University of Crete
Computer Science Department

Semantic Query Routing and Planning in Peer-to-Peer Database Systems: The SQPeer Middleware

George Kokkinidis

Master's Thesis

Heraklion, July 2005

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Δρομολόγηση και Επεξεργασία Σημασιολογικών Επερωτήσεων σε Δυομότιμα Συστήματα

Εργασία που υποβλήθηκε από τον

Γεώργιο Φ. Κοκκινίδη

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Γεώργιος Κοκκινίδης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Αναπληρωτής καθηγητής, Επόπτης

Δημήτρης Πλεξουσάκης, Αναπληρωτής καθηγητής, Μέλος

Γρηγόρης Αντωνίου, Καθηγητής, Μέλος

Δεκτή:

Δημήτρης Πλεξουσάκης, Αναπληρωτής καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Ιούλιος 2005

Semantic Query Routing and Planning in Peer-to-Peer Database Systems: The SQPeer Middleware

George Kokkinidis

Master's Thesis

Computer Science Department, University of Crete

Abstract

Peer-to-peer (P2P) computing is currently attracting enormous attention. In P2P systems a very large number of autonomous computing nodes (the peers) pool together their resources and rely on each other for data and services. More and more P2P data management systems rely nowadays on intensional (i.e., schema) information for integrating and querying peer bases. Such information can be easily captured by emerging Semantic Web languages such as RDF/S. However, a full-fledged framework for evaluating semantic queries over peer RDF/S bases (materialized or virtual) is missing.

In this thesis, we present the SQPeer middleware for processing RQL queries over peers, whose bases are advertised by RVL views. SQPeer utilizes the notion of RDF/S-based Semantic Overlay Networks (SONs) for organizing the peers into communities, where semantic queries can be efficiently processed and executed. The notion of RDF/S-based SON is examined with respect to three different architectural alternatives, i.e., a hybrid, a structured and an ad-hoc one, with each one posing different challenges. The novelty of SQPeer lies on the use of intensional peer views used for advertising the content of peer bases. A routing phase is responsible for identifying peer advertisements relevant to a specific query based on appropriate query/view subsumption techniques. On the other hand, a query planning phase

uses the obtained data localization information to construct appropriate distributed query plans considering data distribution in the system for obtaining both complete and correct results. The produced query plan is executed in a fully distributed way by contacting the necessary peers and sending them the appropriate (sub-)queries. Peer communication for exchanging query (sub-)plans and results relies on appropriate communication channels.

Compile and run-time optimization strategies are employed in order to create optimal query plans. Heuristics concerning the ordering of joins and unions are used to transform the query plan into a more efficient equivalent one. Both communication and processing cost is used to decide between data, query or hybrid shipping execution policies. Additionally, adaptability in the formulated query plans is possible by monitoring the query execution and altering the whole or part of the query plan at run-time, when peer bases become unavailable or system resources are exhausted.

Finally, SQPeer's query processing algorithms allow an interleaved execution of the query routing and planning phases. Especially in a structured P2P setting, this interleaved execution permits the creation of multiple query plans in several iteration steps that when combined produce a complete answer. More importantly, intra-peer processing is favored with the additional benefit on obtaining more relevant results as soon as possible.

Supervisor: Vassilis Christophides
Associate Professor

Δρομολόγηση και Επεξεργασία Σημασιολογικών Επερωτήσεων σε Δυομότιμα Συστήματα

Γεώργιος Κοκκινίδης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Τα Δυομότιμα (Peer-to-Peer ή πιο απλά P2P) Συστήματα έχουν γίνει ιδιαίτερα δημοφιλή τον τελευταίο καιρό. Στα συστήματα αυτά ένας μεγάλος αριθμός από αυτόνομους κόμβους διαθέτει τους πόρους του και επικοινωνούν μεταξύ τους για την ανταλλαγή δεδομένων και υπηρεσιών. Όλο και πιο πολλά Δυομότιμα Συστήματα Διαχείρησης Δεδομένων βασίζονται στην σημασιολογία των δεδομένων τους για την ολοκλήρωση και επερώτηση των βάσεων δεδομένων των κόμβων που τα απαρτίζουν. Τέτοιου είδους πληροφορία μπορεί εύκολα να αναπαρασταθεί με την βοήθεια γλωσσών του Σημασιολογικού Ιστού, όπως η RDF/S . Παρ'ολα αυτά, μέχρι σήμερα δεν υπάρχει ένα πλήρες πλαίσιο εργασίας για αποτίμηση σημασιολογικών ερωτήσεων πάνω από RDF/S βάσεις δεδομένων (υλοποιημένες ή ιδεατές).

Στην παρούσα αναφορά, παρουσιάζουμε το SQPeer σύστημα για επεξεργασία RQL επερωτήσεων πάνω από κόμβους, των οποίων οι βάσεις δημοσιοποιούνται μέσω RVL όψεων. Το SQPeer ενσωματώνει την έννοια των Σημασιολογικών Διαστρωματωμένων Δικτύων (SON) για την οργάνωση των κόμβων σε κοινότητες, όπου σημασιολογικές επερωτήσεις μπορούν αποτελεσματικά να επεξεργαστούν και να εκτελεστούν. Η έννοια των SON βασισμένων στην RDF/S εξετάζεται συναρτήσει τριών διαφορετικών αρχιτεκτονικών, μιας υβριδικής, μιας δομημένης και μιας αδόμητης, με κάθε μία να θέτει τις δικές της προκλήσεις. Σε ένα τέτοιο πλαίσιο, η φάση δρομολόγησης είναι υπεύθυνη για την αναγνώριση όψεων σχετικών με μια δεδομένη επερώτηση και βασίζεται σε

τεχνικές υπαλληλίας επερωτήσεων/όψεων. Από την άλλη πλευρά, η φάση δημιουργίας πλάνων εκτέλεσης χρησιμοποιεί την ήδη υπάρχουσα πληροφορία δρομολόγησης για να κατασκευάσει τα απαραίτητα πλάνα εκτέλεσης έχοντας υπόψιν την κατανομή της πληροφορίας στο σύστημα για την απόκτηση πλήρων και σωστών αποτελεσμάτων. Το παραγόμενο πλάνο εκτέλεσης εκτελείται με ένα πλήρως κατανεμημένο τρόπο στέλνοντας στους απαραίτητους κόμβους τις σχετικές (υπο-)ερωτήσεις. Η επικοινωνία μεταξύ των κόμβων για την ανταλλαγή (υπο-)πλάνων και ενδιάμεσων αποτελεσμάτων βασίζεται σε ειδικά κανάλια επικοινωνίας.

Στρατηγικές βελτιστοποίησης χρησιμοποιούνται για τη δημιουργία βελτιστοποιημένων πλάνων εκτέλεσης. Ευριστικές μέθοδοι που αφορούν την διάταξη των τελεστών ένωσης (union) και συνένωσης (join) χρησιμοποιούνται για την κατασκευή αποτελεσματικότερων πλάνων εκτέλεσης. Το κόστος επικοινωνίας και επεξεργασίας χρησιμοποιείται για αποφάσεις σχετικά με τεχνικές μεταφοράς δεδομένων ή επερωτήσεων κατα την διάρκεια εκτέλεσης των επερωτήσεων. Τέλος, η προσαρμοστικότητα στα παραγόμενα πλάνα εκτέλεσης είναι απαραίτητη για την παρακολούθηση της εκτέλεσης των επερωτήσεων και την δυνατότητα μεταβολής του πλάνου εκτέλεσης όταν οι βάσεις των κόμβων γίνονται μη διαθέσιμες ή οι πόροι του συστήματος εξαντλούνται.

Τέλος, οι αλγόριθμοι δρομολόγησης και δημιουργίας πλάνων του SQPeer επιτρέπουν την επικαλυπτόμενη εκτέλεση των δυο αυτών φάσεων. Ειδικότερα για μια δομημένη δυομότιμη αρχιτεκτονική, η επικαλυπτόμενη αυτή εκτέλεση οδηγεί στην δημιουργία πλάνων εκτέλεσης σε πολλαπλά βήματα τα οποία συνδυαζόμενα προσφέρουν πλήρη αποτελέσματα. Πιο σημαντική όμως, είναι η προώθηση της επεξεργασίας μεταξύ ίδιων κόμβων με το επιπλέον προνόμιο της ταχύτερης απόκτησης σχετικών με την επερώτηση αποτελεσμάτων.

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Αναπληρωτής Καθηγητής

Στους γονείς μου

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη μου κ. Βασίλη Χριστοφίδη για την άψογη συνεργασία μας τα τελευταία χρόνια. Οι γνώσεις που απέκτησα μέσω της γόνιμης εργασίας δίπλα του αποτελούν σημαντικά εφόδια. Χωρίς την στήριξή του και την συνεχή βοήθεια που πάντα ήταν διαθετειμένος να προσφέρει, η παρούσα εργασία δεν θα μπορούσε να ολοκληρωθεί.

Επίσης, θα ήθελα να ευχαριστήσω τον καθηγητή μου και μέλος της εξεταστικής επιτροπής κ. Δημήτρη Πλεξουσάκη με τον οποίο είχα την τιμή να συνεργαστώ επιτυχώς ήδη από τις προπτυχιακές μου σπουδές. Επιπλέον, να ευχαριστήσω τον καθηγητή και μέλος της επιτροπής κ. Γρηγόρη Αντωνίου.

I would also like to thank Professor Val Tannen for his contribution during the initial steps of my thesis. Additionally, many thanks to Arnaud Sahuguet, whose thesis has been an inspiration and whose code I borrowed for the evaluation of my own work.

Ιδιαίτερα θα ήθελα να ευχαριστήσω το Πανεπιστήμιο Κρήτης και το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας για τις γνώσεις και τις εμπειρίες που μου προσέφεραν όλα αυτά τα χρόνια.

Ένα μεγάλο ευχαριστώ ανήκει σε όλους τους συμφοιτητές μου και τους συναδέλφους με τους οποίους συνεργάστηκα καθ΄ όλη την διάρκεια των σπουδών μου. Αισθάνομαι τυχερός που μερικές από αυτές τις συνεργασίες κατέληξαν σε πραγματικές φιλίες. Ευχαριστώ λοιπόν την γνωστή παρέα Βάσω, Δέσποινα, Γιώργο, Παναγιώτη και Νίκο για όλες τις εμπειρίες που μοιραστήκαμε και που θα θυμόμαστε για όλη μας την ζωή. Κυρίως όμως θέλω να ευχαριστήσω την Ιωάννα για την συμπαράστασή και την στήριξη που μου παρείχε όλον αυτόν τον καιρό, καθώς και για την πάντα ευεργετική παρουσία της.

Τελευταίο αλλά μεγαλύτερο ευχαριστώ όμως αξίζει στην οικογένειά μου και πιό συγκεκριμένα στους γονείς μου, Φίλιππο και Αναστασία, στην αδελφή μου, Φανή, και στην γιαγιά μου, Φανή. Παρόλο που βρίσκονταν μακριά πάντα ήταν δίπλα μου και με στήριξαν σε όλες τις δυσκολίες. Η εργασία αυτή ελπίζω να αποτελέσει μια μικρή ανταμοιβή για τις θυσίες και τις προσπάθειές τους όλον αυτόν τον καιρό.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last few years there has been a drastic increase in the exchange of electronic information. Each user connected to the Internet has access to a huge amount of online information. This information is usually dispersed through a great number of available bases. Moreover, the user is usually interested in a small portion of the available information and is not willing to spend an excess amount of either time or processing and communications resources for acquiring it.

Towards the goal of organizing data dispersed in different sites, the notion of distributed databases was introduced. A *distributed database* is described as a collection of multiple, logically interrelated databases distributed over a computer network. A *distributed database system* is the software system that permits the management of the distributed database. It is primarily responsible for making the distribution of the data transparent to the user. A system like this, in its simplest form, constitutes of a centralized server that supports a global schema and offers distributed database services, e.g., distributed query processing or data consistency management. The fundamental principle behind data management is data independence, which enables applications and users to share data at a conceptual level, while ignoring implementation details. By this approach, schema management, high-level query capabilities, automatic query processing and optimization, and complex object support can be provided.

As a step forward, peer-to-peer (P2P) computing offers new opportunities for

building highly distributed data systems by evolving the distributed database approach. P2P systems are currently attracting enormous attention, spurred by the popularity of file sharing systems such as Napster [Nap], Gnutella [Gnu], Freenet [CSWH01], Morpheus [Mor] and Kazaa [Kaz]. Such systems have millions of users connected and sharing huge amount of data. Thus, P2P systems need to handle a very large number of autonomous computing nodes (the peers) that pool together their resources and rely on each other for data and services. Unlike client-server computing, a P2P system operates without central coordination, thus providing a very dynamic environment, where peers can join and leave the network at any time. Additional advantages include direct and fast peer communications, self-organization, decentralization in data storage and processing, and the ability to scale up to large number of peers, while ensuring fault-tolerance.

In order to address data management issues in a P2P context, we propose the *SQPeer* middleware. Its main goal is to provide the necessary functionality for semantic query routing and planning of distributed queries posed in a P2P context. More precisely, SQPeer works on top of a dynamic P2P database system consisting of autonomous peers sharing their local bases.

1.1 Motivation

Despite the recent emergence of P2P systems, most of these systems have severe limitations in contrast to traditional data management systems: file-level sharing, read-only access, simple keyword-based search and poor scaling. In most cases, searching in a P2P system relies on simple selection conditions on attribute-value pairs or IR-style string pattern matching. Simple techniques (e.g., network flooding) are used to look up and retrieve relevant data. Moreover, both communication and processing resources are wasted, since no optimizations are usually considered. These limitations are acceptable for file-sharing applications, but in order to support highly dynamic, ever-changing, autonomous social organizations (e.g., scientific or educational communities), we need richer facilities in exchanging, querying and inte-

grating structured and semistructured data hosted by peers. Moreover, considering data management issues in P2P systems is a quite challenging task due to the scale of the network and the autonomy and unreliable nature of peers.

When considering data management, the main requirements of a P2P system are the following:

- **Autonomy:** an autonomous peer should be able to join or leave the system at any time without restriction. It should also be able to have control on its stored data.

- **Query Expressiveness:** the query language should allow the user to describe the desired data at the appropriate level of detail. The simplest form of a query is a key lookup, which is only appropriate for finding files. But in a more complex scenario, where peers store (semi-)structured data, a high-level query language is necessary.

- **Efficiency:** the efficient use of the P2P system resources (bandwidth, computing power, storage), which is completely decentralized, should result in lower cost and thus higher throughput of queries, i.e., a higher number of queries can be processed by the P2P system at a given time. Specifically, query processing can be performed in a distributed way by a number of peers, thus taking advantage of remote peer resources.

- **Quality of Service:** refers to the user-perceived efficiency of the system, e.g., completeness of query results, data consistency, data availability, query response time, etc.

- **Fault-tolerance:** efficiency and quality of services should be provided despite the occurrence of peer failures caused by the dynamic nature of the system.

In addition to the above, the importance of intensional (i.e., schema) information for integrating and querying peer bases has been highlighted by a number of recent projects [BGK+02]nejdl03 [HIST03] [ACMH03]. More precisely, the notion of Semantic Overlay Networks (SONs) [CGM03] [TXKN03] appears to be an intuitive way to

cluster together peers sharing the same schema (or different schemas glued together with appropriate mapping rules) about a community domain or application model. This approach facilitates query routing, since each peer has the means to identify relevant to a query peers, instead of broadcasting query requests on the network. A natural candidate for representing such schemas is the Resource Description Framework /Schema Language (RDF/S). The modeling primitives of RDF/S are crucial for P2P databases where monolithic RDF/S schemas and resource descriptions cannot be constructed in advance and peers may have only incomplete descriptions about the available resources. In this context, several declarative languages for querying and defining views over RDF/S description bases have been proposed in the literature such as RQL [KAC+02] and RVL [MTCP03]. However, a full-fledged framework for evaluating semantic queries over peer RDF/S bases (materialized or virtual) is still missing. More precisely, a semantic processing mechanism should produce an appropriate execution plan in order to efficiently answer a query by retrieving data dispersed throughout the system.

1.2 Contributions

Considering the aforementioned functionality, we present the design and implementation issues of the SQPeer middleware for semantic query routing and planning in an RDF/S-based P2P system.

SQPeer utilizes RDF/S for expressing, querying and creating views over the contents of a peer base. More precisely, conjunctive RQL queries expressed against an RDF/S schema-based SON are represented in our middleware as *query patterns*. A novel technique is introduced for advertising peer RDF/S bases using intensional information. In particular, we are employing *RVL view patterns* for declaring the parts of a SON RDF/S schema which are (or can be) actually populated in a peer base. *Query/view subsumption* techniques are also used to identify peer views relevant to a given query with the use of their intensional information.

Query processing in SQPeer is responsible for the generation of appropriate *query*

plans. More precisely, a semantic query routing phase includes a *data localization algorithm* that matches a given RQL query against a set of RVL peer views in order to determine relevant peer bases based on appropriate query/view subsumption techniques. This algorithm relies on a *fragmentation phase* that efficiently breaks a complex query pattern given as input into a set of simpler query patterns. A lookup service is also used to efficiently discover the remotely available peer advertisement information. In the end, the routing phase produces *query patterns annotated with the relevant peer views*.

We introduce a semantic query planning phase that involves an *algebraic translation algorithm* responsible for formulating query plans from annotated query patterns taking into account the involved data distribution (e.g., vertical, horizontal) in peer bases. The produced query plan is executed by establishing appropriate *communication channels* between the relevant peers. Additionally, we discuss several *compile* and *run-time optimization opportunities* for SQPeer query plans and introduce an appropriate cost model for executing a dynamic programming approach in order to optimize the formulated query plans.

As far as the execution order of the query routing and planning phases is concerned, we propose two different alternatives affecting both the nature and the evaluation of the produced query plans. So in contrast to a *sequential* execution of the two phases, we also present an *interleaved execution of query routing and planning phases* as an alternative scenario in processing a distributed query.

In addition, we overview recent approaches for Distributed and P2P Database Systems. The systems are categorized based on the their routing and planning capabilities, with each category facing different challenges over the common problem of distributed query processing.

Finally, we present how SQPeer can be utilized in order to deploy *hybrid*, *structured* and *ad-hoc* P2P database systems. The challenges and special characteristics of each architecture are considered for efficiently evaluating a query in the specified context.

The work presented in this thesis has been published in the proceedings of the First International Workshop on P2P Computing and Databases (P2P&DB) [KC04], in the

proceedings of the Third Hellenic Data Management Symposium (HDMS), as a book chapter in "Semantic Web and Peer-to-Peer" by S. Staab and H. Stuckenschmidt, and has been presented in the 8th International Workshop of the DELOS Network of Excellence on Digital Libraries on Future Digital Library Management Systems (System Architecture & Information Access) [KSDC05].

1.3 Outline of the Thesis

The thesis is structured as follows:

Chapter 2 discusses the related work and presents an overview and a proposed classification for Distributed and P2P Database Systems.

Chapter 3 describes the use of RDF/S in a SON-based P2P system. More precisely, we illustrate the representation of RQL queries and RVL views as query and view patterns respectively. We also discuss the notion of query/view subsumption. Furthermore, we discuss issues emerging from the consideration of different architectural alternatives, i.e., hybrid, structured and ad-hoc P2P architectures.

In Chapter 4 we describe how SQPeer query plans are generated. More precisely, the core algebra used for the representation of RQL queries as query plans is introduced. Then, the algorithms for routing and planning are illustrated by additionally considering the fragmentation, data localization and algebraic translation phases. Cost-based optimization issues are also addressed with the use of a dynamic programming-based query optimizer. The interleaved query routing and planning scenario is given as an alternative for processing and executing a query considering data distribution and optimization aspects. Then, we illustrate how the query is executed with the use of appropriate communication channels and several run-time optimizations that can be performed during query execution. Through a set of experiments we point out the advantages of interleaved query routing and planning compared to their sequential execution.

Finally, Chapter 5 summarizes our work and contributions and discusses future work.

Chapter 2

State Of The Art

In a P2P system, a large number of autonomous nodes (the peers) pull together their data and/or computing resources, relying on each other for data and services. P2P systems can be seen as an extension of traditional client/server systems, where nodes may behave as both servers and clients at the same time. Furthermore, each peer may enter or leave the system at will, thus providing a higher degree of autonomy; the system, however, should continue being functional irrespectively of the autonomous decisions of each peer. P2P computing introduces a paradigm of decentralization going hand in hand with self-organization and high degrees of autonomy among the participating peers.

In a P2P Data Management System (PDMS) each peer maintains its own database, which shares either in part or as a whole with the rest of the system. As usual, the key concept in a PDMS is query evaluation, which, since data is network-bound, consists of two steps: (i) query planning (i.e., identifying which sources to access and in what way), and (ii) query routing (i.e., shipping computation and/or data across peers in order to compute the query result.) Before describing how query routing and planning is implemented in PDMSs, however, we will give an overview of query processing in the more traditional paradigm of Distributed Data Management Systems (DDMS) [OV91]. Figure 2.1 illustrates the main query processing steps in a DDMS [Kos00]. The query processor accepts a query and produces a query plan specifying precisely how the query is going to be executed. Query plans are usually represented as trees.

The nodes of such trees correspond to the operators of the query, while the leafs represent the information sources (e.g., tables) that need to be combined to produce the final result. In the first phase, the query is *parsed* and transformed into an algebraic representation for further processing. *Logical optimization* applies algebraic transformations regardless of the data distribution in the DDMS. These transformations include normalization, elimination of redundancies and simplification of algebraic expressions. The next phase, called *data localization*, generates a location-based query plan, where all relations of the query are bound to specific data sources. A single (logical) relation may be bound to one or more (physical) remote sources. The *physical optimizer* receives a localized query plan and by using heuristic or cost-based techniques produces an optimized query plan taking into account inter- and intra-source query processing and communication cost. The optimized plan is finally sent to the *execution engine* responsible for forwarding the subplans to the appropriate sources and monitoring their evaluation. The execution engine is also responsible for (a) establishing the proper communication links between remote sources contributing to the query plan and (b) adapting query plans [ILW+00] during query execution in case of run-time problems.

DDMSs handle data management issues by providing advanced capabilities, such as schema management, high-level query languages, access control, automatic query processing and optimization, etc. Most importantly, users can transparently access and acquire data residing in several remote databases. However, a centralized control, usually provided by a single server, is required, reducing the system to scale up to tens of databases. On the other hand, PDMSs adopt a completely decentralized approach to data sharing [VP04]. By distributing data storage, query processing and execution across autonomous peers, these systems can scale up to very large number of peers, without the need for central control and powerful servers. In addition, efficiency and quality of services are provided despite the occurrence of peer failures caused by the dynamic nature of the system.

More precisely, autonomy, which is one of the most significant characteristic of a PDMS, can be distinguished into four kinds [KP04]. *Storage* autonomy refers to the

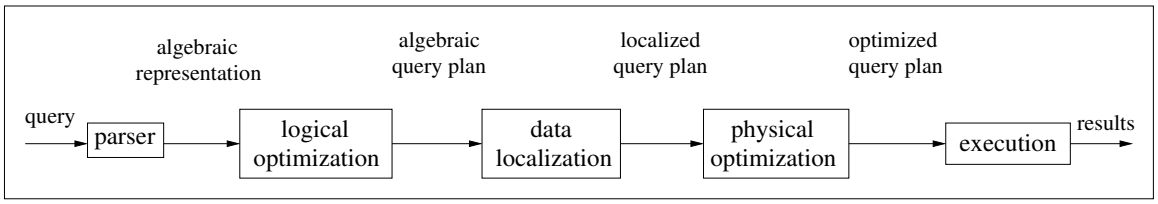


Figure 2.1: Typical architecture of query processing in DDMS

freedom of what a peer stores in its base and what it wants to advertise or replicate to remote peers. *Execution* autonomy refers to the ability of a peer to locally process and answer queries allowing coordination between peers for efficiently performing distributed query processing. *Lifetime* autonomy refers to the peers' freedom to join and leave the system arbitrarily, which actually differentiates these systems from the traditional DDMSs. Finally, *connection* autonomy refers to the topology of the system enabling a peer to select with which and how many peers it will connect to.

In this context, several assumptions made by DDMSs during query processing are revised by PDMSs creating additional requirements:

- *Single administrative structure*: A query optimizer in traditional DDMSs decides to fragment a query into certain subqueries and execute it into different peers. These peers will not deny the execution of a query, e.g., due to processing load, since their selection is done centrally. Such a "good neighbor" assumption is not usually true in a PDMS.

- *Uniformity*: Traditional DDMS query optimizers generally assume that all processors and network connections are of the same speed. Additionally, every peer is considered to be capable of executing all algebraic operators. Both assumptions do not usually hold in real-scale PDMSs.

- *Fixed Query Plans*: Unlike DDMSs where the query plans are fixed, run-time adaptation of plans to the network or peer failures is an important feature of PDMS. Adaptability can be achieved by either changing entirely or partially the currently executed query plan.

In DDMSs only one peer is responsible for all the query processing steps, except

for the last one, since a set of peers are involved in query execution. On the other hand, a peer in a PDMS can potentially assist in all query processing steps. Despite the plethora of PDMSs proposed in the literature, a typical P2P query processing architecture is still missing. For the purposes of our survey, we focus on the following four main query processing activities and highlight their specific characteristics in PDMSs:

Query Routing. Query routing is responsible for finding the peers relevant to a query by considering the data partitioning (either horizontal, vertical or mixed) on each peer. Query routing relies on advertisements concerning the peers' bases. Advertisements usually contain information about the data (extensional) or the schema (intensional) of the peers. In particular, in PDMSs focussing on data integration, intensional advertisements take the form of views. Peer advertisements can be either centrally stored in a registry at a single peer, or they can be distributed through a number of peers of the PDMS. In the distributed alternative, each peer contains a subset of the peer advertisements that when combined offer complete data localization information. In this case, the lookup service for retrieving the distributed advertisements may be performed either by a single peer or by multiple collaborating peers. Query routing may involve a query fragmentation phase, where the query is split into distinguished fragments matching the available peer advertisements.

Query Planning. Query planning is responsible for generating a distributed query plan that governs the query's execution in a fully distributed way by considering all local or remote peer bases contributing fully or partially to the query. The main challenge during query planning is to determine an appropriate ordering among the query operators, in order to execute them efficiently in remote peers. This ordering is also dictated by the planner's choice between intra- or inter-peer processing. Based on this decision the query plan is created by promoting operations whose operands are stored either at the same (intra) or different (inter) peers. Data can be partitioned among the peers vertically or/and hori-

zontally. In vertical partitioning, joins are used to combine data that reside in remote peers. Alternatively, in horizontal partitioning, set union is employed to assembly the final answer. Query planning may be either under the responsibility of a single peer or of a number of cooperating peers. In the former case, query planning usually relies on a global knowledge of peer base advertisements, resulting in the construction of optimal and complete plans. In the latter case, this knowledge is distributed at different peers and only partial query plans are created at each peer. More specifically, these partial plans are created in multiple steps by considering peer base advertisements obtained at each consequent peer.

Query Optimization. The goal of the physical query optimization phase (see Figure 2.1) is to produce an efficient query plan with respect to the estimated query evaluation time. Most widely used optimization techniques are either heuristic- or cost-based [OV91, LPR98, Sah02]. Heuristics allow us to optimize query plans by, for example, pushing as much as possible query evaluation to the same peers. This actually promotes intra-peer processing, which by definition is more efficient than the inter-peer one. The cost model used in the cost-based optimization estimates the total resource consumption or the response time for a given query. The difference between these two cost metrics is that the former does not consider possible parallelization in plan execution, i.e, the fact that subplans may be executed simultaneously at different peers. The cost is usually computed by adding the processing cost of the query plan's operators at each relevant peer and the communication cost from the data exchange between them in order to complete query execution. For calculating this cost, appropriate statistics concerning peers' network and bases are needed. Statistical information may be either centralized or distributed among peers. In the latter case, optimization can be performed with incomplete statistical information, leading to partly optimized query plans. More precisely, query optimization can be decomposed into local and global optimization phases [OV91]. In local opti-

mization, a peer receiving a (sub)query is responsible for ordering the operators of the query using statistics regarding the contents of its local base. In global optimization, a peer responsible for executing a query to other peers, decides the ordering of the query operators using statistics about the communication cost or the processing capabilities of remote peers. In addition, it decides which peer will undertake the responsibility of executing each operator of the query. Since the number of all possible query plans, i.e., all possible operator orderings, can be fairly large, appropriate plan enumeration algorithms are required. IBM's System R [SAC+79] dynamic programming algorithm is one of the most popular algorithms employed for generating an optimal plan if the cost model is sufficiently accurate. Plans are constructed in a bottom-up fashion, since more complex plans are built from simpler ones by selecting at each step an optimal subplan. Extensions like iterative dynamic programming [KS00] have been also proposed with the goal of decreasing dynamic programming's exponential complexity by further pruning the search space.

Query Execution. Query execution is responsible for organizing the query's evaluation phase by (i) executing the subqueries at the appropriate peers, with respect to the produced query plan and (ii) sending the results back to the peer that posed the query. The produced query plan may be either centrally coordinated or fully distributed. In the former case, a single peer is responsible for sending the subqueries to the appropriate peers and for performing the rest of the processing locally after receiving the results. In the latter case, several peers communicate with each other in order to execute the plan. Subquery results are exchanged between the participating peers and when complete results are computed, they are sent back to the requester. In this respect, appropriate query execution policies are considered by assigning peers to plan operators, namely, data shipping, query shipping, or hybrid shipping. In data shipping, a peer asks for appropriate resources and all further processing takes place locally. In query shipping, subqueries are sent to remote peers, thus pushing operators

	Centralized Query Planning	**Distributed Query Planning**
Centralized Query Routing	*(1) One peer contributes to both QP and QR* ▷ Wide-Area DDBs	*(2) Several peers contribute to QP and one peer to QR* ▷ Wide-Area DDBs
	Query Scheduling Systems: DQS [LPR98], ObjectGlobe [BKK+01], Ginga [PLP02], SAIL [SHB04]	Auction-based Query Planning Systems: Mariposa [SAL+96], Query Trading [PI04]
Distributed Query Routing	*(3) Several peers contribute to QR and one peer to QR* ▷ PDMS	*(4) Several peers contribute to both QP and QR* ▷ PDMS
	Systems: ***a) Distributed Index Shipping*** RepositoryGuide [BG03] ***b) Clustered PDMS*** XPeer [SMGC04], Edutella [NWS+03] [BDK+04], Resource Sharing Clustered P2P [TXKN03], SWAP [ETB+03], KeX [BBMN02] ***c) Structured PDMS*** AmbientDB [BT03] [Bra03], P2P XML Processing [GWJD03a], SemanticPeer [TDL04], Bookmark-driven QR [BMWZ04], RDFPeers [CF04]	Systems: ***a) Mapping-driven PDMS*** Piazza [HIST03], GridVine [ACMHP04] ***b) Mobile Planning PDMS*** HyperQueries [KW01], ubQL [Sah02], MQP [PMT03], ActiveXML [MAA+03] [ABC+04], XML Search Engine [GWJD03b] ***c) Adaptive Planning PDMS*** FREddies [HJ04], SwAP [Zho03]

Table 2.1: Categorization of systems based on query routing and planning alternatives

to be executed remotely. A hybrid approach is possible by combining the two previous execution policies. The decision for the appropriate shipping approach is taken during the query plan optimization along with the query operators ordering as discussed previously.

