Comprehensive Summaries of Uppsala Dissertations
from the Faculty of Science and Technology 901

Query Processing for Peer Mediator Databases

BY

TIMOUR KATCHAOUNOV

ACTA UNIVERSITATIS UPSALIENSIS
UPPSALA 2003

Dissertation at Uppsala University to be publicly examined in Siegbahnsalen, Ångström Laboratory, Tuesday, November 11, 2003 at 13:00 for the Degree of Doctor of Philosophy. The examination will be conducted in English

Abstract
Katchaounov, T. 2003. Query Processing for Peer Mediator Databases. Acta Universitatis Upsaliensis. *Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 901. 73 pp. Uppsala. ISBN 91-554-5770-3

The ability to physically interconnect many distributed, autonomous and heterogeneous software systems on a large scale presents new opportunities for sharing and reuse of existing, and for the creataion of new information and new computational services. However, finding and combining information in many such systems is a challenge even for the most advanced computer users. To address this challenge, mediator systems logically integrate many sources to hide their heterogeneity and distribution and give the users the illusion of a single coherent system.

Many new areas, such as scientific collaboration, require cooperation between many autonomous groups willing to share their knowledge. These areas require that the data integration process can be distributed among many autonomous parties, so that large integration solutions can be constructed from smaller ones. For this we propose a decentralized mediation architecture, peer mediator systems (PMS), based on the peer-to-peer (P2P) paradigm. In a PMS, reuse of human effort is achieved through logical composability of the mediators in terms of other mediators and sources by defining mediator views in terms of views in other mediators and sources.

Our thesis is that logical composability in a P2P mediation architecture is an important requirement and that composable mediators can be implemented efficiently through query processing techniques. In order to compute answers of queries in a PMS, logical mediator compositions must be translated to query execution plans, where mediators and sources cooperate to compute query answers. The focus of this dissertation is on query processing methods to realize composability in a PMS architecture in an efficient way that scales over the number of mediators.

Our contributions consist of an investigation of the interfaces and capabilities for peer mediators, and the design, implementation and experimental study of several query processing techniques that realize composability in an efficient and scalable way.

Keywords: data integration, mediators, query processing

Timour Katchaounov, Department of Information Technology. Uppsala University. Box 337, SE-75105 Uppsala, Sweden

ISBN 91-554-5770-3
ISSN 1104-232X
urn:nbn:se:uu:diva-3687 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-3687)

To my wife Adela.

Acknowledgements

The person I owe most for the completion of this dissertation and from whom I learned most about research is my advisor Tore Risch. My gratitude and appreciation to him for his constant help and energy, being always available for an advice, a tough discussion, or even a bug-fix. I would like to thank my opponent Tamer Özsu and the dissertation committee Nahid Shahmeri, Per Svensson, and Per Gunningberg who generously gave their time and expertise to evaluate this work.

I would also like to thank Marianne Ahrne who proof-read parts of my dissertation. Marianne, Eva Enefjord, and Gunilla Klaar were of great help in many organizational issues.

My former advisors and mentors Vanio Slavov and Vassil Vassilev not only taught me the foundations of Computer Science but also encouraged me to pursue a doctoral degree. Stefan Dodunekov suggested that I apply for a doctoral position in Sweden.

Special thanks to my fellow graduate student and friend Vanja Josifovski who helped me with my first steps in the AMOS *II* code, and with whom we co-authored several papers. Vanja was also very helpful in finding my first database job at the IBM Silicon Valley Lab.

Many thanks to the former members of EDSLAB Jörn Gebhardt and Hui Lin for becoming my friends and making my stay in Linköping more enjoyable. Thanks also to all current members at Uppsala Database Lab whose constant questions pushed my understanding to the limits and helped me clarify many research and technical issues.

While in Uppsala I was lucky to meet Monica, Zeynep, Brahim, Karim, Elli, Lamin and Russlan who became my friends for life. Thanks to them now I see the world from a much wider perspective. Mo, Elli and Brahim, some of my best time in Sweden was when we shared a place together.

During my first months in Sweden I was very happy to meet again Plamen, years after school. Thanks to him and Pepa I got in touch with the Bulgarian group in Uppsala who were always ready to help.

I am grateful to all my old friends from Bulgaria, they made me understand that home is where one is loved, so home for me is wherever they are.

Most of all I owe the inspiration to follow a career in research to my parents. They, and my brother, always encouraged me to follow new challenges. It is

thanks to their unconditional love and support I managed not to give up at times of deep home-sickness and I stayed to complete this project. My gratitude and love to all of you.

My dear Adi, I had to go all the way to Sweden so that we can meet. So I believe that the stress and the many lonely evenings you had to endure during the last two years were a natural part of our being together. I will never be able to thank you enough for your love and patience during this time and I look forward to our future together.

This work was funded by the Swedish Foundation for Strategic Research (contract number A3 96:34) through the ENDREA research program, and by the Swedish Agency for Innovation Systems (Vinnova), project number 21297-1.

List of Papers

This dissertation comprises of the following papers. In the summary of the disseration the papers are referred to as *Paper A* through *Paper F*.

[A] Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional data integration in a distributed mediator system. In *The Functional Approach to Data Management*. Springer-Verlag, 2003.

[B] Timour Katchaounov and Tore Risch. Interface capabilities for query processing in peer mediator systems. *Technical report 2003-048*, Department of Information Technology, Uppsala University, 2003.

[C] Timour Katchaounov, Vanja Josifovski, and Tore Risch. Scalable view expansion in a peer mediator system. In *Eighth International Conference on Database Systems for Advanced Application, (DASFAA'03)*, pages 107–116, IEEE Computer Society, March 2003.

[D] Vanja Josifovski, Timour Katchaounov, and Tore Risch. Optimizing queries in distributed and composable mediators. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, CoopIS'99*, pages 291–302, IEEE Computer Society, September 1999.

[E] Vanja Josifovski, Timour Katchaounov, and Tore Risch. Evaluation of join strategies for distributed mediation. In *5th East European Conference on Advances in Databases and Information Systems, ADBIS 2001*, volume 2151 of *Lecture Notes in Computer Science*, pages 308–322, Springer-Verlag, September 2001.

[F] Timour Katchaounov, Tore Risch, and Simon Zürcher. Object-oriented mediator queries to internet search engines. In *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 176–186, Springer-Verlag, September 2002.

Papers reprinted with the permission from the respective publisher:

Other papers and reports

In addition to the papers included in this dissertation, during the course of my Ph.D. studies I have authored or co-authored the following papers and reports listed in chronological order.

1. Hui Lin, Tore Risch, and Timour Katchaounov. Object-oriented mediator queries to xml data. In *Proceedings of the First International Conference on Web Information Systems Engineering, WISE 2000*, volume II, IEEE Computer Society, June 2000.
2. Timour Katchaounov, Vanja Josifovski, and Tore Risch. Distributed view expansion in composable mediators. In *Proceedings of the 7th International Conference on Cooperative Information Systems, CoopIS 2000*, volume 1901 of *Lecture Notes in Computer Science*, pages 144–149, Springer-Verlag, September 2000.
3. Krister Sutinen, Timour Katchaounov, and Johan Malmqvist. Using distributed database queries and composable mediators to support requirements analysis. In *Proceedings of INC0SE'2001*, 2001.
4. Hui Lin, Tore Risch, and Timour Katchaounov. Adaptive data mediation over xml data. *Journal of Applied Systems Studies*, 3(2):399–417, 2002.
5. Timour Katchaounov. Query processing in self-profiling composable peer-to-peer mediator databases. In *Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 627–637. Springer-Verlag, 2002.

Contents

Introduction

The pervasive use of wide-area computer networks and ultimately the Internet provides the capability to physically interconnect millions of computing devices[1]. However the nodes in such global networks are designed and evolve independently of each other which results in heterogeneity at various levels, starting from the hardware platforms and operating systems to the abstract models used to describe reality. The ability to physically connect distributed, autonomous and heterogeneous computing systems on a large scale presents new opportunities for better sharing and reuse of existing computational resources and information and for the creation of new computation services and new information from the combination of existing ones. One approach to realize these opportunities is to provide abstractions above the physical network in a separate layer called *middleware* that shields the users from various aspects of the heterogeneity and distribution in a global network.

A particular kind of middleware systems are *data integration* systems that address the problem of heterogeneity and distribution of large amounts of data in a computer network. The main purpose of data integration systems is to provide a logically unified view of distributed and diverse data so that it can be accessed without the need to deal with many systems, interfaces and syntactic and semantic data representations. The need for data integration occurs in many diverse contexts that vary in the degree of distribution and autonomy, the level of diversity of the data sources in terms of their data model and computational capabilities, the complexity of the modeled domain, the amount and dynamics of data, performance and data timeliness requirements, and the type of queries posed.

Various data integration solutions are suitable depending on the combination of values for each of these parameters. Database technology provides high-level abstractions of data, data retrieval, and manipulation operations. Naturally, the ideas from database technology are applied to the problems of data integration so that unified views of many data sources can be specified in terms of declarative query languages. Two main approaches exist for the design of data integration systems based on database technology - the *materialized approach* based on data warehouse technology and the *virtual ap-*

[1] According to the Internet Software Consortium (http://www.isc.org/) the number of hosts advertised in the DNS in January 2003 is 171,638,297.

proach based on the mediator concept. Data warehouse systems are centralized repositories where distributed data is collected, unified and stored in the same physical database, and is accessed without accessing the original data sources. Mediator systems [55] provide a logically unified view of the data sources (a virtual database) and the means to access and combine relevant data "on the fly" directly from the data sources. We describe the materialized and the virtual data integration approaches in more detail in Sect. 2.

Typically, database systems are designed to work in an enterprise context with a centralized organizational structure where scalability is been sought in terms of the data size or number of concurrent users. Since data warehouse systems are essentially traditional DBMSs and their main concept is that of a centralized data repository for all unified data, they are suitable mainly for centralized organizations. While the mediator approach itself does not imply a centralized architecture, most existing mediator systems have either centralized or two-tier architectures that make them suitable for the same type of centralized organizations as data warehouses.

However, due to the wide-spread use of computing technology and wide-area networks, the need for data integration and the opportunities it brings are relevant in many other social contexts than centralized organizations where database technology is commonly used. Some typical examples are scientific communities, alliances of companies, groups of individuals, to name a few. These social contexts are characterized by many independent and distributed units ready to share some of the data and services they own, so that when combined with other sources, new valuable information is produced. This information can be used by others either to satisfy their needs or to further integrate more data, services and information to provide higher-level integration services. Another important characteristic is the complexity and diversity of data in terms of its degree of structure. In contrast with traditional enterprise environments where data is well structured and mostly of a tabular nature easy to represent in terms of the relational data model, many new application areas need the integration of both complex and highly nested data such as product models, and of semi-structured data such as HTML or XML documents.

Based on these observations we conclude that there is a need for a new type of data integration systems based on database technology that are suitable for the sharing and integration of large number of autonomous, distributed and heterogeneous data sources and computation services with complex data. Such a system should fulfill several high-level requirements:

R1 (autonomy): The autonomous and distributed nature of the participating entities (e.g companies or research units) should be preserved because no one owns all data sources, and most likely no single entity has the knowledge how to integrate all data sources.

R2 (decentralization): There should be no need for centralized administration because in most cases no participant would like to relinquish control to someone else.

R3 (evolution): Each of the participants' knowledge and information needs may evolve at various rates, which requires that separate parts of the system evolve independently.

R4 (flexibility): It is hardly possible to predict all social contexts where data integration may be useful. Therefore a data integration system should lend itself to easy adaptation and customization by various types of users and in various social environments.

R5 (self-management): With a large number of autonomous participants, the cost of human maintenance of a large number of integrated views of many data sources can be prohibitively high. Therefore, a large scale data integration system should be able to maintain itself automatically, ideally with no human participation beyond the management of the data sources by their owners.

R6 (scalable integration): The process of data integration requires a lot of domain knowledge and is a complex and time consuming activity that will be mainly a human task in the foreseeable future. It is important that this process can be scaled to large number of autonomous sources.

R7 (abstraction): The heterogeneity of the data sources in terms of their data models and capabilities requires that a data integration system has powerful modeling capabilities so that it can represent and integrate the contents of diverse sources without loosing semantics.

R8 (scalable performance): Finally, and most importantly, a data integration system should provide high overall scalable performance in terms of both the number of nodes and data size.

While requirements *R1 - R7* are related to the high-level functionality and architecture (visible to its users) of a data integration system, the last requirement *R8)* is related to the internal implementation of such a system.

To fulfill requirements *R1 - R7* we propose a distributed mediator architecture based on the peer-to-peer (*P2P*) paradigm. The architecture is described in detail in Sect. 3. As proposed in [55], here mediators are relatively simple software modules that encode domain-specific knowledge about data and share abstractions of that data with other mediators or applications. Each mediator is a database system with its own storage manager, query processor and multi-mediator query language which can reference database objects in other

mediators. More complex mediators are defined through these primitive mediators by logically composing new mediators in terms of other mediators and data sources. Logical composability is realized through *multi-mediator views* defined in terms of views and other database objects in other mediators and data sources.

Many architectures are possible that fulfill the general requirements in one or another degree. It is hardly possible to show that one architecture is superior to another from the users' perspective. Most likely in the future there will be many P2P data integration systems that will differ in various aspects of their architecture. Only time and active usage in real-life problems can tell what is the right combination of features for a usable system. However, we believe that no matter what is the exact architecture, any such system will have to deal with the same fundamental problems with respect to the translation of logical mediator compositions into executable plans, that is *query processing*, which is the focus of this dissertation. No matter what is the particular architecture one of the most important issues for its usefulness is that of scalable performance. That is why our main goal is not the design of a complete architecture for mediation, but instead is the investigation of query processing techniques that are generic for such systems. The presented architecture provides the framework for the design and implementation of query processing techniques that will provide scalable performance and make the architecture useful in practice.

Therefore, at high-level, the research question we will address in this dissertation is: given an architecture that fulfills the general requirements *R1 - R7*, is it possible to design query processing techniques that will achieve high overall scalable performance in that architecture? This is a very general question that can be decomposed into many related sub-problems each having different answers depending on the particular architecture chosen and the requirements we put on a mediator system.

Our thesis is that logical composability in a P2P mediation architecture is an important requirement and that composable mediators can be implemented efficiently through query processing techniques.

In the rest of the dissertation we *i)* present a specific mediation architecture based on the P2P paradigm, *ii)* describe composability as the main requirement for the components in this architecture, *iii)* analyze several important problems related to processing queries in such an architecture, and *iv)* describe and evaluate experimentally the corresponding solutions which show that, indeed, it is possible to realize composability in a P2P mediator system with low overhead. The results are verified experimentally through an implementation of composable peer mediators in the AMOS II mediator database system.

Background

The title of this dissertation combines three independent concepts: *mediators*, *peer-to-peer systems* and *query processing*, that provide the foundation for our work. All three concepts have been extensively (re)defined and used in the literature in various senses. To provide a basis for the rest of our discussion, in this section we provide definitions of these concepts. As with most high-level architectural concepts, our definitions are necessarily informal.

To provide better understanding, we position the three concepts in a wider context. That is why first we discuss the area of data integration and the main approaches to implement data integration systems. Next we focus on the *mediator* approach to data integration as it is the basis for our work. Then we discuss *peer-to-peer systems*. Finally, we turn our attention to the area of *query processing*. Along with our main exposition we introduce several more related concepts that are used in the rest of our work.

2.1 Data Integration

The area of data integration is concerned with the problem of combining data residing in different autonomous sources and providing users with unified and possibly enriched views[1] of these sources, and the means to specify information requests that correlate data from many such sources. A *data integration system* provides the means to define such integrated views and to process information requests against these views. The purpose of data integration systems is to hide the complexity of many diverse sources and present to the users a single interface to the data in all sources. As illustrated with the cloud on Fig. 2.1, there is no specific architecture for data integration systems, nor is there one standard technology to implement such systems. However, for reasons we will describe below, the most common research approach is to use techniques from the database and knowledge management areas. General concepts and architectures related to data integration from the perspective of the database systems area can be found in [44], [53]. A recent overview of the theoretical aspects of data integration from a formal logical perspective can be found in [31].

[1] Here we use the term *view* in a general sense as the logical organization of the data the user sees.

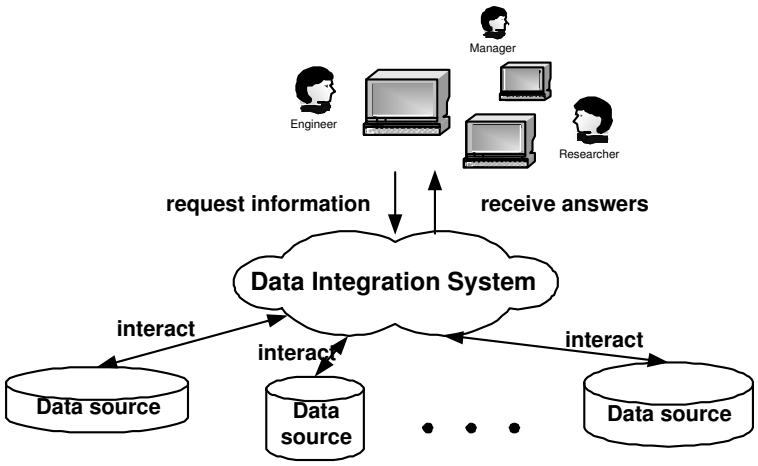


Figure 2.1: Data integration system

Data integration has long been important for decision support in large enterprises because of the benefits it can bring due to improved decision making. However, recently many more areas of human activity rely on information technology to create, store and search information, such as engineering, health care, scientific research, libraries, and personal uses. These application domains lead to several important characteristics of the data integration problem:

- The information needs of the users of an integrated system can be diverse and dynamic, and cannot be predicted in advance. For example a genetics researcher or a mechanical engineer would hardly know in advance the kind of information and the sources they need to access in order to solve some problem. This requires that data integration systems provide flexible means for the specification of information requests.
- Typically, the sources cannot be changed and may not be even aware of their participation in a data integration system. To take into account and integrate existing sources, data integration requires a bottom-up design approach that starts from the sources and incrementally constructs a unified view in terms of the sources' data.
- So far, the most common use of data integration systems is for information requests. There are several reasons for that. Typically, data integration is needed for decision-making which necessarily begins with request(s) for information and may (or may not) result in need for changes in the initial data. Many sources, such as most Web sources, provide read-only access. Finally, propagating updates to autonomous sources poses many hard problems related to their consistency.

2.1.1 Data sources

Since data sources are important in data integration, let us first look at what they are and what their properties are. *Data sources* are uniquely identifiable (in some scope) collection(s) of stored or computed data, called *data sets*[2], for which there exist programmatic access, and for which it is possible to retrieve or infer a description of the structure of this data, called *schema* and possibly additional information about the source. All the information about the contents of a source (its schema, data size, etc.), the computational capabilities of a source (the interface to access the data), and possibly other information about a source as reliability, information quality, etc., are collectively called *source meta-data*. Data sources may contain very large or even infinite amounts of data such as data streams from sensors or financial data, or results from computer simulations.

A data source can be anything from a file that is accessible via the file system API of an operating system, a Web page accessible through a Web server via the HTTP protocol, a CAD simulation accessible through a CORBA interface, to a complex database managed by an RDBMS accessible through an ODBC driver. From our definition it follows that in general a data source cannot be identified neither with one single software component, nor with a single storage element. Therefore a data source is defined by the combination of a software component and the data (stored or computed) that it provides access to. Given the practically unlimited number of ways to combine various technologies to access data, describe and store data, the concept of a data source is a loose term and in some cases it can be hard to decide precisely what constitutes one data source.

An important aspect of data sources is that there is no single generic method to retrieve data source schemas, and to associate a schema with a source. Some sources such as RDBMS may store and provide the source schema as part of the data source itself but separately from the actual data. In other cases, such as XML and RDF documents, the data sets in a source (in this case called *documents*) may be self-descriptive and schema information may be embedded inside the data sets. Finally, some sources as Web pages may not provide any schema at all, but methods can be developed to analyze the data and extract its structure.

Data sources may share the same type of interface and/or system to access their data but differ in terms of their contents. Thus it is important to distinguish between types of data sources and data source instances. For example all Oracle DBMSs are the same type of data source, however each particular installation of the DBMS is a different instance of the Oracle DBMS. Due to

[2]Here we use the term *set* in an informal sense. Formally speaking data sets can have either set or multi-set semantics.

the many possible combinations of common and different features of all potential data sources, it is not always possible to clearly separate between data source types and instances. For example a feature that is common only for a small group of data source instances may be considered as that group's characteristic and used to distinguish this group of sources as a new kind of sources. On the other hand, such an approach may result in an unmanageable number of source types. As in other modeling problems, it is up to the designer of a data integration system to decide which sources constitute a type of their own.

From this discussion we can derive several important properties of data sources - heterogeneity, autonomy and distribution.

- **Heterogeneity.** Data sources may be heterogeneous at many levels. Based on [43] we distinguish three general levels of heterogeneity:
  - **Platform heterogeneity.** At this level sources differ in the operating system and hardware they use, physical representation of data, methods to invoke the functions that provide programmatic access to the source's data, network protocols, etc.
  - **System heterogeneity.** At this level data sources differ mainly in two aspects. Data sources may use different sets of concepts, called *data models* to model real world entities. A variety of methods may be used for data access and manipulation. The collection of methods to access and manipulate data in a source is called *source capabilities*. Source capabilities may vary from a query language like SQL to a sequential file scan. Corresponding to our description of data sources, system heterogeneity is related to types of data sources.
  - **Information heterogeneity.** This level of heterogeneity relates to the data itself, that is to the data source instances. Their contents can differ at a logical level, because there exist many ways to model the real world. The resolution of this type of heterogeneity is called *schema integration*. Various taxonomies have been proposed to classify the differences between source instances at the logical level [58, 28, 43, 27, 22, 47]. Most works agree on two main types of information heterogeneity: semantic and structural heterogeneity. Different real-world concepts can be related to different concepts at the data source level which leads to *semantic heterogeneity*. Semantic heterogeneity manifests itself, for example, in different names for the same thing or the same name for different things, or using different units and precision. *Structural heterogeneity* (also called *schematic*) is related to the use of different concepts at the data model level, such as: different data types, objects vs. types, or types vs. attributes to model the same real-world entities.
- **Autonomy.** Because of organizational or technical reasons, data sources are usually independent and even not aware of each other. This independence is referred to as *autonomy*, which is related to the distribution of control (and

not data) [44]. In the organizational sense autonomy means that sources are controlled by independent persons or groups. In its technical sense autonomy is related to distribution of control [45]. Various overlapping definitions of autonomy are given in the literature that reflect its different aspects. In [41] node autonomy is classified in several types: *naming autonomy* relates to how nodes can create, select and register names of system objects, *foreign request autonomy* reflects the freedom a node has if and how to serve external requests and with what priority, *transaction autonomy* describes the ability of a node to choose transaction types and to choose when and how to execute transactions. In addition [41] recognizes heterogeneity as a type of autonomy - that is the autonomy in the choice of data model, schema, interfaces, etc. In [11] autonomy is defined as *design autonomy* - the freedom to choose data model and transaction management algorithms, *communication autonomy* - the ability to make independent decisions on what information to provide to external systems and when, and *execution autonomy* - any system can execute local transactions in any way it chooses. Another important facet of autonomy is the independent lifetimes of data sources, called *lifetime autonomy*.

- **Distribution.** Typically data sources reside on different computer nodes and thus are naturally distributed. As in [44] we use the term distribution with respect to data. However data sources may not only store but may also compute data. Thus, the distribution aspect of data sources is related to both data and function distribution, rather than just distribution of stored data.

Thus, data integration has to solve a wide variety of problems ranging from access to the data, unification of the data at various levels of abstraction, extraction of meta-data, and correlation of data items from disparate sources, to name a few. Naturally, all these operations have to be performed within reasonable time and resource limits, and therefore a major issue for any data integration solution is performance and scalability both in the data size and number of sources.

2.1.2 General approaches to data integration

Computer networks and network protocols allow to bridge the distribution gap between many data sources. However, networks only allow to bring data together and possibly unify it at the lowest physical level of representation (such as byte order). Thus we consider networks as an enabler for other technologies that can solve the problems brought by the heterogeneity, autonomy of data sources and the performance requirements for their integration.

Standards.

Standards are only a partial solution for heterogeneity. They can be applied only in well-defined domains where consensus can be reached about data representation and programming interfaces to data. It is hardly possible to foresee and standardize all possible ways in which data sources may be combined, thus even in a single domain, if standards are achieved, there are many aspects that cannot be fully standardized, for example the way people understand and model the world. Also standards often evolve and even compete, thus there is often the need to align different standards and to update systems with support for new standards which may be very costly and difficult. That is why even if standards can be enforced, there still will be heterogeneity of data sources at many levels.

Middleware.

One possible solution to the data integration problem is to migrate all disparate systems to one homogeneous, possibly distributed system. This is hardly a viable alternative as it may require all software at the data sources and all their applications to be rewritten and all data source owners to reach consensus about data representation and system interfaces.

Because of these mainly organizational reasons, data integration problems require solutions that do not interfere with the data sources and do not require changes of the data sources. To address this requirement, many data integration solutions introduce a unifying software layer called *middleware* [5]. Middleware is a very broad term used for a very wide spectrum of software systems and technologies. The goal of middleware technologies is to provide a degree of abstraction that hides various aspects of system heterogeneity and distribution. While many middleware technologies are not designed specifically to solve data integration problems, they can be applied for data integration either directly or as parts of more complex solutions.

Distributed object technologies.

One type of widely used middleware are distributed object frameworks such as CORBA [45], DCOM [33], Java RMI, and Web services [50]. All distributed object technologies have several features in common. They provide a general purpose way to specify procedural interfaces to some computation services and transparent access to remote objects. These technologies are concerned with the ability of distributed heterogeneous systems to transparently invoke each other's services and exchange data (often in the form of objects) across heterogeneous platforms and languages with different type systems. However, distributed object technologies are not concerned with how to efficiently compose distributed services and leave this task to the programmer. Since distributed object technologies are based on general-purpose procedural lan-

guages (typically object-oriented), their direct application to data integration has the following problems: *i)* they do not provide high-level constructs for the integration of many data sources and require "manual" programming to encode the transformation and combination of data from many sources, *ii)* every time when new information need arises or a new data source has to be added the middle object layer has to be changed, which may require a lot of (re)programming, *iii)* they do not expose the implementation of the services which prevents global optimization of composed services (e.g. a Web service that uses other Web services), and *iv)* it is infeasible to perform such global optimizations of composite services even if their implementation is available. This makes the direct application of general purpose distributed object management technologies unsuitable for the integration of many data sources especially in cases when the sources contain large amounts of dynamic data, and changing user information needs. Thus, distributed objects are enabling technologies on top of which more advanced solutions can be built.

Database technology.

A natural choice of technology for data integration are database management systems (DBMS) [17]. Database technology presents a high level of abstraction of large data sets and the operations to manage and query such data sets through declarative interfaces. Query languages and standardized data models allow the implementation of scalable and flexible systems that can manage and access very large data sets with very little programming effort compared to procedural frameworks.

However, database technology has been developed to manage homogeneous data sets (using the same physical and logical organization) that are fully controlled by a DBMS and therefore are not autonomous. For reliability and performance reasons DBMS technology has been extended to manage distributed data. Still, distributed DBMSs (DDBMS) are homogeneous systems that consist of the same type of nodes that operate as one system, and therefore neither the nodes of a DDBMS nor the data it manages are autonomous.

In order to be applicable to data integration problems, database technology has been extended and modified in various ways to support heterogeneity and autonomy. An exhaustive discussion of the architectural alternatives for database systems depending on the degree of autonomy, distribution and heterogeneity is given in [44]. Reference [9] provides an overview and classification of approaches to querying heterogeneous data sources along several other architectural dimensions. Here we overview in the following two sections the two most popular approaches to data integration middleware based on database technology - data warehouse and mediator systems.

2.2 Data Warehouses.

One possibility to integrate data from many sources is to extract data of interest from the sources, transform that data into a uniform representation and then load it into a central repository, a *data warehouse*, that provides uniform access to the integrated data. This approach is often called *materialized* because it physically materializes the integrated view by copying transformed data from the sources. Due to the maturity and wide use of relational database technology, it has been the primary choice to implement data warehouse systems. Data warehouses are built as a subject-oriented databases that are specialized in answering specific decision-support queries. This approach allows for avoiding the replication of all data from all sources which often may be infeasible or even impossible, and allows for the fine-tuning of a database for complex ad-hoc decision-support queries. A simplified architecture of a data warehouse is shown in Fig 2.2.

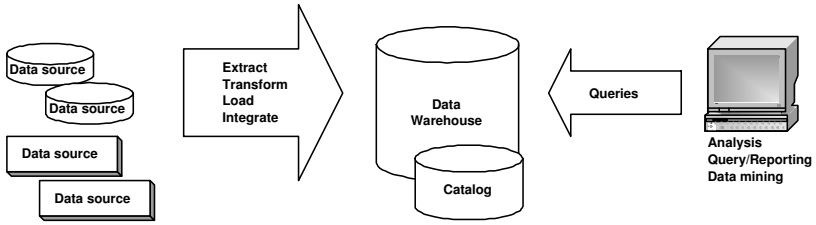


Figure 2.2: Simplified data warehouse architecture

A data warehouse integrated schema is first designed that logically integrates the data sources. The most common type of data sources are *operational databases*, that is, relational DBMS used for the day-to-day operation of an enterprise, tuned for on-line transaction processing (OLTP). Other types of data sources can be used as well, such as Web pages, specialized biological and engineering databases. To populate a data warehouse, the data is first extracted from multiple data sources. Then the data has to be cleaned, that is, anomalies such as missing and incorrect values are resolved, and transformed into uniform format. After extraction and cleaning the data is loaded into the warehouse. During loading data can be further processed by checking integrity constraints, sorting, summarization and aggregation. Thus data loading materializes the integrated views defined during the design phase of a data warehouse. To support decision-making data warehouses are designed to store historical data that is, organized in predefined dimensions that correspond to subjects of interest. Periodically the data warehouse is refreshed by propagating changes in the sources to the warehouse database. The process of loading and/or updating a data warehouse often may take many hours or even

days. That is why data warehouses are refreshed from time to time (once a day, or even once a week) and the users do not have access to the most recent data. Since a data warehouse has to accommodate all data of interest from the sources for long periods of time, its design requires very careful planning in advance both of its logical and physical organization, which can be a very time-consuming and complex process. A detailed overview of data warehouse technology can be found in [8].

2.3 Mediator Database Systems

An alternative to the data warehouse approach is to keep all data at the sources and access the sources on per-need basis to retrieve and combine only the data that is relevant to a request. For that, an intermediate software layer is introduced that presents to the users a logically integrated view of the data sources. Since this integrated view is not materialized explicitly by the user, this approach to data integration is often called *virtual*.

The requirements for the functionality, interfaces, and architecture of a virtual integration layer are analyzed in [55], and based on this analysis an architecture for a *mediation* layer is specified, illustrated on Fig. 2.3. A mediator layer is a virtual middle layer that separates the functions related to data integration from the data management functions of the data sources and the presentation functions of the applications. The goal of this layer is to simplify, abstract, reduce, merge, and explain data. It consists of *mediator* modules, defined in [55] as "a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications".

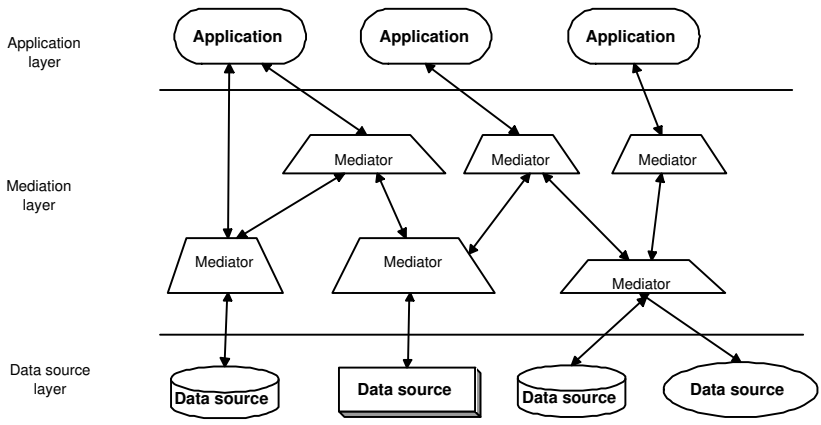


Figure 2.3: Mediation architecture

The mediation architecture is targeted at the integration of large number of autonomous and dynamic data sources that are typically available on the Internet or other wide-area networks. In this environment, maintainability is of uttermost importance. For better maintainability, a mediation layer is designed in a modular way and consists of a network of small and simple mediator modules specialized in some domain. Thus every mediator can be maintained by one domain expert or a small group of experts. Mediators share their abstractions with higher levels of mediators and applications which can use the domain knowledge encoded in lower-level mediators. Applications and mediators that require information from different domains use one or more other specialized mediators. Each mediator presents its own integrated view of some sources and mediators and thus adds more knowledge to the mediator network. An important consequence is that there is no single global view of all sources. There may be a large number of mediators to choose from. To facilitate knowledge reuse and discovery, mediators should be *inspectable* and provide data about themselves. A logical application of mediators is to use some of them as *meta-mediators* that facilitate the access to mediator and data source metadata. According to [56] the main tasks of a mediation layer, called *mediation services*, are:

- accessing and retrieving relevant data from multiple data sources,
- abstraction and transformation of the retrieved data into a common representation and semantics,
- integration and matching of the homogenized data,
- reduction of the integrated data by abstraction

Since mediators do not store the source data themselves, all functions related to data access, integration and delivery have to be performed dynamically "on-the-fly".

The concept of a mediator does not prescribe a particular implementation technology. However, as indicated in [55], a declarative approach to mediator design can bring the necessary maintainability and flexibility required for the integration of large number of dynamic sources. In particular mediators should support declarative interfaces to the applications and other mediators. Most practical implementations of mediator systems are based on database technology. For such mediator systems we use the term *mediator database systems (MDS)*. Below we will focus on mediator database systems and will use common database terminology to describe the structure and operation of mediator database systems.

Data integration in an MDS is performed in two main stages. The first stage, *data model mapping*, specifies how to retrieve data from each of the sources and how to convert the source data to the data model of the mediator system. This step deals with system heterogeneity, and provides a uniform representation of all data sources in terms of the mediator data model, called

the *common data model (CDM)*. The second stage, *schema integration*, deals with the information heterogeneity of sources' data on a logical level. During this stage identical objects in different sources are matched and schema and data instance conflicts are resolved. Since all sources' data is mapped to the mediator CDM, at this stage the CDM and the mediator query language can be used to define database views that logically unify the data sources.

Thus the data model and the query language of the mediator serve as the single interface to all integrated sources. User's information requests are then expressed in terms of the mediator query language. The actual retrieval and transformation of data from the sources is typically performed on demand when users pose queries to the integrated schema of the MDS. Other modes of data delivery are possible such as publish/subscribe, push, and broadcast [36].

These two integration stages often require very different approaches. As pointed out in Sect. 2.1 the data sources may present extremely diverse interfaces to their data and use very different data representations. This often requires a Turing-complete programming language to be used to specify the access to the sources and the required low-level data transformations. On the other hand, once the data has been transformed into a CDM and can be manipulated by a query language, semantic transformations can be specified declaratively.

Based on this two-phase integration approach, mediator systems are usually organized into two architectural tiers, each responsible for some of the tasks specific for the mediation layer. The first tier is typically responsible for the data model mapping phase. It is usually implemented as software components, called *wrappers*, that implement a uniform programming interface which hides all access details to the sources. Typical wrapper functions are retrieval of source data and its translation into the mediator CDM, access to (or inference of) source meta-data and statistics. The second tier, usually called the *mediator tier*, provides conflict resolution primitives across multiple sources. These primitives can be expressed in the query language of the mediator system, because the data from all sources is translated in the data model of the mediator by the wrappers. This two-tiered architecture is often referred to as the *mediator-wrapper* approach.

Notice, the term "mediator" was used in two senses - denoting the general mediator concept as presented in [55], and denoting only the mediator tier of an MDS. In addition projects such as TSIMMIS [13] and AURORA [58] use the term mediator in the sense of the integration views defined in a mediator, while they use correspondingly the term *mediator template* or *mediator skeleton* to denote the mediator system itself. Other works do not specify the exact meaning of the term "mediator" and often use it in all three senses. We provide a precise definition of the mediator concept as we use it in in this work in Sect. 3.4.

