:frequency ?freq .
}
ORDER BY ?frequency DESC
LIMIT 10
```

Figure 2.6: Example for a SPARQL Query.

flagged with a question mark.

A SPARQL engine searches for subgraphs of the RDF graph in the underlying knowledge base. These subgraphs must be isomorphic to the query pattern. Variables are to be considered as wildcards which match any URIref or literal. Each matching subgraph creates a query hit which is returned.

The **`ORDER BY`** clause defines how the matches are ordered if there is more than one hit. Finally, the **`LIMIT`** clause states that only a defined number of hits will be returned. It can be combined with an **`OFFSET`** clause to obtain e. g. results number 10 through 20.

SPARQL supports a **`FROM`** clause to specify the RDF knowledge base which is to be queried. In this thesis, this is not necessary as the knowledge base consists of all triples in the p2p network. Thus we do not support it.

Further options of SPARQL include `FILTER`, `OPTIONAL`, and `UNION`. **`FILTER`** filters the results of the query according to the bindings of some variables. In our example, we could specify `FILTER (?freq > 1000)` if the frequency is specified in MHz and we are only interested in resources with a CPU having at least 1 GHz. **`OPTIONAL`** allows to specify that parts of the query graph are optional. Thus a hit is already generated if the query graph without the optional part is matched. Only if the optional part is also found, the variables occurring in it are bound. The **`UNION`** construct specifies that two query patterns are alternatively matched. `UNION`, `FILTER`, `OPTIONAL`, and the basic query patterns can be nested. A more formal description of the SPARQL semantics can be found in [85].

Normally, the output of a query is a list that contains valid bindings for the variable. As an alternative, it is also possible to specify a **construct query**. Such a query returns an RDF graph for every result. The graph can simply be the matched subgraph; however the shape of the result graph can also be specified in the query. This allows to transform RDF graphs using SPARQL.

2.1.5 GLUE Information Model

In this section, we briefly describe the GLUE information model [4] as an example of Grid resource description which is used today. The GLUE model mainly consists of a set of UML class diagrams that define how resources are described. It has no binding to a particular data model. The GLUE information model is widely used e. g. in conjunction with Globus WS-MDS. Additionally, there is a binding to XML Schema that defines how resource descriptions are written as XML using the GLUE model[2].

The information model defines a number of classes, their relationships, and attributes. The main classes are shown in figure 2.7, a further part of the class diagram is shown in figure 2.8 which is related to hosts. We have omitted the attributes for simplicity. Details can be found in [4].

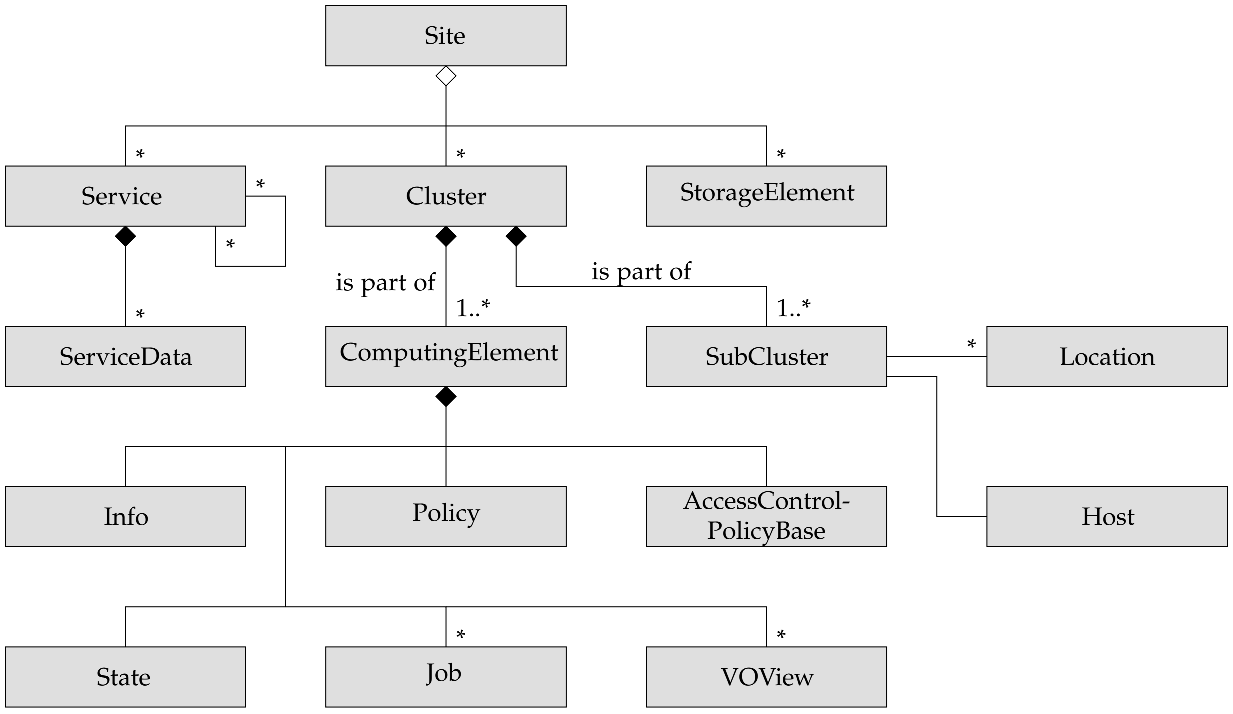


Figure 2.7: GLUE Information Model, Part 1.

The main class is a **Site**, which serves as an anchor for the description of a resource provider in the Grid. Resources are classified as Service, Cluster, or StorageElement. Each provider may have an arbitrary number of each of these resource types. These three main types reflect the origins of Grid computing where HPC resources and storage space where the dominant resource types. They are treated separately from other services. The documentation of the GLUE information model however contains a hint that future releases of the model might hide the traditional resources behind a higher-level service class, thus reflecting the service-oriented view of Grids.

[2]see `http://infnforge.cnaf.infn.it/glueinfomodel/index.php/Mapping/XMLSchema`

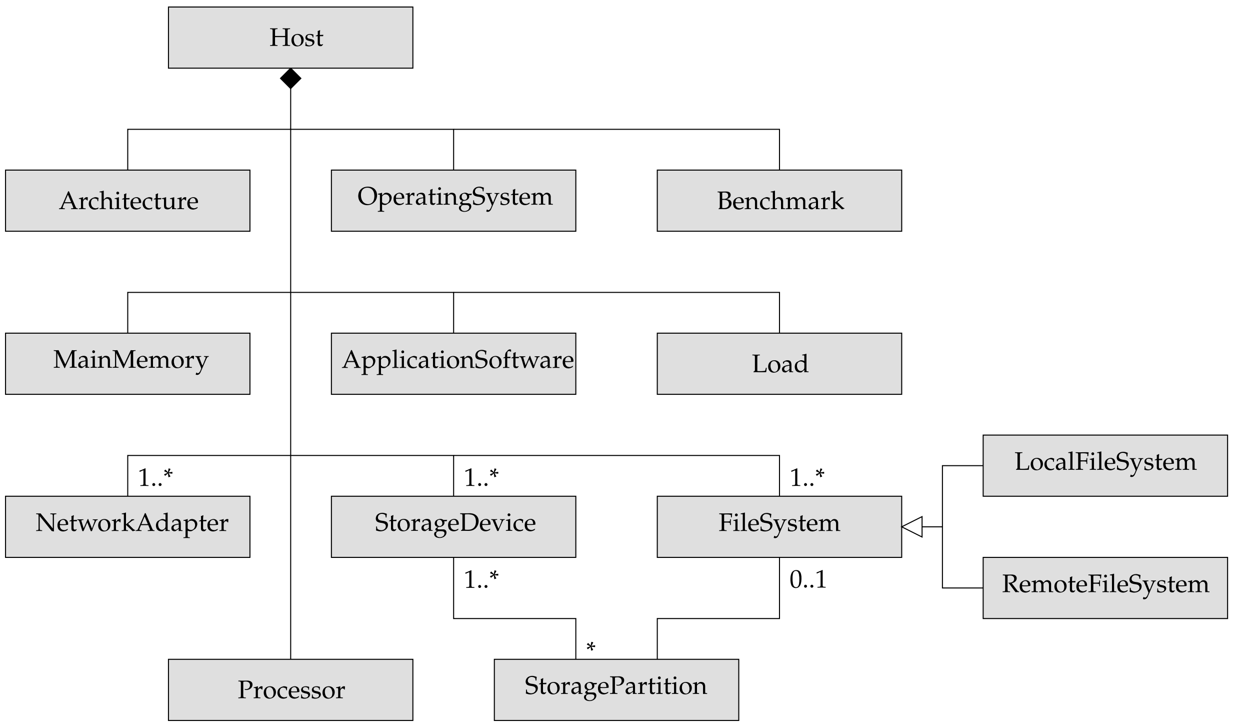


Figure 2.8: GLUE Information Model, Part 2.

The **Service** class is very generic, containing links to a WSDL description and an attribute called “Semantics” which contains a URL where a detailed description of the semantics of the service can be found. Further information can be stored in ServiceData objects, which consist of a key/value pair.

As an example of the remaining two main elements, we focus on the class **Cluster**, leaving out the StorageElement class. The Cluster class in GLUE refers to a set of computational resources that might be heterogeneous. A subset of these resources that is homogeneous is modeled using the **SubCluster** class. The **Host** class encapsulates the information for a single host. The SubCluster consists of a set of hosts. The **Location** class contains information where software is located on this sub cluster.

The **ComputingElement** provides another view for the cluster. It is a logical view, typically a queue of a resource management system which provides access and a job submission interface for the Cluster. There may be multiple CEs for a single cluster, and there might also be more ways to access the cluster bypassing the queues. For each CE, a number of associated classes are used to describe aspects like general information, policies and access control, VO related things, active jobs, and status information.

A more detailed view on the individual hosts is provided in the part shown in figure 2.8. It is used to store monitoring-related information. We do not go into every detail of the proposed classes. As an example, we show the attributes of the **Processor** class in figure 2.9[3].

Property	Type	Mult.	Unit	Description
Vendor	String	1		Name of the processor vendor
Model	String	1		Processor model as defined by the vendor
Version	String	1		Processor version
ClockSpeed	Int32	1	MHz	The clock speed
InstructionSet	String	1		Processor instruction set
OtherProcessor Description	String	1		Other processor description, to be used for extra information not covered by the schema

Figure 2.9: Attributes of the Processor Class in GLUE.

2.2 Peer-2-Peer Networks and DHTs

Peer-2-peer (in short p2p) systems have gained much attention through file sharing systems like Napster and Gnutella [103]. The basic idea is to replace a central server with a fully or nearly fully distributed system where the participants in the network are peers which are clients and servers at the same time. So each participant in the network shares its resources with the other peers, accessing the remote resources and servicing requests from them at the same time. Thus an aggregated power is available outreaching the potential of centralized servers.

P2p computing has many facets. In this thesis, the **resource sharing** aspect and the possibility to build a **highly dynamic** and adaptive environment is of most interest. Also important is the **failure tolerance** of p2p systems, which tend to be stable even in case of multiple simultaneous node failures.

An important idea for p2p networks was described by Kubiatowicz in his article "extracting guarantees from chaos" [73]. The idea is that p2p systems consist of a collection of unreliable components. Upon this unreliable foundation, the p2p layer builds a reliable overlay layer. Thus certain **guarantees emerge** from the collective behavior of the peers that are not present when looking at individual nodes in the network. These emerging properties like fault tolerance are important for our work.

Napster is not a real p2p system, as it relies on a centralized server for indexing purposes. Each peer connected to the system publishes an index of its documents to the central server and uses this server to find documents relevant for a query. Only the documents themselves are transferred in a p2p style without interaction with the server.

[3]Taken from [4]; we omitted cache-related attributes.

Gnutella is one of the first truly decentralized p2p systems. In Gnutella, each peer only knows a limited set of neighbors to which it is connected. This allows for easy maintenance of the network topology, as no complicated state tables have to be maintained. However, the only message routing strategy available is flooding, which means that a query is forwarded to any known host. By this, eventually every node in the network will see a query and potentially respond to it. This implicates a large consumption of network and processing resources. In order to limit congestion and to avoid infinite cycles, messages typically carry a time to live field which limits the number of routing hops. Thus a message is only forwarded to a part of the network. However, it cannot be guaranteed that the message eventually reaches all relevant peers. So no completeness in query evaluation is possible. Even worse, it cannot be guaranteed that the answers are the most relevant for a given query; merely, just a random collection of answers is returned.

In the following, several techniques have been developed to overcome these problems. Some of these techniques use the same basic principles but aim to improve the network topology so that the possibility to get relevant results increases. In the related work chapter 6, we describe some of these techniques in greater detail. An alternative way to overcome the problems of Gnutella-style unstructured networks are **distributed hash tables** (DHTs). In this thesis, we concentrate on DHT-based networks.

The basic idea of DHTs [73] is as follows. Each object that is stored in the network gets a unique identifier from a circular identifier space (e. g. 0 to $2^{128}-1$). The identifier is built by using a hash function with some world-unique identifier of the object. In the case of RDF, we will see that hashing the URIrefs and literals is a good way to produce the ID of an object. After that, each node in the networks is responsible to store objects from a well-defined range of the identifier space. To find a certain object, its ID is determined and the node is searched which is responsible to store this ID. To store a new object in the network, also the responsible node is contacted and either the object itself (if it is small enough) or an index pointing to the node which originally holds the object is stored.

The method is visualized in figure 2.10. The circular identifier space allows for 2^{128} different IDs. Each of the 5 nodes in the network is assigned a subset of the identifier space. According to the DHT algorithm in use, these subsets are not of equal size. In the figure, they are all contiguous. However, this is not always the case.

Now consider 6 objects to be stored in the network. Each of the objects has a key which is used for storage and retrieval. The key is hashed to determine the node which is responsible for this key. In our case, the hash value of the key for object 1 is within the subset of the identifier space for node 5. Thus, either the object itself or an index pointing to the object is stored on node 5. In this way, the 6 objects are stored on well-defined nodes and can later be retrieved by contacting these nodes.

From the previous discussion it follows that the main operation a DHT has to provide is to *determine the node which is responsible for a given key*. This operation is called the **lookup operation**. It is typically realized as a routing mechanism which forwards a message to the relevant node. According to the message type, this node can then an-

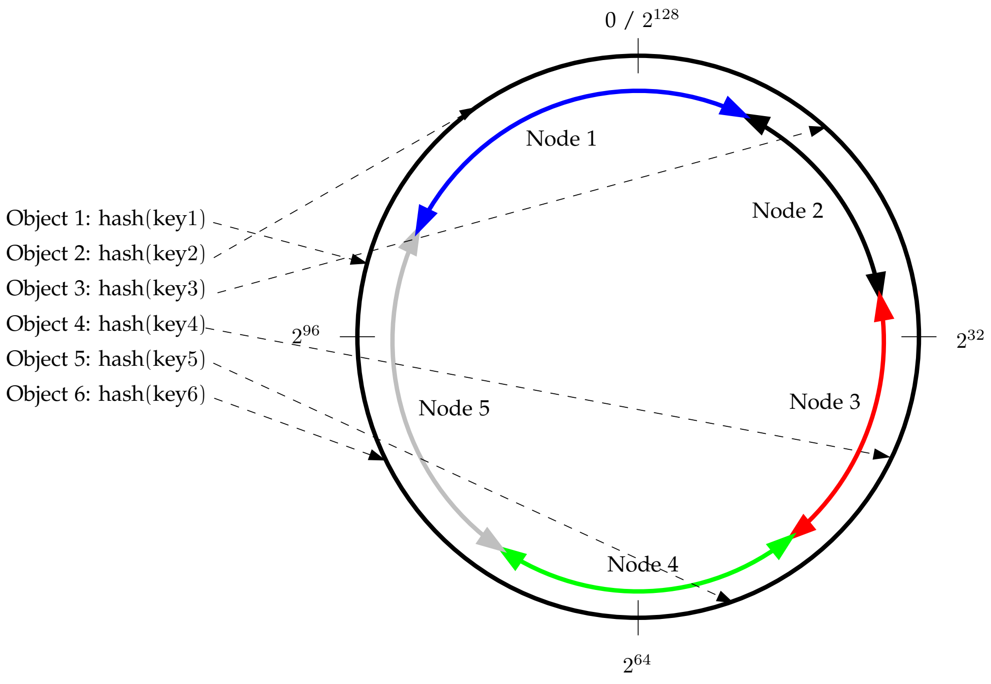


Figure 2.10: Distributed Hash Tables.

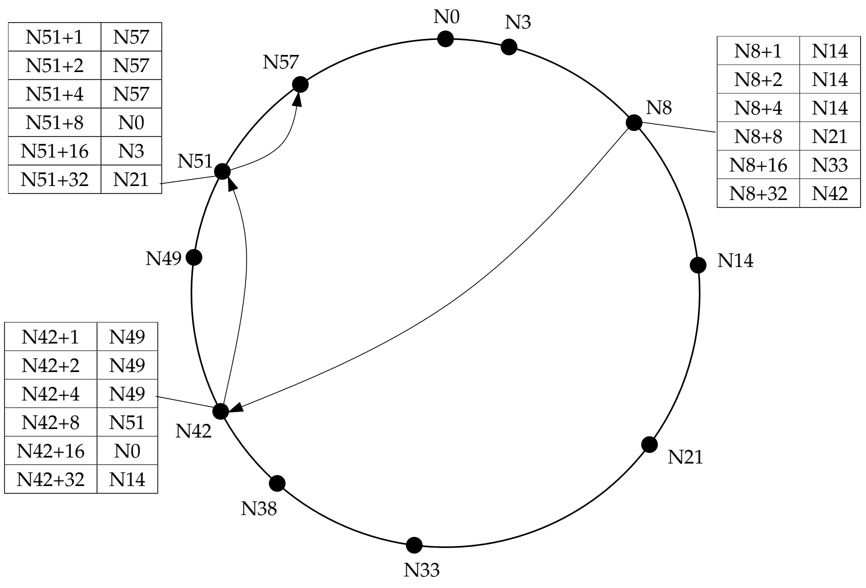
swer e. g. by storing a new object, or by supplying a previously stored object or index information pointing to the object.

There are several algorithms which implement DHTs. They all have some common design goals. They aim to be scalable and to support highly dynamic and unreliable networks. Scalability means that the number of routing hops needed for the lookup operation should increase modestly with a growing network size, and that the storage and maintenance effort for status information is limited. Supporting dynamics means that the algorithm should cope with high rates of arriving and failing nodes, and with failing network links.

In the following subsections, we describe two representative implementations of DHTs: Chord and Pastry. Chord was one of the first DHTs and is still very popular. As it's algorithm is relatively simple, it is often used to explain DHTs. Pastry is the network we are using in this thesis. It is slightly more complex than Chord, however it has additional interesting features like locality. For Pastry, there also exists a robust implementation called FreePastry [96] which is freely available. We use it for the prototype of the system described in chapter 5.

2.2.1 Chord

In Chord, the nodes form a ring. Each node has a node identifier in the range 0 to $2^m - 1$, where m is a configuration parameter. The nodes maintain links to their successors and predecessors in the ring. Each element stored in the network is assigned to the node with the next ID on the ring. Routing would be already possible with only the successor link, by forwarding a message around the ring until it has reached its final destination. However, the number of hops needed for routing would be on average half the number of nodes in the network. Chord uses so-called **finger tables** to enable faster routing. The idea is to maintain links to the closest nodes on the circle which are at least in distance $2^0, 2^1, \ldots, 2^{m-1}$ on the ID space. Figure 2.11 shows a visualization of the Chord finger table[4]. In the example, $m = 6$ and thus the ID space is $0 \ldots 63$.



N51+1	N57
N51+2	N57
N51+4	N57
N51+8	N0
N51+16	N3
N51+32	N21

N8+1	N14
N8+2	N14
N8+4	N14
N8+8	N21
N8+16	N33
N8+32	N42

N42+1	N49
N42+2	N49
N42+4	N49
N42+8	N51
N42+16	N0
N42+32	N14

Figure 2.11: Chord Finger Table Lookup.

The figure shows the finger tables for nodes N8, N42, and N51. Consider the table for node N8. The first three entries in this finger table would be nodes N9, N10, and N12. As neither of these nodes exists, the fingers point to the next existing node, which is N14. The following entry is $\mathrm{N}8{+}8 = \mathrm{N}16$. As this node also does not exist, the finger points to node N21.

Now let's look at the **routing mechanism**. Assume node N8 wants to send a message to the node that is responsible for key 55. Routing is done clockwise. Each node forwards the message to one of the nodes in its finger table. The target node's ID should be as close as possible to the target ID, however it must not be beyond it. Thus N8 forwards the message to N42, which is the largest possible step. N42 forwards the message to N51, as the next entry in the routing table, N0, is already beyond the target ID. N51 detects that the first entry in the finger table, N57, is already beyond the target ID 55. Thus this node must be the target node, as there can be no other node between N51 and N57. Thus the message has reached its destination in three routing steps, while the whole network has eleven nodes.

[4]example adapted from [104]

In general, the routing needs $O(\log N)$ steps with high probability, where N is the total number of nodes in the network. Informally, this is because each routing step in average bridges half of the remaining distance to the target node. As the actual layout and positioning of the nodes is random, no absolute guarantee can be made – thus the clause *with high probability*. For a more formal proof of the routing costs, see [104].

An important part of each DHT implementation is the mechanism to update the routing table in case of **node arrivals and node departures**. Owing to arbitrary failures, node departures take place in general without notice. This means that the DHT cannot rely upon a well-defined "departure protocol" which is initiated by a node planning to leave the network. Instead, other nodes just detect that the node does not respond any more. They have to update their state tables accordingly.

When a new node joins a Chord ring, it chooses a new ID K at random which it is going to use in the network. As a prerequisite to join a Chord ring, the node has to know at least one other node which is already part of the ring. To determine its own position in the ring, it uses the known node to route a message to the node currently responsible for the chosen ID K. This node is by definition the node which immediately follows K in the ring. Thus it is the successor of the new node.

A **stabilization protocol** is run which regularly updates the successor and predecessor pointer of a node. During this stabilization, each node contacts its successor, which can then update its own predecessor pointer. The nodes also compare their successor and predecessor pointers and correct them in case something is wrong. Additionally, a node periodically tests whether its predecessor has failed. If this is the case, the pointer is cleared and will be refilled through the stabilization protocol.

Routing is even possible without a finger table or with a wrong finger table. As long as there is at least a valid successor pointer in each node's state, a message will be routed clockwise around the ring until it reaches its destination. Thus routing can be used to initialize and update the finger table entries. This is done periodically by each node. For each entry of the finger table, a message is routed to the target ID of this entry. It will reach the successor node of the ID on the ring, which will then be stored in the finger table.

In case a node dies without notice, each node additionally stores n successors of its direct successor. The information is updated periodically by querying the successor table of the direct successor. When the direct successor dies, the following entry in the successor table is used instead and the successor table is refreshed accordingly. Through the stabilization protocol, also the predecessor links will be adjusted.

2.2.2 Pastry

Another implementation of the DHT scheme is Pastry [97]. In Pastry, each node has a randomly chosen ID from the circular ID space $0, \dots, 2^{128} - 1$. The IDs are written as numbers to the base 2^b, where b is a configuration parameter. Routing is based on prefixes. In each routing step, Pastry aims to forward a message to a node whose common prefix with the target ID of the message is larger than with the current node's ID. For this purpose, each Pastry node stores pointers to nodes with various prefixes in the routing table. In addition, Pastry maintains a set of nodes which are the closest nodes on the identifier space, called the leaf set, and a neighborhood set of nodes which are close with respect to network locality. In the following, we concentrate on the basic routing mechanism and omit the neighborhood set and locality-related issues in Pastry. See [97] for a further discussion on locality in Pastry.

We start with the **routing table**. Figure 2.12 contains an example taken from [97] for Pastry node 10233102. In this example b is 2, so that all IDs are written as base 4 numbers. Furthermore, the identifier space in the example is restricted to 2^{16} bits for clarity.

10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

(a) Leaf Set.

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

(b) Routing Table.

Figure 2.12: Node State for Node ID = 10233102, see [97].

The i-th row[5] of the routing table always contains pointers to nodes which share the first i digits with the current node's ID. In column j, there is a pointer to a node which has a j on the j-th digit. There is no pointer on cell (i, j) if the i-th digit of the node's ID is j. If there is no node with the mentioned restrictions, the corresponding cell in the table is empty. In each node ID of the example, the common prefix and the digit corresponding to the column number are marked by dashes. Look e.g. at the cell in light gray, which is row 3 and column 1. It must contain an ID which shares the first three digits with the current node ID (102), and which has a 1 at digit 3. The rest is arbitrary. In this case, node ID 10211302 fulfills these restrictions.

When routing a message to a destination ID, Pastry always aims to maximize the common prefix of the next hop's ID and the target ID. If the routing table is completely filled, in each step at least one more digit will be identical. However, if some cells are

[5]Row and column numbers start with 0.

empty, it means that no such node is known. In this case the message is routed to the node whose ID is numerically closest to the target ID.

When the message approaches its final destination, the **leaf set** comes into play. This table, shown in figure 2.12(a), contains l entries: $l/2$ nodes following the current node on the identifier space, and $l/2$ nodes which precede it on the ID space. As soon as the target ID of a message is within the range of the leaf set, it can be delivered immediately to its target node. With this routing procedure, it can be shown that a message reaches its destination with high probability within $O(\log_{2^b} N)$ steps [97].

When a new node joins a Pastry network, it acts similarly to a node joining a Chord ring: it chooses a random ID and uses an already known Pastry node to forward a message to the newly chosen ID. During the routing of this message, each node sends its routing tables to the new node. The joining node uses this information to initialize its routing tables.

In case of failing nodes, other Pastry nodes have to repair their state tables. As routing is already possible with only the leaf set, the network stays operational unless all nodes on one half of the leaf set of some node fail simultaneously. This can only be the case when $|L|/2$ nodes with adjacent node IDs fail at the same time. As node IDs are randomly chosen, these nodes are dispersed both in geography and network location. Thus such a case is unlikely to happen even for small values of l.

Both the leaf set and the routing table are **repaired lazily**. This means, that the entries are periodically checked and repaired as soon as a failure is detected. The entries are repaired by contacting nodes which might know valid entries. For the leaf set, these are the outermost life entries in the leaf set. For the routing table, these are the other entries in the same row, and after that the entries in the next rows of the routing table.

2.2.3 Common API for DHTs

As explained in this section, a DHT is a pattern which can be implemented by various algorithms. From the application's point of view, the underlying DHT is a black box. The black box provides certain functionalities. The application is only interested in the properties (like complexity of the routing), but not in the details of how it is done. With such a viewpoint, it is possible to exchange the DHT implementation without affecting the application. For this to be possible, the application programming interface (API) provided by the DHT must be standardized. Currently, no official standard exists. However, there is a proposal "Common API for Structured Peer-to-Peer Overlays" described in [33], which formulates the basic elements of such an API. In our prototype, we stick to this API as far as possible to maximize the independence from the underlying DHT.

Figure 2.13 shows the layered architecture of a DHT based application and the interface which is defined by the common API. In the following, we explain the methods of this API due to [33]. However, we restrict the description to those methods which are

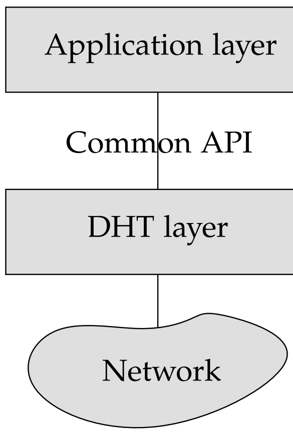


Figure 2.13: Common API for Structured Peer-to-Peer Overlays.

actually used by our prototype.

The API uses some specific terminology. A node of the network which is responsible for a given key, is called the **root node** for this key. Furthermore, the API assumes that the p2p network supports a replication mechanism. To implement this, each key additionally has a set of `r`**-roots**. The `r`-root for key `k` is the node which is responsible to hold the `r`-th replica for the object with key `k`. The `r`-th replica is also said to be of **rank** `r`. The root node is also called the **rank 0 node**. Some implementations may not support replicas or only up to certain rank. If a network implements replicas, the routing algorithm must assure that a message is delivered to a replica node in case the root node fails.

Furthermore, the API assumes that each node has a **neighborhood**. This is a set of nodes which are directly known to the node, like the set of successors and the predecessor in Chord or the leaf set in Pastry.

The main functions are `route` and `deliver`. Route is used to send a message to a node which is responsible for a given key, and deliver is a call-back invoked at the target node when the message has arrived. The method `forward` is also a call-back invoked on each node on the routing path.

VOID `route`(KEY *key*, MESSAGE *msg*, NODEHANDLE *hint*)
Routes the message *msg* to the node which is responsible for *key*. The call is asynchronous, no acknowledgement is sent, and no quality of service is guaranteed. The optional parameter *hint* may contain the address of a node which will be used as first routing hop. If the hint is good (e. g. it contains the target node's address), the message can be delivered with only one hop. In the contrary, a bad hint may introduce a further unnecessary routing step.

VOID `deliver`(KEY *key*, MESSAGE *msg*)
This method is a call-back which has to be provided by the application. The p2p routing mechanism calls this method upon arrival of a new message for this node.

VOID `forward`(KEY *key*, MESSAGE *msg*, NODEHANDLE *nextHopNode*)
This is a call-back method which is called at each node on the routing path of message *msg*, including the source node and the target node. It is called just before the message is forwarded to the next hop, which has already been determined as *nextHopNode*. Within the method, each parameter may be modified. When *key* or *nextHopNode* are modified, the routing behavior is altered.

In order to detect changes in the network structure, there exists a further call-back method `update`:

VOID `update`(NODEHANDLE *handle*, BOOL *joined*)
This is a call-back method that is invoked by the routing layer when the local neighborhood (with respect to ID locality) of the current node has changed, either because a new node has joined the network (*joined* equals TRUE) or has left the network. The address of the regarding node is given in *handle*.

The function `replicaSet` determines the nodes where replicas are stored, and the `range` function determines the ID ranges for which a node is responsible.

NODEHANDLE [] `replicaSet`(KEY *k*, INT *maxRank*)
This method returns the set of all r-roots for key k where $r \leqslant$ maxRank. If there are fewer replicas supported than maxRank, only the supported number of r-roots is returned.

BOOL `range`(NODEHANDLE *handle*, INT *r*, KEY *leftKey*, KEY *rightKey*)
This method returns the range(s) of keys for which the node *handle* is currently r-root. The method can only be called using the handle of the local node or one of the nodes in the neighborhood of the current node. As there might be multiple unconnected ranges, the method must be called multiple times to determine every range. In each call, the range which is the next range clockwise starting with *leftKey* is returned. The method returns TRUE if a valid range has been found.

2.3 Subgraph Isomorphism

We have seen in the SPARQL section 2.1.4 that subgraph isomorphism is the core of querying RDF. Thus we now look at the classical algorithm for subgraph isomorphism from Ullmann [109]. Although it is not directly applicable for RDF queries, it provides the general idea.

Consider two graphs $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$, where $|V_a| = |V_b|$ and $|E_a| = |E_b|$. The **graph isomorphism** problem is the problem of finding a bijection between V_a and V_b, such that every two vertices which are adjacent in G_a are mapped to two vertices which are adjacent in G_b. If such a mapping exists, we say the two graphs are isomorph.

The **graph subgraph isomorphism** problem is the problem of finding a subgraph of a graph G_b which is isomorph to a given graph G_a. We also call G_a the **query graph** and G_b the **model graph** because the pattern G_a is searched within G_b. Ullmann's algorithm for finding graph subgraph isomorphisms is the most common general-purpose algorithm for this problem. In the following, we describe this algorithm.

Ullmann's algorithm works with an adjacency-matrix representation of graphs. Let $p_a := |V_a|$ and $p_b := |V_b|$. Let $A = [a_{ij}]$ be the $p_a \times p_a$ adjacency matrix for G_a, and $B = [b_{ij}]$ analogous. The algorithm originally requires the graphs to be undirected, and thus A and B to be symmetric. However, we describe here a slight modification which can handle directed graphs. Thus A and B can be any square matrix. As the graphs are not weighted, we assume that $a_{ij}, b_{ij} \in \{0, 1\}$. For convenience, we define $P_a := \{1, \ldots, p_a\}$ and $P_b := \{1, \ldots, p_b\}$.

A mapping between A and B is described by a matrix $M = [m_{ij}]$ with p_a rows and p_b columns. Note that $p_b \geqslant p_a$ as else there could not be a subgraph of G_b isomorph to G_a. Each row of M must contain exactly one 1, and each column of M must contain at most one 1. If $m_{ij} = 1$ this means that the i-th vertex of G_a is mapped to the j-th vertex of G_b. A condition to test whether a given matrix M is indeed an isomorphism between A and some subgraph of B is the following. Compute $C = (M(MB)^T)^T$. C is the adjacency matrix of the subgraph of B containing only the vertices of B used in the mapping, and re-ordered according to the mapping. For each edge in A there has to be an edge in C:

$$\forall\, i, j \in P_a \times P_a : [(a_{ij} = 1) \Rightarrow (c_{ij} = 1)] \tag{2.1}$$

The general idea of Ullmann's algorithm is to start with a matrix M^0 with $m^0_{ij} = 1$. Each element of M can be thought of as a candidate mapping. If $m^0_{ij} = 1$, we have to consider mappings where i is mapped to j. In the first step, the algorithm removes obvious cases where no mapping is feasible. An obvious case is where the in-degree or out-degree of vertex j in G_b is smaller than the in-degree or out-degree of vertex i in G_a, respectively:

$$m^0_{ij} = \begin{cases} 0 & : \left(\sum_{k=1}^{p_a} a_{ik} \geqslant \sum_{k=1}^{p_b} b_{jk}\right) \vee \left(\sum_{k=1}^{p_a} a_{ki} \geqslant \sum_{k=1}^{p_b} b_{kj}\right) \\ 1 & : \text{else} \end{cases} \tag{2.2}$$

Starting with M^0, the algorithm generates each possible matrix M with $m_{ij} \leqslant m^0_{ij}$ with exactly one 1 in each row and at most one 1 in each column. During the generation of the matrices, a so-called *refinement procedure* is called which removes elements of M which cannot be valid mappings. To understand this refinement, consider an element $m_{ij} = 1$, which means that vertex i of G_a is mapped to vertex j of G_b. Thus we look at each edge that originates from vertex i in G_a. These edges are described by the elements of the i-th row of A which are 1. For each of these edges there has to be a corresponding edge in B:

$$\forall k \in P_a : [(a_{ik} = 1) \Rightarrow \exists l \in P_b : (m_{kl} = 1 \wedge b_{jl} = 1)] \tag{2.3}$$

This can be explained as follows. If $a_{ik} = 1$, then there is an edge from vertex i to vertex k in G_a. There has to be a possible target vertex for k in G_b. We call it l, thus m_{kl} has to be 1. Additionally, there has to be an edge in G_b from vertex j has to vertex l, i. e. $b_{jl} = 1$.