In Table 2.1, we propose a classification of PDMSs in two axes by considering their query routing and query planning capabilities. As far as query planning is concerned, we distinguish between those systems that produce query plans in a centralized way and those that produce them in a distributed way. The difference lies to the fact that one or several peers may contribute to the query planning process. Respectively, in query routing we distinguish between systems that are based on centralized or distributed advertisements. This means that either one peer or a specific advertisement registry maintains all necessary data localization information, or each peer independently handles a subset of peer advertisements and, consequently, participates in the

routing phase. The order between retrieving data localization information and producing the query plan can be either sequential or interleaved. In the former case, query routing is guided by query planning. The first and third family (i.e., (1) and (3)) of systems shown in Table 2.1 involves systems where the two phases are distinct. The second and fourth family of systems (i.e., (2) and (4)) query routing and planning are indistinguishable, since the query plan is produced in several steps according to the relevant peer advertisements, which are available at each point in time.

As far as query routing is concerned, the choice of the structure for maintaining the peer advertisements is implied by the substantial tradeoff between reducing the number of lookup requests and the amount of distribution and autonomy we desire from the PDMS. Extensional advertisements are used in the systems of the first and third categories, while intensional ones are used by all the systems except the ones of the first category.

Another classification criterion would be the topology of the PDMS, but we consider that this choice is subsumed by the two factors already considered in our classification schema. More precisely, there exist three possible architectural alternatives for P2P systems. In *ad-hoc* or *pure P2P systems*, each peer simply needs to know and initially contact at least one other peer. The main challenge in this context is routing, since each peer is connected to the system through its neighbors, so a clever advertisement structure and routing process should be implemented. The *structured P2P systems* organize their peers according to specific constructs, which provide the means for an efficient and distributed organization and storage for peer advertisements. The *hybrid approach* involves the existence of special peers, called *super-peers* [YGM03], which have better processing capabilities and are usually highly-available peers. Super-peers actually organize the simple peers and are capable to hold all the necessary advertisements of their subordinate peers. In hybrid PDMSs query routing is completely performed at the super-peer level.

The classification scheme based on query routing and planning implies four distinct families of systems, which are described thoroughly below.

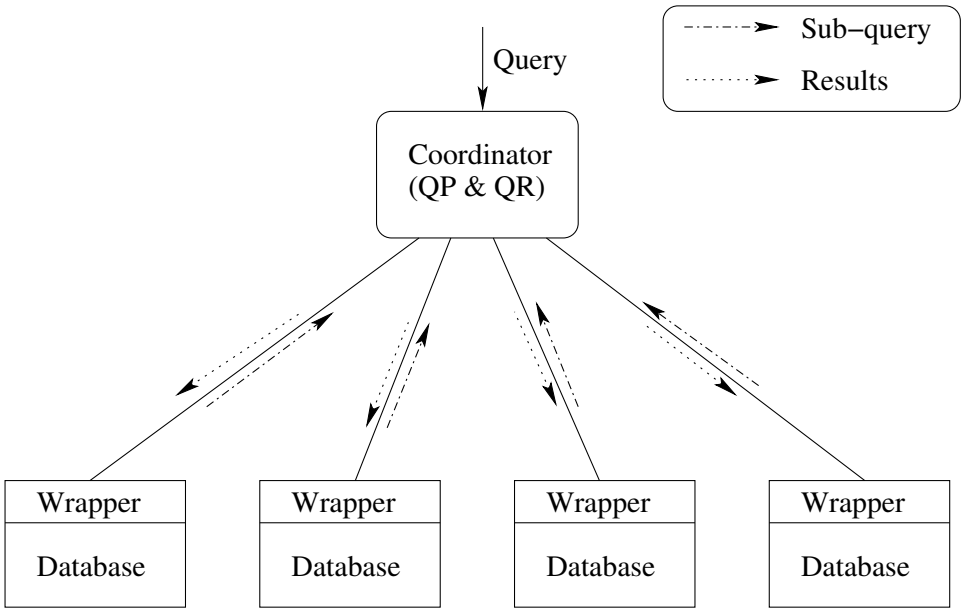


Figure 2.2: Centralized QR and QP

2.1 Centralized Query Routing and Centralized Query Planning

The first category comprises systems where both query routing and planning are centralized, i.e., a single peer controls query processing and remote peers contribute in query execution as simple servers, as seen in Figure 2.2. Traditional DDMSs can be included in this category, since in these systems a centralized registry maintains a global knowledge of the peer capabilities and employs this knowledge to produce the appropriate query plans.

Query routing is based on the centralized advertisement registry and the coordinator (or mediator) peer decides which peers should be used for executing parts of the initial query (i.e., subplans of the generated query plan). After a complete query plan is produced, the coordinator contacts the respective peers in order to send them the subqueries they can answer. Each peer executes the query locally and sends the produced results back to the coordinator, which then produces the complete answer. The main architecture in Figure 2.2 illustrates the interactions between the involved peers and how routing is performed by systems belonging to this category. By this

approach, the benefit of having global knowledge is exploited, thus leading to a centralized control of both query planning and choice of execution alternatives (e.g., data *vs.* query shipping).

Query processing in these systems slightly differs from the model presented in Figure 2.1. A *local query optimization* layer is introduced; local optimization is performed by all the peers involved in the query plan. After subqueries are sent to appropriate peers for execution, the receiving peer locally optimizes its subquery by using only its local statistics and schema information. Local optimization relies on techniques from traditional database systems. Through this layer, each peer locally contributes to the optimization of the query, since all global decisions are already taken by the coordinator peer at the global query optimization layer. The global and local query optimization layers correspond to the physical optimizer shown in Figure 2.1.

In the following subsections we present different approaches for centralized query routing and planning.

Distributed Query Scheduling Service

In [LPR98], a distributed query scheduling service (DQS) is introduced in the context of the DIOM data integration system based on the relational data model and SQL-like queries. DIOM relies on a two-tier architecture and offers services at both the mediator and the wrapper tier. DIOM mediators utilize peer advertisements for efficient processing of distributed queries. Additionally, each peer base has a wrapper in order to be capable of communicating with the DIOM system, since it additionally provides the appropriate translations between terms of the underlying source and those used by the mediator. The peer bases considered by DIOM range from structured to non-structured ones. Each such base is autonomous, but if its exported schema changes it should notify its respective DIOM mediator. Each mediator organizes a metadata repository, which contains all available peer advertisements. Both types of data partitioning (i.e., horizontal or vertical) are considered.

After a user issues a query, an ordered binary query plan tree is created by employing commonly used single peer optimization methods [OV91]. Then, query rout-

ing and parallel query plan generation are used to produce an appropriate so-called *location-based* query plan. Query routing locates the relevant peer bases for answering parts of the query from the corresponding metadata repository manager. The parallel query plan generation decides the degree of intra-operator parallelism used throughout the query plan, i.e., the number and position of union operators throughout the query plan. Additionally, the union collector peers are decided for handling each union and thus manage the parallel execution of query (sub-)plans.

The main service provided by DQS is query optimization. Global optimization criteria are used to decide between different execution policies for a given plan. A cost model taking into account communication, processing cost and response time is used to estimate the total cost of a query plan. A three-phase optimization approach is employed. At first, during query compilation, well-known heuristics are used, such as pushing down selections and/or projections, or considering only join orderings that do not produce Cartesian products. The next phase is query parallelization in which parallel execution of the query plan is addressed. Appropriate heuristics are used to decide the union and join ordering that produces the best plan possible. The final optimization phase comprises of peer selection and query execution, where the peers for executing the query plan operators are chosen. This selection is cost-based and all available peer choices are considered for finding the most efficient one. This phase actually decides between data and query shipping alternatives.

ObjectGlobe

ObjectGlobe [BKK+01] is a distributed query processor for peer bases. ObjectGlobe supports a nested relational data model, thus relational, object-relational and XML data sources can be easily integrated. Peers in ObjectGlobe have certain roles. Data providers supply the data, cycle providers handle the execution of the query and metadata providers perform the query routing phase. The metadata providers maintain repositories containing the attributes used in a peer base, appropriate statistics and authorization information, i.e., both intensional and extensional peer advertisements. All metadata providers form a backbone, where each peer publishes descrip-

tions of their relations expressed in RDF. Data consistency is preserved by replicating the appropriate metadata knowledge to all backbone providers.

Query processing in ObjectGlobe distributes query operators to cycle providers in order to execute complex queries. The lookup service provides the necessary information on what relations should be accessed and if such access is alowed (e.g., user authentication or read/write privileges). Furthermore, the query optimizer creates a valid query execution plan, based on the received data localization information, trying to additionally fulfill possible user constraints. An iterative dynamic programming algorithm is used to find the optimal execution plan by considering both horizontal and vertical data partitioning. The cost model employed considers processing and communication overheads, but also considers whether users' constraints in terms of response time or execution cost can be satisfied. Each operator of the output plan is annotated with the cycle provider that is responsible for its execution. Then, the query plan is transmitted to the responsible cycle providers and necessary communication paths are established. Finally, the execution phase is initiated and results are returned to the user.

ObjectGlobe provides a Quality of Service (QoS) model and allows QoS constraints for the query plan to be posed by the user; for example, constraints on the number of results returned or the cost in terms of the evaluation time of the query. These constraints restrict the search space of query plans, since only plans that satisfy them all are enumerated. A monitoring system focusses on query execution in order to detect and react to potential quality violations by changing cycle providers during query execution.

Ginga

The Ginga system [PLP02] addresses adaptive query processing issues in a wide area DDMS. A Ginga server, taking on the role of coordinator, is responsible for processing a query and obtaining results from all the underlying peer bases.

A query is initially processed by the server's query manager, which coordinates client query sessions and routes queries according to DIOM's routing service. Data

localization involves the selection of the appropriate peer bases by discarding those that cannot contribute to the query. Each query is transformed into a set of subqueries, with each subquery assigned to one of the chosen peers. Since no horizontal partitioning is considered in Ginga, the obtained query plans only comprise of only joins.

The adaptive nature of Ginga query plans is achieved in two phases. The *reactive adaptation engagement phase* is performed before query execution and during query planning. An initial optimized plan is generated using DQS. Additionally, a set of alternative plans are also considered. Assumptions on runtime problems that may be encountered are made. The optimal plan is added to the set in case these assumptions hold. Along with each such alternative plan are the triggers capturing the runtime environmental changes that will generate the selection and execution of each respective query plan. The *reactive control phase* is performed during query execution. The conditions involved in the aforementioned triggers are checked and when one is fired, the corresponding alternative query plan replaces the current one. This change is done only if it is beneficial for the total execution of the query and depends on how far into execution of the current query plan the system has currently progressed. Ginga's centralized control of query execution facilitates monitoring and adaptation distributed query plans.

SAIL

In [SHB04] an architecture for querying distributed RDF bases is introduced. Queries are expressed in a query language similar to RQL. SAIL consists of several RDF bases with a mediator on top for processing queries. Each such RDF repository should provide a certain RDF API (called SAIL) to communicate with the mediator SAIL. Research focusses on the problem of creating an appropriate index structure for distributed RDF bases and implementing query planning and optimization algorithms at the mediator level.

The SAIL index structure contains information about subpaths or complete paths that can be constructed by employing an index hierarchy over a given RDF schema.

The hierarchy is formed in such a way that each index of a schema path p always contains the indices for all prefixes of p. For example, for the SeRQL query "$\{A\}$ *author* $\{W\}$ *carriesout* $\{P\}$ *topic* $\{$*'RDF'*$\}$", which asks for all the authors that carry out a project in the area of RDF, the index hierarchy for the path (*author*, *carriesout*, *topic*) will include (*author*, *carriesout*) and (*carriesout*, *topic*), along with the singleton paths (*author*), (*carriesout*) and (*topic*). Along with each path, the index contains the peer bases that can answer it and the number of results that each peer base holds. This information can be used either for estimating the communication cost, or for deciding on the join ordering. Both horizontal and vertical partitioning are considered.

The query planning algorithm uses the index structure to create a complete query plan. Starting from the complete path, the algorithm iterates by splitting this path into all its possible path fragments. In each iteration, a join is used to connect the results from the path fragments (vertical partitioning) and a union aggregates the results obtained from previous iteration steps (horizontal partitioning). Query optimization resolves the problem of join reordering at the mediator and does not consider possible interchange between joins and unions of the query plan. The cost model employed considers both communication overhead (for retrieving the results from the underlying bases), as well as join reordering at the mediator.

The obtained plan is executed by sending the appropriate subqueries to the underlying bases and shipping the results back to the mediator for local processing. By the nature of the produced query plans, all inter-site joins and unions are executed by the SAIL mediator.

2.2 Centralized Query Routing and Distributed Query Planning

In systems of this category several peers may contribute to query planning, although a centralized registry is used for holding all necessary peer base advertise-

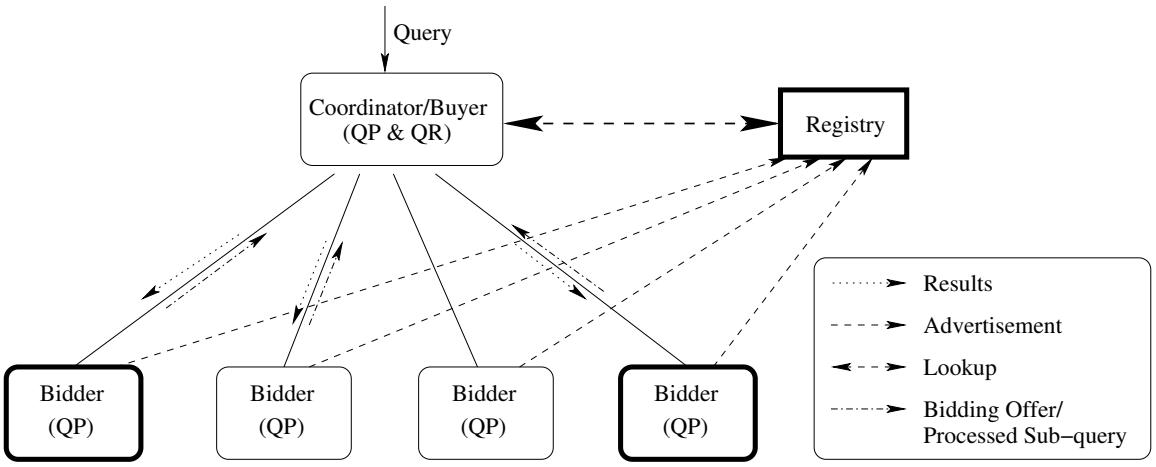


Figure 2.3: Centralized QR and Distributed QP

ments. The existence of a centralized registry supports the creation of complete query plans at the peer responsible for query processing, but does not determine plan execution. The latter is carried out in a distributed fashion.

Query processing in this category bears similarities with the traditional DDMS approach. However, this category poses some new challenges. Along with data localization, a *query fragmentation* phase breaks the query into distinct fragments, which are subplans containing one or more query operators. These fragments are routed independently and are considered to be answered as a whole by one or more peers. Each fragment's execution, however, is not considered to be performed by the peer it is initially sent to, as is the case in the previous category, but several peers may contribute by answering parts of it. The fragmentation and localization phases are interleaved, since fragmentation is affected by the answering capabilities of the underlying peer bases. Since query subplans can be executed at different peers, the query optimizer may decide to push a subplan to be executed at another peer for optimization reasons.

The two systems of this category use an auction-based query processing approach. A bidding phase is used to decide both on query fragmentation and localization. The peers involved in the bidding phase contribute to query planning, although data localization information is centrally available. In Figure 2.3, an example of how such

a system works is shown. Initially, each bidder advertises descriptions of its stored data to a centralized registry. Then, a broker acting as the coordinator can ask and obtain all the data localization information concerning a given query. Each bidder after receiving a query can simply return its preferred bid or can additionally further process the query according to its own capabilities. Thus, each bidder participates in the query planning phase by contributing subplans to the broker based on local knowledge. We must point out that not all bidders will finally contribute to query execution, since winners (e.g., the ones depicted with a bold outline in Figure 2.3) are elected from the bidding phase and these are the ones who should be contacted by the coordinator to answer the query. A bidder wins when it offers the best execution cost for the bidding subplan. This cost can be measured either in total execution time or in some other predefined metric.

In the following subsections we present different approaches for centralized query routing and distributed query planning.

Mariposa

Mariposa [SAL+96] addresses distributed data management issues in a wide-area network (WAN) based on the relational data model and the SQL3 query language. Each such query references tables that may be fragmented at different peers. These fragments can obey range- or hash-based partitioning criteria or may simply be unstructured, when no criteria is introduced.

Cost-based global optimization in Mariposa follows a microeconomic paradigm. Peers decide their role in the system (i.e., brokers or bidders) and are able to take decisions concerning their behavior on buying, selling or bidding scenarios. Name servers, acting as registries, are special peers responsible for storing and providing the relevant metadata for a given query, including the peers that can answer it. Advertisements are used by the bidders, i.e., peers that bid for a certain query, in order for the names servers of the system to become aware of them.

Query processing in Mariposa works in the following way. Queries are submitted by a client application. Each query starts with a certain budget, which defines what

the user is willing to pay in terms of time or in terms of some other user-specified metric. The query should be processed within this allocated budget. The query is fragmented into a set of subqueries, processed by different peers. Since each subquery answered by a peer comes at a certain cost, appropriate peer bases should be selected through a bidding process. The query is parsed by receiving appropriate metadata from a name server, thus identifying for example the name and type of each attribute and the location of the fragments of each table it needs to access. A single-peer optimizer generates a locally optimized query plan as if all the referenced tables were stored locally. The next phase, i.e., the query fragmentor, uses the localization information of the table fragments in order to efficiently transform the optimized query plan. These transformations include the decomposition of each single resource of the query plan into subqueries, one per fragment of the table responsible for answering this resource. Union is used to merge the partial results for each fragment. Moreover, joins are decomposed into one join subquery for each pair of fragments of the two joined relations. Lastly, operations can be organized accordingly in order to be processed in parallel for intra-query synchronization reasons. Finally, the broker receives a query plan containing a collection of subqueries that combined can answer the original query. It then sends requests to the bidders responsible for each subquery. Each bidder sets its price for the execution of the subquery it receives and sends it back to the broker. The broker decides on which bid it should accept according to rules obtained by its preferred protocol. The relevant peers are notified for executing the fragments they have been assigned. A coordinator handles the tasks of assembling partial results and returning the final answer to the user.

Query Trading

Similar to the Mariposa approach, this work [PI04] focuses on distributed query optimization by using a query trading approach. The architecture of the system is similar with the one used in Mariposa, with autonomous peers that act as buyers or sellers and use their own negotiation protocols. Queries are seen as commodities exchanged between peers and query optimization is considered as the trading nego-

tiation process for deciding the best execution plan for obtaining these commodities from the sellers.

The query planning and optimization algorithm iterates on the bargaining process between buyer and seller nodes by considering different negotiated queries and possibly different sellers for each iteration. The input of the algorithm is a query Q with an initially estimated cost C. More precisely, a set of subqueries q, which combined produce query Q, with their estimated costs start the iterative bargaining process. The output is the best estimated execution plan P for query Q, i.e., the one that minimizes the total execution cost. This cost takes into account the network resources and current workload of the sellers. During each iteration step the buyer sends the set of subqueries to the selected sellers. Each seller identifies the parts of each subquery that it can answer and rewrites it by removing all non-local relations and binding those available locally to the appropriate tables. Then the optimal local plan for these queries is identified. A modified dynamic programming algorithm, which holds intermediate optimal subplans, is used to find this plan and its cost. The final plan, along with the intermediate results, is returned to the buyer, since they are considered for the buyer's optimal plan resolution. The buyer, after receiving the sets of offers made from the sellers, selects the winning ones. The buyer query planner considers these subqueries and finds a set of candidate execution plans for the original complete query by again using a dynamic programming approach. An analyzer examines each candidate execution plan and produces an enhanced set of subqueries for the next iteration step. These new subqueries may be simple modifications of the previously used ones (for example by removing redundant attributes or attributes not used in the final plan). During each iteration the best execution plan is selected and if it is better than that of the previous iteration step or if the set of subqueries is modified by the analyzer, the iteration continues. When the iteration completes, the selected plan is executed by informing the appropriate sellers and obtaining the final results.

2.3 Distributed Query Routing and Centralized Query Planning

Systems in this category rely on distributed peer base advertisements and create, at a single peer, a complete query plan according to the data localization information obtained. The efficiency of the routing process is the primary focus of these systems. PDMSs that are appropriately organized and generate query plans in a centralized way are introduced. The peer responsible for the query initially requests and retrieves the necessary data localization information by using distributed (intensional or extensional) peer advertisements. This suggests the necessity of intensional or extensional peer organization, since broadcasting messages to the whole system for retrieving data localization information consumes both bandwidth and processing resources. Afterwards, the peer locally computes the appropriate query plan. Again, the information gained concerning data distribution in the system implies the plan fragmentation into subplans that can be answered by the same peer. The query plan is finally executed, sending the subplans to the relevant peers and retrieving the results for further local processing. So, the routing phase is introduced first and query planning is performed in the sequel.

We can split this category into two main subcategories; systems that organize their peers into clusters based on the notion of Semantic Overlay Networks (SONs) and the structured systems that use Distributed Hash Tables (DHTs) to efficiently implement the routing phase.

SONs were initially introduced in [CGM03] as a form for organizing semantically relevant peers. In SONs, peers classify their data into one or more leaf concepts of a specific classification hierarchy. This classification also dictates the arrangement of the peers to their relevant SONs (see Figure 2.4). Each peer can be part of many SONs but for illustration purposes only a single SON is presented in the figure. Query processing identifies which SONs and correspondingly which peers are better suited to answer a query. When a peer issues a query, it first classifies it and then sends it to the appropriate SONs. Then, the peers within each SON find matches to the query

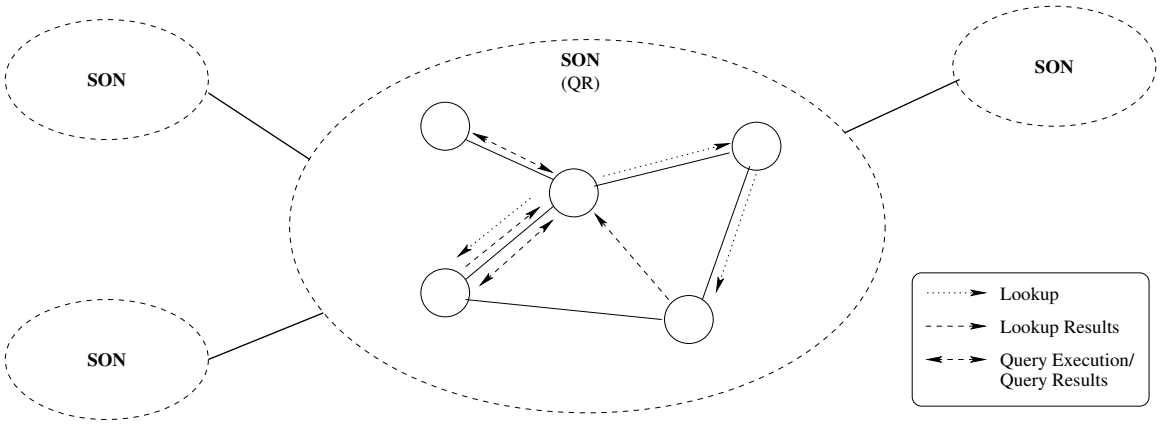


Figure 2.4: Distributed QR and Centralized QP in a Clustered PDMS

by propagating it within the SON. Data localization information is sent directly to the initiator peer and used to create the appropriate query plan. The joining of a peer into the system and the propagation of a query inside a certain SON can be done either by a flooding Gnutella-like mechanism or by the help of super-peers.

On the other hand, DHTs are useful lookup structures for P2P applications. DHTs allows to quickly reduce the amount of nodes needed to answer a given query with data spread through the P2P system (see Figure 2.5). Given a key, the corresponding data item can be efficiently located using only $O(\log n)$ network messages, where n is the number of peers in the network. Several protocols (e.g., Chord [SMK+01], CAN [RFH+01]) implement DHTs and provide lookup operations for mapping a given key to a specific peer. All the peers are responsible for the routing phase by providing and/or retrieving information about peer ids that can answer a given query. Query execution involves contacting a set of peers and retrieving results for the query.

Before we describe the systems for the above two categories, we make note of a system following a different approach in query processing, following what is known as the *index-shipping* approach. Through index-shipping, query processing focuses on routing indices among different peers and evaluating chains of joins between them. This approach is different than the rest of the systems, since the main concern now lies not in the efficient execution of the query plan through the appropriate selection of data and query (fragment) shipping, but in the efficient discovery and evaluation

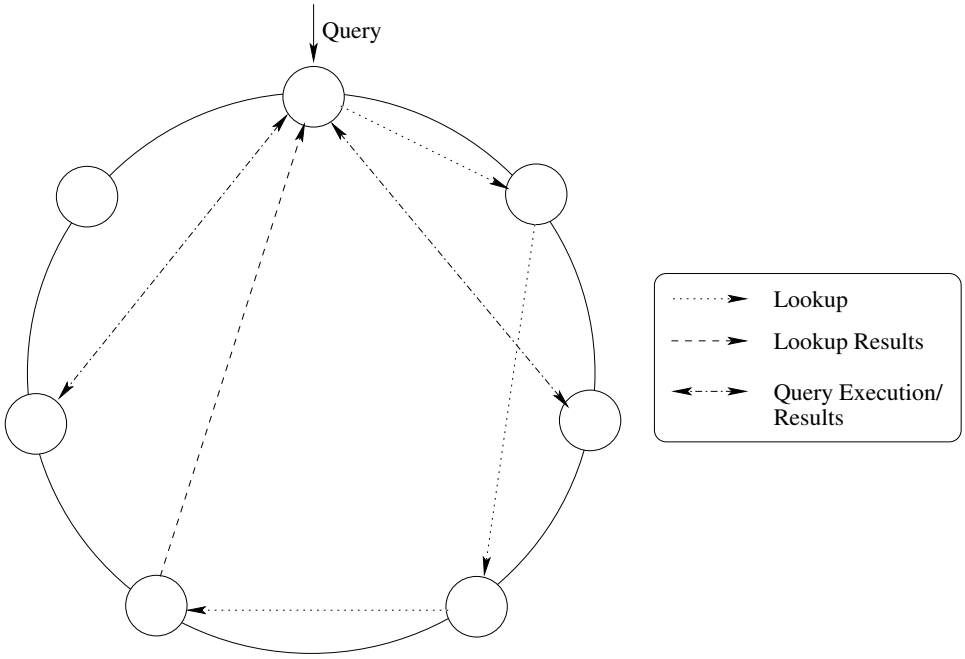


Figure 2.5: Distributed QR and Centralized QP in a Structured PDMS

of the data localization information used for the creation of complete query plans.

2.3.1 Distributed Index Shipping

RepositoryGuide

In [BG03] a distributed approach for a large-scale XML repository is introduced. This work discusses the problem of efficient fragmentation of a globally accepted XML schema by using index structures that can be well distributed through an XML repository. XML data are assumed to be modeled as rooted, node-labeled trees. Queries consist of path and tree patterns, where edges represent ancestor-descendant relationships and node labels represent constraints, similar to XPath. The global XML schema is given in a tree-structured representation, called RepositoryGuide (RG). An efficient fragmentation scheme for the RG is introduced and an extension to this RG, called Distribution RG (DRG), is used to hold the information on which peers are responsible for each fragment. These fragments are distributed throughout the system. Each fragment is bounded to a list of peer ids that can answer it. DRG

can be fully replicated at each peer and thus query planning can be supported by any peer. Vertical fragmentation is considered, though an approach for horizontal fragmentation is discussed as future work.

A query can be issued at any peer and query results should be returned to the initiator peer. The query tree pattern is initially matched with the local DRG for determining potential matches based on label paths. These matches return lists of peer ids that may be stored either locally or remotely and should be properly joined in order to produce the final set of peers that need to be contacted. This introduces the concept of index shipping, since indices are routed through the system and processed appropriately for producing the data localization information required for the planning phase. Consequently, distributed query processing in this setting involves identifying the most efficient way to gather these peer id lists into a common peer and compute structural joins between them. Then at query execution, the detected peers are contacted and the final XML results are produced.

2.3.2 Clustered PDMSs

XPeer

XPeer [SMGC04] is an XML PDMS, which manages data dispersed over a network of autonomous peers. A hybrid P2P architecture is used and no global schema is defined. Peers share XML data, which can be queried by a subset of XQuery without universally quantified predicates and sorting operations. Peers are logically organized into clusters managed by super-peers. These clusters may be organized according to schema similarities between the peers, i.e., peers exporting data with similar schemas are clustered together, thus super peer-based semantic overlay networks are created. Each peer advertises a description of its base, which contains all the distinct paths in its XML documents along with statistical information about value ranges. Super-peers use this knowledge to handle both query routing and planning. Super-peers are organized into a tree-like logical topology.

XPeer query processing initially involves the transformation of queries into al-

gebraic expressions by a user-selected peer. Then, the peer routes the query to its super-peer in order to locate the appropriate peers that can answer each expression. If more than one peers are capable of answering an expression, the super-peer can decide between unioning the results (union operator) or choosing the one that executes the query the fastest (choice operator). The query is routed inside the super-peer network in a hierarchical way until the whole query is localized. Then it is sent back to the initiator peer for executing it. Before execution, logical optimization is performed by applying well-known algebraic rewritings, e.g., selection push-down. For execution, the query is split into single-location subqueries, which are sent to the relevant peers and are executed locally.

Edutella

The Edutella project [NWS+03] explores the design and implementation of a schema-based P2P infrastructure for the Semantic Web. In Edutella, a number of extensible RDF/S schemas describe peer content. Super-peers, i.e. peers with advanced capabilities, undertake the responsibility of message routing, integration/mediation of peer bases and query planning. Appropriate indices are used for the lookup phase and by acquiring the appropriate data localization information the (sub-)queries are initially routed within the super-peer backbone and then to their respective "sub-peers", where the actual data resides. These indices consider the schemas and properties used by a certain peer (intensional) and also the property values that may contain (extensional). The HyperCup topology [SSDN02] is used to form the super-peer backbone in such a way that super-peers can communicate in an efficient way.

A query processing mechanism in such a schema-based P2P system is presented in [BDK+04]. Query evaluation plans (QEPs) containing selection predicates, compression functions, joins, etc., are pushed from clients to super-peers where they are executed. Super-peers use an optimizer for generating partial query plans. Initially, each logical resource of the query is bound to a resource direction, if it is sent to another super-peer, or physical resource, if it is directed to a sub-peer's base, based on the index. The plan created consists of two parts, (i) the locally-executed query

plan and, (ii) the remaining subqueries that should be broadcasted to the rest of the super-peers. The remote subqueries are grouped by host and are sent to the relevant super-peers. Then, cost-based optimization is performed in order to identify the peer that should execute each operator of the local query plan. Cost estimation is based on response times, transfer rates or even result sizes of previous queries. Different optimization strategies are considered, such as equivalence transformations of a query plan aiming at turning a join of unions into a union of joins. Additionally, collector nodes are selected during plan generation in order to handle the union of resources from different peers. This selection is made by considering each peer load for load balancing purposes.

