University of Crete

Computer Science Department

Indexing Views to Route and Plan Queries in a Peer Data Management System

Lefteris E. Sidirourgos

Master of Science Thesis

Heraklion, November 2005

Indexing Views to Route and Plan Queries in a Peer Data Management System

Lefteris E. Sidirourgos

Master of Science Thesis

Computer Science Department, University of Crete

Abstract

P2P computing gains increasing attention lately, since it provides the means for realizing computing systems that scale to very large numbers of participating peers, while ensuring high autonomy and fault-tolerance. Peer Data Management Systems (PDMS) have been proposed to support sophisticated facilities in exchanging, querying and integrating (semi-)structured data hosted by peers. In this thesis, we are interested in routing and planning graph queries in a PDMS, where peers advertise their local bases using fragments of community RDF/S schemas (i.e., views). We introduce an original encoding for these fragments, in order to efficiently check whether a peer view is subsumed by a query. We rely on this encoding to design an RDF/S view lookup service featuring a stateless and a statefull execution over a DHT-based P2P infrastructure. We design and implement a mechanism based on an interleaved execution of the routing and planning activities in order to distribute the processing of a query. We finally evaluate experimentally our system (a) to demonstrate its scalability for large P2P networks and arbitrary RDF/S schema fragments, (b) to estimate the number of routing hops required by the two versions of our lookup service and (c) to demonstrate the degree of distribution achieved by the interleaved query routing and planning. To the best of our knowledge this is the first system offering the aforementioned functionality and performance.

Contents

1 Introduction **1**

2 RDF/S based PDMS **7**

2.1 RDF/S schemas 8

2.2 RDF/S peer base advertisements and queries 10

2.3 RDF/S query and view subsumption 14

3 An Encoding for RDF/S Schema Graph Fragments **17**

4 A DHT-Framework for RDF/S Queries **25**

4.1 Peer joins, departures and updates 27

4.2 Lookup Service 31

5 Interleaved Query Routing and Planning **39**

5.1 Query Fragmentation and Planning 41

5.2 Statefull Query Routing and Planning 46

5.3 Stateless Query Routing and Planning 50

6 Experimental Evaluation **55**

6.1 DHT-based Schema Index and Lookup Service 56

6.2 Interleaved Query Routing and Planning 60

7 Related Work **67**

8 Conclusion and Future Work **73**

Bibliography **75**

List of Tables

2.1 Class and property query/view patterns 12
2.2 Graph fragments specified by query/view patterns 13

4.1 AdjSub Cube traversal for RDF/S schema fragments. 32

List of Figures

2.1 An RDF/S schema graph 8
2.2 Peers belonging to the same SON and their views 10
2.3 Two cases of view subsumption. 15

3.1 Class and Property subsumption hierarchies 18
3.2 AdjSub Cube for an RDF/S schema 19
3.3 Encoding Algorithm for view V 21

4.1 A Chord ring with eight peers. 26
4.2 Peer *pr3* joins the network. It advertise its view $V0$ and the vertically subsuming views 28
4.3 Algorithm to compute all vertically subsuming views 29
4.4 Distributed sublookup algorithm 35
4.5 Routing hops on the Chord ring for the statefull and stateless version of the sublookup algorithm 36

5.1 All possible fragmentations of query Q 42
5.2 Plans for the query fragmentations depicted in Figure 5.1 . . . 43
5.3 Subplan 1.1 optimized by applying the algebraic equivalence . 44
5.4 Statefull execution of the interleaved routing and planning . . 47
5.5 Steps taken by the coordinator peer during the statefull planning 49

5.6 Statefull planning algorithm at peer p 49
5.7 Stateless execution of the interleaved routing and planning . . 51
5.8 Algorithm of the stateless execution policy 52

6.1 Distribution of views over peers in networks of different size. . 56
6.2 Number of routing hops for networks of different size. 58
6.3 Number of routing hops for queries of different size 59
6.4 Dynamic Programming 60
6.5 Iterative Dynamic Programming 61
6.6 Distribution of peer vies over the network. 62
6.7 Planning Time per Round 63
6.8 Distribution of the total workload over peers 64

Chapter 1

Introduction

Scientific or educational communities are striving nowadays for highly autonomous infrastructures enabling to integrate structured or semi-structured data hosted by peers. In this context, we essentially need a *P2P data management system* (PDMS), capable of supporting loosely coupled communities of databases in which each peer base can join and leave the network at free will, while groups of peers can collaborate on the fly to process queries and provide advanced data management services on a very large scale (i.e., thousands of peers, massive data). A number of recent PDMSs [BGK+02, CGM03, HIMT03, NWS+03] recognize the importance of intensional information (i.e., descriptions about peer contents) for supporting such services. Capturing explicitly the semantics of databases available in a P2P network using a schema enables us to (a) support expressive queries on (semi-)structured data, (b) deploy effective methods for locating remote peers that can answer these queries and (c) build efficient distributed query processing mechanisms.

In this thesis, we are interested in routing and planning graph queries addressed to an RDF/S based PDMS. More precisely, we consider that peers

advertise their local bases using fragments of community RDF/S schemas (e.g., for e-learning, e-science, etc.). These advertisements are specified by appropriate RDF/S views and they are employed during query routing to discover the partitioning (either horizontal, vertical or mixed) of data in remote peer bases. Moreover, peers should share computational power in order to evaluate queries in a distributed manner. The main challenges in this setting is (a) to build an effective and efficient lookup service for identifying, in a decentralized fashion, which peer views can fully or partially contribute to the answer of a specific query; and (b) to design a query planning execution policy that distributes the workload over the peers obtained by the lookup service. Our work is motivated by the fact that a sequential execution of the routing and planning phases for a specific query is not feasible solution in a PDMS context. As a matter of fact, due to the very large number of peers that can actually contribute to the answer of a query, an interleaved query routing and planning will enable us to obtain as fast as possible the first answers from the most relevant peers while the query is further processed by others. More precisely, we make the following contributions:

- we introduce an original encoding of arbitrary RDF/S schema graph fragments for checking whether a peer view is subsumed by a query;
- we design and implement a DHT-based schema index to smoothly distribute view advertisements over peers;
- we design and implement an RDF/S view lookup service that identifies which peers can fully or partially contribute to the answer of a graph query;
- we design and implement an interleaved query routing and planning execution policy that distributes the planning workload over peers and

obtain as fast as possible the first answers from the most relevant peers while the query is further processed by others;

- we experimentally demonstrate the scalability of our DHT-based schema index for networks of different sizes, as well as, estimate the number of routing hops required by a centralized and a distributed execution of the proposed lookup service. Finally, we demonstrate the benefits from the interleaved query routing and planning execution in terms of the degree of distribution.

Part of the work presented in this thesis has been published in [KSC05, SKD05, KSDC05].

To the best of our knowledge no other PDMS offers the aforementioned functionality. Compared to the data indexes maintained by *data-driven PDMSs* that publish directly peer bases on the network [TP03, GWJD03, BT03, CF04, ACMHP04, HHK05], the distributed index on peer views maintained in our system is smaller in size. In fact, it is equal to the number of fragments (e.g., subgraphs) that can be extracted from the RDF/S schemas. Also, it requires a considerably smaller number of messages to be exchanged when peers join or leave the network. Finally, since schema fragments advertised by peers evolve less frequently than their actual bases, such an index does not need frequent updates. As a result, in our approach index maintenance costs are reduced. Unlike other *schema-driven PDMSs* [NWS+03, ETB+03] which maintain a simple inverted list of the RDF/S classes (or properties) actually populated in peer bases, our framework is capable of routing in one step complex graph queries. The proposed lookup service is able to immediately identify peers matching an RDF/S schema graph fragment without the need to decompose queries. Last but not least, our framework exploits the computing power of the P2P network

for fairly distributing in different peers the routing, planning and execution load of queries. It is also worth noticing that PDMSs like Piazza [HIMT03] rely on the mappings established between the individual peer schemas to route queries on semantically related peers rather than on a distributed index. We consider that schema heterogeneity is an orthogonal issue, although our system can be easily extended to also address query reformulation issues [CKK+03].

We believe that our framework is particularly suited for supporting large scale autonomous organizations for which neither a centralized warehouse nor an unlimited data migration from one peer to another are feasible solutions due to societal or technical restrictions. As a matter of fact, many applications (such as networks of institutes sharing scientific knowledge) require data to remain to their natural habitants rather than flowing around the P2P network. However, in our context, peers agree to publish and query their bases according to a number of globally known schemas (e.g., defined by various standardisation bodies). As stated in [SRvdWB05, HHL+03], in a large scale P2P network involving thousands of peers, obtaining the complete answer of a query is infeasible due to network bandwidth limitations and computational cost. For this reason, a mechanism that allows to predefine the amount of data returned or the number of peers contacted is mandatory. Finally, although we rely on RDF/S schemas and Chord for deploying a structured P2P infrastructure, the results presented in this work can be easily adjusted to other data models, like XML, and DHT protocols, like CAN [RFH+01].

The rest of this thesis is organized as follows. In Chapter 2, we overview the proposed framework by focusing on how expressive RDF/S queries employed to retrieve data from the P2P network are matched against the views

published by the peers to advertise their bases. In Chapter 3, we introduce our encoding of arbitrary RDF/S schema fragments. In Chapter 4, we detail how this encoding can be employed to build a DHT-based schema index supporting effective and efficient lookup of intensional peer base advertisements. In Chapter 5 we detail the interleaved query routing and planning execution. In Chapter 6, we analyse experimentally the performance figures of our framework. Finally, Chapter 7 position our work w.r.t. related systems and Chapter 8 summarizes our contributions and future work.

Chapter 2

RDF/S based PDMS

In our framework, we consider that every peer provides descriptions about information resources available in a P2P network that conform to a number of community schemas (e.g., for e-learning, e-services, etc.). Peers employing the same schema to construct such descriptions in their local base belong essentially to the same *Semantic Overlay Network (SON)* [CGM03]. The notion of SONs appears to be an intuitive way to cluster together peers sharing the same model for a particular domain or application for expressing useful queries and exchange information with others. Of course, a peer may belong to more than one SON, depending on the semantics of its base while it may host only a part of the semi-structured descriptions available in the network. In our context, a PDMS is the union of a number of SONs where queries are answered with data residing at peer bases belonging to the same SON as the query. Moreover, if there exist semantic mappings between the SONs of a PDMS, queries are answered with data residing at peer bases that are reached through the paths of those mappings.

In order to design an effective and efficient P2P query routing mechanism we need to address the following issues: (a) how can a SON be defined? (b)

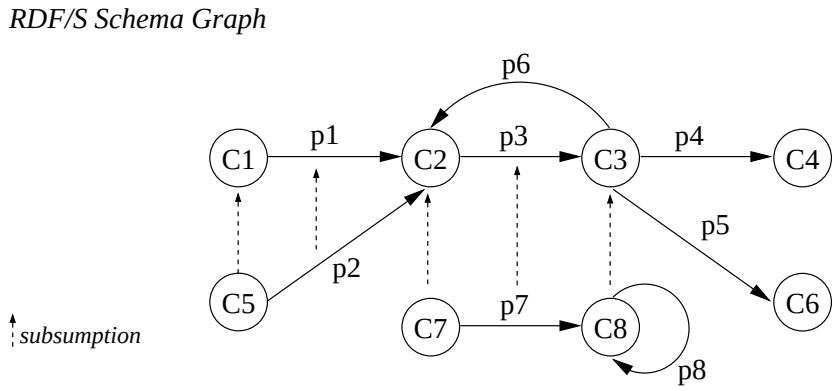


Figure 2.1: An RDF/S schema graph

how do peers advertise their bases in a SON? (c) how do peers formulate queries in a SON and finally (d) how do peers decide which views of a SON match their queries? In the following sections, we will present the main design choices of our framework in response to the above issues.

2.1 RDF/S schemas

A natural candidate for representing descriptive data (ranging from simple structured vocabularies to complex reference models [MACP02]) about various information resources available in a SON is the Resource Description Framework and Schema Language (RDF/S) [RDF]. The core primitives of RDF/S schemas are classes and properties. Classes describe general concepts or entities. Properties describe characteristics of classes or relationships between classes. Both classes and properties may be related through subsumption. Every property defined in an RDF/S schema has a *domain class* (i.e., the class that has this property) and a *range class* (i.e., the value of this property). A property and its domain and range classes form a *schema triple*, denoted by $(domain(p),\ p,\ range(p))$. An RDF/S schema is a set of schema

triples forming a directed labelled (multi)graph, called in the sequel *RDF/S schema graph*. For example, consider the RDF/S schema graph shown in Figure 2.1. The circular nodes are labeled with class names (e.g., $C2$, $C3$), while the solid edges with property names (e.g., $p3$). The dashed edges represent the subsumption relationships of classes (e.g., between $C7$ and $C2$) or properties (e.g., between $p7$ and $p3$). Formally, an RDF/S schema graph is defined as follows.

Definition 2.1 *An RDF/S schema graph is a directed multigraph* $\mathcal{R} = (\{C \cup L\}, P, \prec^c, \prec^p)$*, where:*

1. *C is a set of nodes labelled with an RDF/S class name.*
2. *L is a set of nodes labelled with a data type (RDF/S literals).*
3. *P is a set of edges (c_1, p, c_2) from a node c_1 to a c_2 labelled with a property p, where domain$(p) = c_1$ with $c_1 \in C$ and range$(p) = c_2$ with $c_2 \in C \cup L$.*
4. *$\prec^c$ is a partial order imposed on nodes in C (RDF/S class subsumption).*
5. *$\prec^p$ is a partial order imposed on edges in P (RDF/S property subsumption).*

The framework presented in this thesis can be applied to a wide-range of application needs: from one SON defined by a unique RDF/S schema, to a SON defined by several interconnected RDF/S schemas, to several SONs defined by different RDF/S schemas. This is due to the expressiveness of the RDF/S data model which (a) allows easy reuse or refinement of descriptive schemas employed by peers through subsumption of both classes and

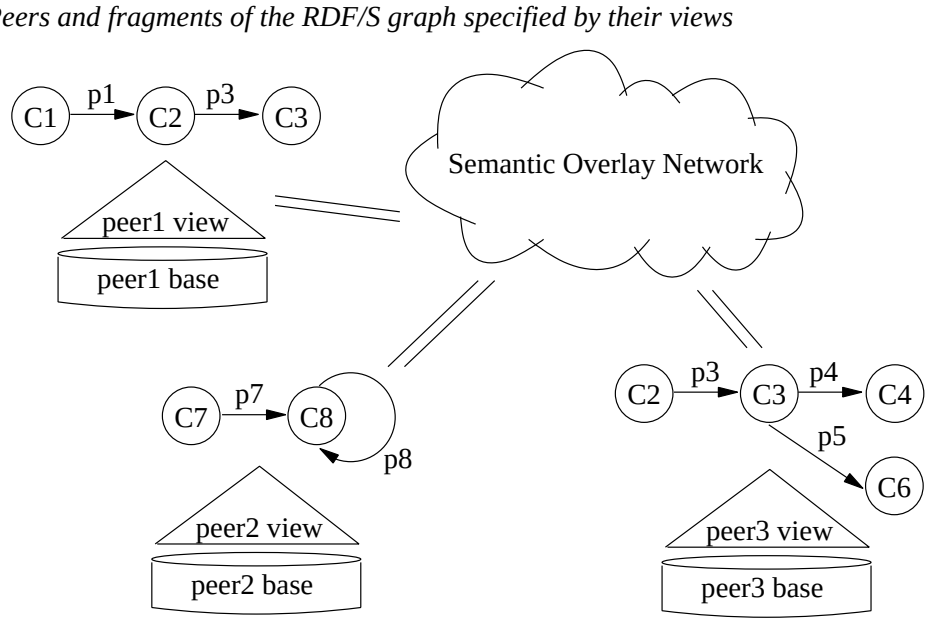


Figure 2.2: Peers belonging to the same SON and their views

properties; (b) permits irregular heterogeneous descriptions in the sense that a resource may be multiply classified under several classes from one or several peer schemas (identified by appropriate namespaces) and (c) extends the scope of a resource description beyond the physical boundaries of an XML file hosted by a peer.