At the semantic level of data integration there are two distinguished approaches to logically specify the relationship between a mediated schema and the schemas of the data sources. In the first approach the integrated (also called "global") schema is described as views in terms of the local schemata of the sources. This is the approach known as *global-as-view (GAV)*. As opposed to GAV, the second approach first defines a global integrated schema. Then the contents of the sources is defined as views over this global schema. This approach is known as *local-as-view (LAV)*, since the source schemata is expressed as views in terms of the global view. An overview and comparison of the two approaches can be found in [32, 52].

Very few systems fully implement the general mediator architecture described here. Most such systems are either centralized or have a fixed 2- or 3-tier architecture. Furthermore, most such systems provide read-only access to the data sources.

One of the advantages of using database technology as a basis for the implementation of mediator systems is that much of the research and practice in the database area can be reused. Since both the integrated views and the user information requests are expressed in terms of a query language, the area most important to mediation is query processing. We discuss the general and the mediation specific concepts related to query processing in Sect. 2.5.

2.4 Peer-to-peer Systems

According to the Oxford English Dictionary the primary meanings of the word *peer* are "1. An equal in civil standing or rank; one's equal before the law. 2. One who takes rank with another in point of natural gifts or other qualifications; an equal in any respect". The concept of *peer-to-peer (P2P)* is a general software architecture paradigm at the same level of abstraction as client-server computing. Systems with P2P architecture consist of software components, called *peers*, that share and use each other's resources to perform a common task. The shared resources can be computing power, storage space, bandwidth, and even human presence. Two recent overviews of the general aspects of P2P and of the most popular P2P systems can be found in [3, 40].

Due to its general nature, the concept of P2P systems has been understood and defined in various ways. Here we provide several recent definitions. The Intel P2P Working Group[3] defines P2P computing as "the sharing of computer resources and services, including the exchange of information, processing cycles, cache storage, and disk storage for files, by direct exchange between systems. P2P computing approach offers various advantages: (1) it takes advantage of existing desktop computing power and networking connectivity, (2)

[3] www.peer-to-peerwg.org

computers that have traditionally been used solely as clients communicate directly among themselves and can act as both clients and servers, assuming whatever role is most efficient for the network, and (3) it can reduce the need for IT organizations to grow parts of its infrastructure in order to support certain services, such as backup storage." According to [49], "P2P is a class of applications that takes advantage of resources - storage, cycles, content, human presence - available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers".

P2P systems are based on three fundamental principles [3]:

- **Resource sharing** requires that peers (some or all) share some of their resources with other peers.
- **Decentralization** means that a system consisting of many peers is not controlled centrally.
- **Self-organization** is required in view of decentralization so that autonomous peers can coordinate to perform global activities based on local shared resources.

Initially the term P2P has been used for distributed file sharing and simple keyword search, made popular by the Napster[4] system. While P2P is often considered equivalent to distributed file sharing applications used in systems such as Gnutella, Kazaa.

However many other systems targeted at different application areas fall into the P2P category. Distributed computing systems as SETI@home[5], Entropia[6] use P2P technology to share processing power resources. Such systems are useful for complex computational tasks that can be split into smaller ones and then distributed among available peers. Another application area is collaboration. Such systems allow users to collaborate, often in real time to perform a common task without relying on a central infrastructure. Popular applications are Jabber[7] for messaging, Groove[8] for combined messaging and document sharing, project management, etc. Another type of systems are P2P platforms such as JXTA[9] and FastTrack[10] that provide generic APIs to build P2P systems.

Technically, two general types of P2P architectures are distinguished: *pure* P2P systems do not have any centralized server or repository of any kind and all nodes are equal, while *hybrid* P2P systems employ one or more central

[4] www.napster.com
[5] setiathome.ssl.berkeley.edu
[6] www.entropia.com
[7] www.jabber.org
[8] www.groove.net
[9] www.jxta.org
[10] www.fasttrack.nu

servers, e.g. to obtain meta-data such as network addresses of peers, and/or have some nodes with special functionality. *Super-peer* architectures [59] are a kind of hybrid architectures with hierarchical organization, where groups of peers communicate with all other peers through super-peers.

2.5 Query Processing and Optimization

The main purpose of a mediator system is to retrieve, combine and enrich existing data through queries in a declarative language. Therefore one of the most important functionalities of a mediator is the ability to efficiently process queries. *Query processing* is a collective term that stands for all techniques used to compute the result of a query expressed in a declarative language. Usually query processing is performed in two distinct steps. *Query optimization* transforms declarative queries into an efficient executable representation called *query evaluation plan or query execution plan (QEP)*. *Query evaluation* takes a QEP and interprets it against a database to produce query results.

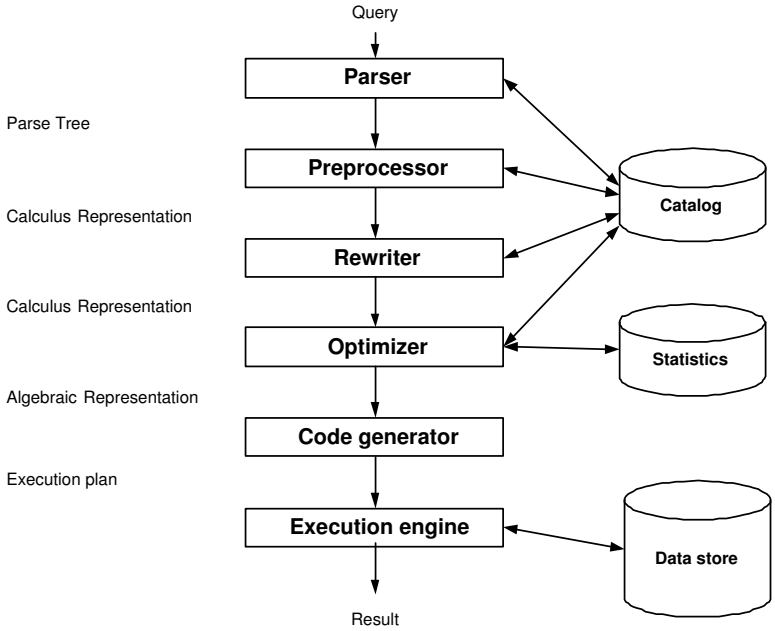


Figure 2.4: Simplified DBMS query processor

A simplified diagram of a DBMS query compiler is shown on Fig. 2.4. The *parser* checks input queries for syntactic correctness and translates them into an in-memory representation called *parse tree*. The parse tree is analyzed for semantic correctness by the *preprocessor*. The semantic analysis includes

checks such as whether relation and attribute names actually correspond to existing relations with corresponding attributes, whether all attributes and constants are type-compatible with their usage, etc. Semantically correct parse trees are translated into an internal representation. The actual compilation of the query begins with this internal representation.

In most modern database compilers [53, 44] query optimization is performed in two main stages each using a different internal representation of the query. The first, called *query rewriting*, is based on equivalent logical transformations of some kind of a calculus form of the query. A calculus is a non-procedural representation of the query where the desired result is expressed via a logical formula equivalent to some variant of predicate calculus. This phase is performed by the *rewriter*. The goal of the calculus-based rewrites is to simplify the query and to transform it into some normalized form suitable for subsequent optimization.

The next compilation phase, called *query optimization*, accepts a calculus query representation and transforms it into an equivalent algebraic form. This phase is performed by the *optimizer* which applies algebraic laws to produce a more efficient algebraic representation. An algebra is a formal structure consisting of sets and operations on the elements of those sets. For example relational algebra is a formal system for manipulating relations. The operands of relational algebra are relations. Its operations include the usual set operations (since relations are sets of tuples), and special operations defined for relations: selection, projection and join. Since algebraic operators have precedence and order of application of the operators, a query algebra is procedural in the sense that it prescribes how to construct the result of a query. Abstract algebraic operations may be implemented by various algorithms, each with different execution cost. To produce an optimal QEP, the query optimization phase usually searches the space of all logically equivalent algebraic expressions that compute a query, assigns to the logical operators all applicable algorithms and computes the cost of executing all the operators in the plan according to their order and chosen implementation. An algebraic representation of a query where the operators are associated with evaluation algorithms and cost functions for those algorithms is called *physical algebra*. In order to choose the best possible QEP the optimizer uses a cost model to evaluate the quality of each candidate plan. For this various measures can be used such as resource consumption or total execution time.

An optimal physical algebra expression where all algebraic operations are assigned an implementing algorithm can serve directly as a QEP of a query, and can be directly interpreted. Optionally there may be a final phase, performed by the *code generator*, that transforms the algebraic expression into some lower-level representation, e.g. CPU instructions.

Finally the resulting QEP can be executed by the *execution engine* or may

be stored for future use.

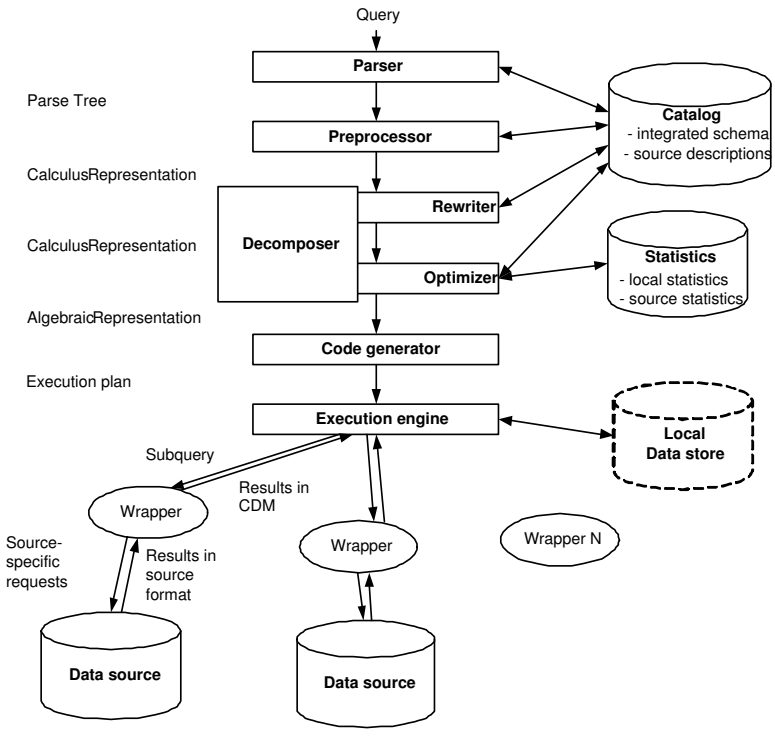


Figure 2.5: Simplified query processor of a mediator

Query processing for mediator systems introduces several more phases and components shown on Fig. 2.5. In the figure the additional components are denoted with continuous lines. A *decomposer* component identifies which portion of the original query can be computed by each of the sources. Decomposition is performed either by the rewriter or by the optimizer or both, depending on the particular mediator architecture. Decomposition is also present in query processors for distributed DBMS [44]. However, there the query processor needs not take into account the heterogeneous computing capabilities of the data sources. As a result of decomposition the original plan is split into *sub-queries*, each executable by one particular data source. The subqueries are then submitted to the corresponding wrappers, which in turn translate them into requests specific for the type of source accessed. The source return results in their format which are mapped to the mediator CDM by the wrappers and assembled by the execution engine as final results.

A P2P Architecture for Mediation

In this section we present a software architecture for the integration of many distributed and autonomous data sources that fulfills the need for scalable data integration in a distributed and autonomous environment. As in [14] by software architecture we mean "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time." Software architectures provide a high-level view of software systems that allows to define and reason about their general properties. We define our architecture in a top-down fashion. To fulfill requirements *R1-R7* defined in Sect. 1 we base our architecture on two fundamental ideas - mediators and peer-to-peer systems. These two design choices are justified in Sect. 3.1. For system architectures based on these two ideas we use the term *peer mediator system (PMS)* architecture. Next we identify some important requirements that stem from these design choices in Sect. 3.2. Then we identify and describe the components of a PMS and how they form together a PMS. Finally we discuss the interactions between the components that meet the architecture specific requirements in Sect. 3.2. The presented PMS architecture is partially implemented in the framework of the AMOS II mediator system and is described in *Paper B.*

3.1 Design Motivation

Mediators.

We base our architecture on the mediator approach because the materialized (data warehouse) approach is not suitable for our target environment for several reasons. With a large number of sources it may not always be practical or possible to materialize all data from all sources due to its volume or source limitations. On the other hand if only a subset of the data in the sources is to be materialized in a data warehouse, it may not be possible to know in advance what queries will be issued in the future and thus what data to materialize. With autonomous data sources it may not be always possible to access all their data due to security or request size restrictions. For sources that compute their data dynamically instead of storing it explicitly (such as simulations), the source contents may be infinite and therefore impossible to pre-compute and store. Even more, for sources that perform specialized computations it may not be

possible to pre-compute and store all possible results. Such sources require that results are computed on the fly depending on their input. Finally due to potentially very large amounts of data in the data sources, it may not be possible to update a data warehouse in a timely fashion, and to keep up data warehouse updates with the change rate of the sources, which would result in materialized views that are gradually getting older than the source data.

The mediator approach presents a solution to these problems. Since mediators are virtual databases that provide only a logically unified view of many data sources, all data access and transformations are performed on the fly during query execution. This allows to:

- avoid the storage and update overhead of the materialized approach,
- answer any query over a virtual database with the most current data from all sources without prior knowledge of a query workload,
- request only small portions of the sources' data that relevant to a query,
- optimize query execution plans so that the least possible amount of data is retrieved from the sources and transferred over the network,
- reuse the computational capabilities of the sources, such as feature extraction or pattern matching operations, and thus compute wider classes of queries,
- take into account source security and access limitations.

In addition, mediators may transparently materialize and cache intermediate results and some source data for improved performance.

Peer-to-peer architecture.

The mediator approach does not prescribe any particular architecture apart from that there is a distinct middle layer between the user applications and the data sources that separates the application logic from the data management logic at the data source layer.

The simplest possible approach to design a mediator layer is to encapsulate all mediation functionality in a single centralized mediator system. This is the approach typically used in most existing mediator systems. However, with a centralized approach only few of the general requirements *R1-R7* can be fulfilled. The main reason for that is the assumption that all integration can be performed by a single entity (individual or organization), which is not true in our target environment. On the contrary, when large numbers of distributed data sources are available, it will be most likely that many independent domain experts will have the knowledge of how to integrate only portions of the data in a subset of all sources. Therefore in a centralized mediation architecture:

- only the autonomy of the data source owners can be preserved, but not that of the domain experts that integrate the sources, who will have to reveal all their specialized knowledge to the mediator owner,
- data integration will require the coordinated access to and modification of a

one global virtual database schema by many independent entities, a process that may not scale over many sources and many participants,
- central administration is required and therefore coordination between many independent parties, which may not always be possible,
- it may be expected that a single mediator that covers many knowledge domains and accesses large number of sources of many different types will be extremely complex and hard to maintain,
- finally, a centralized mediator presents a single point of failure.

In summary a centralized mediator architecture requires central management and concentration of all knowledge required to integrate all sources in one place, something that is hardly possible on a large scale across geographic and organizational boundaries.

However, as envisioned originally in [55], a mediation system may in general consist of many distributed mediators specialized in some knowledge domain where each mediator integrates only a small subset of all available sources and shares its data abstractions with a higher level of mediators and applications, a vision that naturally maps into a P2P architecture. Autonomy and decentralization are inherent properties of the P2P paradigm for distributed computing, therefore one natural way to design a distributed mediation system that fulfills the organizational requirements for autonomy *(R1)* and decentralization *(R2)* is to design this system as a *peer mediator system (PMS)* consisting of autonomous mediator peers that interact with each other and with the data sources and user applications. Next we discuss some important requirements for a PMS that follow from the design choice of peer-to-peer architecture.

3.2 Requirements

Before describing our architecture for peer mediators, we first discuss the important requirements that peers participating in a PMS should meet. We divide these requirements into two groups. First are the ones that we address through the contributions presented in this dissertation. For completeness, we present a non-exhaustive list of additional requirements that are important for a successful implementation of a PMS, but are outside of the scope of this work. We consider the next three requirements to be fundamental for the realization of a PMS architecture which is why we chose to focus our work at their study and fulfillment.

Logical composability.

The main value of a PMS is in its ability to not only distribute the integration effort among many autonomous participants, but in that it provides the means to assemble integrated views of both data sources and other integrated views

and thus reuse human efforts and knowledge encoded in the mediators.

Two main approaches exist to realize compositions of distributed software components. One is through distributed technologies such as RPC, CORBA, or Web services. These approaches are procedural, require a lot of programming effort, are rather static, and result in more or less tightly coupled distributed systems that are hard to evolve. Therefore we do not consider these approaches to be directly suitable for dynamic systems such as PMSs. We term distributed systems that can interoperate through such procedural approaches as *physically composable*.

A much more flexible and scalable approach is to specify mediator compositions logically in terms of a declarative language. This requires that the peers in a PMS *i)* have a query language and a view definition mechanism that provides constructs to refer to both views and stored data in other mediators, that is define and access data in *global views (queries)* and *ii)* are able to share their views and stored data with other peers, that is define some schema objects as *public* and provide the means to access them. Having these two properties allows to transitively define arbitrary logical compositions of peers in terms of each other, a property we name *logical composability*.

Logical composability extends the concept of logical data independence in traditional databases across many distributed peers and allows peers to evolve without affecting each other as long as the view interfaces are kept intact. Another advantage of logical composability is that mediators can reuse indirectly abstractions exported by other peers without even knowing their existence which promotes reuse and autonomy.

Physical composability.

To realize logical composability it is necessary that peers are able to generate executable plans to compute the extensions of many transitively composed global views. That is, peers must be able to translate logical view compositions into physically composed access plans across many mediators and sources. In order for such plans to be executed peers must support programmatic interfaces to communicate over a network. These programmatic interfaces can be implemented via one or more of many available technologies for distributed interoperability [33], such as RPC, CORBA [45], DCOM, and more recently SOAP [2].

Location transparency.

Large number of computer nodes, typically used as “dumb” Internet clients, connect to the Internet via temporary connections and identify themselves through dynamic physical (IP) addresses (such as computers connected over a modem, LANs with DHCP, subnetworks behind NAT) that may change over time. Many of these nodes may host mediator peers managed by the node

owner(s) and possibly used by other such nodes. Due to the mobility of many computing devices, peer owners may migrate their peers from one node to another (e.g. when a peer has been moved from an office workstation to a portable node). To support such scenarios, peers should not be bound to physical addresses or to physical nodes. This requires that peers are somehow uniquely logically identified within a PMS in a way that allows to dynamically map logical peer identifiers to physical locations.

Logical identification of peers allows both users and peers to abstract from the physical network details. In order to be able to refer to remote peers by their logical identifiers, peers have to be able to perform *name resolution*, that is, map logical identifiers to physical addresses. For a PMS to scale in number of peers and users, name resolution must be performed in a fully automated and transparent manner that scales over large numbers of peers.

Requirements outside of the dissertation scope.

An implementation of the PMS architecture that would be useful in practice raises a number of additional problems that will not be addressed by this work. Below we discuss some of these problems that we consider to be important for a successful implementation of a PMS.

Information discovery: The task of identifying relevant sources of information is *information discovery*. These sources can be both other mediators that provide already existing abstractions of data sources and other mediators, or directly data sources. The result of information discovery consists of logical identifiers of peers and optionally additional meta-data about peer contents such as relation names and attributes, file names, functions, etc. Information discovery requires that mediator peers are able to store, exchange and query meta-data about other mediators and data sources, a feature described as inspectable mediators in Sect. 2. In a P2P architecture, information discovery poses additional performance problems since there is no central meta-data repository and thus large number of global meta-data requests may need to be processed. A related problem is that of *bootstrapping* a PMS with initial meta-data so that a set of disconnected peers can "learn" about each other and form a PMS together.

Schema integration: One of the most important problems in data integration in general is how to describe mappings between an integrated schema and the sources' schemas. In a PMS this problem is exacerbated by the potentially very large number of views distributed among autonomous mediators. Thus, a PMS requires information modeling concepts at the query language level that will provide the users with scalable tools to

easily integrate large number of sources. In addition tools and methods are necessary to perform schema integration in an (semi-)automated way.

Dynamic availability: Due to their autonomy, the peers in a P2P system may control their own availability independent of other peers. At the same time, on a global network some peers may become unreachable due to network problems or simply because the nodes they reside on were disconnected from the network. That is why peers should be able to join and leave a PMS at any time without disrupting the overall operation of the system. This requires a mechanism for the peers to detect each others' availability and gracefully react when some peers are not available. The most challenging problem here is to define the semantics of integrated views when some of the views' sources are unavailable and to process queries against such views in a way most suitable for the user.

Security: In a PMS system users may have conflicting interests and even malicious intentions. Thus, care should be taken in a PMS that users cannot access restricted information, tamper with information that travels through many peers, and disrupt the operation of the system as a whole. Two problems specific for a PMS architecture are, e.g.: *i)* a highly decentralized system catalog with security related information such as users, groups, passwords, keys and permissions may lead to performance problems, and *ii)* when integrated views are defined, it may happen that some global execution plans are non-executable due to local security restrictions which requires the query processor of a PMS to be able to take security restrictions into account.

3.3 External System Components

To describe our P2P mediation architecture we first analyze the types of software components that participate in a PMS. According to the conceptual mediation architecture described in Sect. 2.3, an integrated information system consists of three types of software components, each with specific purpose and functionality, divided into three layers: data sources, mediators and user applications. The data source components and their interfaces are given a priori. Many applications exist that derive their data from various information systems over standard interfaces. It is desirable that a data integration system provides the means to reuse such applications. Therefore the design of a data integration system can be viewed as a two step process. Existing sources and applications require a bottom-up step that puts some requirements on the mediation layer. After these requirements are stated, the design of the mediation layer itself can

be done in a top-down fashion. That is why we first observe and analyze the main properties of the components in the data source and the application layers which are external from the view point of the mediators. Then in Sect. 3.4 we define the internal architecture of the mediation layer so that it fits best our observations and requirements.

3.3.1 Data sources.

Section 2.1 defines a data source as a uniquely identifiable couple of a software component and its data where a method exists to acquire some source metadata that contains at least the source schema and possibly other information about the source. In this section we investigate in more details the properties important for data integration of the data source components.

Low-level interfaces.

Data sources provide access primitives that allow external components to invoke some computation at the data sources, and to send and receive data. The collection of all access primitives of a data source comprises its *low-level interface*. We distinguish two kinds of such interfaces. *Global data sources* support network-based interface(s) and are globally identifiable and globally accessible by remote systems over a network. Examples of global sources are Web sites, Internet search engines, Web services, LDAP and DNS servers, etc. *Local data sources* do not have globally unique identifiers and there is no method to access them by external components over a computer network. Typically local interfaces are provided in the form of call-level APIs. Examples of local sources are ODBC and JDBC sources, local files, and software components accessible via an API (e.g. a B-tree index library). To make local data sources globally accessible to all peers in a PMS, one or more mediator peers must serve as intermediary between the local source and the rest of the PMS.

The large number and diversity of the low-level interfaces to existing and future data sources, requires that a mediator system is easily extensible with new functionality for the access to a variety of sources.

Computational capabilities.

A higher level of abstraction above low-level interfaces are the data sources' capabilities which are related to, but often not equivalent to the low-level interface(s) supported by the sources. In fact the same capabilities may be accessible via different low-level interfaces, e.g. for RDBMS typically these are ODBC, JDBC, and a call-level API all providing access to the same functionality. Thus capabilities are not equivalent to interfaces. By capabilities we mean the abstract computations that a source can perform over some optional input data. Based on similarities and differences in their supported capabilities the data source components can be subdivided into four levels of abstraction.

- Type of source. This is the most general classification of data sources according to which all sources with the same set of capabilities are of the same kind. Some examples are all relational DBMSs that support the SQL'92 standard, or all installations of a particular DBMS like Oracle 9i or DB2 v7.2, or all installations of the Google search engine. All sources of these kinds have their own specific capabilities either by virtue of being instances of a particular software system or by fully implementing some standard. Typically such data source kinds will be defined by standards or by some well-known systems.
- Source instance. Many kinds of sources are customizable and extensible. Thus, particular source instances (typically represented by a system of some type being installed on a computer node) may differ in the functionality they provide. For example a relational database may contain special user-defined functions, created by its local administrator. Of course capabilities present in one or few source instances may gradually become adopted by a vendor, and then such group of capabilities may form a separate kind of sources.
- Schema instance. The above two classifications look at a source as a whole. It is possible that a source can perform certain computations over some of its data sets, but not over others. A typical example are Web forms where scans can be performed over some data sets (e.g. get all countries), other data sets may allow only selections (e.g. retrieve all cars of a specific make), while third ones may allow only joins (e.g. get all parts supplied by suppliers in Sweden). Thus, the capabilities of a source may change with respect to its current schema and are not inherent for the source instance. Such limitations may be due to only few queries being publicly accessible through a Web interface, or because the data access is hard-coded in some procedural language.
- Data instance. Finally at the lowest level of abstraction a source instance with particular schema may have varying capabilities depending on its current data contents. For example, if a Web form presents a choice of cities where users can look for housing, this page can be viewed as a source with two data sets - that of cities and of properties. However, the housing information that can be retrieved depends on the contents of the cities data set.

Given the wide variety of interfaces and capabilities of the data sources, one of the major problems for mediator systems is how to utilize existing capabilities over the available low-level interfaces, how to compensate for missing capabilities, and finally how to find sources with some specific set of capabilities, e.g. a matrix multiplication source or an image matching source. Solving this problem requires that mediators are able to represent in some way the capabilities of the sources they access. Ideally such a representation of capabilities should be easy to specify, query and manipulate both "manually" by humans

and automatically by the mediators so that both new kinds of sources and new source instances can be easily added, existing ones modified and queried for their capabilities.

Relationships between data sources.

Data sources and/or the data items in the sources can be interrelated in a variety of ways, the most common of which we discuss below.

- **Data ↔ meta-data**. One possible way to acquire source meta-data is to retrieve it from another source. An example of such sources are XML files with external DTD or XSchema descriptions, and Web services described in UDDI registries. Thus data sources may be related by a data - meta-data relationship. This relationship may be "known" to some of the involved sources (e.g. as a URI in an XML document that points to its DTD), to third source(s) or mediators, or to humans. To facilitate source discovery and automated integration meta-data sources should be accessible in the same way as other sources. To allow for uniform treatment of data and meta-data at any level, we do not distinguish meta-data sources from data sources, but we require that a mediator system can model this relationship in terms of its CDM.
- **Data ↔ index-data**. Sources may also serve as indexes to other source's data. One example are text document indexes that provide fast access to external documents either in a file system or on the Web. According to our definition of a data source, indexes can be considered as data sources of their own. In such case a relationship exists between the index source(s) and the data source(s) it indexes. For example the Google and AltaVista Internet search engines can be considered as indexes of most Web documents on the Internet. Knowledge of the relationship between index and data sources can be very important for the overall performance of a mediator system and can provide alternative more efficient access paths to external data. In the cases when a data source does not provide a "scan" interface, an index may be the only way to access the data in the source. Utilizing the index - data relationship is the only way to retrieve data from such limited sources.
- **Data ↔ nested data**. Certain data sources may have nested structure, that is, access and combine data from other data *sub-sources*. Due to the diversity of all possible types of sources it is very hard to automatically detect and model the structure of arbitrarily composed data sources. This may not be possible either because the sources do not contain information about their own structure or do not provide access to this information, or because of security and privacy restrictions. Therefore in most cases data sources can be considered to be *atomic* from the view point of an external system, that is their nested structure is "invisible" to a mediation system.
  However, such *compound* data sources may provide the means for external

systems to inspect their internal structure. Typically such sources would use a language to describe the composition of many sub-sources and would provide some way of retrieving definitions of source compositions. Examples of such sources are DBMS products with support for external sources in their data definition and query languages (e.g. the SQL/MED standard [39]). In other cases the source structure may be specified manually by a human. Either way a mediator system may benefit from the knowledge of the relationship between sources and sub-source(s) in two ways. If the sub-sources are directly accessible by an external system, then a mediator system may generate more efficient source access plans that bypass the container source and access the sub-sources directly. If the container source provides a language interface, then the mediator may generate more efficient requests in terms of the container source language, e.g. by combining multiple requests.

- **Inter-source semantic constraints**. The contents of data sources may be semantically related in various ways. A source may be a replica of another source, or there may be functional dependencies between sources. A mediator may utilize this knowledge to provide integrated views with richer semantics, to generate more efficient access plans to the sources and to generate integrated data with better quality.

3.3.2 Applications.

User applications send requests to the mediation layer on behalf of a user, and deal with the presentation of mediator replies to the user. By definition applications are not capable of processing requests by themselves.

Many applications or application development frameworks have been developed that provide advanced data analysis and visualization functionality, and support standard interfaces for data access. To utilize such legacy applications and frameworks a mediator system must be able to support some data access standards (such as ODBC/JDBC, EJB, etc.) and provide the means to be easily extensible with new interfaces.

Since these standard interfaces are not developed with any particular system in mind, they may not be suitable for future applications that would access integrated data through a mediation system. Standard interfaces suffer from several deficiencies: *i)* they already assume a predefined set of functionalities that may not be sufficient to express all capabilities of a mediator system, *ii)* they are based on data models that may not be expressive enough to translate all concepts at the mediator CDM, and *iii)* they may not provide the necessary level of performance. Therefore a mediator system should provide rich specialized interfaces for more effective and efficient access to the mediation layer by new applications. To support the needs of future applications, the

mediators should provide at least two types of specialized interfaces.

To allow arbitrary applications to access arbitrary mediators across the network in a flexible manner, the mediators may provide a low-level network interface directly based on some transport protocol as TCP/IP. Typically such interface would be implemented by advanced applications that support a mediator network protocol, need data processing functionality not present in the mediators and need to access more than one mediator. The advantages of a network interface are that it allows for loose coupling between the application(s) and the mediator(s) that is independent from programming languages, operating systems and hardware. However, such *global applications* require more intelligence built in them so that they can discover and communicate effectively and efficiently with many distributed mediators and combine the retrieved data. Thus, low-level application-to-mediator network interfaces would result in very complex applications that implement much of the functionality already present in the mediators.

In order to avoid such complex applications, all functions related to the retrieval and combination of data from many mediators can be delegated to a single, specially designed mediator that serves as the application's *gateway* to all other mediators. This approach allows applications to stay relatively simple and delegate all tasks related to the efficient access to many remote mediators to the gateway mediator. For this, a high-level function call interface is needed to provide future and existing applications with the ability for simple and distribution transparent access to mediators. Such an interface would provide the means for applications to be easily mediator-enabled either by directly embedding a mediator system in the application through an API or providing a high-level client-server interface. Applications that access a gateway mediator are called *local* because they are not aware of the distribution of the mediators and they typically access only one gateway mediator.

3.4 Mediator components and their functionality

In this section we describe the functionality and architecture of the mediator components which are the focus of our work. For the design of the mediator components of a PMS we follow the mediator-wrapper approach described in Sect. 2.3. We design each mediator as an autonomous extensible DBMS with a query language interface, a view definition facility, local persistent storage, its own catalog and query processor. All mediators share the same query language and data model, and are capable of processing queries in terms of this query language. A very important feature of the mediators is that they treat each other as data sources, which serves as the basis for mediator composability.

To define what is a mediator, we first distinguish a *mediator system* and a *mediator instance*. A mediator system is a software system represented by

program code and initial data necessary for the system to operate. A mediator instance is either a mediator system instantiated as a process on a computer node, or a mediator system that was executing on a computer node and which state was persistently stored so that the mediator instance can be fully restored. Thus a mediator instance would normally be a mediator system that is being used to integrate data sources, and contains integration views, and possibly other stored data and meta-data defined by the user(s). For short, we will use the term “mediator” in the sense of “mediator system”.

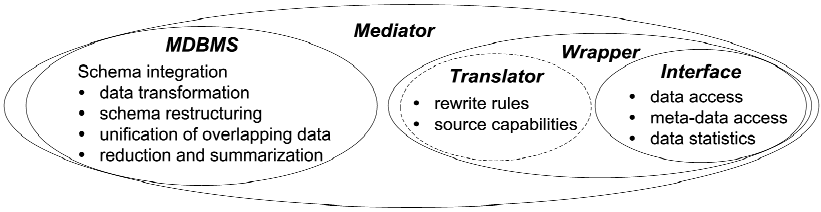


Figure 3.1: Distribution of mediator functionality across components.

At a high level, the mediators are divided into two architectural tiers: a *mediator DBMS (MEDBMS) tier* that is responsible for information integration and processing of user queries, and a *wrapper tier* responsible for source access[1]. A mediator consists of one MEDBMS, one wrapper for external mediators and any number of optional wrappers for other types of sources. Wrappers are designed in a generic way so that one wrapper can access multiple instances of the same data source type. For reusability, simplicity and flexibility of the mediators, the mediation functionality described in Sect. 2.3, p. 2.3, is distributed between the wrapper and the MEDBMS tiers as illustrated on Fig. 3.1. In the next two sections we describe the functionality of the two mediator tiers.

3.4.1 Wrappers

The wrapper components are responsible for data model mapping from the source’s data model to the mediator CDM. Unlike other mediator architectures [58, 51] wrappers are internal, non-autonomous components of a mediator, that are tightly connected to and controlled by the mediator. Thus wrappers are not components of a PMS by themselves and are “invisible” outside their mediators. Each wrapper component consists of two main sub-components - a *source interface* and an optional *translator*.

[1]Thus we resolve the first naming problem mentioned in Sect. 2.3, by naming the part of the mediator complementary to the wrapper as “MEDBMS tier” instead of using the overloaded term “mediator”.

The source interface provides functions to connect to sources of some type, access data and meta-data in the sources, manage session information, and, when possible, retrieve source statistics. The data access functions of a wrapper are responsible for sending input data to the data source, the invocation of some functionality at the data source, the retrieval of the resulting data, and transformation of that data into the mediator CDM. Source interface functions return to the MEDBMS data objects in terms of its own CDM. In addition, during data transformation the source interface component may perform various data cleaning and semantic enrichment tasks, such as replacing missing values with defaults, or inferring the type of retrieved data (e.g. recognizing strings as dates or numbers). Source interfaces hide only some of the system heterogeneity of the sources - that of their low-level interfaces.

As already mentioned in Sect. 3.3.1 many types of sources can still be heterogeneous in their computational capabilities. For such sources wrappers need a translator component that "knows" how to translate operations expressed in terms of the mediator query language into operations that can be computed by the corresponding type of sources. A translator consists of source capabilities descriptions and rewrite rules. Source capabilities roughly describe the operations that a source supports, while rewrite rules provide detailed translation of expressions in the mediator query language into requests or language expressions in executable the sources. Examples of data sources for which only a source interface is sufficient are storage managers such as BerkleyDB[2] which provides simple data access operations that can be easily mapped directly to operations in the MEDBMS.

An example of simple sources that require translation is a source that provides only range access via non-strict inequalities only. If queries to such sources require strict inequalities, the strict inequality in the query has to be translated into a combination of a non-strict inequalities that can be computed by the source and additional inequality tests that to be performed by the MEDBMS. It is possible to access some types of data sources both only through a source interface or with an additional translator for better performance. As an example we point to RDBMS sources. They can be treated simply as storage managers with a simple interface to scan tables and get tuples by key. Then all other operations must be performed by the MEDBMS. For better efficiency a translator may be added that would push whole query subexpressions to the relational source. This approach to wrapper building provides the means to construct wrappers incrementally - first provide a minimal wrapper only with data and meta-data access functionality, and then gradually add functionality for source statistics, and a translator with source capabilities and rewrites.

[2] www.sleepycat.com

3.4.2 Mediator DBMS

The MEDBMS component provides functionality to perform schema integration of many data source instances and to query integrated schemas. This functionality is available through constructs of the mediator query language that are suitable for the resolution of various types of information heterogeneity. Unlike wrappers which are created for each data source type, the integration constructs deal with the semantics of the data in the sources and therefore are used at the data source instance level.

To fulfill requirement *R7*, Sect 1, our mediators provide a functional and object-oriented (OO) common data model and a relationally complete query language based on the Daplex functional data model [48]. The mediator data model and query language are described in detail in *Paper B*. The functional OO data model provides powerful modeling capabilities that allow to represent the data in most most existing kinds of sources starting from flat files, to relational databases [12], object databases and even product models of engineering artifacts [29]. In particular, the concept of function in the query language presents a perfect match to the view of data sources as sets of computations that possibly require input data.