Thus, the refinement procedure checks (2.3) for each $i \in P_a, j \in P_b$ where $m_{ij} = 1$. If the condition is not fulfilled, m_{ij} is set to 0. When resetting an m_{ij} to zero, (2.3) might get violated for another pair i', j'. Thus the procedures loops until no m_{ij} violates one of the conditions.

Consider a matrix M with at most one 1 per column and exactly one 1 per row. Condition (2.3) is obviously necessary for M to be a graph subgraph isomorphism. We now show that it is also sufficient. Consider such a matrix M and i, j with $a_{ij} = 1$. Looking at (2.1) we have to show that $c_{ij} = 1$. First, we know that there exist unique x, y such that $m_{ix} = 1$ and $m_{jy} = 1$. x and y are the images of i and j under M. For m_{ix}, (2.3) must hold:

$$\forall k \in P_a : [(a_{ik} = 1) \Rightarrow \exists l \in P_b : (m_{kl} = 1 \wedge b_{xl} = 1)] \tag{2.4}$$

As $a_{ij} = 1$, (2.4) must hold for $k = j$:

$$\exists l \in P_b : (m_{jl} = 1 \wedge b_{xl} = 1) \tag{2.5}$$

There is only one $m_{jl} = 1$, as each column has at most one 1. Thus l must be y, because $m_{jy} = 1$:

$$m_{jy} = 1 \wedge b_{xy} = 1 \tag{2.6}$$

Now we can calculate c_{ij}. Recall that $C = (M(MB)^T)^T$.

$$c_{ij} = \sum_{k \in P_b} m_{jk} \sum_{l \in P_b} m_{il} b_{lk} = m_{jy} \sum_{l \in P_b} m_{il} b_{ly} = m_{jy} m_{ix} b_{xy} = 1 \tag{2.7}$$

This shows that (2.3) is also a sufficient condition for M to be a graph subgraph isomorphism. The complete algorithm works as follows. Note that we have written the algorithm as an recursive function in order to avoid the gotos used in the original paper. The algorithm uses two functions `buildM0` and `refine` which work as explained above. It is shown in figure 2.14 and figure 2.15.

```
input  : A: Adjacency matrix for $G_a$ (Query Graph)
input  : B: Adjacency matrix for $G_b$ (Model Graph)
$M^0 \leftarrow$ buildM0() ;
foreach $k \leftarrow 1, \dots, p_b$ do
    $F_k \leftarrow 0$ ;
end
ullmann_recursive($A, B, M^0, 1, F$) ;
```

Figure 2.14: Function ullmann.

2.4 Bloom Filters

Bloom filters [17] are an efficient way to represent large sets as small bit-sets. They are used e. g. in the context of distributed databases.

The basic idea is to encode a set $\mathcal{S}$ as a bit-set of arbitrary size by using multiple different hash functions. Assume we use a bit array of size m, and assume further we use k hash functions. We assume that the domain of these hash functions corresponds to the length of our bit array, i. e. $h_i \mapsto \{1, \dots, m\}$. To insert a new element x into the Bloom filter, we apply k hash functions $h_1, \dots, h_k$ to x and set the corresponding bits in the bit array, see figure 2.16.

To test whether a given item y is a member of the set, y is also hashed via the h_i functions, and the resulting bits are tested. If all k bits are set, then y is assumed to be a member of the set. By this it is clear that every item y which is indeed a member of the set $\mathcal{S}$ will be recognized by the Bloom filter, however there might be **false positives**, which means that an item y' which is not in $\mathcal{S}$ might erroneously be claimed to be in the set through the Bloom filter, as illustrated in figure 2.17. These false positives occur if all bit positions which are referenced from $h_1(y')$ through $h_k(y')$ are already set due to other elements stored in the filter. Note however that these k bits might as well have been set due to k different items stored in the filter. In contrast, **false negatives** are not possible. Every element stored in the filter is reliably detected.

By construction, Bloom filters support the operations *insert element* and *membership test* for a given element. It is not possible to reconstruct the set $\mathcal{S}$ from the filter. As explained, the membership test may produce false positives. It is an important issue to calculate the probability p for false positives. This probability depends on the parameters k (number of hash functions), m (size of the bit array), and n (number of elements inserted into the Bloom filter). Intuitively, it is clear that the false positive rate will be larger with a raising number of elements inserted into the Bloom filter when fixing m and k. It is also obvious that a larger value for m will lower this rate when fixing n and k. However, the effect of varying k is not that obvious. A larger value for k will on the one hand require more bits to be set for a positive membership test, thereby lowering the risk of false positives. On the other hand, a larger k will also result in more bits set to one in the filter, thus rising the risk. Thus we have to look a

```
input : A: Adjacency matrix for $G_a$ (Query Graph)
input : B: Adjacency matrix for $G_b$ (Model Graph)
input : M: Mapping constructed so far resp. feasible mappings
input : d: Current vertex in $G_a$ to be mapped
input : F: Lists which columns in M are used
foreach $k \leftarrow 1, \ldots, p_b$ do
    // loop over each vertex in $G_b$
    if $m_{dk} = 1$ and $F_k = 0$ then
        // no mapping to vertex k so far, and the mapping is feasible
        $M' \leftarrow M$ ;                              // save the current state
        $F_k \leftarrow 1$ ;                // record that vertex k of $G_a$ is mapped
        foreach $j \leftarrow 1, \ldots, p_b$ do
            // we selected vertex k, so all other mappings from d are
               cleaned
            if $j \neq k$ then
                $m_{dj} \leftarrow 0$ ;
            end
        end
        if refine(M) = SUCCESS then
            // the refinement revealed no contradictions
            if $d = p_a$ then
                // we have successfully mapped all vertices
                M is an isomorphism ;
            else
                // go on with the next vertex of $G_a$
                ullmann_recursive(A, B, M, d + 1, F) ;
            end
        end
        // reset the state and proceed to next possible mapping for
           vertex d in $G_a$
        $M \leftarrow M'$ ;
        $F_k \leftarrow 0$ ;
    end
end
```

Figure 2.15: Function ullmann_recursive.

little closer.

We assume that each hash function yields each element in $\{1, \ldots, m\}$ with equal probability. The probability that a single bit is set to one by hash function h_i is thus $1/m$. Thus the probability that an individual bit is *not* set through h_i is the inverse

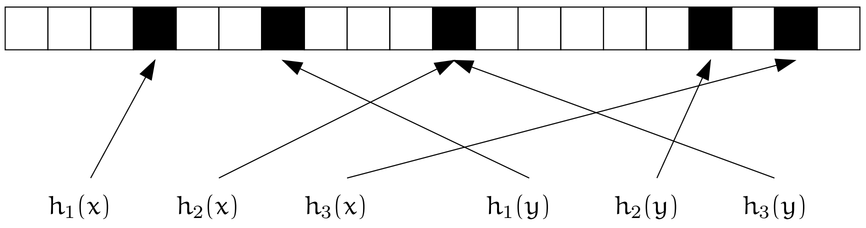


Figure 2.16: Insertion of Elements into a Bloom Filter.

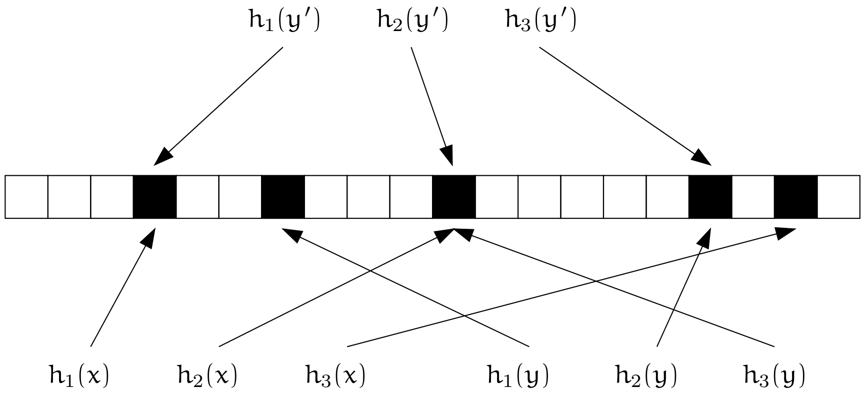


Figure 2.17: False Positive Test with a Bloom Filter.

probability:

$$1 - \frac{1}{m} \tag{2.8}$$

As we have k hash functions, the probability that a single bit is *not* set after inserting an item to the filter is

$$\left(1 - \frac{1}{m}\right)^k \tag{2.9}$$

After inserting n elements into the Bloom filter, each individual bit might have been set by each of the k hash functions for each of the n elements. This results in the following probability for a single bit to be still zero:

$$\left(1 - \frac{1}{m}\right)^{kn} \tag{2.10}$$

The inverse probability is the chance for a bit to be set after n insertions:

$$1 - \left(1 - \frac{1}{m}\right)^{kn} \tag{2.11}$$

A false positive occurs if each of the k bits corresponding to the hash function values is set. This probability is:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{2.12}$$

We are now interested to find an approximation that expresses the probability p of false positives via the ratio n/m. We start with the following equation: $\ln(1+x)$ can be written as follows

$$\ln(1+x) = \sum_{r=1}^{\infty} (-1)^{r+1} \frac{x^r}{r} \tag{2.13}$$

setting $x = -1/m$ yields

$$\begin{aligned}
\ln(1 - \frac{1}{m}) &= \sum_{r=1}^{\infty} (-1)^{r+1} \frac{(-1)^r}{rm^r} = -\sum_{r=1}^{\infty} \frac{1}{rm^r} & (2.14)\\
\Rightarrow kn \ln(1 - \frac{1}{m}) &= -\sum_{r=1}^{\infty} \frac{kn}{rm^r} & (2.15)\\
&= -\left[\frac{kn}{m} + \frac{kn}{2m^2} + \frac{kn}{3m^3} + \cdots\right] & (2.16)
\end{aligned}$$

if we approximate $kn \ln(1-1/m)$ with $-kn/m$ (with an error smaller than $O(kn/m^2)$) we get:

$$\frac{-kn}{m} \approx kn \ln(1 - \frac{1}{m}) \tag{2.17}$$

The parameters k, n, and m are all positive. Thus $-kn/m$ is negative, and thus we can do the following approximation:

$$\begin{aligned}
e^{\frac{-kn}{m}} &\approx e^{kn \ln(1 - \frac{1}{m})} & (2.18)\\
&= \left(1 - \frac{1}{m}\right)^{kn} & (2.19)
\end{aligned}$$

Thus we can approximate p as follows:

$$\left(1 - e^{-k\frac{n}{m}}\right)^{k} \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} = p \tag{2.20}$$

This approximation is interesting because it expresses p in terms of n/m: the ratio of inserted elements to the size of the filter. In order to find good values for k, n, and m, we first couple the values of n and m. The smaller n/m is, the more bits are available per element to store, and the smaller gets p. For a fixed value of m/n, we want to get the optimal value for k. As k is the number of applied hash functions, it has to be integer and should not be too large because of the computational overhead. The table in figure 2.18 yields good values for k for various values of m/n. Please note that due to the above approximation, these values are only valid for sufficiently large values of m and n. If you e.g. look at value $m/n = 3$, with values $m = 3$ and $n = 1$, you get for $k = 2$ the probability 23.7% with the approximation. However, calculating the exact value yields 30.9%. For larger values of m and n, e.g. $m = 30$ and $n = 10$, the exact value 24.2% is much closer to the approximation. For $m = 300$ and $n = 10$ we get the exact value 23.7%.

m/n	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
2	**39.3%**	40.0%	46.9%	55.9%	65.2%	73.6%	80.7%
3	28.3%	**23.7%**	25.3%	29.4%	35.1%	41.8%	49.0%
4	22.1%	15.5%	**14.7%**	16.0%	18.5%	22.0%	26.3%
5	18.1%	10.9%	**9.2%**	9.2%	10.1%	11.6%	13.8%
6	15.4%	8.0%	6.1%	**5.6%**	5.8%	6.4%	7.3%
7	13.3%	6.2%	4.2%	3.6%	**3.5%**	3.6%	4.0%
8	11.8%	4.9%	3.1%	2.4%	**2.2%**	2.2%	2.3%
9	10.5%	4.0%	2.3%	1.7%	1.4%	**1.3%**	1.3%
10	9.5%	3.3%	1.7%	1.2%	0.9%	**0.8%**	0.8%

Figure 2.18: Approximate Values for p for Various Bloom Parameters.

2.5 Summary

This section has given an overview of the a priori known techniques and algorithms building the foundation for our system. We have described approaches to modeling, starting with a description of what an ontology is, describing the Semantic Web and the resource description framework developed within the Semantic Web, and its query language SPARQL. We have further described the GLUE information model that is commonly used for resource description in the Grid.

The second part of the chapter has described the p2p technologies that we use as underlying routing infrastructure. After that, we have described two specific issues important for our work. As RDF querying bases on subgraph isomorphism, we have described the most commonly used algorithm to solve this problem, and we have described Bloom filters that we use to reduce network load during query processing.

3 Resource Matching

This sections focusses on the main application of our system: Resource Matching within Grids. First, we give an overview of the matching process in the context of Grids. We then consider the problem how resources and queries for resources can be described. We first look at standards from the Grid to see how resource description is done today. We further explain why we consider these methods insufficient for large world-scale Grid environments. We point out issues where traditional resource description benefits from more flexible approaches like RDF and RDF Schema. Further, we look at the question how we can organize the resource description data, and conclude with a summary.

3.1 Resource Matching in the Grid

The typical life cycle of a job in the Grid is visualized in figure 3.1. In the first step, the **providers describe their resources**. This can be done either manually, or by using some automated process which reads and converts system information. The resource information consists both of **static and dynamic data**. The static data contains those aspects that change seldom, typically with human intervention. Changes might occur when a new system is integrated in the Grid, new software is installed, or components are exchanged (both hardware and software). Dynamic aspects are those that change permanently like current utilization or queue-length, available network bandwidth, information about faults, etc. However, the boundary between static and dynamic data is not sharp.

In order to be able to automatically evaluate the resource descriptions, a **formalism** must be used that is machine-readable. The expressivity of this formalism determines how flexible and rich the resource descriptions can be. This is especially true for what we call **background knowledge**, which is knowledge *about the resources*. Examples for background knowledge include information about compatibility like "an Itanium processor is backward compatible to a Pentium processor", dependencies like "to run the program X, you need a License of type Y", and generic information about resource types like "an Itanium processor is a 64 Bit processor".

In the following step two, the resource information from the providers is **stored in a database**. This can be a central database, a hierarchical system like Globus MDS [32], or a distributed system. For large scale Grid systems, the scalability of this system is crucial. Another important feature is the time needed to propagate changes in the original data sets at the providers.

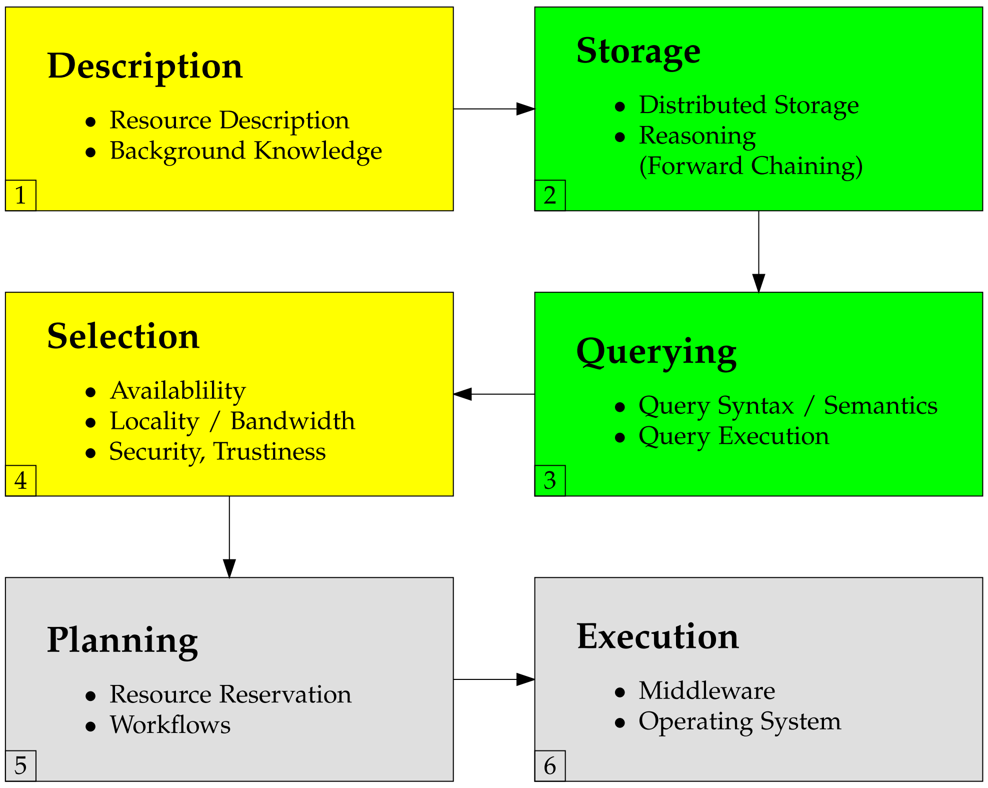


Figure 3.1: Job Life Cycle in the Grid.

If the formalism supports inference, and there is background knowledge stored in the system, the **reasoning** has to be performed, either a priori by generating and storing inferred knowledge (forward-chaining) or during the query evaluation.

In the third step, the user **formulates a query** and sends it to the resource matcher. She uses a query syntax appropriate for the system and expresses all requirements for the job within the query. The query might either be written down directly by the user, or generated by some other software component. Consider as an example a fault tolerant resource management system which detects a resource outage and searches for spare resources. In this case it builds a resource query which contains every detail about the searched spare resource.

The **query semantics** determine which kind of queries are possible. It defines valid constructs like disjunction, conjunction and negation, types of filters, etc. It also includes the possibility to sort the query results and to limit the results to the top k matches.

The systems's way to perform the **query execution** determines which knowledge the query is evaluated against and the processing time needed until the results are returned. However, also the reasoning plays an important role in this step. If the query semantics include some reasoning with the background knowledge, and the inference has not been done during step two above, it has to be done during query execution.

Step four is the **selection of resources**. In this step the user selects from the possibly large set of resources returned in the querying step those that are interesting for her. There is a broad number of selection criteria. Users are interested first of all in *available* resources. Either the user has a fixed time for execution in mind and has to check the availability at this point, or the user might be interested that the job is executed as early as possible and thus looks at parameters like current utilization or queue length. Other criteria include the location of the resources, pricing issues, or trustiness of the provider and security considerations. Location might be geographic in case the user needs physical access to the resource, or network location to get a high bandwidth or low latency connection to the resource. All these aspects must be included in the resource description so that the user knows them and is able to take them into consideration.

After the selection, the execution of the job **can be planned**. This includes the reservation of the selected resources [63], or the planning of a whole workflow where aspects like the timing of the workflow elements, the overall execution time, network links between the resources, etc., play a role. In the final step, the **job is executed** by submitting it through the Grid middleware to the resources. There, the local resource management system and operating system take over control.

Steps five and six are not within the focus of this thesis. The main issues we are concerned with are step one to four, with a special emphasis on step two and three. Step four, the selection of resources, is supported through our top k evaluation strategy.

We have described the process as a strictly sequential process. However, in case of problems there might be loop-backs. Consider e. g. the case that the reservation of the resources fails. Then the user goes either back to step four to select another match or even back to step three to run another query.

3.2 Modeling Resources with RDFS

In this section, we describe how we model resource descriptions and matching with RDF. Although we focus on traditional Grid resources like high performance cluster computers or large scale storage resources, our system is open to match anything which can be described using RDF.

3.2.1 Resource Description

Typical standards for resource description like the GLUE information model (see section 2.1.5) are based on classes and attributes for these classes. These models can be translated directly to RDF and RDF Schema. The model itself can be described by defining an RDF Schema class hierarchy. The relations between the classes are encoded as RDF properties. Unnamed associations have to be named because blank properties are not allowed in RDF. As a property is not bound to specific classes, the `rdfs:domain` and `rdfs:range` properties should be defined as well. To model the cardinality constraints, concepts from OWL have to be borrowed. Attributes and their types are also modeled using RDF properties, which blurs the distinction between associated classes and attributes of a class.

After translating the schema, individual resources can be described in RDF. Objects are RDF resources, their attribute values and the relations to other objects are encoded using RDF triples with predicates from the schema.

This transformation is a straight-forward process, which defines a new binding for the GLUE information model like the existing binding to XML Schema[1]. However, after this we can start to use the power of RDF to achieve more. We can use generic knowledge to enhance the matchmaking process.

3.2.2 Example

We start with an example how we use RDF [80, 14] in combination with RDFS [18] to describe resources and background knowledge about resources. For simplicity, we do not use GLUE but a reduced artificial information model. The model uses the namespace prefix `pc2:`. The techniques shown here apply to other models as well.

Consider a cluster named SFB running the Debian distribution with kernel version 2.4 of the Linux operating system. The cluster nodes are equipped with Itanium2 processors. Part of the RDF graph describing this system is shown in figure 3.2. Both the operating system and the processor are described by blank nodes; we are not interested in these entities themselves but rather in their type and their properties.

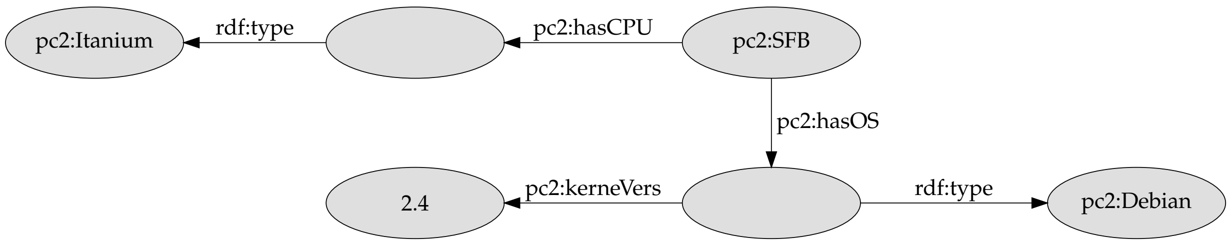


Figure 3.2: RDF Graph for the SFB Cluster.

A simple query graph is shown in figure 3.3. It looks similar to the first graph, but there are some differences. First, the entity which is queried is denoted with a question mark. Second, the type of the operating system is specified as `pc2:Linux` compared to `pc2:Debian` in the resource description. Third, the type of the processor is specified as a combination of two types: `pc2:Intel` and `pc2:64Bit`. Thus, the query is more generic than the description. The entire query graph should be read as a template which is to be matched with a subgraph of the resource description graph.

In order to be able to match this query with the resource description, we need additional background knowledge, which we encode in RDF Schema. It encompasses

[1]see `http://infnforge.cnaf.infn.it/glueinfomodel/index.php/Mapping/XMLSchema`

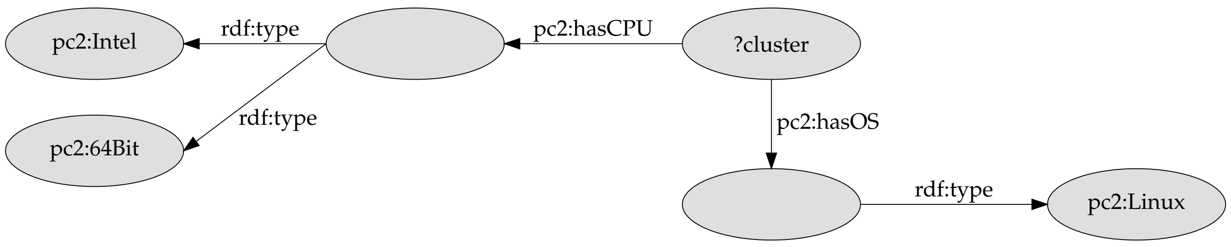


Figure 3.3: Query Graph.

information like `pc2:Debian` is a subclass of `pc2:Linux`. Figure 3.4 shows how this knowledge is encoded in RDF/S.

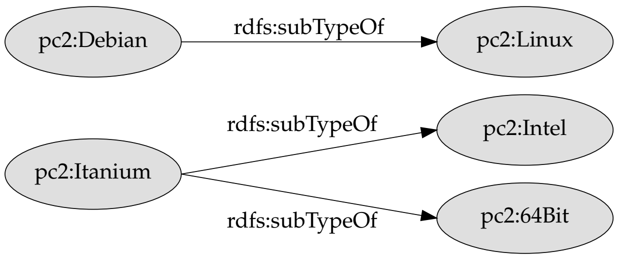


Figure 3.4: Graph for Schema Knowledge.

The background knowledge is integrated in the query process by applying the RDF Schema entailment rules (see section 2.1.3). These rules are executed in a forward-chaining manner, thereby generating new triples. For instance, entailment rule “rdfs9” of [55] generates new `rdf:type` edges. In the resulting graph, shown in figure 3.5, a subgraph is searched which matches the query graph.

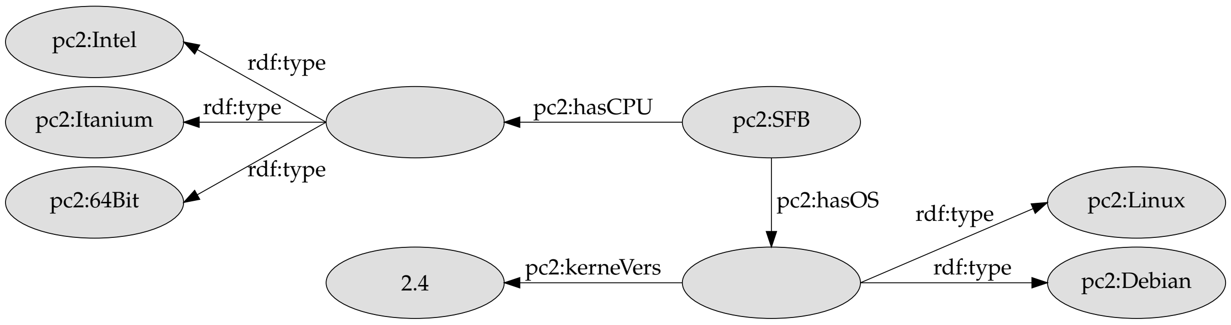


Figure 3.5: Model Graph after Applying RDFS Rules.

3.2.3 Integrating Background Knowledge

The introductory example already shows one type of background knowledge which is very important. Information about the class hierarchy can be used during the matchmaking process. This allows to locally extend the class hierarchy, including very specific classes for entity types that are elsewhere unknown. Through the RDF Schema mechanism, these entities are published also to be objects of more generic types, thus being discovered by queries searching for the generic entities.

Consider for instance the attributes `OperatingSystemName`, `OperatingSystemRelease`, and `OperatingSystemVersion` in the GLUE class **Host**. The GLUE standard [4] contains the following advice how these attributes are filled:

In particular, we propose to use the command `lsb release -d` and to extract the output as follows: the name is given by the content between `Description:` and `release`; the release is given by the content between `release` and the character `(`; the version is given by the content between the character `(` and the character `)`. For instance, if the output of the command `/usr/bin/lsb release -d` is `Description: Fedora Core release 4 (Stentz)`, then the operating system related attributed should be filled as follows: `GlueHostOperatingSystemName: Fedora Core`, `GlueHostOperatingSystemRelease: 4` and `GlueHostOperatingSystemVersion: Stentz`.

This implicates that it is difficult to search for these definitions. The operating system names are not grouped to more generic types, thus only a search for a name like "Fedora Core" is successful. It is not possible to search for all RedHat derivates, neither for anything that is a Linux type of operating system.

The same holds for property hierarchies. Schemas can be extended to use highly specialized properties which are then propagated towards more generic properties. An example is a generic property `pc2:hasComponent` which can be specialized to include the type of the component. Thus queries that look for specific types of components can directly use the specialized property, those that are interested in any kind of component use the generic form.

RDF allows to use further kinds of background knowledge. As triples can also be used to describe classes, generic information about types can be encoded. In figure 3.6, the *class* `pc2:Itanium` is used as subject and several triples describe further information that hold for every Itanium processor. This description is an alternative to the way the same information has been encoded in the above example, allowing a clearer distinction between several aspects.

This way of modeling includes e. g. the possibility to use filters in queries for the word length. If the word length is encoded as a class called `pc2:64Bit` as shown in figure 3.4, queries for CPUs with a word length greater or equal that 32 bit are not possible without listing every word length explicitly. In figure 3.6, the word length is an integer literal, which can be used in filters (see section 2.1.4).

These two options can also be combined together. Consider e. g. a processor class hierarchy where Itanium2 is defined to be a subclass of Itanium, see figure 3.7. A provider describes its resource to have an Itanium2 processor, see figure 3.8. Now look at the query in figure 3.9. It asks for a cluster with a processor that is manufactured by Intel.

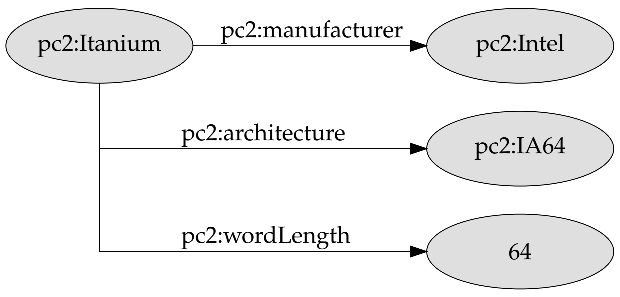


Figure 3.6: Information about the Itanium Processor.

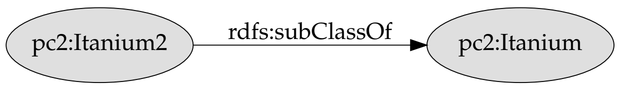


Figure 3.7: Information about the Itanium2 Processor.

How can this query be answered? First, the RDF Schema rules use the subclass relationship defined in figure 3.7 to generate a new triple stating that the CPU of `pc2:ClusterX` is also of type `pc2:Itanium`. This new triple provides a link between `pc2:ClusterX` and the background knowledge for Itanium shown in figure 3.6, which includes the manufacturer Intel. Summarizing, the query is answered using three steps: First, three different RDF fragments are combined. Second, RDF Schema reasoning is applied to the union of these fragments. Third, the query pattern is matched to the resulting RDF graph.

3.2.4 Classifying Properties

RDF includes the possibility to make statements about properties. This is also useful for resource matching. Consider e. g. the case that different ontologies have evolved in the network that are not directly comparable. Each of them uses a different property to connect the resources with the CPU-related information.

In this situation, it is possible to encode information about these properties and to select the properties within the query. The query in figure 3.10 consists of two parts. In the upper part, the variable `?cpuProp` is bound to *any* property that is classified as a `pc2:propCPU` property. In the lower part, this variable is used as a property to select the resources. This makes it possible to match resources encoded using different ontologies.

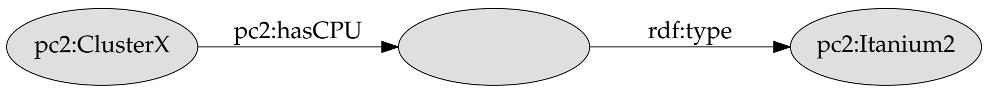


Figure 3.8: Resource with Itanium2 Processor.

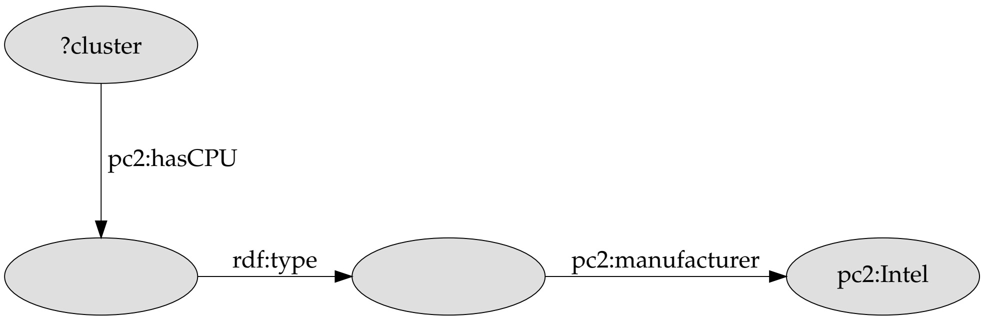


Figure 3.9: Query for a Cluster with an Intel Processor.

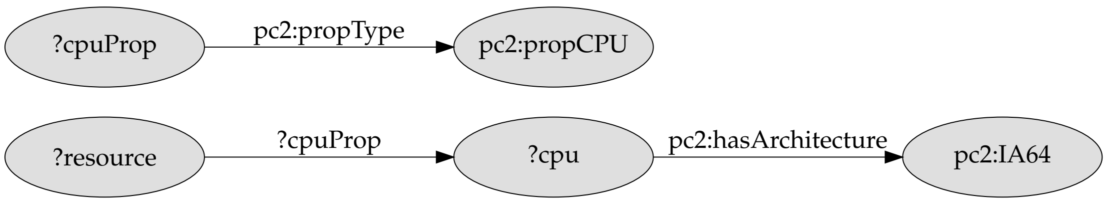


Figure 3.10: Query with Flexible Property Selection.

3.2.5 Static and Dynamic Data

Traditionally, there is a separation between resource description data which is static and used during the matchmaking process, and monitoring data which is highly dynamic. It is typically not part of the matchmaking process, but rather used to ensure proper functioning of the systems.

However, we think that monitoring data is also relevant for matchmaking. As a basic example, the availability status of a cluster is an interesting attribute to be considered in the matchmaking. Also further monitoring data like the current queue length or the percentage of free space on some storage device is interesting when looking for resources.

The GLUE schema separates dynamic from static data in different parts of the information model. The part shown in figure 2.7 is supposed to contain the matchmaking data which is static, and the host-related part shown in figure 2.8 contains information related to functional monitoring. However, this separation is not complete. For example, the MainMemory class contains both the attributes RAMSize and RAMAvailable. The former is static, while the latter is dynamic.

In this thesis, we do not specifically distinguish between static and dynamic data. We assume that every triple which is stored in the system has attached a life time, which indicates after which span of time the triple is assumed to become invalid. This life time is used in the distribution mechanism and for the soft-state removal of triples. Thus, we have a flexible approach that supports both static and dynamic data.

3.2.6 Origin of the Information

Information used for resource description may originate from various sources. This includes hand-written documents, monitoring systems like Nagios[2] or Ganglia[3], and system tools like the `/proc` filesystem. For this thesis, we neither consider the interfaces to these systems, nor the conversion of the emitted information to RDF. We assume that every piece of information is already encoded in RDF and there exist background knowledge which can be used. However, we argue that RDF is extremely flexible and thus vast amounts of data or information can be converted to RDF and thus be integrated in our system.

3.2.7 More Semantics

In this section, we have shown ideas how the power of RDF and RDF Schema can be used for resource description in the Grid. Already with the basic principles, valuable advancements for resource matching can be achieved. The basic principle is that background knowledge is combined with concrete knowledge about resources to do reasoning and to answer the queries.

The more reasoning features are supported, the more flexible the system is. In this thesis, we concentrate on the main elements of RDF Schema, the class hierarchy and the property hierarchy. We further allow to combine RDF graphs encoding background knowledge about classes originating from arbitrary nodes.