2.2 RDF/S peer base advertisements and queries

Each peer should be able to advertise the content of its local base to others with respect to the RDF/S schemas of the SONs they belong to. Using these advertisements a peer can become aware of the data hosted in remote peer bases. However, since an RDF/S schema may contain numerous classes and

properties not necessarily populated in a peer base, we need a fine-grained definition of schema-based advertisements. To this end, we employ views to specify the *fragment* of an RDF/S schema graph for which all classes and properties are populated in a peer local base. Figure 2.2 illustrate three peers belonging to the SON defined by the RDF/S schema graph of Figure 2.1, and their views. Each view specifies a different fragment of the RDF/S schema graph which are used by the peers to advertise their bases. In a similar way, peers can retrieve data from the PDMS by issuing queries, which also specify a particular RDF/S schema *fragment* of interest.

Queries in our framework are formulated in RQL [KAC+02], a full-fledged RDF query language which provides sophisticated pattern matching facilities against RDF/S schema and data graphs. RQL queries allow us to retrieve the contents of any peer base, namely resources classified under classes or associated to other resources using properties defined in the RDF/S schema. It is worth noticing that RQL queries incur both *intensional* (i.e., schema) and *extensional* (i.e., data) filtering conditions. Additionally, peers employ RVL [MTCP03], an extension of RQL, for defining views to specify the fragments of an RDF/S schema for which all classes and properties are populated in a peer base. Both languages employ patterns to extract the RDF/S schema graph fragments which are relevant to the data requested by a query/view. Table 2.1 summarizes the basic class and property *path patterns*, which can be employed in order to formulate complex RQL/RVL query/view patterns (capital letters denote variables, and small letters denote constants). With the exception of the RQL/RVL distinction between exact (denoted with ^) and extended pattern matching for class ($\hat{}c\{X\}$ and $c\{X\}$) and property ($\{X\}\hat{}p\{Y\}$ and $\{X\}p\{Y\}$) instances, all the other patterns are encountered in the majority of the RDF/S query languages. In the rest of this thesis we

Path Patterns	**Interpretation**
Class Path Patterns	
\$C	$\{c \mid c$ is a schema class$\}$
$c\{X\}$	$\{[c, x] \mid c$ a schema class, x in the interpretation of class $c\}$
$\hat{}c\{X\}$	$\{[c, x] \mid c$ a schema class, x in the exact interpretation of class $c\}$
Property Path Patterns	
@P	$\{p \mid p$ is a schema property$\}$
$\{X\}p\{Y\}$	$\{[x, p, y] \mid p$ is a schema property, $[x, y]$ in the interpretation of property $p\}$
$\{X\}\hat{}p\{Y\}$	$\{ [x, p, y] \mid p$ is a schema property, $[x, y]$ in the exact interpretation of property $p\}$
$\{X; [\hat{}]c\}[\hat{}]p\{Y; [\hat{}]d\}$	$\{[x, c, p, y, d] \mid p$ is a schema property, c, d are schema classes, c is a subclass of p's domain, d is a subclass of p's range, x is in the (exact) interpretation of c, y is in the (exact) interpretation of d, $[x, y]$ is in the (exact) interpretation of $p\}$
$\{X; [\hat{}]c\}@P\{Y; [\hat{}]d\}$	$\{[x, c, p, y, d] \mid p$ is a schema property, c, d are schema classes, x is in the (exact) interpretation of c, y is in the (exact) interpretation of d, $[x, y]$ is in the interpretation of $p\}$

Table 2.1: Class and property query/view patterns

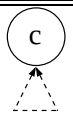
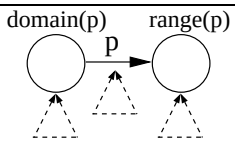
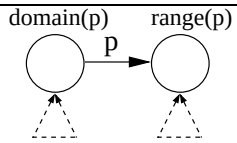
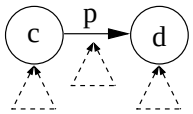
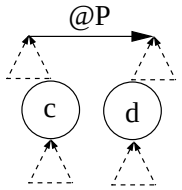
Pattern	*Graph Fragment*	*Pattern*	*Graph Fragment*
$c\{X\}$		$\hat{}c\{X\}$	
$\{X\}p\{Y\}$		$\{X\}\hat{}p\{Y\}$	
$\{X;c\}p\{Y;d\}$		$\{X;c\}@P\{Y;d\}$	

Table 2.2: Graph fragments specified by query/view patterns

stick on the notion of RDF/S schema fragments specified by these patterns, rather than their syntax on RQL or RVL.

Definition 2.2 *Given an RDF/S schema graph* $\mathcal{R} = (\{C \cup L\}, P, \prec^c, \prec^p)$*, a fragment specified by a query or view pattern over* $\mathcal{R}$ *is a subgraph* $\mathcal{R}' = (C', P')$ *such that* $C' \subseteq C$ *and* $P' \subseteq P$*.*

Table 2.2 illustrates the fragments of the SON RDF/S schema graph specified by the patterns of Table 2.1. More precisely, the pattern $c\{X\}$ can be used to retrieve all classes that are instances of class c or any class subsumed by c, while $\hat{}c\{X\}$ consider only classes that are in the exact interpretation of class c (no subsumed classes are considered). The pattern $\{X\}p\{Y\}$ can be used to retrieve all the instances (X, Y) of the domain and range classes of property p. Note that this pattern takes also into account the class and property subsumption relationships (denoted by the dashed triangles) to include in the result transitive instances of domain/range classes. Pattern $\{X\}\hat{}p\{Y\}$

is similar to the previous, with the exception of considering only the exact interpretation of property p (i.e., no properties subsumed by p will be included in the result). The next pattern, $\{X;c\}p\{Y;d\}$, retrieves all the instances (X,Y) of the class c and d, where c and d are subclasses of the domain and range class of property p respectively. Note that c, p or d can appear with a leading $\hat{}$ donating the exact interpretation, and if so, the dashed triangle will be omitted in each of the corresponding class or property. Finally, the pattern $\{X;c\}@P\{Y;d\}$ will return all the properties relating instances of the classes c and d, respectively. These properties can be either defined to have c and d as domain and range classes respectively but also any of the classes subsuming them. Again, the classes can appear in the pattern with a leading $\hat{}$.

We can easily observe the similarity in the intensional representation of both peer base advertisements and query requests as RDF/S schema graph fragments. By representing in the same logical framework what data are requested by a SON (i.e., queries) and what data are actually hosted in each peer base of the SON (i.e., views), we can easily understand the data partitioning (horizontal, vertical, mixed) in remote peers relative to a query. Moreover, this framework can be easily extended to reformulate queries expressed against a SON RDF/S schema in terms of the heterogeneous schemas or data models (e.g., relational, XML) employed locally by the peer bases [CKK+03].

2.3 RDF/S query and view subsumption

In order to decide which peer advertisements match a SON query, we need to check whether the classes and properties of the RDF/S schema fragments

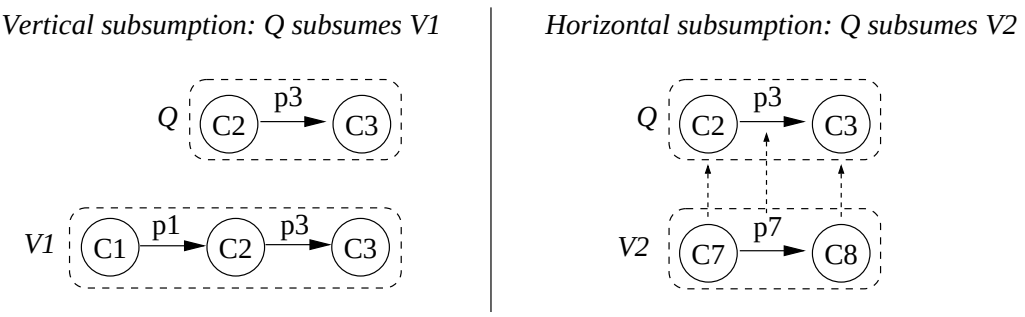


Figure 2.3: Two cases of view subsumption.

specified by the corresponding peer views are subsumed by those of the query. As studied in [SKCT05] checking containment of conjunctive RVL views and RQL queries involving arbitrary RDF/S schema and data patters is an NP-complete problem. For this reason we restrict our framework to the patterns illustrated in the previous section that return data according to a known schema fragment. We define subsumption between RDF/S schema graph fragments as follows:

Definition 2.3 *Let the RDF/S schema graph* $\mathcal{R} = (C, L, P, \prec^c, \prec^p)$. *Let also* $\mathcal{R}' = (C_1, P_1)$ *and* $\mathcal{R}'' = (C_2, P_2)$ *be two fragments of* $\mathcal{R}$, *specified by a query pattern* Q *and a view pattern* V, *respectively* ($C_1, C_2 \subseteq C$ *and* $P_1, P_2 \subseteq P$). Q *subsumes* V *(or* V *is subsumed by* Q*) if:*

1. $\forall c_1 \in C_1,\ \exists c_2 \in C_2,\ c_1 = c_2$ *or* $c_2 \prec^c c_1$, *and*

2. $\forall p_1 \in P_1,\ \exists p_2 \in P_2,\ p_1 = p_2$ *or* $p_2 \prec^p p_1$.

Notice that in the above definition, all classes/properties in Q must be present or subsume a class/property in V. However, V may have additional classes and properties. Figure 2.3 illustrates two different subsumption cases. In the left part of Figure 2.3, query Q *vertically* subsumes view $V1$ in the sense that $V1$ has property $p1$ with domain class $C1$ that are not present in

Q. However $V1$ is subsumed by Q since it contains a fragment that matches Q. On the right part of Figure 2.3, Q *horizontally* subsumes $V2$ since all classes and properties in $V2$ are subsumed by the classes and properties of Q. A query may subsume a view in either the above two ways or in any combination of them. Therefore, we need efficient support to decide subsumption of RDF/S schema graph fragments. For this purpose, we will present in the next chapter an encoding allowing to check whether an RDF/S schema fragment is subsumed by another, in linear time to the number of schema triples of the fragments.

Chapter 3

An Encoding for RDF/S Schema Graph Fragments

This chapter introduces a succinct representation of RDF/S schema graphs, based on a structure called *Adjacency and Subsumption Cube* (in short *Adj-Sub Cube*) allowing to encode fragments of arbitrary size and structural form (i.e., linear, tree or graph) of the original RDF/S schema graph. The *Adj-Sub Cube* is a structure that does not need to be implemented by any peer. Instead, it's used to derive an encoding function for RDF/S schema graph fragments involved in a SON. An *AdjSub Cube* provides (a) adjacency information for nodes (i.e., whether a class is related to another one via a certain property) and (b) subsumption information for classes and properties (i.e., whether a class/property subsumes another).

An adjacency matrix of a graph $G = (V, E)$ is a $n \times n$ matrix A, such that $A(i, j) = 1$ if $(i, j) \in E$ or 0 otherwise. Such a structure is not suitable for representing an RDF/S schema graph because there may be more than one edges connecting the same vertices (provided that these edges are of different labels) and there may exist self-loops. For example, properties $p3$ and $p6$ in

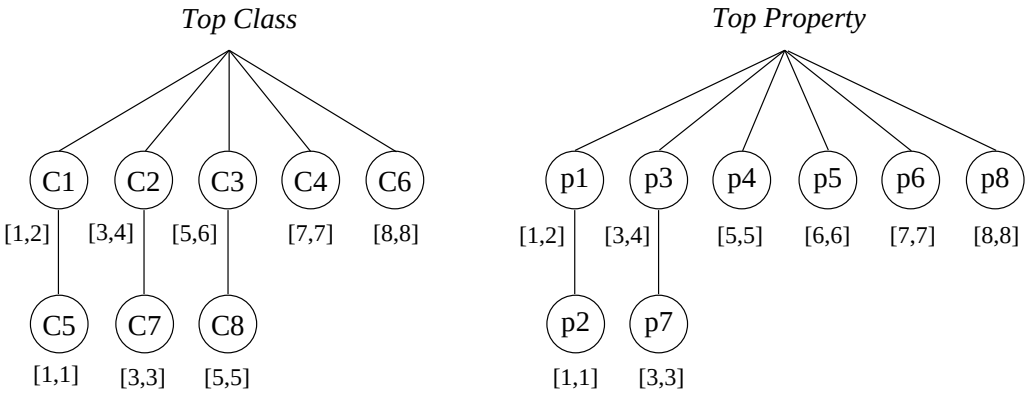


Figure 3.1: Class and Property subsumption hierarchies

Figure 2.1 are both adjacent to classes $C2$ and $C3$, while property $p8$ is a self-loop. The *AdjSub Cube* extends the concept of adjacency matrix by adding a third dimension to map labeled edges (i.e., the properties). The first dimension (vertical) represents the nodes that appear as the domain classes of a property, the second (horizontal) the nodes that appear as the range classes of a property and the third the labels of the properties. Moreover, it imposes an ordering of classes and properties on each dimension based on an interval encoding of the RDF/S class and property subsumption hierarchies.

In general, an interval encoding over a subsumption hierarchy is maintained using labels of the form [*start, end*] such that every interval of a child node is contained in the interval of its parent. In this paper we employ the encoding of [ABJ89] where a tree node u is labeled with [$index(u)$, $post(u)$]: $post(u)$ is the number assigned to u when a postorder tree traversal is considered, while $index(u)$ is the lowest of the *post* numbers assigned to u's descendants. Note that $index(u) \leq post(u)$ and that $u \prec^c v$ (or $u \prec^p v$) iff $index(u) \geq index(v) \wedge post(u) < post(v)$. In addition, the number of nodes of a sub-tree rooted at node u is equal to $post(u) - index(u)$. Figure 3.1 illustrates

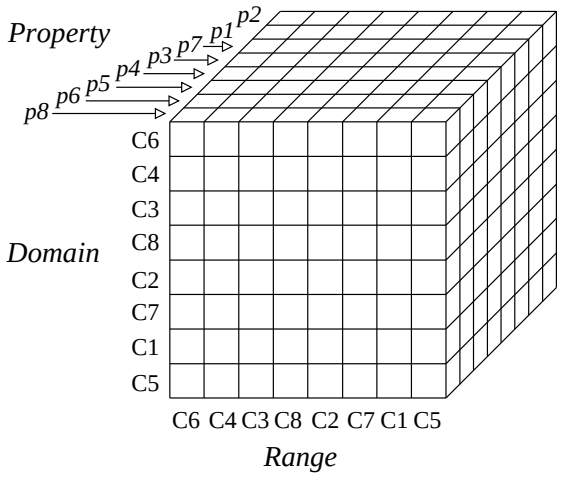


Figure 3.2: AdjSub Cube for an RDF/S schema

the interval encoding of the class and property subsumption hierarchies given in the SON RDF/S schema of Figure 2.1. An *AdjSub Cube* exploits the *post* numbers of the classes and properties to arrange them in each dimension. It follows that if we organize classes (properties) in the inverse order from the one obtained in the postorder, subsumed classes (properties) will succeed the subsuming classes (properties).

For simplicity, we consider only tree shaped subsumption hierarchies although this work can be extended to DAGs [CPST03]. Such an extension would be to organize classes and properties in the reverse order of that obtained by a Depth First Search (DFS) of the DAG hierarchy. However, in this case the above mentioned properties of the interval encodings are not valid and more, and more checking must be done to ensure that a class (or a property) is subsumed by another. This checking consist of a simple comparison between the additional interval labels employed in case of DAGs [CPST03].

Given an RDF/S schema graph $\mathcal{R}$, we define the corresponding *AdjSub Cube AS* as follows: for every *schema triple* (d, p, r) of $\mathcal{R}$, where p is the

property, d the domain class and r the range class of the property, we set cell $AS_{ijk} = 1$, where $i = |C| - post(d)$, $j = |C| - post(r)$ and $k = |P| - post(p)$. The remaining cells in the *AdjSub Cube* are set to 0. Figure 3.2 illustrates an example of the *AdjSub Cube* that corresponds to the RDF/S schema graph presented in Figure 2.1. We next define formally the *AdjSub Cube*.

Definition 3.1 *Let an RDF/S schema graph $\mathcal{R} = (C, L, P, \prec^c, \prec^p)$, and $d, r \in C$, $p \in P$. An AdjSub Cube AS for $\mathcal{R}$ is an $|C| \times |C| \times |P|$ bitmap such that:*

$$AS(i,j,k) = \begin{cases} 1 & \text{if } domain(p) = d \text{ and } range(p) = r \\ 0 & \text{otherwise} \end{cases}$$

where $i = |C| - post(d)$, $j = |C| - post(r)$ and $k = |P| - post(p)$.