More specifically schema integration in our architecture is decomposed into the following tasks.

- **Data transformation.** While the wrapper tier performs various data transformations, this is done automatically for all data sources of the same type. Often these automatic transformations may not be sufficient and additional transformations may be necessary that are related to the data semantics and thus depend on the source instance. For example strings in a Web document may be converted by a wrapper to numbers, but the application domain may require these numbers to be rounded to some precision. Data transformations may also be necessary to extract individual items from complex values, e.g. to extract the first and last names of persons from a string, or to merge individual items into one value. Data transformations are seldom used alone. Typically they are used as parts of the more complex transformations described next.
- **Schema restructuring** is used to map both semantically and structurally heterogeneous sources into uniform representations which can be further integrated. Schema restructuring involves operations like: renaming of attributes and data sets, using data transformation to align attribute data types, addition of new (possibly computed) attributes or merge of several attributes into one, changing the schema concept used to represent a concept in a data source, and restricting a data set to some subset. Schema restructuring is performed over the schema elements of a data source instance. The result of schema restructuring are schema elements that represent real-world entities from the same domain in the same way.

- **Unification of overlapping data.** When integrating data sources that model the same or related application domains, the sources may contain data items that represent real world objects of the same kind. There are two general cases: either some real-world entities are represented in more than one data source *overlapping sources*, or there is no overlap between the sources. The latter case is the simpler one. For non-overlapping sources it is sufficient to restructure their schemas so that they have compatible structure, after which the sources can be merged by a union operation.
  The case when sources overlap poses two problems. First, it requires that data objects which represent the same real-world entity are matched. This requires object identity to be defined in some way and (possibly) different representations of object identities to be mapped. This can be solved either by applying schema restructuring, or by directly using data transformation. Second, once object identity can be established, matching data items may not agree on the values of some attributes. In some cases such data conflicts can be resolved automatically by default operations for each attribute data type, e.g. always take attribute values from one of the sources, or always compute average of numeric values. However, in many cases the data semantics may be more complex and may require a human to explicitly specify data conflict resolution rules.
  When source overlap, the user may want to define a view that contains various subsets of all objects in the sources. The most common case is a view that contains all real world objects from all sources without the duplicates. Another case is a view that contains only the objects present in all sources. Finally a user may be interested in the real-world objects present only in some sources.
- **Reduction and summarization.** The integration of many sources may result in views that contain very large amounts of data while a user may be interested only in some general properties of data sets as a whole like trends averages, etc. Data reduction and summarization tasks can be performed as part of any of the previous two stages or separately over the integrated views.

To support these schema integration tasks, our mediators' query language has several features that interact with each other: *i)* support for extensibility through foreign functions, *ii)* a view definition facility, *iii)* reflectiveness, by which schema objects are treated as other data items and can be queried, and *iv) global query facilities* that allow for a mediator to specify queries in terms of database objects in other data sources, including mediators. Next we introduce our mediator data model and query language in terms of which these features are realized and point out how the mediator language constructs realize the data integration functions listed above.

Data model and query language.

The basic modeling concept in the mediator data model is the *object*. Objects are classified in *types*. Attributes of objects and relationships between types of objects are expressed through *functions*. While objects model real-world entities, in general functions represent computations. Depending on how a computation is implemented we distinguish several kinds of functions - *stored* functions store explicitly the result of a computation, *derived* functions specify the result of a computation as a declarative query defined in terms of other functions, *database procedures* describe computations in a procedural language that uses the mediator data model, and *foreign* functions represent computations specified in an external language(s) and/or module(s). To model arbitrary computations, functions are annotated with binding patterns [34] that specify inputs and outputs. Each binding pattern may have its own implementation that computes the foreign function in the most efficient way. To allow the MEDBMS query processor to pick the best foreign function implementation when several are applicable, each binding pattern also has a cost function associated with it. Functions that have more than one binding pattern associated with them are called *multi-directional*. All kinds of functions can be used anywhere in the query language where a function can be used.

All objects of a type constitute the *extent* of that type. Thus types can be viewed as named sets of objects with the same structure. All types are organized in a multiple inheritance hierarchy where the extent of a subtype is a subset of the extents of its super-types (*extent-subset semantics)*.

The mediator data model is *reflective* [38] in the sense that all data model concepts are represented in terms of the data model by *meta-objects* classified in *meta-types*. Types and functions are objects themselves and are instances correspondingly of the meta-types *Type* and *Function*. Other meta-types describe various aspects of the schema of a mediator, its knowledge about other mediators, data sources, applications and even its internal state. Since all meta-type objects are no different from the user objects, mediators are inspectable via their query language through queries that can freely mix user types and meta-types. This approach provides flexibility when inspecting mediators combined with the simplicity of using the same query language for data and meta-data retrieval.

Data integration functionality.

Data transformation and data reduction and summarization are supported directly through foreign functions and database procedures. Since foreign functions can be implemented in external languages as C and Java, the mediator user may add new functions that perform arbitrary specialized computations to transform data in an application domain-specific manner (e.g. to apply an image filter to image data) or to summarize domain-specific data (e.g. to com-

pute the average lightness of images). Foreign functions[34] are similar to, but simpler and yet more expressive, than user-defined functions (UDFs) in object-relational DBMS.

The mediator query language has a view[3] definition capability through derived functions which are named and parameterized queries specified in terms of an SQL-like **select-from-where** statements and *derived types*, which are types with their extents specified as queries. Database views address different aspects of schema restructuring and unification of overlapping data. For the schema restructuring tasks it is sufficient to use derived functions. For schema unification a more suitable construct are derived types which provide simple to use syntax to specify rules to match data items from different data sources, and rules to reconcile conflicting attribute values.

Views by themselves are not sufficient to integrate many data sources. For that the mediator query language has the ability to refer to schema elements and objects in other data sources and use them transparently in all language constructs as if they are local. This allows free mixing of local and remote functions, types and objects both in derived types and derived functions. We call this feature *global query facilities* because the query language allows to refer to any globally accessible object in a mediator or a data source. Logical compositions of mediators and other data sources are defined declaratively in terms of each mediator's global query facilities when views in one mediator are defined in terms of other data sources and views in other mediators.

The reflective nature of the mediator data model, combined with its global query facilities and meta-model of data sources (described in *Paper B*), allows queries to be issued over the meta-data of any mediator peer and/or data source. This allows to perform information discovery in a network of mediators and data sources through regular queries. The resolution of structural heterogeneity can be approached by parameterizing schema elements in the integration views (e.g. parameterized relation names) and mixing data and meta-data in the same query or view.

Integrated schemas in terms of the mediator query language are constructed from the sources' schemas in a bottom-up fashion using the global-as-view approach. First, storage elements and computations in the sources are mapped by the corresponding source wrappers to mediator schema objects. After this initial step, data sources logically become part of the mediator database, but there still are semantic differences between the data in different objects. These differences are reconciled through the definition of views (derived types and functions) defined in terms of the source types and functions. These integrated views are then available to other mediators for further integration according to their needs and application domain.

[3]Here we use the general term *view* to denote any declarative specification of derived data from other stored or derived data in terms of a query language.

3.5 Systems of Peer Mediators.

In this section we describe how the three types of software components, described in Sect. 3.3, form together a peer mediator system.

Originally mediators are defined [55] as a middle layer that is distinct both from the underlying data source layer and the higher application layer. However, many software components that belong either to the data source or to the application layer exhibit some functionality characteristic for mediator systems. Since mediators may serve both as intelligent data sources themselves for other mediators and as applications for others, we will consider a *Peer Mediator System (PMS)* to consist of not only the mediators themselves but also of global data sources and global applications all of which communicate over a network interface. This approach allows uniform treatment of all issues regarding the joint operation of applications, mediators and data sources. In particular it allows to design a system that is capable of introspection and through that to facilitate (semi-) automatic integration and adaptive behavior. From our general mediation architecture it follows that a PMS has at least one mediator, and any number of global applications that access any number of data sources through the mediator(s). Collectively the mediators, the global applications and global data sources are the peers in a PMS.

Every peer has a globally unique logical identifier called *peer name* independent of its physical network address. Since data source peers use global naming schemes different from that of the PMS, there is at least one mediator peer that provides a mapping between logical names in the PMS to network-dependent locations of the data sources. Global applications by definition support the inter-mediator interface and thus support the same naming scheme as the mediators. The set of peers whose names are known to a peer are called the peer *neighbors* and are related by a *neighbor* relationship.

We define a *PMS instance* as follows. Given a nonempty set of peers N with at least one mediator peer, where the peers can physically reach each other over a computer network, a *PMS instance* is the set of all peers P, $P \subset N$, formed by the transitive closure of the relationship *neighbor* over the set of peers N starting from one of the mediator peers in N.

Mediator peers can form named groups, called *communities*, which would be typically formed because of common interests of the mediator owners. Groups can be nested arbitrarily and mediators can participate in more than one groups, as in [20, 7]. Some mediators act as *meta-mediators* that know about other mediators and groups. Each group has at least one meta-mediator that stores at least the logical identifiers of its members, their corresponding physical addresses and the name of the group. Meta-mediators may store additional information both about mediators and mediator groups such as content descriptions, statistics, etc.

A PMS is completely decentralized - there is no global meta-data repository (catalog) with information about all peers, and there is no central controller that coordinates all peers. As a consequence:

- No peer has global knowledge about all other peers.
- Since the mediators are completely autonomous, there may be several mediators that define different (possibly conflicting) virtual databases over the same set of sources.
- The only way that data and meta-data can be acquired by peers is by sending requests (usually as queries) to known peers (which may trigger queries to other peers).

Mediators cooperate directly with each other and all control, data and meta-data are distributed among the mediators. Each mediator peer chooses the peers it wants to cooperate with (both as its clients and/or servers) among its neighbors, and has only limited knowledge about a subset of all available peers. Each mediator locally plans its actions based on its local knowledge. Global computations that involve many mediators are planned as a result of many local cooperative decisions. Applications and mediators may recursively initiate cooperation between peers on behalf of other peers or applications.

Most importantly, there is no global integrated view as in federated database systems and centralized mediators. Each mediator defines its own integrated view over a subset of all data sources and mediators and makes some part of its integrated view available to other mediators and applications for further integration or querying. Each mediator peer has total control of its own schema. Finally, mediators may join and leave a PMS at any time.

Our definition of a PMS allows PMS instances to have a very wide range of logical topologies from a client-server with many applications, 1 mediator, and many data sources, to a “pure” P2P system where all applications have gateway mediators and all data sources are embedded in mediators, and there is a network of mediators between the application and the data source layer.

An example of a PMS is shown on Fig. 3.2 where several mediators are defined in terms of other mediators and data sources. In the example, applications access data in several data sources of different kinds (two RDBMS, one Internet search engine and a Web site) through a collection of composed mediator servers. The directed arcs connecting the mediator nodes and data sources correspond to the relationship ”defined in terms of” between them, that is, the mediators that point to other mediators or sources contain views that are defined in terms of views or data in the pointed to mediators and sources. We illustrate this in the upper-left mediator in the figure, where a global view is defined in terms of other two mediators. It is important to point out that mediator compositions are not defined as static networks of mediators but are dynamically generated through the definition of queries or views. Each query or a view uses only a subset of all logical links, defined by the transitive closure

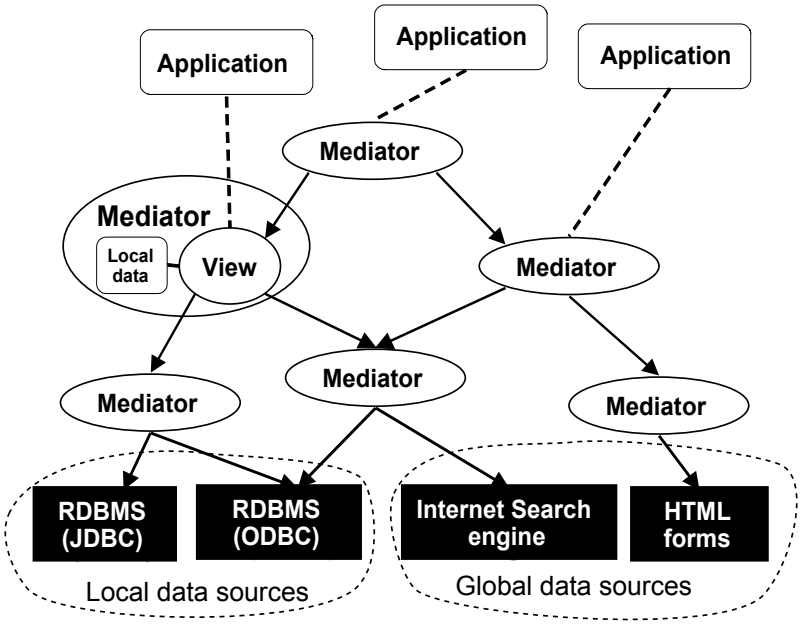


Figure 3.2: An example of a peer mediator system

of the logical relationship ”defined in terms of” between all views referred by that particular query or view. Thus Fig. 3.2 is a simplified view of the union of several superimposed logical mediator compositions.

The advantages of the P2P approach for mediation are that it allows the domain experts to own and control independently their mediators in the same way as data source owners have total control over the data sources. Each mediator may evolve at its own pace as long as it preserves its public interface. In the foreseeable future it may be expected that data integration will remain a predominantly “manual” task that requires a lot of domain knowledge and human participation. A P2P architecture allows to distribute the integration effort between many autonomous domain experts and thus scale the integration process. The domain knowledge encoded in the mediators is shared so that other more complex mediators can be composed in terms of simpler ones and thus integrate data across many data sources and knowledge domains in a scalable manner. Finally, a P2P architecture promotes reuse of computation resources such as storage, CPU cycles, and specialized software and hardware.

The Problem of Query Processing in Peer Mediator Databases

A successful implementation of the PMS architecture presented in Sect. 3 must fulfill a wide range of requirements, some of which are discussed in Sect. 3.2. Our focus is on the most fundamental requirement for the PMS architecture, that of composability of mediators in terms of other mediators and data sources. As discussed in Sect. 3.2 the fulfillment of this requirement results in a data integration architecture that meets the general requirements *R1-R7* for large-scale data integration stated in Sect. 1.

The problem we address in this dissertation at a high-level is how to implement mediator composability effectively and efficiently in a PMS architecture so that a PMS system can scale over the number of composed mediators. This general problem can be decomposed into two sub-problems described below.

- *Scalable integration.* The first aspect of mediator composability is how mediator compositions are defined so that many views from many mediators are integrated into higher-level reconciled views. Compared to centralized mediation architectures, in a PMS this problem has the additional complication that the views are defined in many mediators and there is no central repository that keeps track of all existing views in all mediators. We address this problem by providing a query language with global query capabilities. However, the problems remain *i)* how to discover the relevant views to a problem domain, and *ii)* how to specify in a scalable manner integrated views over large number of views. While these two problems are very important, in our work we assume that method(s) exist to specify integrated global views over many mediators. This can be done manually directly in terms of the global query capabilities of the mediator query language[23, 24], or (semi-) automatically through the use of visual tools and inference mechanisms[57]. In addition the mediator query language may be extended with more expressive constructs for data integration. Given the high expressive power of the mediator query language and data model, we believe that future tools or language constructs can be expressed in terms of the existing language features. Thus in the rest of our discussion we will assume that integrated global views are preexisting and are specified in terms of the mediator query language as presented in *Paper B*.

- *Query processing.* Assuming that the means exist to specify mediator compositions, the next important problem is how to provide scalable performance for the computation of queries against composed mediators so that a PMS is usable. Composability of mediators has two dimensions. Logical composability is related to the means of specifying compositions of mediators in terms of a declarative query language. Physical composability is related to the means by which mediators physically interact with each other and with external sources as one distributed system. In order to compute answers of queries in a PMS, logical mediator compositions must be represented as physical ones.

  Our mediator compositions are described in terms of a query language, therefore the process of translating logical view compositions into physical ones is in fact query compilation, while the computation of query results according to a physical composition of mediators is query execution.

The two problems are tightly interrelated. On one side various approaches can be envisioned to integrate many mediators and views such as tools and language constructs. On the other side only some of these approaches may be viable because of limits on their performance. Based on our analysis of related work in the area of data integration, we conclude that while a considerable amount of work has been done in the area of data models and query languages for data integration that can be applied to a PMS architecture, the problems specific to query processing in peer-to-peer architectures for data integration have not been adequately addressed.

Thus, query processing in peer mediators itself poses a wide variety of challenges. In the remaining of this section we discuss several interrelated subproblems that we address in this dissertation.

Capabilities of inter-mediator interfaces.

One of the most fundamental issues for a distributed system is how to design the public interfaces of the components in this system so that they can interoperate, are easy to evolve, and are efficient. In large scale P2P systems it is also important that the peer interfaces provide enough expressive power so that the distributed system as a whole can self-organize itself to perform efficiently as a whole. In particular the interfaces of the PMS components should be sufficient for them to cooperatively process global queries in an efficient way.

As noted in our discussion on data sources in Sect. 3.3, low-level interfaces provide the communication infrastructure for distributed systems, but they do not solve the problems of the semantics and granularity of the interfaces, that is, what functionality is exposed through an interface and what is the granularity of the interface. By functionality we mean what computations does one system expose through its interfaces, and by granularity we mean at what granularity does a system provide a view of its internal state through an interface.

Thus, independent of the low-level infrastructure used for interoperability between the components in a PMS, there is a large space of design choices related to the functionality that PMS components should expose to enable efficient cooperation between them. *Paper B* investigates what computational capabilities a software component should provide in order for that component to participate as a peer in a PMS.

Overhead of logical mediator compositions.

Logical composability and autonomy of mediators poses several challenges to the computation of queries over integrated global views. Since there is no global control in a PMS, every mediator owner has the freedom to compose arbitrary global views defined in terms of any of the known and accessible mediators and data sources. This ability to compose new mediators in a globally uncontrolled manner may result in enormous redundancy in large mediator compositions. Typically a mediator will be aware of and will integrate a relatively small number of sources and neighbor mediators that provide information of interest. However, the neighbor mediators may derive their information from any number of other mediators and sources not known directly to the first one. In this way it may be common that data from the same mediator(s) and/or source(s) is indirectly integrated by a mediator through many levels of other mediators, where each one eventually adds some value by restructuring and enriching the information from the lower levels.

If queries over such composed mediators are executed naively by following the logical links between the mediators, this may result in many redundant computations performed by each of the underlying mediators, as well as in many redundant network accesses and data transfers, which may result in an unusable PMS.

Therefore methods need to be developed that remove these redundancies and generate efficient query execution plans (physical compositions of mediators). Since logical mediator compositions are essentially views defined in terms of other views, these views can be expanded (unfolded) as in traditional DBMS. However, in a P2P setting there is no central catalog and typically no mediator "knows" the definitions of external views. Another issue is that in traditional database design the database schema is designed in a top-down fashion and one may expect it to be relatively well designed and have relatively small number of levels in the view definitions. However, due to the uncontrolled bottom-up design of data integration solutions, it may be expected that very large numbed of views will be nested very deeply. Finally, due to mediator autonomy, some mediators may refuse to make their view definitions available to others, e.g. because they want to hide their information sources. To respect each other's autonomy, mediators should be able to negotiate if and which views can be expanded, and be able to compile and execute queries in

all cases. Therefore view expansion in a P2P setting may not be as "simple" as in a traditional DBMS setting. *Paper C* studies the problem of view expansion in the presented PMS architecture.

Decentralized query processing.

A decentralized architecture of many autonomous, but equal in capabilities peers, such as the PMS architecture, presents new opportunities and problems for the processing of global queries. In a centralized distributed DBMS system, there is one controlling peer, typically the peer where a query is issued, that is responsible for the compilation and execution of its queries. This is possible because there is a central catalog with all meta-data necessary to produce optimal QEPs, and because the component DBMSs give up their autonomy and leave the control to one peer. However, in a decentralized system, no peer has global knowledge, or global control over the other peers. One alternative to approach the lack of meta-data is to request it from the other peers involved in a query. Another possibility is to use the fact that the other mediator peers have their own query processors and local meta-data and thus may take better decisions regarding local queries. Thus, instead of exchanging meta-data, an alternative is to submit queries for remote compilation. In addition such distributed compilation provides the means for load balancing during query compilation. Another side of the problem is query execution in a centralized system. There, one peer controls other peers during query execution. As a result all data flows through the central peer. In a P2P mediator system, where peers are distributed across a wide-area network with highly varying link parameters, centralized data flow may be far from optimal. Instead, it may be much more efficient to exchange data directly between peers that are connected with fast links and and let them cooperate to compute intermediate results which can be shipped to the query peer or some other intermediate peer. As with cooperative compilation, such cooperative execution provides the additional possibility for utilization of the resources of all peers. In *Paper D* we study one particular method for optimizing global queries through distributed compilation that produces decentralized QEPs.

Distributed join methods for mediation.

Data integration problems often require cross- source or mediator join operations because of overlapping information in the sources and/or mediators. Join is known to be the most expensive operation in database systems. The presented PMS architecture is different from centralized and distributed but homogeneous DBMS architectures in that joins have to be made between mediators and sources with limited capabilities or computational sources often over slow network connections. With such sources, data produced by one of the join operands is required by the other operand as input, and therefore this interme-

diate result data has to be shipped from one operand to the other. Such joins are often called *dependent* because the execution of one of the join operands depends on the execution of the other. Thus mediator systems need specialized methods for the execution of dependent joins that take into account and reduce data shipping costs together with the cost of join computation. The focus of *Paper E* is the design and study of three mediation-specific join strategies.

Access to diverse sources.
A mediation system would typically access a wide variety of data sources. It is hard to predict in advance even what will be the future kinds of sources that need to be integrated as a data integration system evolves. Therefore the mediator components must be designed in a way that allows new sources to be added easily and dynamically. Since sources are accessed through wrapper components, this amounts to the question how to design a generic mediator-to-wrapper interface and meta-model of data sources that allows the addition of new wrappers for new kinds of sources. Another, more specific question is, given the presented mediation architecture, is it flexible enough to easily accommodate new kinds of sources? We address this problem in *Paper F*, where we design and investigate a wrapper for several Internet search engines as an example of non-database-like data sources.

Related Work

In this section we overview works related to the PMS architecture which serves as the basis for our work, we point out the similarities and differences between our architecture and other projects, and summarize how these projects relate to the query processing problems described in Sect. 4.

5.1 Distributed Database Systems

There is a large body of knowledge on query processing in distributed databases [30] that may provide partial solutions for problems related to query processing in a PMS. However, the autonomy, distribution and extensible object-oriented data model of the PMS architecture proposed here, poses new problems different from the ones related to distributed databases [37]. In distributed database systems (DDBMS) the peers are homogeneous, there is a single site that controls query processing and a centralized catalog, usually replicated in all databases. In contrast to that, in a P2P system there is no central controlling site and all meta-data is distributed among the peers. Various new problems arise from that. Here we mention only some of them: to produce a global QEP all peers involved in a query have to cooperate in the compilation process because no peer has complete knowledge of execution cost; peers have to request cost information from other peers over the network which incurs high cost of getting the cost; due to autonomy, cost information may not be available at all peers; peers cannot assume that every other peer is capable of the execution of arbitrary query fragments; therefore predicates might not be freely pushed in the QEP from one mediator to another.

5.2 Mediator Systems

A considerable number of mediator systems [10, 13, 18, 25, 51, 46, 35, 58] have been proposed with varying architectures in terms of their degree of distribution, autonomy and data model.

Many of the mediator proposals and systems have a centralized architecture, that is they consist of a single mediator component, interacting with the data sources via wrapper components. The wrappers themselves may or may not

be distributed with respect to the mediators. However, even in mediator architectures with distributed wrappers, all meta-data and control in the mediator-wrapper system as a whole are concentrated in the mediator, and wrappers in such architectures are not autonomous. Therefore, we will consider mediator systems with distributed wrappers to have still centralized architectures, where composability is not an issue. Typical example of centralized mediator systems are Garlic [13] and DB2 Federated DBMS [18]. The TSIMMIS [13] and Pegasus [10] projects mention that distributed mediators may access other mediators, but no results are reported in this area.

Some mediation prototype systems have distributed architectures where mediators access other mediators. Next we compare these mediator systems with our PMS architecture.

The AURORA prototype [58] follows a fixed two-tier mediation model consisting of three types of distributed components. The first tier consists of *homogenization mediators* that deal with schematic mismatches on per-source level, and distributed wrappers that provide access to the data sources. The second tier consists of *integration mediators* responsible for integrating multiple homogenized sources through their respective homogenization mediators. This distributed architecture is similar to our PMS architecture in that there is no single monolithic integrated view, instead the integration process is distributed among many mediators. The main focus of the AURORA project is a methodology for source homogenization and conflict resolution. Various algebraic rewrite methods are proposed for pushing integration and standard relational operations to the sources, but neither distribution of sources and/or mediators is considered in any way, nor the effects of logically composing the integration mediators in terms of many homogenization mediators. Query processing in AURORA is considered only from the view point of a single homogenization mediator, and while not explicitly said, it seems to be performed in a centralized manner. Our PMS architecture generalizes that of AURORA because our mediators can be specialized to perform different roles as homogenization and/or integration, while there is no restriction on the number of mediator tiers.

The DIOM project [46, 35] is one of the few mediator projects that points out composability as an important property for scalable integration of many sources. The project presents the implementation of a distributed mediator architecture where mediators can access other distributed mediators and/or wrappers. One feature that distinguishes DIOM from other mediator projects is that it does not require conflicts to be resolved statically in an integrated schema. Semantic conflict resolution is deferred to query result assembly time instead before or at query compilation time. Thus users can dynamically specify the information they are interested in and their preferences, and the mediator performs automatically source selection and conflict resolution based on user pref-

erences. The DIOM prototype features a query processor aware of the distribution of the sources and capable of dynamic query routing and scheduling, but all query processing is performed in a centralized manner, such that query compilation and execution are controlled by only one mediator. As a result QEPs in DIOM are centralized and always follow the logical composition of the mediators. To the best of our knowledge there are no reports of the actual performance improvements achieved through the query processing approach in DIOM. In particular all reported results describe processing of queries by one mediator against several wrappers and do not address issues specific to processing queries in mediator compositions.

The DISCO mediator system [51] also has a typical mediator-wrapper architecture with distributed wrappers accessible by distributed mediators. Since every mediator is a wrapper, mediators can call other mediators as if they were wrappers. One of the prominent features of DISCO is graceful handling of unavailable sources through partial evaluation semantics that returns partial answers to queries by processing as much of a query as possible and returning the remaining non-processed part of the query to the caller. This approach to handling source unavailability can be applied to our architecture to fulfill the requirement for dynamic source availability described in Sect. 3.2. As in the DIOM system, query processing in DISCO is described only for flat two-tier cases and does not take into account problems (and optimization opportunities) resulting from mediator compositions.

Our conclusion is that, while several projects mention mediator composability as an important feature, none of these projects addresses issues related to query processing in many composed mediators, which is the main focus of our work. However, the described projects address other issues, important for scalable data integration of many data sources, that are complementary to our work.

5.3 Peer Data Management Systems

Several recent works propose P2P architectures for data integration and for the management of distributed and autonomous databases. The ideas presented in these works are the closest to our PMS architecture, and therefore we discuss them here in more detail.

Data management systems based on P2P computing paradigm are discussed in [16, 19, 20, 6] where new problems and opportunities arising from the usage of a P2P paradigm are identified. However, there is little work on implementation issues of such systems, especially related to large number of cooperating query processors. Even more, these works point out problems specific to P2P architectures some of which we address in this dissertation. In the vision paper [16] it is indicated that two fundamental problems in most P2P systems are

the placement and retrieval of data and therefore DBMS technology can and should be applied to P2P systems. At the same time P2P architectures can be useful in DBMS systems to provide system robustness and scalability, eliminate proprietary interests, reduce administration effort and provide anonymity. Of the two main problems mentioned, the paper describes in more detail the problem of data placement. Solutions to this problem can be applied in our PMS architecture, e.g. for efficient caching and replication of data at the mediator peers. One of the problems related to a P2P architecture is that of the extent of knowledge sharing between peers. We analyze and provide some answers to this problem in *Paper A* with respect to an architecture with no centralized catalog.

Another vision work [6], addresses the problem of semantic inter-dependencies in between autonomous peer databases in the absence of a global schema. The paper introduces the Local Relational Model (LRM) as a data model specific for P2P data management systems. Inter-peer semantic dependencies are described through coordination formulas that allow the synchronization of many peer databases. The LRM can be used to mediate between multiple peers and to propagate updates between peers so that consistency is preserved. The architecture proposed for the LRM is described at a very high-level of detail, and at that level of detail it is similar to our PMS architecture. In terms of query processing in the proposed LRM model, the paper lists several P2P-specific problems, but no solutions are proposed.

At the architectural level, the works closest to ours are [20, 19]. Based on the assumption that data integration systems have one global mediated schema that integrates all sources, the two papers advocate the concept of *peer data management systems (PDMS)*, as systems that replace the single logical schema of data integration systems with an interlinked collection of semantic mappings between the peers' schemas. The ideas described in the two papers are implemented in the Piazza peer data management system. The main problem addressed in the two papers is that of schema mediation in a PDMS. To specify schema mappings between peer databases the authors propose a language *PPL* that allows to express both GAV and LAV style mappings between peer schemas. In [20] the *PPL* language is an extension of Datalog, and thus suitable for peers supporting the relational data model. In [19] the mapping language is modified to support RDF and XML sources. With respect to query processing, both works deal with the problem of query answering (reformulation) in the presence of mixed GAV and LAV transitive mappings between peers. The goal of query answering is to reformulate an initial query in terms of schema mappings to a query in terms of the base relations. As the authors notice in [19] they do not address the problem of efficient processing of queries which is essential for the overall performance of a PDMS.

From an architectural perspective, at the level of detail presented in [16, 19,

20, 6], all these proposals including ours are related. The main differences are in the data models proposed, which is functional and object-oriented in our case, and relational and RDF/XML in the other cases; the schema mapping approaches used; and the query processing issues addressed in these works.

Regarding the problems related to query processing in a P2P architecture which are our primary interest, our work and that of [20, 19] are complimentary in several ways. The query reformulation algorithms presented there fully expand all views and rewrite all queries in terms of the base relations. As shown in *Paper C* selective view expansion may often lead to better results with substantially less compilation cost. Thus query reformulation in Piazza can be simplified by not expanding all views (mappings), while our PMS architecture can benefit from a more general method of mapping peer schemas and its query reformulation algorithm. Since the current work on the Piazza system is focused on query reformulation, all our solutions related to query processing in a PMS can be directly applied in Piazza and similar PDMSs.

In [42] a P2P distributed data sharing system, PeerDB, is presented and some of its aspects are experimentally evaluated. PeerDB consists of arbitrary number of autonomous peers each of which consists of a relational DBMS (MySQL), an agent system DBAgent, and a cache manager. Peers find each other through one or more global names lookup servers that provide each node with a unique identity. PeerDB uses an information retrieval approach to the discovery of relevant information. Each relation and attribute in the peers' databases is tagged with keywords. Relevant relations are discovered through keyword matching and ranking. Compared to our PMS architecture, PeerDB does not provide global query facilities and does allow for the definition of integration views across multiple peers. Since there is no global view definition capability, PeerDB does not provide logical composability and the peers constitute a logically "flat" system. PeerDB naturally handles peer unavailability because there is no predefined integration schema. Query processing in PeerDB is performed through "agents" that are dispatched to other peers by the DBAgent component, but the paper neither defines what is an agent, nor it describes by what algorithm(s) agents are dispatched to other peers. Finally, PeerDB does not address issues concerning access to external sources with varying capabilities. Our conclusion is that PeerDB is suitable for the sharing of structured data in a P2P fashion, but it cannot be applied to real data integration problems.

A distributed relational query processor is proposed in [7], where the focus is on dynamic extensibility and security. Advances in this project are complimentary to our work and can be applied in the presented PMS architecture. The project does not specifically address the integration of heterogeneous data sources, neither problems related to redundancy in compositions of many autonomous database.

Summary of Contributions

The hypothesis underlying this work is that a peer-to-peer mediator architecture is more suitable for many real-world data integration problems than a centralized one. It is shown to be possible to design a mediator system with a peer architecture that can process queries efficiently and can scale in terms of the number of peers. The main contributions described in this dissertation are:

- Analysis of the components of a PMS - applications, data sources and mediators. (Sect 3.3)
- Design and implementation of a P2P system for distributed data integration. In the architecture autonomous peers share data and services with other peers without a global coordinator. Mediator peers provide a unified and knowledge-enriched view of many autonomous and heterogeneous sources in terms of a functional and object-oriented common data model and query language. The integrated views can be either queried directly or can be used by other mediators to compose higher-level integration views in terms of views in other peers. (Sect 3 and *Paper A*)
- Analysis of the inter-peer interfaces and corresponding computational capabilities of the peers, the meta-data that needs to be exchanged between the peers, and the query processing techniques that can be used in the presence of some capabilites and meta-data in order to implement a PMS. (*Paper B*)
- Technique, called distributed selective view expansion (DSVE), to efficiently process queries against many composed mediator views. DSVE has been implemented in practice in the AMOS II mediator system and based on this implementation it has been experimentally evaluated. The experimental analysis of this technique shows that it is possible to provide good query performance with low compilation cost in a peer mediator system. (*Paper C*)
- A distributed compilation technique to re-balance left-deep QEPs which due to the autonomy of each peer, not only describe access to distributed sources, but are distributed themselves. The QEP rebalancing technique improves the quality of the QEPs in a peer mediator system by enabling direct decentralized communication between the peers involved in the computation of a query result. The QEP rebalancing technique was implemented in the AMOS II mediator system and studied experimentally. (*Paper D*)
- Design, and experimental study of three join algorithms for a peer mediator

system. Two of the algorithms, called *ship-out*, ship bindings from one of the join operands (local or remote) to another remote operand and thus are suitable for the computation of joins involving sources with limited capabilities. The third *ship-in* algorithm, ships all data to the join site, where the join is computed. Ship-in joins are suitable for sources with a scan interface accessible over a fast network. (*Paper E*)

- Application of mediation for Internet search engines (ISEs). Various ISEs are integrated through a flexible wrapper manager sub-system, called object-relational wrapper for ISEs (*ORWISE*), that utilizes external web wrapper toolkits and allows for flexible and dynamic addition of new ISE wrappers. The design of ORWISE shows that the basic facilities for extensibility in the AMOS II system described in *Paper A* are powerful enough to support such non-database-like sources with ease. (*Paper F*)

In addition, during my work various components of a peer mediator system have been implemented as part of the AMOS II mediator system.

- Design and implementation of a meta-schema that models data sources. The meta-schema allows for declarative manipulation of information related to all data sources through the mediator query language. This allows mediator users to query data source meta-data for discovery of relevant sources. In addition the mediator kernel itself has been changed to reflectively utilize the data source meta-data during query optimization. The meta-schema is described in *Paper A* and *Paper F*.
- Experimental studies of a PMS require that large number of measurements are performed and dependencies on many parameters are investigated. This results in large volumes of distributed measurement data with complex structure. This requires that both the execution of experiments and experimental data collection are performed in an automated way. A natural approach is to use the mediator system itself to manage and collect the experimental data. To enable the performance of large-scale computation experiments in a PMS, I designed and implemented a declarative framework for automated computational experiments built on top of the AMOS II system. The framework allows to configure and execute an experiment, collect all experimental data and plot various dependencies only through the query and stored procedure language of the AMOS II system. The framework was used to perform all experiments in *Paper C*.
- One of the most important types of data sources are RDBMSs. The most wide-spread and standardized way to access RDBMS sources is ODBC. To make our experimental studies more relevant, a wrapper for ODBC data sources was implemented in AMOS II . The wrapper was used in all experiments in *Paper C, Paper D* and *Paper E*.
- Many improvements in most components of AMOS II were necessary to implement the query processing techniques and to perform the experiments

described in this dissertation. Some of the improvements led to orders of magnitude less memory consumption and smaller compilation times.

Summary of Appended Papers

The papers included in this dissertation and summarized in this section are inter-related in the following ways. *Paper A* describes an implementation of a PMS that uses some of the results of *Paper B* to process global queries. *Paper B* investigates inter-peer interfaces and capabilities required for the interoperability between mediator peers and/or data sources in a PMS, and the applicable query processing techniques in the presense of these interfaces. *Paper C* studies in detail how to process queries over mediator compositions specified in the query language described in *Paper A* using the *view shipping* approach described in *Paper B*. *Paper D* investigates query optimization techniques based on the *query shipping* approach described in *Paper B*. *Paper E* describes distributed join methods for the PMS described in *Paper A*. Finally, *Paper F* describes how to add new wrappers for Internet search engines to the PMS presented in *Paper A* as a test case for mediator extensibility.