Resource sharing clustered P2P system, SWAP & KeX

In [TXKN03] a PDMS is presented, where peers are organized into clusters, based on the semantic categories of their documents. Each peer stores a set of documents, which it shares with the rest of the system. Additionally, it keeps an amount of metadata that help organize the routing process. This metadata is kept in tables and includes mappings of documents to semantic categories, mappings of categories to clusters and finally mappings of clusters to lists of peer ids. These tables store data localization information and act as intensional indices for guiding the query through the network. Each such query is a set of user-defined keywords that describe the documents that the user wishes to retrieve.

Query processing in this context works in the following way. User queries are initially submitted to the system. The keywords of the query are mapped to one or more semantic categories. The cluster of peers that is responsible for these categories is found by the appropriate indices and a randomly selected peer of the retrieved cluster is chosen to route the query to. The target peer then identifies its local documents that answer the given query. If the number of results do not exceed the threshold set by the user, the query is sent to a neighbor of the target peer belonging to the same cluster until the desirable number of results is reached. The results are directly sent back to the requester by each processing peer. A MaxFair algorithm for

intra-cluster load balancing is used to achieve fairness, by considering the processing power and the contribution of each peer in the cluster.

Similar approaches to the above system are SWAP [ETB+03] and KeX [BBMN02]. SWAP is an ontology-based P2P knowledge management system consisting of a set of peers called "SWAP nodes". RDF/S is used for the representation and sharing of peer base advertisements and queries are formulated in an RQL-like query language. An internal inference engine initially tries to locally solve the query. If no answers are returned from the local repository, the query is fragmented and distributed to the rest of the system. Fragmentation and routing are based on metadata about the remote peers. The remote peers process these subqueries in the same fashion and eventually return results to the initiator peer. All operators that combine the received results are computed locally to produce the final answer.

KeX peers, on the other hand, use negotiation protocols in order to be organized into groups forming federations that agree in certain conceptual terms, i.e., provide relevant resources. Each peer can play two main roles: a provider and/or a seeker. A provider publishes to the rest of the system its local knowledge along with its categorization concerning a specific hierarchy (intensional). A seeker submits a query by activating a query session. In such a session, the seeker receives asynchronously answers from the providers to which the query is broadcasted to. The providers, after receiving a query, either execute it locally and send results back to the seeker, or broadcast it further. Broadcasting is performa in a semantic way by discovering federations or peers related to the query.

2.3.3 Structured PDMSs

AmbientDB

AmbientDB [BT03] addresses P2P data management issues in applications such as autonomous audio players exchanging music collections. AmbientDB provides full relational database functionality and assumes the existence of a common global schema. However, each peer may contain its own schema as long as it provides

the necessary mappings to the global one. A peer has access to a local content repository that is organized inside a SON-like AmbientDB subsystem. In AmbientDB, apart from the local tables, horizontal partitioning is considered, since fragments of a table may be stored at a number of peers with (Partitioned Tables) or without (Distributed Tables) replicated tuples. A P2P protocol set at each peer is responsible for query routing and uses Chord to connect all peers in a resilient way and as the basis for implementing the distributed table extensional indices. These indices are used to identify all possible fragments of a certain distributed table inside AmbientDB. Each AmbientDB peer contains the index table partition that corresponds to it after hashing the key-values of all tuples contained in the distributed table to a finger that Chord maps on that peer.

Each AmbientDB peer contains a Distributed Query Processor, which is responsible for the execution of queries on all ad-hoc connected peers. A query is posed on both/either local and/or distributed tables. The query processing mechanism is based on a three-level translation of a global query algebra into stream-based query plans, since streams are used to facilitate the exchange of partial results through the appropriate peers. Initially, a user query is posed in standard relational algebra, which provides the operators for selection, join, aggregation and sorting over abstract relational tables. Then, this abstract query plan becomes concrete by instantiating the abstract tables with concrete ones, i.e., the local or distributed tables that exist in the peer bases. Finally, at the execution level, the concrete query plan is executed by choosing between different query execution strategies. These strategies include the selection between distributed (at each peer containing a fragment of a distributed table) or centralized execution of unary operators, i.e., selection, aggregation, or ordering. Additionally, joins can be executed either locally or by routing the tuples of the local table and execute the join at each relevant peer. Moreover, integrity constraints, such as foreign keys, can be exploited, since a join between a local table and a distributed one over a foreign key produces matching tuples only locally. This can minimize communication costs, since the join can be executed at the peer holding the local table after retrieving all the tuples from the distributed tables.

In [Bra03] a super-peer based approach for AmbientDB is used, which allows hierarchical clustering of peers and facilitates query processing by first sending the query to a super-peer and then flooding it to the super-overlay network. The super-peer receiver is responsible for query processing and will eventually return all the results back to the initiator peer.

P2P XML Query Processing, SemanticPeer & Bookmark-driven Query Routing

In [GWJD03a] an approach for processing paths in XML data in a DHT-based P2P system is introduced. A structure of data summaries, which contain all paths leading to a given element tag, are used to assist query routing. Given a certain XPath query, the system initially extracts the "ending" element tags. These tags are used as lookup keys for the DHT to find the peer that stores the relevant data summaries. These data summaries are used in order to find the peer ids that can answer a given query path. Finally, the initiator peer, after receiving the list of relevant peers, contacts them for executing the matching subqueries. System evolution, such as peer arrivals and departures, as well as scalability issues are also discussed.

A similar approach is found in SemanticPeer [TDL04], a DHT-based P2P lookup service. The difference is located in the existence of a specific table that resides at each peer and holds the shared knowledge expressed in RDF between the peers of the system. This shared knowledge is composed of the domain concepts and the instances upon which most users agree. The query can then be processed either by contacting the DHT and/or by checking its local shared knowledge table.

In [BMWZ04] a similar DHT-based P2P system is presented. The improvement offered by the system lies in the pruning of the candidate peers by considering a benefit/cost measure computed as the ratio between the estimated result quality and execution cost. The benefit factor also considers the bookmark overlapping between two peers, i.e., the amount of same Web pages that these peers have crawled, and reflects the thematic similarity between them. Peer bookmarks are seen as intensional indices shared between peers that have similar interests. The bookmark URLs can

also be looked up in the DHT, therefore the bookmark of the query initiator can be used to fetch semantically similar peers.

RDFPeers

RDFPeers [CF04] is a scalable distributed RDF repository based on an extension of Chord, namely MAAN (Multi-Attribute Addressable Network), which efficiently answers multi-attribute and range queries. RDQL is used for querying peer bases and operates at the RDF-triple level without taking RDF Schema information into account. Peers are organized in a virtual circle similar to the Chord ring. In MAAN, each RDF-triple is hashed and stored for each of its subject, predicate or object values in the corresponding positions in the ring. Furthermore, for arithmetic attributes MAAN uses order preserving hash functions in order to place close arithmetic values to neighboring peers in the ring for the evaluation of range queries.

Routing is done as in Chord by searching for each value in the query. Simple query patterns, where subject, predicate or/and object value is given, can be answered in the given context. Consequently, disjunctive, range and conjunctive multi-predicate queries are answered by allowing the expression of constraints on the query's values. From each query the relevant query patterns are identified and the appropriate routing actions are taken. Query routing involves sending lookup requests through the Chord ring and retrieving appropriate data localization information stored at each peer in the ring. Joins are handled by using value constraints on the joining attributes and are performed at the issuing peer.

Finally, the RDFPeers system handles overly popular URIs and literals by applying a popularity threshold for RDF terms. This threshold restricts a certain peer from further storing triples whose number exceeds it. Additionally, load balancing techniques are considered in order to generate a node identifier distribution adaptive to the actual data distribution in the system.

2.4 Distributed Query Routing and Distributed Query Planning

The last category involves systems in which peers contribute in both the routing and the planning phases. Query routing and planning are not anymore two separate phases but are rather executed in an interleaved way. This means that the data localization, plan generation, query optimization and execution phases are interleaved, since the complete query plan is produced at multiple steps that may involve multiple peers.

In particular, query processing (and, consequently, query execution) can be done in multiple steps as the query is being routed through the system gathering additive data localization information. This is posed by the distributed nature of the query plans, which now do not hold the restriction of being created and initiated by the requester peer. Moreover, information on the location of certain parts of the query is usually unknown to the requester, so shipping the query or part of it to the rest of the system is mandatory. Distributed peer advertisements guide the routing of the query plan. Another approach, found in some systems of this category, is the routing of partial results along with the rest of the query plan and its development or its annotation with these data as it is routed through the PDMS.

We will divide the systems into three subcategories. The first group involves those systems where mappings are used to guide the query routing and planning phases. In particular, queries are routed according to the local mappings of each peer and may be reformulated in order to correspond to terms used by remote peer bases. The systems of the second group introduce the notion of mobile query plans, where plans and data are exchanged between peers and each peer may contribute to the processing phase by providing additional data localization information or undertaking the execution of a subplan. A third group involves systems where adaptability at the execution phase is their most important characteristic and, so, query plans can be altered at run-time when network or peer failures arise.

Figure 2.6 outlines an example on how query processing that involves both dis-

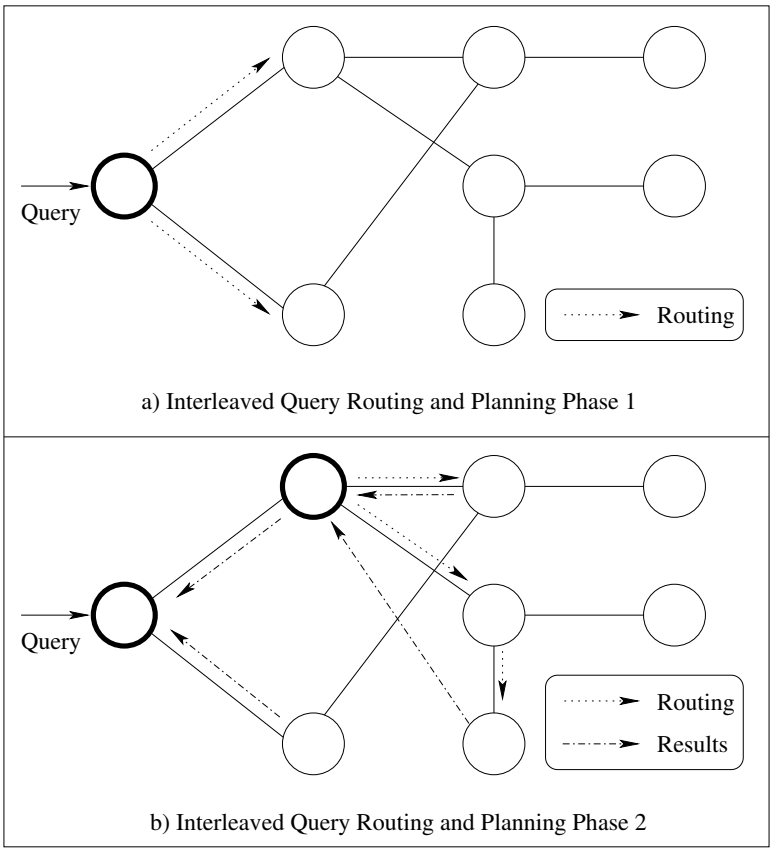


Figure 2.6: Distributed QR and QP

tributed query planning and routing is implemented. Since query planning and routing are interleaved, the query is initially routed through the PDMS and according to the data localization information received acts accordingly. Each peer locally decides on either the execution of a subplan (and receiving and locally processing intermediate results), or further routing of the subquery to a more appropriate peer. In Figure 2.6a, the initiator peer discovers two relevant peers and sends them the relevant subqueries for evaluation. In Figure 2.6b, the second peer decides to further route its subquery in order to find other relevant sources. When the complete query plan is fully localized, the appropriate peers send their partial results to those peers that undertake query processing (depicted by a bold outline) and complete results are sent back to the requester.

2.4.1 Mapping-driven PDMS

Piazza

Piazza [HIST03] is a PDMS for XML, which primarily focuses on the use of schemas and the definition of schema integration and mapping techniques for P2P systems. A Piazza application consists of many peers, each of which can either provide a schema or supply source data (or both). A new peer in the system should become semantically related to a portion of the existing system. Mappings are used in order to semantically glue together those peers. Queries are always posed from the perspective of a given peer's schema, which defines also the preferred terminology used by the user. The query answering algorithm proceeds as follows. Given a query Q over the schema of a node P, the algorithm initially checks for the mappings that refer to local data. Next, the semantic neighbors of P are considered, i.e., all nodes that are related to elements of P's schema by semantic mappings. These mappings are used to reformulate the query in order for other peers to be able to answer it. The answers are returned to the initiator peer and are unioned with the ones obtained locally. This process is done recursively until no useful paths between nodes are found. Concluding, mappings guide query routing and so query processing is reduced to the appropriate reformulation of queries, with no further optimization.

GridVine

GridVine [ACMHP04] presents a DHT-based P2P architecture, where data are expressed in RDF/S and RDQL is used to formulate queries. A two-layer architecture is presented distinguishing the logical data layer from the physical one. The logical layer, i.e., GridVine, supports the operations needed for the maintenance and use of a SON supporting semantic interoperability. Operations at the logical layer allows us to insert RDF-triples, RDF schemas and OWL schema mappings or search for a given query. The physical layer is based on P-Grid, an efficient, self-organizing and fully decentralized access structure. P-Grid uses DHTs and implements its two basic functionalities for retrieving and inserting a value based on a hash key. The values

searched through the DHT may correspond to the subject, predicate and/or object value of the requested RDF triples, therefore P-Grid is based on extensional indices.

Semantic gossiping is presented as a technique for fostering semantic interoperability in a decentralized setting. Peers in GridVine are allowed to create and share mappings from their own schemas to remote ones. These mappings are used to appropriately translate a query and thus to implement query routing between peers employing different schemas. Query processing, apart from the gossiping part, relies on simply sending the query down to the physical layer. P-Grid then uses the DHT to lookup relevant peers, send them the query and retrieve the appropriate results.

2.4.2 Mobile Planning PDMS

HyperQueries

QueryFlow [KW01] is a system offering dynamic and distributed query processing using the notion of HyperQueries. HyperQueries are essentially subplans that exist in peer nodes and guide the processing of a query through the network. The proposed framework is based on virtual attributes expressed in XML, whose values are determined by evaluating remote subqueries on demand.

Hyperlinks are defined by Uniform Resource Identifiers (URIs), which are used for query routing issues. Users initially specify HyperQueries, which are transformed into an operator tree stored as a subplan in a local repository. Three such templates are used by QueryFlow. A dispatch operator, which splits one input stream into multiple output streams, implements the nesting of subplans. A union operator merges the returned results and produces a single output stream. The second pattern is used for sequencing subplans by also using the dispatch operator but no union is used for collecting the results. Further processing is not controlled by the initiator subplan and a surrounding one should exist that will handle them. The final pattern implements the inner subplans, since one input and one output stream exist. This pattern is used for the innermost parts of the query execution, where the actual values of the virtual attributes are determined.

For a more complicated scenario involving multiple peers, QueryFlow introduces two types of HyperQuery execution. Hierarchical HyperQuery execution indicates that a host that receives a query and encounters a virtual attribute acts as intermediary and initiates a nested subquery at a remote host. According to this scenario, the produced objects should be returned to the surrounding plan and eventually at the final step to the initiator node. This hierarchical execution offers a certain amount of control over the distributed query plan. Broadcast HyperQuery execution, on the other hand, suggests the delegation of the evaluation of a certain object to the remote peer. Thus, a fully distributed and autonomous query plan is created and the produced object is returned not to the "upper" peer, but to the actual initiator of the query.

In this framework, certain optimizations are discussed, such as predicate migration and result caching at the intermediary remote hosts. Experiments show the scalability of the framework according to the number of clients and objects that participate in the system.

Ubiquitous Query Language

ubQL [Sah02], which stands for ubiquitous query language, provides a suite of process manipulation primitives that can be added on top of any traditional query language to support distributed query optimization. In ubQL, queries are encapsulated into query processes that can migrate and configured between peers. ubQL distinguishes the deployment and migration of a query process from its execution. Each peer contains (intensional) views, expressed as replicated query processes, and physical data that it shares with the rest of the system. Communication channels support the communication between query processes that reside at different peers and the exchange of data between them.

The deployment phase is responsible for routing the query process through the system and gather the necessary information for its execution. To control the deployment at the level of the language, annotations are used that make possible the migration of subqueries to remote peers. Deployment strategies govern the way a

query process is handled by a peer. After a peer receives a query process, it checks whether it is possible to merge it with its own local information. The peer then has the choice between performing data or query shipping. Moreover, a peer can decide according to the data localization information contained in the query process to optimize it locally and send the corresponding subqueries for further process to remote peers. Optimizations include the application of dynamic programming or a simple reordering of the joins, if only local ones are considered. Different strategies lead to different types of optimizations. The cost model used considers both transmission and local processing cost.

Finally, ubQL supports adaptability of query plans during the execution phase by offering specific adaptive operators. These operators can monitor the execution process and may react by changing the whole or part of the present query plan.

Mutant Query Plans

Mutant Query Plans (MQPs) [PMT03] introduce a new solution to the problem of query routing in a P2P framework. A MQP is a logical query plan, where leaf nodes may consist of URN/URL references, or of materialized XML data. MQPs are themselves serialized as XML elements and are exchanged among the peers. When a node S receives a MQP P, S can resolve URN references and/or materialize URL references, thus offering its local intensional data localization information. Furthermore, S can evaluate and re-optimize MQP subplans adding XML fragments to the leaves. Finally, it can route P to another server. If P is completely reduced to XML markup, it is sent to the target node, i.e., the node originating the query. As a consequence, a MQP traverses the system, carrying partial results and unevaluated subplans, until it is fully evaluated, i.e., it becomes an XML fragment.

The efficient routing of MQPs is preserved by information derived from multi-hierarchical topic namespaces, e.g., educational material on computer science or geographical information. These namespaces are handled in a distributed way by assigning roles to certain peers in the system. Consequently, index or category servers exist for organizing the categorization of queries into areas according to their semantics.

These servers are also responsible for routing, since they provide information for peers that are relevant to a given query.

A similar approach for evaluating active XML documents with embedded web service calls has been presented in ActiveXML [MAA+03] [ABC+04]. In ActiveXML, intensional XML documents, where some of the data are given explicitly, while other parts are defined by means of embedded calls to Web Services, are exchanged between peers. A flexible data exchange model is introduced, where intensional data are materialized either by the sender or by the receiver of an intensional document. The sender, after agreeing with the receiver on the data exchange schema, should choose between a set of equivalent, increasingly materialized documents, by considering whether to materialize intensional information. Algorithms are presented to find one such document by considering the possible restrictions posed by either safety or schema issues on the processed data or services.

XML Search Engine

In [GWJD03b] an XML distributed search engine is presented to support a PDMS. An XML search engine is able to execute containment queries that fully exploit the inherent structure of XML documents. Inverted lists, which map keywords to documents, are used as indices for the local peer data. Additionally, remote data are indexed by peer inverted indices exchanged at the joining phase of each peer. In order to limit the number of peers that one is aware of, horizons are used that confine this number below a certain threshold. This means that indices that should point to peers outside the horizon now point to selected peers at the edges of the horizon, so the number of indices stored locally at each peer is reduced without losing the assurance that the required data localization information will be eventually discovered.

Query processing is governed by the estimation that it is more beneficial to send queries directly to the relevant peer bases, than route the whole query through the system. The search engine breaks the query into simple subqueries and matches them with the peer's indices to identify relevant peers. Intersection or union of the returned set of peers leads to a final set, where the whole query is sent for evaluation

and execution. The choice between intersection or union of these sets is made by considering the query operators used to bind the simple subqueries. By this type of routing and processing only local optimizations are possible and no intra-peer operations are executed.

2.4.3 Adaptive Planning PDMS

FREddies

FREddies [HJ04] is an extension of the centralized Eddy operator [AH00] for use in a P2P query processing system. FREddies operate within the framework of PIER. PIER is a DHT-based P2P query processor. Following the functionality of Eddies, a FREddy is a query plan operator that dynamically routes tuples through local operators, some of which may send the tuple to another peer in the network. A DHT, used by PIER, organizes peers in an overlay routing layer, where data can be directly addressed.

Query processing in this PIER context with the use of FREddies is done in the following way. First, a query plan is created, which determines the operators that are necessary to execute the query and the possible operator routing order by avoiding, for example, cross-products. All nodes participating in the execution are aware of this same plan. When a query arrives at a certain node, all operators, including the FREddy, are instantiated. The FREddy then begins dataflow by requesting tuples from each of its sources. Metadata on each tuple are used to identify the operators it already passed through. Routing policies, including an order-specific, a random choice and one according to the input queue length of each operator, are used to route the tuples through the operators of each FREddy. These policies dictate the order of the processing of tuple until its complete processing. This decision is taken at execution time resulting in run-time optimization. Additionally, by routing each semi-processed tuple with the use of DHT, distributed query processing is achieved.

SwAP

SwAP [Zho03] is another large scale distributed system, where Eddies [AH00] are used to achieve run-time query plan adaptability. Each peer contributing in the query plan is equipped with an Eddy providing adaptability for local operations. A preparatory phase performed by a coordinator peer produces a distributed processing graph. The graph contains all processing peers that handle the necessary operations. Additionally, it contains information on the types of parallelism to be employed. The whole processing plan is then launched and appropriate operators for intra-peer communications are set. Tuples are routed through each peer for processing until it completes all the necessary operations. At each peer, further routing of each query is performed according to run-time statistics. These statistics are gathered by the exchange of virtual tuples between peers, which contain zero data, but provide a way to monitor query execution.

Chapter 3

RDF/S-based SONs

In this chapter, we introduce the notion of RDF/S-based Semantic Overlay Networks (SONs) as the basis for the SQPeer middleware. Initially, we give a description of what is a SON and how we can use it to efficiently organize the peers. Then, RDF/S is introduced to provide efficient mechanisms for describing, querying and creating views on top of a peer base with the additional use of appropriate query and view definition languages. We show how these two worlds merge into what we call RDF/S-based SONs and how we handle the challenges that emerge in a context like this. Finally, we overview three architectural alternatives for deploying SQPeer and how we can cope with the special characteristics of each one.

3.1 Semantic Overlay Networks

SONs were initially introduced in [CGM03] for organizing semantically relevant peers. In SONs peers should classify their stored data into one or more leaf concepts of a specific classification hierarchy. This classification dictates also the assignment of each peer to its relevant SON(s). A peer is added to a SON, if it has any documents classified in its corresponding concept or in a less conservative scenario, if a "significant" number of documents are classified under that concept. Different classification hierarchies may be employed to specify SONs with different characteristics.

In this context, query processing simply identifies which SONs can actually answer

the query. When a peer issues one query, it is initially classified in the same way as its documents are. Then, the query is matched and sent to the appropriate SONs for execution. The peers within the selected SONs find more specific and relevant matches to the query by propagating it within their overlay network. The joining of a peer into the system and the propagation of a query in a specific SON can be done either by a flooding Gnutella-like mechanism or by the help of super-peer nodes, i.e., highly-available nodes capable of organizing the peers in the system.

By introducing the notion of SONs, peers are grouped based on the semantics of their stored data. Since queries are routed according to the same semantically-based classification policy, documents are found faster and only relevant to the query peers are considered. Peers that have few or no results considering a given query will not be contacted, since their classification will assign them to different SONs, thus avoiding wasting processing and communication resources on that requests.

In general, SONs appear to be an intuitive way to cluster together peers sharing the same schema information about a community domain or application model. A natural candidate for representing such descriptive schemas (ranging from simple structured vocabularies to complex reference models [MACP02]) is the Resource Description Framework/Schema Language (RDF/S). RDF/S is chosen, since its modelling primitives are the most appropriate for P2P databases, where monolithic RDF/S schemas and resource descriptions cannot be constructed in advance and peers may have only incomplete descriptions about the available resources. In the following sections, RDF/S along with the appropriate languages for querying and defining views over RDF/S description bases are described and used in the context of a SON, thus introducing the notion of *RDF/S-based SONs*.

3.2 Resource Description Framework and Schema Language

The Web provides a simple and universal infrastructure to exchange various kinds of information. In order to share, interpret, and manipulate information worldwide,

the role of *metadata* is widely recognized. Indeed, metadata allow us to easily locate information available in the Web, by providing descriptions about the structure and the content of the various Web resources (e.g., data, documents, images, etc.) and for different purposes. The emergence of the Resource Description Framework (RDF) [RDFa] [RDFb] was expected to enable metadata interoperability across different communities or applications by supporting common conventions about metadata syntax, structure, and semantics.

The Resource Description Framework (RDF) is a general-purpose language for representing information about resources in the World Wide Web. It is particularly intended for representing metadata that can be identified on the Web and moreover resources that cannot be directly retrieved on the Web. RDF is meant for situations in which this information needs to be processed by applications, rather than being only displayed to people. It provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning. More precisely, RDF provides (i) a standard representation language for Web metadata; and (ii) a schema definition language (RDF/S) to interpret (meta)data using specific class and property hierarchies (i.e., vocabularies).

RDF's vocabulary description language, RDF Schema, is a semantic extension of RDF. It provides mechanisms for describing groups of related resources, the relationships between these resources and determine characteristics of other resources, such as the domains and ranges of properties. In particular, RDF/S (a) enables a *modular design* of descriptive schemas based on the mechanism of *namespaces*; (b) allows easy *reuse* or *refinement* of existing schemas through *subsumption* of both class and property definitions; (c) supports partial descriptions since *properties* associated with a resource are by default *optional and repeated* and (d) permits *super-imposed descriptions* in the sense that a resource may be multiply classified under several classes from one or several schemas.

RDF is based on a directed graph model that implies the semantics of resource descriptions. The basic idea is that a resource (identified by a URI) can be described through a collection of statements forming a so-called RDF description. A specific

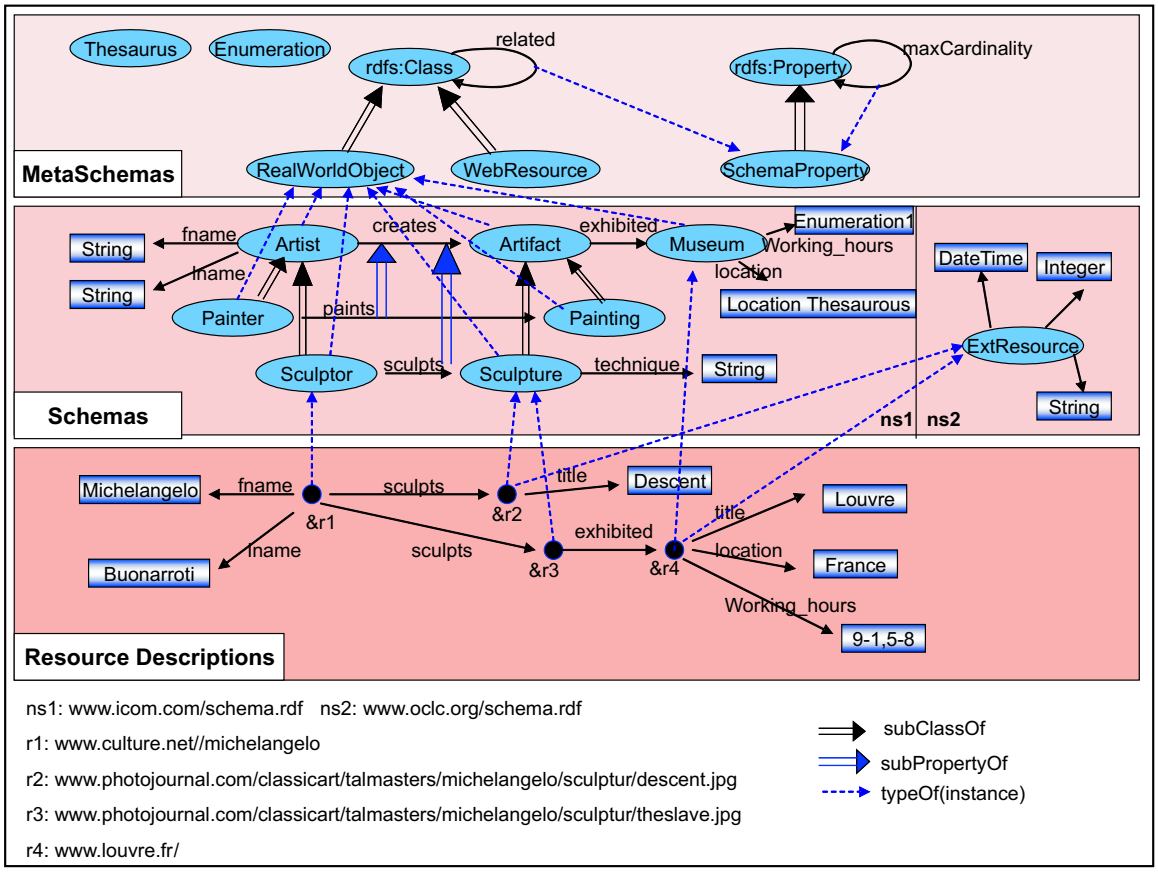


Figure 3.1: The graph representation of RDF

resource together with a named property and its value is an RDF statement. The value of a property can be another resource or a literal. A literal is either a simple string or another primitive data type. RDF/S schemas are then used to declare vocabularies, i.e., collections of classes and properties, that can be used in resource descriptions for a specific application or domain.

We can see RDF through three different points of view - representations: (i) as RDF graph, (ii) as RDF 3-tuples (the set of statements described in triples) and (iii) as RDF syntax (provides some standard ways for describing data using XML).

The RDF graph is a syntax-neutral way of representing RDF expressions using directed labelled graphs. These graphs are also called nodes and arcs diagrams. Each arc represents a named property. Each property connects two nodes, coming from a node representing a resource (drawn as oval) and pointing to another resource or a

literal (drawn as rectangle). An example of the RDF graph is shown in Figure 3.1.

The other way to represent RDF is 3-tuples, also called triples. Each triple {`s, p, o`} corresponds to an arc from the subject `s` to the object `o`, labelled by the predicate `p`. This 3-tuple representation is an easy way to reenact a very huge set of statements, since it needs less space in comparison with the other two representations.

3.3 Constructing an RDF/S-based SON

In SQPeer we consider that each peer provides RDF/S descriptions about information resources available in the network that conform to a number of community RDF/S schemas (e.g., for e-learning or for museum artifacts). These schemas are considered to be available and known to the rest of the system and moreover to be widely accepted by communities of peers. Peers employing the same schema to construct their RDF descriptions in their local bases belong to the same SON. Thus, *RDF-based SONs* are formulated for each RDF schema available and peers are organized in a semantic way into groups having the same interests and needs. In a more complicated scenario, SONs may be constructed by peers contributing RDF descriptions specified by local schemas that can be appropriately mapped to the SON schema.

In the upper part of Figure 3.2, we can see an example of an RDF/S schema defining such a SON, which comprises four classes, `C1`, `C2`, `C3` and `C4`, that are connected through three properties, `prop1`, `prop2` and `prop3`. There are also two subsumed classes, `C5` and `C6`, of `C1` and `C2` respectively, which are related with the subsumed property `prop4` of `prop1`. Finally, classes `C7` and `C8` are subsumed by `C5` and `C6` respectively.