Any fragment of the original RDF/S schema graph, can be represented in the same *AdjSub Cube* created for this schema, by setting to 1 only those cells of the cube that correspond to the schema triples of the fragment. We can enumerate all cells in the *AdjSub Cube* based on their position through the function $pos(AS_{ijk}) = k \times |C|^2 + i \times |C| + j$, where $i, j = 0, 1, \ldots, |C| - 1$ and $k = 0, 1, \ldots, |P| - 1$. Based on this enumeration, every fragment is encoded by assigning a unique number N as defined below:

Definition 3.2 *Let an RDF/S schema graph $\mathcal{R} = (C, L, P, \prec^c, \prec^p)$ and the corresponding AdjSub Cube AS. Every fragment of $\mathcal{R}$ represented in AS, is encoded by assigning a unique number N, such that*

$$N = a_{L-1}2^{L-1} + \ldots + a_1 2^1 + a_0 2^0 \text{ and } a_n = AS_{ijk},$$

where $n = pos(AS_{ijk})$, and $L = |C| \times |C| \times |P|$ the size of the AdjSub Cube AS.

encode(view V)

input: a view V
output: an encoding for view V

1: $C \leftarrow$ number of classes in the RDF/S schema
2: $P \leftarrow$ number of properties in the RDF/S schema
3: $encode = 0$
4: **for** every schema triple t in V
5: $pos = (P - post(t.prop)) \times C^2 + (C - post(t.domain)) \times C +$
$(C - post(t.range))$
6: $encode = encode + 2^{pos}$
7: **end for**
8: **return** *encode*

Figure 3.3: Encoding Algorithm for view V

The *unique number* N that encodes an RDF/S schema graph fragment is a number, where the coefficient a_n of the factor 2^n is set to 0 or 1 depending on the value of the cell, whose position is $pos(AS_{ijk}) = n$. One may compute N if for every *schema triple* (d, p, r) of the fragment computes its corresponding position in the *AdjSub Cube*. The complete algorithm is given in Figure 3.3.

Since N may be a fairly large number, a simple way to store N is with a set of integers $\{n_1, n_2, n_3, \ldots\}$ such that each n_i is the $pos(AS_{ijk})$ of each *schema triple* in the fragment. For example, consider view $V1$ of Figure 2.3 composed of the *schema triples* $t_1 = (C1, p1, C2)$ and $t_2 = (C2, p3, C3)$. For the first *schema triple* t_1 we have: $i = |C| - post(C1) = 8 - 2 = 6$, $j = |C| - post(C2) = 8 - 4 = 4$, $k = |P| - post(p1) = 8 - 2 = 6$ and $pos(t_1) = k \cdot 8^2 + i \cdot 8 + j = 6 \cdot 64 + 6 \cdot 8 + 4 = 436$. Consequently, for the *schema triple* t_2, $pos(t_2) = k \cdot 8^2 + i \cdot 8 + j = 4 \cdot 64 + 4 \cdot 8 + 2 = 290$. Thus, the unique number of view $V1$ is $N(V1) = 2^{436} + 2^{290} \equiv \{436, 290\}$. Likewise,

the unique number of view $V2$ and query Q of Figure 2.3 is $N(V2) = 2^{363}$ and $N(Q) = 2^{290}$, respectively. We can observe that if a view is subsumed (horizontally or vertically) by another then it holds that the unique number of the subsumed view is greater than the unique number of the subsuming view. For example, query Q subsumes vertically view $V1$ and horizontally view $V2$. For both cases of subsumption it holds that $N(V1) > N(Q)$ and $N(V2) > N(Q)$.

Given the above encoding, we can decide if a fragment of an RDF/S schema graph is subsumed by another in linear time to the size of the fragments.

Theorem 3.1 *Given two connected fragments $\mathcal{R}', \mathcal{R}''$ of an RDF/S schema graph $\mathcal{R}$ and their unique numbers defined by the above encoding $N(\mathcal{R}') = 2^{n_1} + 2^{n_2} + \ldots + 2^{n_k} \equiv \{n_1, n_2, \cdots, n_k\}$ and $N(\mathcal{R}'') = 2^{n'_1} + 2^{n'_2} + \ldots + 2^{n'_l} \equiv \{n'_1, n'_2, \cdots, n'_l\}$, respectively, it holds that:*

1. *if $\mathcal{R}'$ subsumes $\mathcal{R}'' \Rightarrow N(\mathcal{R}'') > N(\mathcal{R}')$.*

2. *if $\forall n_i \in \{n_1, n_2, \ldots\}, \exists n''_i = n'_j, n'_j \in \{n'_1, n'_2, \ldots\} \setminus \{n''_1, \ldots, n''_{i-1}\}$: $n'_j \in \bigcup_\delta \bigcup_\lambda [n_i + \lambda|C| + \delta|C|^2, n_i + \lambda|C| + \delta|C|^2 + \kappa] \Rightarrow \mathcal{R}'$ subsumes $\mathcal{R}''$,*

where:

$\delta = post(prop(t)) - index(prop(t))$,

$\lambda = post(domain(t)) - index(domain(t))$,

$\kappa = post(range(t)) - index(range(t))$

and t is the schema triple corresponding to the cell with pos(AS)=n_i.

Proof. If $\mathcal{R}'$ subsumes $\mathcal{R}''$ then from the construction of the *AdjSub Cube* it holds that $N(\mathcal{R}'') > N(\mathcal{R}')$. Given two fragments $\mathcal{R}', \mathcal{R}''$, in order for

$\mathcal{R}'$ to subsume $\mathcal{R}''$, we need to check whether for every schema triple in $\mathcal{R}'$ ($\forall n_i$), there exists a subsumed schema triple in $\mathcal{R}''$ ($\exists n'_j$). Given that the schema triple corresponding to the cell n'_j is subsumed by the schema triple corresponding to the cell n_i, n'_j is positioned somewhere in a sub-cube of the *AdjSub Cube*. This sub-cube is confined by the subsumed triples of n_i, which are represented in the cells of the sub-cube that spans *post(range*$(t)) -$ *index(range*$(t))$ on the range dimension, *post(domain*$(t)) -$ *index(domain*$(t))$ on the domain dimension and *post(prop*$(t)) -$ *index(prop*$(t))$ on the property dimension.

□

The complexity to decide if two views are subsumed given their unique numbers is $O(nk)$ where n and k is the number of schema triples of the views, respectively. This is due to the fact that for every schema triple in $\mathcal{R}'$ (n schema triples) we must search k schema triples of $\mathcal{R}''$ to find a match.

Chapter 4

A DHT-Framework for RDF/S Queries

Structured P2P systems based on Distributed Hash Tables (DHTs) can support large, highly distributed networks while ensuring a fair load distribution among peers at the cost of an extra message overhead when peers join or leave the network. A popular DHT-based protocol for storing and retrieving pairs of *(key, data)* is Chord [SMK+01]. More precisely, the main service supported by Chord is *lookup(key)*, which returns in at most $O(\log n)$ routing hops (i.e., network messages) the peer's address that is responsible for storing the pair *(key, data)*. Peers are associated with keys through their identifiers. A peer's identifier is chosen by hashing the peer's IP address, while a *key identifier* is produced by hashing the key. Identifiers are ordered on an identifier circle *modulo* 2^m, called the *Chord ring*. Key k is assigned to the first peer whose identifier is equal to or follows the identifier of k in the identifier circle. This peer is called the *successor peer* of key k, denoted by *successor(k)*. To locate a key in the Chord ring, each peer maintains a routing table, called the *finger table*, where the i^{th} entry contains the identifier of the first peer that succeeds

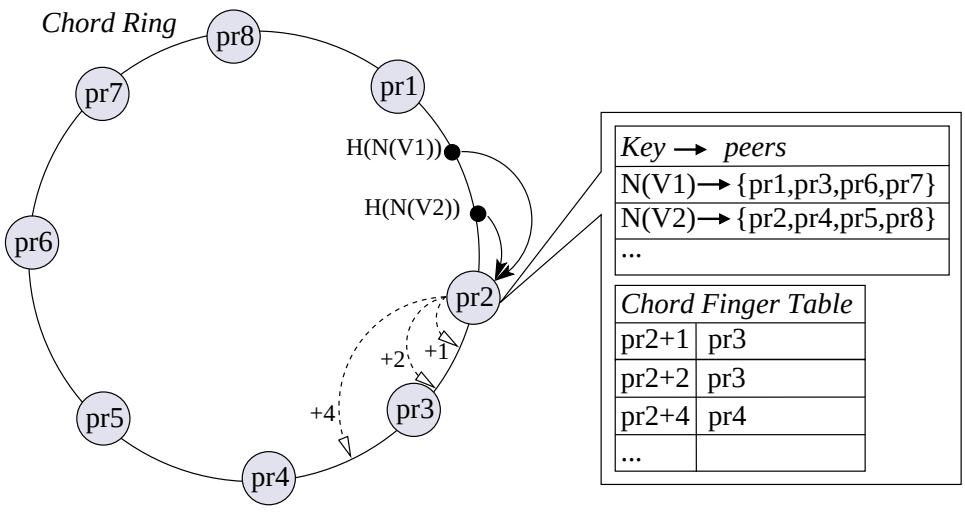


Figure 4.1: A Chord ring with eight peers.

its identifier by at least 2^{i-1}.

In our context, we build a *DHT-based schema index* where a key is the unique number of the RDF/S fragment specified by the view of a peer. Rather than data objects a key designates peer IPs populating the corresponding fragment. To produce a *key identifier* and place it on the *Chord ring*, an order preserving hash function is used to maintain the ordering over subsumed views given by the *AdjSub Cube*. For example, if $N = \{n_1, n_2, n_3\}$ a simple order preserving hash function is $H(N) = (n_1 \cdot n_2 \cdot n_3) \mod 2^m$. Figure 4.1 illustrates eight peers $(pr1, pr2, \ldots, pr8)$ that have published views of the same SON RDF/S schema and thus sharing the same Chord ring. Peer $pr2$ is the successor peer of views $V1$ and $V2$ since the hash values of the unique number of the views are between the identifiers of peers $pr1$ and $pr2$. Peer $pr2$ maintains a table that associates the unique numbers of views $V1$ and $V2$ with peers whose bases populate these views. Moreover, peer $pr2$ route requests according to the finger table as specified by the Chord protocol.

To summarize, each peer stores pairs of the form *(view,* {*peers*}*)* and

replies to a *lookup(view)* request with the set of peers that had advertised the specific view. However, there might be views hashing to the same key identifier or the same keys may correspond to views that are defined over different SONs. The first problem can be easily bypassed by sending along with the lookup request, the unique number of the encoded view. The second problem can be addressed by distinguishing the lookup requests via the unique namespace of the RDF/S schema defining a SON. When SONs are interconnected, the *AdjSub Cube* is defined using the class and property hierarchies of all involved RDF/S schemas. Next, we describe how our DHT-based infrastructure evolves when peers join or leave the network or even update their views.

4.1 Peer joins, departures and updates

Each peer joining the network advertises through a view the fragment of the RDF/S schema which is actually populated in its local base. Recall that a peer is able to not only answer queries that match exactly its view, but also any of its fragments (i.e., views that vertically subsume its view). When the joining peer wishes to inform about its capability to answer queries related to a vertically subsuming view, it issues *lookup(view)* requests to identify the successors responsible for the subsuming views accompanied by a *store(view, IP)* request. Moreover, Chord provides a mechanism for key reallocation for inserting the newly added peer to the DHT index. The predecessor of this new peer sends the *(view,* $\{$*peers*$\}$*)* pairs for which it will be responsible from now on. For example, Figure 4.2 illustrates peer $pr3$ that has view $V0$ and all views that vertically subsume $V0$, namely $V1$ to $V6$. The upper right part of Figure 4.2 illustrates the peers that are contacted on the *Chord ring* when

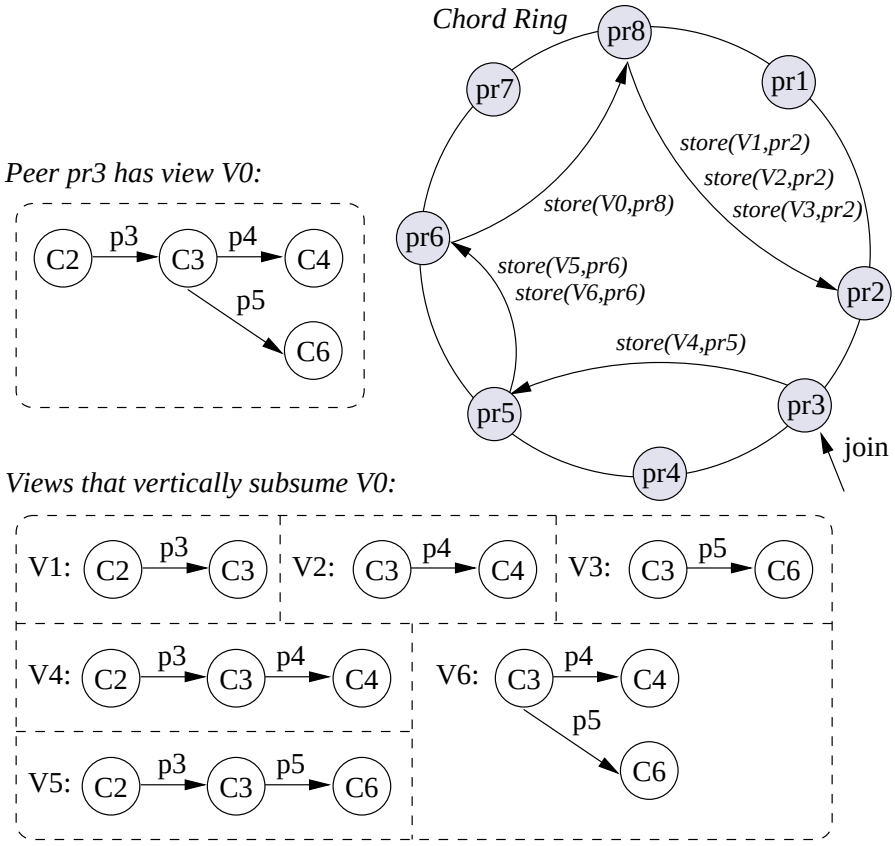


Figure 4.2: Peer *pr3* joins the network. It advertise its view $V0$ and the vertically subsuming views

peer $pr3$ joins the network. Since peer $pr2$ is the successor peer of the key identifiers $H(N(V1))$, $H(N(V2))$ and $H(N(V3))$ it receives a request from peer $pr3$ to advertise the views $V1$, $V2$ and $V3$ and stores the IP address of peer $pr3$ to the set of peers associated with the views $V1$, $V2$ and $V3$.

Figure 4.3 outlines an algorithm that computes all the views that vertically subsume a given view. The algorithm takes as input a view V and decompose it into its constituent schema triples (line 2). Next it creates the next set of subsuming views by adding to the previous created views a new schema triple from the initial view. More precisely, starting from a set T of

VSubViews(view V)

input: a view V
output: all views that vertically subsume view V

```
1: E = ∅
2: S = {the schema triples of view V}
3: E = E ∪ S, T = S, L = ∅
4: while T ≠ ∅
5:         for every view V_t in T
6:             for every schema triple st in S
7:                 if st joins with V_t
8:                     create new view V_n = V_t ⋈ st
9:                     L = L ∪ {V_n}
10:                end if
11:            end for
12:        end for
13:        E = E ∪ L, T = L, L = ∅
14: end while
15: return E
```

Figure 4.3: Algorithm to compute all vertically subsuming views

views with 1 schema triple (line 3), it creates all views with 2 schema triples and stores them to set L (lines 5-9). Next, the newly created views are added to E, while T is set to be equal with L (line 13). The algorithm continues with T now containing all subsuming views consistent of 2 schema triples and thus producing views with 3 schema triple. This procedure is repeated until no new views are created (line 4). We should point out that this algorithm may produce the same view more than ones (e.g., in the presence of cycles in the original view) but with an appropriate data structure, such as a hash table and the unique number of the view used as key, it is easy to identify these duplicates. In general, from our experience while implementing the above algorithm (as well as the others presented in the sequel) we conclude that the most suitable data structure for storing views are *hash sets* where the hash key is the unique number of the view.

The number of subsuming views for a view consistent of n schema triples varies from $\frac{(n+1)\cdot n}{2}$ in case of a linear view, until $\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n$ in case of a star shaped view where all schema triples are joined together in a single class. The algorithm presented in Figure 4.3 runs in $O(N)$ time, where N is the number of vertical subsuming views, since lines 8 and 9 are both implemented in constant time.