Further reasoning features that are interesting to regard in future work include:

- **Transitive properties**. As an example consider the description the a property `pc2:isCompatibleTo`. It is desirable that the formalism allows to declare this property to be transitive, so that a chain of compatibility relationships can be detected automatically.

- **Conversion of units**. Consider a query for a cluster with at least 2 GHz CPU frequency. A resource provider might have advertised its machine to have processors with 2800 MHz frequency. Thus we need a piece of background knowledge encoding that 1 GHz equals 1000 MHz. The reasoning mechanism should integrate this information in the reasoning process.

- **Structural conversion**. When multiple ontologies are used in the system that have completely different underlying models, it is difficult to encode every relevant mediator to allow queries to tap all information sources. The methods shown in this chapter, especially in section 3.2.4, are important initial steps towards this goal. However more ways to encode mediators are needed to be fully flexible with respect to the used ontologies.

[2]see `www.nagios.org`
[3]see `ganglia.sourceforge.net`

3.3 Data Organization

We now look at the question how data can be organized to be useful for resource description. Typical approaches are centralized systems, hierarchical systems, or fully distributed systems. In the first subsection, we argue why we choose a p2p network as foundation for data organization. In the second subsection, we discuss two design alternatives which drive our selection of a structured p2p network.

3.3.1 Peer-2-Peer

Typical RDF reasoners existing today are centralized stand alone systems. Some of them like Sesame [21] can be combined with state-of-the-art database systems and are thus capable of storing large amounts of triples. However, their scalability is limited when numerous data sources from various locations permanently update their data. Also the query processing throughput is limited in a stand alone system. Thus we do not consider centralized systems for our system.

Hierarchical systems like MDS [32] have similar problems, although they are differently organized. In MDS, the general trade-off is as follows. When querying an MDS node, the answer depends on its position in the hierarchy. Nodes which are located near the leafs provide only a limited view of the Grid, spanning only a part of the available resources. Nodes which are closer to the root have a broader view. However, the information in these nodes is typically aggregated and thus only includes coarse-grained summaries. Additionally, the information is fresher the closer a node is to the leaf nodes of the hierarchy.

We target to have a complete view of all available resources as fresh as possible. To achieve this, we use a p2p network which provides the needed scalability properties in conjunction with other interesting properties.

P2p networks fit perfectly with Grids as they have common design principles. In p2p networks, each node is autonomous, which mirrors the provider autonomy in Grid systems [41]. Each node is free to join or leave the network, and determines by itself which information it wants to publish. Furthermore, Grids are highly dynamic constructs. Virtual organizations (VOs) are created or removed as needed e. g. by new projects. Thus resource information for these VOs is published or removed dynamically.

Choosing a p2p network as infrastructure basis for our network, means that we have also to choose between unstructured and structured p2p networks. For this thesis, we use a structured p2p network. In the next subsection, we describe the general design choices that underpin this decision.

3.3.2 Design Choices

We now explain two general design choices we call **global data system model** and **local data system model**, see [56]. We motivate why we pick the former for our system. This leads us to choose a structured p2p network as infrastructure basis.

Consider the general problem description for this thesis. There are numerous information sources scattered over a large network. They might use the same ontology, however they might also use different ontologies. The information in each node consists of some facts, but also of background knowledge that is valuable for answering questions.

One choice to build a system is to have a separate query processor in each node that answers the query using its local knowledge. We call this the local data system model, see figure 3.11. This implicates that a single answer to a query never includes information from multiple nodes. In order to overcome heterogeneities nodes may include *mediator* information that describe how a query is translated from one ontology to another. This is visualized in the figure e. g. by A→C meaning that queries that are shipped from node A to node C are translated from the ontology used by node A to that used by node C. In this approach the main challenge is to route the query to those nodes that are expected to have the best results, and where the translation path is as lossless as possible.

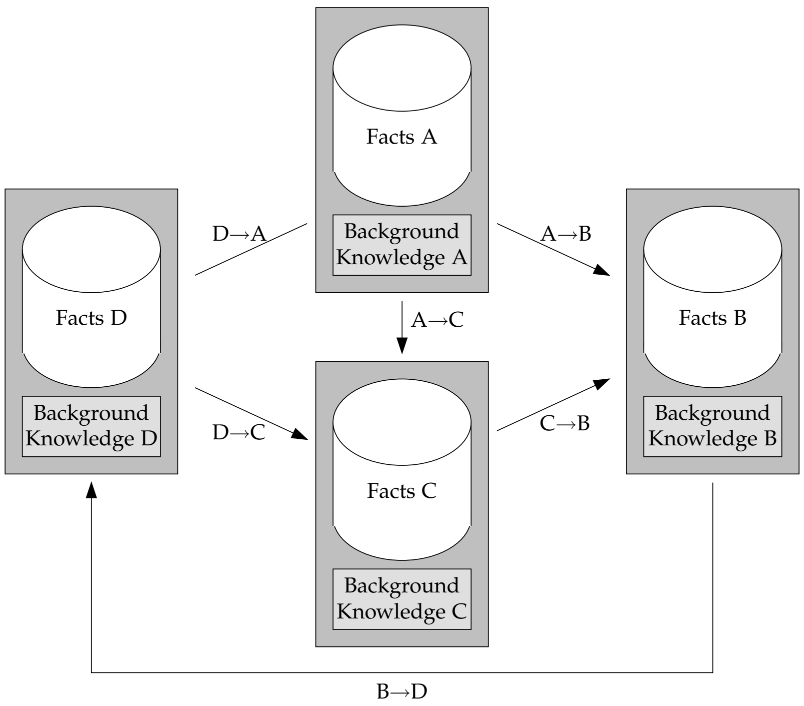


Figure 3.11: Local Data System Model.

Another approach is to look at the entire data scattered over the whole system, and try to answer queries by reasoning with *all entries*. This approach is shown in figure 3.12. It mirrors the behavior of a centralized system where each source contributes its local data. We thus call it global data system model. However, the query processing is done in a decentralized fashion. The challenge is to identify and retrieve *parts* of the information stored at different nodes that is relevant for the query. These parts are combined to form the final answer.

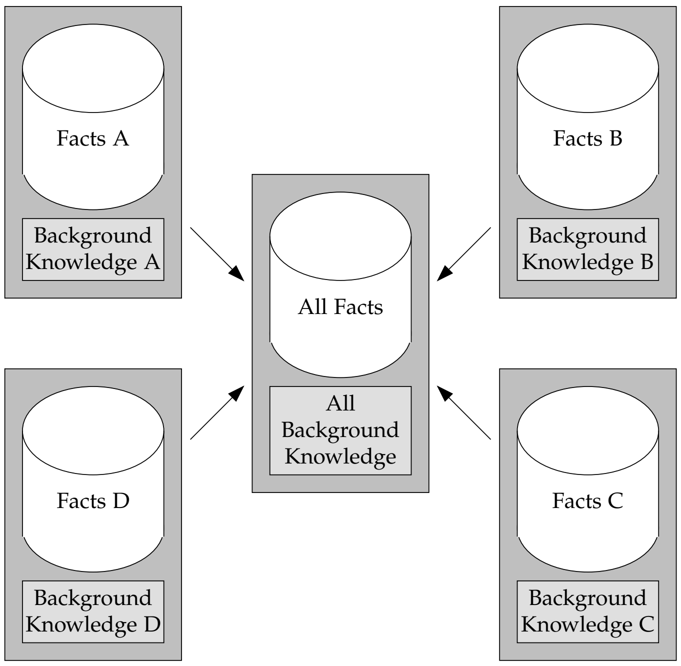


Figure 3.12: Global Data System Model.

The latter approach is superior, because it allows to integrate multiple sources of information in a very fine-grained manner. Small chunks of the information relevant for the query are retrieved and used. Structured p2p networks as described in section 2.2 are a perfect solution in the given scenario. In the next chapter, we show how it is possible to build such a system grounding on a DHT-based p2p network.

3.4 Summary

In this chapter, we have looked at our main application: resource discovery in the Grid. We have shown how the process of resource discovery is embedded in the context of a job's life cycle in the Grid. We have shown how we use RDF/S to model resources and background knowledge about resources. We have identified p2p systems as a perfect infrastructure for organizing the data. We have further argued that it is important to combine pieces of information originating from different sources in individual query answers, and that structured p2p systems are a basis to achieve this goal.

4 P2P based RDF Reasoning and Querying

In this chapter, we describe our system, its architecture and the underlying methods which target the main goals formulated in the introduction. The system stores *semantically rich information* by using the W3C standard Resource Description Framework (RDF) [80]. It is schema-aware and allows to *integrate schema information* originating from different schemas by recognizing RDF Schema information and by applying the appropriate RDFS rules [18]. As underlying infrastructure, it uses a distributed hash table [73] (DHT) to distribute and query the information and thus *assures scalability* both for large numbers of participating nodes, large numbers of concurrent queries, and huge amounts of information.

We start with an overview of the system in the first section and then formalize the query problem. This formalization allows us to look at the complexity of the problem in section 4.2.2. We then describe the distribution of the information over the nodes of the p2p network. The RDF Schema rules are evaluated during this distribution process as described in section 4.4. In the following two sections, we explain two different query evaluation strategies. The first strategy retrieves all matches for a given query. However, as this can sometimes be a large set, and typically only a few results are relevant for the user, we additionally describe a second strategy which efficiently retrieves only a few *good* matches. "Good" means according to a user-defined ordering of the results. An experimental evaluation of the performance of our system is given in chapter 5.

4.1 Architectural Overview

We now describe a tour through our system to give the reader an understanding of the big picture. We highlight the main components and mechanisms, introduce important components and interactions with its ecosystem, and emphasize the interaction between the information providers and information consumers. The main steps are visualized in figure 4.1. The picture mirrors steps one through three from the job life cycle in the Grid as described in the previous chapter (see figure 3.1 on page 46) with a more technical content focussing on the methods presented in this thesis.

We start our tour with the information providers. The information within the network is stored and queried as RDF triples. Thus, other formats have to be converted to RDF. As described in section 2.1.3, the RDF-based information is described as a set of triples consisting of subject, predicate, and object. RDF is flexible and virtually any kind of structured or semi-structured information, including XML documents,

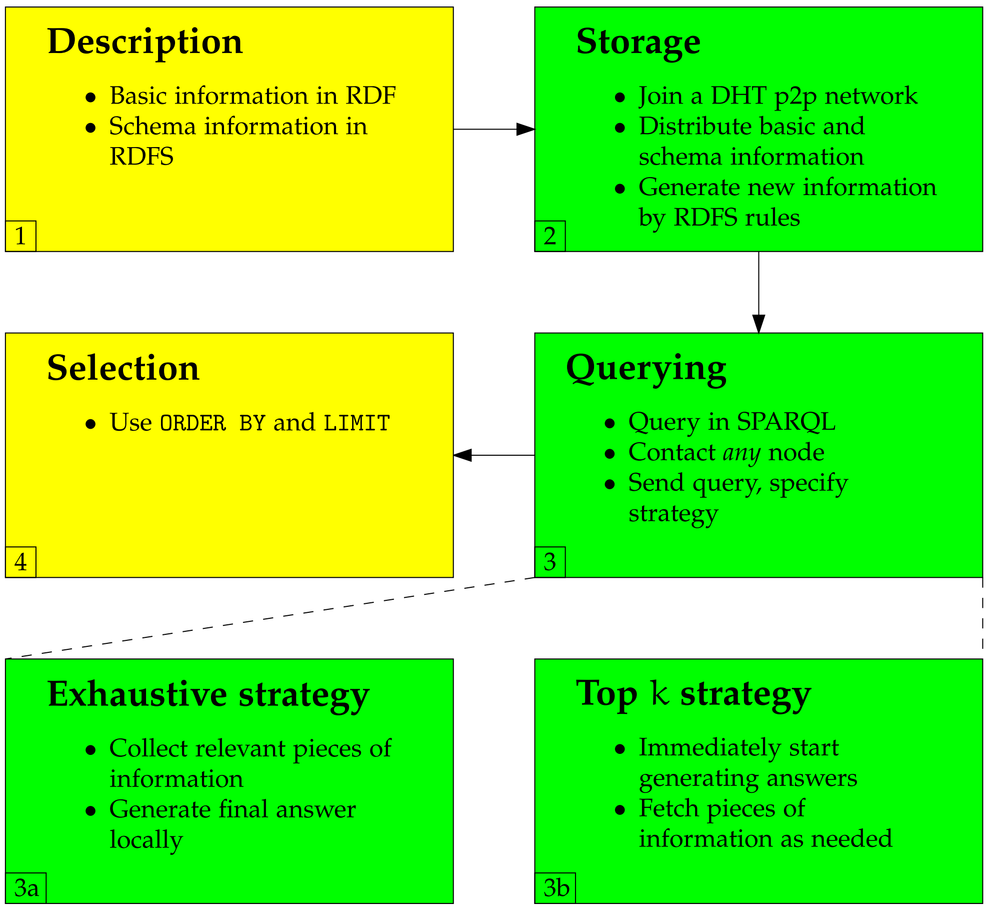

Description
• Basic information in RDF
• Schema information in RDFS
1
Storage
• Join a DHT p2p network
• Distribute basic and schema information
• Generate new information by RDFS rules
2
Selection
• Use ORDER BY and LIMIT
4
Querying
• Query in SPARQL
• Contact any node
• Send query, specify strategy
3
Exhaustive strategy
• Collect relevant pieces of information
• Generate final answer locally
3a
Top k strategy
• Immediately start generating answers
• Fetch pieces of information as needed
3b

Figure 4.1: Process Overview.

relational- or object-oriented databases, and attribute/value based resource descriptions can be mapped to RDF.

Every RDF document uses a certain *schema* describing the used vocabulary. Documents specifying the relationships between different vocabularies are called *mediators*. Both schemas and certain kinds of mediators can be specified using RDF Schema. If these documents are available, they are incorporated in our system and will be used during the distribution of information and during the query matching process.

We do not insist on the existence of schema information or mediators. However, it is important to note that mediation between different schemas will only be possible if the necessary mediators are either explicitly present or can be created by chaining the existing mediators. We do not generate new mediators by some automatism. Automatic schema matching is a complementing task which could be integrated into our system, see [87].

To distribute and query the information, every provider joins a self-organizing DHT-based p2p network, which is the basis for all communications between the providers. The information consumers do not have to join the network; instead, they contact any of the existing providers who will then communicate with the other nodes[1] to collect the necessary bricks to answer the query.

Both the schema information and the basic information is then distributed over the p2p network. Each piece of information will has a time-out value which describes its expected minimal validity period. As individual nodes of the system may fail or a reorganization of the underlying DHT may change the responsibility, a soft-state time-out is the only viable approach to guarantee eventual removal of out-dated information. This implies that information has to be constantly re-distributed.

Thus each node in the network has a process which regularly determines which information has to be disseminated. For each triple, one or more target nodes are identified by the DHT lookup operation. Then the information is sent out to the target node.

RDF Schema reasoning is done in a forward chaining manner. This means that every possible triple which can be generated from the existing knowledge is indeed generated. The rules underlying RDF Schema can be evaluated locally on the nodes of the network. As a result of this evaluation, new triples are generated. These triples also have to be disseminated to the network.

Query processing does not have to be aware of the RDF Schema rules, because they have already been pre-processed. This ensures the efficiency of the query processing mechanism.

Any of the nodes in the p2p network can receive a query. Each node will process it in completely the same way. Thus it is irrelevant whether the contacted provider itself stores some of the relevant information. The procedure always starts by breaking down the query into individual triples and by contacting the nodes which store relevant information for the triples. In the exhaustive query evaluation strategy, all relevant information is collected in one run, while the final answers for the query are generated after that phase. In the top k strategy, candidates are fetched as needed step-by-step. Both strategies employ a backtracking algorithm to determine the answers.

4.2 The RDF Query Problem

As we have seen in section 2.1.3, RDF knowledge is encoded as a graph. We call the graph that contains all knowledge from the network the **model graph**. The main element of each RDF query is a graph pattern, called the **query graph**. Thus the problem of finding a subgraph of the model graph which is isomorphic to the query graph is at the heart of RDF query processing. We now look more formally at this problem.

[1]The term “node” has two meanings in this thesis, as it might either refer to a node of the DHT network or to a node of an RDF graph. In most of the cases, the meaning is clear from the context. Otherwise, we use the term “peer” for the nodes of the DHT network and “vertex” for the nodes of the RDF graph.

4.2.1 Formal Problem Definition

Both the model and the query graph are directed graphs. The labels of the model graph can be URI references, literal values, or blank node labels. For the following discussion, we do not have to differentiate between URI references and literals, so we define the set of labels to be $\mathcal{L}$ which contains both types of entities. The set of blank node labels is denoted by $\mathcal{B}$. Thus each vertex is labelled with an element of either $\mathcal{L}$ or $\mathcal{B}$. An edge of an RDF graph cannot have a blank label, so only elements of $\mathcal{L}$ are allowed here. RDF permits multi-edges, i. e. more than one edge between a pair of nodes, and no stand-alone vertices are allowed, so the graph can be described as a triple set

$$T_M \subseteq (\mathcal{L} \cup \mathcal{B}) \times \mathcal{L} \times (\mathcal{L} \cup \mathcal{B}) \tag{4.1}$$

The query graph is defined analogously. However, instead of blank node labels, we use variables from a set $\mathcal{V}$ of variables, and we allow edges to be labelled with variables. Thus the query graph in triple representation is

$$T_Q \subseteq (\mathcal{L} \cup \mathcal{V}) \times (\mathcal{L} \cup \mathcal{V}) \times (\mathcal{L} \cup \mathcal{V}) \tag{4.2}$$

For convenience, we denote the set of variables occurring in T_Q by $\mathcal{V}_Q$, and the set of literals occurring in T_Q by $\mathcal{L}_Q$. The sets $\mathcal{L}_M$ and $\mathcal{B}_M$ are defined analogously.

The desired semantics for our query evaluation are as follows: given a model graph T_M and a query graph T_Q, find every mapping for the variables occurring in T_Q to the set of blank nodes and literals occurring in T_M, such that for each triple in T_Q there is a matching triple in T_M. Thus we search for mappings

$$R : \mathcal{V}_Q \to \mathcal{L}_M \cup \mathcal{B}_M \tag{4.3}$$

such that for every triple $\langle s, p, o \rangle \in T_Q$ there is a triple $\langle s', p', o' \rangle \in T_M$ such that:

$$\begin{aligned} s \in \mathcal{V}_Q &\Rightarrow s' = R(s) & \quad s \in \mathcal{L}_Q &\Rightarrow s' = s \\ p \in \mathcal{V}_Q &\Rightarrow p' = R(p) & \quad p \in \mathcal{L}_Q &\Rightarrow p' = p \\ o \in \mathcal{V}_Q &\Rightarrow o' = R(o) & \quad o \in \mathcal{L}_Q &\Rightarrow o' = o \end{aligned} \tag{4.4}$$

Note that this definition includes the possibility to match two different variables to the same value, as we do not insist on R being an injective function, which makes the problem a bit different to the subgraph isomorphism problem (see section 2.3).

We impose a further restriction upon T_Q. As we use DHTs as underlying routing layer, we need a fixed key as a starting point. Thus each triple must have some link to another triple which contains such a key. To clearly define this restriction, we need some new terminology.

We call a triple a **key triple** iff it has at least a URIref or a literal either as subject, as predicate, or as object. This means that all triples are key triples except those that consist of three variables. We further call two triples **connected** if they share a common variable. We call a triple **key connected** if it is either a key triple, or connected to a key triple, or connected to another triple which is key connected. Using these definitions, we formulate the following requirement:

We require that each triple must be key connected (4.5)

The other way round, this definition states that there must not be a triple consisting solely of variables that has no connection to other triples that contain elements that can be used during the DHT routing.

4.2.2 Complexity of the RDF Query Problem

We now use the formal problem definition to study the complexity of RDF querying. As the subgraph isomorphism is known to be NP complete [30], it seems obvious that the same holds for our problem. We now show that this is indeed the case.

In order to study the complexity of RDF querying, we define the **RDF query decision problem**. The problem is to determine, given an RDF model graph and an RDF query graph, whether there exists a query answer to the corresponding RDF query problem.

We now show that this problem is NP complete. First, observe that a reasonable encoding of the result function R is polynomial in the input length. It serves as a certificate which can be checked in polynomial time, and thus the problem is in NP.

We now reduce the 3SAT problem to the RDF query decision problem. The proof resembles the proof given by Cook in [30] for the NP completeness of the subgraph isomorphism problem. However, we adapt it to RDF graphs and use 3SAT instead of D_3 (the problem of checking tautologyhood for propositional formulas in DNF with clauses of length 3).

Let $A = C_1 \wedge \ldots \wedge C_k$ be a 3SAT formula in CNF with k clauses. Let $C_i = P_{i1} \vee \ldots \vee P_{i3}$ be a clause, where P_{ij} is either an atom or the negation of an atom.

We now define the corresponding RDF query problem. We start by construct an RDF model graph M. First, we create a blank node for each P_{ij}, called $b_{ij} \in \mathcal{B}$. Then we create a vertex labeled with a unique URIref for each clause C_i: $u_i \in \mathcal{L}$. As we do not need edge labels, we use a single URIref uniformly for every predicate: $e \in \mathcal{L}$. We insert a triple for each P_{ij} which uniquely connects each P_{ij} to its clause: $\langle u_i, e, b_{ij} \rangle \in M, 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3$. Now we connect each pair of P_{ij}, P_{mn} which do not belong to the same clause and which do not contradict: $\langle b_{ij}, e, b_{mn} \rangle \in M$ iff $i \neq m$ and P_{ij} and P_{mn} are not a contradicting pair (i. e. $x, \neg x$ for some atom x). Note that by symmetry also $\langle b_{mn}, e, b_{ij} \rangle$ will be in M. An example for the 3SAT formula

$$C_1 \wedge C_2 \wedge C_3 = (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \quad (4.6)$$

is given in figure 4.2. We have omitted the edge labels and arrow heads, and we have inserted the literals into the blank nodes for better readability.

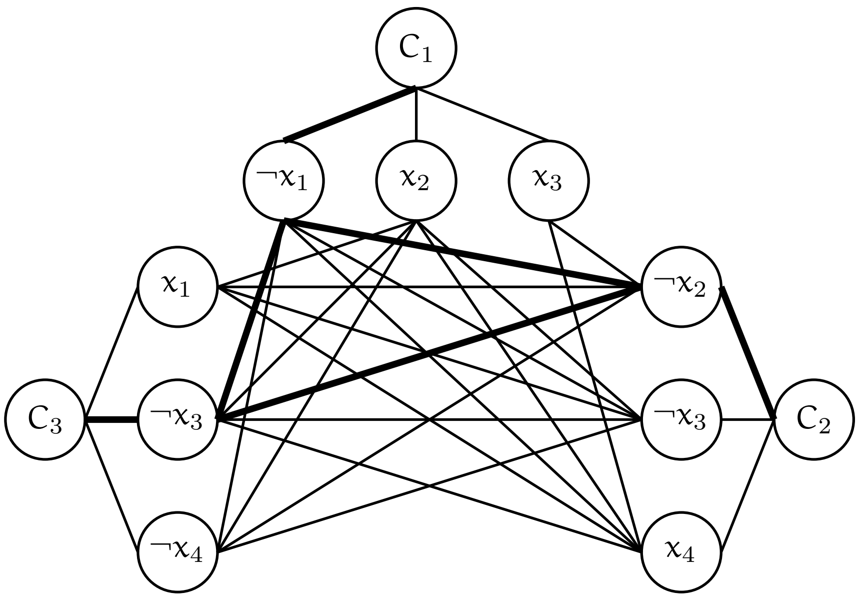


Figure 4.2: Model Graph for 3SAT Reduction.

The query graph Q contains k variables v_i, one for each clause. The v_i are fully connected, i. e. $\langle v_i, e, v_j \rangle \in Q$ iff $i \neq j$. Additionally, we force each v_i to be mapped to a b_{ij} for some j $(1 \leqslant j \leqslant 3)$ by inserting the following triples: $\langle u_i, e, v_i \rangle \in Q$. For the above example, the resulting query graph is shown in figure 4.3.

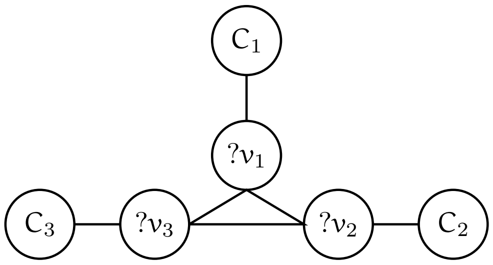


Figure 4.3: Query Graph for 3SAT Reduction.

Clearly, this construction can be performed in polynomial time. We now claim that an instance of this problem is solvable iff the corresponding 3SAT instance is satisfiable. First consider a 3SAT instance A which is satisfiable. As it is satisfiable, there is a truth assignment satisfying the formula, and for each clause, there is at least one literal which evaluates to truth. Pick randomly such a literal P_{ij} for each clause C_i and define $R(v_i) := b_{ij}$. Then each pair b_{ij}, b_{mn} must be connected in M because they cannot contradict as they both evaluate to true. Thus R defines a valid solution to the RDF query problem. The opposite direction of the proof works similarly. A given solution R is used to define a truth assignment. If $R(v_i) = b_{ij}$, look at P_{ij}. If P_{ij} is a negated literal, set its variable to false, else to true. As all these literals are connected, no contradiction can arise, and thus the truth assignment is well-defined and establishes satisfiability of the formula. This concludes the proof.

At a first glance, this is bad news. It means that we cannot hope to find an efficient algorithm to solve our problem. However, the main scalability issue that we expect is a growing size of the model graph. In contrast, we expect the query graph to be of reasonable size even for very large networks.

Thus we are also interested in the **data complexity** of our problem. This is the complexity when looking at a fixed query with different model graph sizes. In [49], it is pointed out that the data complexity of RDF querying is *polynomial*. This applies as well for our definition of the RDF query problem. The number of variants to probe is bounded by n^d, where n is the number of vertices in the model graph and d is the number of variables in the query. With a fixed d, this is a polynomial.

4.2.3 Further Elements of SPARQL Queries

In the foundations chapter, we have looked at the SPARQL query language, see section 2.1.4. It includes further options that go beyond pure search for matching subgraphs. In this thesis, we restrict our focus to subgraphs and the top k related features of SPARQL, namely `LIMIT` and `ORDER BY`. We further restrict the `ORDER BY` when used in combination with `LIMIT` in two ways. First, we support ordering using a single variable. Second, we expect that this variable appears in a triple pattern that contains at least one fixed URIref or literal. Thus we can describe this requirement using the definitions from section 4.2.1 as follows:

We require for the top k case that
the triple containing the `ORDER BY` variable is a key triple (4.7)

Further graph-related options include optional patterns, unions of multiple patterns, and nested patterns. The methods presented in this chapter can be extended to support these features. However to keep this thesis focussed, we have omitted them.

As a default implementation, filters can be applied in a further evaluation step after generating the query results. However, an earlier application would improve performance. This is an interesting field of research beyond the scope of this thesis.

4.3 Knowledge Distribution

In order to query the knowledge, we have to pre-distribute the RDF triples to well-defined nodes in the network to prevent flooding of queries over the network nodes. Each node i has initially stored a set of RDF triples, which contains both schema knowledge and instance knowledge. This knowledge might come from any external source generating RDF triples.

In order to be able to query the knowledge, we have to have a way to find relevant triples for the query, as we do not want to query each node in the network. For this purpose, we connect the nodes via a structured peer-2-peer network which implements a distributed hash table [11].

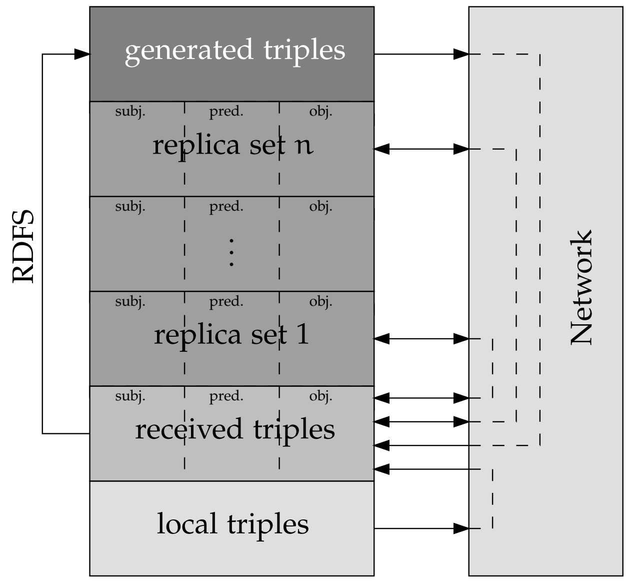


Figure 4.4: Triplestores.

As described in section 2.2, there are different DHT approaches available, all having some kind of lookup mechanism in common. This lookup mechanism enables the user to determine a specific node which is responsible to store data for a certain key.

In our scenario, we use the URI references respectively the literals as keys. We store each triple three times, indexing by the subject, predicate, and object. Each node sends out its own triples to three nodes which are responsible to store the copies. The target nodes store the triples for later retrieval.

After finishing the distribution process, the whole model graph is accessible in a well-defined way over the DHT network. There are several ways to retrieve triples from the network. To retrieve a set of triples, at least one part of the triples must be fixed. We use this part as a key to the DHT network for retrieving all triples with this value.

The dissemination of the triples is done by a process which runs continuously on each node. It wakes up in regular intervals and checks which triples have to be disseminated. Additionally, it can be activated by another process e. g. because the local neighborhood has changed.

Each node has multiple RDF databases where different types of triples are stored, see figure 4.4. In the *local triples* database, all knowledge which originates from the current node is stored. In *received triples*, all triples are stored for which this node is responsible due to the DHT mapping. As nodes may fail, there is also a need to have backup nodes, which hold replicas of the triples. Thus each node stores additionally multiple sets of replica triples. In the set *replica set* i, the triples are stored for which the current node is the `i`-th replication node. Finally, the *generated triples* set contains all triples which were created by applying reasoning rules, see section 4.4.

All local triples are regularly disseminated to the responsible nodes in the network. The local node may occasionally be the responsible node for some of the 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 more unlikely this will be. Thus we do not take special care of these cases, but instead handle them using the local/remote transparency offered by the loopback interface.

Unless a triple has e. g. an identical subject and predicate, it is disseminated to three nodes in the network. The larger the network grows, the higher will be the probability that these nodes are actually different nodes. Thus, for large networks our distribution procedure will result in a triplication of the triple amount, not counting replication.

The root node (see section 2.2.3) which receives a triple is responsible for copying the triples to the replica nodes. It knows the addresses of the replica nodes through the `replicaSet` function of the DHT. Thus the dissemination process additionally sends out all triples in the *received triples* set to the replica nodes.

Furthermore, reasoning rules are applied to the received triples which result in generating new triples. They are stored in the *generated triples* set. These triples, in turn, also have to be sent to the responsible nodes. There, they are stored in the *received triples* set and then copied further to the replica nodes.

4.3.1 Life Cycle of Triples

Owing to the dynamics of p2p networks, it cannot be guaranteed that a request to remove a triple will be successful. It is e. g. possible that a node that takes part in the network only for a short time receives a request to delete a triple. After it has left the network, the old node which has previously been responsible for the triple, becomes root node again. However, it did not receive the delete request, and thus the triple will still be reported by queries.

The only way to guarantee eventual removal of out-dated triples is to use timeouts. Thus each triple has a time to live, and will be removed from the network when this time is over. This means that each triple which still exists has to be continuously refreshed by disseminating it again. It is up to the information source to decide how stable the triple is expected to be. According to this decision, the triple can have either a short time to live, combined with frequent refreshes, or a longer time to live with less frequent refreshes.

4.3.2 Node Departure

In a p2p network, we assume that nodes leave the network without notice, e. g. because they crash, or because the network connection to the nodes fails. The network has to ensure that the operation of the whole network is not affected by such a failure. The

DHT layer ensures that the p2p overlay stays connected. However, another node immediately gets responsible for the ID range previously covered by the departed node. When the DHT fulfills the requirements of the common API, it has to be organized in such a way that the new node which takes over has previously been the first replica node. This means, that for any given ID, if the root node fails, automatically the node with rank 1 becomes root node, the node with rank 2 becomes rank 1 node, etc.

Thus, we do not have to take special provision for failing nodes. Owing to the used DHT scheme, the data from the replication will automatically be available to queries. The only node which has to receive new triples is the node with rank n, if n replicas are used. This will be performed by the new root node (former rank 1 node), which copies all its triples to the replica nodes. This should happen immediately, because multiple nodes may fail in short succession. Thus it is important to distribute the triples instantly without waiting for the next wake-up of the dissemination process. Each node recognizes that a node in its neighborhood has failed through the common API call-back method `update`. Every time this call-back is triggered through the DHT overlay layer, the dissemination process is woken up and immediately starts to deliver triples.

4.3.3 Node Arrival

Although it sounds a bit astonishing, node arrival is more complicated than node departure. This is due to the way DHTs organize access to knowledge. If a new node joins the network, it immediately becomes responsible for some part of the ID space. Thus each request for triples in this range immediately reaches this node. However, distributing the triples to this node may take a while, and thus the node cannot appropriately answer these queries. This means that the node must forward the requests to its replica nodes as long as it does not have all relevant knowledge to answer the query.

The nodes which hold the original versions of the triples in their *local triples* set, may not be in the neighborhood of the new node. Thus they will not be informed of the arrival of a new node through the `update` call-back. For this, they cannot disseminate the triples to the new node immediately; instead, the triples will be distributed during the next distribution period according to the triple life cycle as described in section 4.3.1.

The only nodes which immediately recognize the arrival are the replica nodes and the former root node. These nodes will re-start their dissemination process and send all relevant triples to the new root node. The former rank 1 node additionally sends a message to the new node as soon as it has copied all triples known to him for the given part of the ID space. After receiving this message, the new node can stop forwarding the requests to the replica nodes and answer the request by itself. The new root node additionally uses a time-out to limit the forwarding process.

4.3.4 Accessing the Knowledge

After distributing the knowledge, we need an interface which defines the methods to access the data. These methods are later used by the query evaluation algorithms.

We define three functions, `getBySubject`, `getByPredicate` and `getByObject`, which we use to retrieve sets of triples. As an example we describe the `getBySubject` function. It takes a label as input and retrieves all triples from the network where the subject equals this label. It calls the lookup operation of the DHT network to retrieve the network node which stores these triples. So the execution time of these functions is determined by the time the lookup operation takes plus the transfer time of the result set. It is a central goal of the query algorithm to minimize both the number of calls to these functions and the size of the returned triple sets.

4.3.5 Blank Node Labels

As explained in section 2.1.3, RDF triples may contain blank nodes, which are described by IDs with local scope. However, "local" is not precisely defined. Local might mean a single document, or a source containing various documents. The semantics of how to merge different documents have to be clarified. In RDF, there are two ways to merge documents: either the blank node labels are assumed to have the same scoping, and are thus unchanged. This means that two vertices from two documents sharing the same blank node label, are identified and merged into a single vertex. The other method treats the scoping different by renaming the blank node labels in a consistent way. This avoids name clashes.