3.3.1 RQL Peer Queries

One of the proposed RDF query languages satisfying the need for a sufficiently expressive declarative language for querying both RDF descriptions and schemas is RQL [KAC+02]. RQL is a typed functional language and relies on a formal model for directed labeled graphs permitting the interpretation of superimposed resource

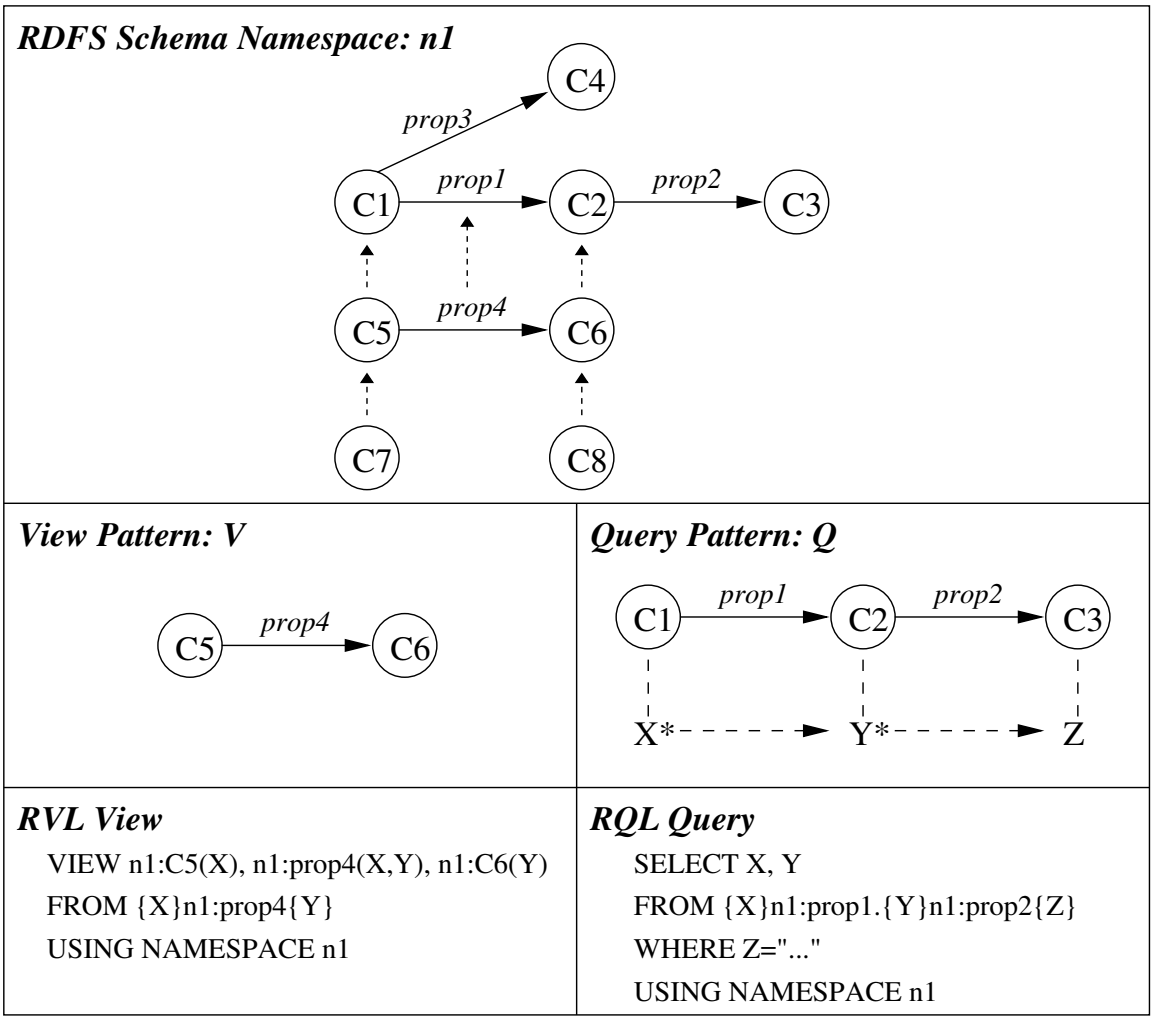


Figure 3.2: An RDF/S schema of a SON, an RVL view and an RQL query pattern

descriptions by means of one or more RDF schemas.

RQL adapts the functionality of semistructured/XML query languages to the peculiarities of RDF. However, the innovation of RQL is that it can query at both schema and instance level, exploring the subclass-subproperty hierarchies and the multiple classification of the resources in a transparent (to the user) way. Another advantage is the ability to support generalized path expressions with variables in the class and property names.

Queries in SQPeer are formulated by peers in RQL, according to the RDF/S schema (e.g., defined in a namespace `n1`) of the SON they belong using an appropriate GUI [ACK04]. 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. As already noticed, RQL queries imply both intensional (i.e., schema) and extensional (i.e., data) filtering conditions. A graphical end-user interface[1] may be used to assist RQL query formulation.

RQL provides a *select-from-where* filter to iterate over the contents of a peer base. For instance, in the bottom right part of Figure 3.2 we can see an RQL query **Q** returning in the *select-clause* all the resources binded by the variables `X` and `Y`. The *from-clause* employs two property patterns (i.e., `{X}n1:prop1{Y}` and `{Y}n1:prop2{Z}`), which imply a join on `Y` between the target resources of the property `prop1` and the origin resources of the property `prop2`. Note that no restrictions are considered for the domain and range classes of the two properties, so the end-point classes `C1`, `C2` and `C3` of `prop1` and `prop2` are obtained from their corresponding schema definitions in the namespace `n1`. The *where-clause*, as usual, filters the binded resources according to the provided boolean conditions (e.g., on variable `Z`).

Conjunctive RQL queries not using aggregate functions and nesting can be seen in a rule-based formalism, which is compatible to Datalog; the only difference is that instead of first-order predicates, RQL path expressions are used.

Definition 3.1 *An RQL conjunctive query has the general form:* $ans(\bar{U}) :- \ldots, E_i(\bar{U}_i), \ldots, U_{im} = U_{jn}, \ldots$ *. The rule's head consists of the query's name ans and the tuple* $\bar{U}$ *of the returned variables; the rule's body consists of a conjunction of RQL patterns* $E_i(\bar{U}_i)$ *and equalities* $U_{im} = U_{jn}$ *between variables and/or constants. Each* $\bar{U}_i$ *involves the variables* X_i, $\$C_i$, $@P_i$, Y_i, $\$D_i$ *- where* $@P_i$ *is a property variable,* $\$C_i$ *and* $\$D_i$ *are class variables,* X_i *and* Y_i *are resource variables - or a subset of them.*

Writing the RQL queries in a rule-formalism demands that a normalization phase initially reduces the complex path expressions found in the *from* clause into the general form of RQL queries. For example, for the following RQL query:

[1] See for example the RQL interactive demo at http://139.91.183.30:8999/RQLdemo/

```
select  X
from    {X}prop1.prop2{Z}
where   Z="...";
```

the normalization phase reduces it to the equivalent query:

```
select  X
from    {X}prop1{Y}, {Y}prop2{Z}
where   Z="...";
```

By replacing the constants found in the patterns with variables and adding the corresponding equalities we get:

```
select  X
from    {X}@P1{Y}, {Y}@P2{Z}
where   @P1="prop1" and @P2="prop2" and Z="...";
```

It is easy to derive the rule that is equivalent to the above RQL query:

$ans(X) \ :- \ X \ @P1 \ Y, \ Y \ @P2 \ Z, \ @P1 = \text{“}prop1\text{”}, \ @P2 = \text{“}prop2\text{”}, \ Z = \text{“}...\text{”}$

Table 3.1 summarizes the basic *class* and *property path patterns*, which can be employed in order to formulate complex RQL *query patterns*. These patterns are matched against the RDF/S schema or data graph of a peer base in order to bind graph nodes or edges to the variables introduced in the *from-clause*. The most commonly used RQL patterns essentially specify the fragment of the RDF/S schema graph (i.e., the intensional information), which is actually involved in the retrieval of resources hosted by a peer base.

In other words, class and property patterns specify the part of the SON RDF/S schema, which is involved in the evaluation of a query issued by a peer. The right middle part of Figure 3.2 illustrates the pattern of query **Q**, where `X` and `Y` resource variables are marked with "*" to denote projections. Note that the end-point classes `C1`, `C2` and `C3` of properties `prop1` and `prop2` are obtained from their corresponding definitions in the namespace `n1`.

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 $c\}$
$C{X;$D}	$\{[c, x, d] \mid c,\ d$ are schema classes, d is a subclass of c, x is in the interpretation of class $d\}$
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 $p\}$
{$C} @P {$D}	$\{\ [c, p, d] \mid p$ 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; $C} @P {Y; $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 interpretation of c, y is in the interpretation of d, $[x, y]$ is in the interpretation of $p\}$

Table 3.1: RQL class and property patterns

In the rest of this chapter, we are focusing on conjunctive queries formed only by RQL class and property patterns, as well as projected variables (filtering conditions are ignored). We should also note that SQPeer's query routing and planning algorithms can be also applied to less expressive RDF/S query languages [HBEV04].

3.3.2 RVL Advertisements of Peer Bases

In the context of a P2P system and more specifically of a SON, each peer should be able to advertise its local base contents to other peers. Using these advertisements a peer becomes aware of the bases hosted by others in the system. Advertisements may provide descriptive information about the actual data values (extensional) or the actual schema (intensional) of a peer base.

In order to reason on the intension of both the query requests and peer base contents, SQPeer relies on materialized of virtual RDF/S schema-based advertisements. In the former case, a peer RDF/S base actually holds resource descriptions created according to the employed community schema(s), while in the latter, schema(s) can be populated on demand with data residing in a relational or an XML peer base. In both cases, the RDF/S schema defining a SON may contain numerous classes and properties not necessarily populated in a peer base. Therefore, we need a fine-grained definition of schema-based advertisements.

In the context of the Semantic Web, information consumers of the same information set, pose different requirements to the way they view information, hence necessitating the existence of mechanisms, such as views, to provide tailored access to the information sources. Specifically in the context of an RDF/S-based SON, each peer is required to relate its locally defined schema with the schema of the SON in order to identify itself to the rest of the system. We employ *RVL views* to specify the subset of a community RDF/S schema(s) for which all classes and properties are (in the materialized scenario) or can be (in the virtual scenario) populated in a peer base. Specifically, if an RVL statement uses the same namespace for both asking and returning RDF descriptions, we are dealing with a materialized view. On the other hand, if these namespaces are different, we have a virtual view scenario, where

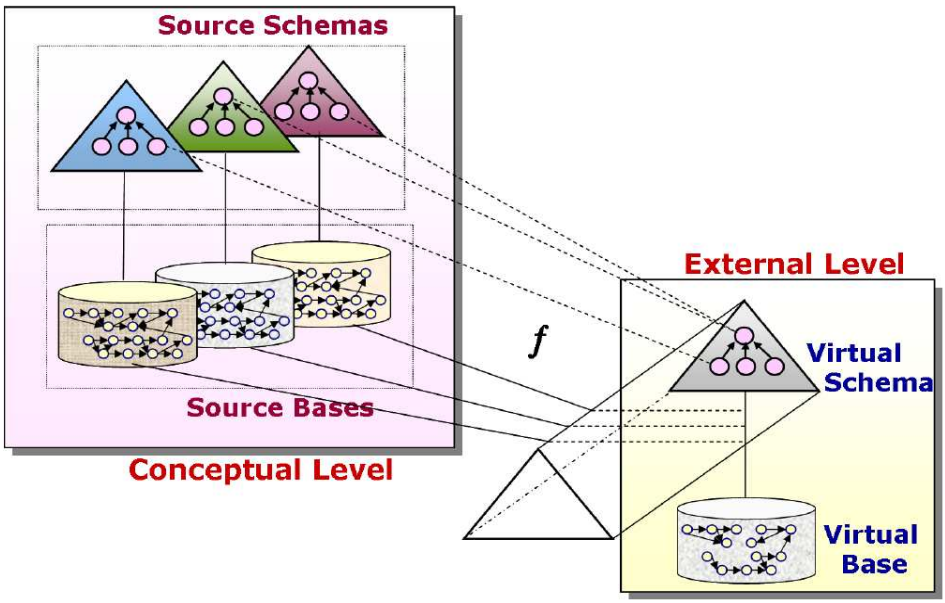


Figure 3.3: RVL Virtual Schemas

the instances are computed from the source base(s) or schema(s) using the RVL program specifying the view. Consequently, the ability of a query language to support mechanisms for defining views over a data set is recognized as an added-value functionality to the expressive power of a query language in traditional database systems, such as relational and object-oriented ones, but also in modern database and P2P applications based on the semistructured data models.

Recognizing the above need, RVL [MTCP03] has been proposed as a view definition mechanism for the Semantic Web. Based on the data model of the RDF/S and by taking advantage of the expressiveness of the RQL query language for RDF/S graphs, this view definition language incorporates the needed functionality and the peculiarities of the underlying data model, in a uniform way. Being the first integrated effort for a view definition language specification, RVL exploits the RQL type system and the abstraction levels of an RDF/S graph to specify two operators, which are able to support all the necessary functionality. This minimality constitutes the most important advantage of RVL.

Figure 3.3 represents the construction of a virtual RVL view, given the source schemas. Generally the definition of an RVL view is in the form:

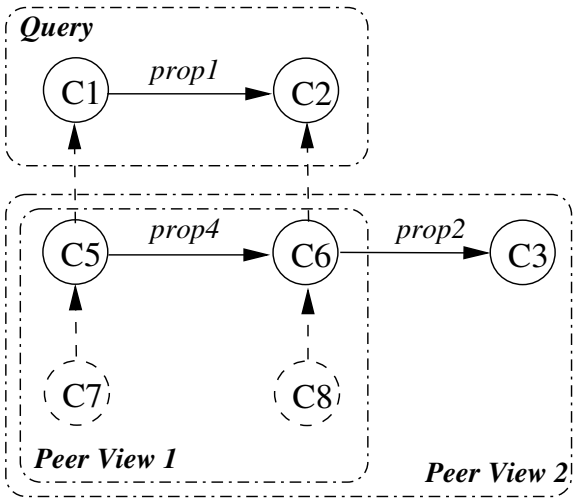


Figure 3.4: Peer view advertisement and subsuming views

```
[VIEW operator
FROM RQL_path_expression
WHERE filtering_conditions
USING NAMESPACE root_schema_namespace]
[ ●●●● ]
USING NAMESPACE root_schema_namespace
CREATE NAMESPACE RVL_view_namespace
```

The *view* clause is used to create constructions of the type defined by the `operator`. The *from* and *where* clause, similarly to the RQL one, is used for the evaluation of the variables defined in the *view* clause and for defining the necessary filtering conditions respectively. Finally, the *using namespace* clause is used to define the prefix of a namespace that is used in the RQL query, while the *create namespace* clause defines the URI of the namespace that is created in the view. Given the definition above, we can see that the functionality of RVL exceeds the one of RQL. RQL is able to query RDF schemas and data, while RVL has the ability to construct virtual schemas and to instantiate them with the use of RQL variable bindings.

The bottom left part of Figure 3.2 illustrates the RVL statement employed to advertise a peer base according to the RDF/S schema identified by the namespace `n1`. This statement populates classes `C5` and `C6` and property `prop4` (in the *view-*

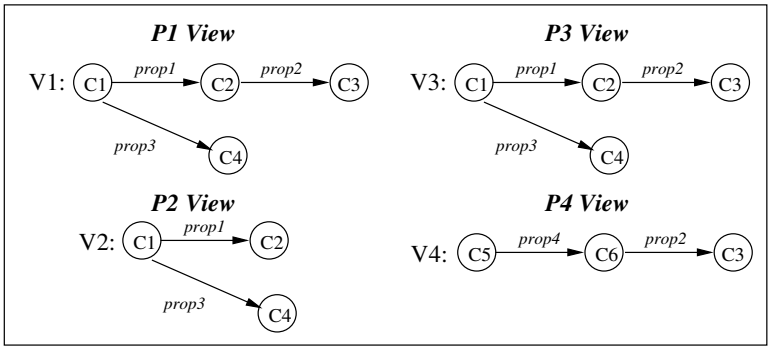


Figure 3.5: Peer view patterns example

clause) with appropriate resources from the peer's base according to the bindings introduced in the *from-clause*. Given the query pattern used in the *from-clause*, `C5` and `C6` are populated with resources that are direct instances of `C5` and `C6` or any of their subsumed classes, i.e., `C7` and `C8`. Actually, a peer advertising its base using this view is capable to answer query patterns involving not only the classes `C5` and `C6` (and `prop4`), but also any of the classes (or properties) that subsume them. For example, Figure 3.4 illustrates a simple query involving classes `C1`, `C2` and property `prop1` subsuming the above peer view 1 (vertical subsumption). The second peer view illustrated in Figure 3.4 extends the previous view with resource instances of class `C3`, which are reachable through `prop2` with instances of `C6`. Peer view 2 can be employed to answer not only a query `{X;C5}prop4{Y;C6}prop2{Z;C3}` but also any of its fragments. As a matter of fact, the results of this query are contained in either `{X;C5}prop4{Y;C6}` or `{Y;C6}prop2{Z;C3}` (horizontal subsumption). So peer view 2 can also contribute to the query `{X;C1}prop1{Y;C2}`.

A more complex example is illustrated in Figure 3.5, comprising the view patterns of four peers. Peer P1 and P3 contain resources related through the properties `prop1`, `prop2` and `prop3`, while peer P4 contains resources related through the properties `prop5` and `prop2`. Peer P2 contains resources related by `prop1` and `prop3`.

As we will see in Section 3.4 the propagation of advertisements depends strongly on the underlying P2P system architecture. Moreover, the lookup service utilized by each architecture, should consider both horizontal and vertical subsumption, as described

above, in order to efficiently identify and return the appropriate data localization information given a specific view.

3.3.3 Query/View Subsumption

We can note the similarity in the intensional representation of peer base advertisements and query requests, respectively, as view or query patterns. This representation provides a uniform logical framework to route and plan queries through distributed peer bases using exclusive intensional information (i.e., schema/typing). However, in order to identify peer views relevant to a given query, we should be able to identify whether a specific view, expressed in RVL, is subsumed ("$\subseteq$") by a given query, expressed in RQL. This problem actually corresponds to the RQL query containment problem, which is further discussed in [Ser05].

Definition 3.2 *A conjunctive RQL query Q1 is contained in a conjunctive RQL query Q2 (Q1 $\subseteq$ Q2) given an RDF description schema DS iff for every description base DB conforming to DS the result of Q1 is contained in that of Q2 ($\forall$ DB Q1(DB) $\subseteq$ Q2(DB)).*

In some cases the containment is obvious. This is the case of simple queries that do not involve complex paths. For example, it is easy to figure out that the following query

```
select  X
from    {X;Painter}paints{Y;Painting};
```

is contained in the following view

```
select  X
from    {X;Painter}creates{Y;Painting};
```

because it is pretty straightforward from the RDF/S semantics of a relevant RDF namespace that when "a painter paints a painting", at the same time he creates a

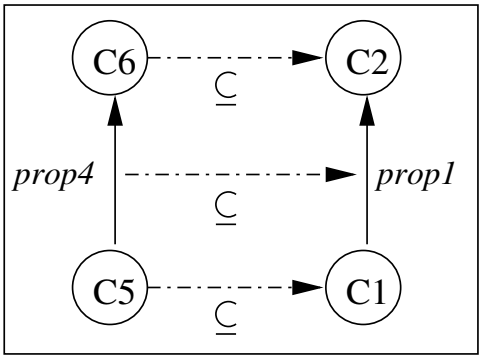


Figure 3.6: A graphical containment example

painting. This is expressed in RDF/S by setting the property `paints` to be subproperty of property `creates`.

Alternatively, in a similar example involving classes and properties of the RDF schema of Figure 3.2, the containment can easily be spotted if the two RQL queries are seen as graphs, where the nodes correspond to classes and the edges to properties. The subject node of the right query subsumes (is a superclass of) the corresponding one of the left query. The same holds for the object nodes. Moreover, the property edge of the right query subsumes (is a superproperty of) the corresponding one of the left query. Thus, all instances satisfying the first query satisfy the second one, too. Figure 3.6 illustrates the containment check between the two graphs.

In SQPeer, as will be further described in the following chapter, queries can be decomposed into their simple patterns, so the above straightforward containment check can be easily utilized. However, deciding containment of more complex queries is not trivial. Moreover, there exist many applications that need dealing with containment in an automated way. Thus, the definition of a sound and complete algorithm for checking RQL query containment is mandatory. The query/view subsumption algorithms presented in Semantic Web Integration Middleware (SWIM [CKK$^+$03]), are used by SQPeer to decide whether a view is subsumed or not by a specific query.

Concluding, we should note that the above representation of both peer advertisements and queries exhibit significant performance advantages. First, the size of the indices, which can be constructed on the intensional peer base advertisements is

considerably smaller than on the extensional ones. Second, by representing in the same way what is queried by a peer and what is contained in a peer base, we can reuse the RQL query/RVL view (sound and complete) subsumption algorithms as presented in this section. Finally, compared to global schema-based advertisements [NWS+03], we expect that the load of queries processed by each peer is smaller, since a peer receives queries that exactly match its base. This also affects the amount of network bandwidth consumed by the P2P system.

3.4 P2P Architectural Alternatives

SQPeer can be used in different P2P architectural settings. Even though the specific P2P architecture affects peers' topology, our proposed query routing and planning algorithms work independently of any particular architectural setting. Recall that the formulation and existence of RDF/S-based SONs lead to minimizing the broadcasting (flooding) in the P2P system, since a query is received and processed only by relevant peers. Before going more on details regarding these issues, we detail the possible roles that peers may play in each setting with respect to their corresponding computing capabilities.

On the one hand, we have *client-peers*, which may frequently join or leave the system. These peers have only the ability to pose RQL queries to the rest of the P2P system. Since these peers usually have limited computing capabilities and they are connected with the rest of the system for short periods of time, they do not participate in the query routing and planning phases. In addition, client-peers do not share their bases or exchange their views, since their brief connection with the system does not allow any kind of useful participation.

On the other hand, we may have *simple-peers* that also act autonomously by joining or leaving the system, but not as frequently as client-peers. Their corresponding bases can be shared by other peers during their connection to the P2P system. When they join the system, simple-peers should either broadcast their RVL views or alternatively request the views of other peers in the system. Thus, a simple-peer connects

physically with the SON(s) it belongs to by identifying peer advertisements similar (in the sense that they are posed under the same RDF namespace) to its own and contacting the peers holding them. These peers eventually will formulate its new neighborhood, thus connecting the simple-peer with the already formulated SONs. Simple-peers have also the ability to pose queries similar to client-peers, but with the extra functionality of executing these queries against their own local bases. Additionally, simple-peers have the ability to coordinate the execution of (sub-)queries on remote peers by undertaking the execution of query (sub-)plans.

Since all peers inside the P2P system are not equal considering their processing or communication capabilities, a small percentage of the peers may play the role of *super-peers*. Super-peers are usually highly-available nodes offering high computing capabilities and each one acts as a centralized server for a subset of simple-peers. Super-peers are mainly responsible for routing (sub-)queries through the system and for managing the cluster of simple-peers which are responsible for. In general, queries received or issued by a simple-peer are first send to its corresponding super-peer, which undertakes the routing process by possibly contacting other relevant super-peers and reply to the simple-peer with the relevant data localization information in order to manage query processing. Each super-peer may accept peers according to their views, thus formulating and efficiently managing SONs. Furthermore, super-peers may play the role of a mediator in a scenario where a query expressed in terms of a globally known schema needs to be reformulated in terms of the schemas employed by the local bases of the simple-peers by using appropriate mapping rules.

In this context, we consider three architectural alternatives distinguished according to the organization of the peers and the distribution of knowledge between them regarding peer base advertisements. The first alternative corresponds to a *hybrid* P2P architecture based on the notion of *super-peers*, the second one is closer to a *structured* P2P architecture based on the notion of *Distributed Hash Tables* (DHTs), while the third one corresponds to an *ad-hoc* P2P architecture based on self-organized SONs.

3.4.1 Hybrid P2P SONs

In a hybrid P2P system [YGM03] [NWS+03] [SMGC04] each peer is connected with at least one super-peer, who is responsible for collecting the views (materialized or virtual) of all its simple-peers. The peers, holding bases described according to the same RDF/S schema, are clustered under the same super-peer. Thus, each peer implicitly knows the views of all its semantic neighbors. In a more sophisticated scenario, super-peers are responsible only for a specific fragment of the RDF/S schema and thus a cluster of super-peers is responsible for the entire schema. Moreover, a hierarchical organization of super-peers can be adopted, where the classes and properties managed at each level are connected through semantic relationships (e.g., subsumption) with the class and properties of the upper and lower levels.

When a simple-peer initially joins the system, it should identify a super-peer and forward its corresponding view (push). All super-peers are aware of each other forming a super-peer backbone, in order to be able to answer queries expressed in terms of different RDF/S schemas (or schema fragments). A simple-peer should be connected to its relevant super-peer according to the peer's intensional information, i.e., the peer view. If the peer base commits to more than one schemas, the simple-peer should consequently be connected to more than one super-peers. By this approach, SONs are formulated with each super-peer being responsible for at least one of them. The exact topology of the P2P system depends on the clustering policy with respect to the number of available super-peers providing the bandwidth and connectivity guarantees of the system.

A client-peer can connect to a simple-peer and send a query request for further processing to the system. The simple-peer forwards the query to the appropriate super-peer according to the schema employed by the query (e.g., by examining the involved namespaces). If this schema is unknown to the simple-peer, it sends the query randomly to one of its known super-peers, which will consecutively discover the appropriate super-peer through the super-peer backbone. In this alternative, we distinguish two separate query evaluation phases: the first corresponds to query

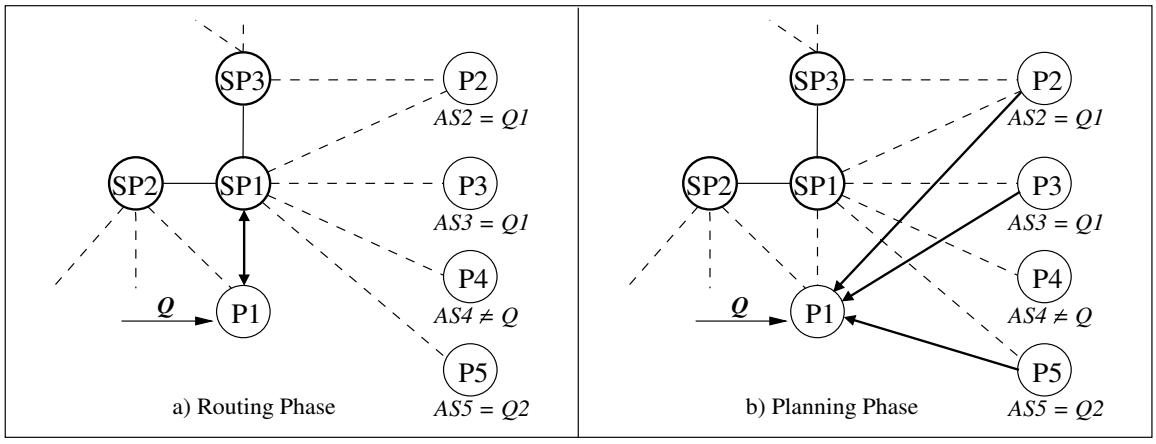


Figure 3.7: SQPeer separated query routing and processing phases in a hybrid P2P system

routing performed exclusively at the super-peers, while the second to query planning, which is usually performed by the simple-peers.

For example, in Figure 3.7, we consider a super-peer backbone containing three super-peers, SP1, SP2 and SP3, and a set of client-peers, P1 to P5. All the simple-peers are connected with at least SP1, since their bases commit to the schema that SP1 is responsible for. When P1 receives a query **Q** consisting of two simple property pattern **Q1** and **Q2**, it initially contacts SP1, which is the super-peer responsible for the SON on which the query is addressed (Figure 3.7a). Since SP1 contains all related peer views, it can also decide on the appropriate fragmentation of the received query pattern according to the view patterns of its simple-peers. Then, SP1 creates an annotated query pattern containing the information that P2 and P3 can answer only the **Q1** pattern, while P5 can answer only the **Q2** pattern. SP1 sends this annotated query pattern to P1 to generate the appropriate query plan. In our example, this plan implies the creation of three channels with P2, P3 and P5 for gathering the appropriate results (Figure 3.7b). The contacted peers send their results back to P1, who joins them locally in order to produce the final answer. Of course, a different execution policy may be selected by peer P1 for either pushing some of the query evaluation to the rest of the peers or for minimizing the response time of the query plan. We should point out that since super-peers contain all the peer views of a SON,

the annotated query pattern for **Q** contains sufficient data localization information for producing not only a correct but also a complete query plan and thus no further routing and processing phases for **Q** are required.

3.4.2 Structured P2P SONs

Alternatively, we can consider a structured P2P architecture [BT03] [CF04] [TP03]. Peers in the same SON are organized according to the topology imposed by the underline structured P2P architecture, e.g., based on Distributed Hash Tables (DHTs) [RFH+01] [SMK+01]. In DHT-based P2P systems, peers are logically placed in the network according to the value of a *hash function* applied to their IP, while a table of pointers to a predefined number of neighbor peers is maintained. Each information resource (e.g., a document or a tuple) is uniquely identified within the system by a key. In order to locate the peers hosting a specific resource, we need to match the hash value of a given key with the hash value of a peer and forward the lookup request to other peers by taking into account the hash table maintained by each contacted peer. In our context, unique keys are assigned to each view pattern and hence peers, whose hash values match those keys, are aware of the peer bases that are populated with data answering a specific schema fragment. An appropriate key assignment and hash function should be used in order neighbor peers to hold successive view patterns with respect to the class/property hierarchy defined in the employed RDF/S schema. This is necessary for optimizing query routing, since successive view patterns are likely to be subsumed by the same query pattern.

Unlike hybrid architecture, in this alternative there is no peer with a global knowledge of all peer views in each SON. The localization information about remote peer views is acquired by the lookup service supported by the system. Specifically, we are interested in identifying peer views that can actually answer an entire (sub-)query pattern given as input.