It should be stressed that a joining peer should advertise only the vertically and not the horizontally subsuming views. Advertising the horizontally subsuming views too, implies that queries are systematically extended to include peer data classified under subsumed classes and properties. However, this functionality is specified by the peer queries and not the peer views (i.e., denoted by the ˆ symbol in the query patterns, see Table 2.2). On the other hand, if vertically subsuming views were not explicitly advertised but considered during query processing, the lookup service would have to search a larger portion of the P2P network and thus incurs an increased cost in routing hops (i.e., to discover all keys that are a multiplier of the key associated with the requested view). It's clear that the cost of advertising those views is significantly less than trying to explicitly locate them each time a new lookup request is issued.

To ensure consistency of the DHT-index when a peer leaves the system, it must pass the key identifiers that holds to its successor peer. This operation is supported by the Chord protocol. Moreover, Chord provides services like *stabilize()* and *fix_fingers()* to support peers departure without any prior notification. Finally, the leaving peer must notify through appropriate messages all peers that have indexed its views to modify their stores.

The most frequent updates in such a system are the updates of the data of a peer base. However, such updates do not have any impact on the DHT-

based schema index, thus they do not cost nothing. The next most frequent type of updates are those done in a peer view. There two ways to deal with such updates. The simplest way is to consider an update as a departure followed by a join. Another solution for trivial updates (such as adding or removing a schema triple from a view) is to let the peer to decide what changes must be performed to the DHT-based schema index to reflect the update effect. Consider for example that $pr3$'s view $V0$ has been updated by adding the (C_1, p_1, C_2) schema triple. Then, the DHT-based schema index can be updated incrementally by advertising only the additional views subsuming the newly created one, since $V0, V1, \ldots, V6$ are still valid.

Finally, the most rare kind of updates are those done to the globally known schema. Such an update is handled as if it was a creation of a new SON. Peers that are aware of the updates may republish their views in the updated SON by leaving and rejoining the network according to the new schema. This implies that for a period of time there will be two SONs (i.e., for the old and the updated schema) and peers should gradually pass from one to another.

4.2 Lookup Service

In this section we present the *lookup service* which identifies all peers whose views are horizontally (or vertically) subsumed by a query. The first step is to lookup the view that specifies exactly the same fragment of the RDF/S schema graph as the one requested by the input query. This view is called the *strict view* and the peer that stores the *(strict view,* {*peers*}*)* pair, the *initial peer*. The initial peer is identified through the *lookup(strict view)* service of the original Chord protocol. The next step is to locate all other views

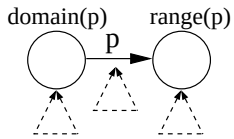
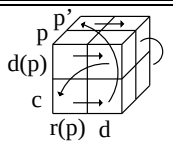
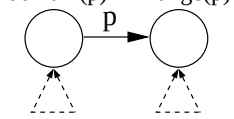
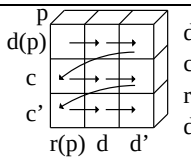
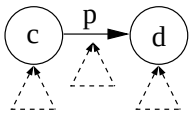
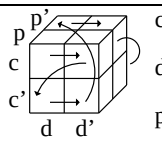
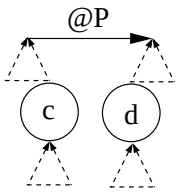
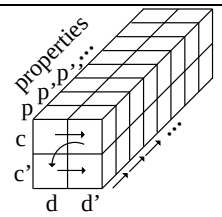
Pattern	*Graph Fragment*	*AdjSub Cube Regions*
$\{X\}p\{Y\}$		 d(p) = domain(p) c subclass of d(p) r(p) = range(p) d subclass of r(p) p' subproperty of p
$\{X\}\hat{}p\{Y\}$		 d(p) = domain(p) c,c' subclasses of d(p) r(p) = range(p) d,d' subclasses of r(p)
$\{X;c\}p\{Y;d\}$		 c,c' subclasasses of domain(p) d,d' subclasses of range(p) p' subproperty of p
$\{X;c\}@P\{Y;d\}$		

Table 4.1: AdjSub Cube traversal for RDF/S schema fragments.

that are horizontally subsumed by the strict view through the invocation of a *sublookup()* service issuing a sequence of lookup requests. The sequence in which these lookup requests are performed is very important since there must be no lookups that address preceding peers. The encoding and the order preserving hash function guarantees that the key identifiers of all the subsumed views will be greater than than the key identifier of the strict view. As a result, all peers storing pairs of horizontally subsumed views will succeed the initial peer in the Chord ring. Thus we avoid to travel all over the Chord ring to reach the destination peer. It should be stressed that the answer to a specific lookup request contains the peers whose views not only match strictly but also vertically subsume by the original query (see Section 4.1).

The main intuition for the *sublookup()* algorithm is that the sequence in which the views are looked up is given by the *AdjSub Cube* (introduced in Chapter 3). Different regions of the *AdjSub Cube* define different sequences of lookups that must be issued in order to discover the views which are horizontally subsumed by an input query. Table 4.1 illustrates four regions of the *AdjSub Cube* corresponding to the RDF/S schema graph fragments of the four basic patterns depicted in the left column of the table. The fragments represented by solid edges and nodes are essentially the views matching strictly patterns while the dashed triangles represent subsumed views.

For the first pattern ($\{X\}p\{Y\}$), the strict view to look up corresponds to the RDF/S schema fragment which comprises the default domain and range of property p. The corresponding region of the *AdjSub Cube* starts at the cell representing the $(d(p), p, r(p))$ schema triple and expands to all cells that represent a subsumed schema triple. The second pattern has the same strict view. However, since no subproperties are considered the region of the *AdjSub Cube* does not have to expand to the dimension of subproperties. The third pattern is similar to the first one, but instead of considering the default domain and range classes of property p, the traversal of the cube begins at the cell that has as domain the class c and as range the class d. The fourth row of Table 4.1 ($\{X;c\}@P\{Y;d\}$) illustrates a pattern ($\{X;c\}@P\{Y;d\}$) that requests all views that may have the class c (and all subclasses) as domain and the class d (and all subclasses) as range. In this case, the strict view is the one that has the top property (not visible in the *AdjSub Cube*), since we need to check all the properties defined in the corresponding RDF/S schema of the SON.

Starting from the cell that corresponds to the schema triple of the strict

view, the arrows on each *AdjSub Cube* region designate the sequence in which the next subsumed view is chosen from the *AdjSub Cube*. In order to always choose the view that has the immediate larger key identifier, we traverse the *AdjSub Cube* by first moving to the right (substituting the range class of the triple), then down (substituting the domain class) and finally inwards (substituting the property). In more complex patterns, for each substitution of either the domain, range or property of each schema triple we substitute recursively the domain, range and property of all of its remaining schema triples. We next describe two versions of the *sublookup()* algorithm suitable for a *stateless* and a *statefull* execution in a DHT-based network. The main difference is that in the case of the stateless execution multiple sublookup requests are issued simultaneous and autonomously from a set of peers while in the statefull case a single peer handles all the sublookup requests thus it keeps "state" information about the progress of the requests.

Figure 4.4 gives the pseudocode for the stateless version of the *sublookup()* algorithm. When a query is issued, the initial peer is identified through a *lookup(strict view)* request. Next, *sublookup(strict view, strict view, initial peer)* is invoked. In order to guarantee that no view is looked up twice, a *substit()* function checks if the given domain or range class has already been substituted. For every substitution done to either the domain, range or property of a schema triple, a new view V is created and looked up. The peer that stores the new view V creates a new instance of the *sublookup()* function and continues its execution until all schema triples of the view are marked. The statefull *sublookup()* algorithm is a simplified version of the stateless one. The initial peer instead of passing the execution of *sublookup()* to the succeeding peers, it computes all the horizontally subsumed views and issues series of lookup requests for each view. As we can see in Figure 4.5, an

sublookup(strict view V_{strict}, current view V, peer pr)

```
1: for all triples t in V that are not marked and the corresponding triple t' in V_strict
2:     r' ← low_{≺^c}(t'.range, t.prop.default_range)
3:     d' ← low_{≺^c}(t'.domain, t.prop.default_domain)
4:     r ← next class of t.range on the AdjSub Cube
5:     d ← next class of t.domain on the AdjSub Cube
6:     p ← next property of t.prop on the AdjSub Cube
7:     if substit(t.range) && (r ≺^c r')
8:       t.range ← r
9:     else if substit(t.domain) && (d ≺^c d')
10:      if substit(t.range)
11:        t.range ← r'
         end if
12:      t.domain ← d
13:    else if (p ≺^p t'.prop)
14:      if (t.range ⪯^c p.default_range || substit(t.range)) &&
         (t.domain ⪯^c p.default_domain || substit(t.domain))
15:        r'' ← min_{≺^c}(t'.range, p.default_range)
16:        d'' ← min_{≺^c}(t'.domain, p.default_domain)
17:        if substit(t.range)
18:          t.range ← r''
           end if
19:        if substit(t.domain)
20:          t.domain ← d''
           end if
21:        t.prop ← p
22:      else
23:        mark t and continue
         end if
24:    else
25:      mark t and continue
       end if
26:    pr ← lookup(V)
27:    sublookup(V_strict, V, pr)
28:    mark t
   end for
```

Figure 4.4: Distributed sublookup algorithm

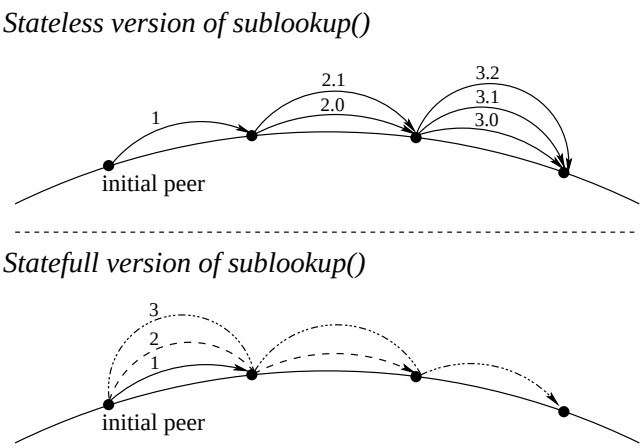


Figure 4.5: Routing hops on the Chord ring for the statefull and stateless version of the sublookup algorithm

advantage of the statefull version is that the whole execution is monitored by the initial peer. Thus, in the case where a succeeding peer stores more than one subsumed views, the statefull version will retrieve all views at once, in opposite to the stateless one contacting the same peer as many times as the number of subsumed views it stores. However, in the statefull version each new lookup request requires in principle more routing hops, since the next subsumed view will succeed the previous one in the Chord ring. It is worth noticing that the stateless lookup service scans the Chord ring in a highly parallel way in order to process a query. We believe that for queries involving a large number of peers the most important factor is the degree of parallelization of the lookup requests rather than the absolute number of routing hops. Moreover, such a parallelization reveals in a natural way the design choices that must be taken when an *interleaved* execution of the routing and planning phases is considered.

To conclude, in a network of N peers, each lookup requires $O(\log N)$ routing hops. Therefore, the total number of routing hops required to locate

all peer bases that can contribute to the evaluation of a query is: $O(\log N)$ to locate the initial peer plus $S \times O(\log N)$ to locate the S subsumed views, where S depends on the size of the involved subsumption hierarchy. However, especially for views with a small number of schema triples, the subsumed views are located in peers that are close to each other in the *Chord ring*, thus as we will see in Chapter 6 much less than $O(\log N)$ routing hops are required in practice for each lookup.

Chapter 5

Interleaved Query Routing and Planning

Query planning in a PDMS is responsible for generating a query plan according to the localization information about the relevant peer bases returned by the underlying lookup service. In addition, it is responsible for computing the cost of alternative operator order (e.g., join reordering) and execution policies (e.g., query or data shipping) for these plans. A PDMS exploits the data storage capabilities of peers but it should also exploit the available computation resources. Since query planning is a costly task, it is beneficial to consider a query planning strategy that assigns portions of a query plan to arbitrary peers. Moreover, in order to achieve autonomy in the computations undertaken by each peer, the generated subplans should be complete (i.e., in the sense that produce valid results) and their physical optimization should be decoupled from the enumeration of the possible logical query plans [DH02].

In our context, since the proposed lookup service identifies peer views that can actually answer an entire (sub)query pattern, we are interested in

an *interleaved execution of query routing and planning activities* in several iteration rounds. This interleaved execution leads to the creation and execution of multiple query plans that when "unioned" offer completeness in the results. Note that the generated plans at each round can be actually executed by the involved peers in order to obtain the first parts of the final query answer as fast as possible. Starting with the original query pattern, at each round smaller fragments are considered in order to find the relevant peers (routing activity) that can actually answer them. These fragments (annotated with the relevant peers) are "joined" (planning activity) to produce valid answers. In this context, the interleaved query processing terminates when the initial query is decomposed into its primary components (i.e., its schema triples). It should be also stressed that the interleaved execution favours intra-site joins (i.e., joins that take place in the same peer), since each query fragment is looked up as a whole and only peers that can fully answer it are contacted. Our approach of decomposing the initial query to smaller fragments is actually a top-down plan construction activity which is distributed over peers. In contrast to centralized approaches where all plans are created and executed by only one peer [ETB+03, NWS+03], our approach distributes the execution of a subplan to several peers. As a result, we achieve a workload balance over the peers during the costly phases of query planning and execution. As a matter of fact, the number of peers contributing to the processing of a query is increased as the complexity of a query is increased (i.e., as the number of subplans increases).

Since our approach favours peers that can answer the whole query, one may claim that the answers returned during the first rounds are more relevant and pertinent than those returned by the last rounds. In this context, a user may predefine the number of results to be returned (similar to a first-rows

optimization [AZ96]) and thus forces the PDMS to execute only few of the rounds involved in the interleaved execution. As we will see in Chapter 6, the user needs in this way to wait only few seconds until the first results are returned given that the first rounds of the interleaved execution involve less peers with a smaller cost to plan and execute subqueries than the next rounds. Such a mechanism allows to fix in advance the amount of resources that will be consumed (e.g., number of peers contacted, execution time, or even amount of results returned). Obtaining the complete query results in a PDMS of thousands of peers may be unrealistic in terms of time and resource consumption (network bandwidth, peer computation power, etc.). As a matter of fact, several recent works point out the need of a *best-effort* approach in evaluating a query over a P2P system in contrast to demanding a priori the complete answer [SRvdWB05, HHL+03].