The overall structure of the dissertation is depicted on Fig. 7.1 where the thin lines represent the relationship "uses results from".

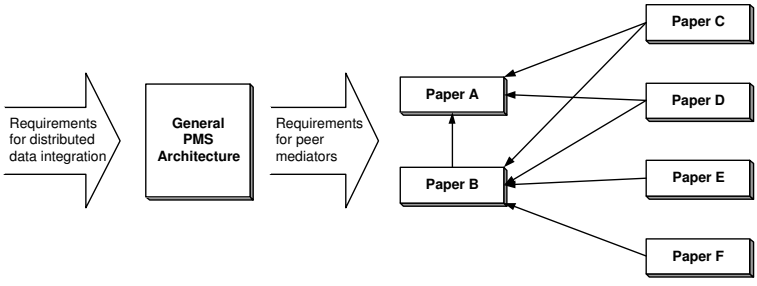


Figure 7.1: Logical organization of the dissertation.

7.1 Paper A: Functional Data Integration in a Distributed Mediator System

This paper describes an implementation of the PMS architecture presented in Sect. 3 in the framework of the AMOS II (Active Mediator Object System) mediator system. In this section we will summarize the main features of the

AMOS II mediator system from *Paper A* and will point out their relationship to the other works that constitute this dissertation. We will also point out the limitations of the current implementation with respect to the general PMS architecture.

With respect to the requirements for a PMS listed in Sect. 3.2, AMOS II mediators are fully composable at the logical and physical level. Location transparency is supported through unique names assigned to each mediator and a name resolution service described below. Peer discovery is supported through the data source meta-schema that models data sources and can be queried via the AmosQL query language. The issues that are not addressed in the current implementation of the PMS architecture are dynamic availability of sources, security, replication and caching.

With respect to the general mediator architecture, the AMOS II mediator system described in this paper is different in that there is only one meta-mediator, called *nameserver* that stores the mapping between logical mediator names and physical network addresses. The second difference is that all data sources are treated as local by the mediators, and thus even if the same global data source is accessed by several mediators, such information cannot be discovered and utilized by the mediators.

Further details about the data model and query language of AMOS II , its reconciliation primitives, and its query processing can be found in *Paper A* and its references.

Comments

The main contribution I made to this paper is in the design and description of the data source meta-model and description and clarification of the multi-database query processing. I wrote the respective sections and participated in other parts of the paper clarifying various aspects of the functional approach to distributed mediation.

7.2 Paper B: Interface Capabilities for Query Processing in Peer Mediator Systems

Each peer in a PMS must provide an interface to its data and meta-data sufficient to allow the cooperative processing of queries in a PMS. In *Paper B* we analyze the computational capabilities and meta-data that a software system has to export through its interfaces in order to participate as a peer in a PMS, and the corresponding query processing techniques that can be applied in the presence of some meta-data and capabilities of the peers. Our analysis is based on the functional data model and query language for data integration, presented in detail in *Paper A*. We model data collections in remote peers as *proxy functions* that describe the data types and relationship of the data items

in a remote data collection. Proxy functions may be implemented in different ways, and queries over such functions may be also processed in different ways depending on the computational capabilities of the peers in a PMS. Based on the concept of proxy functions, we identify and compare six classes of peer capabilities with increasing complexity, summarized below.

- **Single-directional proxy functions (SDPF).** The simplest possible way to interface peers in a PMS, so that mediators can compute inter-peer queries, is to assume that remote peers provide some interface to directly access their data and to associate each proxy function with an implementation that computes the results of the function by using the data access interfaces at the remote peers. Proxy functions implemented in this way are computable in RPC-like manner only in one direction - from their input parameters to their results, and thus are termed as single-directional proxy functions (SDPF).
- **Multi-directional proxy functions (MDPF).** Often it may happen that several SDPFs represent different directions of the same abstract relationship stored at a remote source. As a result users have many alternative ways to specify the same query, each with potentially very different performance. To offload the user from performance considerations, we introduce multi-directional proxy functions (MDPFs) that tie together several SDPFs into one multi-directional proxy function. MDPFs provide higher degree of abstraction for the mediator users than SDPFs, and better query performance through query optimization.
- **Multi-peer proxy functions.** Some of the functions referenced in a global query may be computable at more than one peer. With SDPFs or MDPFs it is up to the user to choose which alternative to use which requires users to deal with performance issues and may result in sub-optimal query performance. To alleviate this, we introduce an additional level of abstraction through multi-peer proxy functions (MPPFs), where one MPPF relates together all MDPFs (or SDPFs) that represent the same computation in different peers. Thus MPPFs shift the task of choosing the optimal peer for a function from the user to the mediator query processor.
- **Plan shipping.** All previous three interface classes assume simple peers that provide only direct access to their data through some interface, and all query operators, such as join, must be performed at the query peers. This requires that all data during query execution is shipped through the query mediator in a centralized manner. However, remote peers may be capable of computing groups of database operations in the form of query plans. Such capability can be utilized by the mediator peers through plan shipping where the mediators compile an inter-peer QEP, identify the portions of this QEP, called sub-plans, that can be computed by other peers, and ship these sub-plans to the remote peers for execution.
- **Query shipping.** Many data sources, such as relational DBMS the medi-

ators themselves, provide a declarative query interface to their data. Thus, an alternative to plan shipping is to group together through query decomposition the query operations computable at the same remote peer into subqueries, and to ship them for compilation and execution to the corresponding peer.

- **View definition shipping.** In a PMS, mediator peers can be freely composed logically in terms of other mediators and data source peers through database views. This may result in a network of logically composed peers with redundancy, where many peers integrate the same source peers and even the same remote views through many different logical paths. This logical redundancy may result in many redundant computations and network data transfers. To discover and remove such redundancy, the peers must be capable of exchanging view definitions, so that the query peers can analyze and optimize together the expanded view definitions.

The analysis of inter-peer interface capabilities and the related query processing techniques presented above is based on our experiences from the implementation of the AMOS II peer mediator system. We describe the implementation of a PMS in the AMOS II mediator system with peer capabilities within each group. The description of our PMS implementation relates together the results of most of the papers in this dissertation. Since queries over many peers are always reduced to SDPFs, and join is one the most common and expensive database operation, in *Paper E* we design and study the performance of three algorithms for computing inter-peer joins over SDPFs. In *Paper D* we study an application of query shipping for rebalancing left-deep global query execution plans to produce decentralized inter-peer QEPs. Finally, *Paper C* investigates techniques to implement view definition shipping that improve the quality of QEPs with low compilation cost.

Comments

The work described in this paper was done by me with discussions with Tore Risch.

7.3 Paper C: Scalable View Expansion in a Peer Mediator System

Views are the central concept for data integration in the PMS architecture. This paper studies in detail the *view shipping* query processing for peers with view shipping capabilities described in *Paper B* as a promising technique for efficient processing of global queries over views defined in many peers.

There are two well-studied approaches to implement distributed information systems. The first treats each of the distributed modules of an information system as black boxes. The modules communicate with each other through

some protocol without revealing the implementation of the services they export. This is the approach used in CORBA based systems [1]. On the other end are distributed database systems where database views are fully expanded [44] independent of the location of the base tables and views that are used in a view definition. We term the first approach as the *black-box* and the second as *transparent box* approach.

The black-box approach provides full autonomy of the mediators, while at the same time compiling queries without expanding all view definitions may result in sub-optimal execution plans due to missed optimization opportunities and many redundancies in mediator compositions. Without view definitions being expanded, client mediators cannot 'see' that their sub-mediators have views implemented in terms of the same common sub-mediator. As the experiments in this paper show, such redundancies often lead to very inefficient QEPs.

To solve the problems of the black-box approach, DBMS query compilers expand view definitions. This 'reveals' to the query compiler the information 'hidden' in the view definitions which allows for better quality execution plans by optimizing together queries with all directly and indirectly referenced view definitions. In a PMS, view expansion may also allow to combine the view definitions from various mediators, discover and remove redundant accesses to intermediate mediators and push the resulting merged query down to the mediator(s) that actually contain/produce the data of interest. As one may expect, such compilation techniques lead to several orders improvement in the quality of a QEP. However, expanding all participating mediator definitions may result in high compilation cost as many more mediators may become 'visible' to the mediator that compiles a query and many more predicates are added to the initial query. In large mediator compositions this may lead to prohibitively high compilation cost because of very large queries and large number of mediators.

A natural idea is to combine both approaches and treat the mediators as *grey boxes* with varying level of transparency. This paper presents and studies experimentally an implementation of the grey-box approach in a new query compilation technique for P2P mediators - *distributed selective view expansion (DSVE)*. In DSVE for better performance mediators can control the level of transparency by selectively expanding only some multi-mediator views. To preserve their autonomy, mediator peers can decide whether to fulfill or not view definition requests. The performance improvements with DSVE are due to more selective queries, smaller data flows between the servers, fewer servers involved in the query execution while spending relatively little effort in query compilation.

DSVE is implemented in the AMOS II mediator system described in *Paper A*. The implementation is studied experimentally in two scenarios with up to 20 mediator peers to determine the effects of selective expansion of multi-

mediator views on the quality of QEPs. The study shows that one of the most important factors for the overall performance of a P2P mediator system is the topology of the logical mediator composition (i.e. of the graph defined by the mediator peers as graph nodes and the relationship 'defined in terms of' as graph arcs). Our experiments show that in mediator compositions with 10 and more peers DSVE reduces query compilation time with orders of magnitude with minor losses in the QEP quality and thus DSVE allows for efficient query processing in logically composed mediators.

Comments

I am the main author of the paper. My main contribution to the paper was in the scenario description, view expansion algorithm description and the experimental section. I proposed the idea to selectively expand views as a generalization of traditional full view expansion. An initial implementation of full view expansion was done by Vanja Josifovski. I modified and extended this implementation to support both full and partial view expansion. I also proposed and implemented the mediation scenario and designed and performed all experiments.

7.4 Paper D: Optimizing Queries in Distributed and Composable Mediators

One of the challenges in processing queries against many composed mediators is how to determine an optimal data flow between the mediators during query execution and where to compute intermediate join results. The approach typically taken in mediator architectures is that the mediator to which a query is posed, called *client mediator*, compiles locally and executes by itself a global QEP for that query. As a result all data and control flow pass through that mediator in a centralized manner, where all inter-mediator joins between its sub-mediators are computed. Depending on the quality of the physical links between all participating mediators and their processing resources, such centralized plans may not be always the most efficient. For example, when the links between the sub-mediators are faster than the links between them and the client mediator it may be more efficient to let those mediators directly exchange data and compute intermediate results without involving the client mediator. As pointed out in *Paper B*, for this mediators must support either a *plan shipping* or a *query shipping* functionality, so that the client mediator can instruct its sub-mediators to process global sub-queries that directly access other sub-mediators.

For the optimization of global queries over many levels of composed mediators we take a two-phase approach which we have implemented in the AMOS II PMS described in *Paper A*. In the first phase, the client mediator

decomposes global queries into sub-queries, each of them local with respect to some remote sub-mediator. To reduce the cost of query optimization, at this phase the optimizer searches only the space of left-deep QEPs. The resulting left-deep query plan tree, stored at the client mediator, specifies the order in which the client mediator will perform joins between the remote sub-queries in the sub-mediators, and may contain additional local operations. Since all joins are performed at the client mediator, this results in centralized plans.

In order to produce decentralized plans where sub-mediators communicate directly and perform some of the joins themselves, the initial QEP has to be decentralized into one or more global sub-plans that are computed by the sub-mediators. This is performed by a second query optimization step, called *query plan tree distribution*, or *tree distribution* for short. The tree distribution optimization phase is the focus of this paper.

One way to decentralize an initial QEP is to let the client mediator's optimizer explore all possible allocations of joins to sub-mediators and then to send join sub-plans (via plan shipping) to those sub-mediators using the plan shipping approach described in *Paper B*. In a PMS architecture this approach has several problems: *i)* it would require centralized decision making for which the client mediator would need to know the cost of executing joins by other mediators, *ii)* since there is no global catalog this would incur many costing requests, and *iii)* it is does not respect the autonomy of the sub-mediators.

A second possibility is to reuse the mediator's capability to accept sub-queries and locally compile them for further execution. Since each mediator provides a global query language, requests for the execution of global sub-plans can be submitted to remote sub-mediators in a declarative form using the query shipping approach and let the sub-mediators decide on the exact execution plan. In this way a global query is compiled cooperatively by the participating mediators, where each mediator both compiles and executes a piece the global QEP. The advantages of this approach are that: *i)* mediators can make cost estimates without performing remote cost requests, *ii)* mediator's autonomy is respected since each one can decide whether and how to execute a sub-query, and *iii)* better load distribution is achieved not only during query execution, but also at query compilation time.

The main contribution of this paper is a tree distribution algorithm based on the query shipping approach. The algorithm starts with a centralized left-deep global QEP. This initial tree is transformed by a series of plan node merge operations. A node merge operation takes two randomly chosen neighbor nodes, generates a sub-query that describes the join of the two nodes and replaces the two original nodes with one that accesses the merged sub-query. A node merge operation is performed only when it reduces the total QEP cost. For that the merged sub-query is compiled at both sub-mediators, the current plan cost is compared with the costs of the two new plans, and the cheapest of the three

is chosen. The process continues until no beneficial merge operations can be performed. The node merge operations replace the inner relations of the initial QEP with composite joins, therefore the algorithm essentially re-balances the initial centralized left-deep global QEP into a set of interacting distributed QEPs, which if looked at one plan distributed among many mediators would have a bushy instead of linear topology.

Comments

The general idea to re-balance distributed QEPs was suggested by Vanja Josifovski. I designed and implemented a distributed QEP rebalancing algorithm on top of the multi-mediator query compiler of AMOS II . I also designed, implemented and performed the experiments that evaluate the performance improvements resulting from the algorithm and wrote the experimental section of *Paper D* that describes the experimental results. The rest of the paper was written jointly by Vanja Josifovski, Tore Risch and myself.

7.5 Paper E: Evaluation of Join Strategies for Distributed Mediation

The distributed mediation architecture described in Sect. 3 and *Paper A* requires that mediators are able to cooperate at the physical level to compute answers of queries over integrated views. One of the most common tasks in data integration is to match overlapping entities in different sources. Since the mediators in the PMS architecture are essentially DBMS, matching of overlapping entities is logically expressed through a *join*. Join is one of the most expensive operations in a DBMS and therefore much attention has to be paid to its physical implementation. While many join variants have been proposed for centralized and distributed DBMS, a PMS system requires new algorithms that support inter-peer joins between mediators and sources with varying capabilities. Thus the design of join methods for a PMS have to take into account two aspects - efficiency and applicability. This paper proposes and evaluates three distributed join algorithms suitable for the computation of inter-mediator and mediator-source joins in a PMS.

Two *ship-out* algorithms ship data from a joining mediator towards the sources. In these algorithms, intermediate result tuples are shipped to the sources where they are used as parameters to remote subqueries or function calls. The first algorithm is an order-preserving semi-join, *PCA* which is suitable when there are no duplicates in the outer collection. The second algorithm, *SJMA*, uses a temporary hash index of possibly limited size to reduce the number of accesses to the data sources. It is suitable when there are duplicates in the outer collection. Both ship-out algorithms are streamed and the data is shipped between the mediator servers in bulks that contain several

tuples to avoid the message set-up overhead. The third algorithm is a *ship-in* join, where the data for the inner join operand is shipped from the remote source into the joining mediator.

The ship-out algorithms are applicable to joins with remote sources that need input data to execute local parameterized computations. If these computations are viewed as relations, then the sources are said to have limited capabilities because elements in these relations can not be retrieved by arbitrary attribute(s). To fully implement the algorithms the remote sources must be also able to accept and store locally whole bulks of data and then locally compute over them. The ship-in join algorithm is applicable to joins with remote sources that can ship to a mediator upon request the whole extent of a query or a computation. Such sources may or may not accept parameters. If they accept parameters, then both ship-out and ship-in join algorithms are applicable.

To analyze the performance of the three join algorithms we have fully implemented them in the PMS architecture presented in *Paper A*. Our performance study shows that the ship-out joins perform better that the ship-in join when: *i)* early first results are important, *ii)* joins are performed over slow lines, *iii)* mediator memory is limited. In particular, the PCA algorithm is simpler to implement, while the SJMA algorithm performs considerably better for outer collections with duplicates. The ship-in join generally performs better when the communication is over a fast network. Finally the ship-out algorithms shift the CPU load to the sources, while the ship-in join puts more of the CPU load on the join mediator.

Comments

The join algorithms described in this paper were proposed and implemented by Vanja Josifovski. I designed and performed the experiments and wrote the experimental section of the paper. Parts of the supporting code for the implementation was done by me together with various improvements necessary to make the implementation complete.

The published version of the paper contains a technical error - in Table 1 and Table 2 the resulting temporary relation tmp has to be inverted together with the final result of the example join.

7.6 Paper F: Object-Oriented Mediator Queries to Internet Search Engines

An important issue in design of a mediation system is its ability to easily incorporate new types of sources. In the mediation architecture presented in Sect. 3 and *Paper A*, mediators access data sources through wrapper components which interact with the mediator system through its facilities for extensibility - foreign functions, user-defined types and a call-level interface. The

work presented in this paper investigates the flexibility of the extensibility facilities related to the design and addition of new wrappers. For that, an "exotic" (from database view point) type of global sources is chosen - Internet search engines (*ISEs*). Internet search engines differ from typical database-like data sources in several ways:

- Their data access interfaces are non-standard, typically requiring programmatic access to HTML forms.
- Their contents is represented as semi-structured documents without an explicitly defined schema. The structure of the ISEs' content differs in structure among ISEs and even often changes over time for each ISE.
- ISEs do not have a standardized query language.

This requires that a system that accesses ISEs is very flexible. Due to the dynamic nature of the ISEs, it should be possible to easily modify and update existing ISE wrappers, preferably in a dynamic "on-the-fly" manner. Since the data delivered by ISEs have varying structures the mediator system has to be able to model the schemata of the ISEs and to reconcile the semantic differences between them. A large body of work exists that targets the problem of automatic schema extraction from semi-structured data. That is why a desirable feature of a wrapper solution for Web sources (as ISEs) is to easily incorporate new and existing *wrapper toolkits* that perform automatic schema extraction.

The paper describes a component of the AMOS II mediator system described in *Paper A*, called *ORWISE (Object-Relational Wrapper of Internet Search Engines)* that allows to easily add new ISE wrappers or update existing ones. Each kind of search engines is modeled as a subtype of the type *ISE* under *DataSource*, described in *Paper A*. New ISE wrappers are added to a mediator through the foreign function *orwise* that is overloaded for each ISE sub-type. Each implementation of *orwise* takes a query string in the language of the particular kind of ISE (e.g. Google) and invokes the wrapper specific for that kind of ISE through the ORWISE component. The ISE wrapper submits the ISE query through a low-level wrapper generated by a wrapper toolkit to the ISE. The data returned by the ISE is then parsed by the low-level wrapper typically into strings. Finally the ORWISE component semantically enriches the resulting ISE data by translating it into objects of type *DocumentView* that describe Web documents. This enrichment uses routines built-in ORWISE that map strings into AMOS II types.

In summary, the ORWISE component provides *i)* the ISE schema for describing and querying data from any ISE in terms of subtypes of type *DataSource* and the overloaded function *orwise*, *ii)* a mechanism to specify search engine specific translators by redefining *orwise* and adding new ISE subtypes, and *iii)* facilities to allow different wrapper toolkits to be easily plugged into the system.

The design of ORWISE shows shows how to include a global data source the PMS framework. In addition it shows that the approach to use foreign functions, overloading and user-defined types to develop new wrappers is indeed very flexible and can easily accommodate even non-database-like global data sources as ISEs.

Comments

The initial idea to wrap ISEs proposed by myself. I also designed the ORWISE component with discussions with Simon Zürcher. Simon Zürcher implemented and tested ORWISE. The paper was written jointly by me and Tore Risch using as a basis a technical report from Simon Zürcher.

Future Work

The presented mediation architecture poses a wide range of problems to be solved as shown by our analysis of requirements in Sect. 3.2. The fulfillment of each of these requirements is a research area of its own. Here we focus on some future directions that follow directly from the main focus of this work - scalable performance in composable mediators.

Topology-aware heuristics for view expansion

A direct continuation of the work presented in *Paper C* is to design an efficient heuristic for selective view expansion that utilizes the knowledge of the topology of the logical composition of mediators and targets the view expansion process towards those mediator views that will produce highest increase in QEP quality with the least compilation effort. In our ongoing work we evaluate several such heuristics.

Adaptivity in mediator compositions

Ideally query processing in a PMS should scale up to hundreds and even thousands of mediator peers. In most cases it is impossible to perform precise cost and selectivity estimates when integrating many mediators and diverse data sources over a global network. This may lead to sub-optimal query execution plans. Even if all necessary statistics information is available it is also infeasible to perform full cost-based query optimization in the traditional System R style due the potentially very large number of mediators, sources and views. Our current experience from experiments with mediator compositions of over 20 mediators show that incorrect cost and selectivity estimates can lead to orders of magnitude worse query execution plans (QEP). Several factors specific to peer mediators contribute to the incorrect cost estimates. In most cases it is not possible to acquire statistics about the data stored in the data sources. This is even harder when the data in a source is actually computed and not stored. Imprecise cost modeling may result in that the errors in cost and selectivity estimates increase by orders when propagated through many mediators. Finally data sources, network conditions and mediator load can all change in an unpredictable manner. Therefore it is essential for a mediator system to adapt to an unpredictable and changing environment.

Adaptive query processing for single-site query processors has been addressed by various works [54, 26, 4], to name a few. A good overview of

adaptive query processing can be found in [21, 15]. Many of the proposed approaches can be integrated with the solution proposed here to implement adaptive behavior of each of the mediator peers. However these approaches do not address all the complexity of the problem of adaptivity in a P2P mediator architecture. A centralized query processor usually has direct access to the data structures of a QEP and therefore it has the full power to modify the QEP at any time and adapt its execution accordingly. In a P2P mediator system a QEP is distributed among all peers participating in the evaluation of a query. Because of autonomy, no peer has direct access to the fragments of a global QEP in the other peers. Instead, the query processors of autonomous peers have to cooperate through network protocols in order to change a global QEP and adapt during query processing. Thus adaptivity in P2P mediators requires not only single-site adaptation, but also cooperative adaptation by all participating peers, so that sub-optimal global execution plans can gradually converge to more efficient ones.

Integrated self-profiling

As a basis for adaptivity, mediator systems should be able to measure various parameters of their environment and their own operation and that of neighbor sources and mediators, store this measurements and use them to detect sub-optimality and to adapt by recomputing the affected QEPs.

One approach to measure system performance and manage measurement data is to integrate a database-based profiling system with the query processor of each mediator peer. This will enable the query processor of a mediator to measure parameters related to its own operation, the sources it accesses and the network, and then use the accumulated information for better future decisions. The main idea behind such an integrated profiling approach is to use the mediator system itself in a reflective manner to store all measurement data in the database itself. The benefits of this approach are that the full power of the mediator query language will be available to update, retrieve and analyze the distributed measurement data. Potentially there may be large amount of profile data with dynamically changing distribution across many mediator peers. Using the global query capabilities of the mediator system in a reflective manner to access the profile data would allow to let the system automatically compute the best access path to the data without the need to hard-code it and to easily modify the decision-making procedures inside the optimizer.

With a main-memory mediator database system, such as AMOS II, we can expect very fast updates and retrievals of the measurement data. This will allow to minimize the the performance penalty of profiling during normal system usage. The extensibility of AMOS II allows to define custom data structures and functions to store and update profiling data in the most efficient manner while still preserving a query interface to that data. Finally the architecture

of the AMOS II mediator system allows any system component to be profiled in a generic manner. An interesting direction is to profile the operation of all critical components of the query engine and to introduce adaptivity not only at the level of the query execution plans but other system components as well, e.g. the query compiler itself.

The major challenges are how to minimize the performance penalty of profiling, to ensure that the necessary profiling data can be accessed very fast as this will be done from inside the query engine and finally the ability to dynamically control what parameters are being measured.

Adaptive rebalancing of global QEPs

One potentially useful application of the integrated self-profiling is to adapt the distributed data flow of global QEPs. In *Paper D* we investigated rebalancing of global QEPs that allows the query compiler to generate decentralized plans at each mediator. QEP rebalancing takes a centralized plan where all communication between one mediator and all its direct sub-mediators passes through the controlling mediator and transforms it whenever favorable into a plan with side-wise information passing, where some of the communication is performed directly between the sub-mediators. For this sub-plans of the centralized QEP are sent to the nearest mediators (in terms of logical composition) and further compilation of the sub-plans is delegated to neighbor peers. The peers in turn may further decide to apply rebalancing to the sub-plans received for compilation.

While *Paper D* shows that distributed QEP rebalancing removes some of the overhead of logical mediator composition, this is done in a static manner. Future work for this project is to extend QEP tree rebalancing to allow mediators to automatically adapt the data flow of distributed QEPs to changes that may occur in a P2P mediator system.

Important research issues related to adaptive QEP rebalancing, and to adaptivity in general are: detecting sub-optimal performance and adapting to it; reuse parts of a QEP when re-adapting to save compilation work; reuse of the intermediate query execution results - if only some of mediators' plans are reoptimized only the execution of a sub-plan could be restarted instead of recomputing the whole result from scratch.

References

[1] *Object Management Architecture*. John Wiley & Sons, New York, 1995.

[2] SOAP Version 1.2 Part 0: Primer. W3C Candidate Recommendation, http://www.w3.org/TR/soap12-part0/, December 2002.

[3] Karl Aberer and Manfred Hauswirth. An Overview on Peer-to-Peer Information Systems. In *Proceedings of Workshop on Distributed Data and Structures (WDAS-2002)*, 2002.

[4] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. *ACM SIGMOD Record*, 29(2):261–272, 2000.

[5] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[6] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Workshop on the Web and Databases, WebDB 2002*, Madison, Wisconsin, June 2002. SIGMOD 2002.

[7] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[8] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[9] Ruxandra Domenig and Klaus R. Dittrich. An Overview and Classification of Mediated Query Systems. *SIGMOD Record*, 28(3):63–72, 1999.

[10] W. Du and M. Shan. Query Processing in Pegasus. In *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Pretince Hall, Englewood Cliffs, 1996.

[11] Weimin Du and Ahmed K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 347–355. Morgan Kaufmann, August 1989.

[12] Gustav Fahl and Tore Risch. Query Processing Over Object Views of Relational Data. *VLDB Journal*, 6(4):261–281, 1997.

[13] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)*, 8(2):117–132, April 1997.

[14] David Garlan. Research directions in software architecture. *ACM Computing Surveys (CSUR)*, 27(2):257–261, 1995.

[15] Anastasios Gounaris, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakellariou. Adaptive Query Processing: A Survey. In *Proc. 19th British National Conference on Databases, BNCOD*, Sheffield, UK, July 2002. Springer-Verlag.

[16] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *WebDB Workshop on Databases and the Web*, June 2001.

[17] Laura Haas, Eileen Lin, and Mary Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–, 2002.

[18] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of 23rd International Conference on Very Large Data Bases, VLDB'97*, pages 276–285, Athens, Greece, August 1997. Morgan Kaufmann.

[19] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proceedings of the twelfth international conference on World Wide Web*, pages 556–567. ACM Press, 2003.

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

[21] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[22] Richard Hull. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 51–61. ACM Press, May 1997.

[23] Vanja Josifovski and Tore Risch. Functional Query Optimization over Object-Oriented Views for Data Integration. *Journal of Intelligent Information Systems*, 12(2-3):165–190, 1999.

[24] Vanja Josifovski and Tore Risch. Integrating Heterogenous Overlapping Databases through Object-Oriented Transformations. In *Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99*, pages 435–446. Morgan Kaufmann, September 1999.

[25] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. Garlic: a new flavor of federated query processing for DB2. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 524–532. ACM Press, 2002.

[26] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 106–117, Seattle, Washington, USA, June 1998. ACM Press.

[27] Vipul Kashyap and Amit P. Sheth. Semantic and Schematic Similarities Between Database Objects: A Context-Based Approach. *VLDB Journal*, 5(4):276–304, 1996.

[28] Won Kim, Injun Choi, Sunit Gala, and Mark Scheevel. On resolving schematic heterogeneity in multidatabase systems. pages 521–550, 1995.

[29] Milena Gateva Koparanova and Tore Risch. Completing CAD Data Queries for Visualization. In *International Database Engineering & Applications Symposium*, pages 130–139. IEEE Computer Society, 2002.

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

[31] Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM Press, 2002.

[32] Alon Y. Levy. Logic-based techniques in data integration. pages 575–595, 2000.

[33] Scott M. Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.

[34] Witold Litwin and Tore Risch. Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, 1992.

[35] Ling Liu and Calton Pu. An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed and Parallel Databases*, 5(2):167–205, April 1997.

[36] Ling Liu, Ling Ling Yan, , and M. Tamer Özsu. Interoperability in Large-Scale Distributed Information Delivery Systems. In *Advances in Workflow Systems and Interoperability*, pages 246–280. Springer-Verlag, 1998.

[37] Hongjun Lu, Beng-Chin Ooi, and Cheng-Hian Goh. Multidatabase query optimization: issues and solutions. In *Proceedings RIDE-IMS '93., Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 137–143, Vienna, Austria, April 1993.

[38] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[39] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna G. Kulkarni, and Peter M. Schwarz. SQL/MED - A Status Report. *SIGMOD Record*, 31(3), 2002.

[40] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, HP Labs, 2002.

[41] H. Garcia Molina and B. Kogan. Node autonomy in distributed systems. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 158–166. IEEE Computer Society Press, 1988.

[42] Wee Siong Ng, Beng Chin Ooi, Lee Tan, and Aoying Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *19th International Conference on Data Engineering*, March 2003.

[43] Aris M. Ouksel and Amit P. Sheth. Semantic interoperability in global information systems. *ACM SIGMOD Record*, 28(1):5–12, 1999.

[44] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, second edition edition, 1999.

[45] M. Tamer Özsu and Bin Yao. Building component database systems using CORBA. pages 207–236, 2001.

[46] Kirill Richine. Distributed Query Scheduling in The Context of DIOM: An Experiment. Tech. report TR97-03, Department of Computing Science, University of Alberta, 1997.

[47] Fèlix Saltor and Elena Rodríguez. On Intelligent Access to Heterogeneous Information. In *Intelligent Access to Heterogeneous Information, Proceedings of the 4th Workshop KRDB-97*, volume 8 of *CEUR Workshop Proceedings*, pages 151–157, August 1997.

[48] David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *TODS*, 6(1):140–173, 1981.

[49] Clay Shirky. What Is P2P ... And What Isn'. http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html, November 2000.

[50] Michael Stal. Web services: beyond component-based computing. *Communications of the ACM*, 45(10):71–76, 2002.

[51] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[52] Jeffrey D. Ullman. Information Integration Using Logical Views. In *6th International Conference on Database Theory - ICDT '97*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, January 1997.

[53] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[54] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost Based Query Scrambling for Initial Delays. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 1998*, pages 130–141, Seattle, Washington, USA, June 1998. ACM Press.

[55] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[56] Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, Sept.-Oct. 1997. also in IEEE Intelligent Systems.

[57] Ling-Ling Yan, Rene J. Miller, Laura M. Haas, and Ronald Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *ACM SIGMOD Conference*, May 2001.

[58] Ling-Ling Yan, M. Tamer Özsu, and Ling Liu. Accessing Heterogeneous Data Through Homogenization and Integration Mediators. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, pages 130–139. IEEE Computer Society, 1997.

[59] Beverly Yang and Hector Garcia-Molina. Designing a Super-peer Network. In *IEEE International Conference on Data Engineering*, March 2003.

Acta Universitatis Upsaliensis

Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series *Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology*. (Prior to October, 1993, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science".)

Distribution:
Uppsala University Library
Box 510, SE-751 20 Uppsala, Sweden
www.uu.se, acta@ub.uu.se

ISSN 1104-232X
ISBN 91-554-5770-3

Paper A:

Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional data integration in a distributed mediator system. In *The Functional Approach to Data Management*. Springer-Verlag, 2003.

Functional Data Integration in a Distributed Mediator System

Tore Risch
Uppsala University
Tore.Risch@it.uu.se

Vanja Josifovski
IBM Almaden Research Center
vanja@us.ibm.com

Timour Katchaounov
Uppsala University
Timour.Katchaounov@it.uu.se

Abstract

Amos II (Active Mediator Object System) is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AmosQL. Through its distributed multi-database facilities many autonomous and distributed Amos II peers can interoperate. Functional multi-database queries and views can be defined where external data sources of different kinds are translated through Amos II and reconciled through its functional mediation primitives. Each mediator peer provides a number of transparent functional views of data reconciled from other mediator peers, wrapped data sources, and data stored in Amos II itself. The composition of mediator peers in terms of other peers provides a way to scale the data integration process by composing mediation modules. The Amos II data manager and query processor are extensible so that new application oriented data types and operators can be added to AmosQL, implemented in some external programming language (Java, C, or Lisp). The extensibility allows wrapping data representations specialized for different application areas in mediator peers. The functional data model provides very powerful query and data integration primitives which require advanced query optimization.

1 Introduction

The mediator/wrapper approach, originally proposed by [43], has been used for integrating heterogeneous data in several projects, e.g. [16, 42, 14, 5]. Most mediator systems integrate data through a central mediator server accessing one or several data sources through a number of 'wrapper' interfaces that translate data to a common data model (CDM). However, one of the original goals for mediator architectures [43] was that mediators should be relatively simple distributed software modules that transparently encode domain-specific knowledge about data and share abstractions of that data with higher layers of mediators or applications. Larger networks of mediators would then be defined

through these primitive mediators by composing new mediators in terms of other mediators and data sources.

The core of Amos II is a open, light-weight, and extensible database management system (DBMS) with a functional data model. Each Amos II server contains all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, and a functional query language named AmosQL. The system can be used as a single-user database or as a multi-user server to applications and to other Amos II peers.

Distribution

Amos II is a distributed mediator system where several mediator peers communicate over the Internet. Each mediator peer appears as a virtual functional database layer having data abstractions and a functional query language. Functional views provide transparent access to data sources from clients and other mediator peers. Conflicts and overlaps between similar real-world entities being modeled differently in different data sources are reconciled through the mediation primitives [18, 17] of the multi-mediator query language AmosQL. The mediation services allow transparent access to similar data structures represented differently in different data sources. Applications access data from distributed data sources through queries to views in some mediator peer.

Logical composition of mediators is achieved when *multi-database views* in mediators are defined in terms of views, tables, and functions in other mediators or data sources. The multi-database views make the mediator peers appear to the user as a single virtual database. Amos II mediators are composable since a mediator peer can regard other mediator peers as data sources.

Wrappers

In order to access data from external data sources Amos II mediators may contain one or several *wrappers* which process data from different kinds of external data sources, e.g. ODBC based access to relational databases [11, 4], access to XML files [28], CAD systems [25], or Internet search engines [22]. A wrapper is a program module in Amos II having specialized facilities for query processing and translation of data from a particular class of external data sources. It contains both interfaces to external data sources and knowledge of how to efficiently translate and process queries involving accesses to a class of external data sources. In particular external Amos II peers known to a mediator are also regarded as external data sources and there is a special wrapper for accessing other Amos II peers. However, among the Amos II peers special query optimization methods are used that take into account the distribution, capabilities, costs, etc. of the different peers [20].

The name server

Every mediator peer must belong to a group of mediator peers. The mediator peers in a group are described through a meta-schema stored in a mediator server called *name server*. The mediator peers are autonomous and there is

no central schema in the name server. The name server contains only some general meta-information such as the locations and names of the peers in the group while each mediator peer has its own schema describing its local data and data sources. The information in the name server is managed without explicit operator intervention; its content is managed through messages from the mediator peers. To avoid a bottleneck, mediator peers usually communicate directly without involving the name server; it is normally involved only when a connection to some new mediator peer is established.

AmosQL

AmosQL is functional language having its roots in the functional query languages OSQL [31] and DAPLEX [38] with extensions of mediation primitives [18, 17], multi-directional foreign functions [29], late binding [13], active rules [39], etc. Queries are specified using the select - from - where construct as in SQL. AmosQL furthermore has aggregation operators, nested subqueries, disjunctive queries, quantifiers, and is relationally complete.