This implies an *interleaved execution of query routing and planning phases* in several iteration rounds leading to the creation and execution of multiple query plans that when "unioned" offer completeness in the results. This is in contrast to tradi-

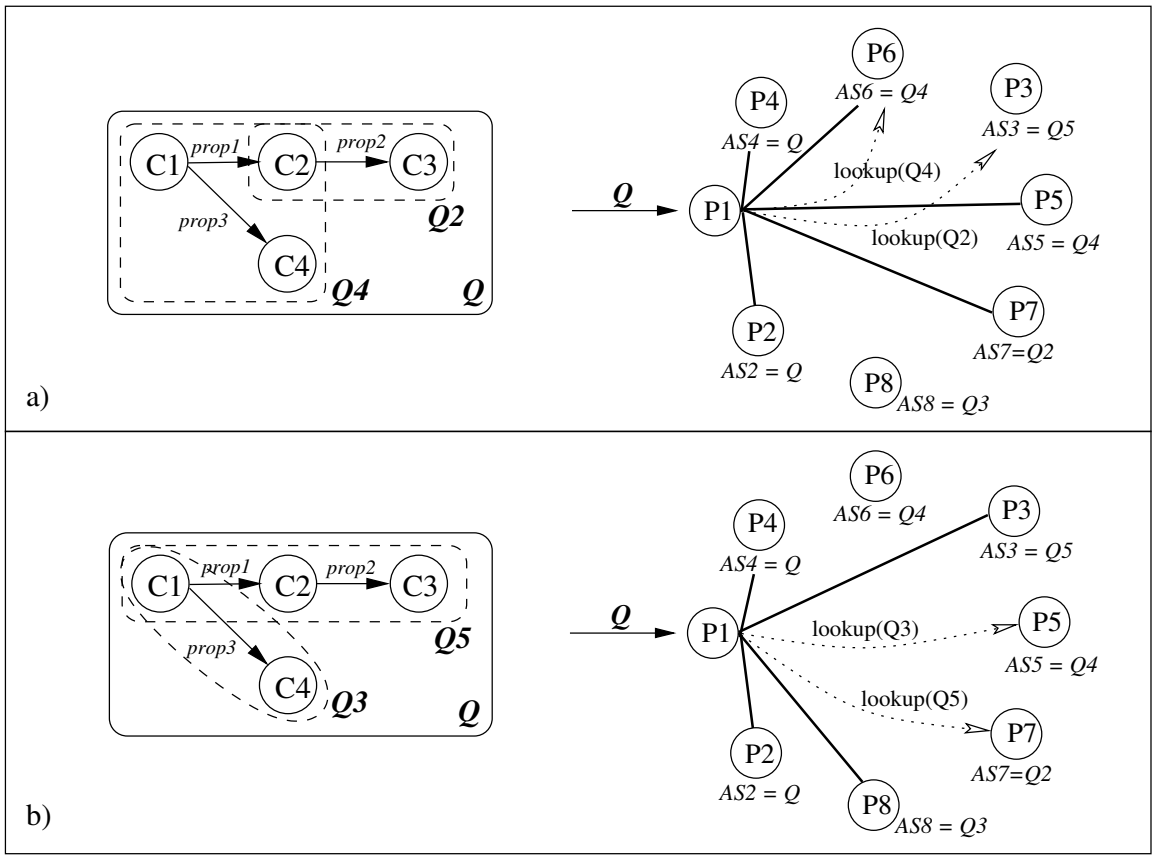


Figure 3.8: SQPeer interleaved query routing and planning mechanism in a structured P2P system for fragmentation of size 2

tional dynamic programming algorithms, where the best possible plan is obtained in terms of execution cost and no guarantees for complete results are given. Note that the generated plans at each round can be actually executed (in contrast to bottom-up dynamic programming algorithms) by the involved peers in order to obtain the first parts of the final query answer. Starting with the initial query pattern, at each round, smaller fragments are considered in order to find the relevant peer bases (routing phase) that can actually answer them (planning phase). In this context, the interleaved query processing terminates when the initial query is decomposed into its basic class and property patterns. It should be also stressed that SQPeer interleaved query routing and planning favors intra-peer joins, since each query fragment is looked up as a whole and only peers that can fully answer it are contacted.

For example, in Figure 3.8 we consider that peers P1 to P8 are connected in a

structured P2P system. When P1 receives the query **Q**, it launches the interleaved query routing and planning. At round 1, P1 issues a lookup request for the entire query pattern **Q**, and annotates **Q** with peers P2 and P4. In this initial round, plan `Plan 1 =` $Q@P2 \bigcup Q@P4$ is created and executed. At round 2, the fragmentor is called with *#joins* equal to 1. The two possible fragmentations of query **Q** are depicted in Figures 3.8a and b. First, peers P6 and P3 are contacted through the lookup service, since they contain the list of peer bases answering query fragment patterns **Q4** and **Q2** respectively (seen in the left part of Figure 3.8a). P6 returns the list of peers P2, P4, P5 and P6, while P3 returns peers P2, P3, P4 and P7. For this fragmentation, the query plan `Plan 2 =` $\bigcup(\bowtie(Q4@P2, Q2@P3), \bowtie(Q4@P2, Q2@P4), \ldots,$ $\bowtie(Q4@P6, Q2@P4), \bowtie(Q4@P6, Q2@P7))$ is created and executed by deploying the necessary channels between the involved peers (see right part of Figure 3.8a). It is worth noticing that the generated plans at each round do not include redundant computations already considered in a previous round. For example `Plan 2` produced in round 2 excludes the query fragment plan $\bowtie(Q4@P2, Q2@P2)$ generated in round 1. Next, peers P5 and P7 are contacted through the lookup service, since they contain the list of peer bases answering query patterns **Q3** and **Q5** respectively (seen in the left part of Figure 3.8b). P5 returns the list of peers P2, P4, P5, P6 and P8, while P7 returns peers P2, P3 and P4 and the query plan `Plan 3 =` $\bigcup(\bowtie(Q3@P2, Q4@P3),$ $\bowtie(Q3@P2, Q4@P4), \ldots, \bowtie(Q3@P8, Q4@P3), \bowtie(Q3@P8, Q4@P4))$ is created and executed (see right part of Figure 3.8b). Again, `Plan 3` is disjoint with the plans already generated. At the last round (*#joins* equals to 2), we consider all basic property and class patterns of query **Q** and run one more time the routing and planning algorithms to produce query plans returning the remaining parts of the final answer.

3.4.3 Ad-hoc P2P SONs

Finally, in an ad-hoc P2P architecture when a peer first joins the system, it becomes aware only of its physically close neighbors. The peer should identify, by sending appropriate requests, at least one other related peer for each of its RDF/S community schemas. The related peers identified in this way, form the semantic

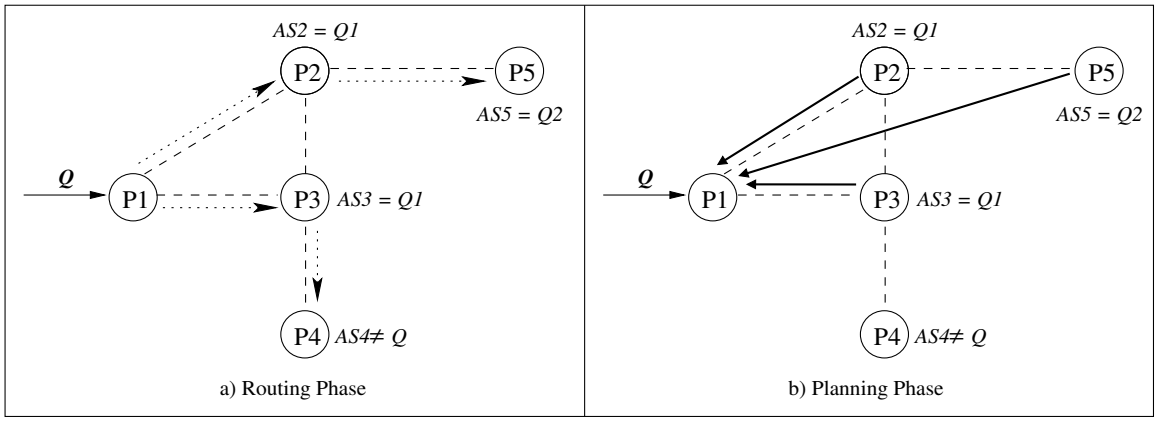


Figure 3.9: SQPeer query processing mechanism in an ad-hoc P2P system

neighborhood of the peer. In the next step, the peer explicitly requests the peer views of its neighbor peers (pull). Then, when a peer receives a relevant query, it applies locally the query routing and planning phase and creates a query plan. When the peer receives a query, whose schema is unknown or which cannot be answered by the semantic neighbors of this peer, it could request the peer views of a 2-depth, 3-depth, etc. neighborhood, until a relevant peer is found and thus constructing progressively *self-adaptive SONs*.

Unlike super-peers, in the ad-hoc scenario there are no real guarantees that a peer can actually generate a complete query plan. The query processing algorithm produces a query plan according to its local knowledge about relevant peers and therefore the query plan may contain "holes", i.e., subplans with incomplete data localization information (denoted by **?**). In order to discover the answering peers and fill the holes (except from flooding the network with routing requests), the query plan can be forwarded to other peers which are known to be able to answer at least a part of the initial query plan. By establishing appropriate channels, the peers receiving a partial plan can in turn interleave query planning and routing using their locally stored peer advertisements. The first peer that is able to fill all the "holes" and generate a complete query plan, holds also the responsibility of executing it and send the results back to the root peer through the already deployed channels.

Figure 3.9 depicts an example where peers P1 to P5 are connected in a self adaptive

SON. P1 is aware of the views of its neighbor peers, i.e., P2 and P3. When P1 receives the query **Q**, it uses the already gathered peer advertisements in order to apply locally the routing phase. Since P2 and P3 can answer the **Q1** part of **Q** and since no known neighbor peer can answer the **Q2** part, P1 creates the query plan `Plan 1 =` $\bigcup(\bowtie(Q1@P2, Q2@?), \bowtie(Q1@P3, Q2@?))$. P1 may then decide to request the peer views of a 2-depth neighborhood (Figure 3.9a). According to the received data localization information, P4 cannot contribute to the query plan, while P5 can fill the holes of the already produced query plan, since it can answer **Q2**. P1 can then execute the query plan `Plan 2 =` $\bigcup(\bowtie(Q1@P2, Q2@P5), \bowtie(Q1@P3, Q2@P5))$ by deploying the necessary channels between the involved peers (Figure 3.9b). The appropriate subplans are sent to peers P2, P3 and P5 and the results are returned to P1, where there are "joined" and "unioned" locally producing the final complete answer.

3.5 Comparison of the Architectural Alternatives

The above architectural alternatives exhibit different behaviors on routing and planning a query.

In the structured architecture, we have self-adaptive SONs, while in the super-peer architecture SONs are created in a more static way, since each super-peer is responsible for the creation and further management of its assigned SON(s). On the contrary, in the ad-hoc scenario SONs are formulated by the self-organization of the peers without any restriction on the structure (as is the case in the structured architecture).

It should be stressed that while in the structured and ad-hoc architecture, peers handle both the query routing and planning load, super-peers are primarily responsible for routing and simple-peers for planning queries in two distinct phases.

Additionally, super-peers contain a global knowledge of the simple-peer views in a SON, while in the structured and ad-hoc alternative this knowledge is distributed, since each peer is aware only of a small number of peer advertisements in the SON.

However, a structured P2P system can offer an efficient lookup service for obtaining locally the relevant peer views, while in the ad-hoc system peers should either request or handle partial data localization information with appropriate techniques (i.e., partial query plans). This makes super-peers capable of offering complete data localization information concerning their SONs, while in the structured and ad-hoc scenario this information should be requested via one or several lookup phases. This makes these two architectures good candidates for applying the interleaved query routing and planning scenario.

Chapter 4

Query Processing in SQPeer

Query processing in SQPeer is responsible for generating *distributed query plans* guiding the execution of the query in the P2P system. The creation of the query plans is based on an annotated query pattern, which in turn is formulated by considering the relevant peer views gathered during the routing phase. The produced query plan specifies precisely how the query is going to be deployed and executed at the selected peers contributing to the final answer. As already discussed in Section 2, query plans are represented as trees, where the nodes correspond to the operators of the query, while the leaves represent the (complex or simple) path patterns extracted from the query that need to be combined to produce the final result. We should point out that these path patterns actually correspond to peer views, since the routing phase identifies the peers that answer these patterns as a whole. Additionally, the query plan specifies the query operators that each peer should process in order to combine intermediate results into the final answer.

Query processing can be broken into several phases based on the DDMS architecture already presented in Section 2. Initially, the query is parsed and the generated query patterns are handled by the routing algorithm. A fragmentor is responsible for breaking the query into distinguished fragments and for each one the lookup service is utilized to find relevant data localization information. Then, a data localization algorithm produces an annotated query pattern by annotating each fragment of the query with the peers that can handle it. The produced pattern is then send to

the algebraic translation algorithm, where an appropriate query plan is produced by translating the pattern into the SQPeer query algebra. Next, this query plan is passed to an optimizer in order to apply heuristic and/or cost-based optimization techniques taking into account inter- and intra-peer query processing and communication cost. Finally, the optimized plan is sent to the execution engine responsible for forwarding the already distinguished subplans to the appropriate peers and monitoring their evaluation. Peer communication is achieved by the use of appropriate communication channels that additionally provide the means for query plan adaptation during query execution in case of run-time failures.

4.1 Core Algebra and Equivalences

In this section the core algebra [KAC+02] used for the representation of the RQL queries as query plans is described.

4.1.1 The Operators

The core of the algebra we are using consists of common operators of object-oriented query algebras that are defined on sets of ordered tuples. More precisely, these operators are: *union* ($\cup$), *intersection* ($\cap$), $difference$ ($\setminus$), *selection* (σ), *join* ($\bowtie$), *semi-join* ($\ltimes \rtimes$), *d-join*($< . >$) and *mapping* (χ). The set operators as well as the selection, join and semi-join operators are well-known from the relational context. The *d-join* operator is used for performing a join between two sets, the second of which depends on the first. Finally, the *mapping* operator is used to transform a set of objects to a set of tuples, so as to preserve associativity of join operations, also adding a name to the generated tuple, since tuple attributes are named. Moreover, this operator is used to apply a function to every member of a given set.

4.1.2 Translating RQL queries to the core algebra

Using the above operators we can express all different kinds of (possibly nested) RQL queries. A translation phase is introduced, in which appropriate definitions for

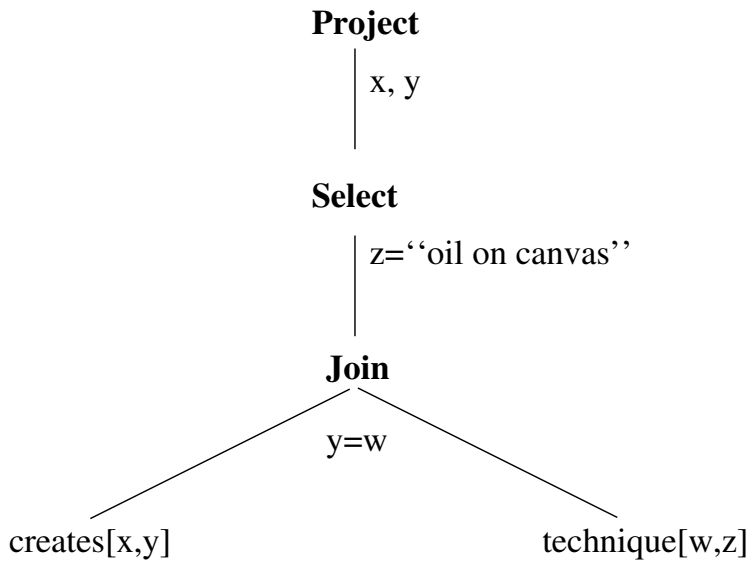


Figure 4.1: Evaluation plan of Query **Q1**

RQL operations are collected in an extra *define* clause. We illustrate this translation using a query **Q1** as an example:

Q1: Find the resources that have `created` something, `using_technique` "oil on canvas", as well as the creations themselves

`select` X, Y
`from` {X}creates{Y}.using_technique{Z}
`where` Z = "oil on canvas"

, which is translated into the query:

`select` X, Y
`from` creates A, using_technique B
`define` X = A[0], Y = A[1], W = B[0], Z = B[1]
`where` Z = "oil on canvas" and Y = W

This query involves two scans on the extents of the properties `creates` and `using_technique`. These scans return pairs (sequences) of values and we need to define appropriate sequence accessor operations (as shown in the *define* clause) on

which variables at the extremities of these paths are range-restricted. The corresponding query evaluation plan is shown in Figure 4.1.

More specifically, for translating **Q1** and eventually represent it as a query evaluation plan, the following operations for each clause of the query are performed:

- For the two path components in the `from` clause, we construct two *scan* operations on the extent of the corresponding properties (called A for *creates* and B for *using_technique*). For each of these properties we define a constant operation (serving as a reference to the corresponding collection), which serves as the domain of the scan variable.

- The scan variables A, B return sequence values with two elements. In order to obtain *from* and *to* values of a property from such scans, we create two sequence access operations (denoted by [0], [1]). Then, we create two *name* operations, in order to assign the names of the variables in the query (i.e., X, Y) to the operation that valuates these variables (i.e., the two sequence access operations).

- The composition of the two path components implies a join condition between their extremities. In this query, the appropriate condition is equality between the *to-values* of *creates* and *from-values* of *using_technique*.

- Then we process the conditions in the `where` clause; in **Q1** there is an equality condition between variable Z and a constant operation that represents the string "oil on canvas". The constructed operation is and-ed with the join condition that is implied by the path composition. As a result, the *root* selection operation of **Q1** is the *and* operation.

- Finally, the comma-separated list of projections in the `select` clause is translated into a sequence constructor operation, since RDF sequence values are used to represent tuples. Note that for queries with only one projection we do not need a sequence constructor; in this case, the operation (variable or other)

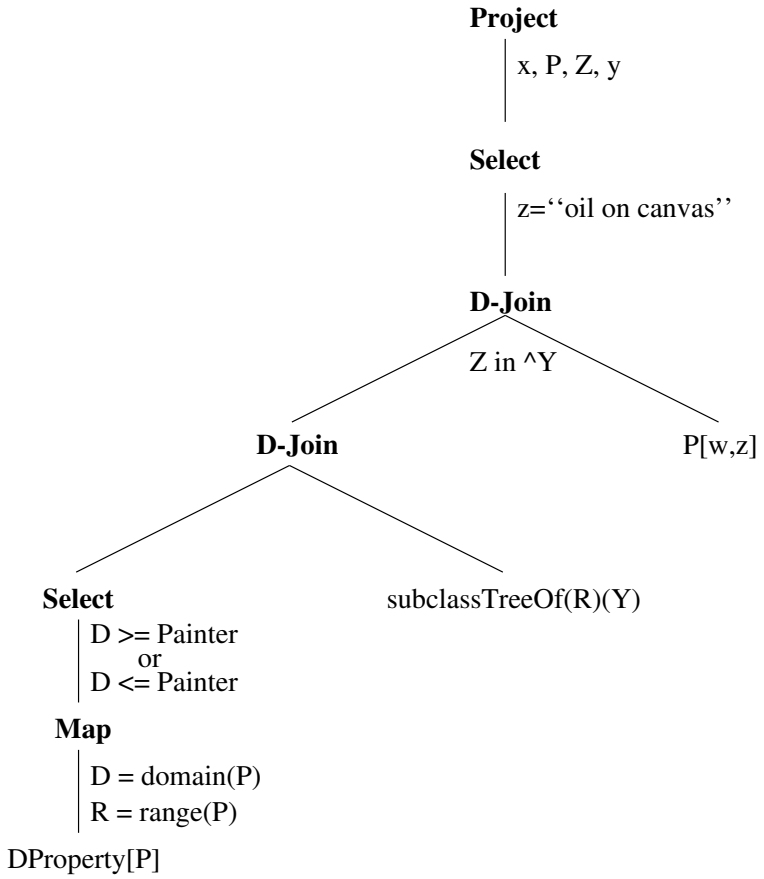


Figure 4.2: Evaluation plan of Query **Q2**

that appears in the `select` clause of the query is considered as the projection operation of the query.

Things are more complex when the path element is not a ground property or class from a given schema, but instead are variables, either already bound from another path or using RQL's shortcut notation for class ($) or property (@) variables. Consider, for example, the following query **Q2**:

Q2: Find all properties and their range classes that can be applied on class `Painter`

```
select  X, @P, Z, $Y
from    {X;Painter}@P{Z;$Y}
```

This query can be rewritten, using the *define* clause, in the following intermediate form:

`select` X, P, Z, Y

`from` DProperty P, P Q, subclasstreeof(R) Y

`define` X = Q[0], Y = Q[1], D = domain(P), R = range(P)

`where` D = Painter and X in Painter and Z in ^Y

This translation involves the application of two functions (*domain*, *range*) on a set (*DProperty*). *DProperty* corresponds to an RQL built-in metaclass used in order to be able to retrieve only data properties, that are defined at schema level (i.e., relations between resources or attributes of resources). As we explained previously, the above application of the functions can be expressed using the *mapping* operator (χ). Moreover, the scan on each property name returned by the first scan implies a *d-join*, since the second scan depends on the first one. The corresponding query evaluation plan is shown in Figure 4.2.

For the query plans considered in SQPeer, we will focus on the *join* and the *union* operators. We should point out that in order to perform query routing in SQPeer, we require information given from the `from` and `select` clauses. However, `where` clause provides information necessary in the planning phase. Since the *join* and *union* operators consume the majority of the processing time, their evaluation is the main concern of most of the distributed database and P2P query plan optimizers.

Using the above presented query algebra, we can proceed in describing the fragmentation and algebraization phase utilized by SQPeer, after we first introduce several possible query plan equivalences and heuristics.

4.1.3 Query Plan Equivalences

An issue concerning the creation of the query plans is the placement of the unions in the produced query plan tree. Usually, unions exist at the bottom of the plan tree combining all the results for each resource answered by several peer bases. These results are later combined by the joins and produce the final answer. We can push unions to the top and consequently push joins closer to the leaves. This makes possible (a) to evaluate an entire join at a single peer (intra-peer processing) when its view

is subsumed by the query fragment, and (b) to parallelize the execution of the union in several peers. The latter can be achieved by allowing for example each fragment plan (consisting of only joins) to be autonomously processed and executed by different peers. The former suggests applying the following algebraic equivalence as long as the number of inter-peer (i.e., between different peers) joins in the equivalent query plan is less than the intra-peer one. This heuristic comes in accordance to best effort query processing strategies for P2P systems introduced in [RSWB05]. Moreover, promoting intra-peer processing exploits the benefits of query shipping as discussed in [FJK96].

Algebraic equivalence: Distribution of joins and unions

Given a subquery $\bowtie(\bigcup(Q_{11},\ldots,Q_{1n}),\bigcup(Q_{21},\ldots,Q_{2m}))$ *rewrite it into* $\bigcup(\bowtie(Q_{11},Q_{21}),\bowtie(Q_{11},Q_{22}),\ldots,\bowtie(Q_{1n},Q_{2m}))$.

In order for the produced query plans to identify intra-peer (i.e., between the same peer) processing and thus consider the fact that one peer can answer more than one successive patterns (unless more sophisticated fragmentation is considered as discussed in the following section), the following heuristics are introduced:

Heuristic 1:

Given a subquery $\bowtie(Q_1@P_i,\ldots,Q_n@P_i)$ *rewrite it into* $Q@P_i$, *where* $Q = Q_1 \bowtie \ldots \bowtie Q_n$.

Heuristic 2:

Given a subquery $\bowtie(\bowtie(QP,Q_1@P_i),Q_2@P_i)$ *rewrite it into* $\bowtie(QP,Q@P_i)$, *where* $Q = Q_1 \bowtie Q_2$ *and* QP *any non-empty query subplan.*

These two heuristics actually identify parts of the query plan that can be answered by the same peer and group them in order to be executed as a whole. It should be stressed that in order to identify all appropriate intra-peer joins, the two heuristics should be applied recursively, until no further change in the query plan is possible. The query fragmentor practically embodies these heuristics by considering complex patterns as a single fragment as will be presented in the next section.

4.2 Query Fragmentation and Algebraization

4.2.1 Query Fragmentor

Query processing involves a query fragmentation phase, where the query is split into distinguished fragments, which will eventually be executed as a whole by the same peer. More precisely, the fragmentor splits the query based on the existing data distribution, since the fragments should exploit as much as possible the answering capabilities of the involved peers. By executing more complex fragments at a single peer, not only we consider horizontal data distribution but additionally the cost of the query plan is minimized and results that are more relevant to the original query are returned as fast as possible. A description of how the fragmentation phase works is given below.

A *fragmentor* receives as input a complex query pattern in order to produce a set of simpler query patterns, according to the number of joins (input parameter *#joins*) between the resulting fragments, which are required to answer the original pattern. Recall that a query pattern is always a fragment graph of the underlying RDF/S schema graph.

Each *fragment* (or fragment pattern) consists of a connected set of property patterns of the initial query pattern. A set of fragments that when combined produce the complete initial query pattern is considered as a *fragmentation* of this query pattern. Since there are multiple ways to break a query into a specific number of fragments, the fragmentor produces a set of all the possible fragmentations given a specific *#joins*. In the simplest case *#joins* equals to 0, so the whole query schema subgraph is returned. The variable value may increase until no further fragmentation of the query is possible, i.e., the total number of simple property patterns contained in the input query pattern is produced. At this case, the query or view pattern is decomposed into its basic class and property patterns and all the joins involved in it are considered.

Since the fragmentation phase affects query processing by restricting the type of the created query plans, the input parameter *#joins* is determined by the optimization techniques employed by the query processor. More information concerning the

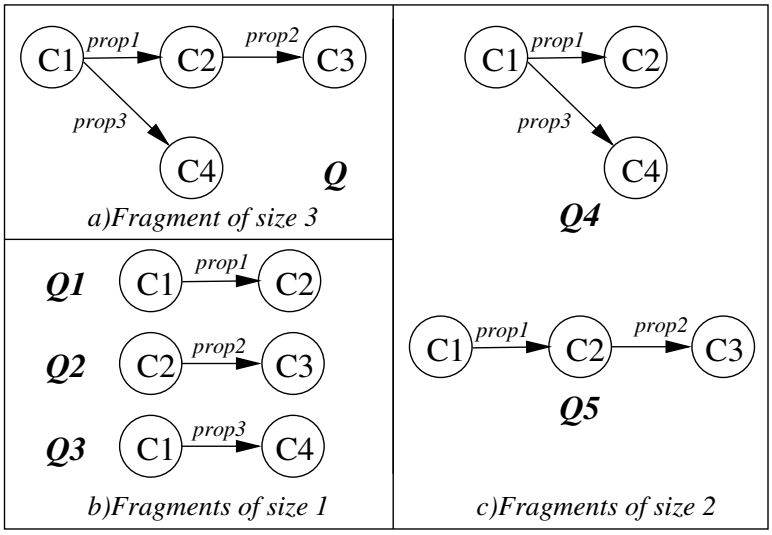


Figure 4.3: Fragments for the query pattern **Q**

execution of the fragmentor and the choice for its input parameter is discussed in Section 4.4 in conjunction with the proposed query processing and execution alternatives.

An example of all possible fragments of a query pattern **Q**, whose size equals to 3, is given in Figure 4.3. Easily we note that only one fragment of size 3 is possible that is the whole query pattern. Respectively the number of fragments of size 1, i.e., the simple property patterns **Q1**, **Q2** and **Q3** that consist the query, equals to the size of the initial query pattern, which in our example is 3. Finally, only two fragments of size 2 are possible, i.e., **Q4** and **Q5**. Note that the number of the intermediate produced fragments is governed by the structure (i.e., linear, tree or graph) and naturally the size of the input query. The fragmentations returned by the algorithm are all the possible combinations of the above fragments that produce a correct result for the considered query. In our example, the set of all the possible fragmentations of the query **Q** is the following {{**Q**}, {**Q4**, **Q1**}, {**Q5**, **Q3**}, {**Q1**, **Q2**, **Q3**}}.

4.2.2 Query Routing and Data Localization Algorithm

Query routing is responsible for finding the relevant to a query peers (or more precisely their advertisements) by taking into account data distribution (vertical,

horizontal and mixed) of peer bases committing to a SON RDF/S schema. In particular, query routing is responsible for the data localization and the lookup phases. The main role of *data localization* is to translate the operations on relations in order to bear on local data that may be stored in remote or local peer bases. This is done by using data distribution information obtained by the lookup service. The *lookup service* is responsible to identify and return the relevant peer advertisements with the possible help of a peer view index embedded in the P2P system.

In SQPeer, data localization information takes the form of peer views (also referred to as peer advertisements) that are used for efficiently and intensionally advertising their peer contents (see Section 3.3.2). Peer advertisements can be either collected at specific peers that act as central routers (each one handling specific type of peer views) or distributed at a number of peers all over the P2P system. In the centralized alternative, where the views are centrally stored, all peers are required to publish their bases to the appropriate central registry. In the distributed alternative, each peer contains a subset of the peer base advertisements that need to be combined in order to offer the complete data localization information. The first alternative requires from the lookup phase to contact that central registry and simply return the appropriate data localization information. However, in the distributed scenario a more sophisticated lookup service is necessary to efficiently contact the appropriate peers and gather the advertisements that are relevant to the query searched. Nevertheless, in both alternatives the lookup phase should consider subsumption issues between the query and the corresponding views, as discussed in Sections 3.3.2 and 3.3.3.

Prior to the initiation of the routing phase, a *fragmentor* is employed to break a complex query pattern given as input into more simple ones, as discussed in the previous subsection. For each fragment pattern produced, the data localization phase is executed and all the available views are checked for identifying those that can answer it. Such a data localization algorithm for SQPeer is discussed below.

The *data localization algorithm* takes as input a query pattern and returns a query pattern annotated with information about the peers that can efficiently evaluate it.

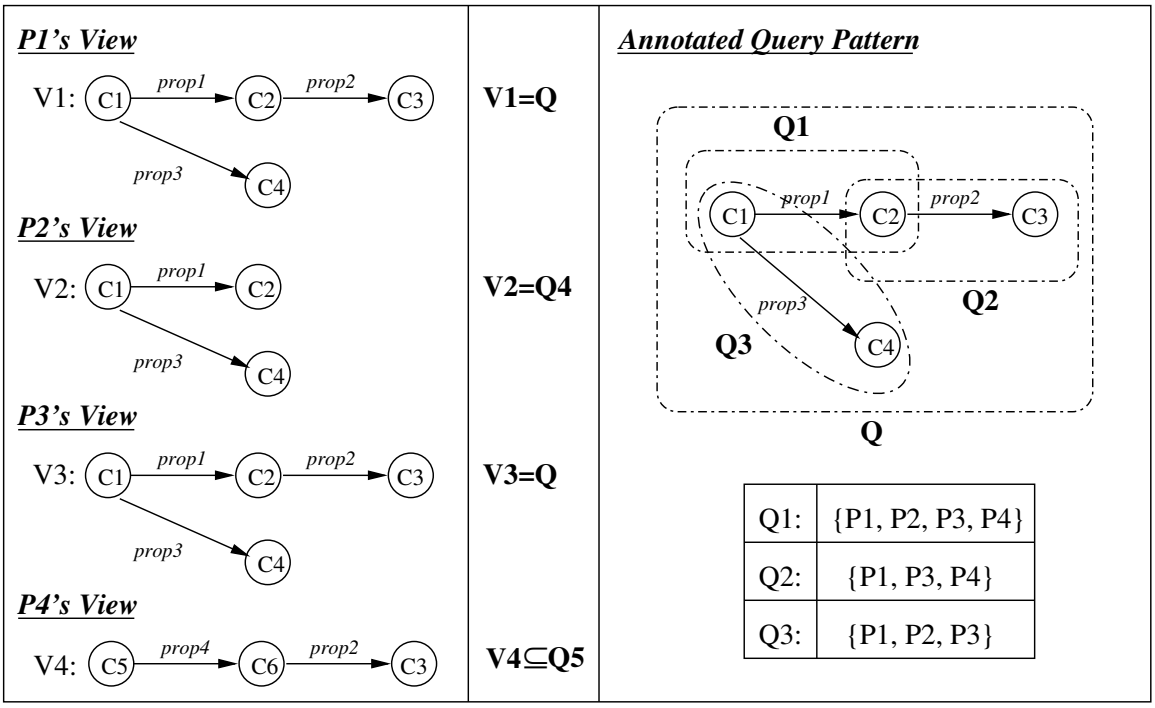


Figure 4.4: An annotated RQL query pattern

A lookup service (i.e., function *lookup*), which strongly depends on the underlying P2P topology, is employed to find peer views relevant to the input pattern. The query/view subsumption algorithms of [CKK+03] are employed to determine whether a query can be answered by a peer view. More precisely, function *isSubsumed* checks whether every class/property in the query is present or subsumes a class/property of the view. A pseudocode depicting how data localization works is given below:

DataLocalization(QP)

1 $QP' \leftarrow$ *construct an empty annotated query pattern for* QP
2 $VP \leftarrow lookup(QP)$
3 **for** *all view patterns* $VP_i \in VP$, $i \leftarrow 1$ **to** n
4 **do if** $isSubsumed(VP_i, QP)$
5 **then** *annotate* QP' *with peer* P *responsible for* VP_i
6 **return** QP'

Figure 4.4 illustrates an example of how SQPeer data localization algorithm works given the RQL query **Q** composed by three property patterns, namely **Q1**, **Q2** and

Q3, as well as the views of four peers. The middle part of the figure depicts how each fragment pattern matches one of the four peer views. The variable *#joins* in our example is set to 2 so the three simple property patterns of query **Q** are checked. A more sophisticated example with a different fragmentation will be presented in Section 4.2.3. P1's and P3's views consist of the property patterns **Q1**, **Q2** and **Q3**, so all three patterns are annotated with P1 and P3 respectively. P2's view consists of patterns **Q1** and **Q3**, so they are both annotated with P2. Finally, P4's view is subsumed by patterns **Q1** and **Q2**, since `prop4` is a subproperty of `prop1`, therefore **Q1** and **Q2** are annotated with P4. In the right part of Figure 4.4 we can see the annotated query pattern returned by the SQPeer data localization algorithm, when applied to the RQL query and RVL views of our example.