5.1 Query Fragmentation and Planning

The interleaved query routing and planning involves a query fragmentation phase where the query is split into distinguished fragments. A component, called *fragmentor*, is involved in order to produce all possible fragmentations of the query. The *fragmentor* takes as input the number of joins which are required between the produced fragments in order to evaluate the original query. The output is all the possible fragmentations of the query for a specific number of joins. The *fragmentor* is a slight variation of the algorithm presented in Chapter 4 which computes all vertically subsuming views (Figure 4.3). At each iteration round of the interleaved routing and planning, the number of joins is increased by 1, starting from 0 joins, until the query is decomposed to its primitive components (i.e., schema triples). For example,

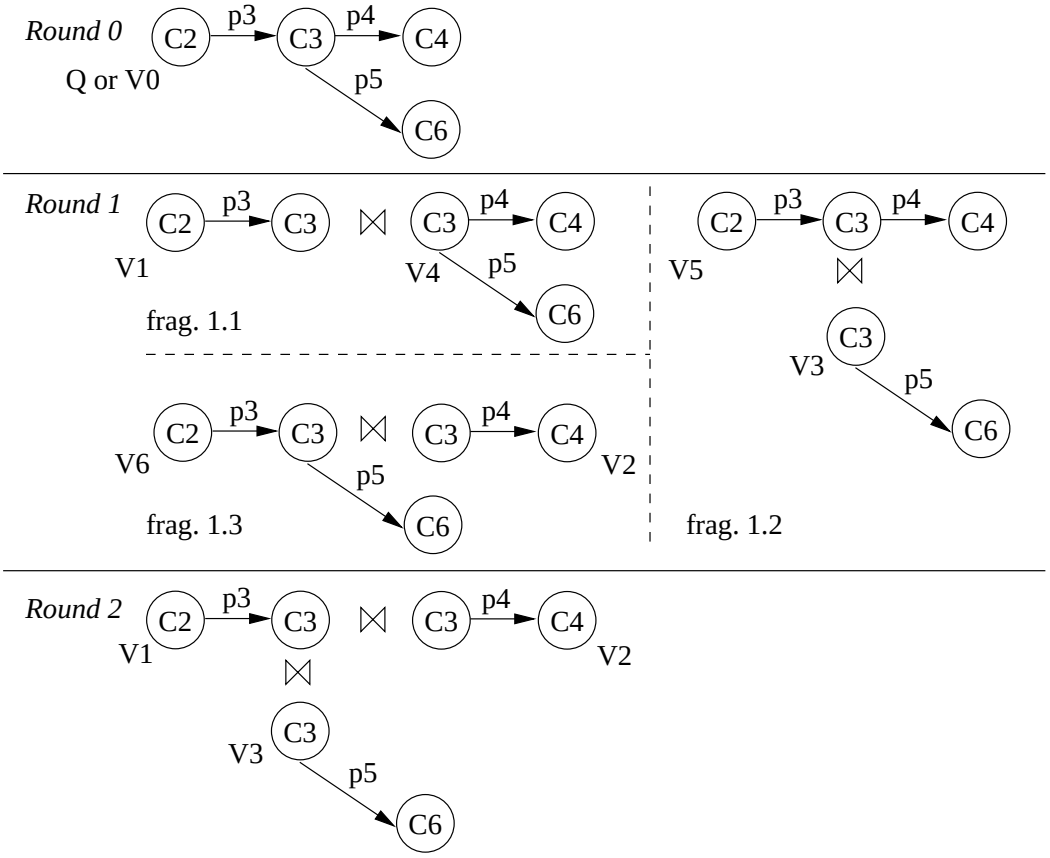


Figure 5.1: All possible fragmentations of query Q

if the query has n schema triples, the *fragmentor* starts from 0 joins (the whole query) until $n-1$ joins. As a result, the first rounds of the interleaved execution will consider peers that can answer the whole or at least a big portion of the initial query. Figure 5.1 illustrates the possible fragmentations of a query Q with 3 schema triples: $(C2, p3, C3)$, $(C3, p4, C4)$ and $(C3, p5, C6)$. Round 0 considers the whole query Q. Round 1 considers all possible fragmentations of Q with 1 join, namely $V1 \bowtie V4$, $V5 \bowtie V3$ and $V6 \bowtie V2$. Finally, round 2 considers only one fragmentation, which is essentially the decomposition of Q to its schema triples (2 joins).

Figure 5.2 illustrates how the fragmentations depicted in Figure 5.1 can be

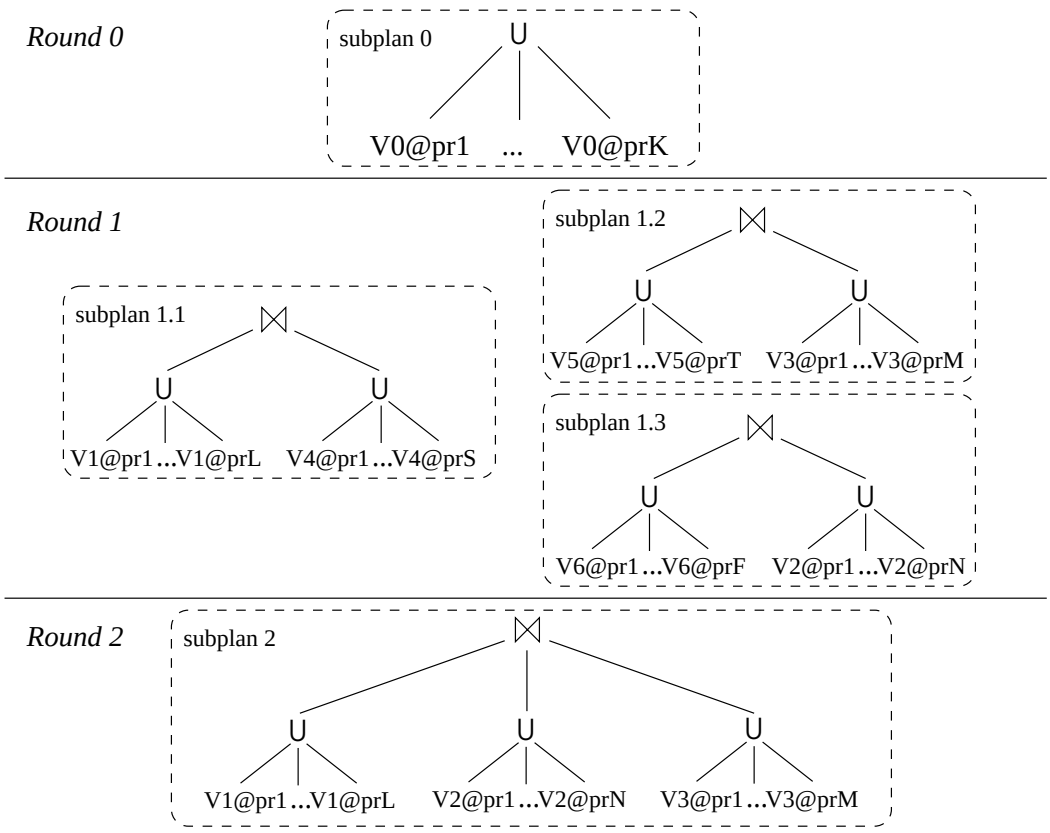


Figure 5.2: Plans for the query fragmentations depicted in Figure 5.1

planned. In this example, we assume that peers $pr1 \ldots prK$ have advertised view $V0$ and all vertically subsuming views (including views $V1, V2, \ldots V6$). Moreover, each of the views $V1, V2, \ldots V6$ may be advertised by additional peers, depicted with different letters (e.g., $S, T, \ldots$) in Figure 5.2. The partial results concerning a specific view that are obtained from these peers are "unioned" (horizontal distribution) and each of the unions are "joined" (vertical distribution) resulting to the subplans 0, 1.1, 1.2, 1.3, 2. The union of all subplans produce the final plan. We can easily observe from our example that taking into account the vertical distribution ensures *correctness* of query results (i.e., produce a valid answer), while horizontal distribution

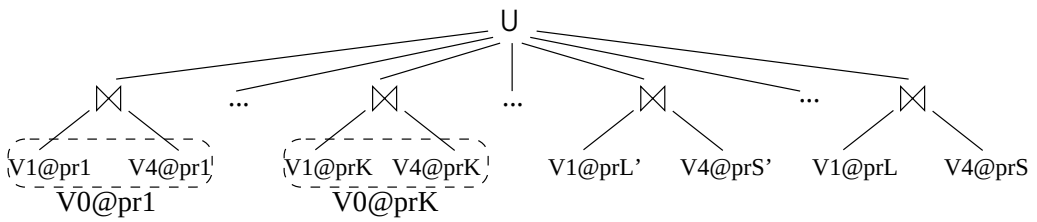


Figure 5.3: Subplan 1.1 optimized by applying the algebraic equivalence

in query plans favours *completeness* of query results (i.e., produce more and more valid answers).

It is worth noticing that the produced query subplans contain unions only at the bottom of the plan tree. We can push unions at the top and consequently move joins closer to the leaves. The following algebraic equivalence is applied over the subplans to distribute joins and unions over peers:

$$\bowtie (\cup(V_{11}, \ldots V_{1n}), \cup(V_{21}, \ldots, V_{2m})) \equiv$$

$$\equiv \cup(\bowtie (V_{11}, V_{21}), \bowtie (V_{11}, V_{22}), \ldots, \bowtie (V_{1n}, V_{2m})) \text{ [OV99]}$$

This algebraic equivalence make possible (a) to identify entire joins at a single peer, and (b) to parallelize the execution of the union in several peers (when duplicate elimination is not considered). The latter, as we will detail in the sequel, is achieved by allowing entire subplans to be autonomously processed and executed by different peers. The former is crucial to eliminate redundant plans in a round since such joins have been already considered in a previous round of the interleaved routing and planning. If a peer had advertised both views $V1$ and $V4$ then it had also advertised view $V0$ (since $V0$ is vertically subsumed by $V1$ and $V4$) and thus it would have been contacted at the first round of the interleaved execution. For example, the subplan 1.1 of Figure 5.2 is transformed into the equivalent plan of Figure 5.3. One can easily observe that subplan 1.1 does not take into account the fact that one

peer (e.g., $pr1$) can answer more than one views and thus the results of this join have already been computed in a previous round. The plan of Figure 5.3 identify those joins (depicted with dashed rectangles) and can prune the corresponding subtrees from the plan, thus eliminating any redundancy in the results generated by the advertisement of the vertical subsuming views of a peer.

The proposed DHT-based schema index and the sublookup service makes it possible to employ the above heuristic without any additional planning cost and thus generate in a natural way logical (sub)plans that (a) can be distributed over peers, (b) are redundant free while (c) minimizes the number of routing messages send through the network. Finally, since peers receiving already constructed logical (sub)plans they only undertake the task of transforming them into physical ones by deciding operators' ordering as well appropriate query or data shipping, according to a predefined cost model and statistics that may gather from the involved peers. In [Kok05] such a cost-based optimization is presented along with the classic dynamic programming algorithm [SAC+79] used for enumerating alternative physical plans.

In the following sections, we describe two different execution policies for interleaving the query routing and planning activities. Section 5.2 describes a *statefull* execution of the interleaved routing and planning, which relays on a *coordinator peer* to memorize already computed plans. Section 5.3 describes a *stateless* execution policy in which there is no coordination between peers and thus does not introduce a single point of failure, however it comes with the cost of some redundant invocations of the lookup service.

5.2 Statefull Query Routing and Planning

In this section, we describe a *statefull* execution of the interleaved query routing and planning. In this policy, through a *coordinator peer*, peers are assigned a specific query subplan and issue sublookup requests to obtain the relevant localization information. In addition, they report back to the *coordinator peer* the localization information obtained for this subplan. Such a scenario favours the implementation of advanced query processing techniques such as caching, as well as, the ability to choose peers with low workload for undertaking planning tasks. However, it introduces a single point of failure. In case the *coordinator peer* fails or leaves the network unexpectedly, a *stateless* execution or a choice of another *coordinator peer* may take place.

Figure 5.4 illustrates the *statefull* routing and planning of the query Q depicted in Figure 5.1. When a user formulates a query to the network, a *coordinator peer* is chosen. This can be either the peer where the query was originally issued or any other peer with low workload. During the first iteration, the coordinator peer issues a sublookup request to retrieve all peers that had advertised a view that is horizontally subsumed by query Q. The data obtained from the bases of those peers are returned to the user. Next, if the results returned so far are not sufficient w.r.t. to the user's need, the next round of the interleaved execution is performed. For each of the 3 possible fragmentations of round 1 (Figure 5.1), the initial peers of the larger fragments, called the *dominant* fragments, are located through a lookup request issued by the *coordinator peer*[1]. In this example, peer *pr4* is the initial peer of the *dominant* fragment (view $V4$) of the fragmentation 1.1 (Figure 5.1). Respectively, *pr3* is the initial peer for view $V5$ (the dominant fragment of

[1]If there are more than one fragments of the same size, one is picked randomly to be the *dominant* fragment.

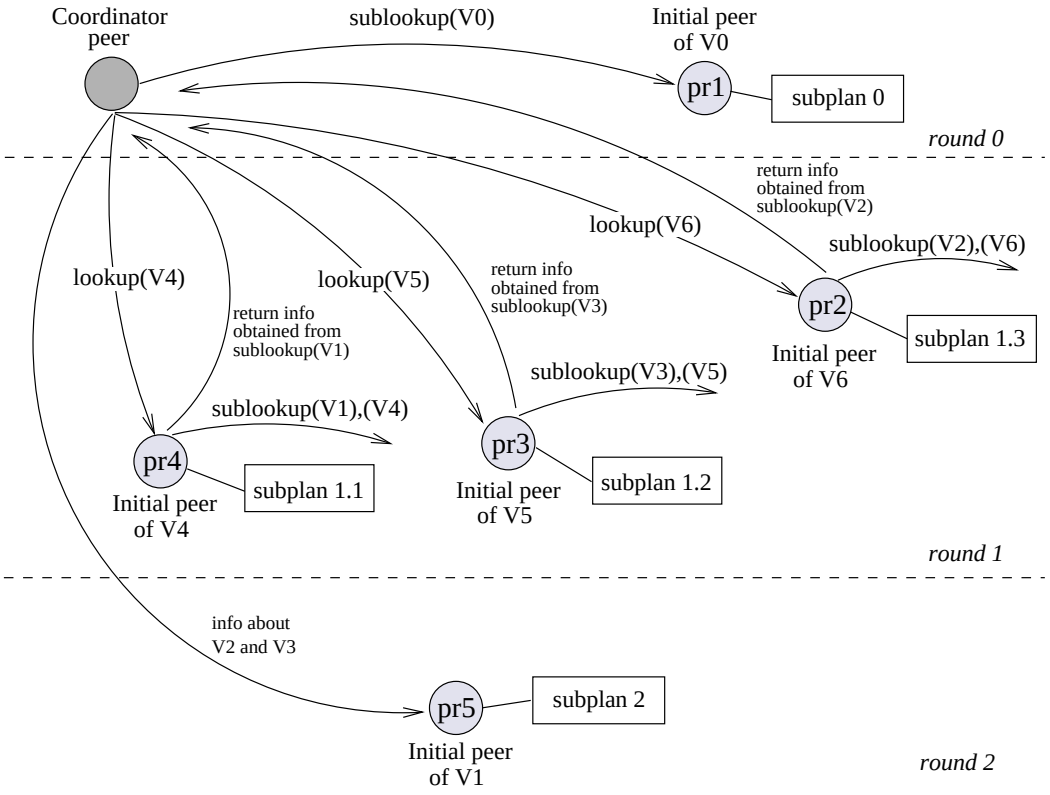


Figure 5.4: Statefull execution of the interleaved routing and planning

fragmentation 1.2) and *pr2* for $V6$ (the dominant fragment of fragmentation 1.3). The *coordinator peer* assigns to each of the initial peers the responsibility to route and plan the corresponding fragmentation. For example, peer *pr4* is responsible to locate all peers that have advertised views that are horizontally subsumed by $V1$ or $V4$, and to deploy the plan for $V1 \bowtie V4$ (subplan 1.1 of Figure 5.2). Respectively, peer *pr3* issues sublookup requests for views $V5$ and $V3$ and peer *pr2* for views $V6$ and $V2$.

At this point, the initial peers may inform the *coordinator peer* about the localization information obtained by the sublookup requests of the fragments having a smaller size than the *dominant* one. The *coordinator peer* caches

then this information since in the next iteration rounds of the interleaved routing and planning this information will be needed again. The localization information obtained by a sublookup requests could be either the peer IPs that actually populate a specific fragment or the actual data that populate those bases[2]. The capabilities of both the P2P network and the *coordinator peer* are the factors that define what is actually cached. Clearly, caching the results obtained by a sublookup request minimize the number of lookup requests but increases the memory demands on behalf of the *coordinator peer*.

When all localization information of the sublookup requests are gathered at the initial peers, the planning algorithm begins. The plans that are created are the unions of all joins that can be done between different peers as discussed previously. Moreover, during the creation of the plan at each initial peer, the cost of alternative execution policies (query shipping, data shipping) and join ordering (recall that the round determines the number of joins) is computed and the less costly is chosen.

Figure 5.5 outlines the algorithm executed at the *coordinator peer* when it receives a query Q. For each round of the interleaved query routing and planning (line 1) it invokes the *fragmentor* to obtain the fragmentations of query Q for a given number of joins (line 2). For each fragmentation it identifies the *dominating* fragment and after locating the initial peer p it delegates to p the responsibility of planning the specified fragmentation (lines 3-6). Moreover, if there are cached results from a sublookup request issued in a previous round for any of the fragments of the specific fragmentation, the *coordinator peer* sends these results to peer p (line 7). Before the *coordinator peer* continues to the next round of the interleaved execution it waits for

[2]In a data-driven PDMS the lookup service retrieves the data itself rather than the peer bases that store those data.