Query optimization

The declarative multi-database query language AmosQL requires queries to be optimized before execution. The query compiler translates AmosQL statements first into *object calculus* and then into *object algebra* expressions. The object calculus is expressed in an internal simple logic based language called ObjectLog [29], which is an object-oriented dialect of Datalog. As part of the translation into object algebra programs, many optimizations are applied on AmosQL expressions relying on its functional and multi-database properties. During the optimization steps, the object calculus expressions are re-written into equivalent but more efficient expressions. For distributed multi-database queries a multi-database query decomposer [20] distributes each object calculus query into local queries executed in the different distributed Amos II peers and data sources. For better performance, the decomposed query plans are rebalanced over the distributed Amos II peers [17]. A cost-based optimizer on each site translates the local queries into procedural execution plans in the object algebra, based on statistical estimates of the cost to execute each generated query execution plan expressed in the object algebra. A query interpreter finally interprets the optimized algebra to produce the result of a query.

Multi-directional foreign functions

The query optimizer is extensible through a generalized foreign function mechanism, *multi-directional foreign functions*. It gives transparent access from AmosQL to special purpose data structures such as internal Amos II metadata representations or user defined storage structures. The mechanism allows the programmer to implement query language operators in an external language (Java, C or Lisp) and to associate costs and selectivity estimates with different user-defined access paths. The architecture relies on extensible optimization of such foreign function calls [29]. They are important both for accessing external

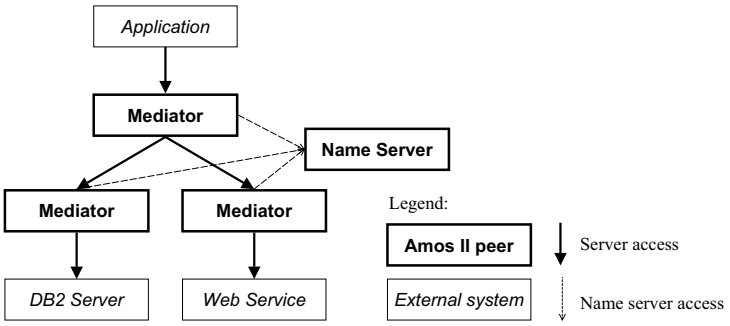


Figure 1: Distributed mediator communication

query processors [4] and for integrating customized data representations from data sources.

Organization

Next the distributed mediator architecture of Amos II is described. Then the functional data model used in Amos II is described along with its functional query language followed by a description of how the basic functional data model is extended with data integration primitives. After that there is an overview of the distributed multi-mediator query processing. Finally, related work is discussed followed by a summary.

2 Distributed Mediation

Groups of distributed Amos II peers can interoperate over a network using TCP/IP. This is illustrated by Fig. 1 where an application accesses data from two distributed data sources through three distributed mediator peers. The thick lines indicate communication between peers where the arrows indicate peers acting as servers.

The *name server* is a mediator peer storing names, locations, and other general data about the mediators in a group. As illustrated by the dotted lines, mediators in a group communicate with the name server to register themselves in the group or obtain information about other peers.

The figure furthermore illustrates that several layers of mediator peers can call other mediator peers. Notice, however, that the communication topology is dynamic and any peer can communicate directly with any other peer or data source in a group. It is up to the distributed mediator query optimizer to automatically come up with the optimal communication topology between the peers for a given query. The query optimizers of the peers can furthermore exchange both data and schema information in order to produce an optimized distributed execution plan.

In the figure, the uppermost mediator defines mediating functional views integrating data from them. The views include facilities for semantic reconciliation of data retrieved from the two lower mediators.

The two lower mediators translate data from a wrapped relational database and a web server, respectively. They have knowledge of how to translate AmosQL queries to SQL [11] through JDBC and, for the web server, to web service requests.

When an Amos II system is started, it initially assumes stand-alone single-user mode of operation in which no communication with other Amos II systems can be done. The stand-alone system can join a group by issuing a *registration* command to the name server of the group. Another system command makes the mediator a peer that accepts incoming commands from other peers in the group.

In order to access data from external data sources Amos II mediators may contain one or several *wrappers* to interface and process data from external data sources. A wrapper is a program module in a mediator having specialized facilities for query processing and translation of data from a particular kind of external data sources. It contains interfaces to external data repositories to obtain both meta-data (schema definitions) and data. It also includes data source specific rewrite rules to efficiently translate and process queries involving accesses to a particular kind of external data source. More specifically the wrappers perform the following functions:

- *Schema importation* translates schema information from the sources into a set of Amos II types and functions.
- *Query translation* translates internal calculus representations of AmosQL queries into equivalent API calls or query language expressions executable by the source.
- *Source statistics computation* estimates costs and selectivities for API calls or query expressions to a data source.
- *Proxy OID generation* executes in the source query expressions or API calls to construct *proxy OIDs* describing source data.
- *OID verification* executes in the source query expressions or API calls to verify the validity of involved proxy OIDs, in case they have become invalid between different query requests.

Once a wrapper has been defined for a particular kind of source, e.g. ODBC or a web service, the system knows how to process any AmosQL query or view definition for all such sources. When integrating a new instance of the source the mediator administrator can define a set of views in AmosQL that provide abstractions of it.

Different types of applications require different interfaces to the mediator layer. For example, there are call level interfaces allowing AmosQL statements to be embedded in the programming languages Java, C, and Lisp. The call-in interface for Java has been used for developing a Java-based multi-database object browser, GOOVI [6].

The Amos II kernel can also be extended with plug-ins for customized query optimization, fusion of data, and data representations (e.g. matrix data). Often specialized algorithms are needed for operating on data from a particular application domain. Through the plug-in features of Amos II , domain oriented algorithms can easily be included in the system and made available as new query language functions in AmosQL. It is furthermore possible to add new query transformation rules (re-write rules) for optimizing queries over the new domain.

3 Functional Data Model

The data model of Amos II is an extension of the Daplex [38] functional data model. The basic concepts of the data model are *objects, types,* and *functions.*

3.1 Objects

Objects model all entities in the database. The system is reflective is the sense that everything in Amos II is represented as objects managed by the system, both system and user-defined objects. There are two main kinds of representations of objects: *literals* and *surrogates.* The surrogates have associated object identifiers (OIDs), which are explicitly created and deleted by the user or the system. Examples of surrogates are objects representing real-world entities such as persons, meta-objects such as functions, or even Amos II mediators as meta-mediator objects.

The literal objects are self-described system maintained objects which do not have explicit OIDs. Examples of literal objects are numbers and strings. Literal objects can also be *collections*, representing collections of other objects. The system-supported collections are *bags* (unordered sets with duplicates) and *vectors* (order-preserving collections). Literal are automatically deleted by an incremental garbage collector when they are no longer referenced in the database.

3.2 Types

Objects are classified into *types* making each object an *instance* of one or several types. The set of all instances of a type is called the *extent* of the type. The types are organized in a multiple inheritance, supertype/subtype hierarchy. If an object is an instance of a type, then it is also an instance of all the supertypes of that type; conversely, the extent of a type is a subset of all extents of the supertypes of that type (extent-subset semantics). For example if the type `Student` is a subtype of type `Person`, the extent of type `Student` is also a subset of the extent of type `Person`. The extent of a type which is multiple inherited from other types is a subset of the intersection of its supertypes' extents.

There are two kinds of types, *stored* and *derived* types. Derived types are used mainly for data reconciliation and are described in the next section.

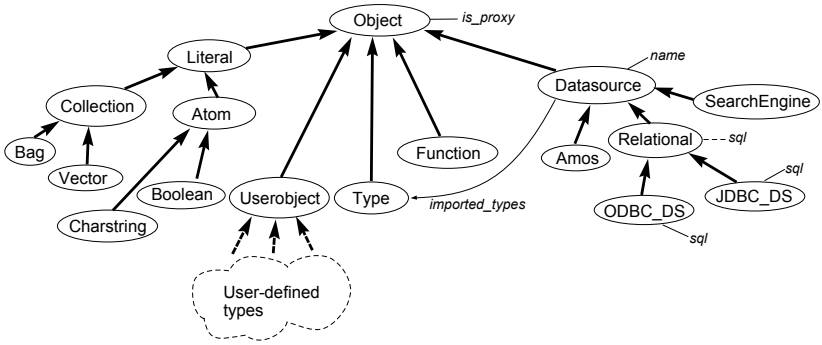


Figure 2: System type hierarchy

Stored types are defined and stored in an Amos II peer through the `create type` statement, e.g.:

```
create type Person;
create type Student under Person;
create type Teacher under Person;
create type TA under Student, Teacher;
```

The above statements extend the database schema with four new types: A `TA` object is both a `Student` and a `Teacher`. The extent of type `Person` is the union of all objects of types `Person, Student, Teacher,` and `TA`. The extent of type `TA` is the intersection of the extents of types `Teacher` and `Student`.

All objects in the database are typed, including meta-objects such as those representing the types themselves. The meta-objects representing types are also stored types and instances of the meta-type named `Type`. In the example the extent of the type named `Type` is the meta-objects representing the types named `TA, Teacher, Student,` and `Person`.

The root in the type hierarchy is the system type named `Object`. The system type `Userobject` is the root of all user defined types and the extent of type `Userobject` contains all user-defined objects in the database.

The major root types in the type hierarchy are illustrated by the *function diagram* on Fig. 2 where ovals denote types, thin arrows denote functions, thick arrows denote type inheritance, and literal function result types are omitted for readability. The type `Datasource` and its subtypes and functions are explained later in Sect. 4.2.

Every object has an associated *type set*, which is the set of those types that the object is an instance of. Every object also has one *most specific type* which is the type specified when the object is created. The full type set includes the most specific type and all types above the type in the type hierarchy. For example, objects of type `TA` have the most specific type named `TA` while its full type set is {`TA, Teacher, Student, Person, Userobject, Object`}.

The type set of an object can dynamically change during the lifetime of the object through AmosQL statements that change the most specific type of

an object. The reason for such facilities is because the *role* of an object may change during the lifetime of the database. For example, a TA might become a student for a while and then a teacher.

3.3 Functions

Functions model the semantics (meaning) of objects. They model properties of objects, computations over objects, and relationships between objects. They furthermore are basic primitives in functional queries and views. Functions are instances of the system type `Function`.

A function consists of two parts, the *signature* and the *implementation*:

The *signature* defines the types, and optional names, of the argument(s) and the result of a function. For example, the signature of the function modeling the attribute `name` of type `Person` would have the signature:

```
name(Person)->Charstring
```

Functions can be defined to take any number of arguments, e.g. the arithmetic addition function implementing the infix operator '+' has the signature:

```
plus(Number,Number)->Number
```

The *implementation* specifies how to compute the result of a function given a tuple of argument values. For example, the function `plus` computes the result by adding the two arguments, and `name` obtains the name of a person by accessing the database. The implementation of a function is normally non-procedural, i.e. a function only computes result values for given arguments and does not have any side effects. The exception is *database procedures* defined through procedural AmosQL statements.

Furthermore, Amos II functions are often *multi-directional* meaning that the system is able to inversely compute one or several argument values if (some part of) the expected result value is known [29]. Inverses of multi-directional functions can be used in database queries and are important for specifying general queries with function calls over the database. For example, the following query, which finds the age of the person named 'Tore', uses the inverse of function `name` to avoid iterating over the entire extent of type `Person`:

```
select age(p) from Person p where name(p)='Tore';
```

Depending on their implementation the basic functions can be classified into *stored, derived,* and *foreign* functions. In addition, there are *database procedures* with side effects and *proxy* functions for multi-mediator access as explained later.

- *Stored functions* represent properties of objects (attributes) locally stored in an Amos II database. Stored functions correspond to attributes in object-oriented databases and tables in relational databases.

- *Derived functions* are functions defined in terms of functional queries over other Amos II functions. Derived functions cannot have side effects

and the query optimizer is applied when they are defined. Derived functions correspond to side-effect free methods in object-oriented models and views in relational databases. AmosQL has an SQL-like *select* statement for defining derived functions and ad hoc queries.

- *Foreign functions* provide the low level interfaces for wrapping external systems from Amos II. For example, data structures stored in external storage managers can be manipulated through foreign functions. Foreign functions can also be defined for updating external data structures, but foreign functions to be used in queries must be side effect free.

  Foreign functions correspond to methods in object-oriented databases. Amos II furthermore provides a possibility to associate several implementations of inverses of a given foreign function, *multi-directional foreign functions*, which informs the query optimizer that there are several access paths implemented for the function. To help the query processor, each associated access path implementation may have associated cost and selectivity functions. The multi-directional foreign functions provide access to external storage structures similar to data 'blades', 'cartridges', or 'extenders' in object-relational databases.

- *Database procedures* are functions defined using a procedural sublanguage of AmosQL. They correspond to methods with side effects in object-oriented models and constructors. A common usage is for defining constructors of objects along with associated properties.

Amos II functions can furthermore be *overloaded* meaning that they can have different implementations, called *resolvents*, depending on the type(s) of their argument(s). For example, the salary may be computed differently for types `Student` and `Teacher`. Resolvents can be any of the basic function types[1]. Amos II's query compiler chooses the resolvent based on the types of the argument(s), but not the result.

The *extent* of a function is a set of tuples mapping its arguments and its results. For example, the extent of the function defined as

```
create function name(Person)-> Charstring as stored;
```

is a set of tuples $< P_i, N_i >$ where P_i are objects of type `Person` and N_i are their corresponding names. The extent of a stored function is stored in the database and the extent of a derived function is defined by its query. The extents are accessed in database queries.

The structure of the data associated with types is defined through a set of function definitions. For example:

```
create function name(Person) -> Charstring as stored;
create function birthyear(Person) -> Integer as stored;
create function hobbies(Person) -> Bag of Charstring as stored;
create function name(Course) -> Charstring as stored;
create function teaches(Teacher) -> Bag of Course as stored;
```

[1] A resolvent cannot be overloaded itself, though.

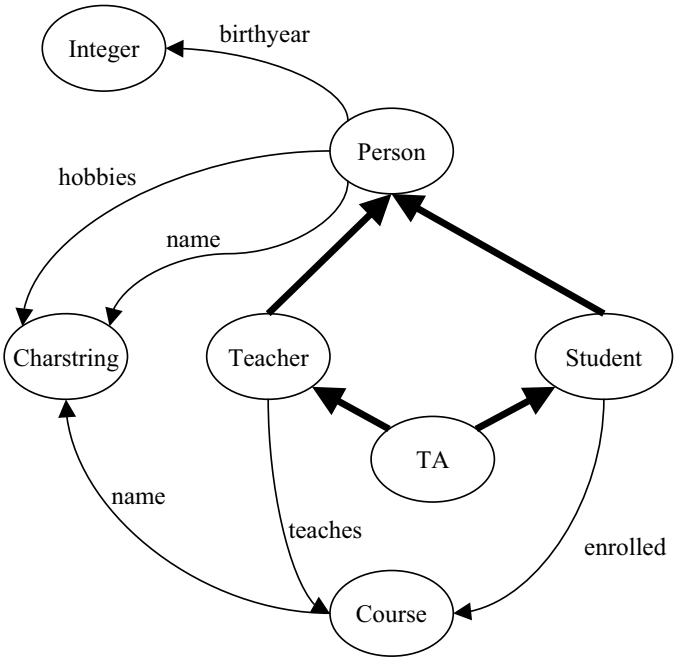


Figure 3: Function diagram

```
create function enrolled(Student) -> Bag of Course as stored;
create function instructors(Course c) -> Bag of Teacher t as
     select t
     where teaches(t) = c; /* Inverse of teaches */
```

The above stored function and type definitions can be illustrated with the function diagram of Fig. 3.

The function `name` is overloaded on types `Person` and `Course`. The function `instructors` is a derived function that uses the inverse of function `teaches`. The functions `hobbies, teaches`, and `enrolled` return sets of values. If 'Bag of' is declared for the value of a stored function it means that the result of the function is a bag (multiset)[2], otherwise it is an atomic value.

Functions (attributes) are inherited so the above statement will make objects of type `Teacher` have the attributes `name, birthyear, hobbies`, and `teaches`.

We notice here that single argument Amos II *functions* are similar to *relationships* and *attributes* in the entity-relationship (ER) model and that Amos II *types* are similar to ER *entities*. The main difference between an Amos II function and an ER relationship is that Amos II functions have a logical *direction* from the argument to the result, while ER entities are direction neutral. Notice that Amos II functions normally are invertible and thus can be used in

[2]DAPLEX uses the notation `->>` for sets.

the inverse direction too. The main difference between Amos II types and the entities in the basic ER model is that Amos II types can be inherited.

Multi-directional foreign functions

As a very simple example of a multi-directional foreign function, assume we have an external disk-based hash table on strings to be accessed from Amos II. We can then implement is as follows:

```
create function get_string(Charstring x)-> Charstring r
       as foreign "JAVA:Foreign/get_hash";
```

Here the foreign function `get_string` is implemented as a Java method `get_hash` of the public Java class `Foreign`. The Java code is dynamically loaded when the function is defined or the mediator initialized. The Java Virtual Machine is interfaced with the Amos II kernel through the Java Native Interface to C.

Multi-directional foreign functions include declarations of inverse foreign function implementations. For example, our hash table can not only be accessed by keys but also scanned, allowing queries to find all the keys and values stored in the table. We can generalize it by defining:

```
create function get_string(Charstring x)->Charstring y
   as multidirectional
      ("bf" foreign "JAVA:Foreign/get_hash"
            cost {100,1})
      ("ff" foreign "JAVA:Foreign/scan_hash"
            cost "scan_cost");
```

Here, the Java method `scan_hash` implements scanning of the external hash table. Scanning will be used, e.g., in queries retrieving the hash key for a given hash value. The *binding patterns*, `bf` and `ff`, indicate whether the argument or result of the function must be bound (`b`) or free (`f`) when the external method is called.

The cost of accessing an external data source through an external method can vary heavily depending on, e.g., the binding pattern, and, to help the query optimizer, a foreign function can have associated costing information defined as user functions. The `cost` specifications estimate both *execution costs* in internal cost units and *result sizes* (fanouts) for a given method invocation. In the example, the cost specifications are constant for `get_hash` and computed through the Amos II function `scan_cost` for `scan_hash`.

The basis for the multi-directional foreign function was developed in [29], where the mechanisms are further described.

3.4 Queries

General queries are formulated through the `select` statement with format:

```
select <result>
from   <type extents>
where  <condition>
```

For example:

```
select name(p), birthyear(p)
from   Person p
where  birthyear(p) > 1970;
```

The above query will retrieve a tuple of the names and birth years of all persons in the database born after 1970.

In general the semantics of an AmosQL query is as follows:

1. Form the cartesian product of the *type extents*.
2. Restrict the cartesian product by the *condition*.
3. For each possible variable binding to tuple elements in the restricted cartesian product, evaluate the result expressions to form a result tuple.
4. Result tuples containing NIL are not included in the result set; queries are *null intolerant*.

It would be very inefficient to directly use the above semantics to execute a query. It is therefore necessary for the system to do extensive query optimization to transform the query into an efficient execution strategy. Actually, unlike in SQL, AmosQL permits formulation of queries accessing indefinite extents and such queries are not executable at all without query optimization. For example, the previous query could also have been formulated as:

```
select nm, b
from Person P, Charstring nm, Integer b
where b = birthyear(p) and
      nm = name(p) and
      b > 1970;
```

In this case, the cartesian product of all persons, integers, and strings is infinite so the above query is not executable without query optimization.

Some function may not have a fully computable extent, e.g. arithmetic functions have an infinitely large extent. Queries over infinite extents are not executable, e.g. the system will refuse to execute this query:

```
select x+1 from Number x;
```

4 Functional Mediation

For supporting multi-database queries, the basic data model is extended with *proxy* objects, types, and functions. Any object, including meta-objects, can be defined by Amos II as a proxy object by associating with it a property describing its source. The proxy objects allow data and meta-data to be transparently exchanged between mediator peers.

On top of this, reconciliation of conflicting data is supported through regular stored and derived functions and through *derived types* (DTs) [18, 19] that define types through declarative multi-database queries.

4.1 Proxy objects

The distributed mediator architecture requires the exchange of objects and meta-data between mediator peers and data sources. To support multi-database queries and views, the basic concepts of objects, types, and functions are generalized to include also *proxy objects*, *proxy types*, and *proxy functions*:

- *Proxy objects* in a mediator peer are local OIDs having associated descriptions of corresponding objects stored in other mediators or data sources. They provide a general mechanism to define references to remote objects.
- *Proxy types* in a mediator peer describe types represented in other mediators or data sources. The proxy objects are instances of some proxy types and the extent of a proxy type is a set of proxy objects.
- Analogously, *proxy functions* in a mediator peer describe functions in other mediators or sources.

The proxy objects, types and functions are implicitly created by the system in the mediator where the user makes a multi-database query, e.g.:

```
select name(p) from Personnel@Tb p;
```

This query retrieves the names of all persons in a data source named `Tb`. It causes the system to internally generate a proxy type for `Personnel@Tb` in the mediator server where the query is issued, M. It will also create a proxy function `name` in M representing the function `name` in `Tb`. In this query it is not necessary or desirable to create any proxy instances of type `Personel@Tb` in M since the query is not retrieving their identities. The multi-database query optimizer will here make such an optimization.

Proxy objects can be used in combination with local objects. This allows for general multi-database queries over several mediator peers. The result of such queries may be literals (as in the example), proxy objects, or local objects. The system stores internally information about the origin of each proxy object so it can be identified properly. Each local OID has a locally unique OID number and two proxy objects are considered equal if they represent objects created in the same mediator or source with equal OID numbers.

Proxy types can be used in function definitions as any other type. In the example one can define a derived function of the persons located in a certain location:

```
create function personnel_in(Charstring l) -> Personnel@Tb
       as select p from Personnel@Tb p
          where  location(p) = l;
```

In this case the local function `personnel_in` will return those instances of the proxy type for `Personnel` in mediator named `Tb` for which it holds that the value of function `location` in `Tb` returns 1. The function can be used in local queries and function definitions, and as proxy functions in multi-database queries from other mediator peers.

Multi-database queries and functions are compiled and optimized through a distributed query decomposition process fully described in [20] and summarized

later. Notice again that there is no central mediator schema and the compilation and execution of multi-database queries is made by exchanging data and meta-data with the accessed mediator servers. If some schema of a mediator server is modified, the multi-database functions accessing that mediator server become invalid and must be recompiled.

4.2 Data source modeling

Information about different data sources is represented explicitly in the Amos II data model through the system type `Datasource` and its subtypes (Fig. 2). Some subtypes of `Datasource` represent generic kinds of data sources that share common properties, such as the types `Relational` and `SearchEngine` [22] representing the common properties of all RDBMSs and all Internet search engines, respectively. Other subtypes of `Datasource` like `ODBC_DS` and `JDBC_DS` represent specific kinds of sources, such as ODBC and JDBC drivers. In particular the system type `Amos` represents other Amos II peers. Instances of these types represent individual data sources. All types under `Datasource` are collectively called the *datasource types*.

Since wrappers and their corresponding datasource types interact tightly, every wrapper module installs its corresponding types and functions whenever initialized. This reflexive design promotes code and data reuse and provides transparent management of information about data sources via the Amos II query language.

Each datasource type instance has a unique name and a set of imported types. Some of the (more specific) subtypes have defined a set of low-level access functions. For example the type `Relational` has the function `sql` that accepts any relational data source instance, a parameterized SQL query, and its parameters. Since there is no generic way to access all relational data sources this function only defines an interface. On the other hand the type `ODBC_DS` overloads this function with an implementation that can submit a parameterized query to an ODBC source. These functions can be used in low-level mediator queries which roughly corresponds to the *pass-through* mode defined in the SQL-MED standard [32]. However normally the low-level data access functions are not used directly by the users. Instead queries that refer to external sources are rewritten by the wrapper modules in terms of these functions. In addition datasource types may include other functions, such as source address, user names, and passwords.

4.3 Reconciliation

Proxy objects provide a general way to query and exchange data between mediators and sources. However, reconciliation requires types defined in terms of data in different mediators. For this, the basic system is extended with *derived types* (DTs), which are types defined in terms of queries defining their extents. These *extent queries* may access both local and proxy objects.

Data integration by DTs is performed by building a hierarchy of DTs based on local types and types imported from other data sources. The traditional inheritance mechanism, where the corresponding instances of an object in the

super/subtypes are identified by the same OID, is extended with declarative query specification of the correspondence between the instances of the derived super/subtypes. Integration by sub/supertyping is related to the mechanisms in some other systems as, e.g., the integrated views and column adding in the Pegasus system [9], but is better suited for use in an object-oriented environment.

The extents of derived subtypes are defined through queries restricting the intersection of the extents of the constituent supertypes. For example:

```
create derived type CSD_emp under Personnel p
       where location(p)=''CSD'';
```

This statement creates a derived type `CSD_emp` whose extent contains those persons who work in the CSD department. When a derived type is queried the system will implicitly create those of its instance OIDs necessary to execute the query.

An important purpose of derived types is to define types as views that reconcile differences between types in different mediator servers. For example, the type `Personnel` might be defined in mediator `Tb` while `Ta` has a corresponding type `Faculty`. The following statement executed in a third mediator, M, defines a derived type `Emp` in M representing those employees who work both in `Ta` and `Tb`.

```
create derived type Emp
       under Faculty@Ta f, Personnel@Tb p
       where ssn(f)=id_to_ssn(id(p))
```

Here the `where` clause identifies how to match equivalent proxy objects from both sources. The function `ssn` uniquely identifies faculty members in `Ta`, while the function `id` in `Tb` identifies personnel by employee numbers. A (foreign) function `id_to_ssn` in M translates employee numbers to SSNs.

The system internally maintains the information necessary to map between OIDs of a derived type and its supertypes.

An important issue in designing object views is the placement of the DTs in the type hierarchy. Mixing freely the DTs and ordinary types in a type hierarchy can lead to semantically inconsistent hierarchies [24]. In order to provide the user with powerful modeling capabilities along with a semantically consistent inheritance hierarchy, the ordinary and derived types in Amos II are placed in a single type hierarchy where it is not allowed to have an ordinary type as a subtype of a DT. This rule preserves the extent-subset semantics for all types in the hierarchy. If DTs were allowed to be supertypes of ordinary types, due to the declarative specification of the DTs, it would not have been possible to guarantee that each instance of the ordinary type has a corresponding instance in its supertypes [24].

The DT instances are derived from the instances of their supertypes according to an extent query specified in the DT definition. DT instances are assigned OIDs by the system, which allows their use in locally stored functions defined over the DTs in the same way as over the ordinary types. A selective OID generation for the DT instances is used to avoid performance and storage overhead.

The concept of derived types and its use for data integration is fully described in [18].

The regular DTs, defined by subtyping through queries of their supertypes, provide means for mediation based on operators such as join, selection, and projection. However, these do not suffice for integration of sources having overlapping data. When integrating data from different mediator servers it is often the case that the same entity appears either in one of the mediators or in both. For example, if one wants to combine employees from different departments, some employees will only work in one of the departments while others will work in both of them.

For this type of integration requirements the Amos II system features a special kind of DTs called *Integration Union Types* (IUTs) defined as supertypes of other types through queries. IUTs are used to model unions of real-world entities represented by overlapping type extents. Informally, while the regular DTs represent restrictions and intersections of extents of other types, the IUTs represent reconciled unions of (possibly overlapping) data in one or more mediator server or data sources. The example in Fig. 4 illustrates the features and the applications of the IUTs.

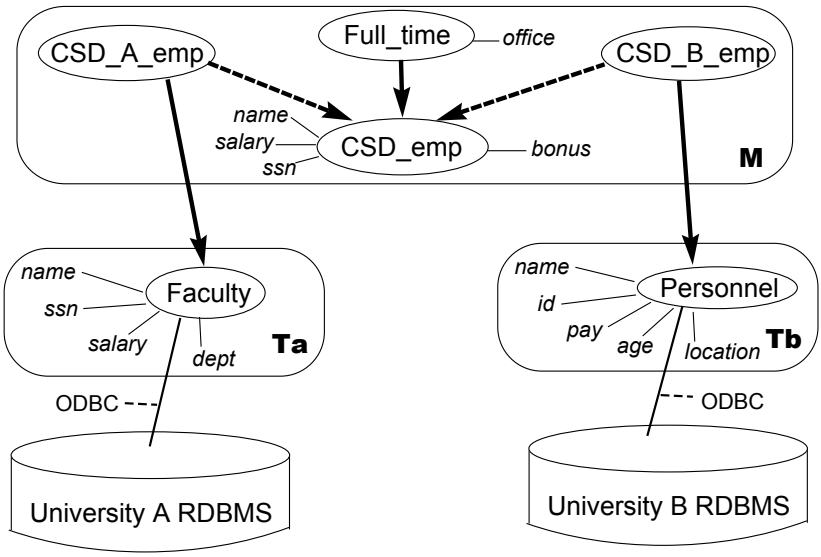


Figure 4: An Object-Oriented view for the computer science department

In this example, a computer science department (CSD) is formed out of the faculty members of two universities named *A* and *B*. The CSD administration needs to set up a database of the faculty members of the new department in terms of the databases of the two universities. The faculty members of CSD can be employed by either one of the universities. There are also faculty members employed by both universities. The full time members of a department are assigned an office in the department.

In Fig. 4 the mediators are represented by rectangles; the ovals in the rect-

angles represent types, and the solid lines represent inheritance relationships between the types. The two mediators `Ta` and `Tb` provide Amos II views of the relational databases *University A DB* and *University B DB*. In mediator `Ta` there is a type `Faculty` and in mediator `Tb` a type `Personnel`.

The relational databases are accessed through an ODBC wrapper in `Ta` and `Tb` that translates AmosQL queries into ODBC calls. The ODBC wrapper interface translates AmosQL queries over objects represented in relations into calls to a foreign function executing SQL statements [4]. The translation process is based on partitioning general queries into subqueries only using the capabilities of the data source, as fully explained in [20].

A third mediator M is setup in the CSD to provide the integrated view. Here, the semantically equivalent types `CSD_A_emp` and `CSD_B_emp` are defined as derived subtypes of types in `Ta` and `Tb`:

```
create derived type CSD_a_emp
  under Faculty@Ta f
    where dept(f) = 'CSD';

create derived type CSD_b_emp
   under Personnel@Tb p
    where location(p) = 'Building G';
```

The system imports the external types, looks up the functions defined over them in the originating mediators, and defines local proxy types and functions with the same signature but without local implementations.

The IUT `CSD_emp` represents all the employees of the CSD. It is defined over the *constituent subtypes* `CSD_a_emp` and `CSD_b_emp`. `CSD_emp` contains one instance for each employee object regardless of whether it appears in one of the constituent types or in both. There are two kinds of functions defined over `CSD_emp`. The functions on the left of the type oval in Fig. 4 are derived from the functions defined in the constituent types. The functions on the right are locally stored.

The data definition facilities of AmosQL include constructs for defining IUTs as described above. The integrated types are internally modeled by the system as subtypes of the IUT. Equality among the instances of the integrated types is established based on a set of key attributes. IUTs can also have locally stored attributes, and attributes reconciled from the integrated types. See [19] for details.

The type `CSD_emp` is defined as follows:

```
CREATE INTEGRATION TYPE CSD_emp
  KEYS ssn Integer;
  SUPERTYPE OF
    CSD_A_emp ae: ssn = ssn(ae);
    CSD_B_emp be: ssn = id_to_ssn(id(be));
  FUNCTIONS
    CASE ae
      name = name(ae);
      salary = pay(ae);
```

```
    CASE be
      name = name(be);
      salary = salary(be);
    CASE ae, be
      salary = pay(ae) + salary(be);
  PROPERTIES
      bonus Integer;
END;
```

For each of the constituent subtypes, a `KEYS` clause is specified. The instances of different constituent types having the same key values will map into a single IUT instance. The key expressions can contain calls to any function.

The `FUNCTIONS` clause defines the reconciled functions of `CSD_emp`, derived from functions over the constituent subtypes. For different subsets of the constituent subtypes, a reconciled function of an IUT can have different implementations specified by the `CASE` clauses. For example, the definition of `CSD_emp` specifies that the `salary` function is calculated as the salary of the faculty member at the university to which it belongs. In the case when s/he is employed by both universities, the salary is the sum of the two salaries. When the same function is defined for more than one case, the most specific case applies. Finally, the `PROPERTIES` clause defines the stored function `bonus` over the IUT `CSD_emp`.

The IUTs can be subtyped by derived types. In Fig. 4, the type `Full_Time` is defined as a subtype of the `CSD_emp` type, representing the instances for which the salary exceeds a certain number (50000). The locally stored function `office` stores information about the offices of the full time CSD employees. The type `Full_Time` and its property `office` have the following definitions:

```
create derived type Full_Time under CSD_emp e
                               where salary(e)>50000;
create function office(Full_Time)->Charstring
       as stored;
```

5 Query Processing

The description of type hierarchies and semantic heterogeneity using declarative multi-database functions is very powerful. However, a naive implementation of the framework could be very inefficient, and there are many opportunities for the extensive query optimization needed for distributed mediation.

The query processor of Amos II, illustrated by Fig. 5, consists of three main components. The core component of the query processor is the *local query compiler* that optimizes queries accessing local data in a mediator. The *multi-database query compiler, MQC* allows Amos II mediators to process queries that also access other mediator peers and data sources. Both compilers generate query execution plans (*QEPs*) in terms of an *object algebra* that is interpreted by the *QEP interpreter* component. The following two sections describe in more detail the sub-components of the local and the multi-database query compilers.

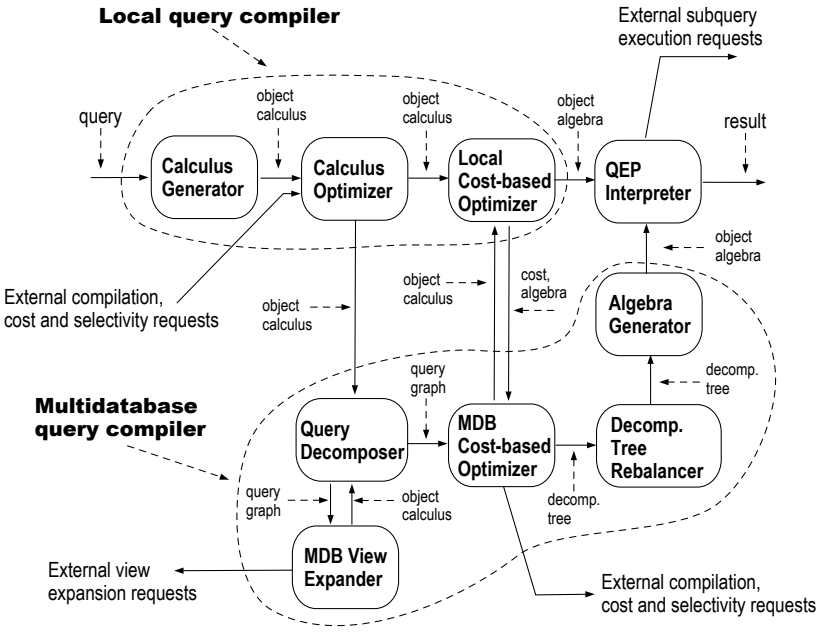


Figure 5: Query processing in Amos II

5.1 Local query processing

To illustrate the query compilation of single-site queries we use the sample ad hoc query:

```
select p, name(parent(p))
  from person p
  where hobby(p) = 'sailing';
```

The first query compilation step, *calculus generation*, translates the parsed AmosQL query tree into an *object calculus* representation called ObjectLog [29]. The object calculus is a declarative representation of the original query and is an extension of Datalog with objects, types, overloading, and multi-directional foreign functions.

The calculus generator translates the example query into this expression:

$$\{\, p, nm \mid$$
$$p = Person_{nil \rightarrow Person}() \wedge$$
$$pa = parent_{Person \rightarrow Person}(p) \wedge$$
$$nm = name_{Person \rightarrow Charstring}(pa) \wedge$$
$$'sailing' = hobby_{Person \rightarrow Charstring}(p)\}$$

The first predicate in the expression is inserted by the system to assert the type of the variable p. This *type check predicate* defines that the variable p is bound to one of the objects returned by the *extent function* for type `Person`, $Person()$,

which returns all the instances (the extent) of its type. The variables nm and pa are generated by the system. Notice that the functions in the predicates are annotated with their type signatures, to allow for overloading of function symbols over the argument types.