In this work, we assume that blank node label are already renamed to be world-wide unique. This can e. g. be achieved by appending the node ID to each blank node label. The merge of different RDF documents on a single network node is within the responsibility of the local peer and not further regarded in this thesis.

4.4 RDFS Rules

The RDF semantics document [55] describes how RDFS entailment can be viewed as a set of rules which generate new RDF triples from existing ones. For our scenario, the taxonomy-related rules are most important. First, they ensure that the `rdfs:subClassOf` and `rdfs:subPropertyOf` relationships are transitive. Second, they propagate instances of classes and properties towards more generic classes and properties. As an example, the class-related rule states: (see section 2.1.3)

If X is a sub-class of Y,
and A is an instance of X,
then A is also an instance of Y.

We want to derive new information from *every* piece of knowledge that exists in the network. Thus we assume that the triples that constitute the precondition for a rule originate from arbitrary nodes. The crucial question is how we find matching triples that trigger a rule when combined.

Our distribution scheme presented in the previous sections solves this problem inherently. As the pre-conditions of all rules share at least one common URIref, there is always at least one node were all triples are locally known. This means that all RDFS rules can be evaluated locally without network interaction. However, the resulting triples have to be further distributed to the responsible nodes.

The whole process works as follows. First, the triples are disseminated from the local triple store through the described mechanism. Each node collects the received triples in its *received triples* store. The RDFS rules are evaluated over this triple store. These rules propagate instances along the class hierarchy towards more generic classes and triples along the property hierarchy towards more generic properties. The new triples are stored in the *generated triples* store and redistributed again, as described above. Through the dissemination process, they reach the received triples store on other nodes. Now they are ready to be used again for the RDFS reasoning.

The reasoning process wakes up in regular intervals and checks for new triples in the received triples store. After it has finished, the triples can be sent out during the next iteration of the dissemination process. When looking at an individual instance which is located at some very specialized class at the bottom of the class hierarchy, it will take a maximum of n rounds until the instance is propagated through the class hierarchy, where n is the longest path in the class hierarchy. The same holds for triples which are propagated along the property hierarchy.

Thus the time spent until every triple is generated depends on the intervals at which the reasoning and distribution process wake up. The shorter this interval is, the faster the process is finished. However, the longer the interval is, the more triples are grouped into a single packet during distribution, which saves network resources.

The transitivity of the `rdfs:subClassOf` and `rdfs:subPropertyOf` predicates is supported implicitly. The triples are not generated, but the taxonomy rules implicitly propagate instances or pairs of instances to every class / property in the transitive closure of the `rdfs:subClassOf` / `rdfs:subPropertyOf` relation.

In section 4.3.1, we have described the soft-state lifetime management of triples. The mechanism implicates that each triple has an attached timeout which enforces its removal from the network. For the generated triples, this timeout is the *minimum* of the timeouts of the triples which make up the precondition. Thus the removal of the generated triple is enforced as soon as *any* of the preconditions is removed.

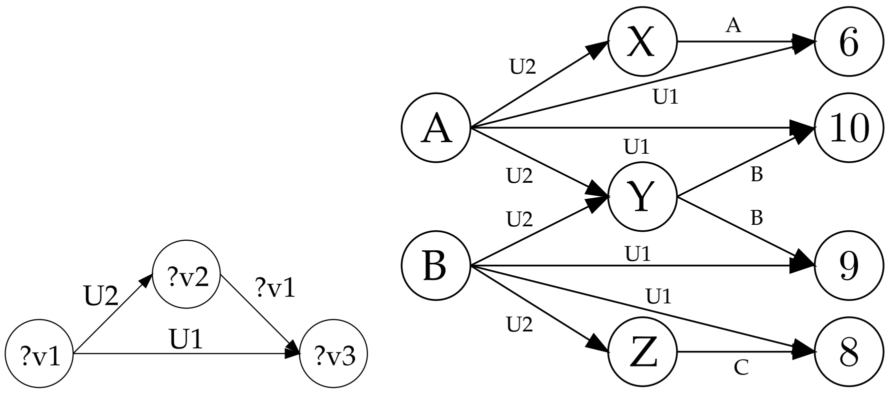
We have published this method in [59]. Although this paper focusses on Description Logic based information rather than RDF based information, its main method is similar. We later adapted it to RDF.

4.5 Query Evaluation

The remainder of this section is concerned with query evaluation. As all supported RDF Schema rules are already processed, we do not have to consider RDFS any more.

We will consider two different query strategies. One is a strategy aiming to retrieve all results for a given query, while the other only retrieves a small set of "good" results, subject to a user-defined criterion.

To explain the query strategies, we use a running example. The example both illustrates the exhaustive as well as the top k evaluation. The query graph is given in figure 4.5(a), while the model graph is given in figure 4.5(b). For simplicity, we omit datatypes and namespace prefixes within this example.



(a) Query Graph T_Q. (b) Model Graph T_M.

Figure 4.5: Running Example.

The query contains three variables: ?v1, ?v2, and ?v3. ?v1 is used both as a subject and a predicate of a query triple. Although this might be surprising at a first glance, it is a valid RDF construct. This is due to the fact that RDF does not have a clear separation between the instance layer and the schema layer. Thus it is possible to assert *statements about properties*, e. g. to classify properties (see section 3.2.4). These statements can be queried with RDF query languages, which leads to such constructs as seen in the example. However, it is clear that our example is artificial. We have constructed it so that we can point out various details during the discussion of our strategies. In realistic examples, all these issues arise, however not in such a condensed form.

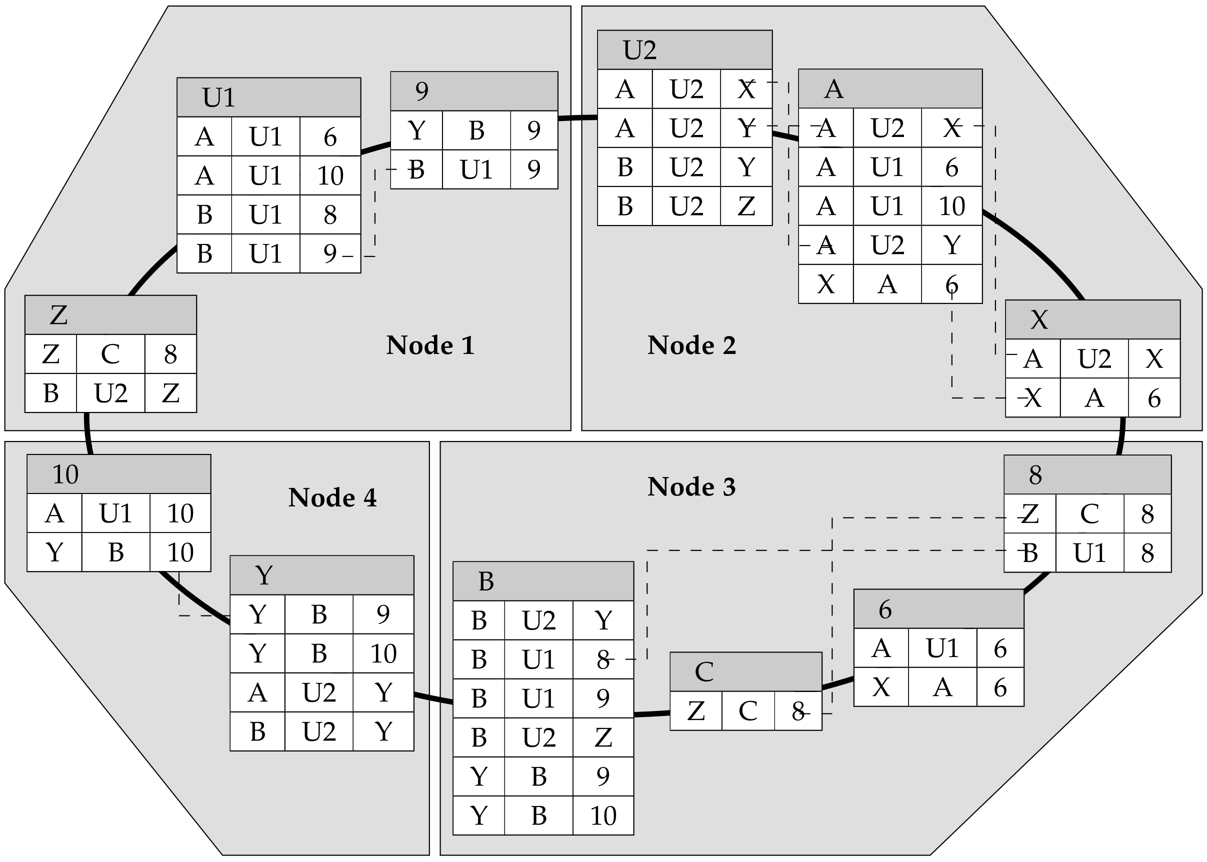
The query has two matches in the model graph with the following bindings:

?v1	?v2	?v3
A	X	6
B	Y	9

The distribution of the triples in the model graph over the p2p network due to the method from section 4.3 is visualized in figure 4.6. Each triple is stored three times, using its subject, predicate, and object as key for the DHT network. Thus for each of the twelve indices occurring in the model graph, a block of triples has to be stored. The DHT algorithm determines which block of triples is stored on which nodes. A possible distribution with four nodes is shown in the figure. However the actual distribution is subject to the DHT algorithm and random parameters like the node IDs.

We have not shown the local triple store, only the received triples. It can happen that a triple is stored according to multiple indices on the same node. Then the triple will be physically stored only once. However, we include the triple in each block and connect the duplicates by a dashed line.

We use replication to ensure failure tolerance in case a node drops out of the network. This replication is not shown in this figure for clarity.



U1		
A	U1	6
A	U1	10
B	U1	8
B	U1	9

9		
Y	B	9
B	U1	9

Z		
Z	C	8
B	U2	Z

U2		
A	U2	X
A	U2	Y
B	U2	Y
B	U2	Z

A		
A	U2	X
A	U1	6
A	U1	10
A	U2	Y
X	A	6

X		
A	U2	X
X	A	6

10		
A	U1	10
Y	B	10

Y		
Y	B	9
Y	B	10
A	U2	Y
B	U2	Y

B		
B	U2	Y
B	U1	8
B	U1	9
B	U2	Z
Y	B	9
Y	B	10

C		
Z	C	8

8		
Z	C	8
B	U1	8

6		
A	U1	6
X	A	6

Figure 4.6: Triple Distribution.

4.5.1 Goals of the Query Evaluation Strategy

During the evaluation of an RDF query, we have to contact various nodes in the network to evaluate sub-queries. The results will then be transferred back. We assume that network latencies and bandwidth restrictions will be the main factors which determine the runtime of our queries. Furthermore we have to consider that routing a message to a target node in a DHT network typically costs $O(\log N)$ routing steps. Thus the goal of our algorithm design is to minimize both the number of messages that are routed and their size.

However, we have to consider an additional factor. We do not only want to minimize the query evaluation time for a single query. We also aim to maximize the throughput of the whole system. This can only be achieved if all nodes are equally loaded during query evaluation. We present a further discussion of this topic in the experimental evaluation in section 5.4.6.

4.6 Exhaustive Query Evaluation

The exhaustive query evaluation algorithm works in two phases. In the first phase, we determine candidate sets for each of the triples in the query graph, as well as for the variables. The candidate sets for the variables and for the triples are mutually dependent, thus we have a refinement procedure which successively removes candidates from both sets which are not suitable. In the second phase, matching combinations of triple candidates are searched locally in the candidates gathered in phase one.

Thus, the first phase collects a subgraph of the model graph distributed over the network which is large enough to contain every result for the query. The second phase is a subgraph matching in this smaller, local model graph which reveals the final results of the query.

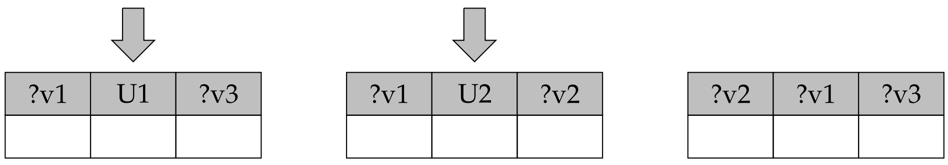
We have published the initial version of the exhaustive query evaluation algorithm in [60]. The optimization based on Bloom filters has been published in [57].

4.6.1 Determination of Candidate Sets

The task of this phase is to identify parts of the model graph which are relevant for the query. The main focus of the algorithm is to reduce network load during this phase. This means, that we want to contact as few nodes as possible, and to transfer a minimal amount of data. We present different strategies, which are evaluated and compared in the following chapter.

Consider our running example. The initial situation is as shown in figure 4.7. We have three triples in the query, and have to fetch candidates for these triples. Owing to the distribution scheme and DHT algorithm, we need a fixed value which can be used as a key for the DHT routing. In this initial situation, only two values are fixed:

the predicates of the first two triples, U1 and U2. The other values are variables whose values are so far unknown.



?v1	U1	?v3

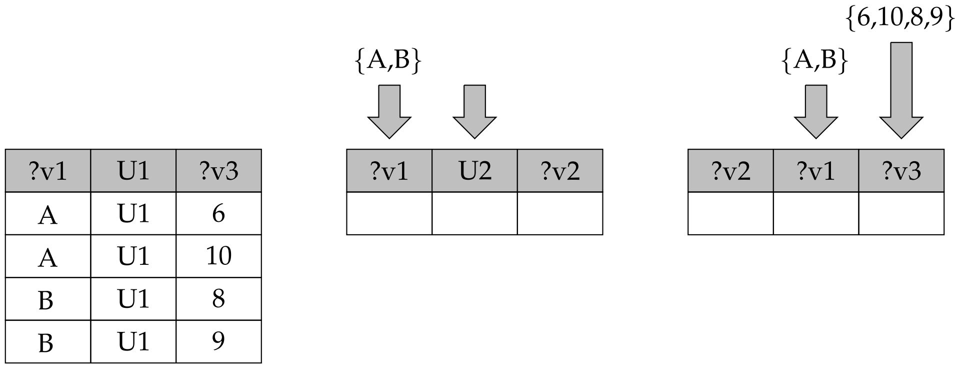
?v1	U2	?v2

?v2	?v1	?v3

Figure 4.7: Identification of Candidates, Step 1.

In the first step, we have to select one of these values, either U1 or U2, and use it as a key for the DHT routing. Consider for example the algorithm chooses U1. Thus the node which processes the query sends a message to the node responsible for U1 and asks it to send back all triples which have the predicate U1. Owing to our triple distribution scheme, this will be the complete set of triples with this predicate, except for triples that have not yet been distributed.

The responsible node is node 1, which can be seen in figure 4.6. It sends back the four triples. After receiving the candidates, the algorithm tries to get candidates for one of the remaining two query triples. Now, not only the fixed value U2 can be used as DHT key, but also the variables ?v1 and ?v3. As we have received a complete candidate set for the first triple, we also known all possible values for ?v1 and ?v3: ?v1 ∈ {A,B} and ?v3 ∈ {6,10,8,9}. This means, that we can also use these values as DHT keys. However, if we choose e. g. ?v3, we would have to send four query messages to the nodes responsible for the literals 6, 10, 8, and 9. The situation is shown in figure 4.8.



?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	8
B	U1	9

?v1	U2	?v2

?v2	?v1	?v3

Figure 4.8: Identification of Candidates, Step 2.

Now consider the algorithm chooses to use the values of ?v1 to retrieve further candidates. It sends out two query messages which are received by node 2 and node 3, see figure 4.6. These two nodes send back all triples they know with subject A and B, which are in sum eight triples. The new situation is depicted in figure 4.9.

However, some of the triples generate contradictions: the predicate of the query triple is U2, while four of the candidate triples have predicate U1. These are marked in

?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	8
B	U1	9

?v1	U2	?v2
A	U2	X
A	U1	6
A	U1	10
A	U2	Y
B	U2	Y
B	U1	8
B	U1	9
B	U2	Z

?v2	?v1	?v3

Figure 4.9: Identification of Candidates, Step 3.

light gray in the above picture. These triples are immediately removed by a `refinement` procedure, which detects contradictions and removes the affected triples. After this refinement, new candidate sets for the variables are known. These candidate sets can be used to fetch candidates for the last query triple, see figure 4.10.

?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	8
B	U1	9

?v1	U2	?v2
A	U2	X
A	U2	Y
B	U2	Y
B	U2	Z

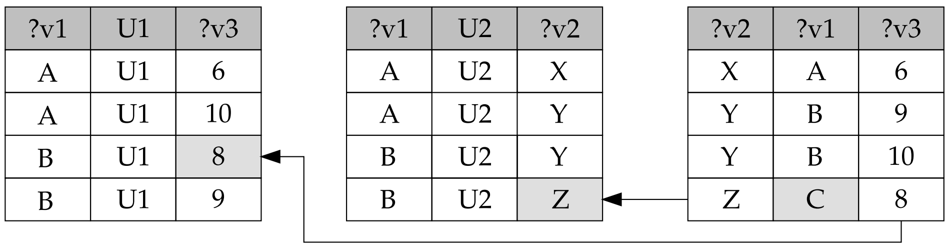
{X,Y,Z} {A,B} {6,10,8,9}

?v2	?v1	?v3

Figure 4.10: Identification of Candidates, Step 4.

In this example, the algorithm chooses to use the known values for ?v2 to retrieve the candidates for the last query triple. Thus to retrieve all triples with subject X, Y, and Z the contacted nodes are 2, 4, and 1, respectively. Through this procedure, four candidates are retrieved. These candidates are shown in figure 4.11. In the picture, the predicate value C of the fourth candidate is marked. This value is a contradiction, due to the known values for variable ?v1. As ?v1 already occurs in the first query triple, and the candidates for this triple only allow the values A or B for ?v1, it will never be matched with C. Thus the `refinement` procedure will remove the last candidate for the third triple. After removing this candidate, further contradictions arise. From the remaining candidates for the last query triple we can deduce that ?v2 will never be bound to Z, and that ?v3 will never be bound to 8. Thus also the candidate ⟨B, U2, Z⟩ for query triple two and the candidate ⟨B, U1, 8⟩ for query triple one can be removed.

This completes the first query phase, the determination of the candidate sets. The resulting candidate sets are shown in figure 4.12.



?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	8
B	U1	9

?v1	U2	?v2
A	U2	X
A	U2	Y
B	U2	Y
B	U2	Z

?v2	?v1	?v3
X	A	6
Y	B	9
Y	B	10
Z	C	8

Figure 4.11: Identification of Candidates, Step 5.

?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	9

?v1	U2	?v2
A	U2	X
A	U2	Y
B	U2	Y

?v2	?v1	?v3
X	A	6
Y	B	9
Y	B	10

Figure 4.12: Identification of Candidates, Final Step.

In summary, we can write down the steps we have seen so far. While there are query triples without candidates, the algorithm performs the following steps:

1. choose a query triple without candidates, which contains either a fixed value or a variable, which already has candidates
2. use the triples stored in the p2p network to collect candidates
3. remove any triples which generate contradictions

In section 4.5.1, we have seen that both the number of messages sent and the size of the messages are the most relevant factors for the run-time of DHT-based algorithms. Thus we are interested in minimizing these key indicators. The interesting points where we can influence these indicators are the first and the second step in the above enumeration. In the first step, the next triple is chosen. In the example, we have chosen the triple randomly. We are now going to present various methods to do better. In section 4.6.3, we will see how we can improve the second step. The third step is local, so here we cannot reduce network communication.

First, we look at the number of messages. In each loop, we have to send one message if we are using a fixed value, like U1 in the example. If we are using the candidates for a variable as we did for the second and third query triple, we have to use as many messages as there are candidates for the variable. Even if a single node is responsible for more than one candidate's value, this does not reduce the number of messages, as the sending node does not know this a priori.

Thus the first strategy is to use either a fixed value, if one is available, or to use the variable with the smallest candidate set currently known. This ensures to minimize the number of messages sent in the next step. We now describe this initial version in more detail.

At each time, the algorithm maintains a set of candidates for each triple, denoted $C_T(t)$, $t \in T_Q$ and a set of candidates for each variable denoted $C_V(v)$, $v \in \mathcal{V}_Q$. Candidate sets may be undefined. As a short-cut, we will write $C_V(v) = \Delta$ iff the candidate set for v is not defined. We furthermore define $|C_V(v)| := \infty$ iff $C_V(v) = \Delta$. As it will simplify the algorithms presented later, we further define the candidate set of a fixed value (either literal or URI reference) to be the one-element set containing that value: $C_V(x) = \{x\}$ iff $x \in \mathcal{L}$.

During each step of the algorithm, some network communication will be used to retrieve the next candidate set, leading to new estimates for the other triples. We use the notion of the *specification grade* of a triple to see where we expect the smallest communication overhead. Owing to the way we distribute the triples, we can either use the subject, the predicate, or the object to retrieve the candidates. Each of these can either be a variable or a fixed value. If it is a fixed value, we have to use a single message to retrieve the candidate set. If it is a variable, the number of messages is determined by the current number of candidates for this variable. If there are no candidates for the variable so far, we cannot use it to retrieve a candidate set.

Thus we define the specification grade of a triple's element as follows:

$$sg1(x) = \begin{cases} |C_V(x)| & : \quad C_V(x) \neq \Delta \\ \infty & : \quad C_V(x) = \Delta \end{cases} \tag{4.8}$$

Because of our definition above, this can be written short-hand as $sg1(x) = |C_V(x)|$. The specification grade for a triple is the minimum specification grade of its elements:

$$sg1(\langle s, p, o \rangle) = \min(sg1(s), sg1(p), sg1(o)) \tag{4.9}$$

The idea behind this definition is that the specification grade determines the number of messages needed. The algorithm is shown in figure 4.13.

An important part of the algorithm is the refinement procedure. We have already seen in the example that some of the retrieved candidates are removed because they contradict the candidate sets for the variables which already exist. However, not only the newly retrieved candidates may be affected. Also the candidates that have been retrieved in prior steps might have to be refined. As each new triple candidate set leads to new, possibly smaller, candidate sets for variables, some of the older triple candidates might contradict these newer, refined variable candidate sets. They have also to be removed, which in turn leads possibly to smaller variable candidate sets, and so on. This procedure is repeated until no more candidates are removed.

Thus there are two ways of refinement. First, we can look at a variable's candidate set. We compare it with the candidate sets for each triple where this variable occurs. If a candidate does not occur within the triple candidate set, it has to be removed from

```
input  : A query graph $T_Q$
output: A set $C_T(t)$ of candidate triples for each triple $t \in T_Q$
1  foreach $t \in T_Q$ do $C_T(t) \leftarrow \Delta$ ;                // init triple candidates
2  foreach $v \in \mathcal{V}_Q$ do $C_V(v) \leftarrow \Delta$ ;       // init variable candidates
3  while there is a $t \in T_Q$ s.t. $C_T(t) = \Delta$ do
       // there is still a triple without candidates
4      determine a triple $t = \langle s, p, o \rangle$ where
5        • $C_T(t) = \Delta$, and
6        • $sg1(t) \leqslant sg1(t') \, \forall t'$ with $C_T(t') = \Delta$ ;
7      if there is no such triple then
8          return error ;                        // requirement (4.5) violated
9      end
       // t is now a triple with a minimal SG value
       // we use the element of t with the smallest SG value as key to
          fetch the candidates
10     if sg1(t) = sg1(s) then
11         $C_T(t) \leftarrow \cup_{x \in C_V(s)}$getBySubject$(x)$ ;
12     else if sg1(t) = sg1(p) then
13         $C_T(t) \leftarrow \cup_{x \in C_V(p)}$getByPredicate$(x)$ ;
14     else
15         $C_T(t) \leftarrow \cup_{x \in C_V(o)}$getByObject$(x)$ ;
16     end
       // filter the candidates according to the variable candidates
17     $C_T(t) \leftarrow \{\langle s, p, o \rangle \in C_T(t) : s \in C_V(s) \wedge p \in C_V(p) \wedge o \in C_V(o)\}$ ;
       // remove further contradictions
18     if refine($C_T, C_V, \{t\}, \emptyset$) = error then
19         return error ;
20     end
21 end
22 return ok ;
```

Figure 4.13: Algorithm candidates.

the variable candidate set. The other way around, we look at the candidate set for a triple and remove any candidates where there is some value not within the matching variable's candidate set. We always keep track of the set of changed variables V and changed triples T, so that we do not have to check every set. The refinement procedure performs these two refinement steps in an alternating way until no more candidates have been removed. The complete algorithm is shown in figure 4.14.

The crucial question is how we can further reduce the network load. The refinement procedure is uncritical, as it works completely local. Thus we have to look at the order in which the triple candidates are retrieved from the network. The definition of the

```
input  : Candidate sets $C_T$ and $C_V$
input  : A set of modified query triples $T$
input  : A set of modified variables $V$
output: The candidate sets $C_T$ and $C_V$, without contradictions
output: An error flag if some candidate set became empty

while $V \neq \emptyset$ or $T \neq \emptyset$ do
    // there are pending modifications
    foreach $t \leftarrow \langle s, p, o \rangle \in T$ do
        // the candidates for query triple t have been modified
        if $s \in \mathcal{V}$ then
            // filter the var. candidates according to the triple cands.
            $C_V(s) \leftarrow C_V(s) \cap \text{subject}(C_T(t))$ ;
            if $C_V(s)$ has been changed then
                $V \leftarrow V \cup \{s\}$ ;                    // mark s as modified
            end
        end
        // similar code for predicate and object
        $T \leftarrow T - \{t\}$ ;                     // modification is processed
    end
    foreach $v \in V$ do
        // the candidates for variable $v$ have been modified
        foreach $t \in T_Q$ do
            if $subject(t) = v$ then
                // $v$ occurs in the subject of t $\rightarrow$ filter candidates of t
                $C_T(t) \leftarrow \{\langle s', p', o' \rangle \in C_T(t) : s' \in C_V(v)\}$ ;
                if $C_T(t)$ has been changed then
                    $T \leftarrow T \cup \{t\}$ ;                // mark t as modified
                end
            end
            // similar code for predicate and object
        end
        $V \leftarrow V - \{v\}$ ;                     // modification is processed
    end
    if some $C_V(v)$ or $C_T(t)$ is empty then
        return error ; // a candidate set is empty:  query has no solution
    end
end
return ok ;
```

Figure 4.14: Algorithm refine.

specification grade as given above ensures a minimal number of messages being sent in the current step. However, it can lead to a large number of candidates for the triple, leading to both a high bandwidth consumption and a large number of messages in further steps. Furthermore, if we already have candidates for other variables in the triple, we can use these candidates to reduce the size of the returned candidate set. In the following two subsections, we introduce methods to benefit from these ideas.

4.6.2 Look Ahead

The first enhancement is to introduce a look ahead for the candidate set size in order to choose the next triple during the first phase of the query evaluation. We implement this look ahead by summing up the result set sizes for each query message instead of only counting the number of messages. This is a trade-off, as it results in further messages during the calculation of the specification grade, however, it might lead to a better path through the query graph with fewer candidates to transfer.

If you look back to the situation in figure 4.8, we have four choices, and thus we have to calculate four possible result set sizes. Comparing the four choices with the triples in figure 4.6, leads to these variants:

Triple	Variant	Messages	Returned triples
t_2	Use candidates {A,B} as subject	2	8
t_2	Use fixed value U2 as predicate	1	4
t_3	Use candidates {A,B} as predicate	2	3
t_3	Use candidates {6,10,8,9} as object	4	8

This example shows that it is indeed possible that a larger number of messages leads to a smaller result set. However, it is unlikely to gain a benefit when we already know an option with n triples return size and the option to test needs at least n messages. Thus we order the options by number of messages and calculate the result set size until the number of messages for the following option exceeds the smallest result set so far.

To retrieve the needed statistical information, we define the following functions: `cntBySubject`, `cntByPredicate`, and `cntByObject`, respectively. They work similarly as the `getBySubject` etc. functions, however, instead of returning a triple set, they only return its size.

The new definition of the specification grade is as follows. We have to define three different functions, referencing to the three elements of a triple. We only describe the subject function *sgs*; *sgp* and *sgo* are analogous.

$$sgs(x) = \begin{cases} \sum_{s \in C_V(x)} \texttt{cntBySubject(s)} & : \quad C_V(x) \neq \Delta \\ \infty & : \quad C_V(x) = \Delta \end{cases} \tag{4.10}$$

The specification grade of a triple is now defined as

$$sg2(\langle s, p, o \rangle) = \min(sgs(s), sgp(p), sgo(o)) \tag{4.11}$$

During the collection of the candidate sets, we retrieve the specification grade of a triple multiple times. Thus we define a further version *sg3* which implements a *cache* which is valid during the evaluation of a single query. Thereby, we can reduce the overhead introduced by the additional lookup-operations.

4.6.3 Bloom Filters

When we retrieve candidates for a triple, we have to choose an element whose value or candidates we use as a key for the DHT lookup. Assume we choose the subject variable which has a set of candidates already. Then we contact all nodes that are responsible for one of the candidates for the subject variable.

We might also have candidates for the other two elements. Each of these elements can either be a fixed value, or a variable. A variable can have some candidates already because it was used in a previous triple. If we further transfer the known candidates for these variables during the `getBySubject` function, the target nodes can reduce the result sets. However, the candidate sets for the variables might be also large, so that the reduction of the result set is outweighed by the additional transfer of the candidate sets.

Consider again the running example, step 4 as shown in figure 4.10 on page 73. The last query triple consists of three variables. For each of the variables, there are already some candidates. We chose to use the candidates for the subject variable ?v2 as DHT key. Thus the nodes responsible for X, Y, and Z where contacted. We did not send the candidate sets for ?v1 and ?v3 to these nodes. Thus we also retrieved the candidate triple $\langle Z, C, 8 \rangle$, which was then removed, because it contradicts the candidate set $\{A,B\}$ for ?v1.

If we had sent all candidate sets to the target nodes, we had sent to each of the three target nodes two sets, one of size two and one of size four. As the result only decreased by one triple, the additional bandwidth consumption to send the known candidates would be larger than the saved bandwidth while returning the triple sets.

Bloom filters [17] are ideal for this situation. As explained in detail in section 2.4, a Bloom filter is a compact representation of a set using an array of bits of a fixed size. Each element which is stored in the filter is hashed multiple times using different hash functions. The bits corresponding to the hash values are set in the filter. Membership test is done in the same way. Thus, each element of the set is reliably detected. However, so-called false positives are possible. This means, that an element is detected to be a member of the set which is in fact a non-member.

Thus we can encode the candidate sets for the other variables as Bloom filters and send these filters to the target node. This node locally sorts out non-matching results and sends back the reduced candidate set. Owing to the false positives of the Bloom filter, there may be too many candidates. This does not matter, as they are removed as before by the refinement procedure. On the contrary, no *false negatives* are possible. Thus, each set member is reliably detected, and no candidate will be lost, which ensures the correctness of the query results.

Bloom filters can be used both in the `getBySubject` etc. functions and the `cntBySubject` functions. The former can be combined with all versions of the specification grade; the latter results in a new definition of *sg4*, which results in a better look ahead as the already known candidates for the other elements of the triple are included. We combine the *sg4* version with the caching mechanism of *sg3*. The cache is filled as soon as a *sg4* value is calculated. This means, that we might later loose better estimates when we have received candidates for more variables.

As an example consider a query pattern ⟨X, ?v2, ?v3⟩. When calculating *sg4* for X, bloom filters are used to encode the candidates for ?v2 and ?v3. Assume that the candidates are $C_T(?v2) = \{A, B, C\}$ and $C_T(?v3) = \{E, F\}$ when the SG value is calculated for the first time. In later iterations of the `candidate` algorithm, we might have to recalculate the SG value. At this time, the sets $C_T(?v2)$ and $C_T(?v3)$ might be smaller. Thus we possibly obtain a value for SG from the cache that is too large. This might lead to prefer another triple in the candidate fetch. However, the caching effect reduces the number of messages dramatically.

Bloom filters can use bit vectors of different size and a varying number of hash functions. We use the values presented in section 2.4 to determine good combinations of parameters, and evaluate the prototype with different combinations. For the results, see chapter 5.

4.6.4 Optimization for `rdf:type`

As we have seen in section 2.1.3, the `rdf:type` predicate is used to encode the type of an individual. In RDF datasets, `rdf:type` is typically the URIref which occurs with the highest frequency. Thus it is seldom a good idea to use `rdf:type` as DHT key when fetching candidates. To prevent or algorithm from querying every time the statistics for `rdf:type`, without ever using it, we implement an optimization into all SG versions: The SG of a triple element is MAXINT iff the triple element is `rdf:type`.

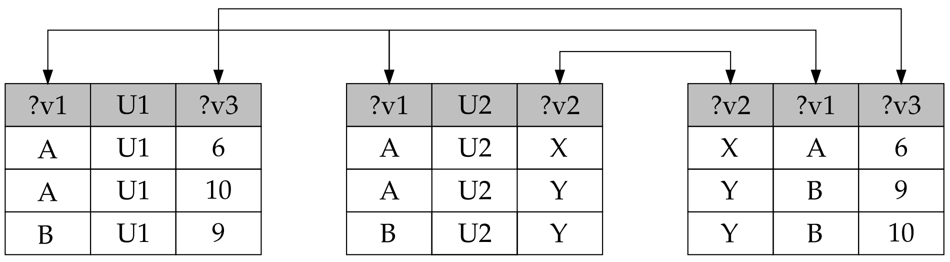
This prevents the algorithm from choosing `rdf:type` as DHT key. However, we do not set the SG value to ∞ because we want the algorithm to use it if there are no other choices.

4.6.5 Final Evaluation

After having retrieved candidate sets for all triples and variables, we have to do the final evaluation to retrieve matches for the query. This is done completely local. However, it can be computationally expensive. In general, every combination of candidates for the triples has to be considered and tested, which are exponentially many. In fact, the query complexity version of RDF querying is NP complete, see section 4.2.2. However, the data complexity version is in P.

We employ a backtracking algorithm. In each step, it chooses a triple which has more than one candidate, and fixes one of the candidates in each iteration of a loop. By fixing a triple candidate, we also fix values for the variables. Triple candidates are only chosen if they do not contradict with previous variable assignments.

The starting situation for the final evaluation in the running example is shown in figure 4.15. As we have four candidates each for query triple one and two, and three candidates for the third triple, there are $3 \cdot 3 \cdot 3 = 27$ possible combinations of candidates. Each variable occurs multiple times, as indicated by the arrows. Thus a combination of candidates is only a match if it assign each variable an single unambiguous value. The two possible matches are indicated in figure 4.16.