Statefull routing and planning (Coordinator peer)

coord(Q):
input: a query Q with n schema triples

1: **for** i from 0 until $n-1$
2: $F_i \leftarrow$ *fragmentor(Q, i)*
3: **for** each fragmentation $f \in F_i$
4: $d \leftarrow$ *dominating* fragment of f
5: $p \leftarrow$ *lookup(d)*
6: *plan_fragmentation(f)* at peer p
7: ***send*** at peer p any sublookup results obtained from previous rounds
8: **end for**
9: ***wait*** for any sublookup results from the initial peers of this round
10: **end for**

Figure 5.5: Steps taken by the coordinator peer during the statefull planning

Statefull routing and planning (at peer p)

plan_fragmentation(f):
input: a fragmentation f

1: ***wait*** for any sublookup results send from the coordinator peer
2: **for** all fragments q_i of f not cached by the coordinator peer
3: *sublookup(q_i)*
4: ***send*** results of *sublookup(q_i)* back to the coordinator peer
5: **end for**
6: *cost computation* and *physical plan creation* for f

Figure 5.6: Statefull planning algorithm at peer p

sublookup results returned by the initial peers of this round in order to cache them and re-send them in the next round (line 9).

Figure 5.6 outlines the algorithm executed when a peer p receives the sub(plan) for fragmentation f from the coordinator peer. After gathering any sublookup results cached by the coordinator peer (line 1), it issues new sublookup requests for all unknown fragments of the fragmentation f and send the results back to the coordinator peer (lines 2-5). Finally, it creates a physical plan based on the cost model of [KSC05] in order to decide in which peers and in what order the execution of the plan will take place (line 6).

5.3 Stateless Query Routing and Planning

In this section we describe a *stateless* execution of the interleaved query routing and planning activities. The stateless policy does not require any peer to coordinate the entire execution and thus it does not introduce a single point of failure. As in the statefull policy, multiple subplans of the same query Q travel through the network and when a peer receives a request to process a subplan it issues sublookup requests to locate peers populating the involved fragments. However, the next rounds of the interleaved execution are not computed by a single peer, instead each peer computes the next fragmentation of the query Q by adding one extra join to the fragmentation that already have been assigned to. Finally, it forwards the newly created subplans to the *initial peers* of the *dominant* fragments (as in the case of the statefull policy). Given that the results of sublookup requests in this policy cannot be cached and that no peer knows the state or the progress of the interleaved execution, some peers may produce the same subplans for the next iteration rounds. However, since the new subplans are forward to

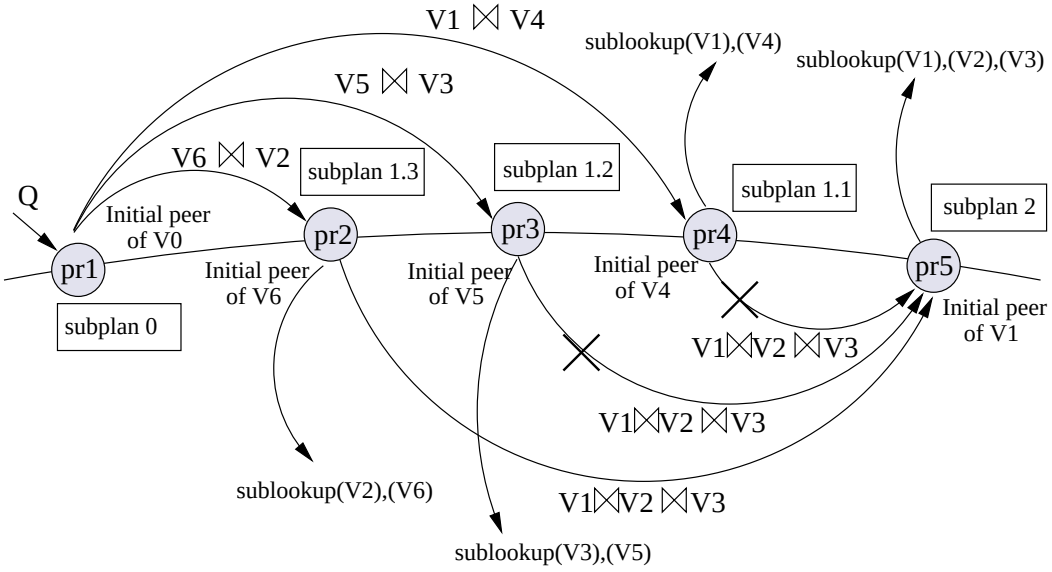


Figure 5.7: Stateless execution of the interleaved routing and planning

the same peer (i.e., the initial peer of the dominant fragment), the initial peer (given that memorize old requests) can identify multiple requests for the same subplan and ignore them.

Figure 5.7 illustrates the *stateless* routing and planning of the query Q depicted in Figure 5.1. Peer $pr1$ receives the query Q since it is the initial peer of view $V0$. It process the subplan 0 of Figure 5.2. Then, $pr1$ computes the next round of the interleaved execution by running the *fragmentor* algorithm with one join as a parameter. It creates the three fragmentations $V1 \bowtie V4$, $V5 \bowtie V3$ and $V6 \bowtie V2$ (Figure 5.1) and forwards the execution of the interleaved query routing and planning to the initial peers of views $V4$, $V5$ and $V6$, namely peers $pr4$, $pr3$ and $pr2$. Peer $pr2$ issues two sublookup requests for the two fragments of the subplan 1.3, namely, views $V2$ and $V6$. Accordingly, peer $pr3$ issues sublookup requests for views $V3$ and $V5$, and peer $pr4$ for views $V1$ and $V4$. Finally, $pr3$ and $pr4$ process the subplans 1.2 and 1.3 respectively.

Stateless routing and planning

plan_fragmentation(Q, f):
input: a query Q and a fragmentation f with i fragments

1: **for** all fragments $r_i \in f$
2: *sublookup(r_i)*
3: **end for**
4: *cost computation* and *physical plan creation* for f
5: $F \leftarrow$*fragmentor(Q, $i+1$)*
6: **for** each fragmentation $f' \in F_i$
7: $d \leftarrow$ *dominant* fragment of f'
5: $p \leftarrow$ *lookup(d)*
6: *plan_fragmentation(Q, f')* at peer p
8: **end for**

Figure 5.8: Algorithm of the stateless execution policy

Each of the peers $pr2$, $pr3$ and $pr4$ independently continue to the next round of the interleaved query routing and planning. They launch the next round by adding one more join to the fragmentation received from peer $pr1$ and thus they consider fragmentations with two joins. Since the query Q has three schema triples the resulting fragmentations are all the same, namely $V1 \bowtie V2 \bowtie V3$. Peer $pr5$ receives from three different peers the same fragmentation to plan. However, it process the subplan 2 of Figure 5.2 only ones while ignoring the rest of the requests. In order to process subplan 2, peer $pr5$ issues sublookup requests for views $V1$, $V2$ and $V3$ although this information was obtained in a previous round of the interleaved execution.

Figure 5.8 outlines the algorithm of the stateless execution policy. When a peer receives a request to plan a given fragmentation f of the query Q, it issues the appropriate sublookup requests to identify the peers that had advertised the corresponding fragments of f (lines 1-3). Next, it process the plan for fragmentation f (line 4). Finally, it invokes the *fragmentor* component to obtain the fragmentations of query Q with one added join and

forwards the execution of the interleaved query routing and planning to the initial peers of the dominant fragment of each fragmentation f' (lines 5-8).

Both execution policies presented in this chapter succeed to distribute the workload of query planning over the peers which are relevant to the original query. The statefull execution policy relies on a *coordinator peer* for caching intermediate localization information obtained by sublookup requests and assigning to each peer a different (sub)plan. Hence, the *coordinator peer* offers resources (mainly memory), while the rest of the peers offer CPU cycles to facilitate query planning. On the other hand, the statefull policy blocks the interleaved execution to cache intermediate results (line 9 of Figure 5.5 and line 1 of Figure 5.6). In contrast, the stateless policy does not block the execution of the interleaved routing and planning nor requires a peer to coordinate the interleaved rounds. However, more lookup and sublookup requests are issued since there is no state information kept anywhere. In addition, the stateless policy requires from all peers to remember the plans created in the near past since they must identify redundant requests.

Finally, the number of peers that are contacted and essentially undertake the responsibility to plan a fragmentation is the same as the number of the *dominant* fragments at each round[3]. These peers are the initial peers in the case of the stateless policy, while in the statefull one the coordinator peer may arbitrary choose between those peers. Since, at each round the dominant fragments are disjoint the initial peers will be distinct, provided that each peer is responsible for only one fragment. For a linear query with n schema triples, the number of different fragmentations created at the first round are $\binom{n}{1}$, while in the second round there are $\binom{n}{2}$ fragmentations, until $\binom{n}{n}$ fragmentations in the last round. Therefore, the total fragmentations

[3]Across rounds the same fragment may be the dominant fragment.

considered during the interleaved execution are 2^n. In the extreme case of a star shaped query with n schema triples, the total number of fragmentations considered in all rounds are $\frac{n!}{2^n}\binom{2n}{n}$ [cXB05]. In Section 6.1 we detail the factors that guarantee that only one fragment is assigned to each peer. In Section 6.2, we prove that we achieve a fair workload balance where each peer undertakes the responsibility of planning only one fragmentation at each round.

Chapter 6

Experimental Evaluation

The goal of the first set of experimental results presented in this chapter is to demonstrate the scalability of the DHT-based schema index with respect to the distribution of the keys for the encoded peer views, as well as to estimate the number of routing hops required to locate peer views that are subsumed by an input query. We conducted our experiments for different sizes of peer networks and views with varying number of schema triples and structural form (linear, tree or graph form). The RDF/S schema that was used in our experiments was created synthetically based on real application examples [MACP02].

In the next set of experiments, we demonstrate the benefits of distributing the planning tasks over peers, as defined in the interleaved query routing and planning. We conducted experiments for estimating the planning time required by the interleaved execution. We considered the evaluation of a query involving the entire RDF/S schema in a predefined network setting. We conclude our experiments by comparing our results with an ideal execution in which the execution time is uniformly distributed to all peers.

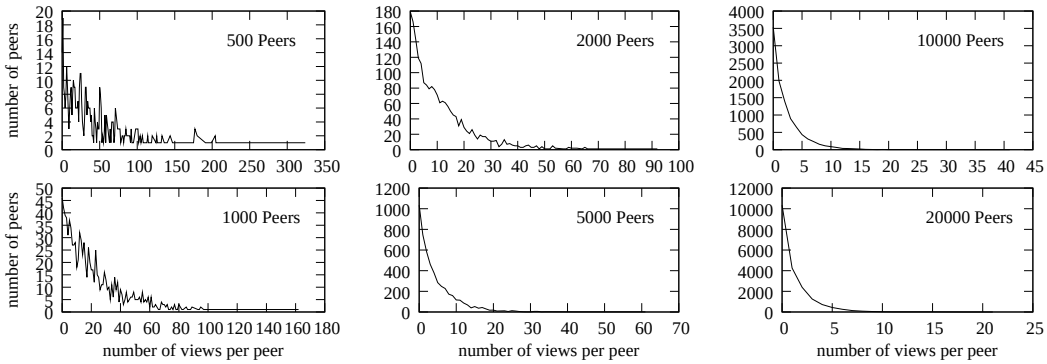


Figure 6.1: Distribution of views over peers in networks of different size.

6.1 DHT-based Schema Index and Lookup Service

Our experiments rely on the original Chord protocol simulator [Cho], modified to support the distributed index of RDF/S fragments, as well as to implement both the two versions of the lookup service presented in Section 4.2. The stabilization algorithm of the Chord protocol and the key reallocation algorithms when peers join, leave or die unexpectedly, were kept intact. Thus, we considered that the system was formed and stabilized before any view was stored or looked up.

An important characteristic of the proposed encoding function (based on the *AdjSub Cube*) is related to the uniform distribution of the views (keys) over the DHT nodes. As a matter of fact, as the network grows the keys are equally distributed in an increasing number of peers and thus the number of views stored in each peer is constantly decreased. For large scale networks (i.e., 5000 to 20000 peers), this behaviour is confirmed by the experiments reported in Figure 6.1 where we stored approximately 2000 views in networks of varying size. Given that the discriminating factor in our experiments is

the ratio of views per peers, we keep constant the number of views stored in each network. Since our query patterns capture only the structure and the semantics of an RDF/S schema (and not arbitrary joins on property values) the views considered here correspond to distinct fragments of a specific RDF/S schema. For networks of small size (i.e., 500, 1000 and 2000 peers) where the ratio of views per peer is above 1, the views exhibit a skewed distribution (i.e. a small number of peers indexing a large number of views). This is due to the fact that the unique identifiers of the views comprising only one schema triple, have small hash values thus they are stored exclusively on the peers placed at the beginning of the Chord ring. To overcome this problem one can force some peers to hash their identifier in the beginning of the Chord ring, and thus make more peers responsible for views with few schema triples. Alternatively, a simpler solution for small networks will be to consider a smaller identifier circle (i.e., by decreasing m of *modulo* 2^m) and thus place peers closer to each other.

The four graphs of Figure 6.2 illustrate the total number of routing hops per number of subsumed views involved in the evaluation of a query pattern, for both the statefull and stateless versions of the lookup algorithm. In addition, in each graph we illustrate both the theoretical ($S \times \log n$) and experimental ($S \times \frac{1}{2} \times \log n$) number of routing hops required by the original Chord [SMK+01] protocol. Clearly, the two versions of the lookup service decrease the number of routing hops up to 50% than those required by Chord. For networks of small size, we can observe that the statefull lookup service outperforms the stateless one since it requires less than half routing hops for large views. This can be easily justified since the stateless algorithm fails to identify the peers that can answer more than one subsumed view at once, and therefore it contacts the same peers over and over. This is not the case

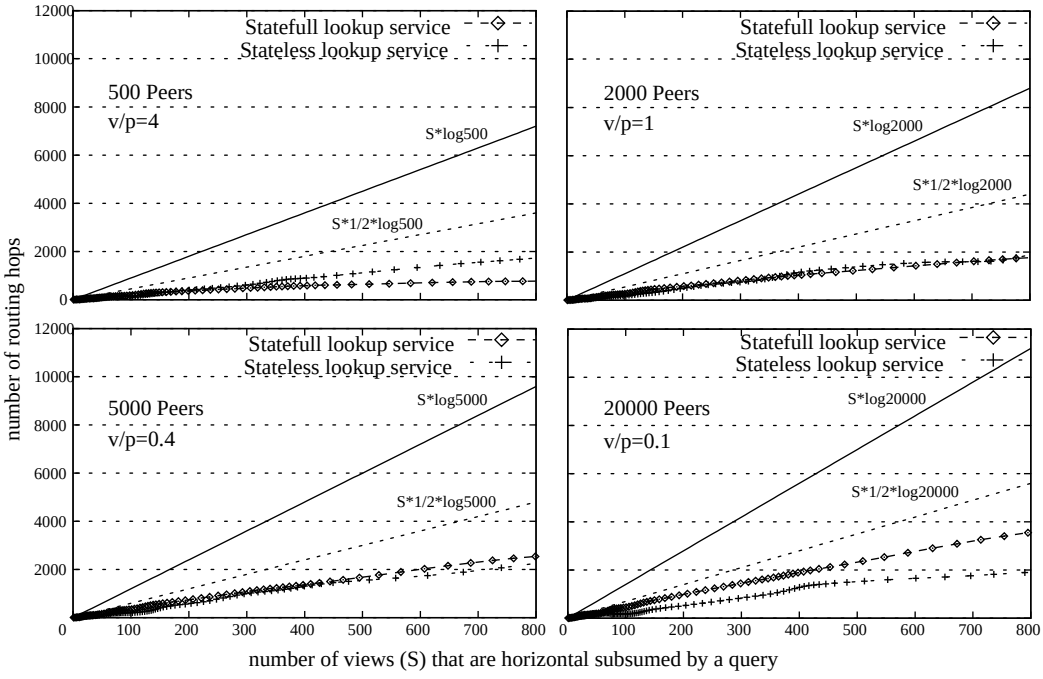


Figure 6.2: Number of routing hops for networks of different size.

of the statefull lookup since the *initial peer* gathers all views from a peer and never contacts the same peer twice. However, when the network grows, the statefull version exhibits poor performance. As a matter of fact, as the network grows it becomes more unlikely to find peers storing more than one view, hence the advantages of the statefull approach fade out. On the other hand, the number of hops required by the stateless version only slightly increases when the size of the network doubles and clearly outperforms the statefull one when the ratio views per peer falls bellow 1.