The *calculus optimizer* of the query optimizer first transforms the unoptimized calculus expression to reduce the number of predicates, e.g. by exploring properties of type definitions. In the example, it removes the type check predicate:

$$\{\ p, nm\ \mid \\ pa = parent_{Person \rightarrow Person}(p)\ \wedge \\ nm = name_{Person \rightarrow Charstring}(pa)\ \wedge \\ 'sailing' = hobby_{Person \rightarrow Charstring}(p)\}$$

This transformation is correct because p is used in a stored function (`parent` or `hobby`) with argument or result of type `Person`. The referential integrity system constrains instances of stored functions to be of correct types [29].

The *local cost-based optimizer* will use cost-based optimization to produce an executable object algebra plan from the transformed query calculus expression. The system has a built-in cost model for local data and built-in algebra operators. Basically the cost-based optimizer generates a number of execution plans, applies the cost model on each of them, and chooses the cheapest for execution. The system has the options of using dynamic programming, hill climbing, or random search to find an execution plan with minimal cost. Users can instruct the system to choose a particular strategy.

The optimizer is furthermore extensible whereby new algebra operators are defined using the multi-directional foreign functions, which also provide the basic mechanisms for interactions between mediator peers in distributed execution plans.

The *query execution plan interpreter* will finally interpret the execution plan to yield the result of the query.

5.2 Queries over derived types

Queries over DTs are expanded by system-inserted predicates performing the DT system support tasks [18]. These tasks are divided into three mechanisms: (i) providing consistency of queries over DTs so that the extent-subset semantics is followed; (ii) generation of OIDs for those DT instances needed to execute the query; and (iii) validation of the DT instances with assigned OIDs so that DT instances satisfy the constraints of the DT definitions. The system generates derived function definitions to perform these tasks. During the calculus optimization the query is analyzed and, where needed, the appropriate functions definitions are added to the query. A selective OID generation mechanism avoids overhead by generating OIDs only for those derived objects that are either needed during the execution of a query, or have associated local data in the mediator database.

The functions specifying the view support tasks often have overlapping parts. [18] demonstrates how calculus-based query optimization can be used to remove redundant computations introduced from the overlap among the

system-inserted expressions, and between the system-inserted and user-specified parts of the query.

Each IUT is mapped by the calculus optimizer to a hierarchy of system generated DTs, called *auxiliary types* [19]. The auxiliary types represent disjoint parts of the outerjoin needed for this type of data integration. The reconciliation of the attributes of the integrated types is modeled by a set of overloaded derived functions generated by the system from the specification in the IUT definition. Several novel query processing and optimization techniques are developed for efficiently processing the queries containing overloaded functions over the auxiliary types, as described in [19].

5.3 Multi-database query processing

The *Multi-database Query Compiler (MQC)* [20, 17] is invoked whenever a query is posed over data from more than one mediator peer. The goal of the MQC is to explore the space of possible distributed execution plans and choose a 'reasonably' cheap one. As the local query compiler, the MQC uses a combination of heuristic and dynamic programming strategies to produce a set of distributed object algebra plans.

The distributed nature of Amos II mediators requires a query processing framework that allows cooperation of a number of autonomous mediator peers. The MQC interacts with the local optimizer as well as with the query optimizers of the other mediator peers involved in the query via requests to estimate costs and selectivities of subqueries, requests to expand the view definitions of remote views, and requests to compile subqueries in remote mediator peers. The generated local execution plan interacts with the execution plans produced by the other mediator peers.

The details of the MQC are described in [20]. Here we will overview its main sub-components.

- The *query decomposer* identifies fragments of a multi-database query, *subqueries*, where each subquery can be processed by a single data source. The decomposer takes as input an object calculus query and produces a *query graph* with nodes representing subqueries assigned to an execution site and arcs representing variables connecting the subqueries. The benefit of decomposition is twofold. First, complex computations in subqueries can be pushed to the data sources to avoid expensive communication and to utilize the processing capabilities of the sources. Second, the multi-database query optimization cost is reduced by the partitioning of the input query into several smaller subqueries.

  Query decomposition is performed in two steps:

  1. *Predicate grouping* collects predicates executable at only one data source and groups them together into one or more subqueries. The grouping process uses a heuristic where cross-products are avoided by placing predicates without common variables in separate subqueries.

2. *Site assignment* uses a cost-based heuristics to place those predicates that can be executed at more than one site (e.g. θ-joins), eventually replicates some of the predicates in the subqueries to improve the selectivity of subqueries, and finally assigns execution sites to the subqueries.

- The *multi-database view expander* expands remote views directly or indirectly referenced in queries. This may lead to significant improvement in the query plan quality because there may be many redundancies in large compositions of multi-database views.

  The multi-database view expander traverses the query graph to send expansion requests for the subqueries. In this way, all predicates defined in the same database are expanded in a single request. This approach allows the remote site to perform calculus simplifications of the expanded and merged predicate definitions as a whole and then return the transformed subquery. However, when there are many mediator layers it is not always beneficial to fully expand all view definitions, as shown in [21]. The multi-database view expander therefore uses a heuristic to choose the most promising views for expansion, a technique called *controlled view expansion*. After all subqueries in the query graph have been view expanded the query decomposer is called again for predicate regrouping.

- The *multi-database (MDB) query optimizer* decides on the order of execution of the predicates in the query graph nodes, and on the direction of the data shipping between the peers. Execution plans for distributed queries in Amos II are represented by *decomposition trees*. Each node in a decomposition tree describes a join cycle through a client mediator (i.e. the mediator where the query is issued). In a cycle, first intermediate results are shipped to the site where they are used. Then a subquery is executed at that site using the shipped data as input, and the result is shipped back to the mediator. Finally, one or more post-processing subqueries are performed at the client mediator. The result of a cycle is always materialized in the mediator. A sequence of cycles can represent any execution plan. As the space of all execution plans is exponential to the number of subqueries in the input query graph, we examine only the space of left-deep decomposition trees using a dynamic programming approach. To evaluate the costs and selectivities of the subqueries the multi-database optimizer sends compilation requests for the subqueries both to the local optimizer and the query compilers of the remote mediators.

- The *decomposition tree rebalancer* transforms the initial left-deep decomposition tree into a bushy one. To avoid that all the data flows through the client mediator, the decomposition tree rebalancer uses a heuristic that selects pairs of adjacent nodes in the decomposition tree, merges the selected nodes into one new node, and sends the merged node to the two mediators corresponding to the original nodes for recompilation. From the merged nodes, each of the two mediators generate different decomposition sub-trees and the cheaper one is chosen. In this way, the input

decomposition tree is rebalanced from a left-deep tree into a bushy one. The overall execution plan resulting from the tree rebalancing can contain plans where the data is shipped directly from one remote mediator to another, eliminating the bottleneck of shipping all data through a single mediator. See [17] for details.

- The *object algebra generator* translates a decomposition tree into a set of inter-calling local object algebra plans.

6 Related Work

Amos II is related to research in the areas of data integration, object views, distributed databases, and general query processing. There has been several projects on intergration of data in a multi-database environment [5, 8, 10, 12, 14, 16, 23, 27, 30, 41, 42]. The integration facilities of Amos II are based on work in the area of OO views [1, 3, 15, 26, 33, 36, 37, 40].

Most of the mediator frameworks reported in the literature (e.g. [16, 42, 14]) propose centralized query compilation and execution coordination. In [9] it is indicated that a distributed mediation framework is a promising research direction, but to the best of our knowledge no results in this area are reported. Some recent commercial data integration products, as IBM's Federated DB2, also provide centralized mediation features.

In the DIOM project [30] a framework for integration of relational data sources is presented where the operations can be executed either in the mediator or in a data source. The compilation process in DIOM is centrally performed, and there is no clear distinction between the data sources and the mediators in the optimization framework.

The Multiview [36] object-oriented view system provides multiple inheritance and a capacity-augmented view mechanism implemented with a technique called Object Slicing [26] using OID coercion in an inheritance hierarchy. However, it assumes active view maintenance and does not elaborate on the consequences of using this technique for integration of data in autonomous and dislocated repositories. Furthermore, it is not implemented using declarative functions for the description of the view functionality.

One of the few research reports describing the use of functional view mechanisms for data integration is the Multibase system [8]. It is also based on a derivative of the DAPLEX data model and does reconciliation similar to the IUTs in this paper. An important difference between Multibase and Amos II is that the data model used in Multibase does not contain the object-oriented concept of OIDs and inheritance. The query optimization and meta-modeling methods in Amos II are also more elaborate than in Multibase.

The UNISQL [23] system also provides views for database integration. The virtual classes (corresponding to the DTs) are organized in a separate class hierarchy. However, the virtual class instances inherit the OIDs from the corresponding instances in the ordinary classes, which prohibits definition of stored functions over virtual classes defined by multiple inheritance as in Amos II. There is no integration mechanism corresponding to the IUTs.

[35] gives a good overview of distributed databases and query processing. As opposed to the distributed databases, where there is a centralized repository containing meta-data about the whole system, the architecture described in this paper consists of autonomous systems, each storing only locally relevant meta-data.

One of the most thorough attempts to tackle the query optimization problem in distributed databases was done within the System R* project [7] where, unlike Amos II, an exhaustive, cost-based, and centrally performed query optimization is made to find the optimal plan. Another classic distributed database system is SDD-1 [2] which used a hill-climbing heuristics as the query decomposer in Amos II.

7 Summary

We have given an overview of the Amos II mediator system where groups of distributed mediator peers are used to integrate data from different sources. Each mediator in a group has DBMS facilities for query compilation and exchange of data and meta-data with other mediator peers. Derived functions can be defined where data from several mediator peers are abstracted, transformed, and reconciled. Wrappers are defined by interfacing Amos II systems with external systems through its multi-directional foreign function interface. Amos II can furthermore be embedded in applications and used as stand-alone databases. The paper gave an overview of Amos II's architecture with references to other published papers on the system for details.

We described the functional data model and query language forming the basis for data integration in Amos II. The distributed multi-mediator query decomposition strategies used were summarized.

The mediator peers are autonomous without any central schema. A special mediator, the name server, keeps track of what mediator peers are members of a groups. The name servers can be queried for the location of mediator peers in a group. Meta-queries to each mediator peer can be posed to investigate the structure of its schema.

Some unique features of Amos II are:

- A distributed mediator architecture where query plans are distributed over several communicating mediator peers.
- Using declarative functional queries to model reconciled functional views spanning over multiple mediator peers.
- Query processing and optimization techniques for queries to reconciled views involving function overloading, late binding, and type aware query rewrites.

The Amos II system is fully implemented and can be downloaded from *http://user.it.uu.se/∼udbl/amos*. Amos II runs under Windows and Unix.

ACKNOWLEDGEMENTS:

The following persons have contributed to the development of the Amos II kernel: Gustav Fahl, Staffan Flodin, Jörn Gebhardt, Martin Hansson, Vanja Josifovski, Jonas Karlsson, Timour Katchaounov, Milena Koparanova, Salah-Eddine Machani, Joakim Näs, Kjell Orsborn, Tore Risch, Martin Sköld, and Magnus Werner.

References

[1] S. Abiteboul, A. Bonner: Objects and Views. *ACM Intl. Conf. on Management of Data (SIGMOD'91)*, 238-247, 1991.

[2] P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie Jr.: Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4), 602-625, 1981

[3] E. Bertino: A View Mechanism for Object-Oriented Databases. *3rd Intl. Conf. on Extending Database Technology (EDBT'92)*, 136-151, 1992.

[4] S. Brandani: Multi-database Access from Amos II using ODBC. *Linköping Electronic Press*, 3(19), Dec., 1998, *http://www.ep.liu.se/ea/cis/1998/019/*.

[5] O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*. Prentice Hall, 1996.

[6] K.Cassel and T.Risch: An Object-Oriented Multi-Mediator Browser. *2nd International Workshop on User Interfaces to Data Intensive Systems*, Zrich, Switzerland, May 31 - June 1, 2001

[7] D. Daniels, P. Selinger, L. Haas, B. Lindsay, C.Mohan, A.Walker, P.F.Wilms: An Introduction to Distributed Query Compilation in R*. *2nd International Symposium on Distributed Data Bases*, 291-309, 1982.

[8] U. Dayal, H-Y. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, 10(6), 628-645, 1984.

[9] W. Du, M. Shan: Query Processing in Pegasus, In O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*. Prentice Hall, 449-471, 1996.

[10] C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Multidatabase Query Optimization. *Distributed and Parallel Databases*, Kluwer, 5(1), 77-114, 1997.

[11] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, Springer, 6(4), 261-281, 1997.

[12] D. Fang, S. Ghandeharizadeh, D. McLeod, A. Si: The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database System. *9th Intl. Conf. on Data Engineering Conf. (ICDE'93)*, IEEE, 467-475, 1993.

[13] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions. *21st Conf. on Very Large Databases (VLDB'95)*, 335-344, 1995

[14] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y.Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Intelligent Information Systems (JIIS)*, Kluwer, 8(2), 117-132, 1997

[15] S. Heiler, S. Zdonik: Object views: Extending the Vision. *6th Intl. Conf. on Data Engineering (ICDE'90)*, IEEE, 86-93, 1990

[16] V.Josifovski, P.Schwarz, L.Haas, and E.Lin: Garlic: A New Flavor of Federated Query Processing for DB2, *ACM SIGMOD Conf.*, 2002.

[17] V.Josifovski, T.Katchaounov, T.Risch: Optimizing Queries in Distributed and Composable Mediators. *4th Conference on Cooperative Information Systems*, CoopIS'99, 291-302, 1999.

[18] V.Josifovski, T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration. *Intelligent Information Systems (JIIS)* 12(2-3), Kluwer, 165-190, 1999.

[19] V.Josifovski, T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. *25th Conf. on Very Large Databases (VLDB'99)*, 435-446, 1999.

[20] V.Josifovski and T.Risch: Query Decomposition for a Distributed Object-Oriented Mediator System. *Distributed and Parallel Databases J.*, 11(3), pp 307-336, Kluwer, May 2002.

[21] T.Katchanouov, V.Josifovski, T.Risch: Distributed View Expansion in Object-Oriented Mediators, *5th Intl. Conference on Cooperative Information Systems, CoopIS'00*, Eilat, Israel, LNCS 1901, Springer Verlag, 2000.

[22] T. Katchaounov, T. Risch, and S. Zrcher: Object-Oriented Mediator Queries to Internet Search Engines, *International Workshop on Efficient Web-based Information Systems (EWIS)*, Montpellier, France, September 2nd, 2002.

[23] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: Schema Architecture of the UNISQL/M Multidatabase System. In W. Kim (ed.): *Modern Database Systems - The Object Model, Interoperability, and Beyond*, ACM Press, 621-648, 1995.

[24] W. Kim and W. Kelley: On View Support in Object-Oriented Database Systems, In *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press/ Addison-Wesley Publishing Company, New York, NY, 1995.

[25] M.Koparanova and T.Risch: Completing CAD Data Queries for Visualization, *International Database Engineering and Applications Symposium (IDEAS 2002)*, Edmonton, Alberta, Canada, July 17-19, 2002.

[26] H. Kuno, Y. Ra, E. Rundensteiner: *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation.* Univ. of Michigan Tech. Report CSE-TR-241-95, 1995.

[27] E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, John Wiley & Sons, 25(5), 533-562, 1995.

[28] H.Lin, T.Risch, and T.Katchaounov: Adaptive data mediation over XML data. In special issue on 'Web Information Systems Applications' of *Journal of Applied System Studies (JASS)*, Cambridge International Science Publishing, 3(2), 2002.

[29] W. Litwin, T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 517-528, 1992

[30] L.Liu, C.Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed and Parallel Databases*, Kluwer, 5(2), 167-205, 1997.

[31] P. Lyngbaek: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[32] J.Melton, J.Michels, V.Josifovski, K.Kulkarni, P.Schwarz, and K.Zeidenstein: SQL and Management of External Data, *SIGMOD Record*, Vol. 30, No. 1, 70-77, March 2001.

[33] A. Motro: Superviews: Virtual Integration of Multiple Databases. *IEEE Transaction on Software Engineering*, Vol. 13(7), 785-798, 1987.

[34] K. Orsborn: Applying Next Generation Object-Oriented DBMS to Finite Element Analysis. In W. Litwin, T. Risch (eds.): *Intl. Conf. on Applications of Databases (ADB'94)*, Springer, 215-233, 1994.

[35] M.T.Özsu, P.Valduriez: *Principles of Distributed Database Systems*, Prentice Hall, 1999.

[36] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones and P. Marron: The MultiView project: object-oriented view technology and applications, *ACM Intl. Conf. on Management of Data (SIGMOD'96)*, 555, 1996.

[37] M. Scholl, C. Laasch, M. Tresch: Updatable Views in Object-Oriented Databases. *2nd Deductive and Object-Oriented Databases Conference (DOOD91)*, 189-207, 1991.

[38] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), 140-173, 1981.

[39] M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th Intl. Conf. on Data Engineering (ICDE'96)*, IEEE, 392-401, 1996.

[40] C. Souza dos Santos, S. Abiteboul, C. Delobel: Virtual Schemas and Bases. *Intl. Conf. on Extending Database Technology (EDBT'92)*, 81-94, 1994.

[41] S. Subramananian, S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient Views. *ACM Intl. Conf. on Management of Data (SIGMOD'98)*, 319-330, 1998

[42] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Date Engineering*, 10(5), 808-823, 1998

[43] G Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3), 38-49, 1992.

Paper B:

Timour Katchaounov and Tore Risch. Interface capabilities for query processing in peer mediator systems. *Technical report 2003-048*, Department of Information Technology, Uppsala University, 2003.

Interface Capabilities for Query Processing in Peer Mediator Systems

Timour Katchaounov and Tore Risch
Uppsala University
firstname.lastname@it.uu.se

Abstract

A peer mediator system (PMS) is a decentralized mediator system based on the P2P paradigm, where mediators integrate data sources and other mediators through views defined in a multi-mediator query language. In a PMS mediator peers compose views in terms of views in other peers - mediators and sources, or directly pose queries in the multi-mediator query language to some peer. All peers are fully autonomous and there is no central catalog or controller. Each peer in a PMS must provide an interface to its data and meta-data sufficient to allow the cooperative processing of queries by the PMS. We analyze the computational capabilities and meta-data that a software system has to export in order to participate as a peer in a PMS. For the analysis we identify and compare six classes of peer interfaces with increasing complexity. For each class we investigate the performance and scalability implications that result from the available capabilities and required meta-data. Our results are two-fold: *i)* we provide guidelines for the design of mediator peers that can make best use of the interfaces provided by the data sources, and *ii)* we analyze the tradeoffs in the design of inter-mediator interfaces so that mediator peers can efficiently cooperate to process queries against other composed mediators. Finally we describe the choices made in a concrete implementation of a PMS.

1 Introduction

Global computer networks, and the Internet in particular, provide the technical means to interconnect large numbers of distributed software systems owned and maintained by many independent persons and organizations. This capability to interconnect many such systems presents many new opportunities for information sharing and reuse. Most of these distributed systems are deployed and maintained independently of each other in ways that suit best the local needs of their users. This results in fully autonomous systems that are heterogeneous at many levels starting from the use of different platforms and languages to heterogeneity at the logical level in the ways real-world concepts are modeled.

The area of data integration is concerned with the problem of integration of heterogeneous data managed and/or produced by many heterogeneous and autonomous systems, called *data sources* to emphasize the data access/management aspect of software systems. One approach to integrate many such data sources is *mediation* [53, 54] where data from many sources is accessed and combined "on-the-fly" in a coherent

view through a network of mediator components, each specialized in some knowledge domain. Most existing mediator systems, such as [20, 16], reduce the general mediation concept to centralized architectures that consist of one mediator that integrates many data sources into a coherent view. Such data integration solutions are suitable for enterprise contexts, due to more or less centralized organizational control and data ownership. However, as recognized by recent research on data integration [18, 22, 41, 7], many new application areas exist where centralized coordination is not feasible, such as scientific cooperation, distributed engineering, and dynamic company alliances. These areas of human activity may benefit enormously from the ability to combine information shared by others to produce new information enriched with their local knowledge.

Peer mediators.

For the integration of many autonomous data sources we use a decentralized data integration system architecture based on the initial mediation paradigm [53] where many domain-specialized mediators share their data source abstractions so that other mediators can reuse and combine these abstractions for higher levels of applications and/or mediators. To allow each of the mediators to be maintained by independent experts or groups of experts, we design the mediators as database systems with a high-level query language that allows to define *logical compositions* of peers by defining views in the mediator peers in terms of views in other mediator and data source peers. The mediator owners integrate data sources and other mediators through the definition of views in terms of the mediator query language, and export some of these integrated views to other mediators and applications. The mediators and the sources are organized in a distributed system that follows the peer-to-peer (P2P) paradigm, called *Peer Mediator System (PMS)* [25, 31, 46, 30]. In a PMS there is no central catalog or coordinator of any kind, and the mediators are fully autonomous peers that cooperate together to process queries to their integrated views. A PMS consists of three types of peers divided into three layers - data sources, mediators and applications. Data sources manage stored and/or computed collections of data and provide programmatic interfaces for the access to their data. Mediators access relevant data from one or more data sources, transform the data into a common representation, and match and integrate the transformed data so that mediator users are presented with logically coherent views of the data sources. Applications accept information requests from the users, send these requests to the mediator layer and deal with the presentation of mediator replies to the users. Applications always access the data sources through one or more mediators, thus any PMS consists of at least one mediator and any number of applications and data sources. In order to compute answers to user queries, the mediators have their own query processors that translate logical peer compositions into *physical compositions* that consist of interacting implementations of query operators organized into query execution plans (QEPs).

The advantages of a P2P approach for mediation are that it allows the domain experts to own and control independently their mediators in the same way as data source owners have total control over the data sources. Each mediator may evolve at its own pace as long as it preserves its public interface. Furthermore, in a PMS for better scalability the integration effort can be distributed among many domain experts and reused through logical compositions of mediators. Finally, a P2P architecture

promotes reuse of computation resources such as storage, CPU cycles, and specialized software and hardware.

Problem description.

Among the most important issues in the design of any distributed system are the program interfaces between the components of the system, the underlying computational capabilities exposed through these interfaces, and the information that needs to be exchanged between the system's components, so that all components can cooperate to execute a common task. Since the main purpose of a PMS is to integrate many data sources and to fulfill information requests through a declarative query language, the common task to be executed by all peers in a PMS is that of processing queries. Thus, an important problem in the design of a PMS is the relationship between the interfaces of the peers to their data and meta-data, the corresponding computational capabilities available through these interfaces, and the meta-data that the peers require, so that the mediator peers can efficiently process queries against many logically composed peers.

Peer interfaces In a PMS there are three types of program interfaces - application to mediator, mediator to mediator (inter-mediator), and mediator to data source. In general it is possible to envision applications that will need to combine data from many mediators. However such applications will duplicate much of the mediator functionality. To simplify such applications, they may always access a network of mediators through a designated "gateway" mediator. Thus we can consider that applications use the same inter-mediator interface as the mediators. Since mediator peers can serve as powerful data sources for other mediators, we treat mediator and data source peers uniformly and we indicate the specific type of peers only when necessary. This uniform treatment of mediators simplifies and generalizes our discussion and allows conclusions to be drawn both about a range of design alternatives for the capabilities and meta-data for the inter-mediator interfaces together with conclusions about the mediator to source interfaces and required meta-data.

Interface capabilities Peer interfaces can be considered at two levels of interoperability. At a *syntactic* level interfaces must match in the data types that can be exchanged between peers and the agreed protocol. This level of interoperability is addressed by a wide variety of standards and protocols, here called *physical interfaces*, starting from message-based protocols implemented directly on top of some transport protocol as HTTP, to higher abstraction remote procedure call based protocols such as RPC [8], CORBA [11], JavaRMI, SOAP [4], and even to some degree database access protocols as ODBC and JDBC. All these access methods provide different degrees of abstraction and performance. However, most of these interoperability standards provide none or very limited means to describe the semantics of the functionality that is invoked through them, e.g. even ODBC varies in terms of the SQL dialects accepted by data sources. In addition, most of these access methods can be simulated on top of the others, e.g. CORBA calls can be used to send complex XML documents as asynchronous messages, while HTTP can be used to transport SOAP messages that describe remote procedure calls. Thus the same computational capability implemented by a remote peer may be accessed through a variety of physical interfaces independent of the capability itself.

A higher level of abstraction above physical interfaces are the computational capabilities of the data sources. By capabilities we mean the abstract computations that a source can perform over some optional input data. The computational capabilities of a peer available through its low-level interfaces are called the *interface capabilities* of the source. For example the typical interface capabilities of a relational DBMS include the ability to compile SQL queries and store them for future execution[1] and the ability to directly execute SQL statements or precompiled queries. In this paper we abstract ourselves from the low-level interfaces, the choice of which has been studied elsewhere [29, 35, 45]. Our goal is to consider what are the publicly available computation capabilities, that is interface capabilities, of the peers in a PMS.

In the literature the terms "limited source capabilities" or "limited capabilities" are usually understood in two ways based on modeling sources as collections of relations. The first, used in e.g. [56, 14], which we name *limited access capabilities*, refers to the the fact that many sources, such as Web forms and specialized computation sources, require that some of the attributes of a source relation must be provided in order for a source relation to be accessible, thus the source's relations have limited access paths compared to corresponding relational tables. In relational terminology this means that source relations do not provide a scan interface. The second view on limited source capabilities, used in e.g. [52] and here called *limited query capabilities*, is more general and considers capabilities as the set of queries that a source can compute with respect to some query language. Our concept of interface capabilities also includes arbitrary computations that a peer may perform, e.g the ability to precompile a query and return a query handle, as long as they are accessible by other peers.

Interface capabilities, meta-data, and query processing. Due to the complete decentralization of PMSs, the peers have very little information about each other, unlike distributed DBMSs [44] where there is a centralized catalog. This requires that at query processing time the peers exchange information needed to compile and evaluate queries that involve many peers. This can be information about the schema, data statistics, resources, availability, capabilities, etc. of the peers. We use the term *meta-data* to denote all kinds of data used to describe the properties of the actual data that is of interest to the end users and the properties of the system that is used to manage/access that data. In particular a *schema* is a special kind of meta-data that describes the structure and relationships of data. Schema information is used both by the user to specify queries and the system to efficiently access the requested data. *Technical meta-data* describes technical aspects of data and data sources which are usually not of interest to the end-user, such as data distribution, access cost, etc. Technical meta-data is used in query processing, ideally without user involvement.

The more meta-data is available to the peers, the more sophisticated query processing can be performed that results in better QEPs either because more alternative plans can be considered, or because of more precise data statistics.

On the other hand one may expect that the more meta-data is exchanged between the peers, the higher the overhead both in the exchange and the utilization of the meta-data. For example, a cost-based query optimizer may send very large numbers of cost requests to other peers to evaluate the time or resources to compute some operation at the peers. At the same time a query optimizer "aware" that certain operations can be

[1] This corresponds to the "prepare" functionality in ODBC/JDBC.

computed at more than one peer, may need to explore much larger space of alternative plans to find an optimal assignment of operations to peers. In large compositions of peers this may lead to prohibitively high query compilation costs.

The exchange of one kind of meta-data may trigger exchange of other kinds of meta-data, e.g. view expansion may require the exchange of additional schema information such as new data types and the schema of new relations; once new remote relations become visible to a query compiler, their access costs have to be retrieved/estimated.

Therefore the design of an efficient and effective PMS has to take into account the tradeoffs between several inter-dependent factors: *i)* the computation capabilities of the peers available through their interfaces, *ii)* the meta-data the peers depend on, and *iii)* the query processing techniques that can be applied in the presence of particular meta-data and interface capabilities. In addition, the more meta-data, the more advanced interfaces are used by the peers and the more advanced query processing techniques are applied that utilize this meta-data and interfaces, the more complex can be the implementation of the mediator peers.

Contributions

This paper investigates qualitatively the relationship between the interface capabilities, meta-data, and query processing in a PMS and their implications on the performance and complexity of a PMS implementation.

To make our discussion more concrete, we base it on a specific functional and object-oriented mediator data model and query language, described in Sect. 2, that we have implemented in the AMOS II mediator system [46]. We compare six classes of peer interface capabilities with increasing degree of complexity. In each of them more meta-data is exchanged between the peers. This allows the mediator peers to perform more sophisticated query processing and to explore more alternative QEPs. Our results are based on our experiences with the implementation of some of these interface classes and the corresponding query processing techniques in the AMOS II mediator system.

For each class we investigate the performance and scalability implications that result from the available capabilities and required meta-data. Our results are:

- for each interface class we identify the minimum requirements for the capabilities and the meta-data that a software system has to make available through its program interfaces to other systems in order to participate as a peer in a PMS and allow the processing of inter-peer queries,

- we analyze the tradeoffs in the design of inter-mediator interfaces so that mediator peers can efficiently cooperate to process queries against other logically composed mediators,

- we provide some guidelines how to design mediator peers that can make best use of the interfaces provided by the data sources,

- finally we indicate how these interface classes have been used for query processing in the AMOS II mediator system [46].

Organization.

The following Sect. 2 introduces the functional mediator data model and query language, and related concepts that we use in the rest of this paper for our analysis. In section 3 we analyze the six interface classes, each in a separate sub-section. We describe our implementation of the interface classes in the AMOS II peer mediator system in Sect. 4. Section 6 provides an overview of related work. Finally we summarize and discuss our findings in Sect. 6.

2 Functional Peer Mediators - data model, query language and terminology

Many kinds of data sources have an object-oriented (OO) data model, while others have relational or semi-structured data models. Therefore we believe that object-orientation is necessary for the integration of many types of data sources with complex data typically found in, e.g. the scientific and engineering areas. The data sources may or may not have OO features; however, the mediators in a PMS should have a data model sufficiently expressive to model all types of data sources.

Below we describe the functional and object-oriented mediator data model and related concepts that we will use throughout the rest of this paper. A detailed description of the mediator data model and query language can be found in [46].

Basic concepts. The basic modeling concept in our functional mediator data model is the *object*. Objects are classified in *types*. Attributes of objects and relationships between types of objects are expressed through *functions*. While objects model real-world entities, in general functions represent computations. Functions consist of two parts - a *type signature* and an *implementation*. The type signature defines the types of the arguments and the results of a function, and a logical direction from the function inputs to its outputs. For example the function :

```
grade(student, course) -> integer;
```

that models the relationship between courses, students and their grades in a university database, would have the signature $student, course \rightarrow integer$. The *extent* of a function is the set of all tuples that describe the mapping of all inputs to corresponding outputs.

Kinds of functions. Depending on how a function implementation computes its result(s) we distinguish several kinds of functions. *Stored* functions store explicitly the result of a computation, *derived* functions specify the result of a computation as a declarative query defined in terms of other functions and types, and *foreign* functions represent computations specified in an external language(s) and/or module(s).

Connection to the SQL data model. We relate functions to concepts in the SQL:1999 object-relational data model (as the most popular one) in the following way. Stored functions correspond to stored relations (tables), derived functions correspond to views, and foreign functions correspond to user-defined functions (UDFs). Thus, functions

are a unifying concept for stored tables, views and various kinds of UDFs (scalar and table functions, stored procedures) in the relational data model. The uniformity of the concept of functions greatly simplifies the discussion of many aspects of query processing that neither depend on the implementation of functions (whether they are stored, derived or foreign), nor they depend on the kind of functions (whether they are single or bag-valued, etc.). This allows us to relate our discussion to different data models and query languages.

Binding patterns and multi-directional functions. In order to model arbitrary n-ary relationships computable in different directions, functions can be *multi-directional*, that is, given values for some of the function input and/or output variables, the mediator can compute the corresponding values for the remaining variables. Function variables for which values are provided are said to be *bound*, all other variables are *free*, thus the *computation direction* of a function is determined by its bound and the free variables. The computation direction where the bound variables coincide with the logical inputs of the function is called the *forward* direction, all other combinations of bound and free variables are *inverses*. Functions computable only in one direction are said to be *single-directional*.

Stored functions are computable in all possible directions, however foreign functions that model arbitrary computations are in general computable only in some directions. To model n-ary relationships that are computable only in some directions, multi-directional functions are annotated with binding patterns. A *binding pattern B* is a mapping from the set of variables V of a function definition into the set of symbols $\{b, f\}$, where b denotes a bound variable, and f a free variable. Binding patterns can be expressed positionally as sequences of $\{b, f\}$, $\langle x_1, \ldots, x_n \rangle$, where each x_i corresponds to the variable v_i at position i in a function definition.

In general different directions of a relationship have to be computed in different ways. For example the foreign function

```
power(number base, number exp) -> number root;
```

can be thought of as an abstract 3-way relation between numbers. This relation is computable only in four of all possible six directions. These four directions are described positionally by the binding patterns: bbf, fbb, bfb, and bbb. Each of the first three binding patterns requires a different algorithm, correspondingly implemented by functions that compute the power, the root, and the logarithm of a two numbers. Thus, each binding pattern is also associated with a concrete implementation in some procedural language. To allow a cost-based query optimizer to pick the best foreign function implementation, each binding pattern also has a *cost function* associated with it that computes the execution cost and selectivity of the particular implementation. In our *power* funcion example, if the function is invoked with the *bbb* binding pattern, all three implementations can be used to check if three numbers are related by a power relationship. However we may expect that one of the three algorithms is more efficient than the others.

Multi-directional functions tie together implementations of algorithms for traversing relationships in different directions. The concept of a multi-directional function is similar to that of a relation adorned with binding patterns as in [51]. Multi-directional functions and cost-based optimization with multi-directional functions are described in more detail in [37].

Global queries. A database query is a declarative specification of the result of some computation. A view is a named query. Queries are expressed through a SQL-like *select* statements. For example the query:

```
select name(s)
from student s
where birthyear(s) > 1980;
```

finds the names of all students born after 1980. Queries that refer to data and/or meta-data in other peers are called *global queries*.

When a query is posed to a mediator peer, we call that mediator the *query peer*. All other peers that contain database objects referenced by a query are called *remote* peers. Both terms are relative to the query and the role of the peer. In order for a mediator peer to express *global* queries and views that involve data from other peers, external meta-data needs to be represented somehow. The term *proxy* denotes a reference to any remote database object stored in another peer. In accordance with the basic functional data model presented above, mediator peers model the contents of other peers in terms of *proxy objects*, *proxy functions*, and *proxy types*. Similar to local functions, proxy functions model both data collections in remote peers, computations in remote peers, and attributes of remote data items. Proxy types correspond to the types (or classes) of the remote data items. Proxy objects correspond to individual data items in remote peers.

3 Classes of Peer Mediator System Interface Capabilities

This section describes and analyses six classes of PMS interface capabilities in terms of the functional data model presented in Sect. 2. We present each of the interface classes in order of increasing amount of meta-data needed for query processing, and increasing space of QEPs that needs to be considered for each class. Each new interface class allows for strictly larger plan spaces to be explored, in which there may exist better QEPs compared to the preceding classes of interfaces.

The ordering we provide is not definite - some of the interface classes are independent of each other and can be combined to form more complex interface classes. For such complex interface classes the combined effect of the properties of each constituent must be considered, which we leave outside of the scope of this discussion.

Our discussion assumes that there is one mediator peer, the *query peer*, to which queries are posed and the queries reference one or more data sources and/or mediator peers. The mediator peers may further integrate other logically composed peers. In our discussion of query optimization for each interface class for brevity we assume that the mediators use cost-based query optimization, however most of our conclusions are independent of the particular optimization method, e.g. whether it is exhaustive or not, or whether it is static (performed before query execution) or dynamic (performed at query execution time). In Sect. 6 we discuss the importance of dynamic approaches to query optimization [23, 17] for a PMS and some of the concequences of dynamic query optimization for each of the interface classes.

3.1 Single-Directional Proxy Functions

The simplest possible way to interface peers in a PMS so that mediators can compute queries that refer to data in other peers is to make proxy functions directly computable. That is, to associate each proxy function with an implementation that computes the results of the function by invoking the corresponding computation at a remote peer. With this approach the interface capabilities of each remote peer are determined by its proxy functions, and the results of these functions directly represent the contents of the peers. For example a university may provide a Web site with five HTML forms that allow to list all students, all courses, all students taking a course, all courses of a student, and the grade of a student for a course. This Web site can be represented by five proxy functions[2]:

```
all_courses() -> bag of course;
all_students() -> bag of student;
enrolled_in(course) -> bag of <student, grade>;
signed_for(student) -> bag of <course, grade>;
grade(student, course) -> grade;
```

Figure 1: University data source

3.1.1 Functionality

An implementation of a proxy function has to: *i)* translate the function parameters into a format understandable to the remote peer, *ii)* package the translated parameters together with additional meta-data that specifies the corresponding remote computation into a remote procedure call descriptor, and ship this call descriptor to a remote peer, *iii)* invoke the corresponding computation at the remote peer, *iv)* ship back the computation result(s), and *v)* translate the results into the local data representation and return them to the query processor of the mediator peer. We term this functionality as *call shipping*.