In the data localization algorithm presented we have considered the fragmentation of the query into its minimal query patterns (i.e., simple property patterns) by setting the number of the *#joins* variable to be equal to the total number of joins involved in the input query pattern. However, we should note that for a given *#joins* value, there are more than one ways to break a given query pattern, i.e., there are more than one fragmentations possible. Exceptions are the two extreme values for *#joins*, i.e., 0 and the total number of joins of the query pattern, with each one producing only one possible fragmentation. For example for a query of size 4 and for *#joins* equal to 1, we can break the query into two different sets. The first involving queries of size 2, which are joined together, and the second involving queries of size 3 and 1 also joined with each other. Moreover, same type of fragmentations may also differ, for example the fragmentations {**Q4**, **Q2**} and {**Q5**, **Q3**} of query **Q** in Figure 4.4, so each will produce a different annotated query pattern. In this sense, the output of the routing phase is a set of annotated query patterns, one for each fragmentation of the query. This will become more clear in Section 4.4, where the interleaved query routing and planning phase is discussed.

4.2.3 Query Planning and Algebraic Translation Algorithm

Query planning in SQPeer is responsible for generating a distributed query plan according to the localization information returned by the routing algorithm. The

first step towards this end, is to provide an algebraic translation of the RQL query patterns annotated with data localization information.

The *algebraic translation algorithm* relies on the object algebra of RQL (see also Section 4.1). Initially, the annotated query pattern (i.e., a schema fragment) is traversed and for each subfragment, already distinguished by the fragmentor, the annotations with relevant peers are extracted. If more than one peers can answer the same fragment, the results from each such peer base are "unioned" (horizontal distribution). As the query pattern is traversed, the results obtained for different fragments that are connected at a specific domain or range class are "joined" (vertical distribution). The final query plan is created when all subquery fragments are translated.

The generated query plan reflects the data distribution of the P2P system. Since the input annotated graph is created according to a specific fragmentation policy adopted by the routing algorithm, the produced query plan considers each fragment as a single resource answered by the peers in the corresponding annotation set. As a result only inter-peer joins are explicitly constructed by the planning algorithm. All intra-peer joins are already considered by each fragment, hence the peer receiving the specific fragment will have to undertake its execution by locally processing all the involved joins.

A pseudocode depicting how the algebraic translation algorithm works is presented below:

AlgebraicTranslation(AQ', PP)

1 $QP \leftarrow$ NIL
2 $P \leftarrow \{P_1, \ldots, P_n\}$, *set of peers obtained from the annotation of* PP *in* AQ'
3 **if** $P \leftarrow$ NIL
4 **then** $QP \leftarrow PP@?$
5 **else for** *all peers* $P_x \epsilon P$
6 **do** $QP \leftarrow QP \bigcup PP@P_x$ $-$ *Horizontal Distribution*$-$
7 **for** *all subquery patterns* $PP_j \epsilon$ *children*(PP)
8 **do** $TP_j \leftarrow AlgebraicTranslation(PP_j, AQ')$
9 $QP \leftarrow \bowtie_{Cp} (QP, TP_1, \ldots, TP_n)$ $-$ *Vertical Distribution*$-$

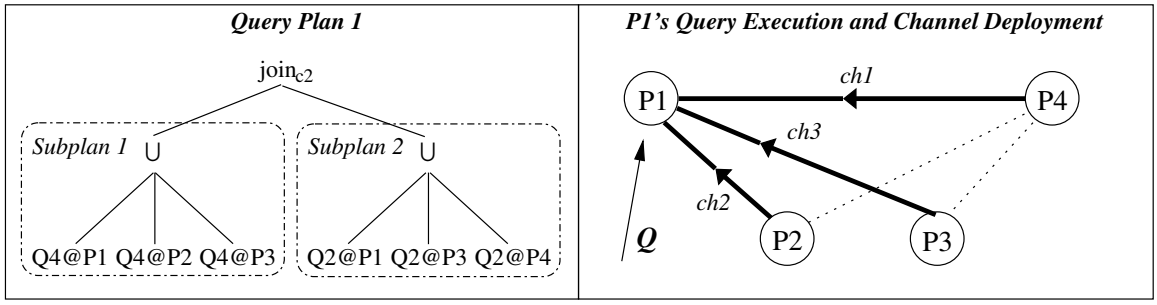


Figure 4.5: Query plan generation and channel deployment in SQPeer

Figure 4.5 illustrates how the RQL query **Q** can be processed given the four peer views (as seen in Figure 4.4), the *#joins* variable equals to 1 and concentrating on the fragmentation policy of breaking the query into its two subqueries **Q4** and **Q2**. In this example, we assume that P1 has already carried out the routing phase by executing the data localization algorithm in order to generate the annotated query pattern for the respective fragmentation. The algebraic translation algorithm, also running at P1, receives as input the annotated query pattern and the root pattern, i.e., **Q4**. It initially translates the root pattern into the algebraic `Subplan 1` depicted in Figure 4.5 (i.e., P1, P2 and P3 can effectively answer the subquery). The partial results obtained by these peers should be "unioned" (horizontal distribution at line 6). By checking all the children patterns of the root, we recursively traverse the input annotated query pattern and translate its constituent fragment plans. For instance, when **Q2** is visited as the first (and only) child of **Q4** the algebraic `Subplan 2` is created (i.e., P1, P3 and P4 can effectively answer the subquery). Then, the returned query plan concerning **Q2** is "joined" (vertical distribution at line 9) with `Subplan 1`, thus producing the plan illustrated in Figure 4.5 (i.e., no more fragments of the initial annotated query pattern **Q** need to be traversed).

The algebraic translation algorithm should also consider in the same manner the fragmentation of the query into its two subqueries **Q5** and **Q3** producing a similar plan. The two produced plans should also be "unioned", thus finalizing the planning phase for the given value of *#joins*. A more elaborate example is presented in Section 4.4.

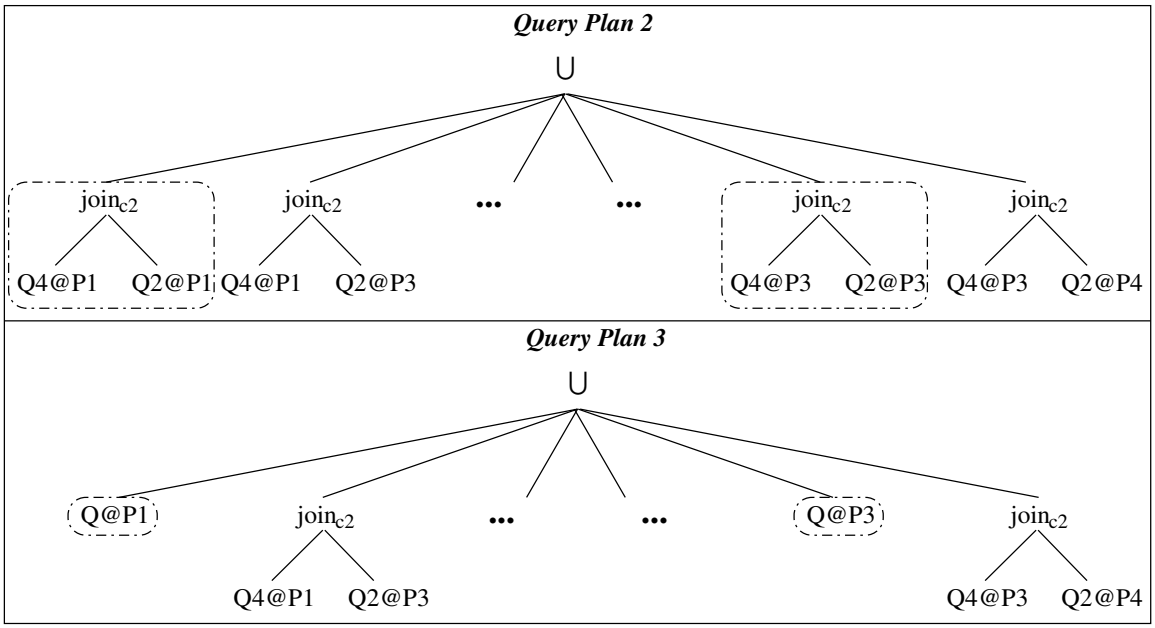


Figure 4.6: Optimizing query plans by applying algebraic equivalences and heuristics

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

Applying the algebraic equivalence introduced in Section 4.1.3, the query plan of Figure 4.5 is transformed into the equivalent `query plan 2` of Figure 4.6. This plan decreases the data transferring cost between the peers due to smaller intermediate results and additionally offers the ability to evaluate this plan in a pipeline way, since each leaf of the union operator can be optimized and executed in parallel. However, this comes with the additional effect of increasing the processing time, since the number of considered plans increases.

Moreover, one can easily observe that `query plan 2` does not take into account the fact that one peer (e.g., P4) can answer more than one successive patterns, unless more sophisticated fragmentation is considered by the data localization algorithm. By applying the two heuristics of Section 4.1.3 on `query plan 2`, `query plan 3` of Figure 4.6 is produced. This allows fragment plans that translate the whole query pattern **Q** to be directly sent for execution to the relevant peers, i.e., joins between

subqueries **Q4** and **Q2** are executed by peers P1 and P3 respectively.

In this context, a question that naturally arises is how SQPeer query planner behaves in cases where there is no sufficient data localization information to produce a valid query answer (e.g., when there are no peers that can answer subquery **Q2**). This occurs when the peer processing the query does not contain sufficient data localization information in order to create a complete query plan and thus the routing phase produces an annotated graph with null annotations for certain patterns of the query. For handling this, SQPeer produces *partial query plans* with "holes" at the non-associative subquery patterns with relevant peers. These holes are denoted by the **?** symbol. In this case, the query plan should be further routed through the system, until appropriate data localization information is gathered in order to answer the unknown part. This additive routing activity of the query plan is governed by the architecture of the P2P system, since different architectures require different approaches in the query (sub-)plan routing. For example in an unstructured P2P system a peer may decide either to first acquire all the appropriate data localization information (by flooding, for example, routing requests) or create a partial query plan in the sense introduced above and route this plan until all data localization information needed is gathered. We should remind that partial query plans may be formulated only in a fully unstructured P2P system, since in the case of a hybrid or a structured P2P architecture already discussed in Section 3.4 the problem of insufficient data localization information is not encountered during the routing phase.

4.3 Cost-based Optimization

Query optimization's goal is to produce an efficient query plan with respect to the estimated query evaluation time. To this end, a cost model is required to estimate the total resource consumption or the response time of a specific query plan. The difference between these two cost metrics is that the former does not consider possible parallelization during the plan execution, i.e, the fact that fragment plans may be executed simultaneously at different peers. In SQPeer, we consider the latter metric,

since the main factor in most of the P2P systems is minimizing the time for receiving the complete answer without concerning on wasting system resources, which is computed by the total cost.

The cost is usually computed by adding the *processing cost* of the query plan operators and the *communication cost* from the data exchange between the peers contributing to the final query result. For computing this cost, appropriate statistics concerning both peers and their bases are needed. *Statistical information* may be either available locally at the peer undertaking query processing or may be on demand requested by the involved peers. Partially optimized query plans may also be created, when the peer contains incomplete statistical information and do not wish or is not permitted to ask the peers involved in the query plan. Query optimization, as already described in Chapter 2, can be distinguished into local and global optimization. In this section, we will focus on global query plan optimization, since a number of remote peers may contribute to the execution of the formulated query plan by undertaking a fragment of it and/or providing locally stored data.

SQPeer's query optimizer receives the query plan created by the algebraic translation algorithm and generates an optimized plan. A cost model is utilized by a dynamic programming approach in order to compute this optimized query plan. The necessary formulas, as long as a description of the approach on cost-based optimization followed in SQPeer is presented below.

4.3.1 Cost Model

In order to decide between different query plan deployments, i.e., decide which peer and in what order will undertake the execution of each query operator, and more importantly to evaluate the execution cost of a query plan, a cost model should be used.

Statistics are used in order to compute both the communication and the processing costs. These statistics include a) the speed of each peer's connection, b) the size of the relevant data stored in each peer base and c) the selectivity of the query operators. The connection speed characterizes the peer's capabilities for sharing and receiving

intermediary query results or requested data stored locally. The size of intermediary results can be estimated using the size of the stored data along with the selectivity of the operators handling these results. The first two statistics can be retrieved by the relevant peer by directly asking the peer for its data and connection capabilities. The third one is the most difficult to be acquired or even to be predicted in a precise way. Each query operator is equipped with a specific cost formula for evaluating the size of the intermediate results in accordance to the size of the input or output relations.

In [OV91] the cost of a query plan depicting the adopted execution strategy can be expressed with respect to either the total cost (measured in time) or the response time. The total cost is the sum of all cost components, while the response time corresponds to the necessary time period for fully executing the given query. A total cost formula is specified as follows:

$$\textbf{\textit{Total\_cost}} = C_{CPU} * \#insts + C_{I/O} * \#I/Os + C_{MSG} * \#msgs + C_{TR} * \#bytes \quad (4.1)$$

Processing time is computed by the first two components and corresponds to the CPU instruction and disk I/O costs. The communication time involves the creation and transmission of the messages needed for the inter-peer communication and data exchange and corresponds to the last two components. The transmission data unit is given here in terms of bytes, but could correspond to different units (e.g., packets).

For the response time both communication and processing time are used, but all possible parallel (both processing and communication) operations must be considered. A response time formula is given below:

$$\begin{aligned}\textbf{\textit{Response\_time}} = {} & C_{CPU} * seq\_\#insts + \\ & C_{I/O} * seq\_\#I/Os + C_{MSG} * seq\_\#msgs + C_{TR} * seq\_\#bytes \end{aligned} \quad (4.2)$$

where seq_x is the maximum number of x operations that must be done sequentially for executing the query. Thus any processing and communication done in parallel is not computed.

Minimizing response time, i.e., exploiting all possible parallel execution, does not imply the consequent minimization of the total cost. On the contrary, the total cost

may be increased from the increase of processing done in parallel. This may not be desirable in certain systems, as is the case of the P2P systems, and usually a compromise between the two cost metrics is made.

In [LPR98] a more simplified cost-based estimation for inter- and intra-peer query processing is used. The cost formula 4.3 consists of three independent factors: communication cost (C), local processing cost (L) and total response time cost (R).

$$\boldsymbol{Cost} = a_{cc} * C + a_{lqp} * L + a_{rt} * R \tag{4.3}$$

While the first two components correspond to the total cost, the third component is identical to the notion of response time seen in [OV91]. The coefficients in front of each of the formulas components can be controlled by the user according to which factor should be the primer concern for the system. For example, with the values (0 0 1), the response time is considered for the evaluation of each query plan. Statistics concerning the cardinalities of the partial results produced by query operators, which are calculated according to the selectivity factors of the operators, are used to estimate the total cost.

Finally, in [Sah02] the response time is computed similarly by using both processing and communication time. Processing time considers the evaluation of joins and cross products, with each one having a specific formula evaluating the CPU time for executing them according to the cardinalities of the input relations. Communication time on the other hand considers the network connections of the involved peers and computes the time for exchanging the necessary partial results. The bandwidth of each peer is modelled as a linear function and ranges from 56k to WAN peer connections.

Based on the above approaches, we first need to introduce the appropriate formulas for computing the cardinality of the output results for each operator of the query plan used in SQPeer. We focus on the two most time-consuming query plan operators, i.e., the join and the union. For the join operator the following cardinality estimation formula is used:

$$\boldsymbol{JoinCard}(\bowtie (Q_1, Q_2)) = card(Q_1) * card(Q_2)/(join\_selectivity)^m \tag{4.4}$$

where m stands for the number of the joined attributes between the two relations of the join. This formula considers the same *join_selectivity* for all the involved joined attributes, although this is not usually true. In the case of path queries, which are considered in SQPeer, the joined attributes are actually the common classes shared by the path patterns involved in the join.

Likewise, the cardinality estimation formula for the union operator is shown below:

$$\textbf{\textit{UnionCard}}(\cup(Q_1, \ldots, Q_i)) = \sum_{n=1}^{i} card(Q_n)/i + \max(card(Q_1),\ \ldots,\ card(Q_i)) \tag{4.5}$$

This formula is suggested by the definition of the union operator, since the union result will always be no less than the cardinality of the largest set participating in the union ($\geq max(card(Q_1),\ \ldots,\ card(Q_i))$) and as large as the sum of the sizes of the operands ($\leq \sum_{n=1}^{i} card(Q_n)$). So, as suggested in [GMUW00], something in the middle is the most appropriate choice, e.g., the average of the sum plus the larger.

Using the two formulas presented above, we can compute the cardinalities of all the intermediate results of our query plan with statistics concerning only the peer views at the leaves of the plan. Since peer views considered by SQPeer can be complex patterns that consist of more than one simple pattern, their statistics can be either retrieved for the complex pattern as a whole or can be computed with the above formulas by using the statistics of the simple patterns. These cardinalities will eventually help us compute not only the processing but also the communication cost in order to decide on the optimal query plan by considering operators reordering and execution.

The processing cost of a query plan can be computed by assigning to each of the query's operators a certain cost that corresponds to the CPU time required for computing it. This cost is proportional to the size of the output relations of the operator, since larger number of relations require more processing time.

In general, communication cost prevails in the computation of the overall query plan cost, in contrast to the processing cost, which usually is much smaller. This assumption, which is considered by most of the distributed database approaches, highlights the importance of favoring intra-peer processing, since communication cost

is minimized by taking into account as much as intra-peer operators as possible and thus producing the minimum amount of intermediate results for inter-peer exchange. Nevertheless, since the progress in peer connections results in faster communications, processing cost becomes more significant and notably affects the decision of executing an operator at a specific peer.

The processing cost formula of the join operator used in SQPeer is given below:

$$\begin{aligned} \textbf{\textit{ProcCost}}(\bowtie(Q_1, Q_2)) = \textit{join\_setup\_cost} + \\ \textit{JoinCard}(\bowtie(Q_1, Q_2)) \ / \ \textit{join\_ratio} \end{aligned} \tag{4.6}$$

The *join_setup_cost* corresponds to the constant time required for setting up the join operator. Respectively, the *join_ratio* is used for simulating the peer processor's speed, therefore faster processors require a bigger ratio.

A similar processing cost formula is used for the union operator, where *union_setup_cost* and *union_ratio* suggest the same thing:

$$\begin{aligned} \textbf{\textit{ProcCost}}(\cup(Q_1, \ \ldots, \ Q_n)) = \textit{union\_setup\_cost} + \\ \textit{UnionCard}(\cup(Q_1, \ \ldots, \ Q_n)) \ / \ \textit{union\_ratio} \end{aligned} \tag{4.7}$$

Note that the above model ignores the size of each tuple but only considers the cardinality of the relations. Moreover, it should be noted that given the nature of a P2P system like SQPeer, it is unrealistic to try and use a more refined cost model, since no accurate statistical knowledge can easily be obtained in a P2P setting neither a control on the local processing time of a remote peer can be systematically performed.

The communication cost in SQPeer is computed according to the following formula:

$$\textbf{\textit{CommCost}}(Q) = \textit{latency} + \textit{card}(Q)/\textit{bandwidth} \tag{4.8}$$

where *latency* represents the cost placed by the network distance between two peers, i.e., the constant time inserted at each link that connects two peers. The communication cost is defined as the sum of the *latency* as described above and the size of the exchanged data (measured by the cardinality of the exchanged relations)

divided by the *bandwidth*. The *bandwidth* is computed as the minimum bandwidth found along the path that connects the two considered peers. A fixed incoming and outgoing bandwidth is considered for each peer in the system. Moreover, a linear function is used for modelling bandwidth, since it provides a good approximation of the reality as proposed in [Sah02].

The processing cost of the operators in the query plan affects their ordering in the query plan. For example, by changing the possible order of the joins execution in the query plan, results in the change of the cardinalities of the intermediate results. This in turn affects the cost of executing the operators, since larger intermediate results means more intermediate processing time. On the other hand, the communication cost affects both the operators ordering and the selection of the appropriate peers for executing them. For the former, it is obvious that larger intermediate results mean larger transmission time for exchanging them between the involved peers. For the latter, since the processing time of a specific operator with specific operands is independent of the peer that handles the operator's execution, processing cost does not affect this selection. On the contrary, the communication cost may vary according to which peer takes the responsibility of executing the operator, since the remotely stored operands should be send to that specific peer.

For the cost estimation of a query plan, SQPeer uses the response time metric (i.e., formula 4.2) as discussed in [Sah02]. The processing and communication cost of each query plan operator is added and the response time of the query plan is computed.

To this end, another metric that may be considered by the optimizer is the processing load of the peers, since a peer that processes fewer queries, even if its connection is slow, may offer a better execution time. This processing load can be measured by the existence of slots in each peer, which show the amount of queries that can be handled simultaneously. The use of this metric in SQPeer is left as future work.

4.3.2 Dynamic Programming in SQPeer

4.3.2.1 Classic Dynamic Programming Algorithm

In this section, we will describe the classic dynamic programming algorithm used for optimizing queries in several distributed database systems as described in [SAC+79]. As input to the algorithm, we consider a query $\mathbf{Q}$ on relations $\{\mathbf{R_1}, \ldots, \mathbf{R_n}\}$. The algorithm outputs a query plan similar to the notion presented in Chapter 2. This plan is produced in a bottom-up way. First, the algorithm selects all access plans for every relation involved in the query, i.e., use of an index scan or simply read the table containing the relation. In the second phase, all possible ways of joining these relations are considered. Initially, all the two-way joins are considered (*joinPlans* function) by building a join plan consisting of two relations. Then all three-way joins are considered and so on until n-way joins are checked. In the last phase, the n-way plans are treated accordingly in order to become finalized (*finalizePlans* function) by for example attaching project or group-by operators.

An important feature of the dynamic programming algorithm is the fact that it discards at every step inferior plans. This approach is called *pruning*, since cheaper plans are kept for the next step and costly ones are pruned (*prunePlans* function). The pruning makes the dynamic programming algorithm to run significantly faster than an exhaustive search, where all expensive plans are carried through all the consequent steps. A pseudocode for the classic dynamic programming algorithm is given below:

ClassicDynamicProgrammingAlgorithm(Q)

```
1  for i ← 1 to n
2      do optPlan({R_i}) ← accessPlans(R_i)
3          prunePlans(optPlan({R_i}))
4  for i ← 2 to n
5      do for all S ⊂ {R_1, ..., R_n} such that |S| = i
6             do optPlan(S) = ∅
7                 for all O ⊂ S
```

8 **do** $optPlan(S) = optPlan(S) \cup$

9 $joinPlans(optPlan(O), optPlan(S \setminus O))$

10 $prunePlans(optPlan(S))$

11 $finalizePlans(optPlan(\{R_1, \ldots, R_n\}))$

12 $prunePlans(optPlan(\{R_1, \ldots, R_n\}))$

13 **return** $optPlan(optPlan(\{R_1, \ldots, R_n\}))$

Cost-based optimization in SQPeer requires the selection of an optimal query plan between different alternatives that considers not only the most appropriate query plan operator ordering but also the optimal selection of the peers that will undertake their execution. SQPeer's query optimizer uses a similar approach to the classic dynamic programming algorithm. Initially, the algorithm considers all relations required for the evaluation of the query. Then, it produces all the two-way joins producing one join plan for each such combination. Iteratively the algorithm considers three-way, four-way join plans, and so on up to n-way joins, where n is the number of relations involved in the plan. At each step of the algorithm, a pruning phase is used. During the pruning, plans that produce the same results are evaluated by using the cost model previously discussed and the best plan is selected for consideration at the next phase. After the final step, a number of equivalent query plans will be produced. Again these plans are evaluated and the best one is chosen for execution.

Since dynamic programming algorithm considers only the joins of the query plan and additionally the union operators reside either at the top or the bottom of the query plan, the union evaluation is done separately. Its actual implementation includes checking all possible peer assignments for each union operator and setting the best such assignment to be the "union collector" peer. The union operator evaluation is done either at the beginning (when the unions reside at the leaves of the query plan) or at the end of the dynamic programming execution (when the unions are pushed at the top of the query plan).

The output optimized query plan includes not only the best possible ordering of the query operators, but also assigns at each such operator the peer, where it is going to be most efficiently executed. An example of how the dynamic programming algo-

rithm works and the choices it makes concerning data or query shipping is presented in a following subsection.

4.3.2.2 Iterative Dynamic Programming Algorithm

Iterative dynamic programming (or IDP) algorithm [KS00] is an extension of the classic dynamic programming algorithm. Its main idea is to apply dynamic programming several times in the process of optimizing a query; either to optimize different parts of a plan separately or in different phases of the optimization process.

IDP works essentially in the same way as dynamic programming with the only difference that IDP respects that the available resources (e.g., main memory) of a machine (in our case a peer) are limited or that the user might want to limit the time spent for query optimization, which is also the case in P2P systems. To describe how IDP works, we assume that a peer has enough memory to keep all access plans, two-way, ..., k-way join plans for a query with n relations, where n>k. In such a situation, the dynamic programming algorithm would crash or be the cause of severe paging of the operating system when it starts to consider (k+1)-way join plans. On the other hand, IDP would generate all access plans, two-way, ..., k-way join plans like dynamic programming, but rather than starting to generate (k+1)-way plans, it would break, select one of the k-way join plans, discard all other access plans and join plans that involve one of the relations of the selected plan, and restart in order to build (k+1)-way, (k+2)-way,... join plans using the selected plan as a building block. A pseudocode for the iterative dynamic programming is given below:

IterativeDynamicProgrammingAlgorithm(Q)

```
1  for i ← 1 to n
2      do optPlan({R_i}) ← accessPlans(R_i)
3          prunePlans(optPlan({R_i}))
4  toDo = {R_1, ..., R_n}
5  while |toDo| > 1
6      do k = min{k, |toDo|}
7          for i ← 2 to k
```

8 **do for** *all* $S \subset$ *toDo such that* $|S| = i$
9 **do** $optPlan(S) = \emptyset$
10 **for** *all* $O \subset S$
11 **do** $optPlan(S) = optPlan(S) \cup$
12 $joinPlans(optPlan(O), optPlan(S \setminus O))$
13 $prunePlans(optPlan(S))$
14 **find** P, V **with** $P \epsilon\ optPlan(V),\ V \subset toDo,\ |V| = k$ **such that**
15 $eval(P) = min\{eval(P') \mid P' \epsilon\ optPlan(W),\ W \subset toDo,\ |W| = k\}$
16 **generate** *new symbol* : T
17 $optPlan(\{T\}) = \{P\}$
18 $toDo = toDo - V \cup \{T\}$
19 **for** $O \cup V$
20 **do** $delete(optPlan((O)))$
21 $finalizePlans(optPlan(toDo))$
22 $prunePlans(optPlan(toDo))$
23 **return** $optPlan(optPlan(toDo))$

In general, any plan enumeration algorithm faces the tradeoff between its complexity and the quality of the generated plans. IDP manages not only to have reasonable (i.e., polynomial) complexity, but also to produce in most situations very good plans. In particular, IDP produces better plans than any other algorithm during situations in which dynamic programming is not viable because of its high (i.e., exponential) complexity.

In SQPeer, we have additionally implemented an IDP algorithm for handling query optimization. More on the results and the comparison with the classic dynamic programming approach is presented in Section 4.7.

4.3.2.3 Data vs Query Shipping

Having the appropriate statistics in hand and by executing the dynamic programming algorithm, a peer actually decides at compile-time between *data, query* or *hybrid shipping* execution policies. We should remind that in the data shipping

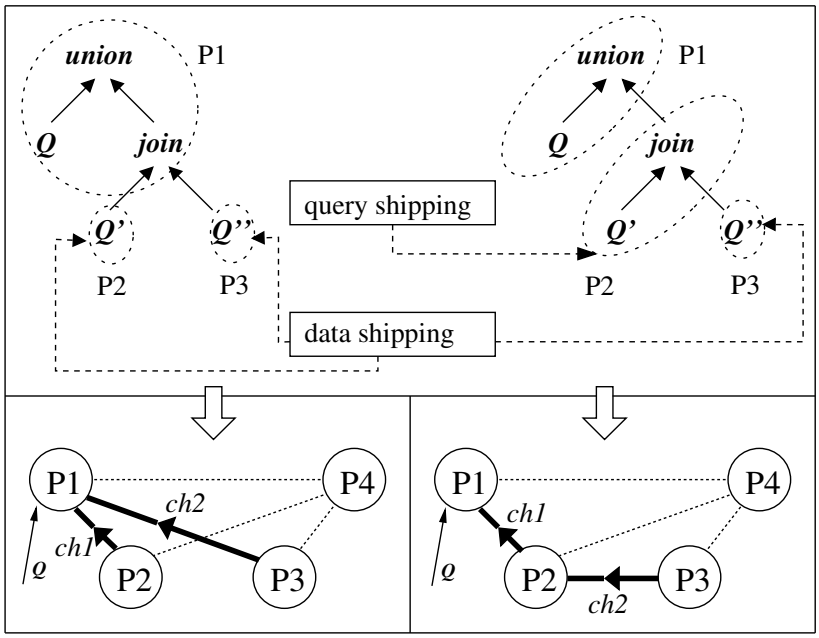


Figure 4.7: Data and Query Shipping Example

scenario, the appropriate resources are asked by the peer bases and all the processing is done locally. Alternatively, in query shipping, subqueries are sent to remote peers, thus pushing operators to be executed remotely. A hybrid approach is possible by combining the two previous execution policies.