The left graph of Figure 6.3 illustrates the number of routing hops required by the stateless version of the lookup algorithm, while the right graph by the statefull one. In particular, in networks of different size, each bar indicates the average number of routing hops required to locate views according to a set of 200 queries. In the boxes next to each bar we give the

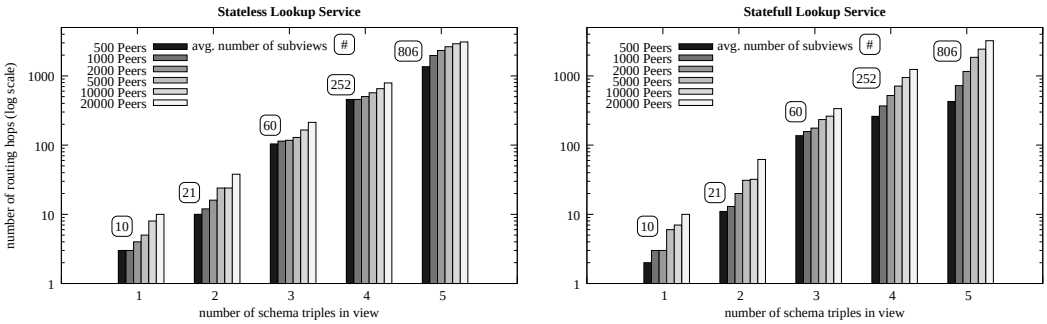


Figure 6.3: Number of routing hops for queries of different size

average number of the views that are horizontally subsumed by these queries. In both cases, as the number of schema triples in the strict view increases, the average number of routing hops increases too. This is due to the fact that there exist more subsumed views, since more RDF/S schema classes and properties are glued together. Another reason is that for large views, the distance between their unique identifiers is greater than the distance between small views, thus more routing hops are required to locate the next view subsumed by the query.

The main conclusions drawn from the above set of experiments are: (a) as the network grows, the DHT-based schema index succeeds to distribute the encoded views in a uniform manner and (b) the proposed lookup algorithms outperform the lookup service of the original Chord. Moreover, the stateless version of the lookup algorithm is more suitable for large networks as it scales gracefully, compared to the statefull one that is more beneficial for small networks. One can then decide, depending of the size of the network which lookup version to use.

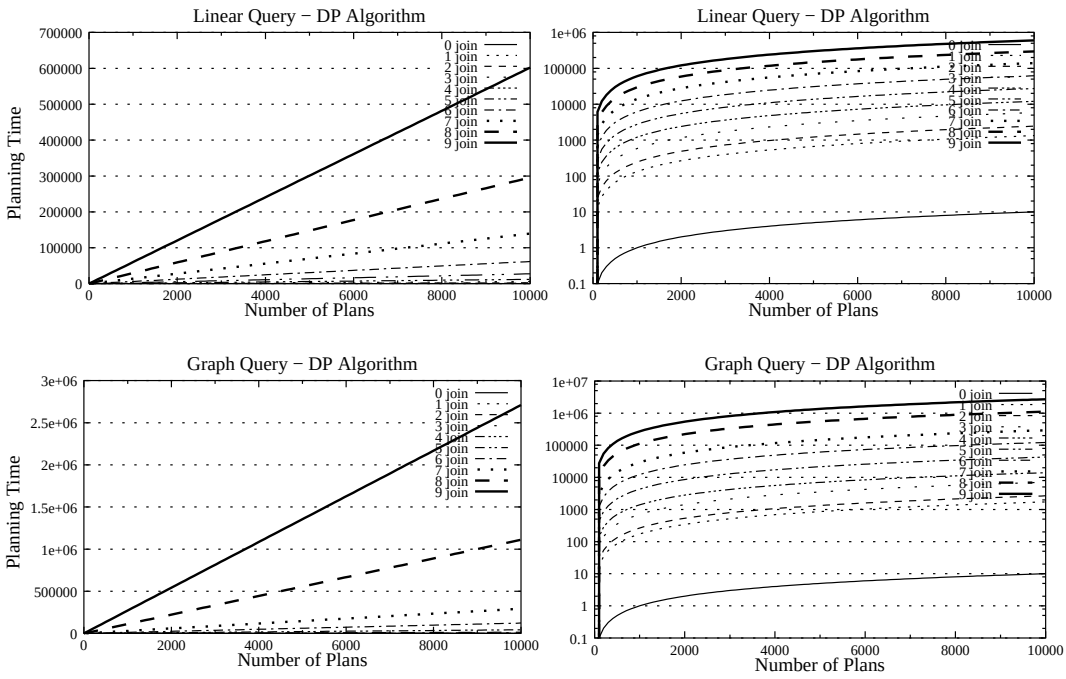


Figure 6.4: Dynamic Programming

6.2 Interleaved Query Routing and Planning

A set of experiments were conducted considering a linear query of 10 schema triples. Apart from their variations in planning time, graph queries or queries of different size demonstrate the same degree of distribution. This is due to the fact that both the lookup service and the interleaved routing and planning are independent of the form and size of the query (w.r.t. the degree of distribution).

First, we employ the dynamic programming algorithm to compute the planning time required to optimize a particular query plan. We have considered query plans that involve up to 9 joins. From the graphs presented in Figure 6.4, we can easily observe that the planning time is linear to the number of the considered query plans. Additionally, the planning time increases exponentially to the number of joins involved in each plan. The

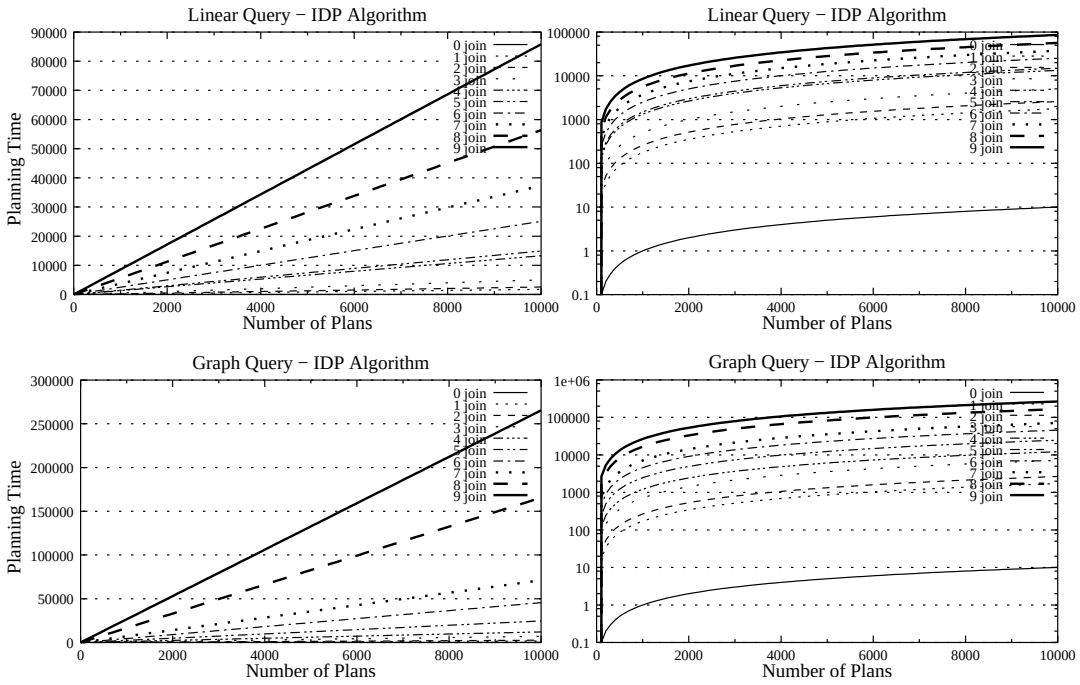


Figure 6.5: Iterative Dynamic Programming

above two observations stress the need for a planing algorithm which avoids as much as possible the concurrent optimization of a large amount of plans and that should consider the fact that when more joins are involved, more optimization time is required.

Next, in order to avoid the exponential time of the dynamic programming algorithm, we conducted experiments with the use of an iterative dynamic programming approach [KS00]. For our experiments we considered the input value k of the algorithm to be equal to 4 (i.e., at each round 4-way join plans are generated). Although it is shown that the planning time is greatly reduced, still for a great number of plans this time becomes prohibitive. More precisely, for a large number of plans that consider a great number of joins the planning time is significant for a centralized planning approach.

In order to illustrate the performance gains of the interleaved query rout-

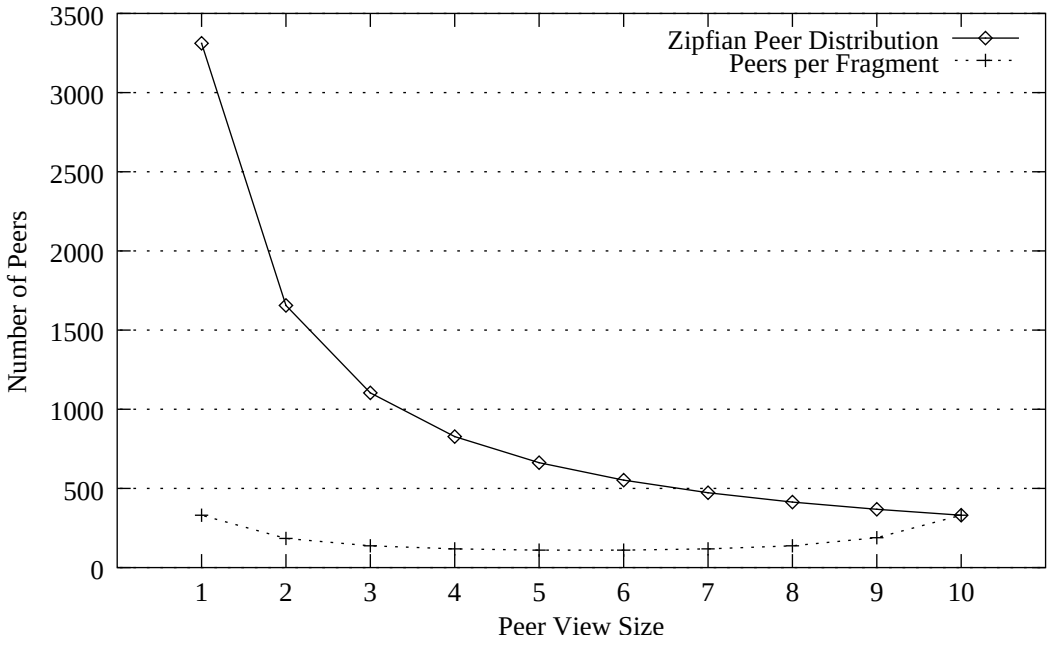


Figure 6.6: Distribution of peer vies over the network.

ing and planning, in the upper curve of Figure 6.6 we depict a zipf distribution of available peer views in a network of 10000 nodes. We additionally consider an RDF/S schema, whose size is 10 (i.e., contains 10 properties linked together in a linear way). This distribution mode dictates that more peers are capable of answering smaller fragments in contrast to a small number of peers that can answer larger ones. For example, approximately 3300 peers provide data for fragments of size 1, while only 330 provide data for the whole RDF/S schema. The lower curve in Figure 6.6 depicts the number of peers capable of answering each fragment of the RDF/S schema. The distribution we considered for fragments of the same size is a uniform one.

We conducted experiments for estimating the planning time required by the interleaved execution. We considered the evaluation of a query involving the entire RDF/S schema in a network setting as previously described. In Figure 6.7 the planning time needed for each round of the interleaved exe-

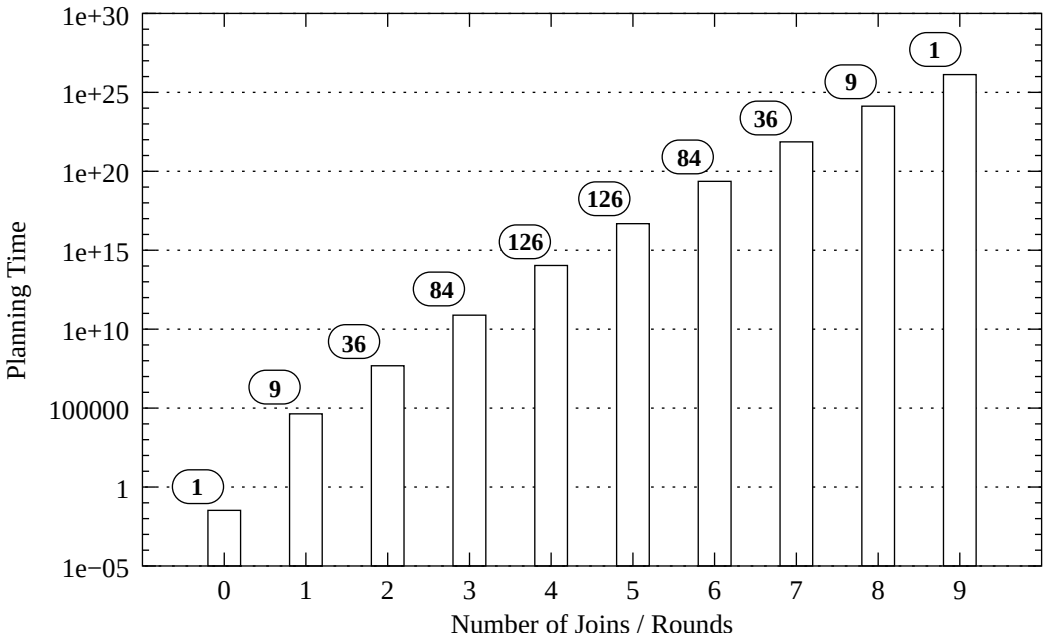


Figure 6.7: Planning Time per Round

cution is shown. At each round the joins considered in the query plan are increased up to a maximum number, which is required to obtain data from the entire schema. As the join number increases, so is the number of plans considered, since both the possible fragmentations and the combination of answering peers increase. For each round, we also show the number of different fragmentations that are possible for each given number of joins (in small boxes on top of each bar). We should point out that each round is independent from the other rounds since all query plans that involve intra-peer processing are ignored by appropriate pruning (Section 5.1) and each round is independently executed by different peers, as described in Chapter 5. Moreover, to obtain a complete answer we have to wait as much as the planning time required by the last round of the interleaved execution. If we ignore communication delays of the routing, all rounds start simultaneous but will finish before the last round does. It should be stressed that

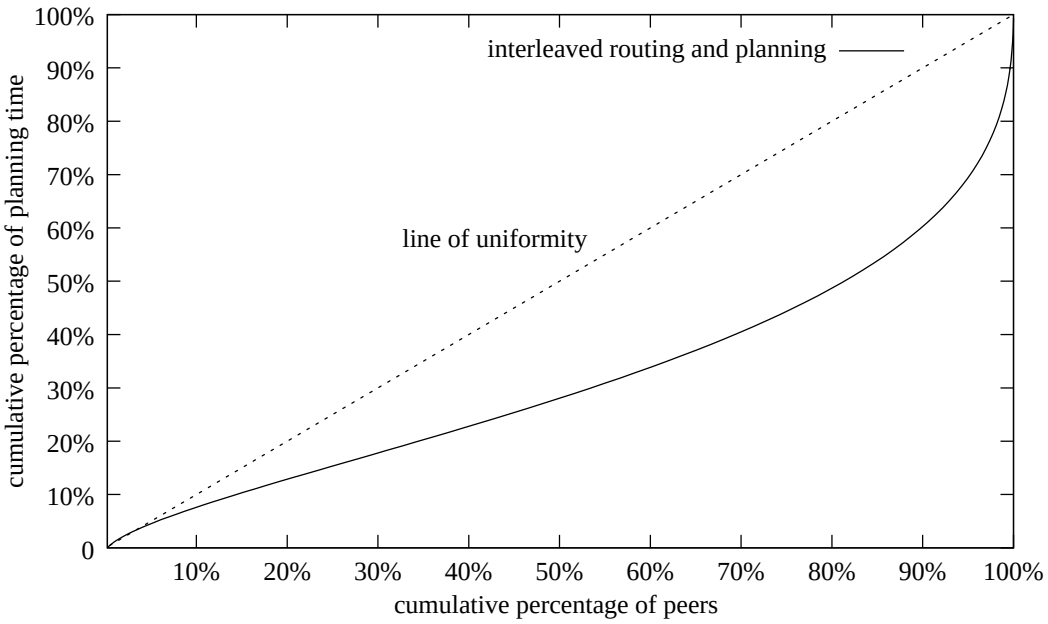


Figure 6.8: Distribution of the total workload over peers

this is not possible in a sequential scenario were each round starts when the previous has finished.