In order to compute a proxy function through call shipping, all logical inputs of the proxy function must be bound prior to execution. The result of call shipping is in general a bag of tuples described by the logical outputs of the function. Since call shipping computes proxy functions only in their logical direction, we call such functions *single-directional proxy functions (SDPF)*. Notice that with call shipping the logical direction of an SDPF coincides with its physical direction, thus the type signature of an SDPF specifies the variable bindings as well. An SDPF F is fully specified by a 4-tuple $\langle T, I, P, C \rangle$, where T is its type signature, I is an implementation of F that realizes a call shipping functionality, P is the peer that contains the remote data collection modeled by F, and C is an identifier for the data collection in the remote peer.

Since the result of a remote call can be generally multi-valued (a multi-set of tuples), it can be either fully materialized at the remote peer and shipped back as a whole, partially materialized and shipped in bulks of tuples, or fully streamed and

[2] We use *bagof* to model collections with duplicates (multisets).

shipped one tuple at a time to the query peer. The feasibility of these strategies depends on the capabilities of the source peer. For example many Web sources return their results as separate pages and provide a navigational interface to retrieve the next set of results. Such pages with results correspond naturally to bulks of tuples. Simpler Web sources may fully materialize request results as large documents. Similarly, sources such as ODBC provide a tuple-at-a-time iterator interface to request results.

3.1.2 Related approaches

We notice that at this level of description, the execution of proxy functions is conceptually the same as that of fine grain remote procedure calls, as implemented by e.g., Sun RPC [1], CORBA, and SOAP. By "fine grain" we mean the type of access where remote procedures access and return individual data items or individual data sets. For example if CORBA is used to implement peer interoperability, then each proxy function corresponds to a CORBA operation, each proxy type to a CORBA interface, and each proxy object to a CORBA object.

3.1.3 Querying

Since user queries over many sources with varying capabilities are defined in terms of proxy functions, an important question is what is the degree of abstraction provided to the user by the SDPF class of peer interface capabilities?

Many data source peers, especially Web sources, may provide many ways to query their contents that require or produce overlapping data. As a result there may be more than one way to retrieve the same information, because there may be several ways to combine proxy functions into queries that are equivalent. For example, assume a mediator user wants to specify the following query to the university Web source described above: "What are the students who have grade '5' in all courses?". Three variants of the query, specified below in the mediator query language on Fig. 2, are: *Q1)* select all courses in the university and all students with their grades enrolled in the courses, and filter the students with grade '5', *Q2)* select all students in the university and all courses they are signed for with corresponding grades, and filter the students with grade '5', and *Q3)* select all courses and all students in the university, get their grade through the *grade* relationship, and check if that grade is '5'.

The queries express the same information request. However, depending on the number of students, number of courses and distribution of students per course, one of the queries may have orders of magnitude better performance that the others. Given that realistic data sources may have large number of data collections with many ways to access their data, SDPF interfaces require that the user is familiar with the data statistics of the source data in order to write efficient queries. An additional problem is that users may not always be able to discover the right proxy functions to answer their information need even if they are familiar with the schema of the remote peers.

3.1.4 Query processing

With the SDPF class of interfaces users have to choose themselves the proxy functions that directly correspond to specific ways of accessing the data at the remote peers. This not only complicates the task of the mediator users, but also prevents the

```
Query Q1:
select s
from student s, course c, grade g
where c = all_courses() and
      <s,g> = enrolled_in(c) and
      g = 5;

Query Q2:
select s
from student s, course c, grade g
where s = all_students() and
      <c,g> = signed_for(s) and
      g = 5;

Query Q3:
select s
from student s, course c, grade g
where c = all_courses() and
      s = all_students() and
      g = grade(s,c) and
      g = 5;
```

Figure 2: Three ways to express the same query with SDPFs.

mediator query optimizer from choosing the best way to retrieve relevant information. With the SDPF class of interface capabilities query processing costs are distributed as follows. All query compilation is performed at the query mediator peer because all other peers are only capable of computing the results of proxy functions for given arguments. Functions with arguments can be looked at as selections and functions without arguments as scans. Since these are all operations that remote peers perform, all other database operations, such as join and union, must be performed by the query peer. The only functionality the remote peers have from the perspective of a mediator query peer is that of selections and scans; therefore query execution is fully controlled by the query peer. Since all communication between the peers is performed via remote call requests, all data from all remote peers always flows through the query peer. Thus if the result of some proxy function is needed as input to another one that is computed at the same remote peer, all data will nevertheless flow through the query peer in a centralized manner.

To summarize, all query processing except for the computation of the proxy functions is performed by the query peers in a centralized manner with respect to each mediator peer. When many levels of peers are composed in terms of each other through views, query peers can "see" only their immediate neighbors. If these neighbors are mediators themselves that integrate other peers through views, requests sent to these mediator peers trigger similar centralized execution, which is "invisible" to the query mediator that submits a request. As a result all query execution follows exactly the logical composition of the mediator peers.

3.1.5 Meta-Data

A PMS where peers inter-operate through SDPF interfaces requires two kinds of meta-data. Schema information is necessary that maps the structure of the data items in the source peers to types in the mediator data model. The proxy functions that model collections and attributes of data items in remote peers are defined over the mediator proxy types that represent the corresponding remote data items, *student*, *course*, and *grade*. This schema information is used by mediator users to specify queries and views over data in other peers, by the implementations of the proxy functions to convert data between the mediator and the remote peer data models, and by the query processor to perform semantic analysis of queries. To allow the mediator to perform cost-based optimization of queries over many peers, the minimal technical meta-data required is the cost and selectivity of each proxy function. In addition, other schema information may be supplied, such as uniqueness constraints for some data item attributes and extent semantics for proxy functions that are known to compute the complete extension of remote data collections.

Since many data sources provide very little or no meta-data at all, the mediator data definition language (DDL) should provide the means for the mediator programmer to specify all this meta-data when defining the mapping from proxy types and functions to the corresponding concepts at the remote peers.

Other data source peers, such as RDBMS and the mediator peers themselves, provide programmatic access to their meta-data. The access to such meta-data can be implemented based on the reflective nature of the functional data model by using meta-data proxy functions that access the meta-data interfaces of remote peers and return meta-objects of type *Function* and *Type*. Thus meta-data proxy functions can implement automated mapping between data models of remote peers and the local data model, so that a schema of any remote peer can be automatically imported as part of the local schema of a mediator peer.

3.1.6 Discussion

The advantages of the SDPF approach are its simplicity and non-intrusiveness. Proxy functions can be added to a relational query processor extensible with user-defined functions (UDFs) because they can be directly implemented as UDFs. The distribution of the peers is fully encapsulated in the implementation of the UDFs, thus a query optimizer does not need to deal with distribution. Typically UDFs provide some way for cost and selectivity information to be associated with them, which allows also to hide the cost of of distribution. The SDPF interface class also requires very little meta-data to be exchanged between the peers and does not violate peer autonomy because the implementation of remote functions is hidden for the other peers.

The problems with the SDPF approach are in that *i)* users must have knowledge of data statistics at the sources to produce efficient queries, *ii)* query specification is hard due to potentially large numbers of proxy functions and their combinations, *iii)* query processing is centralized which requires most processing to be performed by the query mediator peers, *iv)* all computations at remote peers are treated as black boxes modeled by proxy functions, and, as a result, *v)* the execution of global queries follows the path of the logical composition of the peers which may result in many redundant computations because the same computations in the same peers may be redundantly

invoked via many different logical paths.

In summary, this is a simple, but potentially very inefficient approach to add MDB capabilities to a single-site optimizer.

3.2 Multi-Directional Proxy Functions

One of the main disadvantages of the SDPF approach is that it does not provide truly declarative means for the users to specify queries and views that integrate many peers. Users need to know which of several alternative ways to access data is the most efficient one. The main reason for this deficiency is that when proxy functions are implemented with the SDPF approach, they are executable only in one direction because they are directly mapped to some procedures at the remote peer. As a result several proxy functions may represent different ways to access various subsets of the same data collection, as in Fig. 2.

To alleviate this problem, we notice that all single-directional proxy functions that access the same data collection can be viewed as alternative ways to compute some generic function that logically describes the contents of a remote data collection. In the context of our university Web site example, we notice that the proxy functions *enrolled_in*, *signed_for*, and *grade* (from Fig. 1) in fact relate the same information about students - the courses they take and their grades, that is they compute the same relationship in several directions corresondingly described by the binding patterns fbf, bff, and bbf. Thus we can represent the three SDPFs through one abstract proxy function

```
student_info(student, course) -> grade;
```

that relates together all three SDPFs as representing the same relationship, and let the query processor of the mediator choose which SDPF results in best query performance depending on the query where the abstract function is used, and the statistics for each SDPF. As a result remote peers are modeled through sets of such abstract proxy functions that can be computed in multiple directions, and therefore called *multi-directional proxy functions (MDPF).*

3.2.1 Functionality and query processing

Unlike single-directional proxy functions, multi-directional proxy functions are not always directly computable. Instead, they only describe abstract relationships between objects of different types at remote peers. An MDPF can be computed cooperatively by a query peer and the peer where it is defined in two ways that depend on the capabilities of the source peers.

First, a query peer can substitute at query compile time an MDPF with a concrete implementation that invokes some computation at a remote peer. For this, an implementation of multi-directional proxy functions must provide a way to relate an MDPF with the actual implementations that compute the MDPF in particular directions.

This relationship can be established by defining an MDPF M as a tuple $\langle T, \{F_j, B_j\}\rangle$ where T is a type signature, and $\{F_j, B_j\}$ is a set of SDPFs F_j, and corresponding binding patterns B_j. Each binding pattern B_j defines a permutation of the variables in M, such that the logical inputs of F_j correspond to the bound variables in M. This permutation of variables is required because the SDPFs are computable only in the forward

direction. Thus, an implementation of an MDPF must be able to reorder its inputs to fit the variable order of the invoked SDPF and the results of the SDPF to fit the signature of its MDPF. With this approach, the query compiler reduces the problem of computing an MDPF to the computation of some SDPF. Because of this reduction, this approach is applicable to all kinds of peers accessible through SDPFs.

The second possibility to compute MDPFs is to delegate the choice of the most efficient computation of an MDPF to the source peer during query execution time. This requires that the source peer is capable of accepting an MDPF reference together with the bound parameters and deciding on the fly which is the best implementation to access the data collection modeled by the MDPF. The main problem with this approach is that the query compiler at the query peer can not easily estimate the execution cost and selectivity of MDPF remote calls at compile time, because variable bindings can not be known before execution. It remains to be investigated how to process queries with such dynamic MDPFs.

3.2.2 Related approaches

The ODL object-oriented data definition language, which is a part of the ODMG standard [10], provides facilities to define inverse relationships as special kind of class properties. Unlike our functional approach, relationships in ODL can be only binary. The main limitation of ODL relationships is that it is not possible for the users to specify user-defined methods that compute the same relationship in different directions. Instead, the system maintains the relationships transparently for the user. This is convenient for top-down database design, but limiting when specifying interfaces to sources with specialized and/or limited capabilities.

3.2.3 Querying

Queries over peers modeled as collections of MDPFs can be expressed in a more simple and declarative way because many related interface capabilities at the peers are “hidden” behind one abstract proxy function. For example, all three queries from Fig. 2 can be expressed through the single query on Fig. 3.

```
select s
from student s, course c, grade g
where c = all_courses() and
      student_info(s,c) = g
      g = 5;
```

Figure 3: Example of query with MDPF interface class.

3.2.4 Meta-Data

Compared to the SDPF interface class, MDPFs require additional meta-data that relates several interface capabilities at a remote peer as computing the same relationship between data items in different directions. Many data sources do not provide such meta-data, therefore users must be able to manually define MDPFs in the mediator

data definition language, and relate the MDPFs to their corresponding implementations and binding patterns based on the user's knowledge of the sources. For some data sources, such as RDBMS, it may be known in advance that specific kinds of data collections, e.g. tables and views, can be always accessed in all possible directions. Such sources allow MDPFs to be automatically generated and related to their corresponding implementations for different binding patterns.

Since an MDPF relates data items of the same types no matter in what direction it is computed, all schema related meta-data can be encoded in the MDPF itself. However, to allow a query processor to choose the most efficient implementation of an MDPF, the MDPF interface class requires the same technical meta-data as in the SDPF interface class.

3.2.5 Discussion

The MDPF approach has three major advantages - first, it reduces the total number of functions that a user must deal with which simplifies the task of specifying queries and views, second, the user does not have to consider alternative queries for performance reasons, and third, the query processor automatically selects the most efficient implementations of generic proxy functions. Thus the MDPF interface class provides higher level of abstraction while at the same time it provides improved performance compared to the SDPF interface class. Practical implementations of MDPF interfaces require that a mediator query processor can optimize queries in the presence of limited binding patterns and thus are more complex than that of peers based on the SDPF interface class. Query processing with limited binding patterns is studied, e.g., in [37] and later in [14]. Finally, since MDPF is reduced to SDPF, MDPF interface carry over all other problems characteristic of SDPF.

3.3 Multi-peer Proxy Functions

In a peer mediator system peers may have overlapping capabilities and therefore there might be a choice where to perform some of the computations in a QEP and consequently where to ship the data necessary for those computations. This choice may lead to considerable differences in query performance in particular when the processing power and network link capacity varies between the peer nodes. Using our approach to model remote peers as collections of proxy functions, *overlapping peer capabilities* mean that two or more proxy functions represent equivalent computations implemented at different peers. Two functions are *equivalent* if for the same input they always produce the same output and this holds for all possible legal inputs. Thus a function can be freely substituted for any of its equivalent functions. Function equivalence does not necessarily mean that two remote computations are implemented in the same way. If a proxy function represents data that is explicitly stored at a peer, then equivalent proxy functions represent data replicas. In addition equivalent proxy functions may represent other equivalent computations, such as an image similarity matching operator available at several peers. Thus, data replication becomes a special case of equivalent proxy functions. This means that a query processor that takes into account function equivalence will automatically take into account data replication.

Since proxy function equivalence stems from the equivalence of the corresponding computations at remote peers, in the general case the concept of equivalence has to be

applied to the participating SDPFs. Equivalence of MDPFs can be defined in different more or less "strict" ways. Here we consider two MDPFs as equivalent if they are computable in the same directions and for each direction the corresponding SDPFs are equivalent. This definition can be relaxed in various ways which are outside the scope of this work.

3.3.1 Functionality

As we already noticed, one SDPF may be computed through several equivalent computations that are implemented differently in different peers. For example, let us add to our university scenario one more source peer that provides access to its data through an ODBC physical interface, and is known to contain a replica of the function *enrolled_in*. In this case the implementations of the two functions that access the same data in different peers will be different - one will submit HTTP requests, the other - ODBC calls. Therefore we can consider that for each set of equivalent SDPFs that correspond to concrete computations in some peers, there is some abstract function with a binding pattern common for all the SDPFs. We call such abstract functions *multi-peer proxy functions (MPPF)*. An MPPF is fully described by a tuple $\langle T, \{F_j\}\rangle$ where T is a type signature, and F_j is either an SDPF or an MDPF.

3.3.2 Related approaches

The idea of equivalent proxy functions relates to the concept of *location transparency* in distributed DBMS (DDBMS) [44], federated DBMS (FDBMS) [48], and mediator systems [53]. In a DDBMS, for performance and reliability, (partially) replicated relations are distributed among many nodes. A DDBMS provides full location transparency by choosing automatically which replicas to use for querying, thus all replicated relations at the query language level are similar to MPPFs. There are two main differences between replication in DDBMSs and MPPFs. First, table replicas in a DBBMSs are specified in a top-down fashion by the user and are then automatically maintained by the system, and the users cannot "tell" the DBMS that some external data collections are in fact replicas. In contrast to that, MPPFs allow mediator users to specify the equivalence of arbitrary remote data collections that are outside of the control of the mediator. This can be done on per remote data collection basis either manually by the mediator users based on their knowledge or automatically by the mediator system whenever equivalence can be discovered automatically. The second main difference is that, unlike replication in DDBMSs that relates only stored data, MPPFs provide a uniform way to treat as equivalent both computations and stored data in external sources. Both this differences make MPPFs more suitable for data integration than the automatic top-down approach to replication of DDBMSs.

A federated DBMS integrates multiple export schemas into a federated schema that includes information about the distribution of the export schemas and the mapping between different schema elements of the export schemas and the corresponding element in the federated schema. At the level of a federated schema data distribution is invisible for the user in the same way as through MPPFs. However, the design of a federated schema also requires that semantic heterogeneity of the export schemas is resolved, thus federated schemas are more general than MPPFs with respect to the heterogeneity of the component databases.

The concept of local-as-view (LAV) schema mappings in mediator systems [36, 50] is similar in that it provides a single logical view over many distributed data collections as MPPFs. However, LAV schema mappings relate not only equivalent data collections but also ones that represent similar real-world concepts. LAV mappings are more general than MPPFs and can deal also with semantic data heterogeneity and specify how to restructure and reconcile related data collections into one coherent view. MPPFs only account for data collection equivalence and can not handle heterogeneity, however query processing with LAV mappings [36] is much more complex that that with MPPFs.

3.3.3 Query processing

To enable processing of queries over MPPFs, it is necessary that a mediator query processor takes into account the equivalence of remote functionality, and that it explores alternative plans where the computation of equivalent functions is performed at various peers. Thus a mediator query compiler needs to explore a larger search space where not only the order of the operations matter, but also the peers where operations are executed. It is shown in [34] that for a System R style optimizer the time complexity of such distributed query optimization is $O(s^3 * 3^n)$, where s is the number of peers and n the number of relations, while in a single-site optimizer it is $O(3^n)$ as shown in [42]. These results apply only to the case when there are no binding patterns to consider. The combination of MPPFs with MDPFs may result in even bigger search spaces for a query optimizer, because it has to consider both alternative peers for the computation of functions and alternative binding patterns. As a result we may expect higher compilation cost that increases with the number of peers referenced in a query.

One special type of optimization that a mediator peer should perform is when it implements locally some function referenced in a global query, that is some of the MPPFs have local SDPFs in their definitions. In such cases it is always possible to replace an MPPF with a corresponding local function in its definition to avoid communication.

In all other cases, even when several functions in a QEP can be executed at the same remote peer, all the data passed between these functions still has to be shipped through the query peer in a centralized manner. This is due to that MPPFs in a QEP are replaced by SDPFs which allow only centralized query execution, i.e. the peers cannot exchange data directly.

3.3.4 Querying

MPPFs provide an even higher level of abstraction above MDPFs because they “hide” many functions that perform the same computation behind one abstract function. This reduces even further the number of functions a user must deal with to specify queries. For example, if several peers provide the same operation, e.g. an image comparison operation, if the equivalence of these operations is not known to the mediator system, then it is the user who must choose one of these alternative implementations. In a similar way as MDPFs provide an advantage over SDPFs, MPPFs are more general than SDPFs and can be combined with MDPFs to provide a more expressive tool to describe peer sources. Once the equivalence of functions across peers is established

through MPPFs, users need not be aware of distribution and may concentrate on query semantics, letting the mediator query compiler select the most appropriate peers.

The second advantage of MPPFs is that queries specified in terms of MPPFs need not be changed when peers are added and removed, or their capabilities are modified. The only changes needed when the functionality of a peer changes or the peer is added/removed is that of the definitions of the affected MPPFs. Then the mediator query compiler can automatically recompile all affected queries without any user intervention.

3.3.5 Meta-Data

In addition to the meta-data needed by the SDPF and MDPF interface classes, the definition of MPPFs needs meta-data about the equivalence of functions and operations defined in different peers. Function equivalence can be established in several ways. Users can explicitly specify function equivalence via the mediator DDL. Knowledge about standards, query languages and particular systems can be used, e.g. equivalence of built-in functions in DBMSs can be easily established. Automatic and semi-automatic means can be used to detect function equivalence as in [55], e.g. functions with the same name and signature can be considered to have the same semantics. For functions implemented in a declarative language as parameterized queries the notion of query equivalence [24] can be used to detect function equivalence. However, in the general case it is not possible to infer automatically function equivalence. Therefore it is essential that some peers provide the means for the user to specify and store function equivalence information. Ideally this information is stored as part of the meta-schema of the peer(s). MPPFs inherit all other meta-data requirements of SDPFs/MDPFs.

3.3.6 Discussion

Modeling remote peers through MPPFs can considerably reduce query execution times due to lower network overhead, better utilization of processing power and utilization of more efficient data access paths at the remote peers. Once the equivalence of some functionality in a PMS is established, the users may transparently specify queries using functions as generic concepts without the need to take into account where these functions are implemented and what are the query performance consequences. MPPFs also provide better scalability in the data integration process because they allow mediators to automatically adjust to changes through query recompilation.

The cost of these advantages is more complex query processing that requires a distribution-aware query processor. The use of MPPFs expands the search space of a query compiler which allows more efficient QEPs to be found than in the SDPF and MDPF approaches. However this may result in considerably higher compilation costs compared to the use of only SDPFs/MDPFs. As with MDPFs, multi-peer proxy functions inherit all other disadvantages of SDPFs because they are directly replaced by SDPFs at compile time. Finally, with MPPFs mediators can consider only capabilities that are already present at the peers.

3.4 Plan Shipping

The analysis of the previous three classes of inter-peer interfaces, collectively called *proxy function interfaces*, shows that in order to achieve better performance, it is necessary that the peers provide more powerful ways to access their data than through simple RPCs modeled as directly computable proxy functions. The disadvantages of the proxy function interfaces are rooted in that from the perspective of the query peers, all other peers are capable only of computing proxy functions that correspond to individual data collections or object properties. Thus, query processing based on proxy function interfaces has to be performed centrally, and data is always shipped to the computation in the form of call parameters and results. Such centralized query processing can be suboptimal because *i)* even if several computations in a QEP are performed at the same peer, all intermediate data is transferred across the network through the query peer, and *ii)* intermediate results between operations in different remote peers are always communicated through the query peer.

To overcome both limitations it is necessary to decentralize the processing of global queries in a PMS. Then, computations that can be executed at the same peer can be grouped together and sent to that peer for execution, so that network transmission of intermediate results is replaced by in-process communication. Operation grouping may also result in reduction of network traffic because the combined execution of several operations may produce less data than the execution of each operation separately. Intermediate results between operations that do not involve the query peer can be communicated directly between the remote peers much more efficiently than through the query peer, depending on the network links between the peers.

To decentralize the processing of queries in a PMS, the query peers should be able to delegate to other peers the computation of portions of queries, here called *query fragments*, as whole units of processing. This requirement means that source peers must have the capability to "understand" and compute such query fragments. Query fragments, as queries in general, can be represented in two ways - either as algebraic expressions that describe how to compute a query fragment, or as declarative expressions that logically specify the desired result. In this section we will consider the first case when query fragments are communicated as algebraic expressions. The following Sect. 3.5 discusses the second alternative - that of communicating query fragments between peers in the form of sub-queries.

In order for a query peer to instruct other peers to perform complex computations via algebraic expressions, the remote peers must be able to store and execute such expressions on peer's request. We term this interface capability as *plan shipping*.

3.4.1 Functionality

Plan shipping is useful for systems that provide an algebraic interface to their data, but have no query language or query compiler. Since few systems provide public interfaces that allow them to accept QEPs (and thus directly expose their query processors), and there is no standard for query plans, plan shipping is most applicable to peers of the same kind that "know" each other's QEP format, and thus are able to generate and exchange such sub-plans.

To implement plan shipping it is required that, as minimum, query peers can *PSq1)* decompose global plans into sub-plans, each executable by some remote peer, and

PSq2) invoke remote plans via some mechanism. Source peers should be able to *PSs1)* receive a QEP in some form, and *PSs2)* compute and return the result of an invocation of a QEP. Having only these two capabilities means that a query peer has to submit sub-plans every time it executes an global query. To avoid this overhead, source peers should be able to *PSs3) install* sub-plans, that is, to locally store received plans, return handles for such plans, and invoke plans through such handles.

To enable cost-based query compilation the query peers should be able to *PSq3)* estimate the cost of executing sub-plans in another peer, the selectivity of these sub-plans, and the costs of network communication. Query peers can achieve this either by running probing queries, or using historical information about query execution. In addition source peers may be capable of *PSs4)* exporting statistics information about their data.

3.4.2 Related approaches

Plan shipping is used in some DDBMS such as distributed INGRES [12] where one "master" site decomposes an initial query into sub-queries and sends these subqueries to its "slave" sites for execution. Some mediator systems with distributed architecture use plan shipping as well to communicate QEPs to their components. The DISCO system [49] sends algebraic expressions in terms of a logical algebra to their wrappers. The mediators and the wrappers implement the same universal abstract machine. Wrappers evaluate sub-queries in this abstract algebra. The MOCHA mediator system [47] consists of one query processing coordinator (QPC) that performs query optimization and controls query execution of client queries, and of a number of data access providers (DAPs) that provide the QPC with a uniform access mechanism to remote data sources. Similar to DISCO wrappers, DAPs contain a query execution engine that accepts and executes query plans.

3.4.3 Query Processing

In order to utilize the plan shipping capabilities of the remote peers in a PMS, a mediator query processor must be able to *i)* decompose queries into sub-plans each computable at some peer, and an assembly plan computable at the mediator that composes the sub-plans, *ii)* optimize these sub-plans and the mediator assembly plan, *iii)* eventually translate the sub-plans into a representation understandable by the remote peers, and *iv)* execute the sub-plans, collect their results and compute with them the assembly plan to produce the final query result.

Due to the similarity of plan shipping and query shipping (described in Sect. 3.5), we defer the discussion of query processing for plan shipping to Sect. 3.5.3 where we point out the differences between query processing for both interface classes and discuss their advantages and disadvantages.

3.4.4 Meta-Data

Plan shipping assumes that the remote peers are only capable of executing a QEP in some form, but neither of generating nor refining a QEP. Therefore the query peers must be able to generate and ship only correct sub-plans to be processed by other peers. For that a query peer has to "know" about the query capabilities of other peers.

By query capabilities here we mean not only the functions and operators that can be computed by a peer, but also how these functions and operators can be combined to form complex expressions. As with all other classes of peer interfaces, when a peer has incomplete knowledge of the query capabilities of other peers, it can compensate with its own capabilities and perform by itself the computations it can not send to other peers. Therefore peers must have some way of acquiring, representing and using knowledge about other peer's query capabilities.

How can a peer acquire this knowledge? Humans can describe the query capabilities supported by various kinds of peers and store this meta-data at some peer(s). Peers may export meta-data about their and other peers' capabilities, and then query peers can automatically retrieve capability related information. It is very likely that only incomplete or no knowledge about peer query capabilities is available. In such cases peers may learn about each other's capabilities by trial-and-error as in [20]. For this, it is necessary that a peer can reply that it received an illegal plan. Then other peers may send various QEPs to probe whether they are executable. Finally, a combined approach allows to manually describe query capabilities, then other peers can retrieve this meta-data, and when it is incomplete allow peers to infer query capabilities via probing.

How to represent and use query capability information? Ideally capabilities should be modeled explicitly in terms of the data model of the mediator peers. This allows for high-level declarative access to capabilities meta-data, where all available query processing techniques can be used in a reflective manner to retrieve and manipulate this data. Another alternative is to embed the knowledge about peer capabilities in translator modules that "know" how to generate a QEP executable at a remote peer from a query in terms of the mediator query language. In this way all knowledge about peer capabilities is implicit in the code of the translator.

3.4.5 Discussion

Compared to the proxy function interfaces, plan shipping can be expected to provide considerably better performance due to reduced network costs and load distribution at query execution time. When a whole sub-plan for a query is executed at another peer, network costs are reduced because all the data between the operators is exchanged inside the same peer and therefore network communication is replaced by several order of magnitude faster intra- or inter- process communication (depending on the peer implementation). Plan shipping also reduces the number of RPC calls made across the network and replaces them with intra- or inter- process calls. Considering that RPC calls are orders of magnitude slower than function call in the same process and that in a query typically there are millions of such calls, we may expect considerable benefits. Finally, plan shipping can reduce the amount of data transferred over the network by combining data-reducing operations in the same query fragment.

Peers that have the capability *PSs3)* to store sub-plans for future execution raise the problem of how to keep the remote sub-plans consistent with the corresponding plans in the query mediators. There are many kinds of peers such as Web sources, Internet search engines and RDBMSs, to name a few, that do not export interfaces to their internal plan representation and therefore plan shipping is not applicable to them. Another problem with plan shipping is that it requires from the query peer to have very detailed knowledge about the capabilities of other peers and execution costs of each

function and operation. Plan shipping also puts all the query compilation load on the query peer. In a system with many peers this compilation cost may be substantial. Finally, plan shipping requires that peers give up their execution autonomy to the query peers. Many of the disadvantages with plan shipping are alleviated by the more general query shipping to be discussed next.

3.5 Query Shipping

Many important kinds of data sources, such as database systems, provide access to their data through high-level declarative interfaces via some query language. Such sources typically do not allow external systems to submit a pre-compiled QEP in a directly executable form. Therefore the plan shipping approach can not be applied to this type of data source peers. An alternative way to delegate the execution of query fragments to remote peers is to send the query fragments in a declarative form as *sub-queries* in terms of a query language supported by the corresponding peers. Peers with the capability to accept queries through some interface are said to have *query shipping* interface capabilities.

3.5.1 Functionality

For peers to inter-operate via query shipping, the query peers must be able to *QSq1)* identify the query fragment(s) of a query that can be computed by remote peers, and *QSq2)* submit data requests in the form of sub-queries expressed in the query language of a source peer.

Compared to all other interface classes discussed so far, query shipping demands the most advanced capabilities from the source peers. Such peers should be able to *QSs1)* accept sub-queries in terms of some query language and locally compile those sub-queries into sub-plans, and *QSs2)* execute sub-plans and return their results to their query peers. Similarly to plan shipping, source peers that allow only *QSs1)* and *QSs2)*, may require that the query peers ship the same sub-queries many times during the execution of a query, and these sub-queries are compiled and executed at the source peers on the fly. This approach is simple, but can be very expensive. Therefore, the source peers should provide a way to *QSs3)* precompile a sub-query, store the QEP for the sub-query, and return a handle for that QEP, and *QSs4)* allow remote systems to invoke a QEP through a handle.

Sub-queries are normally computed as parts of larger queries, therefore the same sub-query may be invoked many times with different values for the constants in the sub-query. One example is when a remote sub-query produces the data for the inner collection of a join. Then the sub-query is invoked for each value of the outer collection. A naive approach to execute remote sub-queries is to generate and send one sub-query per each set of input constants. This can lead to a large number of queries being sent and compiled at the source peers. Therefore, for better performance, source peers should be able to *QSs5)* compile and execute parameterized queries in a way similar to the *prepare* facility in ODBC and JDBC.

3.5.2 Related approaches

Query shipping is widely used for query processing in DDBMS [44]; multidatabase systems, e.g., [39, 43]; and mediator systems, e.g., [20, 16, 38], to name a few. Large number of projects have considered query processing issues related to query shipping both for distributed DBMS and centralized mediators. A PMS adds additional complexity to the problem because of the autonomy and decentralization of the peers. As a result, schema and data statistics are not readily available as in DDBMS, the peers are not homogeneous in their capabilities and interfaces. Thus query processing in a PMS must take into account the additional cost of acquiring data statistics and schema information about other peers, and compensate for missing capabilities at the remote peers. In the following sub-section we overview the main query processing steps typical for query processing with query and plan shipping. Many aspects of these techniques have been addressed in large number of works, many of which are overviewed in [44] and in [33].

3.5.3 Query Processing

Similar to plan shipping, query shipping requires that mediator peers identify the query fragments computable by other peers and request the execution of these fragments in some way. Thus, to utilize the query shipping capabilities of the peers, query processing in the mediator peers requires the same steps as plan shipping - query decomposition, fragment translation, fragmented query optimization and fragmented query execution. Below we describe each of these steps.

- **Query decomposition.** The most important functionality required to process queries against peers with both query and plan shipping interfaces is *query decomposition* - the capability of the query peers to split a query into fragments that can be computed by the source peers. Query decomposition takes a global query, identifies the remote peers referenced in the query, and splits the query into query fragments computable at the source peers. The peers in a PMS system are heterogeneous in terms of their capabilities, therefore query decomposition has to take into account the varying capabilities of the peers and utilize these capabilities. At the same time query decomposition must leave to the query peer the computation of functions and operations that cannot be processed by any other source peer. The result of query decomposition is an equivalent representation of a query, called a *fragmented query*, where the query fragments can be treated as atomic units of processing composed in an *assembly query* computable at the mediator that composes the intermediate results from the query fragments into the final query result. The difference between query and plan shipping is that with query shipping the query fragments are in a declarative form, here called *sub-queries*, while with plan shipping the query fragments have to have a directly executable algebraic representation as *sub-plans.*

- **Fragmented query optimization.** There may be many alternative ways with different performance to decompose a query into query fragments, and to compose these fragments into an assembly plan. Therefore, a mediator query processor must be able to find an optimal decomposition of a query into fragments, and an optimal execution order of the fragments in an assembly plan. The tradi-

tional database approach [44, 33] is to use cost-based query optimization to find optimal global QEPs. In the context of query shipping, cost-based optimization requires that it is possible to estimate the execution cost and selectivity of each sub-query at each corresponding peer, and the execution cost of the overall QEP that combines the results of the query fragments. Given that the cost is known for each sub-query, the compilation and optimization of the sub-queries themselves must be delegated to the respective remote peers, while the mediator peer optimizes only the assembly plan. If plan shipping is used, the mediator query optimizer must consider the execution costs of each individual proxy function and database operation in a query, and find the optimal order of all operations both in the sub-plans and in the assembly plan.

- **Fragment translation.** When source peers accept query fragments in a form different from the one used by the query peer, query fragments have to be translated in terms of the source peer data access interfaces. This translation step has to take into account differences in the data models, data representations, and data access methods. The translation process can augment query fragmentation when the capabilities meta-data is not detailed enough, so that only correct translations are produced. In such cases if the translation of a fragment fails it can be considered that the fragment can not be computed by the source peers.

- **Fragmented query execution.** Query sub-plans themselves can be viewed as computations that require some input data and produce some result data. Thus, sub-plans can be naturally wrapped by SDPFs in the query peer. Implementations of such SDPFs invoke the corresponding sub-plans installed at remote peers through the sub-plan handles and, as other SDPFs, translate input and result data to these sub-plans into appropriate representations. In this way the assembly QEP in the query peer can be viewed as consisting of SDPFs combined with local operations. Thus all observations about query execution with SDPFs apply to the execution of decomposed queries.

The major difference between plan shipping and query shipping, is that query shipping communicates requests for complex computations in the form of declarative sub-queries. These sub-queries are compiled and executed later by the remote peers. As a result, peers cooperate not only during query execution, but also at query compilation time. Thus, query shipping provides the means to distribute not only the execution of queries, but also the query compilation process in a PMS system. Remote peers that receive sub-queries for compilation can make a local decision based on their own information on how to process a query, i.e. this is the first class of interface capabilities where source peers participate in the compilation of a global QEP.

An interesting case of query processing arises when the source peers are capable of processing global queries themselves. One example of such peers are the mediators peers. If the source peers support the query shipping approach, then it can be applied recursively over the sub-queries. The process of query decomposition and optimization can be distributed among many peers, where every peer decides how to generate a QEP for its sub-queries. A global QEP for the whole query is produced by all peers in a cooperative query compilation process. In this way functions and operations can propagate through many levels of peers and finally be computed at source peers that

the query peer is not even aware of. Such a recursive application of plan shipping is not possible because the source peers are only capable of query execution.

3.5.4 Meta-Data

Query shipping does not require any additional meta-data to be exchanged between the peers compared to plan shipping. In fact, plan shipping requires that the cost for each remote operation and function is available to the query peer, while with query shipping the source peers have to provide only the total execution cost of a whole sub-query. Most existing data sources with query interfaces do not provide such information, thus the query peers need to "quess" the cost of sub-queries, e.g. by probing or using historical information.

The meta-data required for the plan and query shipping approaches is closely related to the capabilities of the peers. Below we analyze the peer capabilities that play a role in the two approaches.