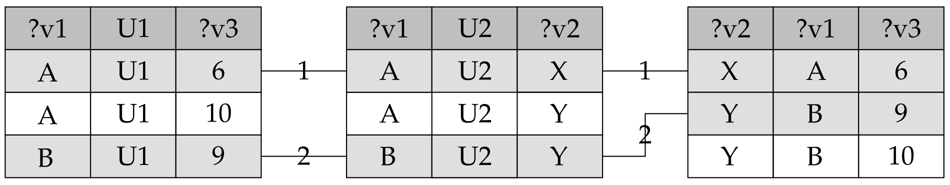
?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	9

?v1	U2	?v2
A	U2	X
A	U2	Y
B	U2	Y

?v2	?v1	?v3
X	A	6
Y	B	9
Y	B	10

Figure 4.15: Final Evaluation.

The algorithm is shown in figure 4.17. It uses a map V to store the variable assignments. A candidate u for a query triple t *contradicts* with an assignment V iff u induces an assignment x for a variable which has already a different assignment $x' \neq x$ in V.



?v1	U1	?v3
A	U1	6
A	U1	10
B	U1	9

?v1	U2	?v2
A	U2	X
A	U2	Y
B	U2	Y

?v2	?v1	?v3
X	A	6
Y	B	9
Y	B	10

Figure 4.16: Final Evaluation: Matches.

4.7 Top k Query Processing

In the previous sections, we have presented an *exhaustive* evaluation algorithm, which aims to retrieve all matches for the current query. The algorithm is designed to minimize this effort, measured in the number of messages and the consumed bandwidth.

However, only in rare cases *every* result is needed. For our main scenario, resource discovery, it is often sufficient to get a list containing e. g. ten matching resources,

```
input  : Query triples $T_Q$
input  : Candidate set $C_T$
input  : A set of variable assignements $V$
1  if there is a $t \in T_Q$ then
       // t is the next query triple to be processed
2      remove t from $T_Q$ ;
3      foreach $u \in C_T(t)$ do
4          if u does not contradict V then
               // u is the next candidate we choose for t
               // selection of u binds some variables, thus we build a new
                  binding set
5              $V' \leftarrow V$ ;
6              store the variable assignments of u in $V'$ ;
7              evaluate($T_Q, C_T, V'$) ;            // step to next query triple
8          end
9      end
10 else
11     the variable assignments in V are a match ;
12 end
```

Figure 4.17: Algorithm evaluate.

preferably with *good* values for certain attributes, like the number of nodes. Additionally, there is a danger in large Grids that generic queries have thousands or millions of matches. Apart from the fact that no client will realistically check or take advantage of each individual match, it is time and bandwidth consuming to generate these matches.

This can be compared with the typical operation of the Google search engine. For average queries, Google finds millions of matches. However, these matches are neither generated immediately, nor are they delivered to the end-user. They are *ranked* and only the best results according to this ranking are delivered.

The main idea behind the RDF top k query processing presented here is to implement the same behavior for our scenario. We aim to devise an algorithm that does not deliver every result, but instead ranks the results and delivers only the k best matches according to the ranking.

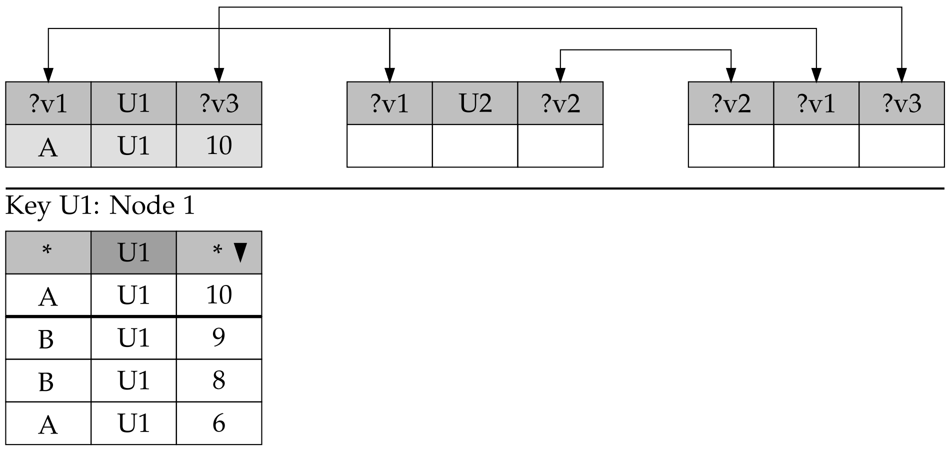
One can imagine numerous ways to define a good ranking for the results of a query. In the context of resource description, this may be e. g. the current load or queue length, the price, security aspects, etc. Ideally, the user should be enabled to define her own ranking criteria, which is a fundamental difference to the Google scenario, where the ranking is system defined.

A ranking could be defined by numerous means. In order to not exceed the scope of this thesis, we restrict to rankings defined by an ordering of the values of a single attribute. It is subject to further research to extend this to arbitrary objective functions that allow the user to combine multiple attributes.

We have published the top k algorithm in [13].

4.7.1 Introductory Example

Now consider the running example of figure 4.5 on page 69. We assume that the user chooses the value of ?v3 as ranking criterion, and that she asks for large values of ?v3. The initial situation is shown in figure 4.18.



?v1	U1	?v3
A	U1	10

?v1	U2	?v2

?v2	?v1	?v3

Key U1: Node 1

*	U1	* ▼
A	U1	10
B	U1	9
B	U1	8
A	U1	6

Figure 4.18: Top k Example, Step 1.

Owing to the ranking criteria ?v3 we start with the first query triple ⟨?v1, U1, ?v3⟩. As both variables are unknown so far, we use U1 as DHT index. We contact node 1, which is responsible for U1 (see figure 4.6), and ask this node to deliver the first triple with predicate U1, ordered by the object's value. The triples with U1 as predicate as stored at node 1 are shown at the bottom of figure 4.18. The returned triple is ⟨A, U1, 10⟩.

In contrast to the exhaustive evaluation presented in the previous section, the algorithm immediately selects this candidate for query triple one and proceeds to the second query triple, without retrieving further candidates for the current triple. By fixing the candidate, we also bind values to ?v1 and ?v3: ?v1 = A and ?v3 = 10. With this binding, the second query triple looks like this: ⟨A, U2, *⟩. We contact the node responsible for U2, which is node 2, and ask for a triples with subject A and predicate U2. Although there is no predefined ordering for ?v2, we have to define one, as we might have to retrieve more candidates later. Thus we need an ordering to specify the current "cursor position" within the candidate list. In the example, we define an ascending order for the object's value, as shown in figure 4.19

The triple returned from node 2 is ⟨A, U2, X⟩. As with the first triple, this yields new binding for variables, in this case the binding ?v2 = X. Now all variables in the query are bound. For the remaining query triple this means that we do not have a

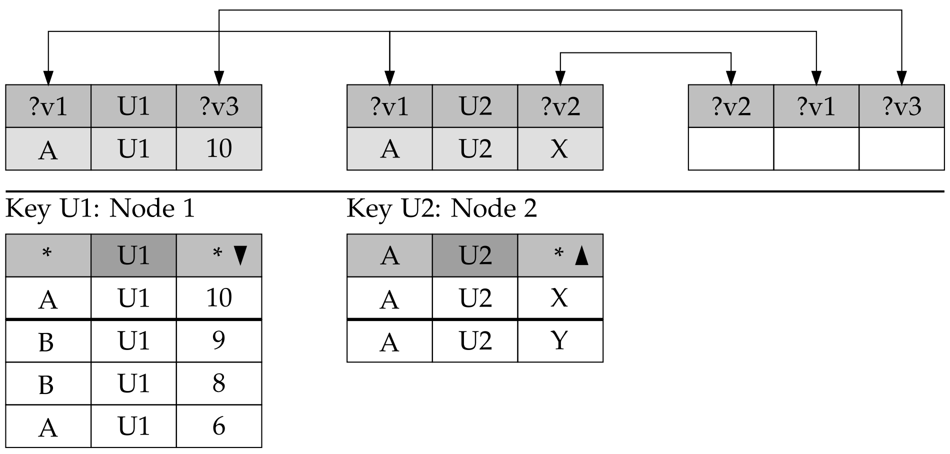


Figure 4.19: Top k Example, Step 2.

candidate list any more, because all three elements of the triple are known. We only have to query whether the triple ⟨X, A, 10⟩ exists. However, this is not the case, see figure 4.20. Thus the previous choices for the other query triples have to be revised.

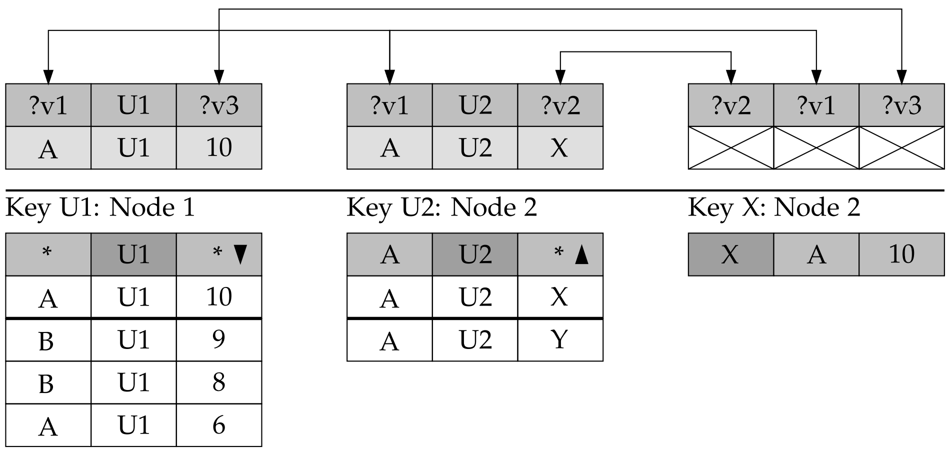


Figure 4.20: Top k Example, Step 3.

Thus the next step is a backtracking step, selecting a new candidate for query triple two. Here, the ordering comes into play. We contact node 2 again and ask for the *next* candidate for the pattern ⟨A, U2, *⟩. This is the triple ⟨A, U2, Y⟩.

Because of scalability reasons, the target nodes do not store any state information. Therefore, the client node has to know the current cursor position and send it later to the other node to fetch the next candidate. This is realized by sending the previous candidate to the target node, asking for the next triple larger than this triple. This implies to have a total ordering of the triples. Thus we specify for each element of the triple which is not a fixed value an ordering, which can be either ascending or descending. Note that we have omitted this in the running example for the subject of query triple one for simplicity. In general we would have specified e. g. an ascending subject ordering in case that there are two or more candidates with equal objects.

Choosing the candidate ⟨A, U2, Y⟩ for the second triple determines the value ⟨Y, A, 10⟩ for the third triple. As this triple also does not exist, we do further backtracking, until we select the second candidate for the leftmost query triple. As this candidate is ⟨B, U1, 9⟩, we now retrieve candidates with subject B for the second query triple. This leads then to the candidate ⟨Y, B, 9⟩ for the third triple. As this one exists, we have found the first match. The match is shown in figure 4.21 and it is the match with the highest possible value for ?v3, as expected by the user.

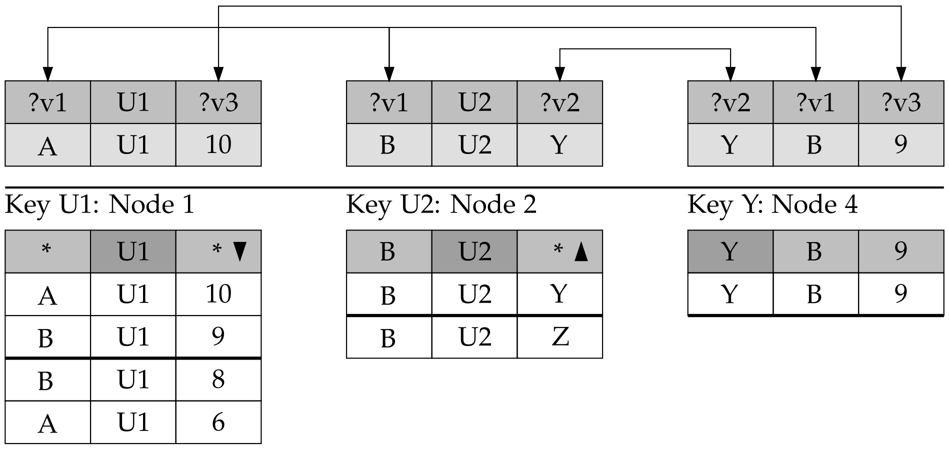


Figure 4.21: Top k Example, Step 4.

By this procedure, we generate the top matches step by step, only retrieving the candidates as needed. After having generated the requested number of matches, the procedure stops.

4.7.2 The Algorithm

The complete algorithm is shown in figure 4.22. The overall operation should be clear due to the above example. The general idea of the evaluation function is to iterate over all possible assignments to triples, assume one, and proceed to a recursive evaluation until we encounter contradictions, find a complete match, realize that we have found a sufficient number of matches, or until we cannot assign any more triples.

An integral part of this algorithm is the `getCandidate` function which delivers the next candidate during in the main loop. A basic implementation uses one lookup each time it is called to retrieve the next candidate from the responsible node in the network. This clearly leads to a very inefficient evaluation. We describe an optimized version of this function in detail in the next section.

```
input : Count of matches so far $nrMatches$
input : Desired maximum number of matches $k$
input : List of query triples $(T_j), 1 \leqslant j \leqslant |(T_j)|$
input : Index of current query triple $i$
input : Map containing the variable bindings $B$
if $i = |(T_i)| + 1$ then
    record $B$ as a match found ;                    // all triples are bound
    return $nrMatches + 1$ ;
end
$c = null$ ;                          // candidate inspected in the last step
loop do
    // we have to fetch the next candidate for $T_i$, which is larger
       than c due to the defined ordering.  the candidate has to
       respect the bindings in $B$
    $t \leftarrow$ getNextCandidate($T_i$, $B$, $c$) ;
    if $t = null$ then
        break ;                              // no more candidates available
    end
    $B' \leftarrow B \cup$ Bindings of $t$ ;                     // update bindings
    $nrMatches \leftarrow$ evalTopK($nrMatches$, $k$, $(T_i)$, $i + 1$, $B'$) ;
    if $nrMatches \geqslant k$ then
        break ;
    end
    $c \leftarrow t$ ;
end
return $nr\_matches$ ;
```

Figure 4.22: Algorithm evalTopK.

4.7.3 Caching and Look Ahead

The presented algorithm sends out one message per step to retrieve the next candidate. Owing to the backtracking, it could be that some candidates will be needed again later. Assume e. g. that the algorithm in the example proceeds until the candidate ⟨B, U1, 8⟩ has been fetched for the first query triple. At this point, the candidates for the second triple will be again ⟨B, U2, Y⟩ and ⟨B, U2, Z⟩. These candidates have been fetched already in a previous iteration. Thus it is useful to keep caches of the fetched candidates.

We do this per query triple, and per used DHT key. By this, the second time the pattern ⟨B, U2, *⟩ is used, no more communication is needed to retrieve the candidates. This already saves a great deal of communication.

However, as we can see from the example, it might be useful to have a kind of look ahead when fetching the candidates. In step 2 and step 3, we have contacted the node responsible for URIref A twice in short succession to ask for the next candidate and retrieved the triples ⟨A, U2, X⟩ and ⟨A, U2, Y⟩. Thus if we had already fetched both candidates we would have saved a message.

In the following, we present a scheme for combining look ahead and caching of candidates which is suitable for all situations that can occur during the top k query evaluation.

To get a clear picture of the various situations, look at the second query triple in our example, which is ⟨?v1, U2, ?v2⟩. The subject of the triple is a variable. As this variable occurred already in a previous triple of the query, it will always be *bound* to a specific value because the candidate that we chose for the first triple determines ?v1's value for the current iteration. The bound value might be different in a later iteration. However, only values that occur in the candidates for the first triple are possible. The predicate is U2. Thus it is a *fixed* value that is never changed during the query evaluation. Finally, the predicate is a variable ?v2, which has not been bound so far during the evaluation. Thus at any time the `evalTopK` algorithm enters its recursion for the second triple, the loop will range over all existing triples which match the pattern ⟨*current value of ?v1*, U2, *⟩.

Thus for each of the three elements of a query triple, there exist three possibilities: the element can be a **fixed** value, a variable which is **bound**, or an **unbound** variable. Furthermore, we have to use one of the elements as DHT key. Here, only fixed elements or bound variables are useful, as we need a precise value in each iteration. In our algorithm, we choose the element which will be used as DHT key only once when a query triple is encountered for the first time. Thus, the caching and look ahead strategy can rely on the same pattern.

We create and use an individual cache for each of the key values. Assume we choose to use the subject ?v1 as DHT key for query triple 2. This means that we have two distinct caches for the values A and B, as these are the values occurring for ?v1 when looping over the candidates for the first query triple.

Each triple can be split into one component which defines the key of the DHT and two remaining components. If possible, we choose a fixed URIref or literal as DHT key, otherwise we iterate over all possible candidates of a variable. In the example, the subject served as DHT key, and predicate and object served as the remaining components. The latter ones were a fixed value and an unbound variable, but this does not need to be the case. In general we can encounter six different cases, where the two remaining components are:

1. two unbound variables,
2. an unbound variable plus a bound variable,
3. an unbound variable plus a fixed URIref or literal,
4. two bound variables,

5. a bound variable plus an fixed URIref or literal, and
6. two URIrefs / literals

A component is bound if it consists of a variable that was seen before in a higher recursion level, unbound with a variable that occurs for the first time, or fixed if it is a URIref or literal. The binding of a variable is the known candidate set, where the variable was found the first time. For each of these cases we define a specially optimized type of cache. These caches are depicted in figure 4.23.

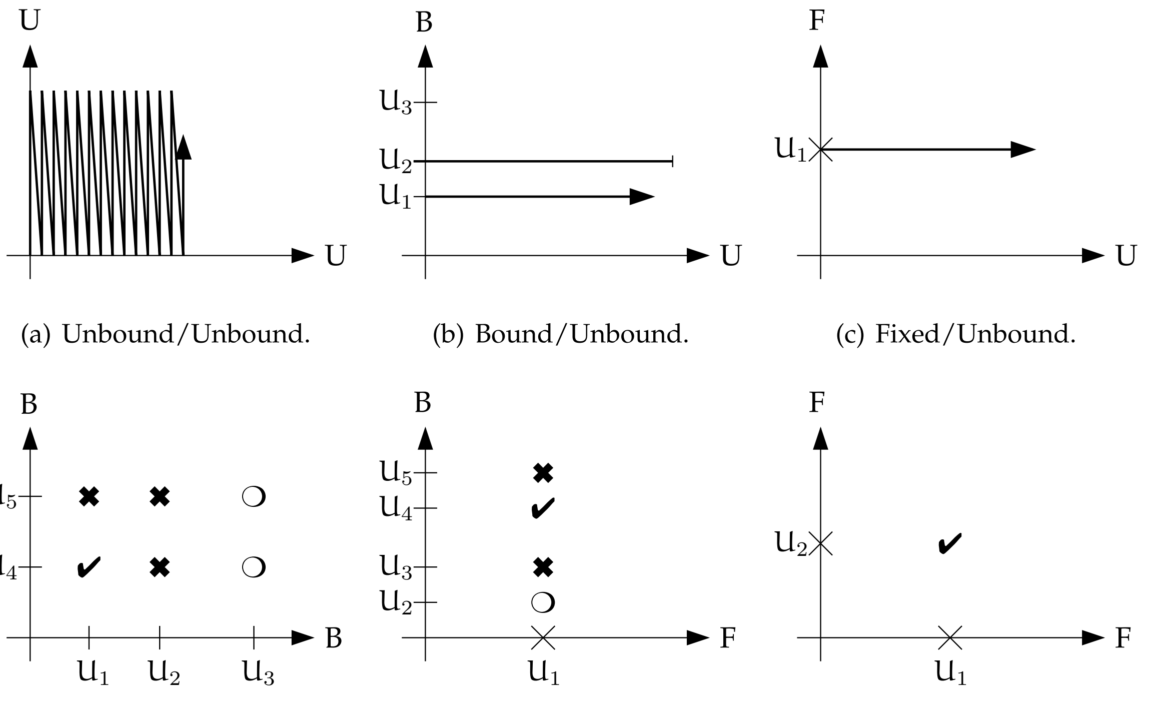


Figure 4.23: Cache Types.

The caches have to query the next chunk of up to c triples for a query. The `getNext-Candidate` function then delivers these chunks triple by triple. For scalability reasons, the peer who delivers the triples does not store any state information, so the requesting peer is in charge of submitting the state along with the actual request. The state can consist of a set of markers, which define the last triples for which we know information already (cases 4.23(a) to 4.23(c)), or of the set of triples we want to gather information about (cases 4.23(d) to 4.23(f)).

The simplest cache for fixed/fixed components (see figure 4.23(f)), which occur if a RDF query contains a triple with three URIrefs, does a simple lookup without look ahead. The state of the cache can be "triple exists in RDF model graph" (represented by a check mark in the figure), "triple does not exist in RDF model graph" (cross), or "unknown whether triple exists in RDF model graph" (circle).

For fixed/bound component pairs (see figure 4.23(e)) the caching is simple as well. A peer requests a chunk of triples by specifying the fixed component and a set of candidates for the bound component for which it wants to retrieve the state.

For bound/bound components (see figure 4.23(d)) we build up a request containing a set of unknown combinations of already known values for the bound variables.

The fixed/unbound cache (see figure 4.23(c)) is similar to the fixed/bound cache, except that it is sufficient to request the next c elements starting after a given position. Therefore, we submit the fixed element and the last inspected value for the unbound element as the request.

The bound/unbound cache (see figure 4.23(b)) extends this by storing and submitting markers for the last known elements in several rows. The peer who processes a request starts sending triples at the first marker until c triples have been sent or continues at the next marker if the row (candidates) does not provide c triples.

For the unbound/unbound cache (see figure 4.23(a)) it is again sufficient to submit a single marker which determines the next triples to be delivered.

4.8 Ordering of the Query Triples

For the backtracking, we order the query triples. This applies both for the final evaluation in the exhaustive method, and for the top k evaluation.

In the exhaustive case, the triples can be ordered in arbitrary ways because we already have candidates for every triple. In our current implementation we order the triples by their candidate set size in increasing order.

In the top k case, the triple which contains the `ORDER BY` variable, is always the first triple. After that, we sort the remaining triples iteratively. In each iteration, all remaining triples are scored. Triples are scored according to their element types, which can be fixed, bound, or unbound. The scores are listed in figure 4.24. The score of a triple is the sum of the scores of its elements.

	fixed	bound	unbound
subject	9	3	0
predicate	2	1	0
object	9	3	0

Figure 4.24: Triple Ordering in Top k.

Thus a triple with a fixed subject or object is scored higher than those that do not have fixed elements, because here only a single node can be used to fetch the candidates. Unbound elements do not contribute to the score. Thus triple containing multiple unbound elements are not selected until more elements become bound.

As a heuristic optimization, predicates have a lower score than the subject or object because predicates typically have larger candidate sets. Because of this, even a fixed predicate gets a lower score than a bound object.

4.9 Summary

In this section, we have described the methods we have developed to tackle the challenges described in the introduction. We have described the general architecture of our system and introduced the query problem formally. Then we have focussed on three main areas: knowledge dissemination and reasoning, exhaustive query evaluation, and top k query evaluation. In the following chapter, we present an experimental evaluation of these methods.

5 Experimental Evaluation

In order to evaluate the algorithms and to compare the different versions, we have implemented a prototype of our system, called BabelPeers. The prototype can be used both for measurements and simulations. Real-life measurements give a better insight into the real behavior of the system, however they are limited in the number of available nodes. We use the PC^2 compute cluster ARMINIUS with up to 128 nodes for our experiments. Simulations complement the experiments by giving results also for networks with more than 128 nodes.

This chapter is organized as follows. In the first section, we describe our prototype. Then we explain the setup for the experiments and describe the test data in section 5.2. In section section 5.3 we give an overall evaluation of the performance of the prototype including dissemination, RDFS reasoning, and query evaluation. The following sections look in greater details at specific elements of the system. Section 5.4 targets the exhaustive evaluation. It compares the various SG methods and discusses the performance in different load situations. Furthermore, load balancing issues and the choice of Bloom filter parameters is discussed. The next section 5.5 deals with top k query evaluation, showing the effects of top k evaluation with an increasing size of the model graph, and the influence of the look ahead during the candidate fetching. Section 5.6 shows the influence of the triple dissemination during the query evaluation and the time needed for RDFS reasoning. The chapter concludes with a summary.

5.1 Prototype Architecture

Within this section, we look at the building blocks of the prototype. They are shown in figure 5.1. Each node participating in the network runs a process which connects to the network. In the diagram, this is the **P2P Node** block together with the **Pastry DHT Layer**. For the Pastry [97] Layer, we use the FreePastry [96] Java library. As we use the Common API [33] interface to the library (10), we can switch to any other DHT implementation also providing this API. Thus we are flexible with respect to the underlying DHT system.

The query client is a separate process which connects to the node via a networked interface. Therefore, it is not mandatory to participate in the p2p network in order to query it. As the operation of the query processor is completely symmetric, the query client can connect to any network node and will receive the same result, as long as the node is connected to the network.

Each node takes care of one or more local information sources, which are RDF files in our case. However, this can as well be interfaces to other information sources like

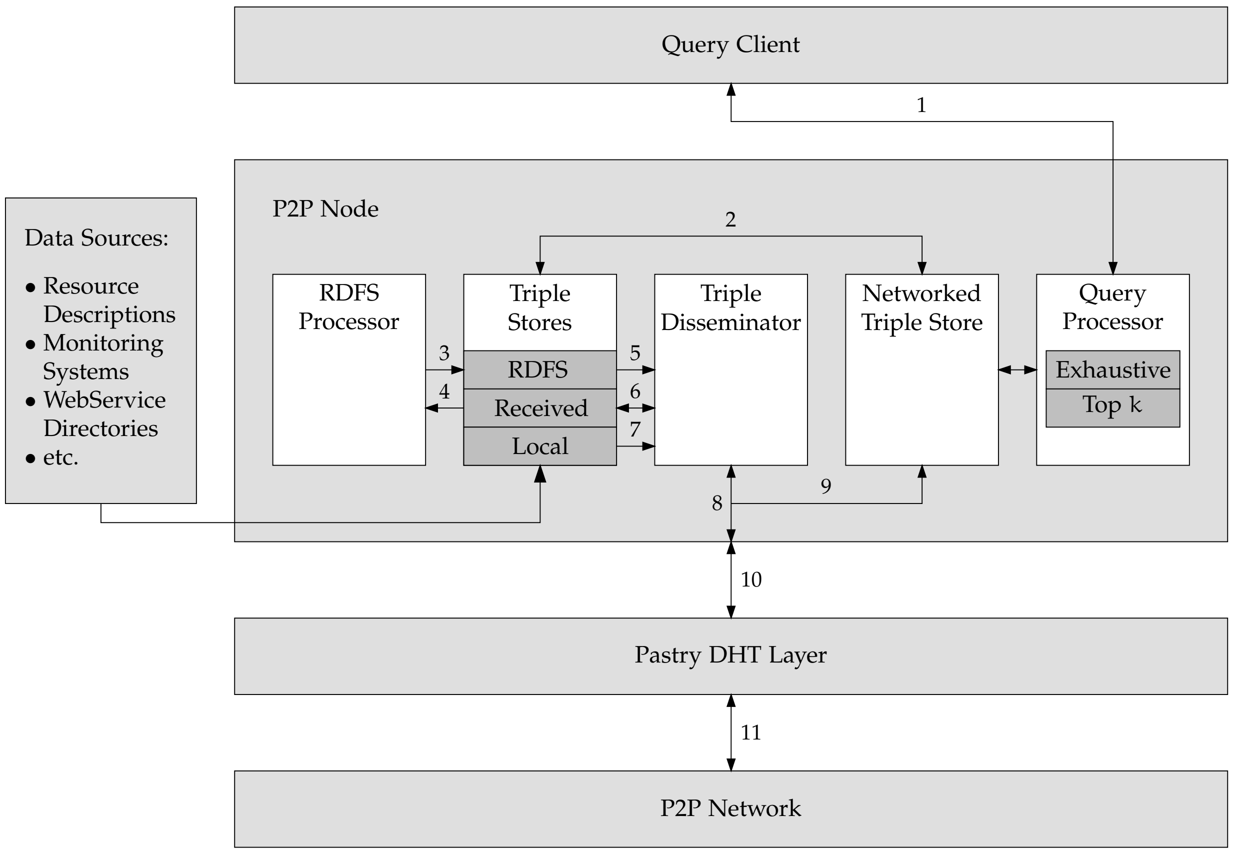


Figure 5.1: Prototype Building Blocks.

monitoring systems, resource information systems, web service directories, etc. As long as the information is encoded in RDF, it can be inserted here. For the soft-state update mechanism, we also need an expected life time for each triple or block of triples.

In figure 5.1, the local information is shown in the sub-block **Local** of **Triple Stores**. As described in section 4.3, each node also maintains two other triple stores for information received from the network and information generated through RDF Schema reasoning, called **Received** and **RDFS**, respectively.

No reasoning is done with the local knowledge, it is also *not* used during query processing. The triples are directly sent to the relevant nodes in the network. This is the job of the **Triple Disseminator**, a permanently running thread periodically checking the contents of the three triple stores (5-7). According to the distribution rules, it sends out the triples to the network, grouped into chunks for efficiency. The disseminator uses standard techniques from networking like a send window [105] for flow control. When it receives a chunk from another node's disseminator, it stores the triples in the **Received** Store (6).

The **Query Processor** has a networked interface where it can receive queries from the client (1). According to the system's settings and the query type, it chooses either the **Exhaustive** or the **Top** k module. When these modules need to access some triples, they contact the **Networked Triple Store**. Thus the Query Processor never looks up triples in the local node's store directly.

The Networked Triple Store marshals the requests from the Query Processor and sends them via Pastry (9,10) and the p2p network (11) to the responsible node. There, the symmetric thread receives the request, unmarshals it, and contacts the Received Triple Store (2) to fetch the result. The result, in turn, is marshaled and sent back to the original node. In some cases, the two nodes may be the same. In this case, the processing is done completely transparent through the loop back interface. However, the probability that a request can be served locally gets smaller the larger the network grows. As we target large networks, we do not care for this case with specific optimizations.

The **RDFS Processor** regularly checks the triples in the Received Store. If new triples have arrived (4), new RDF Schema rules might be fired. The Processor then executes these rules and stores the generated triples in the RDFS Store (3). From this store, they are grabbed by the Triple Disseminator and distributed to the network (5).

5.2 Experimental Setup and Test Data

We now look at the setup of our experiments, and describe the test data we use for the experiments.

5.2.1 Setup of the Experiments

We choose two setups for our experiments, see figure 5.2. In the first scenario called **N Query Processes**, we start both a p2p network node and a query client on each physical node of the cluster partition used for the experiment. The client loops permanently sending a batch of test queries. The whole network runs for an hour, and we calculate the overall throughput by counting how many queries are evaluated correctly during this hour.

In the second scenario called **Single Query Process**, shown on the right side of figure 5.2, only one of the nodes has a query process, the others are mostly idle, only serving requests from the node with the query process. By these scenarios, we can compare the behavior of the system under different load conditions.

For both scenarios, we enforce a uniform distribution of the Pastry node IDs over the peers in order to exclude random effects that typically occur with small networks. Consider for example a test run with 4 nodes. Random IDs could lead to a network where a single node covers half of the ID space. For larger networks, which are the target scenario of this thesis, the probability of having a good distribution is rising when assigning random IDs. Earlier experiments with random node IDs have shown that the results for larger networks are comparable, however that the random distribution leads to unpredictable results for smaller networks. This can either be resolved by running the experiment multiple times and averaging the results, or by using a uniform distribution. We choose the latter way because of the limited compute time available on the ARMINIUS cluster.

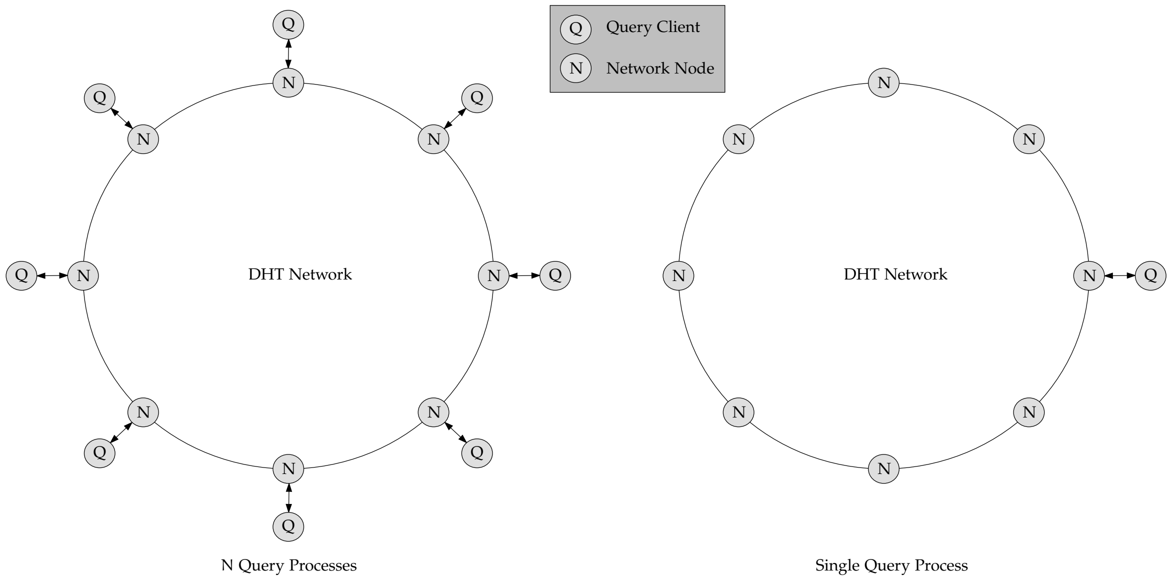


Figure 5.2: Setup of the Experiments.

5.2.2 Generation of Test Data

We have created test data reflecting the expected structure of RDF-based information in our scenario. The goal of the test data is to evaluate all aspects of the system. Thus we designed a flexible way to generate test data that includes complex schema information and queries that result in various access patterns.

We generate the test data in multiple steps. We start by generating a **class hierarchy** and a **property hierarchy**. These hierarchies are generated using the same algorithm, however with different parameters. The algorithm outputs a directed acyclic graph (DAG), whose elements are either classes or properties. The parameters determine how tree-like the DAG will be, which means up to what degree multiple inheritance is used. They further determine how deep the hierarchy is, i. e. how long the average path from an element to the top element is. See figure 5.3 for a sample hierarchy.

Using the class and property hierarchies, we generate instances and relations between these instances. The instances are typed over the classes and the relations between the individuals are generated using the properties from the second hierarchy.

The queries we use are generated as follows: We start by selecting a random subgraph of the current test set. This ensures that each query will have at least one match. In order to have queries of varying difficulty, we randomly replace URIrefs that occur

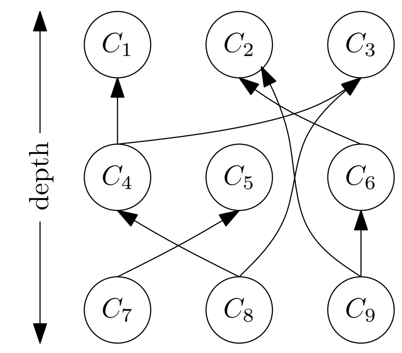


Figure 5.3: Class Hierarchy Generation.