Let us consider the existence of three peers, P1, P2 and P3, and a single query **Q** that can be broken into two simple subqueries, **Q'** and **Q"**, as can be seen in Figure 4.7. P1 can efficiently answer the whole query **Q**, while P2 and P3 can answer subqueries **Q'** and **Q"** respectively. By initiating the execution of the dynamic programming algorithm, access plans will be formulated for the simple subqueries involved in the query. This means that $scan(Q', P2)$ and $scan(Q'', P3)$ will be created as access plans for the two subqueries. The algorithm will then enumerate all two-way join plans using the access plans as the building blocks. This enumeration will consider all the relevant peers that are considered for carrying out the respective join. In our example, the two-way join plans that will be considered are three, i.e., $ship(scan(Q', P2), P1) \bowtie ship(scan(Q'', P3), P1)$, $scan(Q', P2) \bowtie ship(scan(Q'', P3), P2)$ and $ship(scan(Q', P2), P3) \bowtie scan(Q'', P3)$, where $ship(Q, P)$

corresponds to shipping the result of Q to the peer P. This means that either peers P2 and P3 send their data to peer P1 and P1 handles the join operator or one of the two peers will decide to send their data to the other one, which will eventually execute the join operator. While the first corresponds to the data shipping scenario, the other two choices corresponds to query shipping, since part of the query plan (i.e., the join operator) is sent for execution to a peer that contributes to the evaluation of this plan.

The selection between the three possible two-way join plans is performed by pruning the two plans and keeping the cheaper one. The cost evaluation of the plans is done according to formulas 4.6, 4.7 and 4.8 introduced in Section 4.3.1. For example, the first plan should consider the shipping cost of the intermediate results of peers P2 and P3 to the peer P1 and additionally the processing cost of the join at peer P1. On the other hand, the second plan needs only to compute the shipping cost of the intermediate results of peer P3 to peer P2 and the processing cost at peer P2. Therefore, in case the results obtained by executing **Q'** and **Q"** at their respective peers are rather small or the bandwidth between peers P2 to P1 and P3 to P1 is large enough, the first plan will be selected as the most appropriate. On the contrary, if for example P2 offers a large amount of data and the join operator produces small intermediate results, the second plan will probably produce the optimal cost.

In a similar way, the union operator is handled, by enumerating again all the possible plans. The two unions operands are the plan selected by the previously described computation and the plan $scan(Q, P1)$. As an example, if the first plan for the handling of the join is selected as the first optimal operand, then it is quite straightforward that the optimal plan for the union is to execute it locally at peer P1.

In the example of Figure 4.7, we can see two alternatives on how P1 can actually execute the generated query plan. In the left part of the figure we can see the data shipping alternative, since P1 sends queries **Q'** and **Q"** to peers P2 and P3 and joins their results locally. In the right part of the figure we can see the query shipping alternative, since P1 decides to forward the join operation down to P2, which in turn receives the results from P3 and executes the join locally before sending the full

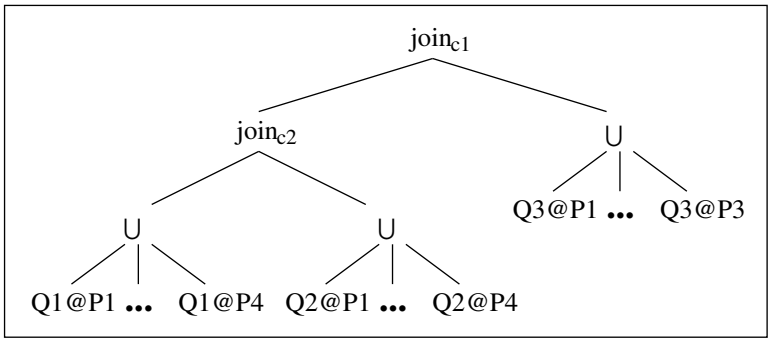


Figure 4.8: Query plan offering complete answer

answer to P1 for further processing.

At the bottom of the figure, we can see the deployment of the channels in SQPeer for each of these two alternative execution policies. More on the channel deployment will be discussed in Section 4.5.

4.4 Interleaved Query Routing and Planning

We distinguish between two scenarios concerning the execution of both query routing and planning algorithms in the context of SQPeer.

The first scenario involves the *sequential execution of query routing and planning phase*, where the user requires the creation of a single query plan offering complete results to the input query. In this scenario the fragmentor's *#joins* equals to the total number of joins involved in the input query. In this way, the produced query plan contains in its leaves, all the simple property patterns of the query and evaluates the answer by considering the most beneficial ordering of all the joins between them. Unions are utilized to gather all the data concerning each simple pattern, before they are further processed by the joins. This scenario offers completeness in the results of the query with the creation of a single query plan. An example of such a query plan can be seen in Figure 4.8, where all the joins and all simple patterns of the query **Q** of Figure 4.4 are considered.

However several tradeoffs concerning the evaluation of this plan should be considered:

- Data localization information for all simple patterns of the query should be gathered, which most probably will be huge considering that many peers will be capable of answering them.

- Furthermore, no intra-peer processing is considered during the routing phase. The planning phase should apply the dynamic programming algorithm with all the possible joins and by checking all the peers offering results, which increases the time required for the optimal query plan generation.

- Finally, the intermediate results produced during the query execution will be fairly large, since all the available data for each simple pattern are processed. This results in the creation of a quite expensive query plan regardless of the applied optimizations.

Considering the aforementioned arguments, there is a need for a second more complicated scenario, where the evaluation of the above complete query plan may be split into several steps. While the first scenario requires a single execution of both query routing and planning phase and a simple fragmentation for the input query, this is not adequate for this second scenario, where more complex fragmentation policies should be considered and many plans should be constructed and executed at several steps.

To this end, we introduce an *interleaved execution of query routing and planning phase* for obtaining a complete and correct answer to a given query. To do this, the data localization algorithm should consider all possible fragmentations of the query at several steps. The number of possible fragmentations is actually the number of combinations of all the possible fragments of the query produced by the execution of the fragmentor procedure for all possible *#joins* values. For each fragmentation policy, i.e., each combination produced by the fragmentor, the algebraic translation algorithm should be executed producing the appropriate query plan.

More precisely, this alternative initiates by considering the whole query pattern and at each step the input *#joins* variable of the fragmentation algorithm is increased by one. The query routing phase produces at each stage a different annotated query

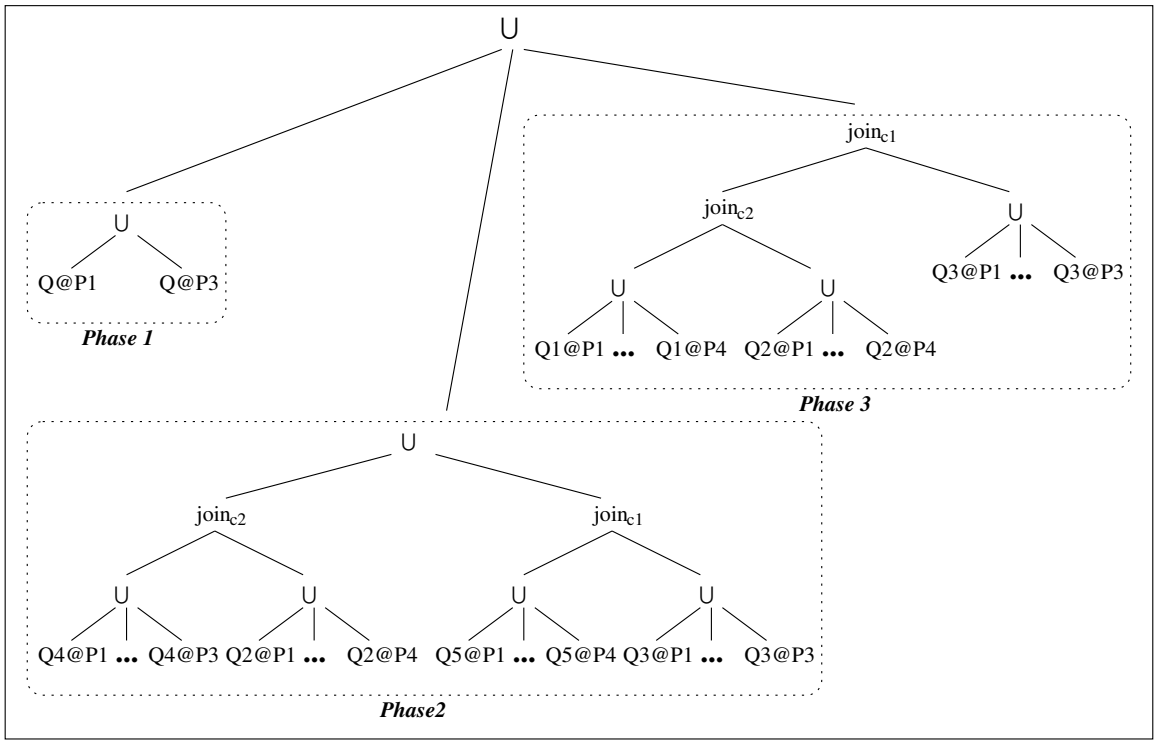


Figure 4.9: Query plans produced during interleaved query routing and planning

pattern (since the fragments are changed), which is passed to the query planner for producing and eventually executing the appropriate query plan. This is illustrated in Figure 4.9, where all the produced query plans for each possible fragmentation of the query **Q** of Figure 4.4 can be seen. Each such plan is unioned with the rest, since their combined results offer a complete answer to the given query. All unions contained inside the query plans of the consequent phases are introduced by the algebraic translation algorithm. However, the union employed for combining the results obtained from each distinct phase (i.e., for each *#joins* value) does not consist part of the respective query plans, but is rather suggested and actually executed at a selected peer (usually the client itself) receiving the final results for the posed query. In Figure 4.9, one such union can be seen at the top of the query plans of the three interleaved phases.

As already dicussed, the interleaved execution starts by considering those peers that answer the whole query pattern and gradually the size of the considered fragments is reduced until the simple property patterns of size 1 are reached. By using

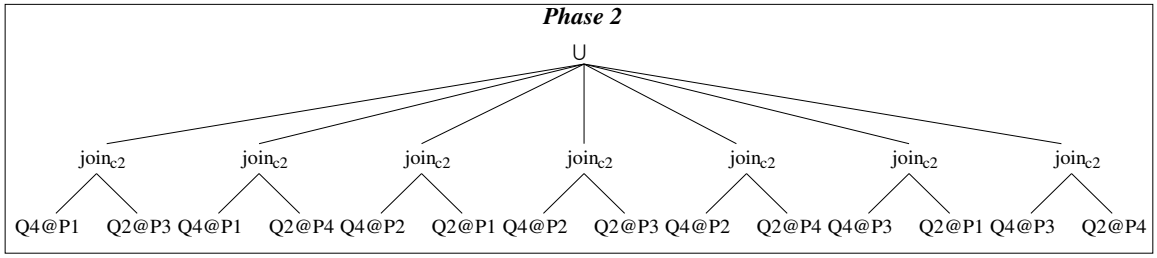


Figure 4.10: Query plan produced after applying algebraic equivalences

this order, it is easy to note that the latter phases of processing take into consideration answers that are already handled by the previous stages. For example, the first query plan of the second phase of Figure 4.9 involves a join at the class **C2**. The unions residing at the bottom of the query plan consider that subqueries **Q4** and **Q2** can be answered by the sets of peers {P1, P2, P3} and {P1, P3, P4} respectively. While the join between the results obtained from subqueries **Q4@P1** and **Q2@P3** produces new correct query answers, the join between subqueries **Q4@P1** and **Q2@P1** produces results already gathered at the first phase of the interleaved execution. This also holds for the results obtained from the evaluation of the subqueries at peer P3.

Receiving duplicates on the results does not only make the evaluation of the plan more expensive, but may also produce false results, since the user's query may not require the existence of such duplicates. In order to solve this we need to apply the algebraic equivalences of Section 4.1.3 and identify the intra-peer joins involved in the query plan. More precisely, the two heuristics discussed in the respective section are used to reorder unions and joins and identify the emerging intra-peer joins. Since these joins, as already depicted by the example, are answered by the previous stages of the interleaved execution, they can be removed. Figure 4.10 shows the first query plan of the second phase of Figure 4.9 after the reordering of unions and joins and the removal of redundant intra-peer processing already considered by previous phases. This results in retrieving at the end of the execution of all the produced query plans a correct and complete answer to the input query. No duplicates are considered at the end of this scenario and no additional cost in both processing and communication time is produced.

On the other hand the interleaved scenario exhibits several advantages. To start with, this scenario is most appropriate in a DHT-based system, where by using the provided efficient lookup service a user receives at a constant and rather inexpensive time the data localization information (in our case the peer views) for any path pattern needed.

More importantly, the interleaved execution permits parallelism, i.e, the pipeline formulation and execution of multiple query plans. This means that the processing phase can be broken easily by assigning to several peers different parts of the query plan. For instance, each peer may produce query plans based on a specific fragmentation by considering a *#joins* value. This corresponds to assigning each phase, as illustrated in Figure 4.9, to a different peer. Additionally, the same approach can be performed for the execution of the query plans, since each peer may undertake the execution of a different query plan.

Since the query is gradually broken into simpler patterns, the results obtained at the initial steps are more relevant and more cost-efficient than those produced at the final steps. This has a twofold value, since the time required for the receipt of the first results is quite small and additionally intra-peer processing is promoted. Receiving the first results as fast as possible is a primary concern for many P2P systems, since many users want to get early answers to their posed queries in the minimum possible time. In the scenario, where a single plan is produced, the user should wait for the planning phase, which is rather costly, to finish and then wait for this one plan execution, which again will consume a great amount of execution time.

On the other hand, the interleaved execution promotes intra-peer processing, since every time the query is broken, inter-peer processing is performed only at the joins emerged by the fragmentation. In particular, the fragments are send to be executed as a whole to the respective peers and since these fragments may involve complex patterns, their processing is locally performed (intra-peer) at the selected peer. For example, in Figure 4.9, the plans of the second phase involve one intra-peer join ($\bowtie$(**Q1**, **Q2**) and $\bowtie$(**Q1**, **Q3**) respectively) and one inter-peer join (at classes **C2** and **C1** respectively), while the plan at the third phase involves only inter-peer joins.

Easily we can notice that at each layer inter-peer processing is added, thus advancing the cost in both processing and execution time. This fact becomes more important, if we consider the cost of the dynamic programming algorithm (exponential if the classic approach is used) for optimizing the query plan. Since in the early steps the query operators considered for the ordering are fewer than their total possible number (e.g., in the second phase only one of the possible two operators are considered), the cost of the dynamic programming algorithm is consequently smaller.

Another fact that should be considered is that when the number of fragments is reduced, consequently the number of relevant peers and data is increased. In our example of Figure 4.9, the expected peers that are capable of answering the complete query **Q** in a real-life scenario are expected to be less than those answering the simple query fragments **Q1**, **Q2** and **Q3**. This makes the query plans, which are produced containing small fragments, more expensive than those where their respective fragments consider more inter-peer joins. Respectively, the query plans produced at the early stages of the interleaved execution, i.e., when the whole query pattern is considered, are less expensive than those produced when *#joins* is closer to the total number of joins of the query.

4.5 Query Execution and Communication Channels

In order to create the necessary foundation for executing distributed query (sub-) plans, as well as for exchanging data between the involved peers, SQPeer relies on appropriate communication *channels* [Sah02].

Through channels, peers are able to route (sub-)plans and exchange the intermediary results produced by their execution. It is worth noticing that channels allow each peer to further route and process autonomously the received (sub-)plans, by contacting peers independently of the previous routing operations. This functionality sets the basis for interleaved execution of query routing and processing, since each peer acts autonomously and may further ask for additional peer advertisements or

process the query by altering the query plan received according to new gathered data localization information. Finally, channel deployment can be adapted during query execution in order to response to network failures or peer processing limitations.

Each channel has a root and a destination node. The root node of a channel is responsible for the management of the channel by using its local unique id. Data packets are sent through each channel from the destination to the root node. Beside query results, these packets can also contain information about network or peer failures for possible plan modification or even statistics for query optimization purposes. The channel construct and operations of ubQL [Sah02] are employed to implement the above functionality in the SQPeer middleware.

Once a query plan is created and a peer is assigned to its execution, this peer is responsible for the deployment of the necessary channels in the system. In Figure 4.5 one such example can be seen, where the peer responsible for the query processing (P1), creates three channels with the appropriate peers (P2, P3 and P4) for enabling the query execution. A channel is created having as root the peer launching the execution of the query and as destination one of the peers that need to be contacted each time according to the plan. Although each of these peers may contribute in the execution of the plan by answering to more than one query fragments, only one channel is of course created.

Query plan adaptability, which is necessary in a fully dynamic P2P system, usually requires to carry out the query routing and planning phases after the initiation of the execution phase. This may produce query plans that consider more update data localization or peer statistical information, thus adapting to possible problems caused at run-time. This means that query execution should also handle the monitoring of the executed query plans in order to re-execute query processing when necessary. Since this service is described as run-time query optimization, it will be discussed in more details in the following section.

4.6 Run-time Query Plan Adaptability

Run-time adaptability of query plans is an essential characteristic of query processing when peer bases join and leave the system at free will or more generally when system resources are exhausted. For example, the optimizer may alter a running query plan by observing the throughput of a certain channel. This throughput can be measured by the number of incoming or outgoing tuples (i.e., resources related through one or several properties). Changing query plans may alter an already installed channel, as well as the query plans of the root and destination peer of the channel. These changes include deciding at execution time on altering the data or query shipping decision or discovering alternative peers for answering a certain subplan. The root peer of each channel is responsible for identifying possible problems caused by environmental changes and for handling them accordingly. It should also inform all the involved peers that are affected by the alteration of the plan. Since the alteration is done on a subplan and not on the whole query plan, only the peers related to this subplan should be informed and possibly a few other peers that contain partial results from the execution of the failed plan. Finally, the root peer should create a new query plan by re-executing the routing and planning phase and not taking into consideration those peers that became obsolete.

We should keep in mind that switching to a different query plan in the middle of the query execution raises new challenges. Previous results, which were already created by the execution of the query to possible multiple peers, have to be handled, since the new query plan will produce new results. Two are the possible solutions to this issue. The ubQL approach [Sah02] proposes to discard previous intermediate results and all on-going computations are terminated. Alternatively [Ive02] proposes a phased query execution, in which each time the query plan is changed, the system enters into a new phase. The final phase, which is called the cleanup phase, is responsible for combining the sub-results from the other phases in order to obtain a full answer. In SQPeer middleware, we have adopted the ubQL approach.

4.7 Experiments in SQPeer

In this chapter, we experimentally evaluate the proposed interleaved and sequential execution of the query planning and routing algorithms in order to point out the need for an alternative way in query processing. The following experiments are conducted by using the ubQL virtual machine ($ubQL^{VM}$), a simulator for the ubQL language as introduced in [Sah02]. All code is written in Java. A SUN Blade 100 Workstation with a 500MHz processor, 128MB of main memory, was used to install the simulator and run the following experiments.

Peers in $ubQL^{VM}$ are modelled as separate execution threads. By this way, complex network configurations can be represented without the need for physical peers. Queries can be deployed and executed in an arbitrary way. Certain tasks, including local processing, query deployment and data shipping, are simulated by the $ubQL^{VM}$. On the contrary, query planning and optimization are physically executed and can be measured at real time. The lookup service is also simulated and relevant peer views are considered to be already locally stored at the peer handling query processing.

The following parameters are considered in order to outline different situations and scenarios for the creation and final execution of the SQPeer distributed query plans:

Queries The structural nature of the queries (e.g., linear, tree, graph) affects both the form of the query plans, as well as, the query planning time based on the dynamic programming approach. We distinguish between two different types of queries, i.e., linear and graph ones.

Data Distribution The distribution of the data inside the P2P system affects the performance of each execution strategy. For example, horizontal distribution implies the use of multiple union operators to combine results spread in remote peers, while vertical distribution requires the evaluation and ordering of inter-peer joins. Moreover, the number of peers that are capable of answering a particular query (i.e., when the peer view is subsumed by the query) affects the cost for formulating, optimizing and executing the query plan.

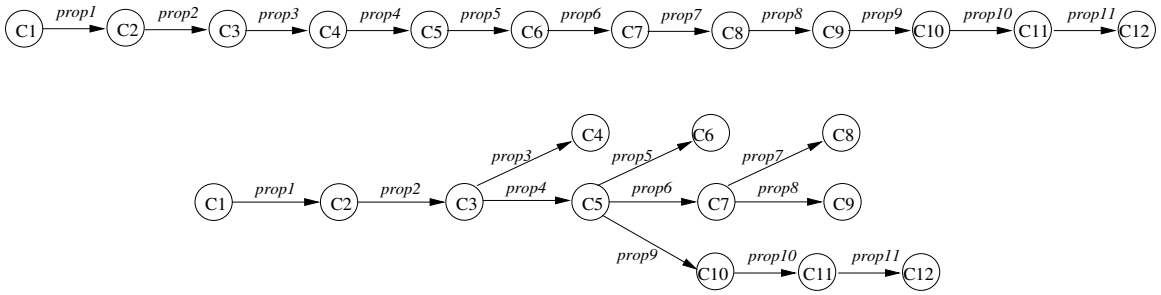


Figure 4.11: Linear- and graph-query examples

Optimization Decisions In order to provide efficient execution time, the query plan needs to be optimized accordingly. Actually, the execution time of the dynamic programming algorithm for producing the optimal query plans consumes the majority of the time required by query planning. More precisely, the distribution of joins and unions with the additional appliance of the interleaved query planning and routing phases exhibit different behaviors in the evaluation of the distributed queries.

As far as the structural nature of the queries is concerned, we have considered two queries of different type, whose graph is shown in Figure 4.11. The first is a linear query consisting of 12 classes and 11 properties. The second is a graph query with the same number of classes and properties. The difference lies in the existence of star joins, i.e., joins where more than two properties are joined at the same class (e.g., the join at class `C3`). Even though, both queries involve 10 joins, the fragmentation of the graph query is a more demanding task considering that a joined class may be shared by more than two simple patterns.

The interleaved routing and planning scenario suggests the creation of multiple plans in several phases, with each phase considering more inter-peer joins than its preceding. We remind that this scenario suggests the appliance of the algebraic equivalence for pushing unions to the top of the query plan produced by each phase (see Figures 4.9 and 4.10). Under the union operator produced by each phase there are a number of query plans that consist of only inter-peer joins and that require optimization. In order to compute this number and thus compare the interleaved

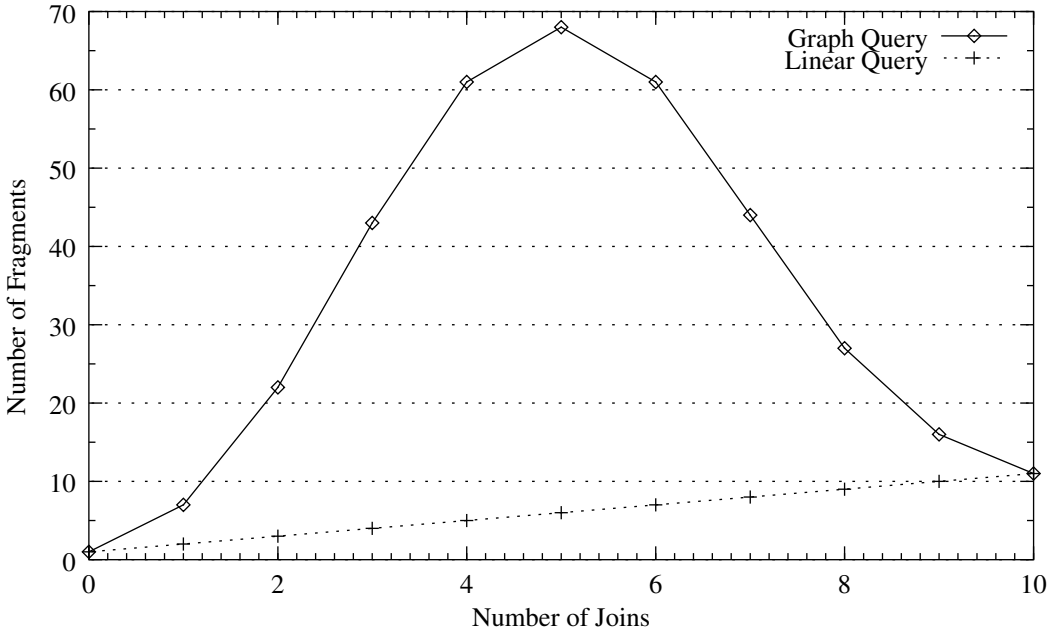
routing and planning strategy (producing several "unioned" plans with partial results) with the sequential one (producing one plan with complete results), we first need to identify the number of fragmentations produced in each phase and then to consider the number of peers that are capable of answering each such fragment.

4.7.1 Query Fragments & Fragmentations

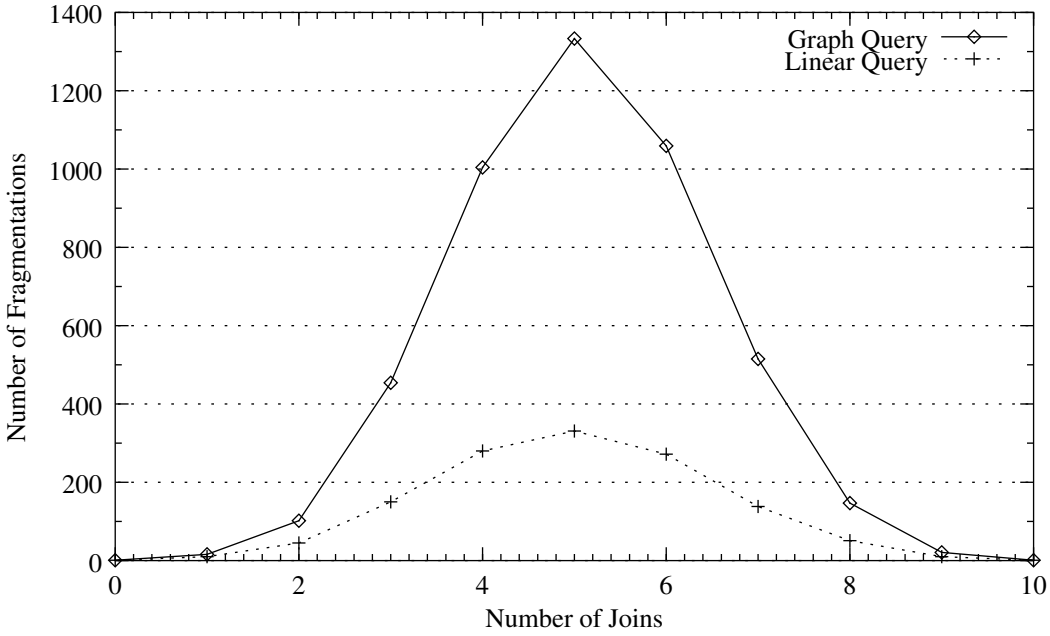
To specify the possible fragmentations for a given query, we initially need to identify all the possible fragments that can be produced. Figure 4.12a shows the number of fragments for the two example queries of Figure 4.11 with respect to the number of joins they involve. As it was expected, the number of fragments for the graph query is fairly larger than the respective one for the linear query due to the existence of star joins.

On the other hand, Figure 4.12b depicts the number of possible fragmentations produced by combining all the already produced fragments. We can see that only one fragmentation is possible, when we have no joins and the fragmentation is the whole query, and when all the possible joins are considered and the fragmentation considers all the simple patterns of the query. Compared to the linear query, the total number of possible fragmentations for the graph query is again significantly larger, since there are more possible combinations for its fragments. This growth affects not only the execution time of the planning algorithm but also the number of possible plans, when all the fragmentations of the query should be considered (as is the case in the interleaved approach). Overall, graph queries require more planning time considering their structural complexity and consequently increase the demands in their fragmentation and routing, since more sophisticated fragmentations are produced.

In the following Table 4.1, we can see the number of fragments that are used to construct the fragmentations of different size for the two example queries. The first table corresponds to the linear query, while the second to the graph one. For instance, for the linear query, when no joins are considered, only one fragment of size 11 is used. Respectively, for the fragmentations with 9 joins, 90 fragments of size 1 and 10 fragments of size 2 were combined to formulate them.



(a) Fragments



(b) Fragmentations

Figure 4.12: Fragments and fragmentations for the linear and graph queries

#frags #joins	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	1
1	2	2	2	2	2	2	2	2	2	2	0
2	27	24	21	18	15	12	9	6	3	0	0
3	183	135	106	74	53	29	16	4	0	0	0
4	560	372	233	136	67	26	6	0	0	0	0
5	984	561	281	115	38	7	0	0	0	0	0
6	1128	530	188	50	8	0	0	0	0	0	0
7	772	258	66	8	0	0	0	0	0	0	0
8	366	84	9	0	0	0	0	0	0	0	0
9	90	10	0	0	0	0	0	0	0	0	0
10	11	0	0	0	0	0	0	0	0	0	0

#frags #joins	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	0	0	0	0	0	0	0	0	0	0	1
1	6	3	3	3	1	1	3	3	3	6	0
2	90	48	38	29	17	21	26	20	17	0	0
3	692	356	237	168	130	107	93	33	0	0	0
4	2287	1129	690	417	271	176	50	0	0	0	0
5	4292	1933	961	497	256	59	0	0	0	0	0
6	4588	1797	699	275	54	0	0	0	0	0	0
7	2924	894	255	47	0	0	0	0	0	0	0
8	1060	232	31	0	0	0	0	0	0	0	0
9	189	22	0	0	0	0	0	0	0	0	0
10	11	0	0	0	0	0	0	0	0	0	0

Table 4.1: Association between fragments and fragmentations of the linear and graph queries

4.7.2 Number of Peers

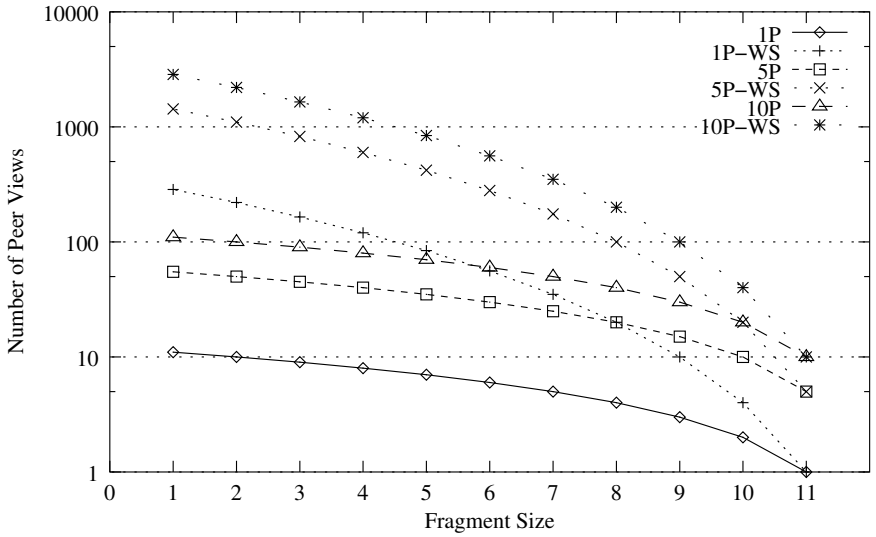
The number of peers answering each consequent fragment of the respective SON RDF/S schema is something that cannot be easily estimated. Moreover, we should consider that a peer that contains data conforming to a certain view is also capable of answering all of its fragments, as suggested by the horizontal subsumption. Thus, the number of peers that answer views of small size is expected to be larger than those that can answer bigger ones. Additionally, vertical subsumption of query/view patterns similarly affects the number of peers answering each query. While the former type of subsumption is affected by the form of the schema graph comprising properties, the latter type is affected by the size of class or property subsumption hierarchies (see Section 3.3.3). For simplicity, in our experiments we focus on the horizontal subsumption, but the same considerations can be made for the vertical one as well.