According to the planning times depicted in Figure 6.7, we compute the distribution of the overall planning activity time over the total number of peers involved in the evaluation of the query. The line of uniformity in Figure 6.8 depicts the case in which the total planning time is evenly distributed among all peers. The curve of Figure 6.8 depicts the distribution of the planning time achieved by the interleaved query routing and planning. We can observe that the 10% of the total time is distributed over the 10% of the total peers. This is actually the planning time of the first two rounds. This observation proves our claim that the first rounds of the interleaved execution will finish fast, while the data returned will be from peer bases that populate the entire query. As the number of joins increase, the planning time increases to and so does the peers that are involved in the query processing. However,

this is true only until the round that 4 joins are considered. After that, although the planning time is increased, the fragmentations are decreased and so does the peers that are involved (Figure 6.7). Due to this effect, the distance between the line of uniformity and the curve of the interleaved routing and planning of Figure 6.8 increases beyond the 50% of peers. To illustrate the most extreme case, consider the last round (9 joins), where the planning time is the highest of all other rounds, but only 1 peer undertakes the responsibility of this fragmentation.

The main observation of this set of experiments is that the interleaved query routing and planning achieves a fair load distribution that is closer to the line of uniformity at the first rounds. The first answers are obtained fast, while the complete evaluation of the query is more time consuming and overloads some peers.

Chapter 7

Related Work

Closely related to our work are DHT-based PDMS addressing routing and planning issues for queries over RDF, XML and relational databases.

RDFPeers [CF04] is a distributed RDF repository based on an extension of Chord, namely MAAN, that stores each triple at three places in the network by applying a globally known hash function to its subject, predicate and object (i.e., *data triple*). A DHT index is built over RDF triples which ignores the semantics of the RDF/S schema during query routing. Such an extensional index extremely increases the total amount of data stored on the network and comes with a significant message overhead when triples are likely to frequently change. Given that at least one from the subject, predicate or object value is known, RDFPeer can locate the data triples with the known value, using the Chord lookup service, in $O(\log N)$ hops. Multi-predicate and range queries over arithmetic properties can be evaluated in $O(\sum_{i=1}^{k}(\log N + N \times s_i))$ routing hops by decomposing the query to the k triples of which is consist of, and visit each time s_i peers, where s_i is the selectivity of the triple on predicate p_i. Finally, to overcome the increased workload of peers storing triples with subject popular URIs, RDFPeers con-

sider a threshold after which it refuses to store any triples. In contrast, our framework favours the scalability, thus there will be no increased workload in a single peer and no data will be lost.

In [HHK05], the authors present a query evaluation algorithm which allows to express RDF/S queries in a structured P2P network by taking into account subsumption relationship between classes and properties defined in an RDF/S schema. Although the authors argue that a global knowledge of an RDF/S schema is needed in order to reason about queries regardless the heterogeneity of the underlying peers, they do not employ schema information to index data or route queries. Instead, they use a similar to RDFPeers idea, where each data triple is stored in three places according to the hash value of the subject, predicate and object. In addition, in order to facilitate horizontal subsumption each data triple is stored in several peers (i.e., peers that are responsible for horizontal subsuming schema triples). The authors do not consider vertical subsumption, thus to evaluate an RDF/S graph query they have to gather all involved data triples to a single peer in order to process locally the query. This policy significantly increases the data amount stored in the network and makes data updates more costly while peers' advertisements are much more costly in terms of exchanged messages. In order to evaluate a query, each triple of the query graph is independently looked up and candidate sets (i.e., set of data that may contribute to the final evaluation of the query) are determined. A single peer gathers locally all candidate sets and uses refinement procedures to remove data from this sets that are not suitable. This refinement procedures are based on the bindings between the triples that contain the variables and the data triples from the candidate sets. Only data triples that match with the triples where the variable occurs are kept. For the final evaluation of the query, all remaining

data triples are tested locally in several combinations to reveal the matches. Since there are exponentially many combinations, every time a candidate for a variable is picked, the refinement procedure is again employed in order to reduces the number of possible combinations. The proposed query evaluation algorithm is similar to the last round of the interleaved query routing and planning. In contrast to our system where data do not travel through the network until the physical plan is determined, they first retrieve all data in a single peer (including data that may not contribute to the final evaluation) and afterwards employ techniques to optimize the query planning and execution.

In [GWJD03], a distributed catalog for XML data is proposed along with appropriate load balancing techniques to fairly distribute the catalog service and adapt the system's behaviour to the query workload. In the DHT-based catalog keys are XML fragments associated with a set of structural summaries (i.e., the XPaths leading to these fragments). A B+-tree is used by each peer to match a given XPath query against the stored summaries. The leaf nodes of the B+-tree point to a set of peers that can answer the matched XPath query. Given a simple XPath (linear path), in the case of Chord, we need $O(\log N)$ routing hops to find the peer responsible for the leaf XPath node and a search over the B+-tree to locate the peers that can actually answer the full XPath. In the case of an XPath query of the form $p = /a_1[b_1]/\ldots/a_n[b_n] op\ value$ where each b_i is in turn a path, the system must first extract all k linear paths and invoke the lookup service for each of them. In contrast to this system we do not distinguish between linear, tree or graph queries and thus in either cases the lookup service is invoked only once and $O(\log N)$ hops are required (without considering subsumed view) to locate peers that can answer the query. Also, there is no need for

a search over a secondary index structure since our index is build directly on the RDF/S schema fragments. For load balancing, the authors propose techniques of splitting and replicating the catalog, while in our framework these are done a priory. Our framework can be easily adapted to build a DHT for XML instead of RDF/S schema fragments and unlike this system, it ensures that the same number of routing hops are required for both linear and tree shaped queries.

A unifying framework for relational query processing over structured P2P networks has been proposed in [TP03]. Each tuple of a relation $R(DA_1, \ldots, DA_k)$ is stored $k+1$ times over the network: one copy of the tuple with consistent hashing over it's primary key, and k replicas distributed in the peers according to an order-preserving hash function based on its k attributes. The authors also introduce the notion of *Range Guards*, i.e., a number of peers which keep additional replicas of all tuples whose values for a specific attribute fall in a specific range. They are used to evaluate range queries over relational data by avoiding costly traversals of the entire Chord ring. The two major drawbacks of this system are a) the need to replicate data several times and thus increasing the maintaining cost, and b) multi-relation/attribute or range queries are costly to route (in some cases needing $O(N)$ hops). For example, to evaluate multi-relation and multi-attribute queries with joins they always need to scan the entire network since there is no way to know in advance which peers have those tuple values that actually join. In our framework, range queries over schema triples are efficiently evaluated since the proposed RDF/S fragment encoding ensures that they will be indexed closed to each other on the Chord ring, without the need of replication and range guards.

PIER [HHL+03] is a massively distributed query engine based on overlay networks. It is built on top of DHTs and runs relational queries. Each tu-

ple indexed in the DHT has a namespace, resourceID and instanceID. The namespace identifies the application or group a tuple belongs to. The resourceID is generally intended to be a value that caries some semantic about the tuple (e.g., the name of the relation that the tuple belongs to). The namespace and the resourceID are used to calculate the DHT key, via a hash function. The PIER Query processor is a dataflow engine supporting the simultaneous execution of multiple operators that can be pipelined together to form traditional query plans. The authors detail four join strategies that are adaptations of the join algorithms designed for parallel and distributes schemes, which leverage DHTs whenever possible. In general, the DHT is used to re-hash (re-index) data according to the value of the joining attributes. If there is more than one join that must be performed in order to evaluate a query, intermediate results are stored in a newly created DHT, identified by a new namespace. In this way, the functionality of a "distributed hash table" is used to implement hash-joins in a distributed environment. The authors argue for a relaxation of certain traditional database research goals in the pursuit of scalability and widespread adoption. More precisely, they argue that in order to cope with the dynamic large scale environment imposed by the Internet we should provide "best-effort" results rather than trying to obtain the complete answers. Moreover, they argue that data should remain in their natural habitats (e.g., a file system or a database of the peer) rather than flowing around the network. Finally, they argue that requiring from thousands of users to design and integrate their disparate schemas incur daunting semantic problems and could easily prevent average users from adopting these technologies. Instead, it is preferable to adopt standard schemas and that there is a natural pathway for this: the information produced by popular software (e.g., ID3 tags). We share the

same ideas with the authors of PIER and our system is designed partially based on these observations.

Chapter 8

Conclusion and Future Work

In this thesis, we presented a DHT-based framework to efficiently route and process plans for expressive RDF/S queries.

We introduced a succinct representation of RDF/S schema graphs, called *AdjSub Cube*, for encoding arbitrary RDF/S schema fragments. This encoding ensures a fast view/query subsumption checking in order to understand the partitioning of data in remote peer bases. Based on this encoding, we designed a DHT-based schema index to uniformly distribute view advertisements over peers. Additionally, we implemented a lookup service for identifying which peers can completely or partially contribute to the answer of a graph query. Finally, we designed an interleaved query routing and planning allowing to obtain as fast as possible the first results of a query available in peer bases while distributing the planning load of the next rounds to arbitrary peers. We experimentally demonstrated that the proposed DHT-based schema index scales gracefully for very large number of peers. Moreover, we compared the routing hops required by a stateless and a statefull version of our lookup service versus the routing hops required by the original Chord protocol. Finally, we experimentally illustrate the degree of distribution of

the planning workload achieved by the proposed interleaved query routing and planning execution. The main conclusion drawn from our experiments is that our lookup service requires less than half of the routing hops required by the original Chord protocol. Moreover, the interleaved query routing and planning distributes the first 10% of the planning workload to the 10% of the total peers contributing to the evaluation of a query. This observation proves that the first answers will be returned as fast as possible. For the remaining results, more time will be required since the planning activity is increased and the peers that undertake planning tasks are decreased.

The results presented in this paper can be easily adjusted to other DHT-based protocols and schema formalisms defining SONs. For example, we can build an *AdjSub Cube* for encoding fragments of an XML schema tree.

We intend to investigate the potential of a P2P infrastructure based on distributed trees [CLGS04, Abe01, JOV05], for implementing the *AdjSub Cube*, in order to further reduce the number of hops required by our lookup service and distribute even fairly the planning workload. In addition, we intend to extend our system with (a) a pruning strategy based on the quality of information returned by a peer base and (d) adaptive planning algorithms that consider the workload of peers and the network capabilities each time a query is processed [AH00, UF93].

Bibliography

[Abe01] Karl Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, Trento, Italy, 2001.

[ABJ89] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Oregon, USA, 1989.

[ACMHP04] Karl Aberer, Philippe Cudre-Mauroux, Manfred Hauswirth, and Tim Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *Proceedings of the 3rd International Semantic Web Conference (ISWC04)*, Hiroshima, Japan, 2004.

[AH00] R. Avnur and J.M. Hellerstein. Eddies:Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, USA, 2000.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in Oracle Rdb. *The VLDB Journal — The International Journal on Very Large Data Bases*, 5(4), 1996.

[BGK+02] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, Madison, Wisconsin, 2002.

[BT03] Peter Boncz and Caspar Treijtel. AmbientDB: Relational Query Processing in a P2P Network. In *Databases, Information Systems, and Peer-to-Peer Computing: First International Workshop, DBISP2P*, Berlin, Germany, 2003.

[CF04] Min Cai and Martin Frank. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, New York, USA, 2004.

[CGM03] Arturo Crespo and Hector Garcia-Molina. Semantic Overlay Networks for P2P Systems. Technical report, Computer Science Department, Stanford University, 2003.

[Cho] The Chord Project. http://pdos.csail.mit.edu/chord/.

[CKK+03] Vassilis Christophides, Gregory Karvounarakis, Ioanna Koffina, George Kokkinidis, Aimilia Magkanaraki, Dimitris Plexousakis, George Serfiotis, and Val Tannen. The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware. In

Proceedings of the First International Workshop on Semantic Web and Databases (SWDB), Berlin, Germany, 2003.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of the 7th International Workshop on the Web and Databases*, Paris, France, 2004.

[CPST03] Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotiris Tourtounis. On Labeling Schemes for the Semantic Web. In *Proceedings of the 12th International Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003.

[cXB05] Cong cong Xing and Bill P. Buckles. On the size of the search space of join optimization. *Journal of Computing Sciences in College*, 20(6), 2005.

[DH02] Amol Deshpande and Joseph M. Hellerstein. Decoupled Query Optimization for Federated Database Systems. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, USA, 2002.

[ETB+03] Marc Ehrig, Christoph Tempich, Jeen Broekstra, Frank van Harmelen, Marta Sabou, Ronny Siebes, Steffen Staab, and Heiner Stuckenschmidt. SWAP - Ontology-based Knowledge Management with Peer-to-Peer Technology. In *Proceedings of the 1st National "Workshop Ontologie-basiertes Wissensmanagemen" (WOW)*, Lucerne, Switzerland, 2003.

[GWJD03] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Locating Data Sources in Large Distributed Systems.

In *Proceedings of the 29th Very Large Data Bases (VLDB) Conference*, Berlin, Germany, 2003.

[HHK05] Felix Heine, Matthias Hovestadt, and Odej Kao. Processing Complex RDF Queries over P2P Networks. In *Proceedings of the 2005 ACM Workshop on Information Retrieval in Peer-to-Peer Networks (P2PIR 2005)*, Bremen, Germany, 2005.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, Berlin, Germany, 2003.

[HIMT03] Alon Halevy, Zachary Ives, Peter Mork, and Igor Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In *Proceedings of the 12th Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003.

[JOV05] H.V. Jagadish, B. C. Ooi, and Q.H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, Trondheim, Norway, 2005.

[KAC+02] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th International Conference on World Wide Web (WWW)*, Honolulu, Hawaii, USA, 2002.

[Kok05] George Kokkinidis. Semantic Query Routing and Planning in Peer-to-Peer Database Systems: The SQPeer Middleware. Master's thesis, University of Crete, Department of Computer Science, Crete, Greece, July 2005.

[KS00] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)*, 25(1), 2000.

[KSC05] George Kokkinidis, Lefteris Sidirourgos, and Vassilis Christophides. *Semantic Web and Peer-to-Peer, S. Staab, H. Stuckenschmidt (eds.)*, chapter Query Processing in RDF/S-based P2P Database Systems. Springer-Verlag, 2005.

[KSDC05] George Kokkinidis, Lefteris Sidirourgos, Theodore Dalamagas, and Vassilis Christophides. Semantic Query Routing and Processing in P2P Digital Libraries. In *Proceedings of the 8th International Workshop of the DELOS Network of Excellence on Digital Libraries on Future Digital Library Management Systems (System Architecture & Information Access)*, Schloss Dagstuhl, Germany, 2005.

[MACP02] Aimilia Magkanaraki, Sofia Alexaki, Vassilis Christophides, and Dimitris Plexousakis. Benchmarking RDF Schemas for the Semantic Web. In *Proceedings of the First International Semantic Web Conference (ISWC)*, Sardinia, Italy, 2002.

[MTCP03] Aimilia Magkanaraki, Val Tannen, Vassilis Christophides, and Dimitris Plexousakis. Viewing the Semantic Web Through

RVL Lenses. In *Proceedings of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[NWS+03] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario Schlosser, Ingo Brunkhorst, and Alexander Loser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In *Proceedings of the 12th International Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003.

[OV99] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition.* Prentice Hall, 1999.

[RDF] Resource Description Framework (RDF). http://www.w3.org/RDF/.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM International Conference on Data Communications*, San Diego, CA, USA, 2001.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, Massachusetts, USA, 1979.

[SKCT05] George Serfiotis, Ioanna Koffina, Vassilis Christophides, and Val Tannen. Containment and Minimization of RDF/S Query

Patterns. In *Proceedings of the 4th International Semantic Web Conference (ISWC)*, Galway, Ireland, 2005.

[SKD05] Lefteris Sidirourgos, George Kokkinidis, and Theodore Dalamagas. Efficient Query Routing in RDF/S schema-based P2P Systems. In *Fourth Hellenic Data Management Symposium (HDMS'05)*, Athens, Greece, 2005.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM International Conference on Data Communications*, San Deigo, CA, USA, 2001.

[SRvdWB05] Kai-Uwe Sattler, Philipp Roumlsch, Christian von der Weth, and Erik Buchmann. Best Effort Query Processing in DHT-based P2P Systems. In *Proceedings of the 1st IEEE International Workshop on Networking Meets Databases (NetDB)*, Tokyo, Japan, 2005.

[TP03] 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, 2003.

[UF93] Tolga Urhan and Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Special issue on Adaptive Query Processing*, 23(2), 1993.