in the subgraph by variables. During this process, we ensure that each URIref is replaced by a unique variable, however we do not always replace every occurrence of the URIref.

The characteristics of the test set, and the effects of RDF Schema reasoning, are shown in figure 5.4(a). We generated 1000 queries for the test set. The distribution of the number of matches for the queries is shown in figure 5.4(b). During the generation, we have checked that the query is no outlier in the sense that it takes too much time in the final phase of the query evaluation time. This can happen due to the random generation of the queries and the fact that the final evaluation is NP complete. Such a query would dominate the overall performance too much.

5.3 Overall Performance

In this section, we start the evaluation with an overview of the performance of the whole system. For this experiment, we used the N Query Processes scenario, with 1000 different queries sent constantly to each of the network nodes. We use an increasing number of nodes from 1, 2, 4, up to 128 nodes of the ARMINIUS cluster. The model graph of the test set is split up into N chunks, where N is the current network size. Thus each triple is stored initially only on one node, and then disseminated to the network. Furthermore, the RDFS reasoning is switched on so that new triples are generated during the run of the experiment.

For the evaluation of the results, we compute locally the answers after all RDFS reasoning. We compare the results of the experiments with these results and count only those results that are correct. Thus a query that returns only partial results is not counted in the throughput. We use evaluation strategy SG4/B, which uses Bloom filters both for the look ahead and the candidate fetching.

The result is shown in figure 5.5(a). Overall, we can see that the scalability of the system is very good. The individual speedup values are as follows:

Value	Base	with RDFS
Triples	110,599	231,864
Other statements	100,000	184,584
Avg. triples per URIref	29.6	62.1
Max. triples per URIref	10,000	46,681
Min. triples per URIref	6	13
Standard deviation	99.7	449.0
unchanged by RDFS reasoning		
Individuals	10,000	
Classes	200	
subClassOf relationships	262	
Properties	1,000	
subPropertyOf relationships	337	

(a) Model Graph.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1	56	610	10,180	4,791	165,600

(b) Query Matches.

Figure 5.4: Characteristics of the Test Data.

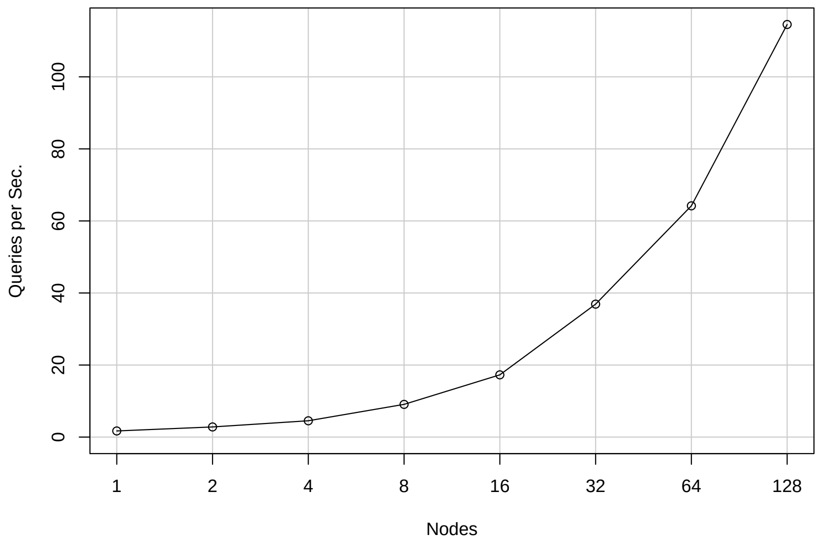
Nodes	2	4	8	16	32	64	128
Speedup	1.65	2.66	5.33	10.14	21.66	37.67	67.22

The top k evaluation generates a much higher throughput. This is clear as fewer results are generated, tapping smaller amounts of data in the network. The results are shown in figure 5.5(b). Here, the speedup looks different. As the top k method needs less CPU time for the subgraph matching, because fewer hits are searched, overall the network communication cost has a greater relative impact on the performance. This can be seen by looking at the speedup of the 2 node version, which is only 1.12. The following table lists all values:

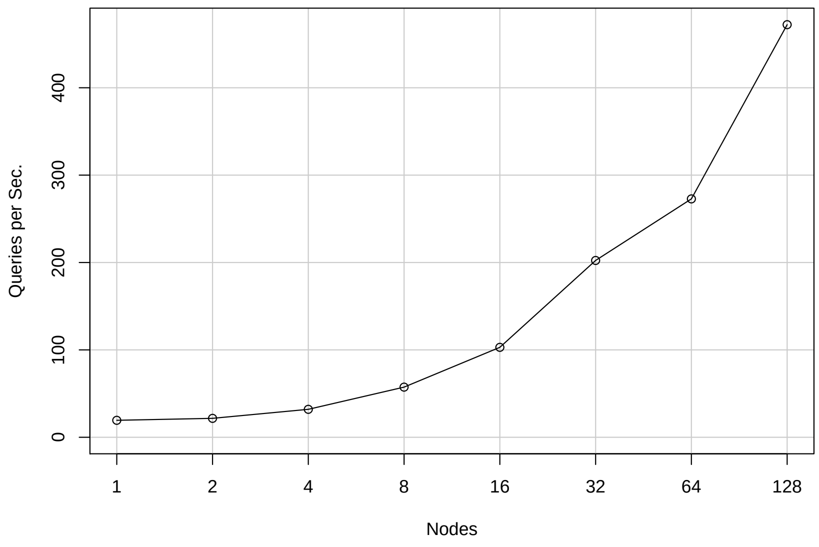
Nodes	2	4	8	16	32	64	128
Speedup	1.12	1.65	2.95	5.30	10.42	14.04	24.30

For larger networks, the figures get better. For example, we observe that the speedup increases by a factor of 1.73 between 64 nodes and 128 nodes. When comparing the 32 nodes run to the 64 nodes run, we see a slight break in the curve; the speedup only increases from 10.42 to 14.04. We have observed this behavior in various experiments with different algorithms (see e. g. the Single Query Process experiment shown in section 5.4.3) on the ARMINIUS cluster. Thus we conclude that it is due to the specific features of the network switch used in this system.

In the following sections, we go into details, analyzing these results and presenting the effects of the various methods we have introduced in the previous chapter.



(a) Exhaustive Evaluation.



(b) Top k Evaluation.

Figure 5.5: Overall Throughput of the Prototype.

5.4 Exhaustive Evaluation

To measure the performance of the different versions of the exhaustive query evaluation algorithm, we have run both simulations and real test runs on the ARMINIUS cluster of PC^2. In the following subsection, we explain the simulation results, followed by a discussion of the measurement results on the cluster. We both use the N Query Processes and the Single Query Process scenario. Then we report on simulations that show the effects of using various Bloom filter parameters. In the final subsection, we discuss load balancing issues during query evaluation.

Before diving into the results, we summarize our notation in the following table. In section 4.6, we introduced variants of the exhaustive evaluation with respect to details in the selection of the next key for the DHT. We call these variants SG1 through SG4. They are explained in figure 5.6. We further combine each of these versions with Bloom filters when actually fetching the candidates. This leads to another set of four versions called SG1/B through SG4/B.

Variant	Explanation
SG1	No look ahead during candidate fetch. Choose the key where the smallest number of lookup operations is needed during the next step.
SG2	Look ahead. Fetch the number of candidates each node would return, decide on the next key using these values.
SG3	Like SG2, combined with caching of the statistical values during the run of a single query.
SG4	Like SG3, but also encode the already known candidates using Bloom filters.

Figure 5.6: Variants of the Exhaustive Evaluation.

5.4.1 Simulation Results

Our simulation shares the query evaluation code with the prototype. However, when a message is sent via the network, it intercepts this message, records statistics, and answers the message using a simulated instance of the foreign node running on the same computer.

Thus we can report in detail on the number of messages sent, their size, etc. The main results are shown in figure 5.7. We have run the full set of 1000 queries over test set A using the Singe Query Process scenario. The presented figures are the accumulated values of these 1000 query evaluations.

The table lists various statistics collected by the simulation for each of the eight versions. The column *Msgs.* lists the total number of messages sent while processing the

Vers.	Msgs.	Bloom	B. Size	Sent	Recv.	Sum
SG1	14,706	0	0	3,881	204,782	208,663
SG2	300,526	0	0	79,343	112,800	192,142
SG3	51,810	0	0	13,674	46,734	60,408
SG4	35,948	3,819	32.8	10,052	133,015	143,066
SG1/B	14,706	2,276	68.3	4,401	37,012	41,412
SG2/B	300,526	1,842	18.2	79,861	91,530	171,390
SG3/B	51,810	1,842	18.2	14,192	25,470	39,662
SG4/B	35,660	4,711	41.8	20,179	10,515	30,693

Figure 5.7: Simulation Results.

queries. The following columns *Bloom* and *B. Size* show the number and accumulated size of the Bloom filters used. The *Sent* column lists the accumulated size of all messages sent during query evaluation containing triple requests or look ahead queries. The *Recv.* column shows the accumulated size of the returned result messages. The final column lists the overall bandwidth consumption during query processing. All messages sizes are given KBytes.

We now look at versions SG1 to SG4. First, we observe that version SG1 uses the largest messages. As no look ahead is done to determine where the smallest result set is expected, the algorithm often retrieves large sets resulting in a high bandwidth consumption. On the contrary, the SG1 version needs the smallest number of messages. Thus, routing in the DHT, which typically costs $O(\log N)$ forwarding steps, is needed seldom. Every other method needs more messages and thus more routing steps. Thus we observe a trade-off between the number of needed routing steps and the message size.

The SG2 versions perform worse compared to SG3, which is clear from the construction. They consume more messages and bandwidth during the look ahead without a better result in terms of the message size. As the look ahead messages also contribute to the overall message size, the overall message size is larger than SG3's. The same holds for version SG2/B. In the following, we will not take care of SG2 and SG2/B any more, as they are always worse compared to SG3 and SG3/B, respectively.

SG3 performs well, the accumulated message size is around 60 MByte, while it is about 209 MByte for version SG1. However, the number of messages is about three times as high, i. e. 51,810 compared to 14,706 for SG1.

From the theory, SG4 is not a very reasonable version. It uses Bloom filters during look ahead, but not during retrieval of the candidates. This results in large messages. However, they are at least about 32 percent smaller than those of SG1, while the number of messages is 2.4 times as large. It is interesting to reason why SG4 produces a smaller number of messages compared to SG3. It uses a similar caching mechanism, thus at a first glance, it should use a similar number of messages. However, SG4 guides the algorithm towards a better path through the query graph. The additional encoding of the known candidates for the other elements of the current candidate triple guides the algorithm to choose the triple which has the smallest candidate set *after removing the obvious contradictions*. This makes further variable candidate sets smaller, leading to a smaller number of messages. This way SG4 reduces the message number. However, due to the contradiction described above the message size is much larger than SG3's.

Now we look at the effect of using Bloom filters when fetching the candidates, i. e. the versions SG1/B to SG4/B. For all versions, the Bloom filters reduce the message sizes without increasing the number of messages[1]. The size of the Bloom filters is negligible small compared to the rest of the traffic. The only draw-back of Bloom filters might be the processing time needed to build and test the filters. However, this effect cannot be measured with our simulation, it has to be evaluated with a real-life test run, which is described in the next sub section.

We conclude from the simulations that there is a trade-off between the number of messages sent over the network and the overall bandwidth consumption. Without measuring the performance of a deployed network, it is difficult to predict which indicator affects the overall performance more seriously. On the one hand, it is clear from the theory that a large number of messages and thus a large number of routing steps will have a worse effect the larger the network grows. However, as the routing steps only grow logarithmical, the impact of this effect should only be visible for large networks. On the other hand, large bandwidth consumption will also be more serious the larger the network gets, as messages will have to be transferred over more links in the underlying network. Overall, versions SG1/B and SG4/B (highlighted in figure 5.7) are the most promising versions, the former having the smallest number of messages, and the latter having the smallest bandwidth consumption.

5.4.2 Test Runs

In order to better understand the effects of the various indicators we deployed the system on the ARMINIUS cluster at PC^2. We used up to 128 nodes from the system, and measured the overall throughput.

In the first test run, we repeated the experiment shown in section 5.3 using different SG versions. As SG2 and SG2/B do not compete well, we left these variants out. The result is shown in figure 5.8.

Versions SG1 and SG4 do not scale well to a larger number of nodes. Both versions even show an decreasing performance for larger networks. This might have various reasons. For SG1, it is most likely due to the high bandwidth consumption. For the SG4 version, both the number of routing steps needed and the bandwidth consumption is

[1]The number of messages for SG4/B is slightly smaller than the number of messages for SG4. This is due to the random generation of the hash functions used within the Bloom filters. They may lead to slightly different SG4 results due to a different number of false positives. This leads in rare cases to slightly different decisions during the candidate fetch.

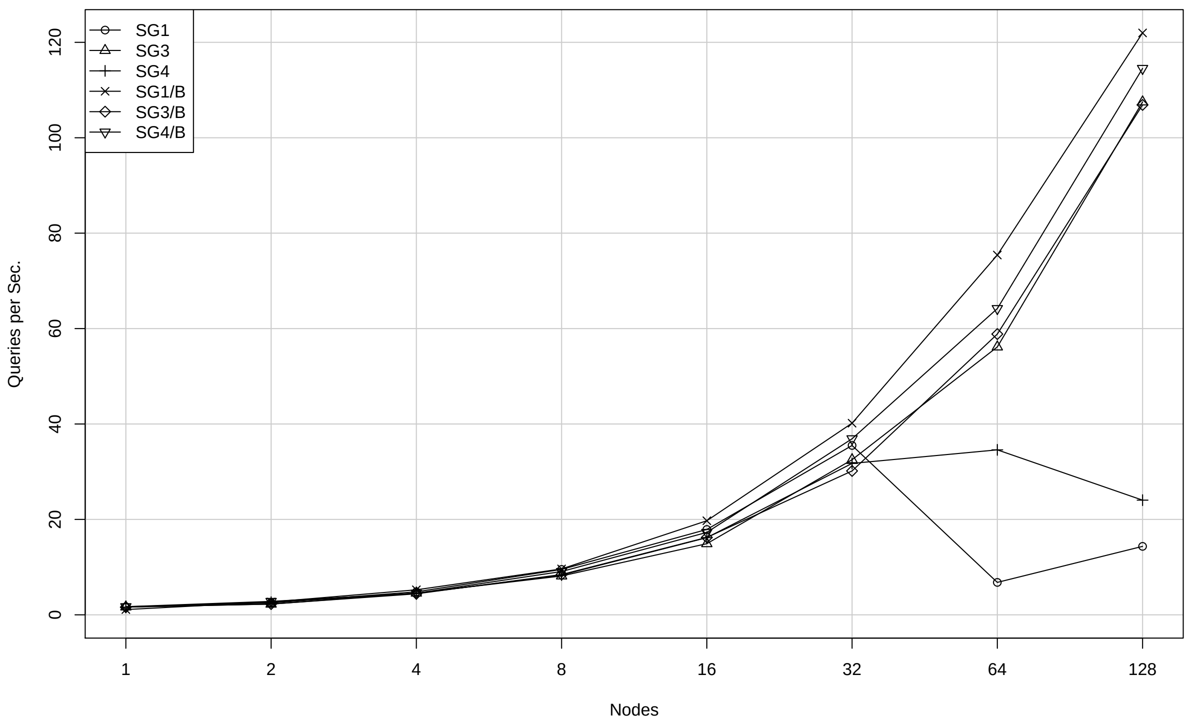


Figure 5.8: Comparison of the SG Versions.

high. However, also load balancing between the nodes plays an important role. We will investigate this issue in section 5.4.6.

SG1/B performs very well. In the given scenario, it is the best version. This indicates that indeed the bandwidth consumption is the problem with SG1, because the access patterns of both versions are identical.

SG3 and SG3/B show similar performance, which is interesting. As the number of messages between these versions is identical, the comparison shows the influence of reducing the bandwidth consumption from 60 MB down to 39 MB. Thus in our scenario of up to 128 nodes, this influence is very small. SG4/B is better than SG3/B, reaching 94% of the performance of SG1/B.

Overall, we can see that indeed the two versions SG1/B and SG4/B that were most promising when looking at the simulation results are the best performing versions. The test runs show that Bloom filters are important to achieve good performance. Also the look ahead is a good means to improve the performance. When combining both methods, we observe a slight decrease in performance. We reach about 94 percent of the performance of the best version SG1/B. However, using the look ahead has many advantages as it can e. g. be used to implement load balancing when nodes include their current load into the answer to the look ahead message. Thus we think that both

SG1/B and SG4/B are promising versions, SG1/B having the best performance and SG4/B offering more flexibility.

5.4.3 Single Query Scenario

We now look at the performance when we have only a single query process, i. e. the Single Query Scenario from figure 5.2. This scenario is interesting because it shows the individual performance of the query evaluation while removing effects that are either due to an overload of the network or an individual overload of some node.

As in the previous section, we evaluated six versions, leaving out only the versions SG2 and SG2/B. The results are shown in figure 5.9. The figure shows the median of the individual query evaluation time using these SG versions.

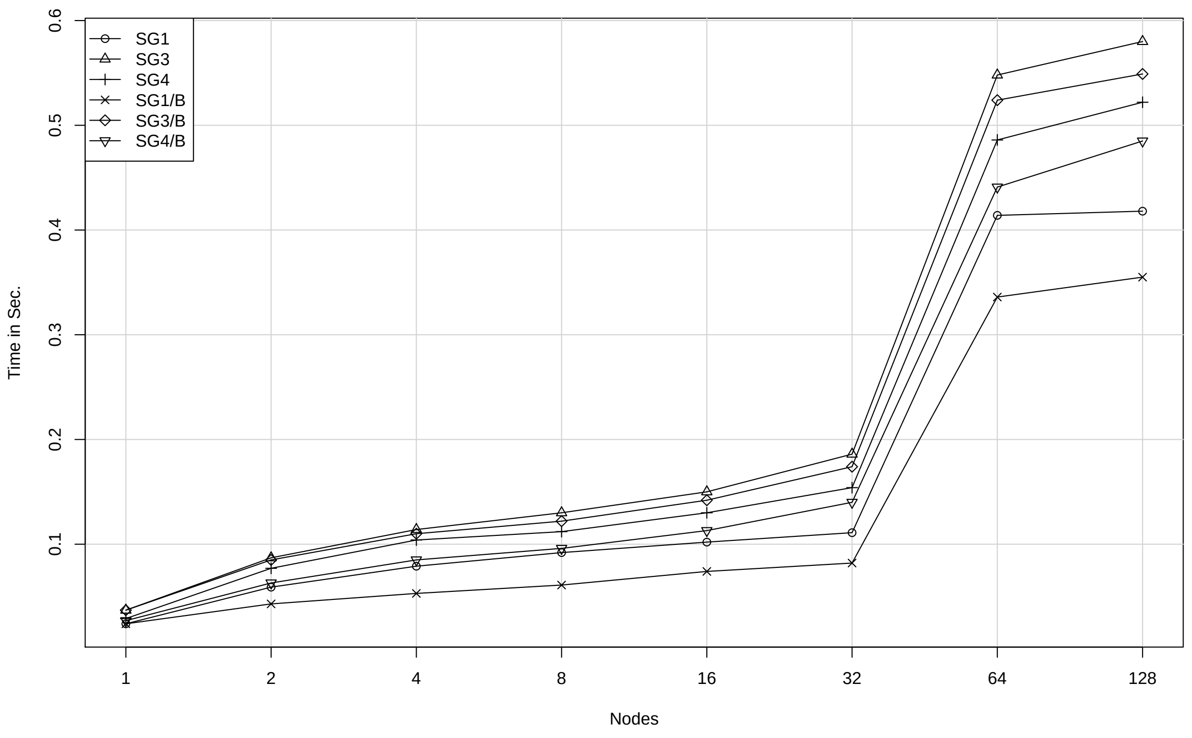


Figure 5.9: Query Evaluation Time in the Single Query Scenario.

For all versions, the query evaluation time increases with an increasing number of nodes. This shows the effect of the DHT routing mechanism that gets the more expensive the more nodes are involved. The increase is moderate, with a steeper increase from 32 to 64 nodes. From 64 to 128 nodes, the increase is modest again. We assume that the steeper increase is due to internals of the network switch of the ARMINIUS cluster. The same effect can be observed in figure 5.5(b).

The best version is still SG1/B. The second best version is now SG1. The SG3 versions are the worst, while SG4 and SG4/B are in the middle, performing similarly. We conclude that in scenarios where both the nodes and the network is lightly loaded, also the simpler SG1 version performs good. It wastes bandwidth, however is saves processing time due to not using any sophisticated methods like Bloom filters. On the contrary, we conclude that the sophisticated version SG4/B is more important the more loaded the network and the nodes are. Thus it is also more important the smaller the network capacity is.

5.4.4 Performance Factors During Exhaustive Evaluation

These experiments show that it is quite difficult to predict the performance of p2p query evaluation strategies. Multiple factors determine the overall performance. Following the path of an individual query, we identify these factors:

1. The query is sent from the query client to the p2p node which will process the query. In our experiments we minimize this time as the query client and the p2p node always reside on the same physical host.

2. In the candidate generation phase, we identify the nodes which store relevant information, and query these nodes. During this phase, multiple messages are sent:

   a) The target node of each message is determined by the DHT routing mechanism. This means, that the message travels along multiple nodes and is forwarded. The logarithm of the overall size of the network is the dominant factor for the number of routing steps. Additionally, the congestion of the network links and the load of the forwarding nodes are relevant.

   b) Once the message has reached its destination, the time to build the answer message on this node counts. This is determined by the type of message, the size of the result, and the load (both I/O processing and CPU load) of the node contributes to the performance of this step. Additionally, the use of Bloom filters influences the CPU time needed to generate the answer.

   c) Now the answer message is returned back to the first node. Here, no DHT routing is necessary, as the address of the node is contained in the message. Only the congestion of the network links between these two nodes is relevant.

   d) The result is now integrated in the candidate set. This consumes the more CPU time the larger the result is.

3. After collecting all candidates for the query, the final evaluation algorithm is executed completely local. Here the speed and load of the local host is relevant. The CPU time consumed here might affect the node's behavior while answering concurrent requests for triples from other nodes.

4. Finally, the result is transferred back to the query client. Here, the size of the result is relevant. As the query client is local, no network bandwidth is needed at this stage.

Our strategies influence these factors in different ways. We observe that there are severe differences between the performance of individual queries and the overall performance when running queries on each node. The strategies generate different *access patterns*, leading to a higher load peak on individual nodes than others, see section section 5.4.6.

5.4.5 Influence of Bloom Filter Parameters

We have seen in the previous sections that Bloom filters are important for a good performance of the exhaustive query evaluation. In this section, we look at the question how the parameters for the Bloom filters are chosen. As discussed in section 2.4, both the number of bits per stored element m/n and the number of hash functions k influence the possibility of false positives for the Bloom filters.

In our scenario, this leads again to a trade-off. On the one hand, a low rate of false positives leads to small messages and thus a low bandwidth consumption. On the other hand, the more bits per element are used the larger the Bloom filters get, wasting bandwidth. Additionally, more hash functions might lead to a higher CPU load of the nodes. This could then influence the throughput of these nodes.

We present simulation results to look at the trade-off between gained bandwidth and lost bandwidth, and processing time. We chose various parameter combinations from figure 2.18 with reasonable rates p of false positives. For each value of m/n from two up to seven we chose between two and four values for k which have the best false positive rate for this m/n value. The chosen parameter versions are shown in figure 5.10.

Version	m/n	k	p
1	2	1	39.3%
2	2	2	40.0%
3	3	1	28.3%
4	3	2	23.7%
5	3	3	25.3%
6	5	2	10.9%
7	5	3	9.2%
8	5	4	9.2%

Version	m/n	k	p
9	5	5	10.1%
10	7	4	3.6%
11	7	5	3.5%
12	7	6	3.6%
13	9	4	1.7%
14	9	5	1.4%
15	9	6	1.3%
16	9	7	1.3%

Figure 5.10: Parameter Combinations for Bloom Filters.

We look at the effects in version SG1/B and SG4/B, as these are two extremes. SG1/B does not have any look ahead, which leads to large result messages. Thus here the

Bloom filters are crucial to reduce the size of these messages. In this version, the trade off is immediate. The larger the Bloom filters, the larger are the query messages, and the smaller the result messages. On the other hand, for SG4/B, we have seen that already SG4 performs much better than SG1. In this case, the gain through Bloom filters is smaller than the gain when comparing version SG1 with SG1/B. However, the quality of the Bloom filters also determines the path through the query graph in SG4. Thus it is possible that a false positive rate which is too high leads to a worse path, which again leads to larger messages later on.

The simulation results are shown in figure 5.11. For each value in the curves, we have run the test queries ten times and averaged the results to get rid of effects due to the randomly chosen hash functions. We plot for each version (1 up to 16 as numbered in figure 5.10) the probability of false positives, the accumulated size of the sent messages and accumulated size of the received messages, and the simulation runtime. Each value is plotted as a percent value compared to the largest value of the curve. The darker vertical bars indicate the border between two parameter versions where the value for m/n increases. The curve "Prob" corresponds to the values computed in section 2.4 as shown in the p column of figure 5.10.

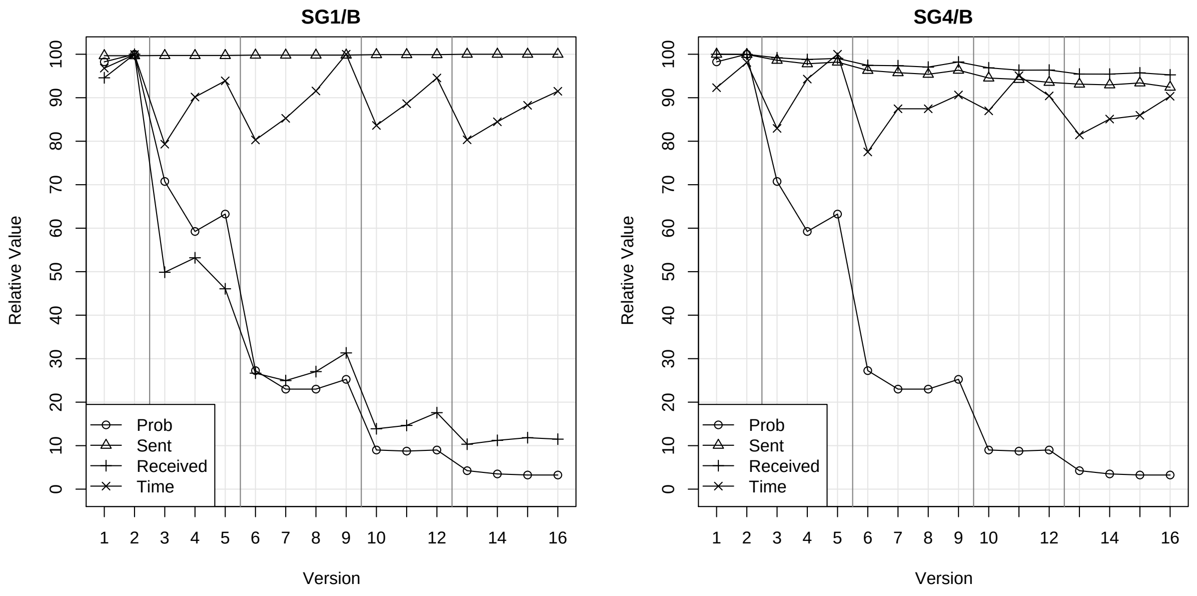


Figure 5.11: Simulations of various Bloom Parameters.

We first look at SG1/B and the size of the answer messages. As we expect these messages to be smaller with smaller false positive rates, we expect this curve to follow the shape of the probability curve. Indeed, the curves look similar. The size of the sent messages (i. e. the query messages containing the Bloom filters) does not change significantly. A closer look at the values reveals that it rises each time m/n increases, as expected. However, we see that for SG1 the differences are negligible. The CPU time consumed by the simulation is an indicator on how the additional hash functions used affect the processing time of the queries when k increases. We see that the CPU time increases with increasing values of k while fixing m/n.

Now look at SG4/B. Here the curves are much smoother. Overall we see that the influence of the Bloom parameters is smaller here. Both the size of the sent messages and the received messages only differs about five to eight percent. Additionally we see that SG4/B does not have the mentioned trade-off: also the size of the query messages gets smaller with smaller false positive rates and larger Bloom filters. This is due to the better path and more accurate look ahead values.

Overall we conclude that a good choice for the Bloom filter parameters is important. It is much more relevant for SG1/B compared to SG4/B. This is due to the look ahead in SG4/B which reduces the overall size of the messages, which makes the influence of the false positive rates smaller. For the results presented throughout this chapter, we used parameter version seven ($m/n = 5$, $k = 3$).

5.4.6 Load Balancing Issues

Load balancing is an important issue in our system. Optimal performance can only be achieved, if all nodes take the same share of the overall load. In this section, we show that the performance of SG1 and SG1/B is also influenced by imbalances in the load distribution, and that version SG4/B is doing better.

We distinguish between **storage load** and **query load**. We define the storage load to be the number of triples a node is storing. The query load is determined by the aggregated processing time for every messages a node receives. These types of load can be totally unrelated. A node might just store a few triples. However, if these triples are popular then it might suffer an extraordinary query load.

To see how the load is distributed in our system, we ran various simulations. First, we look at the storage load. In order to better visualize the load and possible imbalances, we use Lorenz curves [77]. Look e. g. at the curve in figure 5.12. The aim of a Lorenz curve is to show which percentage of the nodes holds which percentage of the load. To generate such a curve, the nodes are sorted ascending by their load share, and the curve plots the accumulated share vs. the accumulated node count. To be comparable, both axes are given in percent.

The ideal Lorenz curve is the 45 degree line, which is always painted as a reference in the figures. It means that every node has the same share. As this is normally not the case, the curve bends down towards the x axis to some degree. The closer a Lorenz curve is to the 45 degree line, the more uniform is the distribution.

In figure 5.12 the **storage load** is analyzed. The line for 128 nodes shows that 80 percent of the nodes only store 70 percent of the triples. In the 1024 node scenario, we see that this share is reduced to 60 percent. We see that the storage load imbalance gets worse the more nodes are in the network. This is natural, as we have used the same test set for all curves, and thus we only have a limited set of URIrefs. As the data

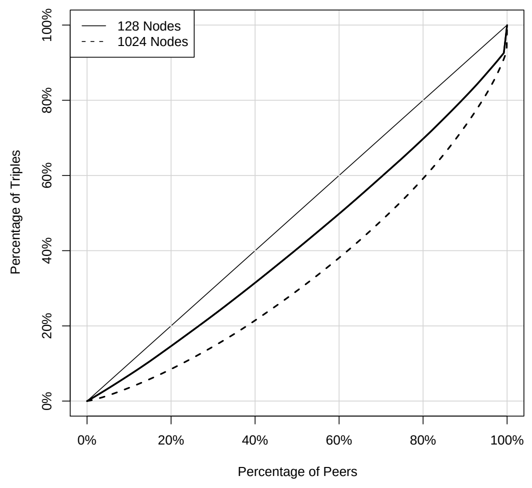


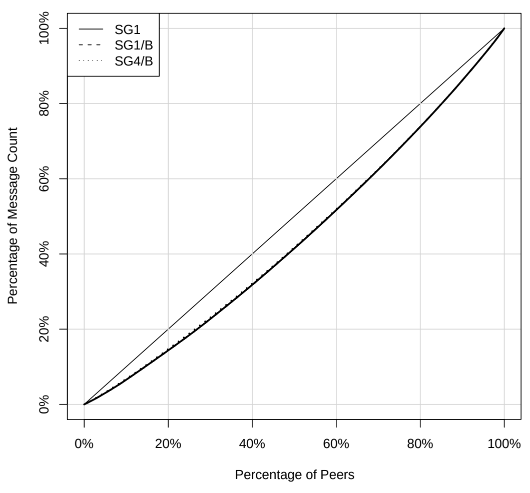
Figure 5.12: Storage Load Distribution.

related to a single URIref is always located on a single node, the triple distribution will eventually reach a state where each node only stores the data related to a single URIref. If there are more nodes than URIrefs, some nodes neither store any triples nor receive any query messages. In real life scenarios, the diversity of data increases the more nodes are participating in the network. This will diminish the storage load imbalances.

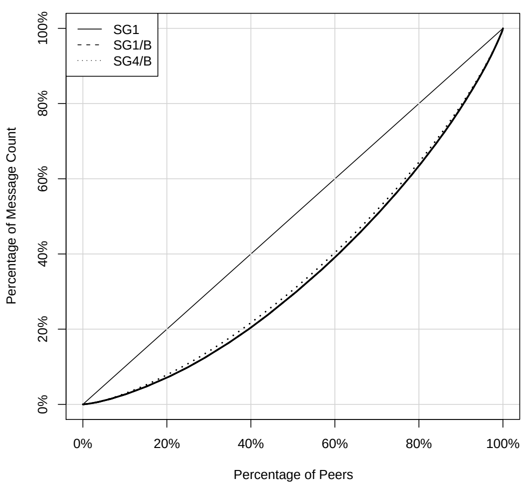
For our scenario, the **query load** imbalances are more serious, as they affect the overall performance of the system. We have simulated runs of our algorithm in networks with 128 and 1024 nodes. These simulations give insight how the query load imbalances behave with an increasing size of the network. We have used query evaluation strategies SG1, SG1/B, and SG4/B. We first look at how the number of messages each node receives is distributed. This is shown in figure 5.13. The distribution is quite good, and it is equally good for all considered SG versions. The imbalances rise as the network size rises. However, increasing diversity of data in larger networks lessens the problem.

We now turn to another indicator for the query load: the aggregated size of the messages received and sent by an individual node. This indicator is more relevant, as it quantifies the query load of a node better. The results are shown in figure 5.14. We observe two facts. First, the versions SG1 and SG1/B produce much larger imbalances compared to SG4/B. Second, both the imbalances and the differences between the SG versions increase with an increasing network size. Especially SG1 has huge imbalances. In both network sizes, only a very small fraction of the nodes manages half of the overall traffic.

In the previous sections we have compared the performance of the SG versions in two different scenarios. In the Single Query Process scenario, SG1 performed quite good.

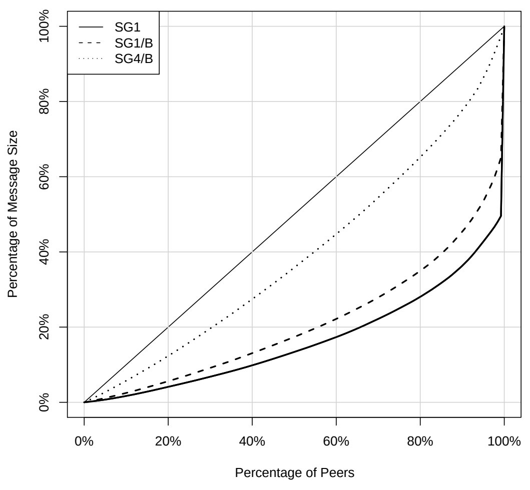


(a) 128 Nodes.