In order to identify the number of peers answering each fragment of the SON RDF/S schema, we consider that each view that can be produced from it can independently be answered by a *constant* number of peers. In order to take into account horizontal subsumption, where views of smaller size can be answered by more peers that those of bigger size, we employ the following formula for computing the number of peers advertising a view with size m:

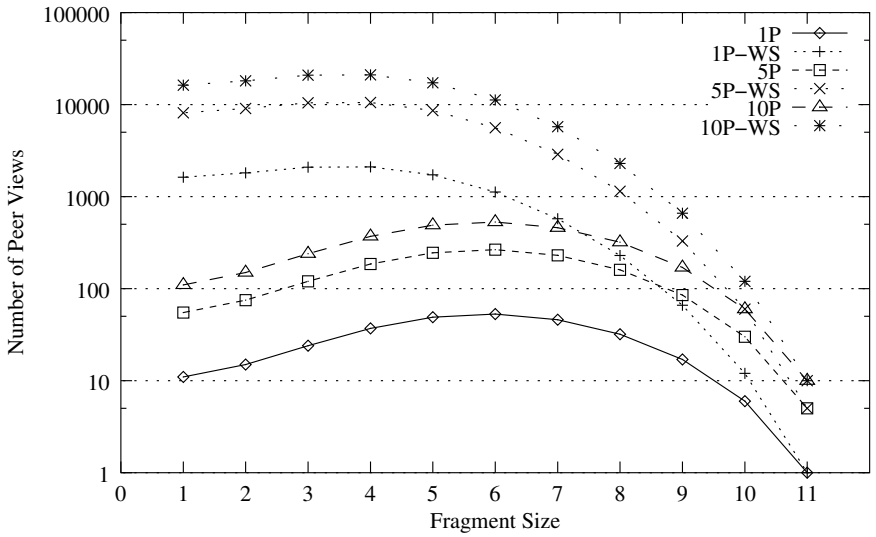
$$P(m) = constant + \sum_{i=m+1}^{n} P(i) \tag{4.9}$$

where $View(i) \subseteq View(m)$ denotes that $View(i)$ is horizontally subsumed by $View(m)$, i.e., a peer answering a view of size m, i.e., $View(m)$, can also answer all view fragments of size i, i.e., $View(i)$.

For example, if only one peer is considered to advertise each view and the schema of query **Q** of Figure 4.4 is used, then **Q** can be answered by one peer, **Q4** and **Q5** can be answered by two peers (considering that $Q \subseteq Q4$ and $Q \subseteq Q5$), **Q2** and **Q3** can be answered by three peers (considering that $Q5 \subseteq Q2$, $Q \subseteq Q2$, $Q4 \subseteq Q3$ and $Q \subseteq Q3$) and finally **Q1** can be answered by four peers (considering that $Q4 \subseteq Q1$, $Q5 \subseteq Q1$ and $Q \subseteq Q1$) (all the view fragments can be seen in Figure 4.3).



(a) Linear query



(b) Graph query

Figure 4.13: Distribution of peer views answering schema fragments

Figure 4.13 depicts the distribution of data considering our two example queries and identifying the number of peers capable of answering them. The *constant* value is set to be 1, 5 and 10, in the sense that one (five or ten) peer(s) contains resources conforming to the whole schema and additionally one (five or ten) peer(s) can answer each fragment of the query.

In the figure, we illustrate for each consequent fragment size and for each *constant* peer value, the total number of peers containing views of size equal to the fragment size. For each *constant* value and for each type of query (linear or graph), we have two different values for the respective number of views. The difference is that the latter considers the horizontal subsumption (depicted with the WS, meaning With Subsumption) by adding the number of peers that answer all the subsumed views. The graph should be read from right to left, since both values considered for the y axis are increased as the size of the fragments decreases (x axis). However, this is not the case in the graph query, where the number of peers decreases from a certain value of the fragment size. This is due to the fact that the number of fragments for the graph query increases rapidly when middle-size fragments are considered.

Easily we can also detect that for the graph query the number of answering peers reaches greater values than in the linear query. This occurs due to both the increase in the fragment number and in the increase of the number of bigger fragments that are subsumed by the smaller ones, which affects the second component of the formula 4.9.

The above experiment shows the way data are distributed through the system, in the sense that smaller queries will most probably be answered by a considerably larger number of peers than bigger ones. The number of answering peers affects the query planning algorithm with respect to the query plan equivalences described in Section 4.1.3. In the case of a complete query plan, where the unions reside at the bottom of the query plan, the number of peers affect poorly the planning phase. However, this comes with a tradeoff in the execution of the query plan, since for each simple pattern considered in the query plan, one peer should undertake the task of collecting all the data answering this pattern, which considering their expected popularity will

be rather huge. In the case of interleaved routing and planning with additional push of the unions to the top of the query plan, the planning time is influenced by the respective number of plans that are created. However, the possibility of parallelization of the planning and execution phases may lighten the processing requirements for this scenario.

In general, data distribution is an important factor for query processing, where complete results need to be returned, since it affects both planning, optimization and execution time.

4.7.3 Number of Plans

As discussed in Section 4.4, in the interleaved query routing and planning phase, the only way to achieve completeness and correctness in the results is to push unions at the top of the query plan. This means that at each consequent phase, as depicted visually in Figure 4.9, a number of plans (consisting of only inter-peer joins) exist under each union. This number equals to all the possible plans created from different fragmentations but with the same number of inter-peer joins. However, at each such fragmentation, if the unions, which combine the data for each fragment from all the peers that can answer it, are pushed up (as seen in Figure 4.10), the number of these plans are increased. More precisely, if we consider n number of fragments and P_n the number of peers answering each fragment, all the possible combinations for a single fragmentation after we push the unions to the top are $\prod_{i=1}^{n} P_i$. However, at each step of the interleaved execution, query plans that are considered at the previous phases of the execution should be removed, since their results are already computed. This is performed after applying the two heuristics explained in Section 4.1.3; all the plans that are already processed during the previous phases should be subtracted from the number of plans captured by our formula. This actually means that only inter-peer processing is considered, since all emerging intra-peer joins have been handled in the previous phases.

Considering the distribution of data shown in Figure 4.13 in the peers of our system, we show the number of possible plans that equals to the number of the

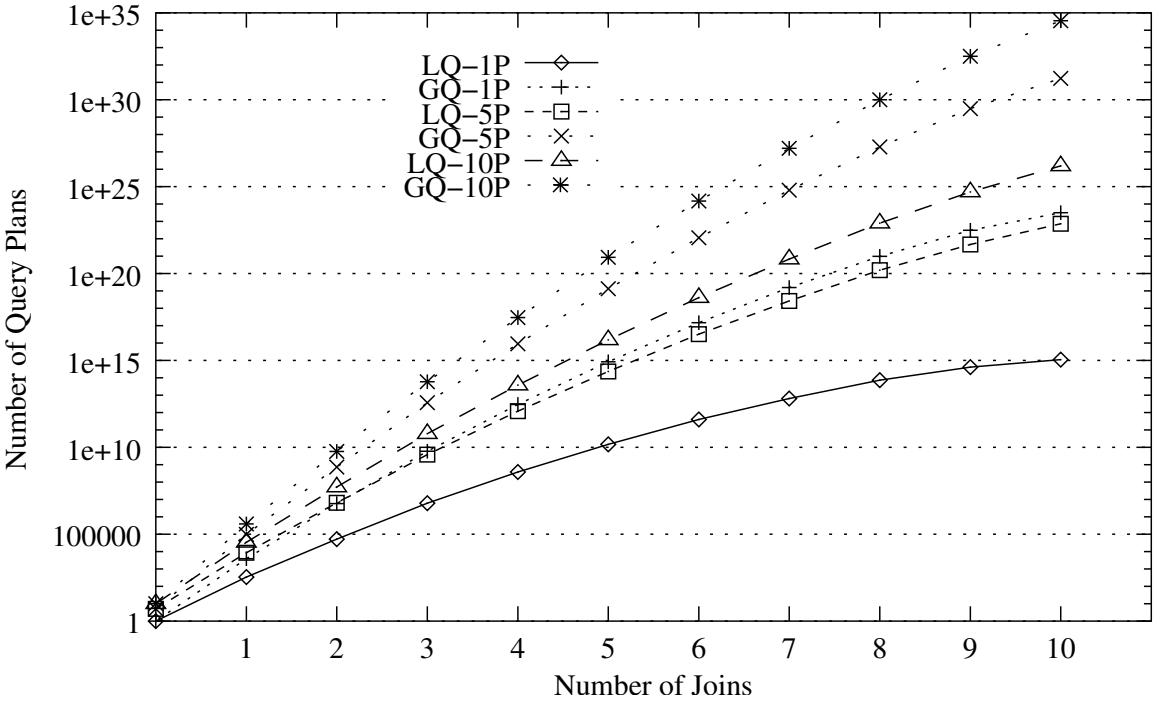


Figure 4.14: Number of plans with respect to the distribution of data

possible combinations with respect to this distribution. We present four types of experiments in Figure 4.14, two with the linear query and two with the graph one. For each type of queries, the *constant* value used for the computation of the number of peers answering each fragment of the query is set either to 1 or 5. The illustrated number of plans are the plans that should be considered if we push unions to the top of the query plan produced by the query planning algorithm. A logarithmic scale was used for the y axis, since the number of plans increases exponentially to the number of inter-peer joins.

By this experiment, we can realize that it is inefficient to optimize and execute at once all the produced plans, so more elaborate techniques should be considered. As far as the type of the queries is concerned, graph queries produce greater number of query plans than the linear queries taking into account both their increased number of fragmentations and their distribution in the system.

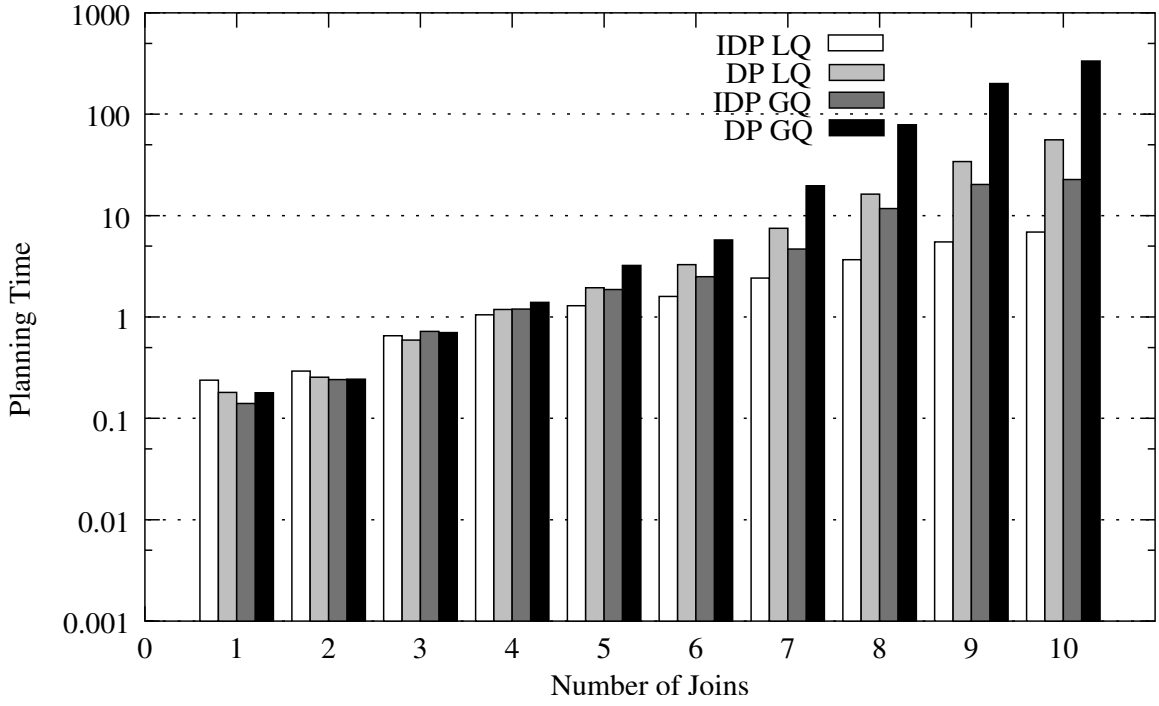


Figure 4.15: Iterative vs Traditional Dynamic Programming

4.7.4 Planning Time

For SQPeer's query planning and optimization phase, a dynamic programming approach is used. More precisely, we have implemented a classic and an iterative dynamic programming algorithm. Figure 4.15 shows one of the experiments performed for our two example queries and for both algorithms. The time presented in the figure concerns the formulation of one query plan considering one possible fragmentation of the query and that only one peer answers each involved fragment. The fragmentation size ranges from one fragment to the maximum number of the simple fragments that can be extracted from the input query. As can be seen, the classic dynamic programming execution time is exponential with respect to the number of joins in the constructed query plan. The iterative approach, on the other hand, offers polynomial execution time with little effect in the quality of the produced results. Additionally, it offers solution, i.e., produces an optimized query plan, in cases where the dynamic approach cannot due to time or resource limitations.

We should note that the planning time is affected not only from the number of joins

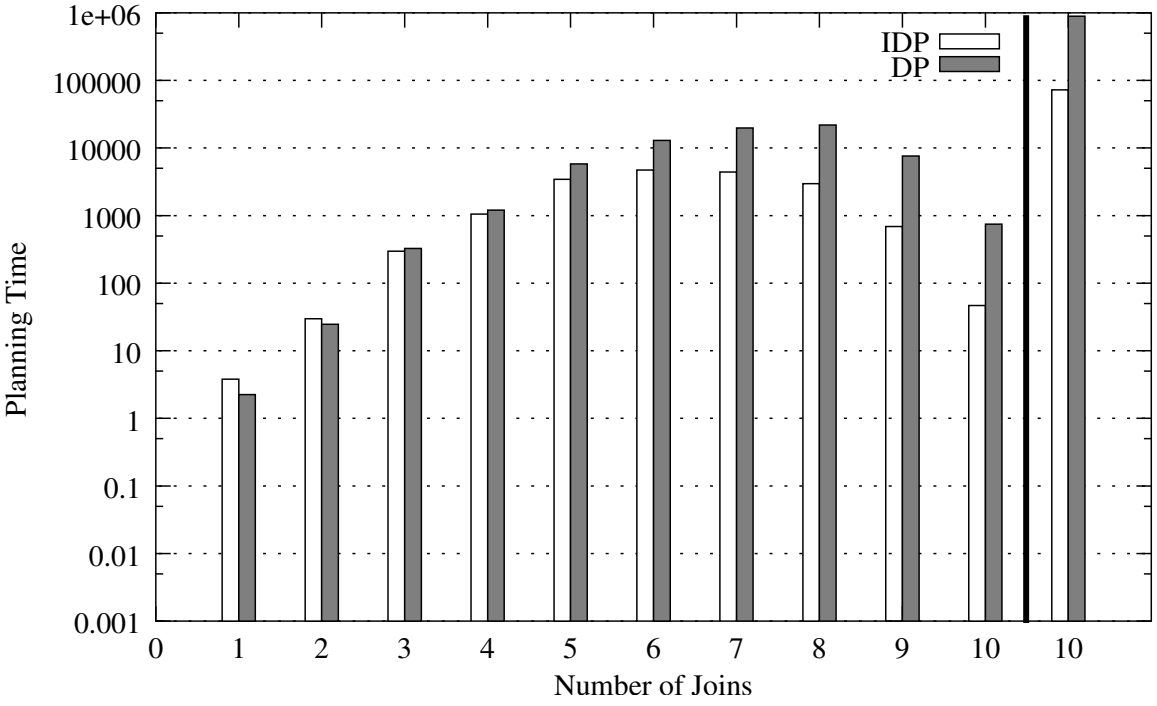


Figure 4.16: Interleave vs Sequential Planning

that need to be reordered, but also from the number of candidate peers for executing the operators of the query. On the other hand, the evaluation of the union operators is rather trivial compared to the rest of the planning time, since their reordering is handled by the algebraic equivalence introduced in Section 4.1.3.

In the sequential scenario considered in Section 4.4, one query plan is formulated that produces complete results for the given query. This means that only one execution of the query planning phase is needed. However, in the interleaved routing and planning scenario the same conditions do not hold, since all possible fragmentation sizes are considered and the planning phase is executed more than once. We have also seen that in order to favor the less expensive intra-peer processing (see Section 4.1.3), we need to push the unions to the top of the query plan by using appropriate query plan equivalences.

Considering the above, we show an experiment of running the planning algorithm by considering the graph query. We run the interleaved scenario with each phase having more inter-peer joins than the previous one. We again consider that only

one peer can answer each fragment of the query, so the number of plans that are formulated for each phase is bounded by the possible fragmentations. As can be seen in Figure 4.16, this number affects the planning time, which at each phase practically equals to the time optimizing one plan with the considered number of joins and multiplying it with the number of possible fragmentations. We can note that in the final phases of the interleaved execution, the required planning time decreases, since the number of fragmentations also decreases. We should also recall that when each phase completes, the formulated query plan can be immediately executed, since the results of each phase are independent from the previous and the following ones. The user can decide on the number of query results he wants to receive by stopping the planning algorithm at any desirable phase. The last column in the figure shows the sequential execution of the planning algorithm. In this scenario, one plan is created having one union and all the possible inter-peer join plans underneath. As can be seen, the planning phase is rather expensive, since all the possible combinations should be checked. Moreover, no parallelization is possible and the optimization should end before the execution of the query plan can begin. In the sequential scenario, where no complex fragments are considered, so data localization information is retrieved only for the simple query patterns, intra-peer joins should also be optimized and identified during the optimization phase. This is not the case in the interleaved scenario, where at each phase complex fragments are looked up and intra-peer joins need not to be optimized.

The above experiments show that under certain circumstances the interleaved query routing and planning phase is the most appropriate processing policy. More precisely, when the user is interested in receiving as quick as possible results concerning its posed query, the interleaved processing should be chosen, since its first plans are produced and executed very quickly. Additionally, the results obtained at the first phases are the most relevant to the query, since peers that can answer bigger fragments of the query are contacted first. However, in the case where result completeness is the user's will, then a single execution of the planning algorithm should be chosen, i.e., the one that examines each simple pattern of the query and produces

a single query plan by using all the available data localization information.

Chapter 5

Conclusion

P2P systems adopt a completely decentralized approach to resource management. By distributing data storage, processing and bandwidth across all peers in the network, they can scale without the need for powerful servers. However, these systems show severe limitations in contrast to traditional data management systems: file-level sharing, read-only access, simple keyword-based search and poor scaling. Thus, in order to support highly dynamic, ever-changing, autonomous social organizations (e.g., scientific or educational communities), we need richer facilities in exchanging, querying and integrating (semi-)structured data hosted by peers.

In this thesis, we have dealt with the above problem by introducing SQPeer as an RDF/S-based P2P data management system providing a fully-fledged framework for evaluating semantic queries over remote peer RDF/S bases (materialized or virtual) in a distributed way.

In particular, we presented how (conjunctive) RQL path queries expressed against a SON RDF/S schema can be represented as semantic query patterns. Peers advertise their content to the rest of the system through appropriate peer advertisements. These advertisements, expressed as RVL view patterns, declare the parts of the RDF/S schema which are (or can be) populated in a peer base. In this way, an intensional representation is achieved for both query requests and peer advertisements.

We detailed a semantic query processing involving both query routing and planning issues. Semantic query routing discovers peer views advertised throughout the

system that are relevant to a given input. A data localization algorithm relies on query/view subsumption techniques to annotate semantic query patterns with information concerning relevant peers. We also presented how SQPeer query plans are created and executed by taking into account the data distribution in peer bases. Issues emerging from the interleaved query routing and planning phase were discussed and several compile and run-time optimization opportunities for SQPeer query plans were illustrated. Based on the above context, we demonstrated the application of SQPeer on several architectural alternatives for hybrid, structured and ad-hoc RDF/S-based SONs. Finally, a set of experiments were conducted for reasoning on the execution of the proposed query planning algorithm with respect to a number of parameters affecting it.

5.1 Future Work

The functionality of SQPeer presented in this thesis can be extended in several ways.

The cost model introduced in Section 4.3.1, although based on previous work on query plan optimization in traditional and distributed database systems, does not reflect a real life scenario. A more refined cost model is needed based on actual operation execution and cardinality statistics that will consider and cope with the characteristics of the underlying P2P system.

The number of query plans that are generated by the query planning algorithm by considering all fragmentation alternatives of the given query pattern can be fairly large. Pruning the available plans may be necessary with the use of appropriate coverage metrics [DH02] [NK01]. It would also be interesting to study the trade-off between result completeness and response time in terms of processing and communication cost using the notion of Top N (or Bottom N) queries [Kos00] [TSBN04]. In the same direction, we can use constraints regarding the number of peers that each query is broadcasted and further processed.

Finally, we can consider adaptive implementations of algebraic operators borrow-

ing ideas from [AH00] [HJ04] [ILW+00] and thus providing more efficient run-time optimization opportunities.

Bibliography

[ABC+04] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France, 2004.

[ACK04] N. Athanasis, V. Christophides, and D. Kotzinos. Generating On the Fly Queries for the Semantic Web: The ICS-FORTH Graphical RQL Interface (GRQL). In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*, Hiroshima, Japan, 2004.

[ACMH03] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. The Chatty Web: Emergent Semantics Through Gossiping. In *Proceedings of the 12th International World Wide Web Conference (WWW)*, Budapest, Hungary, 2003.

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

[AH00] R. Avnur and J.M. Hellerstein. Eddies:Continuously Adaptive Query Processing. In *ACM SIGMOD*, pages 261–272, Dallas, TX, 2000.

[BBMN02] M. Bonifacio, P. Bouquet, G. Mameli, and M. Nori. KEx: a Peer-to-Peer Solution for Distributed Knowledge Management. In *Proceedings*

of the 4th International Conference on Practical Aspects of Knowledge Management (PAKM02), Vienna, Austria, December 2002.

[BDK+04] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In *Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, Toronto, Canada, 2004.

[BG03] J. Bremer and M. Gertz. On Distributing XML Repositories. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, San Diego, California, 2003.

[BGK+02] P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. 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.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe:Ubiquitous Query Processing On The Internet. In *VLDB Journal*, pages 48–71, 2001.

[BMWZ04] M. Bender, S. Michel, G. Weikum, and C. Zimmer. Bookmark-driven Query Routing in Peer-to-Peer Web Search. In *Proceedings of the SIGIR Workshop on Peer-to-Peer Information Retrieval*, 2004.

[Bra03] S. Brahmananda. Design of Peer-to-Peer Protocol for AmbientDB. Master's thesis, University of Twente, 2003.

[BT03] P. Boncz and C. Treijtel. AmbientDB: Relational Query Processing in a P2P Network. In *Proceedings of the International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*. Springer Verlag, 2003.

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

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

[CKK+03] V. Christophides, G. Karvounarakis, I. Koffina, G. Kokkinidis, A. Magkanaraki, D. Plexousakis, G. Serfiotis, and V. Tannen. The ICS-FORTH SWIM: A Powerful Semantic Web Integration Middleware. In *Proceedings of the SWDB'03 International Workshop, Berlin, Germany*, Humboldt-Universitat, Berlin, Germany, 2003.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009. Springer Verlag, 2001.

[DH02] A. Doan and A. Halevy. Efficiently Ordering Query Plans for Data Integration. In *Proceedings of the 18th IEEE Conference on Data Engineering (ICDE)*, 2002.

[ETB+03] M. Ehrig, C. Tempich, J. Broekstra, F. van Harmelen, M. Sabou, R. Siebes, S. Staab, and H. Stuckenschmidt. SWAP - Ontology-based Knowledge Management with Peer-to-Peer Technology. In *Proceedings of the 1st National "Workshop Ontologie-basiertes Wissensmanagemen" (WOW)*, 2003.

[FJK96] M.J. Franklin, B.T. Jonsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proceedings of the ACM SIGMOD Conference*, pages 149–160, Montreal, Canada, 1996.

[GMUW00] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[Gnu] The Gnutella file-sharing system. http://www.gnutella.com.

[GWJD03a] L. Galanis, Y. Wang, S.R. Jeffery, and D.J. DeWitt. Locating Data Sources in Large Distributed Systems. In *Proceedings of the 29th Conference on Very Large Databases (VLDB)*, 2003.

[GWJD03b] L. Galanis, Y. Wang, S.R. Jeffery, and D.J. DeWitt. Processing Queries in a Large P2P System. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2003.

[HBEV04] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*, Hiroshima, Japan, 2004.

[HIST03] A. Halevy, Z.G. Ives, D. Suciu, and I. Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In *Proceedings of the 12th Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003.

[HJ04] R. Huebsch and S.R. Jeffery. FREddies: DHT-Based Adaptive Query Processing via FedeRated Eddies. Technical report, Computer Science Division, University Of Berkeley, 2004.

[ILW+00] Z.G. Ives, A.Y. Levy, D.S. Weld, D. Florescu, and M. Friedman. Adaptive Query Processing for Internet Applications. *IEEE Data Engineering Bulletin*, pages 19–26, 2000.

[Ive02] Z.G. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, 2002.

[KAC+02] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proceed-*

ings of the 11th International World Wide Web Conference (WWW), Honolulu, Hawaii, USA, 2002.

[Kaz] The Kazaa file-sharing system. http://www.kazaa.com.

[KC04] G. Kokkinidis and V. Christophides. Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In *Proceedings of the International Workshop on P2P Computing and Database (P2P&DB)*, Heraklion, Crete, 2004.

[KCPA01] G. Karvounarakis, V. Christophides, D. Plexousakis, and S. Alexaki. Querying RDF Descriptions for Community Web Portals. In *17ie'mes Journees Bases de Donnees Avancees (BDA'01)*, Agadir, Maroc, 2001.

[Kos00] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[KP04] G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges, 2004. Unpublished manuscript.

[KS00] D. Kossmann and K. Stocker. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Transactions on Database Systems*, 25(1), March 2000.

[KSDC05] G. Kokkinidis, L. Sidirourgos, T. Dalamagas, and V. 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*, Schloss Dagstuhl, Germany, 2005.

[KW01] A. Kemper and C. Wiesner. HyperQueries: Dynamic Distributed Query Processing on the Internet. In *Proceedings of the 27th Conference on Very Large Data Bases (VLDB)*, Rome, Italy, 2001.

[LPR98] L. Liu, C. Pu, and K. Richine. Distributed Query Scheduling Service: An Architecture and Its Implementation. In *International Journal of Cooperative Information Systems (IJCIS)*, 1998.

[MAA+03] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F.D. Ngoc. Exchanging Intensional XML Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, 2003.

[MACP02] A. Magkanaraki, S. Alexaki, V. Christophides, and D. Plexousakis. Benchmarking RDF Schemas for the Semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC'02)*, 2002.

[Mor] The Morpheus file-sharing system. http://www.morpheus.com.

[MTCP03] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. In *Proceedings of the 2nd International Semantic Web Conference (ISWC)*, 2003.

[Nap] The Napster file-sharing system. http://www.napster.com.

[NK01] Z. Nie and S. Kambhampati. Joint Optimization of Cost and Coverage of Query Plans in Data Integration. In *Proccedings of the 10th International Conference on Information and Knowledge Management*, Atlanta, Georgia, USA, 2001.

[NWS+03] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and Aa Loser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In *Proceedings of the 12th Conference on World Wide Web (WWW)*, Budapest, Hungary, 2003.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[PI04] F. Pentaris and Y. Ioannidis. Distributed Query Optimization by Query Trading. In *Proceedings of the 9th International Conference on Extending Database Technologies (EDBT)*, Heraklion, Crete, Greece, 2004.

[PLP02] H. Paques, L. Liu, and C. Pu. Ginga: A Self-Adaptive Query Processing System. In *Proceedings of the CIKM*, Virginia, USA, 2002.

[PMT03] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for p2p systems. In *Proceedings of the CIDR'03 International Conference, Asilomar, CA, USA*, 2003.

[RDFa] RDF/XML Syntax Specification. http://www.w3.org/TR/rdf-syntax-grammar/.

[RDFb] RDF Primer. http://www.w3.org/TR/rdf-primer/.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM Conference*, San Diego, California, August 2001.

[RSWB05] P. Rosch, K. Sattler, C. Weth, and E. Buchmann. Best Effort Query Processing in DHT-based P2P Systems. In *Proceedings of the ICDE Workshop NetDB*, Tokyo, Japan, 2005.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Boston, 1979.

[Sah02] A. Sahuguet. *ubQL: A Distributed Query Language to Program Distributed Query Systems*. PhD thesis, University of Pennsylvania, 2002.

[SAL+96] M. Stonebraker, P.M. Aoki, W. Litwin, A. Preffer, A. Sah, J. Sidell, C. STaelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, (5):48–63, 1996.

[Ser05] G. Serfiotis. Optimising and Reformulating RQL Queries on the Semantic Web: The ICS-FORTH SWIM. Master's thesis, University of Crete, 2005.

[SHB04] H. Stuckenschmidt, R. Houben, and J. Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In *Proceedings of the 13th Conference on World Wide Web (WWW)*, New York, USA, 2004.

[SMGC04] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-organizing XML P2P Database System. In *Proceedings of the International Workshop on P2P Computing and Database (P2P&DB)*, Heraklion, Crete, 2004.

[SMK+01] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference*, San Diego, California, August 2001.

[SSDN02] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP-Hypercubes, Ontologies and Efficient Search on P2P Networks. In *Procceedings of the International Workshop on Agents and Peer-to-Peer Computing*, Bologna, Italy, 2002.

[Stu04] H. Stuckenschmidt. Similarity-Based Query Caching. In *Proceedings of the 6th International Conference on Flexible Query Answering Systems (FQAS 2004)*, Lyon, France, 2004.

[TDL04] J. Tian, Y. Dai, and X. Li. SemanticPeer: An Ontology-Based P2P Lookup Service. *Lecture Notes in Computer Science*, (3032):464–467, 2004.

[TP03] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *Proceedings of the Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P), Collocated with VLDB '03*, 2003.

[TSBN04] U. Thaden, W. Siberski, W.T. Balke, and W. Nedjl. Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks. In *Proceedings of the International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, 2004.

[TXKN03] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards High Performance Peer-to-Peer Content and Resource Sharing Systems. In *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*, January 2003.

[VP04] P. Valduriez and E. Pacitti. Data Management in Large-scale P2P Systems. In *Proceedings of the 6th International Conference on High Performance Computing in Computational Sciences (VECPAR)*, Valencia, Spain, June 2004.

[YGM03] B. Yang and H. Garcia-Molina. Designing a Super-Peer Network. In *Proceedings of the 19th International Conference Data Engineering (ICDE), IEEE Computer Society Press*, Los Alamitos, CA, 2003.

[Zho03] Y. Zhou. Adaptive Distributed Query Processing. In *Proceedings of the VLDB 2003 PhD Workshop*, Berlin, Germany, September, 2003.