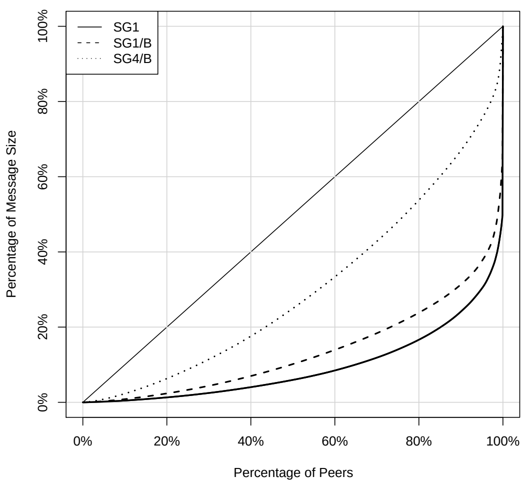


(b) 1024 Nodes.

Figure 5.13: Distribution of Number of Messages Received per Node.



(a) 128 Nodes.



(b) 1024 Nodes.

Figure 5.14: Query Load Distribution.

However in the N Query Processes scenario, it showed serious problems. Also the winning margin of SG1/B over SG4/B is larger in the Single Query Process scenario. We conclude that this is due to the different load balancing behavior of these versions. We furthermore conclude that SG4/B will be more beneficial the larger the network grows.

So far, we have ignored heterogeneities in the network nodes. Here, the versions that include look ahead – especially SG4/B – have a clear advantage. The look ahead value can be used to also include the current load of the queried node. Thus the client node can decide its path through the query graph not only upon the expected triple count, but also upon the load of the target node. This allows to better cope with heterogenous network equipment, resulting in a more robust system.

We summarize that especially query load balancing is an important issue in our scenario. The look ahead mechanism we implemented improves load balancing and provides further options to include steering of the query evaluation based upon the current load situation.

5.5 Top k Evaluation

Now we look at the top k query strategy. In the first subsection, we look at the performance of top k compared to the exhaustive evaluation. In the second subsection, we discuss the influence of the look ahead parameter during the top k evaluation.

5.5.1 Performance Compared to Exhaustive

In section 5.3, we already saw that the top k strategy performs well, delivering a much higher performance than the exhaustive evaluation. In figure 5.15, we directly compare the two curves to show the performance gain when using top k evaluation.

We claim that the top k evaluation is especially useful when queries return large result sets. Thus we additionally use three test sets which have few classes and properties combined with an increasing number of individuals and relationships between these individuals. The test sets consist of 100,000, 500,000, and 1,000,000 triples, and were complemented by a set of 100 test queries. The queries deliver more and more results the larger the model graph is. We ran a simulation to compare the number of triples during the evaluation of each query using the top k evaluation and the SG4/B exhaustive evaluation.

The result is shown in figure 5.16. The plot shows the ratio of the aggregated number of delivered triples in the top k strategy to the aggregated number of delivered triples in the SG4/B strategy for each individual query of the set of 100 queries. It compares these ratios for the three different test sets. We can see that with the smallest test set of 100,000 triples, there are still some queries that request more triples during the top k evaluation, resulting in a ratio greater than one. Already with 500,000 triples, every query requests smaller amounts in the top k scenario. Furthermore, we see that the factors are getting smaller the larger the model graph grows. This confirms our initial assumption that queries with large result sets benefit especially from the top k strategy.

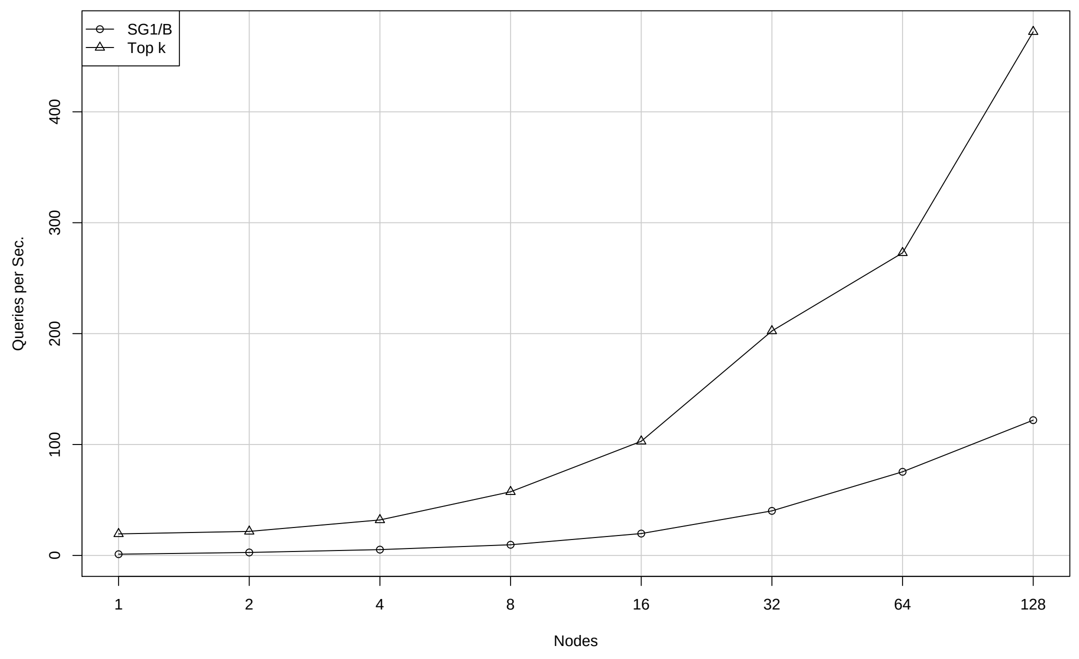


Figure 5.15: Top k Performance compared to Exhaustive.

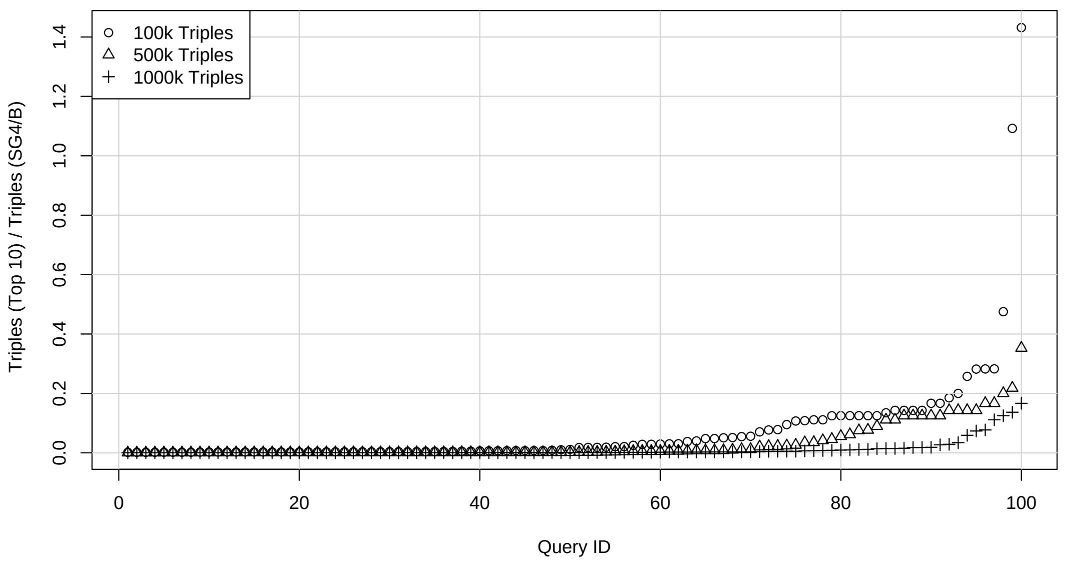


Figure 5.16: Top k Gain with Different Model Graphs.

5.5.2 Parameter Choice for Look Ahead

During the description of the top k mechanism in section 4.7, we introduced a combined mechanism for caching and look ahead that fetches triples needed during the top k evaluation in small chunks. We now discuss how to choose the size of these chunks, called the look ahead parameter.

It is clear that a larger look ahead value will lead to fewer messages which are individually larger. However, the larger the look ahead is, the higher is the possibility that triples are fetched that are never used during the evaluation.

To understand this issue, we ran simulations with two test sets of 60,000 and 600,000 triples. We recorded the aggregated sizes of the sent messages, the received messages, and the total number of messages. Look at figure 5.17. We increased the look ahead parameter in each simulation by factor 2, ranging from 1 (which means no look ahead) up to 1024.

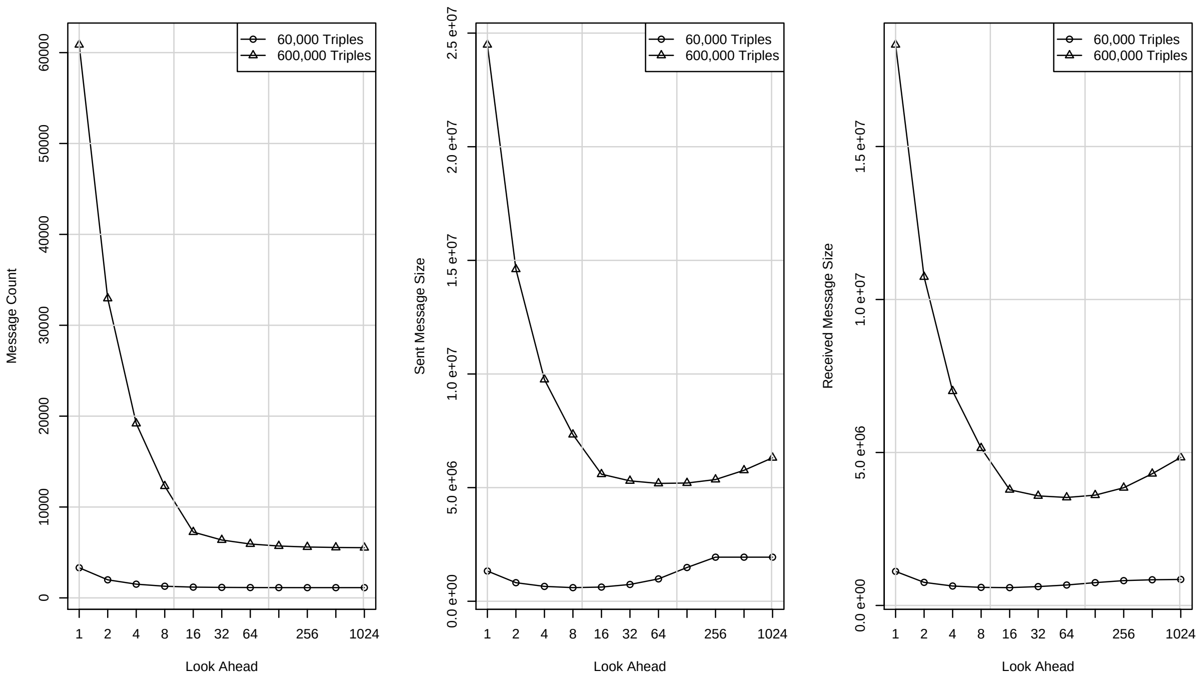


Figure 5.17: Simulations of Look Ahead Values for Top k.

First, we look at the number of messages sent. This indicator becomes smaller the larger the look ahead is. From this point of view, a large look ahead value is desirable. However, as we argued above, we might then fetch too many triples. This can be seen in the curves for the sent and received message sizes. Both curves show that the message sizes initially shrink when applying a look ahead. This is due to the reduced overhead when sending fewer messages. However, with a look ahead value too large, the message size rises again. This point is reached earlier with a smaller model graph. For the experiments presented in this chapter, we chose to use a look ahead of 40 triples.

5.6 Triple Dissemination and Reasoning

The method presented in section 4.3 to disseminate the triples puts load on the network by generating distribution messages. We are interested in measuring the effect of this load on the query processing performance. Further, we are interested to measure the time until every triple has reached its final destination. This is especially interesting when combined with RDF Schema reasoning, because here multiple iterations are needed until every new triple has been generated.

To see the effect of dissemination, we use the results shown in figure 5.8 as baseline. These results were generated with distribution and RDFS reasoning switched on, so that the dissemination load is included in this figure. Each triple was re-sent every 10 minutes, so each node generated network load in regular intervals during the experiment. As every node joins the network after some random delay, especially with large networks a nearly continuous load results.

We now present results on how the performance varies when the dissemination is disabled. In this experiment, we did the RDF Schema reasoning and triple distribution beforehand and switched the dissemination off during the query processing. The rest of the experiment is identical to the previous setup. The results are shown in figure 5.18. It shows the variants SG1/B and SG4/B. For each variant, the curve with dissemination is compared to the same curve obtained without concurrent dissemination.

We see that the curves look quite similar. The dissemination lowers the throughput, however only a few percent. The SG1/B version experiences a greater decrease. For SG4/B, the performance of the version with dissemination is 98.5 %, while it is only 91.7 % for SG1/B, both in the case of 128 nodes. This shows the SG4/B is less vulnerable to changes in the available network bandwidth, as it needs fewer bandwidth.

Additional to the network load generated through the dissemination, another factor causes the lower throughput of the versions with dissemination enabled. As initially no triple is distributed, no query can be answered correctly. We count only those queries for the calculation of the overall throughput that are correctly answered. Therefore, parts of the queries are not counted until the distribution and reasoning process is finished for the first time.

In the experiments, we have waited for three minutes until we start the query clients, thus giving the network a chance to arrange the routing tables and to start with the dissemination.

We now look in detail at this influence. It shows how long our RDF Schema reasoning mechanism takes to generate every triple. Look at figure 5.19. It shows which percentage of the queries are answered correctly on average during every second after

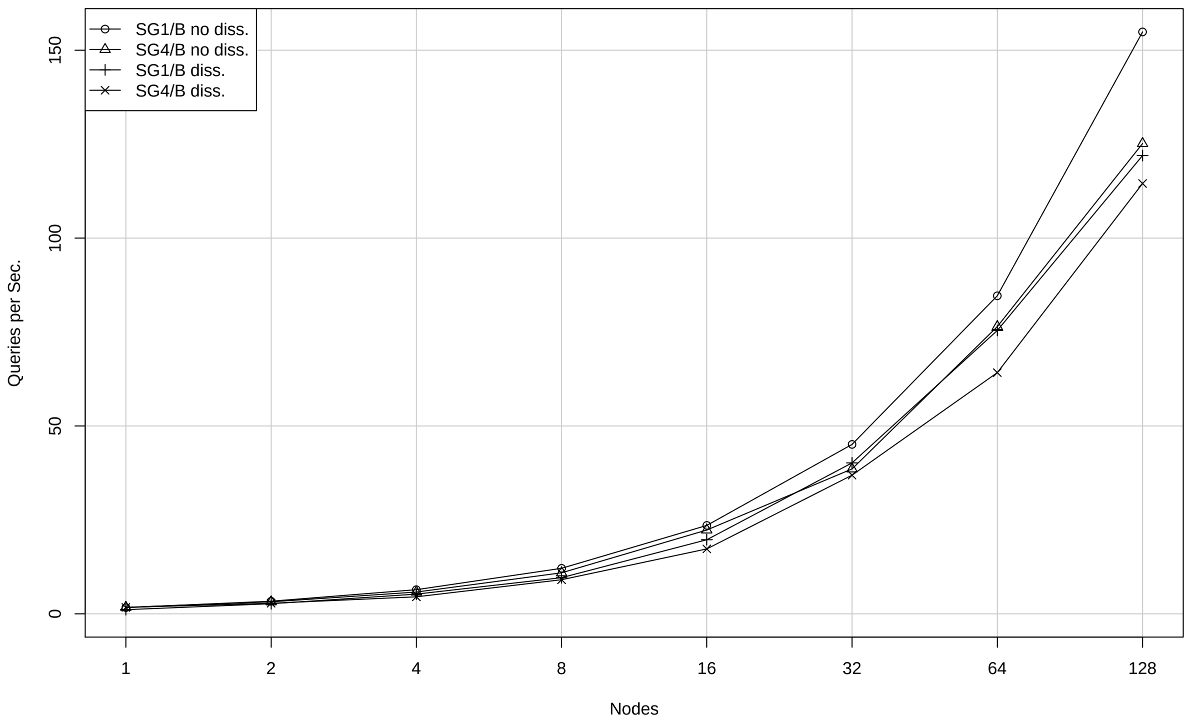


Figure 5.18: Overall Performance without Dissemination.

the initial three minute long phase. In this percentage, we also counted partial results. We can see, that during the first ten minutes, on average between 86 and 100 percent of the expected results have been delivered. After that phase, all conclusions were fully generated, thus every query had been answered correctly.

We conclude that the dissemination and RDF Schema reasoning is fast and does only influence the query processing moderately. Even with a fast update rate of 10 minutes for all triples in the network, the decline of the query performance is small.

5.7 Summary

In this section, we have presented numerous evaluations, both based on measurements on the ARMINIUS cluster of PC^2 and on simulations, that show the good performance of the methods presented in this thesis.

We have shown that both the exhaustive query evaluation and the top k evaluation scale well up to 128 nodes. We could further see that the top k evaluation is much faster than the exhaustive, as expected. Thus we can conclude that it is beneficial to use the top k strategy if not all results are needed.

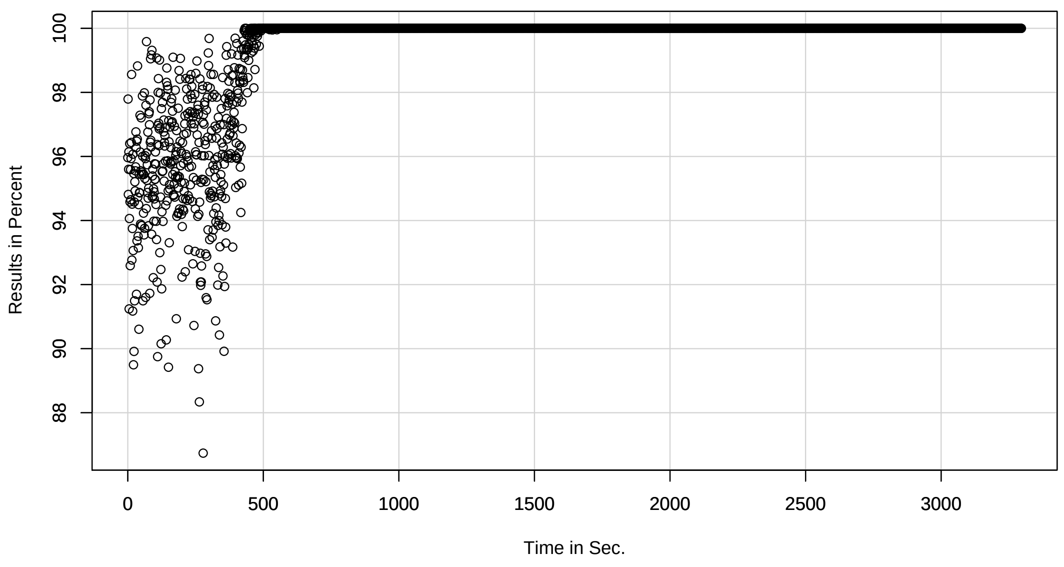


Figure 5.19: Duration of the RDF Schema Reasoning Process.

We have further discussed specific aspects of our methods in detail. We have compared the various SG versions of the exhaustive evaluation, and have discussed load balancing issues during the query evaluation. Further we have argued that the dissemination and reasoning process is fast and does not disturb the query evaluation.

Additionally, we have used simulations to discuss the selection of good parameter values for both strategies, especially the Bloom filter parameters in the case of the exhaustive strategy, and the look ahead parameter for the top k version.

6 Related Work

Within this section, we describe existing work which is related to this thesis. As we have integrated and extended work from different fields, we have to survey various areas of computer science.

We start by looking at the field that provides the primary application for this thesis: Grid computing. Here, we focus especially on resource discovery and matchmaking. As the Grid community has already realized the importance of semantics it has started the Semantic Grid initiative, which aims to transfer results from the Semantic Web community into the Grid. Thus we continue by looking at the Semantic Grid and Semantic Web, and briefly touch the issue of Semantic Web Services.

We then turn to the field of relational databases and discuss the overlaps and differences of our work with the large existing body of work on query processing and especially distributed query processing in relational database systems.

As we base our system on a p2p network, we describe the state of the art in p2p computing in the following section 6.4. Here, we concentrate on p2p based systems that deal with data management in p2p networks, with a special focus on those approaches that include aspects of semantic heterogeneity. This includes some approaches that also use RDF.

6.1 Grid Computing

The basic idea of Grid computing is the **virtualization of resources**. This means that users are no more required to explicitly know the services they are using. Instead, they describe their requirements and the Grid is in charge to find suitable resources which fulfill the users' demands. These requirements contain different sections. First, the technical specification of the resources has to be described. Second, the quality of service parameters like estimated run-time, bandwidth guarantees etc. are of interest. Last, other conditions are relevant, like the trustworthiness of the provider, the available support-options, etc., have to be encoded.

Thus, a versatile mechanism for describing both the resources and the requests is needed, together with a flexible architecture for matchmaking. One of the earliest matchmakers is the **Condor ClassAd-System** [90]. It employs the so-called ClassAd language, which uses mainly pairs of attributes and values, enhanced with a rich language for expressions over the attributes. The system has been extended to support simultaneous matchmaking of multiple resources as a single atomic operation [89].

The two major Grid middleware systems are the **Globus Toolkit** and the **Unicore** system. Within Globus, matchmaking is done by the MDS component (Monitoring and Discovery Services) [32]. MDS has been revised for the new Globus 4 Toolkit. It is now called MDS4 or **WS MDS** as it is implemented using Web Service technology. MDS4 consists of a set of services. The basic services are the **index service** and the **trigger service**. Both services collect data from different sources. The index service has a publish/subscribe based interface, while the trigger service can initiate actions according to the received data. Both are based on the **aggregator framework** which is responsible of collecting data and aggregating it. The index services can be organized hierarchically. Thus services which are closer to the root of the hierarchy contain information about a broader set of resources, however this information will be more out-dated and probably not that detailed compared to the services at the leafs.

For the description of the resources within a Globus based Grid, typically the **GLUE Information Model** [4] is used. It standardizes, as a UML model, a schema which can be used to describe computational resources. The main elements are the site, a service provided by a site, a computing element, a cluster, and a storage element. Each of these classes has a number of standardized attributes, and a number of further classes exists to describe e. g. access control, status information, etc. See section 2.1.5 on page 26 for a more detailed overview.

Unicore [36] has a different architecture. It was originally developed as a convenient way to access high performance resources. Thus, in the base distribution, it does not contain a broker. The resources are described using the **incarnation database** [110], which contains information about the physical properties of the cluster (number of nodes, number of processors, etc.) and the software installed. It also contains mapping information describing how a given program is to be executed on this specific resource. A job is described by the user via the Unicore client, which encodes it as an Abstract Job Object (AJO). Unicore matches the AJO with the incarnation database of the currently selected resource and indicates whether the job can run on the resource or not. However, no search for suitable resources is possible.

The **GRIP project** [91] has developed a resource broker which is capable of mediating between Unicore and Globus. It provides also brokering capabilities for Unicore-based systems. This broker has been further developed in the EUROGRID project [65]. In [23] Brooks and Fellows describe this broker and propose to build a scalable broker architecture by changing the EUROGRID/GRIP approach to allow hierarchies of brokers. The brokers in upper layers of the hierarchy drop detail information in order not to be overloaded.

6.2 Semantic Web and Semantic Grid

As Grids grow larger and larger, the heterogeneity of resources becomes one of the most prominent problems. A large-scale Grid that integrates resources from multiple administrative domains has to mediate between these resources. One way are standards, however, it is often impossible to agree upon a standard, even within an organization. The standardization problem gets worse in a multi-domain scenario. This especially holds true if the domain of discourse is changing rapidly. The other way is the usage of explicit semantics to allow machine-driven mediation and interoperability.

The Grid community has realized that semantic technologies are vital for the future success of large-scale grids. As Grids grow beyond pure networks of high performance resources, including any service and device as resources, this need becomes even more pressing.

The **Semantic Web** community [16] follows a similar goal. Its vision is to make the Web accessible for software agents. It has already developed numerous algorithms, tools and standards to support this aim. Outstanding standards are RDF and OWL. In the following two subsections, we describe RDF-related literature and provide a quick summary of the OWL language. In the final subsection, we describe the Semantic Grid effort.

6.2.1 Resource Description Framework

RDF is the foundation of our work. Thus we now look at literature related to RDF, starting with a paper that motivates why RDF is superior to XML, then focussing on RDF query languages, and finally looking at systems for storing and querying RDF data.

In [35], Decker et al. argue that XML is not sufficient for the Semantic Web. They compare XML with RDF and conclude, that RDF is much more suitable for achieving the goal of semantic interoperability. They explain that XML is only a *syntactic standard*, and XML Schema allows to define *grammars* using the XML syntax. When multiple parties are involved in an XML based communication, and these parties use different schemas, a mediation is only possible by re-engineering the domain model from the XML Schema. The whole translation takes part in the following steps:

1. Re-engineer the original domain model from XML Schema
2. Establish mappings between the entities in the domain model
3. Define translations for the XML documents, e.g. using XSLT

This procedure is significantly simpler and more natural with RDF, as it can directly encode the domain model, without a further translation step, which introduces ambiguities.

Furthermore, the paper explains that other ontology languages can be encoded in RDF Schema. This leads to a meta ontology which defines the modeling primitives of the ontology language using RDF Schema constructs. The ontology itself, and also the instances can again be encoded in RDF. This results in a three tier hierarchy, which uses RDF at each layer and which is capable of encoding knowledge from any ontological framework.

There are numerous **query languages for RDF**. We chose SPARQL [86] because it already became a W3C recommendation and is thus on the track to become a standard.

It includes and integrates many of the ideas of the other languages, which can now be considered predecessors of SPARQL. These older languages include RQL [67], SeRQL [20], and RDQL [99]. A comparison including other languages can be found in [50].

The ontology language RDF Schema [18] builds upon RDF. In [55] formal model-theoretic semantics for RDF and RDFS are given. In this paper, also a set of rules is given which is proven to be equivalent to the model-theoretic semantics of RDF Schema. In [19], Broekstra and Kampman explained that forward-chaining execution of these rules is indeed a viable approach to RDFS inference. If the dependencies between the rules are regarded, a huge number of tests can be avoided, resulting in a scalable algorithm. Based upon this work, they have developed the well-known **Sesame RDF engine** [21], which allows to store RDF triples and to perform RDFS reasoning over them. Sesame is a stand-alone system. The triples may be stored in memory or in a relational database.

Another leading RDF toolkit is the **HP Jena framework** [112]. It consists of a Java API and command-line tools for storing, querying, and modifying RDF and RDFS based data. Like Sesame, Jena is a stand-alone system which can store the triples in memory or in a database. However, the reasoning is always done in memory, and no forward-chaining is applied. Instead, reasoning is done on the fly during query evaluation.

6.2.2 Web Ontology Language

The Web Ontology Language (OWL) [101] defines another language on top of RDF and RDF Schema which allows to specify relationships between classes, properties and individuals in a finer granularity. However, the increased expressivity comes hand-in-hand with an increased reasoning complexity. The language consists of three sub-languages with an increasing set of features, named OWL-Lite, OWL-DL, and OWL-Full. The DL version of OWL needs a Description Logic [7] reasoner; the full version of OWL has been shown to be undecidable [5].

We now provide an overview of the features of OWL without claiming completeness. In **OWL-Lite**, classes and properties can be declared to be equivalent in addition to the sub class and sub property mechanism of RDFS. The nature of properties can be described as transitive, symmetric, reflexive, functional, or inverse functional. A property can also be defined as the inverse of another property. Range restrictions for properties in RDFS are always global; in OWL-Lite they can be restricted to specific classes. Furthermore, the cardinality of properties can be described, however only in the range of 0 or 1. For individuals their equivalence or dissimilarity can be described.

In **OWL-DL**, even more modeling options are available. Classes can be defined by enumerating the individuals that make up the class, and complex classes can be defined using the operators union, complement, and intersection. Complex class descriptions can also be used in sub class and equivalent class definitions. The cardinality constraints can be used to specify any cardinality. In general, OWL-DL is specified to support the features of modern Description Logic (DL) reasoners.

OWL-Lite and OWL-DL complement RDFS with a set of construct that allow more expressivity when modeling domains. However, they also restrict RDFS in that they enforce a clear separation between the instance layer and the class layer. That means, that e. g. a class cannot be an instance of another class. In **OWL-Full** this restriction does not apply, so that only OWL-Full can be considered a true superset of RDFS.

6.2.3 Semantic Grid

The Grid community has recognized already a few years ago that the developments started in the Semantic Web are also useful and relevant for the Grid. The larger Grids grow and the more heterogeneous they become, the more important becomes the meta data usage within the Grid and the semantics supported by this layer.

DeRoure et al. give an overview of the Semantic Grid in the article [34]. They give the following definition: "The Semantic Grid is an extension of the current Grid in which information and services are given well-defined meaning, better enabling computers and people to work in cooperation."

The basic idea is to use technologies from the Semantic Web in every layer of the Grid. In the paper, they list requirements for the Semantic Grid, among them

> "*Resource description, discovery and use.* The system must store and process potentially huge volumes of distributed content in a timely and efficient fashion."

and

> "*Information Integration.* The ability to make meaningful queries over disparate information stores, and to make use of content in ways which may or may not have been anticipated, requires interoperability of information."

DeRoure et al. further identify scalability as a main concern in the context of the Semantic Grid: "Underlying these requirements (at both ends of the spectrum) is the issue of scale. As the scale of the virtual organizations increases, so does the scale of computation, bandwidth, storage and complexity of relationships between services and information. Scale demands automation, and automation demands explicit knowledge."

The Semantic Grid community has already achieved advances, especially in the field of resource description and resource matching. Brooke, Fellows, Garwood, and Goble [22] use ontologies in the above described GRIP project to describe and match resources in a heterogenous Grid consisting of Unicore and Globus components. The emphasis of this work is on interoperability. Tangmunarunkit, Decker, and Kesselman [106] use Description Logics based ontologies in resource matching to gain the ability of mechanical reasoning, thus being able to match requests much more flexible than in former key/value based resource matching systems.

The issue of mechanical reasoning has been immersed by González-Castillo, Trastour, and Bartolini. In [44] they present their experiences on a prototypic implementation of a matchmaker, which is based on a Description Logic reasoner working on service descriptions in DAML+OIL, a predecessor of OWL. Core of this prototype is a matchmaking algorithm that finds appropriate matches by marking selected nodes of a service description tree. By doing this it provides a matchmaking mechanism through Semantic Web technology. However it does not address distributed storage of information.

In [48], Gunter and Jackson argue whether RDF is suitable for describing resources in the Grid. Their conclusion is that "RDF seems to be perfectly able to represent Grid resources, and offers a number of compelling reasons to consider its use for this purpose".

6.2.4 Semantic Grid and Services

Modern Grids are service based. However, Grids include some features that pose additional challenges on service oriented architectures. An important distinction between classical Web Services and Grid Services is that Web Services are stateless, while Grid Services have state. In order to consider state within services, certain additional features have been introduced to the Web Service standards, which has lead to the **Web Service Resource Framework** (WSRF) [40]. The main concept is called a WS-Resource, which is a combination of a stateless Web Service with a stateful resource. A WS-Resource is created using a factory, has a certain life cycle, and includes standardized addressing methods.

In this light, the problem of finding resources becomes equivalent to the problem of describing and finding services. That is why work targeting this area is also related to our work. Similar to the questions we have considered in this thesis, the **Semantic Web Service** community looks at ways to describe services, the formalisms needed to reason about service descriptions, and infrastructures to store and query service descriptions.

Main approaches competing in the field of service description are OWL-S [81], WSMO [95], and WSDL-S [88]. A comparison of WSMO and OWL-S can be found in [74].

In [76] Li and Horrocks analyze the usability of DAML+OIL and DAML-S (a predecessor of OWL-S) for service description. They prove the ability using a prototypic matchmaker basing on a DL reasoner. By simulating an e-commerce scenario they evidenced that the matchmaking process can be executed efficiently. However, their approach is limited by the scalability of the DL reasoner.

6.3 Relational Databases

We use the RDF data model and the SPARQL language for our system. An RDF predicate can be interpreted as a relation, and SPARQL is syntactically quite close to SQL. Thus it is interesting to compare RDF/SPARQL with relational databases. In our case, especially looking at distributed relational databases is interesting.

In [31], Cyganiak describes how RDF based data can be stored in a relational database, by defining a mapping from the RDF data model to the relational data model. He further shows how SPARQL queries can be translated to SQL queries over the translated data model. Thus these queries can be executed using a standard relational database. The article further discusses mismatches between the SPARQL semantics and SQL semantics. A similar approach is described by Harris and Shadbolt in [54]. They explain the 3store RDF database that also bases on relational technology.

Relational query processing over distributed databases is a field where much research has been conducted over the past years. A good overview can be found in [71]. Techniques for distributed query processing (DQP) indeed have relations to our work. However, in our setting, no global knowledge does exists. Thus typically information like histograms of the data distribution is not available. It is an interesting direction of work to include and dynamically update such information in our setting.

A further issue with DQP in our setting are the different data models. An RDF query pattern can be viewed as a combination of numerous joins. Thus joins have a completely different significance in RDF. The data typically consists of countless pieces of information, that are by themselves small. In relational data modeling, joins are typically avoided as far as possible, as they are expensive.

6.4 Peer-2-Peer

Peer-2-peer computing has grown out of its origins of file-sharing towards a versatile utility for distributed data management. The introduction of structured p2p networks [73], e. g. based on distributed hash tables (DHT) [11], has made routing much more predictable. DHTs provide a great enhancement compared to the early p2p networks like Gnutella [102], which use basically a flooding mechanism to route a message to the target node.

Nowadays, a variety of different algorithms implementing the DHT abstraction has been devised. In the foundations chapter, we have already described two main representative DHT systems: Chord and Pastry, see section 2.2. Further systems include the content addressable network (CAN) [92], Bamboo [93], and P-Grid [1]. They differ mostly in their routing strategy, leading to different characteristics concerting routing speed, ability to adapt to changes, etc. For example, Bamboo has been designed to cope with nodes which only take part in the network for a very short span of time.

On top of these DHTs several applications have been designed. The PAST project [98] provides algorithms for replicated and reliable storage of files using the Pastry network. The MAAN network [27] is an extension of Chord allowing to find data items via multiple key attributes. SCRIBE [29] is a publish/subscribe infrastructure using Pastry.

We now look in more detail at existing p2p based approaches that deal with the integration of multiple data sources, which are typically heterogeneous. The survey does not aim to be exhaustive; we rather describe selected works which represent different approaches to the problem.

We start with two systems having their roots in traditional database research: Hyperion [6] and Piazza [53, 52]. Their basic assumption is quite similar: each peer holds a collection of physical relations, and associated schema information. It has furthermore a mediated schema, which represents a homogenous view including the peer's own relations and the mediated schema of the neighbors of this peer. Both systems assume the existence of mapping information. They differ in the types of supported mappings.

Next, we describe the Chatty-Web approach [2], which is a specific solution to the problem of errors and losses during the mediation between different schemas.

Furthermore, we describe some of the systems stemming from the Semantic Web research. GridVine [3], the ICS/Forth RDF Suite [70], and Edutella [83, 82, 24] are systems based on RDF which employ RDF Schema information. Finally, we mention Bibster [51] as an example of a domain-specific application of schema based p2p systems focussing on the exchange of bibliographic meta data.

6.4.1 P2P Data Integration

We now look at work that stems from the field of data integration [75]. Data integration typically looks at the problem how multiple data sources having different schemas can be integrated into a global schema and queried using this unified schema. P2p data integration systems can be seen as a natural evolution step of these systems, moving from a single, centralized global schema towards an arbitrary number of peers where each peer runs a local data integration system integrating both its own data and data from the other peers.

Two representatives for p2p data integration systems are Hyperion and Piazza. Their origins are stand-alone data integration systems for relational or XML-based data. Both aim to transfer and extend the work from data integration to a p2p setting. The basic architecture of both systems is similar, see figure 6.1. Each system consists of peers holding **stored relations**, which are connected via an unstructured p2p network. In Hyperion, the **peer schema** is the schema of the stored relations, while in Piazza the peer schema is a mediated schema which spans both over its own stored relations and the peer schemas of other peers. The relationships between the different schemas are represented through mappings.

Both systems use an unstructured p2p network. Thus each peer has a limited set of connections to neighboring peers, and might store some peer mappings relating its data to the neighbors data. A peer which receives a query formulated using its peer

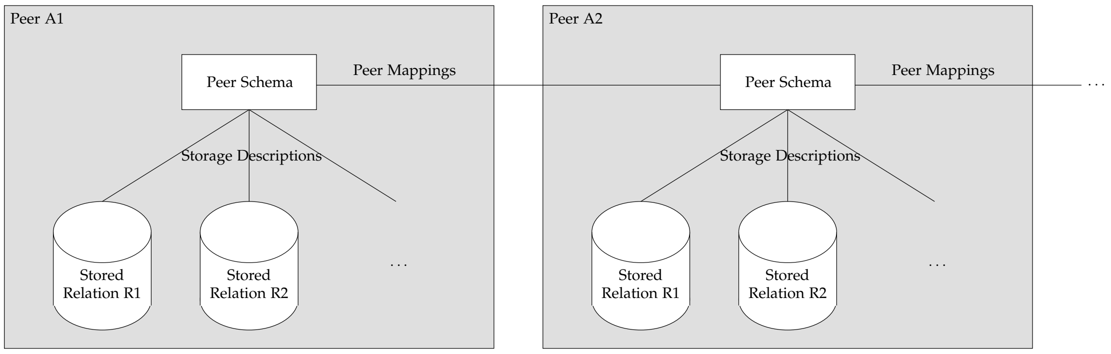


Figure 6.1: Piazza Architecture.

schema, will reformulate the query using the available mappings, and forward it to the other peers.

In Piazza, a global system catalog containing all mappings is assumed during the query reformulation. The authors have identified this flaw and are working towards a distributed version of the query reformulation algorithm. They plan to use DHT techniques to store a distributed system catalog.

In [108], Triantafillou and Pitoura try to unify different approaches to query processing over structured p2p networks. They assume a relational data model and define different query types over these relations, including simple queries involving only a single relation and a single attribute, range queries over attributes, and joins. The infrastructure is based on Chord. Each tuple from each relation is stored on multiple nodes, to be accessible via each attribute's value. Numerical attributes are indexed using an order-preserving hash function, so that a range query can be evaluated by asking every node within a certain sector of the Chord ring. However, this method has a worst-case running time of $O(N)$, as each node in the network might be contacted. The best-case performance is $O(\log N)$. To improve this situation, they insert so-called range guards. These range guards build a second overlay, which store all information (similar to super-peers in other approaches). The algorithm stays the same, however, as it now operates on the range guards instead of the whole network, it has now the worst case running time $O(l)$, where l is the number of range guards. Typical values are $l = \log N$ or $l = \sqrt{N}$.

For join queries, the first observation is that matching tuples are always found on a single node, as they share a common attribute value, which will be hashed to the same node. However, join queries exhibit the same problem as range queries. Unless some other attributes are known, each peer has to be asked to find the result. Again, range guards are used to resolve this problem.

6.4.2 Semantic Query Routing

In this section, we look at two systems related to the question how queries can be routed in p2p systems with different semantic domains. Possibly mediators between these domains exist.

New mappings between different schemas can be generated by exploiting a transitive relationship between existing mappings. However, as translations may be lossy or may have errors, the quality of these new mappings might be poor.

The Chatty Web approach [2] is a way to detect these problems, and to steer the routing decisions in the network according to the observed quality. The underlying system model is the local data system model (see section 3.3.2), typically combined with an unstructured p2p network. Thus each peer or group of peers has its own schema and maintains links together with schema translations to other peers.

The whole network can be seen as a graph, where each group of peers with a common schema is represented by a vertex, while the known translations are the edges. A query that is routed through this graph is continuously translated in order to match the target schema.

An important feature of this graph is the existence of cycles. This means that a node, which routes a query toward another semantic domain, might receive the same query again later. However, the query will have been modified multiple times through various mappings. As the original peer knows also the original version of the query, it is able to compare these two versions and draw conclusions about the quality of the mappings on the used path.

Consider a source node S and a target node T. The source node receives a query using its schema, and forwards the query towards T. Node T expects to receive a query in its own schema. The mappings are on the attribute level and specify how the attributes of S can be expressed as functions over the attributes of T. These mappings are applied to the query, so that it can be evaluated over T's database.

When a peer receives a query, it detects whether the query has passed a cycle. In this case, the similarity between the original version and the new version is measured with different indicators:

- *Syntactic Similarity*: Here, attributes are counted which are missing in the target query. However, as not all attributes share the same importance for a query, they are weighted with both a user defined weight and a system-supplied weight in the case of selection-relevant attributes. The system weight reflects the selectivity of the attribute.

- *Cycle analysis*: Here, the correctness of the resulting query is measured. Each attribute may be preserved (positive score), may have vanished (neutral score), or it may have been replaced by a wrong attribute (negative score). Furthermore, the probability of compensating errors is calculated and taken into account.

- *Result analysis*: This analysis checks to which degree a known functional dependency is respected by the query results returned from peers on the cycle that do not share the same schema.

Subsequently, these measures are used in routing decisions. The user is requested to supply lower bounds for the similarities. Queries are only routed to neighbors where the iteratively updated measures are above these bounds. This ensures that queries will be routed into domains where they are likely to produce valid answers, and prevents flooding.

In [78], Löser follows an approach where data sources are classified in taxonomies and stored in a fully distributed system. It extends traditional distributed hash tables with hierarchical structures, so that the system is able to process hierarchical or conjunctive queries. By marking selected P2P-nodes as Super-Peer (SP) nodes, a centralized SP network within the distributed P2P-network is established. This can be used for load balancing on storage of a large number of peer information and prevents flooding of queries over the whole system.

6.4.3 RDF based P2P Systems

GridVine [3] is an DHT based RDF [80] repository. RDF triples in the network are indexed by subject, predicate, and object. Thus, GridVine can retrieve matches for a triple pattern with a single lookup as long as there is at least one element of the triple known. Based upon this basic lookup primitive, more complex queries are available. GridVine supports the RDQL query language [99].

GridVine also supports RDFS. As RDFS lacks a direct way to describe equality between properties, GridVine borrows the semantics for equivalent properties from OWL [101] to translate queries into other semantic domains. This translation information is also stored using the DHT.

For each query which is received by a peer, this peer looks up translations in the network and can thus reformulate the query to other semantic domains. A translated query is either forwarded to a target peer which then executes the translated query and recursively translates and forwards the query, or the original node stays responsible for the query and iteratively applies multiple translations.

To steer the translation decision and to measure the quality of multiple chained translations, GridVine applies methods from the Chatty Web approach.

The ICS/Forth RDF Suite [70] is also a p2p based RDF store which is aware of RDF Schema. In contrast to GridVine, the RDF triples themselves are not distributed but rather stay at the original node. Thus for an exhaustive answer to a query the system needs to execute the query on each peer which might have answers to the query.

RDF Schema information is used to identify peers which may hold matches for the given query. For this, so-called RVL[1] views are defined representing parts of the

[1]RDF View Language

schema graph. A query graph is broken down into multiple sub graphs; for each sub graph peers are searched that have matching RVL views. By this, both the RDF Schema inheritance (called vertical subsumption) and sub graph relationships (called horizontal subsumption) are respected. The main focus of the work is the query planning and execution, driving the breakdown of the query graph and the forwarding of the sub-queries to the target nodes.

The RDFPeers system [28] uses MAAN [27] to store RDF triples in a structured p2p network. Each part of an RDF triple (subject, predicate, and object) is used as DHT key for the multi-addressable MAAN. RDFPeers provides range queries and a publish/subscribe interface over the stored data.

PeerThing [64] is another approach we developed at PC^2. It consists of an unstructured p2p network for resource discovery in the Grid, where each node in the network runs a local Description Logic reasoner. Thus it does not support the global data system model. In contrast, it supports highly expressive semantics for resource description and matching.

A new system recently described in [72] is called Atlas. The approach described in the paper is very similar to our system, and the motivation is also resource discovery in Grids. However, the paper does not describe details of the query evaluation strategies nor reasoning mechanisms.

6.4.4 Edutella

The Edutella project [82, 24] implements a schema-based p2p system. By this, the authors mean a system which is aware of schema information and uses it in query optimization and routing. Multiple schemas are allowed, but currently no translation is applied to mediate between different schemas. It is up to the user to produce queries which match the schema used to describe the data. However, Edutella is work in progress and aims to fill this gap. In the Edutella white paper [83] a meditator architecture is described in which so-called *query hubs* present views spanning over the data of multiple peers which can be queried using the hub's schema.

Edutella is based on a super-peer architecture. A small number of well equipped nodes are selected to form the super-peer network, while the other peers connect in a star-like fashion to the super-peers. The super-peers are connected in a so-called HyperCuP[2] topology. In this structure, each super-peer can be seen as the root of a spanning tree which is used for query routing, updates of indices, and broadcasting.

Each peer sends indices of its data to its super-peer. This information can be stored in different granularities, like schema, property, property value range, or property value. However, the index never points to individual data entries but to peers. This kind of index is called an SP-P (super-peer - peer) index.

The super-peers share their index information along the spanning tree structures. Thus each SP also holds several SP-SP (super-peer - super-peer) indices which guide

[2]Hypercube P2P

the peer when forwarding the query or parts of it to other super-peers.

Based upon this infrastructure, a query processor and optimizer tries to split the query into multiple parts and to ship the query to target peers which are likely to have results for these parts. The same part of a query might also be shipped to multiple super-peers and peers. The queries may carry code for user-defined operators, so that the operators can be executed on the peer which holds the relevant data.

In order to cope with different query semantics, Edutella defines the *Edutella Common Data and Query Exchange Model* (ECDM). It is based on RDF and is used internally to represent queries and their results. The query language is called RDF query exchange language (RDF-QEL)[3]. In the most general form, RDF-QEL queries are datalog queries, with several built-in predicates suited for the evaluation of RDF-based data. In order to cope with peers with limited query processors the query language has different levels, from rule-less queries over conjunctive, disjunctive, up to linear recursive and finally general recursive queries.

6.4.5 Bibster

The Bibster system [51] is an example application of the SWAP[4] project [37]. It is an unstructured p2p system based on JXTA [79] which targets the exchange of bibliographic meta data like (e. g. BibTeX entries) in academic communities. It is schema aware in the sense that two different schemas are supported. One is the ACM topic hierarchy, the other one is Semantic Web for Research Communities (SWRC)[5].

The information sources are integrated into these schemas a priori by a local component which supports a fixed set of mappings. Thus, no query reformulation is necessary. The classification of the database entries according to the ACM topic hierarchy is used to measure the expertise of a peer. A similarity measure between a query and the expertise of a peer is calculated and used in the routing decisions of the network.

An important aspect of Bibster is the removal of duplicate results. In the field of bibliographic databases, it is likely that numerous peers have overlapping sets of entries. However, these entries are not 100 percent identical, but rather similar. Thus, an important task of Bibster is to detect similar entries in the result set of a query and to remove them.

6.5 Summary

In this section we have described different fields of work that are relevant for our system. We have started by describing resource matching in the Grid, and the influence of the Semantic Web approach to the Grid community. Here, interesting approaches have been devised targeting the issue of semantically rich resource discovery. However,

[3]The specification can be found under `http://edutella.jxta.org/spec/qel.html`
[4]Semantic Web and Peer-to-Peer, `http://swap.semanticweb.org`
[5]`http://ontoware.org/projects/swrc`

these approaches ignore the scalability problem in large Grids. As a further motivation for our work, we have briefly described how service computing and Grids are about to merge and the challenges arising for resource discovery through this merger.

In the last two parts of the chapter, we have looked at relational databases, p2p computing, and especially at approaches to the integration of multiple heterogeneous data sources through p2p networks. Here already much work exists. However, most systems do not implement the global data system model. Those that do, like RDF-Peers, have other drawbacks like the lack of RDFS semantics or sophisticated query processing.

7 Conclusion

As we have stated in the introduction, this thesis focusses on Grid resource discovery. However, it can also be used for further applications. Because of this we start our conclusion by highlighting further use cases where our system is valuable. We continue by explaining the limitations of our work, and describe future work to make the system more versatile. Finally, we summarize the thesis and its contributions.

7.1 Applications of P2P based RDF Querying and Reasoning

If we generalize the problem tackled in this thesis, we can describe it as follows:

Given a **large, world-wide distributed** collection of entities
that produce **machine-readable** meta data,
how can we **efficiently** combine this knowledge, **reason** about it, and **query** it?

Generally speaking, we can see that more and more **entities** in the world **become IT enabled and networked**. This ranges from mobile phones and PDA's that include wireless network access to networked home devices to cars and other devices of daily life. The more they are connected and able to publish their local knowledge to the network, the more valuable this information will be. Other types of devices include sensors e. g. in industrial applications, etc.

There is also another wealth of information available that can only be tapped by humans today: **the Web**. It contains billions of pages providing information about virtually everything you can think of. However, this information is pure text without semantic markup. The provided markup in HTML is only formatting and helps little when a computer tries to *understand* the contents of a page. Thus it is left to humans to *combine* the information from the pages. The goal of the Semantic Web community is to provide the Web's content in a machine-readable format. Here, this thesis contributes an efficient mechanism to harvest this content as it becomes available.

Another application is **network management**. Within the DELIS project[1], we co-operate with Telecom Italia Learning Solutions to tailor our methods to the needs of network management. The idea is that a network consists of numerous entities that

[1]http://delis.upb.de

generate meta data used for monitoring and administration, including facts about utilization, access control, failures, etc. All these pieces of information have to be evaluated together in an efficient way. Reasoning is also useful e. g. to classify device types, failure types, etc.

Distributed ontologies also play a role in agent computing. For example in [107], ontologies are used to control the behavior of agents by describing policies and applying reasoning to evaluate these ontologies. Again, an efficient way to reason about large scale collections of distributed policy information is needed.

Finally, we mention the field of service discovery. The conversion of the Web towards the Semantic Web as described above includes not only information, but also services. This makes services which were previously only accessible through human-tailored Web forms accessible for computers. This rises the need to describe and discover these services, and to efficiently query the distributed collection of service descriptions, including reasoning e. g. about the compatibility of services and data.

7.2 Limitations of our Approach and Further Work

We have designed a system that addresses the integration and querying of semantically heterogeneous distributed information. However, this field is very broad. Each of the various applications listed in the previous chapter poses individual challenges requiring specific features. Thus it is impossible to design a single system that fulfills every demand.

We have concentrated on important foundational aspects like distribution of information, basic reasoning, and efficient querying including top k queries. However, we also have left out a number of further issues that we list in this section.

First of all, we have completely ignored issues like **security, confidentiality, privacy, trust, and consistency**. We have basically assumed that all information published to the network is open to every participant of the network, that the information is correct and nobody tries to cheat, and that the information published from various nodes is consistent. It is clear that these assumptions do not hold in some applications. Thus extensions that address these issues will improve the usability of our system in further use cases. There is already a large body of work addressing trust, reputation, and security in p2p networks, which can be applied to our work.

Furthermore, the system can be improved from a technical point of view in various directions. From our point of view, two topics are especially promising. On the one hand, **load balancing** becomes a more important issue the larger the p2p network grows. Depending on the kind of inserted knowledge, the load imbalances lead to scalability problems as addressed in section 5.4.6. The p2p community has developed several approaches to load balancing in DHT networks, however the special characteristics of our application make most of them unsuitable in our situation. Thus new ways have to be developed to cope with this problem.

On the other hand, **advanced semantics** are important to broaden the range of possible applications and to make the system even more useful for integration and reasoning about heterogeneous data sources. Ideally, the system should support semantics equivalent to very expressive description logics. However, it is difficult to identify relevant knowledge for a query a priori, which induces that every node has to be contacted. As this solution is not scalable, other approaches are sought. Thus we aim to integrate further semantics using either more forward chaining of rules or query rewriting techniques. Interesting concepts include support for transitive properties, inverse properties, etc.

Additionally, it is desirable to support more features included in the **SPARQL query language**. Some of them like filters on individual attributes are straight forward to integrate. Filters including comparisons of multiple attributes are more demanding as the data for these attribute may reside on different nodes. Here are advanced techniques needed to maintain efficiency and to avoid the client-side evaluation of these filters. Other interesting features include optional parts of the query graph and disjunction of multiple queries.

7.3 Summary

In this thesis, we have worked on the question how to query and reason about large-scale, distributed collections of machine-readable information in an efficient way. We have described, implemented and evaluated a system that distributes RDF based information, performs reasoning supporting RDF Schema-based hierarchies of properties and classes, and supports efficient query evaluation.

The contributions of this thesis are as follows:

- We have designed an **architecture** targeting the described problem, which includes a problem-specific application layer above the well-known DHT layer. The architecture consists of the query processor, the RDF Schema reasoner, a disseminator that distributes the information, and a client interface.

- We support a system model we call the **global data system model** where individual answers to queries include information scattered over multiple nodes. This is a clear advantage over systems that generate query results on each node individually.

- We have designed a **dissemination algorithm** which stores information on well-defined nodes taking advantage of the DHT approach. The distribution of the information allows efficient evaluation of **RDF Schema rules**.

- We have designed an **exhaustive query evaluation** algorithm that allows to retrieve every answer for a given query. We have shown various strategies to reduce network load during query evaluation.

- We have further designed a **top** k **query evaluation** algorithm that is efficient even in situations when queries generate huge numbers of answers. The algorithms employs caching and look ahead strategies to reduce network load.
- We have developed a **prototypical implementation** of the system which can be used also for simulations. The implementation is based on the FreePastry library as DHT layer and is realized in Java.
- We have performed both **real life benchmarks and simulations** to gain insights into the behavior of our system. The benchmarks on the ARMINIUS cluster of PC^2 show that the system performs good and scales well. The simulations give further insight in specific aspects of the system.

Overall, we have designed a promising system and showed that the introduced concepts are working correctly and efficiently. This system is a starting point for one of the most important challenges in computer science: How can we harvest the vast amounts of digital information scattered over the world?

List of Figures

Bibliography

[1] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: a self-organizing structured P2P system. *SIGMOD Record*, 32(3):29–33, 2003.

[2] Karl Aberer, Philippe Cudré-Mauroux, and Manfred Hauswirth. Start making sense: The Chatty Web approach for global semantic agreements. *Journal of Web Semantics*, 1(1), December 2003.

[3] Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth, and Tim Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2004.

[4] Sergio Andreozzi, Stephen Burke, Laurence Field, Steve Fisher, Balazs Kónya, Marco Mambelli, Jennifer M. Schopf, Matt Viljoen, and Antony Wilson. Glue schema specification, version 1.2, final specification. `http://infnforge.cnaf.infn.it/glueinfomodel/uploads/Spec/GLUEInfoModel_1_2_final.pdf`, December 2005.

[5] Grigoris Antoniou and Frank van Harmelen. Web Ontology Language: OWL. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*. Springer-Verlag, 2003.

[6] Marcelo Arenas, Vasiliki Kantere, Anastasios Kementsietsidis, Iluju Kiringa, Renée J. Miller, and John Mylopoulos. The Hyperion Project: From Data Integration to Data Coordination. *SIGMOD Record*, 32(3):53–58, 2003.

[7] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.

[8] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Publishing Company, 1999.

[9] Henri Bal et al. Next Generation Grid(s): European Grid Research 2005 - 2010. `ftp://ftp.cordis.lu/pub/ist/docs/ngg_eg_final.pdf`, 2003.

[10] Henri Bal et al. Next Generation Grids 2: Requirements and Options for European Grids Research 2005-2010 and Beyond. ftp://ftp.cordis.lu/pub/ist/docs/ngg2_eg_final.pdf, 2004.

[11] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46(2):43 – 48, February 2003.

[12] Jean-Pierre Banatre et al. Future for European Grids: GRIDs and Service Oriented Knowledge Utilities: Vision and Research Directions 2010 and Beyond. ftp://ftp.cordis.lu/pub/ist/docs/grids/ngg3_eg_final.pdf, 2006.

[13] Dominic Battré, Felix Heine, and Odej Kao. Top k RDF Query Evaluation in Structured P2P Networks. In *Euro-Par 2006 Parallel Processing: 12th International Euro-Par Conference*, 2006.

[14] Dave Beckett. RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar, 2004.

[15] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC3986: Uniform Resource Identifiers (URI): Generic Syntax. http://www.ietf.org/rfc/rfc3986.txt, 2005.

[16] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[17] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[18] Dan Brickley and Ramanathan V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema, 2004.

[19] Jeen Broekstra and Arjohn Kampman. Inferencing and Truth Maintenance in RDF Schema. In *Proceedings of the First International Workshop on Practical and Scalable Semantic Systems*, 2003.

[20] Jeen Broekstra and Arjohn Kampman. SeRQL: An RDF Query and Transformation Language. http://www.cs.vu.nl/~jbroeks/papers/SeRQL.pdf, 2004.

[21] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF. In *The International Semantic Web Conference 2002, Sardinia, Italy*, 2002.

[22] John Brooke, Donal Fellows, Kevin Garwood, and Carole Goble. Semantic matching of Grid Resource Descriptions. In *2nd European Across-Grids Conference*, 2004.

[23] John M. Brooke and Donal K. Fellows. An Architecture for Distributed Grid Brokering. In *11th International Euro-Par Conference (Euro-Par 2005)*, 2005.

[24] Ingo Brunkhorst, Hadhami Dhraief, Alfons Kemper, Wolfgang Nejdl, and Christian Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In *Databases, Information Systems, and Peer-to-Peer Computing, First International Workshop, DBISP2P, Berlin Germany, September 7-8, 2003, Revised Papers*, pages 184–199, 2003.

[25] Lars-Olof Burchard, Felix Heine, Hans-Ulrich Heiss, Matthias Hovestadt, Odej Kao, Axel Keller, Barry Linnert, and Jörg Schneider. The Virtual Resource Manager: Local Autonomy versus QoS Guarantees for Grid Applications. *Future Generation Grids*, 2005.

[26] Lars-Olof Burchard, Felix Heine, Matthias Hovestadt, Odej Kao, Axel Keller, and Barry Linnert. A Quality-of-Service Architecture for Future Grid Computing Applications. In *Proceedings of the 13th International Workshop on Parallel and Distributed Real-Time Systems*, 2005.

[27] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *Journal of Grid Computing*, 2(1), 2004.

[28] Min Cai, Martin Frank, Baoshi Pan, and Robert MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 2, Issue 2*, 2005.

[29] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[30] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[31] Richard Cyganiak. A relational algebra for SPARQL. Technical report, Hewlett-Packard, 2005.

[32] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, 2001.

[33] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[34] David De Roure, Nicholas R. Jennings, and Nigel R. Shadbolt. The Semantic Grid: Past, Present and Future. In *Proceedings of the IEEE*, volume 93:3, 2005.

[35] Stefan Decker, Frank van Harmelen, Jeen Broekstra, Michael Erdmann, Dieter Fensel, Ian Horrocks, Michel Klein, and Sergey Melnik. The Semantic Web - on the respective Roles of XML and RDF. *IEEE Internet Computing*, September/October 2000.

[36] Dietmar Erwin (ed.). UNICORE Plus Final Report - Uniform Interface to Computing Resources. `http://www.unicore.org/documents/UNICOREPlus-Final-Report.pdf`, 2002.

[37] Marc Ehrig, Peter Haase, Ronny Siebes, Steffen Staab, Heiner Stuckenschmidt, Rudi Studer, and Christoph Tempich. The SWAP Data and Metadata Model for Semantics-Based Peer-to-Peer Systems. In *Multiagent System Technologies, First German Conference, MATES 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, pages 144–155, 2003.

[38] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt Academic Press, 2001.

[39] Dieter Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, 2003.

[40] Ian Foster, Steve Graham, Steve Tuecke, Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, William Vambenepe, and Sanjiva Weerawarana. Modeling Stateful Resources with Web Services. `http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf`, 2004.

[41] Ian Foster and Carl Kesselman, editors. *The Grid2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[42] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems. The Complete Book*. Prentice Hall, 2003.

[43] Asunción Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering*. Springer, 2004.

[44] Javier González-Castillo, David Trastour, and Claudio Bartolini. Description Logics for Matchmaking of Services. In Günther Görz, Volker Haarslev, Carsten Lutz, and Ralf Möller, editors, *Proceedings of the KI-2001 Workshop on Applications of Description Logics*, volume 44, 2001.

[45] Jan Grant and Dave Beckett (eds.). RDF Test Cases. `http://www.w3.org/TR/rdf-testcases`, 2004.

[46] Tom R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies, special issue on Formal Ontology in Conceptual Analysis and Knowledge Representation (guest editors: N. Guarino and R. Poli)*, 1993.

[47] Nicola Guarino. Understanding, Building, and Using Ontologies. In *Proceedings of Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1996.

[48] Dan Gunter and Keith Jackson. The Applicability of RDF-Schema as a Syntax for Describing Grid Resource Metadata. `http://www-unix.mcs.anl.gov/gridforum/gis/reports/rdf/GIS-RDF.pdf`, 2001.

[49] Claudio Gutierrez, Carlos Hurtado, and Alberto Mendelzon. Formal aspects of querying RDF databases. In *First International Workshop on Semantic Web and Databases*, 2003.

[50] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *Proceedings of the Third International Semantic Web Conference*, 2004.

[51] Peter Haase, Jeen Broekstra, Marc Ehrig, Maarten Menken, Peter Mika, Mariusz Olko, Michal Plechawski, Pawel Pyszlak, Björn Schnizler, Ronny Siebes, Steffen Staab, and Christoph Tempich. Bibster - A Semantics-Based Bibliographic Peer-to-Peer System. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 122–136, 2004.

[52] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The Piazza Peer Data Management System. *IEEE Transactions on Knowledge and Data Engineering*, 16(7), 2004.

[53] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema Mediation in Peer Data Management Systems. In *19th International Conference on Data Engineering (ICDE)*, 2003.

[54] Steve Harris and Nigel Shadbolt. SPARQL Query Processing with Conventional Relational Database Systems. In *Web Information Systems Engineering WISE 2005 International Workshops. Proceedings*, 2005.

[55] Patrick Hayes. RDF Semantics. `http://www.w3.org/TR/rdf-mt`, 2004.

[56] Felix Heine. Emergent Schema Management for P2P-based Applications. In Roberto Baldoni, Giovanni Cortese, Fabrizio Davide, and Angelo Melpignano, editors, *Global Data Management*. IOS Press, 2006.

[57] Felix Heine. Scalable P2P based RDF Querying. In *First International Conference on Scalable Information Systems (INFOSCALE06)*, 2006.

[58] Felix Heine, Matthias Hovestadt, and Odej Kao. HPC4U: Providing Highly Predictable and SLA-aware Clusters for the Next Generation Grid. In *The 4th Cracow Grid Workshop*, 2004.

[59] Felix Heine, Matthias Hovestadt, and Odej Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In *5th International Workshop on Grid Computing*, pages 76–83, November 2004.

[60] Felix Heine, Matthias Hovestadt, and Odej Kao. Processing complex RDF queries over P2P networks. In *P2PIR'05: Proceedings of the 2005 ACM workshop on Information retrieval in peer-to-peer networks*, pages 41–48. ACM Press, 2005.

[61] Felix Heine, Matthias Hovestadt, Odej Kao, and Axel Keller. Provision of Fault Tolerance with Grid-enabled and SLA-aware Resource Management Systems. In *Proceedings of the Parallel Computing Conference*, 2005.

[62] Felix Heine, Matthias Hovestadt, Odej Kao, and Axel Keller. SLA-aware Job Migration in Grid Environments. In Lucio Grandinetti, editor, *Grid Computing: New Frontiers of High Performance Computing*. Elsevier, 2005.

[63] Felix Heine, Matthias Hovestadt, Odej Kao, and Achim Streit. On the Impact of Reservations from the Grid on Planning-Based Resource Management. In *International Workshop on Grid Computing Security and Resource Management*, 2005.

[64] Felix Heine, Matthias Hovestadt, Odej Kao, and Kerstin Voss. PeerThing: P2P-based Semantic Resource Discovery. In *The 5th Cracow Grid Workshop*, 2005.

[65] Hans-Christian Hoppe and Daniel Mallmann. EUROGRID - European Testbed for GRID Applications. In *GRIDSTART Technical Bulletin*, October 2002.

[66] William H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, 2005.

[67] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A Declarative Query Language for RDF. In *The eleventh international world wide web conference (WWW2002)*, 2002.

[68] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems. An Application-Oriented Approach*. Addison Wesley, 2005.

[69] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts`, 2004.

[70] George Kokkinidis, Lefteris Sidirourgos, and Vassilis Christophides. Query Processing in RDF/S-based P2P Database Systems. *Semantic Web and Peer to Peer*, November 2005.

[71] Donald Kossmann. The State of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[72] Manolis Koubarakis, Iris Miliaraki, Zoi Kaoudi, Matoula Magiridou, and Antonis Papadakis-Pesaresi. Semantic Grid Resource Discovery using DHTs in Atlas. In *3rd GGF Semantic Grid Workshop*, February 2006.

[73] John Kubiatowicz. Extracting Guarantees from Chaos. *Communications of the ACM*, 46(2):33 – 38, February 2003.

[74] Rubén Lara, Dumitru Roman, Axel Polleres, and Dieter Fensel. A Conceptual Comparison of WSMO and OWL-S. In *Proceedings of the European Conference on Web Services (ECOWS)*, pages 254–269, 2004.

[75] Maurizio Lenzerini. Data integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM Press, 2002.

[76] Lei Li and Ian Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proceedings of the twelfth international conference on World Wide Web*, pages 331–339. ACM Press, 2003.

[77] Max O. Lorenz. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 9(70):209–219, 1905.

[78] Alexander Löser. Towards Taxonomy-based Routing in P2P Networks. In *Proceedings of SemPGRID '04*, 2004.

[79] Qusay H. Mahmoud, editor. *Middleware for Communications*. John Wiley & Sons, Ltd, 2004.

[80] Frank Manola and Eric Miller. RDF Primer. `http://www.w3.org/TR/rdf-primer`, 2004.

[81] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic Markup for Web Services. `http://www.w3.org/Submission/OWL-S`, 2004.

[82] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design Issues and Challenges for RDF- and Schema-Based Peer-to-Peer Systems. *SIGMOD Record, Special Issue on Peer-to-Peer Data Management*, September 2003.

[83] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA*, pages 604–615, 2002.

[84] Bernd Oestereich. *Analyse und Design mit der UML 2.1 - Objektorientierte Softwareentwicklung*. Oldenbourg Wissenschaftsverlag, 2006.

[85] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and Complexity of SPARQL. *ArXiv Computer Science e-prints*, 2006.

[86] Eric Prud'hommeaux and Andy Seaborne (eds.). SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query`, 2006.

[87] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[88] Preeda Rajasekaran, John Miller, Kunal Verma, and Amit Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In *Proceedings of the International Workshop on Semantic Web Services and Web Process Composition*, 2004.

[89] Rajesh Raman, Miron Livny, and Marvin Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of the Twelfth IEEE International Symposium on High-Performance Distributed Computing*, Seattle, WA, June 2003.

[90] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *HPDC*, pages 140–, 1998.

[91] Michael Rambadt and Philipp Wieder. UNICORE - Globus: Interoperability of Grid Infrastructures. In *Proceedings of the Cray User Group Summit 2002*, 2002.

[92] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[93] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.

[94] J. Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. MIT Press, September 2001.

[95] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[96] Antony Rowstron and Peter Druschel. FreePastry. `http://freepastry.org`.

[97] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[98] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM SOSP'01, Lake Louise, Alberta, Canada*, October 2001.

[99] Andy Seaborne. RDQL - A Query Language for RDF. `http://www.w3.org/Submission/RDQL`, 2004.

[100] Luciano Serafini and Andrei Tamilin. Local Tableaux for Reasoning in Distributed Description Logics. In *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, 2004.

[101] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL Web Ontology Language Guide. `http://www.w3c.org/TR/owl-guide`, 2004.

[102] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*. Springer, 2005. LNCS 3485.

[103] Ralf Steinmetz and Klaus Wehrle. What Is This "Peer-to-Peer" About? *Peer-to-Peer Systems and Applications, LNCS 3485*, 2005.

[104] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11, 2003.

[105] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 2002.

[106] Hongsuda Tangmunarunkit, Stefan Decker, and Carl Kesselman. Ontology-Based Resource Matching in the Grid - The Grid Meets the Semantic Web. In *Proceedings of SemPGRID '03*, 2003.

[107] Gianluca Tonti, Jeffrey M. Bradshaw, Renia Jeffers, Rebecca Montanari, Niranjan Suri, and Andrzej Uszok. Semantic Web Languages for Policy Representation and Reasoning: A Comparison of KAoS, Rei, and Ponder. In *The SemanticWeb - ISWC 2003*, volume 2870 of *Lecture Notes in Computer Science*, pages 419 – 437. Springer, 2003.

[108] Peter Triantafillou and Theoni Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *Databases, Information Systems, and Peer-to-Peer Computing, First International Workshop, DBISP2P, Berlin Germany, September 7-8, 2003, Revised Papers*, pages 169–183, 2003.

[109] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[110] Sven van den Berghe. Using the Incarnation Database (V4.1). `http://unicore.sourceforge.net/docs/idb_manual.pdf`, 2004.

[111] W3C: World Wide Web Consortium. `http://www.w3.org`.

[112] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proceedings of SWDB'03, The first International Workshop on Semantic Web and Databases*, pages 131–150, 2003.