3.5.5 Discussion

Since query shipping allows sub-queries to be executed at remote peers, network transmission costs can be reduced in the same way as with plan shipping. In addition, query shipping can result in better QEPs for remote sub-queries than plan shipping because each peer has more complete and up-to-date knowledge of the implementation of its local functions, operations and data statistics and can produce more efficient sub-plans than a query peer. In particular, when sub-queries in source peers reference local views, queries can be merged and optimized together with the expanded view definitions to simplify the sub-queries and to discover more efficient access paths to the data. Another performance benefit from query shipping is that it distributes the load of producing a QEP between the query and the source peers because both the query peer and its source peers cooperate to compile fragments of MDB queries.

The implementation of interoperable peers via query shipping requires that the peers provide a query interface to their data and therefore have the capability to process queries themselves. Thus, compared with all previous interface classes, query shipping requires the most complex peer implementation that includes a query compiler. Similarly to plan shipping, the ability to store precompiled sub-queries at remote peers raises the problem of how to keep the remote sub-queries consistent with the master queries in the query mediators.

3.6 View Definition Shipping

In a PMS, mediator peers can be freely composed logically in terms of other mediators and data source peers through database views. There is no central control of the data integration process and no central repository for all integrated views in all mediators. Instead, the users of each mediator define integrated views over a limited number of known peers with very little global knowledge about the rest of the PMS and the composition, capabilities and contents of the other peers. This ad-hoc approach to distributed data integration may result in a network of logically composed peers with redundancy because many peers may integrate the same source peers and even the same remote views through many different logical paths. If query processing in a

PMS follows the logical paths between the peers, this may result in many redundant computations and network data transfers. To alleviate this problem, the query peers must be able to discover the redundancy in the view definitions of their source peers. Similar to query processing over views in centralized DBMSs, for this a query peer must be able to expand the definitions of the views it queries, so that it can analyze these definitions together and optimize together the expanded view definitions. This requires that the source peers with a view definition capability provide some interface to their view definitions, so that the query peers can request these view definitions. We term this *view definition shipping*.

3.6.1 Functionality

In order for peers to exchange view definitions, the query peers have to *VSq1)* recognize which remote collections referenced in a global query are in fact defined as views, *VSq2)* request the definitions of views from remote peers, and *VSq3)* when necessary, translate the received view definitions into a local representation. Correspondingly, the source peers containing views must be able to *VSs1)* accept view definition requests, and *VSs2)* ship view definitions to the query peers.

Source peers with a view definition capability may in turn integrate other peers through their views. There are two alternatives for a query peer to fully expand all multi-level view definitions. First, the source peers may return view definitions in a format that specifies which other remote views are used in a view. Then the query peer may directly request the definitions of these remote views from their peers. Second, the query peer may request that the source peers themselves expand their own views and recursively request view definitions from their source peers on behalf of the query peer. The first alternative *VSr1)* is applicable in cases when the remote peers provide a multi-peer query language, and provide their view definitions, but can not be instructed to expand views themselves. Typically these can be other mediator systems or federated DBMS as DB2 [28] treated as data sources in the PMS. The second alternative *VSr2)* is mainly applicable to the design of the mediators in the PMS, because it is very unlikely that other systems may be instructed to request views from their source peers.

3.6.2 Query Processing

The traditional approach to process queries over views in database systems, here called *full view expansion*, is to recursively expand all view definitions and replace all view references in a query with their corresponding definitions until the query references only stored tables. Applied to a PMS, this approach requires that the query peers recursively request view definitions from their source peers until no views can be further expanded. After all views are expanded, the merged views can be simplified through declarative query rewrites, so that all redundant query operations are removed. Some mediator peers in a PMS may provide *fully virtual views* that only integrate other peers, but do not contribute their own data or operations. The expansion and simplification of such views through query rewrites completely removes all references to such fully virtual views, which reduces the number of peers accessed at query execution time. Expanded remote view definitions may reveal that several remote peers contain views that actually use the same remote common sub-views in some source peers. If global

queries over such peers are executed without expanding view definitions, then the query peer will access the same remote views via several different logical paths that pass through different peers. This may result in many unnecessary data transfers and redundant re-computations of the same views in the source peers. View expansion allows such redundant view compositions to be discovered and simplified through query rewrites.

After view expansion and simplification, query processing can continue through one of the methods for the interface classes described in the previous sections. For example the use of plan or query shipping allows to combine accesses to the same sources into single more selective queries and thus further reduce query processing and network transmission costs.

Full view expansion, however, has some disadvantages. In a PMS with many levels of composed mediators and sources, the expansion of some views may reveal that the views are defined in terms of many more other views in many peers. As a result a mediator query optimizer may have to optimize very large queries over large number of peers. Thus, full view expansion may result in prohibitively high compilation costs. One alternative to full view expansion is to limit the expansion process by some predefined resource, e.g. some number of peers is discovered, or some number of expansions is performed, or a time-out limit. Another alternative is to expand only the most "promising" views, that result in improvements for QEP quality with relatively little compilation cost. We term such a strategy *selective view expansion*. The experimental study of the tradeoffs between no view expansion, partial and full view expansion in [31] shows that the most promising views are the ones that contain common direct or indirect sub-views. Based on this observation, a heuristic view expansion approach may consider the topology of the logical composition of the views in a PMS and select for expansion only the views with common sub-views.

3.6.3 Meta-Data

Apart from the view definitions themselves, query processing with view expansion requires that the query peers can distinguish which of the proxy functions referenced in a query or view are defined as views themselves in their peers. The simplest approach is to use trial-and-error requests and let the query peers send view expansion requests irrespective of whether proxy functions actually represent remote views and deduce from the result if a view was successfully expanded. If the remote peers provide such meta-data, then view meta-data can be attached to proxy functions at their creation time.

In order to apply selective view expansion, the query peers need additional meta-data that provides them with enough information to decide which are the promising views for expansion. For example, remote peers may provide with each view meta-data about the peers accessed by the view. An overlap between the peers of several views is a necessary condition for common sub-views and can be used as an identifier of potential overlapping view definitions.

3.6.4 Discussion

If applied in its complete form view expansion may result in very high compilation costs that may outweigh the benefit in improved query execution times. Thus, the main

complexity in processing queries through view expansion is in implementing a heuristic that can balance the compilation cost with the benefit in QEP quality. View expansion is independent of and can be combined with any of the other interface classes. It may be particularly beneficial to combine view expansion with query shipping, so that expanded view definitions are grouped into sub-queries and shipped for remote processing. In this way many individual requests via different logical paths can be replaced by a single more selective sub-query. Finally, view expansion may radically change the execution data flow compared to the logical paths between the mediators. Unlike all other interface capabilities, view expansion allows that redundant mediators are completely bypassed at query execution time and thus remove much of the overhead of the logical composition of the mediators.

4 Implementation of Peer Mediators

The analysis of inter-peer interface capabilities and the related query processing techniques, presented in the preceding section, is based on our experiences from the implementation of the AMOS II peer mediator system based on the functional data model and query language described in Sect. 2. Below we describe our implementation of five of the six interface classes - SDPF, MDPF, MPPF, query shipping, and view shipping. We did not implement plan shipping because *i)* we did not encounter data sources that provide a plan shipping interface, and *ii)* as an alternative to query shipping, plan shipping is less suitable for the implementation of inter-mediator query processing.

Single-Directional Proxy Functions. The implementation of SDPFs in AMOS II is based on its built-in generic mechanism for extensibility - foreign functions. Thus, SDPFs are foreign functions implemented only in the forward direction, where arguments are bound and results computed, that perform the call shipping functionality described in 3.1.1. In addition to its implementation, each single-directional foreign function that implements an SDPF is associated with the remote peer where the SDPF is computed and with the corresponding remote data set represented by the proxy function. Data sources of the same kind provide the same physical interfaces to their functionality. For code reuse, proxy functions that access data from sources of the same kind share the same implementation. Foreign functions provide a generic way to associate with them either a static vector with cost information or a user-specified function (possibly foreign too) that dynamically computes the cost and selectivity of the foreign function. Similar to a shared implementation of all SDPFs for the same kind of sources, there can be a single source statistics function associated with all SDPFs for the same kind of sources. Schema information, such as the type signature and key information of proxy functions, is accessed again through *schema importation* foreign functions that "know" how to retrieve this type of meta-data from the corresponding kind of sources. Schema importation functions are generic per a kind of source and on invocation automatically create local proxy functions that correspond to schema objects in a remote source peer. Such functions are not defined for sources that do not provide an interface to their meta-data, and instead users create proxy functions manually.

The computation of global queries for data integration typically requires joining data from more than one proxy functions across the network. Therefore specialized join methods are needed that take into account and minimize network costs. We have developed three specialized inter-peer join algorithms, described in [26], that take into account limited access capabilities at the sources and reduce network transmission costs.

AMOS II mediators do not have special wrapper components separate from the foreign functions as, e.g., in DB2 Federated DBMS [19]. Instead wrappers in AMOS II comprise of the implementations of the foreign functions associated with the corresponding proxy functions, the corresponding cost functions, and the schema importation functions. Notice that all these are generic for a source kind. When more complex wrappers are needed that require the translation of SDPFs into source-specific access calls, AMOS II provides a rewrite mechanism that allows rewrite rules to be associated with the proxy functions of a source kind. For example, all JDBC data sources provide the same *executeQuery* method which can be wrapped by one generic implementation associated with all SDPFs that access data in a JDBC source. The query rewrite mechanism would translate all SDPFs that reference the same JDBC source into a single SQL sub-query string and replace all SDPFs of the same source with a single call to the generic SDPF implementation that wraps *executeQuery*, with the translated SQL string as a parameter. For sources that provide their meta-data, the wrapper implementor can define schema importation functions for each source kind. These schema importation functions, analogous to the *IMPORTFOREIGNSCHEMA* statement in SQL/MED [6], then can automatically create proxy functions from the schema of each concrete source and associate these proxy functions with their generic implementation and cost function.

Multi-Directional Proxy Functions. Multi-directional proxy functions in AMOS II are based again on the generic foreign function mechanism that allows to define multi-directional foreign functions and to associate binding patterns and cost functions with each separate implementation of a foreign function. Whenever sufficient source meta-data is available, the schema importation function(s) for a kind of sources automatically creates MDPFs and associates them with the corresponding binding patterns and cost functions. Queries over MDPFs are optimized as any other query over multi-directional foreign functions by cost-based optimization described in [37].

Multi-peer Proxy Functions. Our current implementation provides limited support for MPPFs [27] only for mediators and relational DBMS. There are two kinds of MPPFs. *Single implementation functions (SIFs)* are functions available only in one peer. *Multiple implementation functions (MIFs)* are functions available in all peers. Proxy functions are in general considered to be MIFs. However, for several special kinds of peers such as the mediators themselves and relational DBMS it is known that they always implement certain functions, e.g. inequalities. Such MIFs are known in advance and are pre-defined per each kind of source. Query optimization in the presence of MIFs, described in detail in [27], uses a special function placement heuristics that decides where to compute a MIF.

Query shipping. For better performance, inter-mediator query processing employs a query shipping approach, described in detail in [27]. Query processing is performed in several steps: *i) query decomposition* uses a heuristic procedure to form sub-queries in a way that minimizes data transfer between the peers, *ii) cost-based optimization* uses a dynamic programming algorithm to find optimal left-deep plans that combine the sub-queries into one multi-peer QEP, and *iii) plan tree rebalancing* [25] merges groups of nodes in the left-deep plan and replaces some of the right leaves of the tree with inter-peer sub-plans scheduled for execution at other peers. Tree rebalancing itself is based on query shipping, because the merged nodes of an inter-peer query plan are sent to other peers in the form of sub-queries. This allows remote peers to compile the sub-queries themselves and thus to decide autonomously how to compute the sub-plan.

AMOS II also supports query shipping for relational DBMS data sources. Since relational sources support a query language (SQL) different from that of the mediators, the mediator peers perform sub-query translation and simplification steps [13] in addition to the query processing steps for inter-mediator query shipping.

View definition shipping. As pointed out in Sect. 3.6, all previous classes of interfaces result in query processing that treats other peers as black boxes, and thus query execution follows the logical composition of the peers. To improve the performance of query processing for queries that involve many mediator peers, we implemented view definition shipping for inter-mediator queries in the form of *distributed selective view expansion (DSVE)* [31]. With DSVE mediators request the definitions of remote views that contain common sub-views in lower-level mediators. As shown in [31] this heuristics is promising in that only the most promising views are expanded so that compilation cost is reduced while the quality of the inter-mediator QEPs is improved.

5 Related Work

In this section we overview two types of research. First we look at data integration and/or data management systems with similar P2P architectures to the PMS architecture we described in Sect. 1. Then we classify into several categories works that analyze and compare various aspects of the design of interoperable components in distributed architectures.

5.1 Peer Data Management Systems

Several recent works [18, 21, 7] propose P2P architectures for data integration and for the management of distributed and autonomous databases. In the vision paper [18] it is indicated that placement and retrieval of data are fundamental problems in most P2P systems, and therefore DBMS technology can, and should be applied to P2P systems. That work concentrates on the problem of data placement in a P2P system. Another vision work [7], addresses the problem of semantic inter-dependencies in between autonomous peer databases in the absence of a global schema. Inter-peer semantic dependencies are described through coordination formulas in a *Local Relational Model (LRM)* that allows to specify at a logical level the synchronization of several peer databases.

Based on the assumption that data integration systems have one global mediated schema that integrates all sources, [21] advocates the concept of *peer data management systems (PDMS)*, as systems that replace the single logical schema of data integration systems with an interlinked collection of semantic mappings between the peers' schemas. The main problem addressed in [21] is that of schema mediation in a PDMS. Schema mappings between peer databases are expressed in a declarative language in combined global-as-view (GAV) and local-as-view (LAV) style. With respect to query processing, [21] deals with the problem of how to reformulate an initial query in terms of schema mappings into a query in terms of the base relations, a problem called *query answering*, when there are mixed GAV and LAV transitive mappings between peers.

5.2 Data and query shipping

An important issue in the design of client-server DBMSs is the distribution of processing between clients and servers. To study this issue, [15] first considers two extreme approaches to distribute query processing between clients and servers. With the *data shipping* approach all data is shipped from the servers to the clients and all query operators are executed at the client. Data shipping provides for good utilization of client resources, and is applicable in environments with powerful client nodes and fast networks. With the *query shipping* approach (as defined in [15]) on the other hand queries are submitted for computation at the servers and the clients directly receive the final results. Query shipping reduces communication costs for selective queries, shifts the load from the clients to the servers, and thus is suitable for client-server systems with powerful servers, resource-poor clients, and slow networks. As observed in [15], neither of the two extreme approaches suits all situations. A natural alternative is a *hybrid shipping* approach where some query operators are performed by the clients and others by the servers. The experimental study of the three approaches through a randomized query optimizer and a distributed database simulator show that hybrid shipping is superior to both extreme approaches due to flexible use of the resources at the clients and the servers.

Related to our analysis, data shipping as described in [15] corresponds to query processing for the SDPF interface class. Since the results in [15] refer only to the distribution of query execution, and do not consider the process of optimization itself, they are applicable to both query and plan shipping as described in Sect. 3.4,3.5. Hybrid shipping assumes that the query operators in a QEP can be computed by many nodes, thus this approach corresponds to a combination of the query processing approaches for nodes with both the MPPF and the query shipping interface classes.

5.3 Code shipping

Code shipping is a technique that dynamically extends the capabilities of remote peers by uploading and installing implementations of new functions/database operations. Two recent projects that employ code shipping are the MOCHA [47] and ObjectGlobe [9] prototype systems.

Both works assume that there are one or more libraries of Java classes that implement user-defined types and functions, and that remote peers, which are standalone wrappers, called Data Access Providers in [47], and CPU Cycle Providers in

[9], are capable of accepting class/method definitions and installing them locally. In both projects a centralized query optimizer takes into account the possibility to install user-defined operators at remote peers whenever the execution of the operators at the remote sites reduces the network data transfer.

The main advantage of code shipping is organizational. While many database and mediator systems are extensible, they require the intervention of a database administrator to install user-defined functions/operators, an approach that cannot scale to large numbers of peers. Code shipping performs automatic deployment of user-defined functions/operations whenever there can be a performance benefit from executing the operations at the remote peers, thus reducing administration costs.

Related to our analysis, query optimization for peers with a code shipping capability is similar to that of query optimization for MPPFs. The added meta-data is that of the location of the function/operation implementations, the size of the implementing code, and the local cost of extending a peer with new functionality. In terms of query optimization, the added complexity is that of considering the total cost of installing a function/operation at a remote peer. Once some remote peers are extended with a new function, all copies of this function can be considered as SDPFs that implement on MPPF, and therefore query optimization with code shipping is reduced to query optimization for MPPFs.

Code shipping poses additional security and interoperability problems that are outside of the scope of this work and addressed in [47, 9].

5.4 SQL and Management of External Data

The SQL/MED standard [40, 6], part of SQL:1999, addresses aspects related to the access to external data from SQL, and thus is directly related to our analysis. SQL/MED introduces several new SQL schema concepts in conjunction with SQL syntax extensions, and a set of routines for developing external data source wrappers in a standardized manner. Below we relate the concepts introduced in SQL/MED to concepts in our functional mediator data model and to our classification of interface capabilities.

Main concepts. Remote data collections in SQL/MED are represented through the notion of a *foreign table* and external data sources are represented via the concept of a *foreign server* that allows access to a set of foreign tables. Thus a foreign table in SQL/MED corresponds to the concept of a proxy function in our functional data model, and a foreign server in SQL/MED corresponds to a peer in our terms. If in SQL/MED a foreign server is a set of foreign tables, we see a remote peer as a set of proxy functions. *Foreign-data wrappers* in SQL/MED are code modules that provide access to the same kind of foreign servers, and thus they correspond to a set of SDPF implementations shared among SDPFs for the same kind of peers. Most of the SQL/MED standard deals with the specification of the APIs used by the foreign-data wrappers and the SQL servers to communicate during query planning and query execution.

Single-directional proxy functions. While SQL/MED does not by itself propose a concept similar to SDPFs, there is a corresponding concept in the SQL Foundation part [3] of SQL:1999 that introduces user-defined functions (UDFs) as functions implemented in some external language. Scalar UDFs in SQL:1999 correspond to singleton-valued SDPFs, and table UDFs correspond to bag-valued SDPFs, thus UDFs and SDPFs allow to express access to the same kind of remote data collections.

Multi-directional proxy functions. The closest concept in SQL/MED to MDPFs is that of foreign tables associated with foreign servers. Each foreign data-wrapper responsible for the access to a kind of foreign servers implements all functionality necessary for the access to the remote data collection represented by a foreign table. Unlike binding patterns for MDPFs, all information about limited source access capabilities is hard-coded in the SQL/MED wrappers and cannot be inspected or modified from the query language. Thus SQL/MED does not provide the means to freely associate a group of foreign functions that access the same abstract relation in different ways as is possible with MDPFs. Instead, UDFs in SQL:1999 and foreign-data wrappers in SQL/MED are separate concepts and UDFs cannot be reused in a incremental fashion to design wrappers. Instead the functionality of MDPFs in SQL/MED has to be simulated through a relatively complex wrapper implementation.

Multi-peer proxy functions. SQL/MED introduces *routine mappings* that allow to associate UDFs and built-in SQL functions with procedures in remote servers. Therefore routine mappings correspond to MPPFs that relate together equivalent SDPFs in different peers. There is no corresponding facility in SQL/MED that can relate together several foreign relations in the same way as MPPFs can relate several equivalent MDPFs.

Plan shipping. SQL/MED does not provide any direct support for plan shipping. Instead wrappers always receive sub-queries in a declarative form and the way they produce executable plans for their foreign servers is left to the wrapper implementor. Thus the SQL query optimizer can not generate by itself QEPs for foreign servers.

Query shipping. The main mode of submitting data requests to foreign servers in SQL/MED is to send sub-queries to the wrappers for compilation and execution. Thus SQL/MED fully supports the concept of query shipping as described in this paper. Query compilation with query shipping in SQL/MED is based on a request/reply paradigm. At compile-time the SQL server requests the execution of sub-queries from the wrappers. The wrappers in turn return responses that identify which parts of a sub-query they can handle. The SQL server must compensate for all operations that the wrappers cannot handle. Thus all information about the capabilities of a kind of sources is encoded in the corresponding wrapper and is not visible at the query language level.

View definition shipping. There is no corresponding capability in SQL/MED that provides functionality similar to view definition shipping.

Conclusion. SQL/MED by itself does not provide any guidelines how to implement wrappers and how to compose many peers (some of which SQL servers themselves) into a PMS. From this comparison of SQL/MED and our functional approach to mediation, we conclude that there are many similarities between the two, thus most of our results are directly applicable to any implementation of a PMS on top of an SQL:1999-compliant DBMS.

5.5 CORBA interfaces for database interoperability

The CORBA standard [2] provides an interoperability infrastructure that can be used to bridge platform- and communication- level heterogeneity of multiple databases. As pointed out in [11, 32] an important issue in the design of CORBA interfaces to database systems is the degree of granularity of the interface. At the finest level of granularity database rows or objects are wrapped through CORBA objects and are di-

rectly visible through the CORBA Object Request Service. This is a straightforward way to integrate distributed databases, however this approach has the disadvantage that most of the processing is performed by the client peer, query execution may result in very large number of cross-network requests, and it is hard to optimize requests to remote databases. At the other extreme are coarse granularity interfaces where remote DBMSs are wrapped through a single CORBA interface. This interface typically provides methods to query the data at the remote peers through a query language. The heterogeneity among different databases is resolved by providing different implementations for the same generic database interface. [11, 32] argue that for database interoperability it is necessary to use the second coarse-grain interface approach.

Related to our analysis, the fine-grain type of CORBA interfaces to databases corresponds to that of the SDPF interface class (Sect. 3.1), where CORBA objects correspond to proxy objects; CORBA methods, relationships and attributes correspond to single-directional proxy functions. Coarse-grain access to database peers, where there is one CORBA object that wraps the peer as a whole, corresponds to the query shipping approach in Sect. 3.5.

5.6 Conclusion

To summarize, existing works compare few design alternatives that correspond to some of the interface capabilities and corresponding query processing approaches considered in this paper. In addition, based on our distinction of interface capabilities and physical interfaces, our study is independent of the particular technology (such as CORBA, JavaRMI, etc.) used to implement interoperable peers, and therefore our results have wider applicability than similar works.

To the best of our knowledge there is no systematic study of the interface capabilities of mediator and data source peers, the meta-data that needs to be exchanged between the peers or provided to the peers, and the corresponding query processing capabilities enabled and/or required to process queries against composed mediator and data source peers.

6 Conclusions

We have analyzed *i)* the relationship between the interface capabilities of mediator and data source peers in a PMS, *ii)* the query processing techniques that can be applied at the mediator peers in the presence of various capabilities, and *iii)* the meta-data required for that. We classified peer interface capabilities into several classes with increasing amount of meta-data and increasing complexity of query processing in the mediator peers. Below we summarize the results of our analysis and discuss directions for future work.

Data shipping and computation shipping. Distributed systems in general allow two ways of cooperative processing - either data is shipped to the computation *data shipping*, or the computation is shipped to the data, *computation shipping*. With respect to this general sub-division, the first three proxy function interface classes require “pure” data shipping because they assume that the peers have some pre-existing fixed computational capabilities and data must be shipped to the peers that perform the

computations. Systems cooperating through data shipping may have high data transfer costs and under/over utilization of the resources at the peers. The last three interface classes - plan, query and view definition shipping, fall in the category of computation shipping. Characteristic of all three is that they allow the exchange of computations in limited non- Turing-complete query languages typical for database systems. The major advantage of this is that the exchange of computations between peers not only reduces the amount of data transferred across the network and provides better load distribution, but also allows computations to be combined at the peers and optimized together. In a P2P system with ad-hoc structure there may be many redundant computation requests, thus the optimizability of combined computations is very important so that redundancy can be discovered and removed.

Abstraction and performance. Another dimension for comparison of the six interface classes is the degree of abstraction they allow at the mediator query language level. The first three data shipping approaches provide an increasing degree of abstraction for the user and shift more and more of the performance decisions from the mediator user to the mediator query processor. For complex queries users may make suboptimal decisions which combinations of SDPFs result in best performance, thus the shift to a higher degree of abstraction in terms of MDPFs or MPPFs also results in better query execution performance.

The three computation shipping interface classes allow for query processing techniques that further improve the performance and scalability of pure data shipping, but do not add to the expressibility of the mediator queries.

Logical composition and physical query execution. An important characteristic of the SDPF and MDPF interface classes is that they only allow query execution plans that follow the logical composition of mediators where every proxy function is computed at its own source peer. The MPPF interface class provides more freedom to the mediator query optimizer to decide where to compute proxy functions, thus query execution may differ from the logical peer composition.

The plan and query shipping interface classes provide the means for mediator peers to instruct remote peers to execute complex computations that relate several proxy functions at once. In particular, remote mediator peers can be instructed to compute sub-queries without the involvement of the query mediator. Thus the execution flow may differ considerably from the logical composition of the peers and follow much more efficient network routes than that of the logical peer composition.

In its pure form, plan shipping allows more restricted global QEPs than query shipping because on one hand the query peers do not have knowledge of the implementation of remote data collections (e.g. views in other mediators), while on the other hand the remote peers do not have an optimizer of their own and cannot use their local knowledge to optimize their sub-plans. If plan shipping is combined with view definition shipping, then the query peers can import the definitions of all remote views referenced in a query and, given that detailed cost information is available, produce optimal global QEPs. The retrieval of all necessary meta-data and the subsequent query optimization by the query peers may be very costly due to large optimizer search space and large number of network meta-data requests, therefore we consider that plan shipping is not suitable for data integration systems with a P2P architecture.

Query shipping provides more flexibility than plan shipping and allows the query processors at the remote peers to take their own decisions about the execution of their sub-queries. Thus, a remote peer that provides a query shipping interface may merge sub-queries with local view definitions and recursively generate sub-queries of its own that may be sent to lower levels of peers. As a result query shipping allows cooperating peers to produce global QEPs that can not be produced by plan shipping. An additional advantage of query shipping is that it may propagate the execution of a proxy function through many mediator levels to the peer where it is most efficient to compute that function.

Finally, view definition shipping allows mediators to merge and analyze together view definitions from many mediators. The more view definition a query mediator retrieves the better picture it has of all peers relevant to a query and all operations performed at these peers. This information allows the query peers to discover redundancies in logically composed mediators and even to bypass redundant mediators for more efficient global QEPs.

Meta-Data and complexity of mediator implementation. We notice that query processing for each of the interface classes requires generally more meta-data than the previous ones. This additional meta-data provides information to the query mediators to choose from more alternative plans. However, it also requires more complex query planning that must consider an increasing number of alternative plans.

Design recommendations for PMSs. Based on our analysis of the six interface classes, we notice that query processing for all of them always reduces inter-peer query execution to the execution of SDPFs. Thus, a PMS can be designed incrementally starting from relatively simple mediators, and gradually adding more complex query processing capabilities that utilize the interface capabilities of the other peers.

The first mandatory step in the design of mediator peers in a PMS is to equip the mediators with the basic capability to logically relate SDPFs to remote data collections and to compute these SDPFs efficiently. For mediators based on the relational data model this means that it must be possible to represent remote data collections as relations and to compute these relations. When data integration problems require access to many data collections in many peers, the use of efficient join algorithms between SDPFs is particularly important. The basic SDPF functionality is sufficient to provide a fully-functional P2P data integration system and can be easily implemented on top of any extensible DBMS that supports user-defined functions.

Depending on the capabilities of the other peers, the mediators can be gradually extended to take into account more powerful kinds of peers that have some query processing capabilities. The main challenge is to extend the query processor of the mediators to take into account more meta-data and to explore more alternative query plans. All five interface classes apart from SDPF are independent of each other, therefore the mediators can can be gradually extended in different ways depending on the design goals.

Since mediators are the peers in a PMS that encode user abstractions over other peers, the design of the inter-mediators interfaces is crucial for the overall performance of a PMS. The design of inter-mediator interfaces may be as simple as SDPF, however, for the scalability of the data integration process, we believe that as minimum

mediators must support the MDPF abstraction, and it is desirable that they support MPPFs. With respect to performance, as shown by [15], it is essential that the mediators are capable of query shipping and thus have a query decomposer. Finally, as our study of logical mediator compositions [31] show, view definition may provide orders of magnitude improvement in query execution. We do not consider plan shipping to be suitable for inter-mediator query processing as it requires too much meta-data for a centralized compilation, it violates mediator autonomy, and it is less flexible than query shipping in load distribution and data transfer reduction.

Adaptive query processing. Our discussion of query processing for all interface classes assumed traditional static cost-based optimization which requires reliable cost estimates in order to produce efficient QEPs. Due to the autonomy and limited interface capabilities of many kinds of data source peers, in a PMS in most cases it is impossible to perform precise network and data access cost, and selectivity estimates. Data sources, network conditions and mediator load can all change in an unpredictable manner. Imprecise and changing costs may lead to sub-optimal query execution plans. Even if all necessary statistics information is available it is also infeasible to perform full cost-based query optimization in the traditional System R style due the potentially very large number of mediators, sources and views, which result in very large optimizer search space and thus very high optimization cost. Therefore it is essential for a mediator system to dynamically adapt to an unpredictable and changing environment.

A number of research projects, overviewed in [23, 17], have addressed dynamic and adaptive query processing. Most of the proposed approaches can be directly applied to query processing for the first three proxy function interface classes and plan shipping because in all of them query processing is performed in a centralized manner. However, with query shipping and view definition shipping the query processors of the peers have to cooperate in order to dynamically change a global QEP and adapt during query processing. Thus adaptivity in a PMS requires not only single-site adaptation, but also cooperative adaptation by all participating peers. To the best of our knowledge, current works do not address issues in cooperative adaptation, thus further research is required to investigate adaptive query processing for query shipping and view definition shipping.

Relationship to Web services. Our analysis is applicable not only to mediator database systems, but other middleware technologies as well, such as Web services [5]. A naive fine-grain use of Web services with SOAP RPC corresponds to the SDPF approach. From our analysis it follows that there is a wide spectrum of design alternatives to a simple usage of SOAP that both simplify the integration task for the user and may provide considerable performance improvements. We conclude that Web services may benefit from providing and standardizing service interfaces that provide computational capabilities analogous to the five interface classes apart from SDPF. Since the major performance improvements in a PMS are based on computation shipping capability, Web services can benefit from declarative descriptions that can be exchanged between composed services and optimized in similar ways as discussed in Sect. 3.

Future work. To better understand the effects of the various query processing techniques for each interface class, our next step is to study all interface capabilities ex-

perimentally. In reality, various interface capabilities may be combined in different ways, thus we plan to investigate the consequences of such combinations on the search space for the mediator optimizer and determine if new query optimization strategies are needed for combined interface capabilities.

References

[1] RPC: Remote Procedure Call Protocol Specification Version 2. Internet Network Working Group, Request For Comments 1057, June 1988.

[2] *Object Management Architecture*. John Wiley & Sons, New York, 1995.

[3] ISO/IEC 9075-2:1999, Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation). International Organization for Standardization, 1999.

[4] SOAP Version 1.2 Part 0: Primer. W3C Candidate Recommendation, http://www.w3.org/TR/soap12-part0/, December 2002.

[5] Web Services Architecture. W3C Working Draft, http://www.w3.org/TR/ws-arch/, November 2002.

[6] ISO/IEC 9075-9:2003, Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED). International Organization for Standardization, September 2003.

[7] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. In *Workshop on the Web and Databases, WebDB 2002*, Madison, Wisconsin, June 2002. SIGMOD 2002.

[8] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[9] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam, and Konrad Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[10] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The object data standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., 2000.

[11] Asuman Dogac, Cevdet Dengi, and M. Tamer Öszu. Distributed object computing platforms. *Communications of the ACM*, 41(9):95–103, 1998.

[12] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, pages 169–180. ACM, 1978.

[13] Gustav Fahl and Tore Risch. Query Processing Over Object Views of Relational Data. *VLDB Journal*, 6(4):261–281, 1997.

[14] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query optimization in the presence of limited access patterns. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 311–322. ACM Press, 1999.

[15] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 149–160. ACM Press, June 1996.

[16] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)*, 8(2):117–132, April 1997.

[17] Anastasios Gounaris, Norman W. Paton, Alvaro A.A. Fernandes, and Rizos Sakellariou. Adaptive Query Processing: A Survey. In *Proc. 19th British National Conference on Databases, BNCOD*, Sheffield, UK, July 2002. Springer-Verlag.

[18] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *WebDB Workshop on Databases and the Web*, June 2001.

[19] Laura Haas, Eileen Lin, and Mary Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–, 2002.

[20] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of 23rd International Conference on Very Large Data Bases, VLDB'97*, pages 276–285, Athens, Greece, August 1997. Morgan Kaufmann.

[21] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proceedings of the twelfth international conference on World Wide Web*, pages 556–567. ACM Press, 2003.

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

[23] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[24] Yannis E. Ioannidis and Raghu Ramakrishnan. Containment of Conjunctive Queries: Beyond Relations as Sets. *TODS*, 20(3):288–324, 1995.

[25] Vanja Josifovski, Timour Katchaounov, and Tore Risch. Optimizing Queries in Distributed and Composable Mediators. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, CoopIS'99*, pages 291–302. IEEE Computer Society, September 1999.

[26] Vanja Josifovski, Timour Katchaounov, and Tore Risch. Evaluation of Join Strategies for Distributed Mediation. In *5th East European Conference on Advances in Databases and Information Systems, ADBIS 2001*, volume 2151 of *Lecture Notes in Computer Science*, pages 308–322. Springer-Verlag, September 2001.

[27] Vanja Josifovski and Tore Risch. Query Decomposition for a Distributed Object-Oriented Mediator System. *Distributed and Parallel Databases*, 11(3):307–336, May 2002.

[28] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. Garlic: a new flavor of federated query processing for DB2. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 524–532. ACM Press, 2002.

[29] Kim Jungfer, Ulf Leser, , and Patricia Rodriguez-Tome'. Constructing IDL Views on Relational Databases. In *Advanced Information Systems Engineering: 11th International Conference, CAiSE'99*, volume 1626 of *Lecture Notes in Computer Science*, pages 255–. Springer-Verlag, June 1999.

[30] Timour Katchaounov. *Query Processing for Peer Mediator Databases*. PhD thesis, Department of Information Technology, Uppsala University, 2003.

[31] Timour Katchaounov, Vanja Josifovski, and Tore Risch. Scalable View Expansion in a Peer Mediator System. In *Eighth International Conference on Database Systems for Advanced Application, (DASFAA'03)*, pages 107–116. IEEE Computer Society, March 2003.

[32] Graham J. L. Kemp, Chris J. Robertson, Peter M. D. Gray, and Nicos Angelopoulos. CORBA and XML: Design Choices for Database Federations. In *Proceedings of the 17th British National Conferenc on Databases*, pages 191–208. Springer-Verlag, 2000.

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

[34] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems (TODS)*, 25(1):43–82, 2000.

[35] Ulf Leser, Stefan Tai, and Susanne Busse. Design Issues of Database Access in a CORBA Environment. In *Workshop Integration heterogener Softwaresysteme*, pages 74–87, 1998.

[36] Alon Y. Levy. Logic-based techniques in data integration. pages 575–595, 2000.

[37] Witold Litwin and Tore Risch. Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, 1992.

[38] Ling Liu, Calton Pu, and Kirill Richine. Distributed Query Scheduling Service: An Architecture and Its Implementation. *International Journal of Cooperative Information Systems (IJCIS)*, 7(2-3):123–166, 1998.

[39] Hongjun Lu, Beng-Chin Ooi, and Cheng-Hian Goh. Multidatabase query optimization: issues and solutions. In *Proceedings RIDE-IMS '93., Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 137–143, Vienna, Austria, April 1993.

[40] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna G. Kulkarni, and Peter M. Schwarz. SQL/MED - A Status Report. *SIGMOD Record*, 31(3), 2002.

[41] Wee Siong Ng, Beng Chin Ooi, Lee Tan, and Aoying Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *19th International Conference on Data Engineering*, March 2003.

[42] Kiyoshi Ono and Guy M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 314–325, Brisbane, Queensland, Australia, August 1990. Morgan Kaufmann.

[43] Fatma Ozcan, Sena Nural, Pinar Koksal, Cem Evrendilek, and Asuman Dogac. Dynamic Query Optimization in Multidatabases. *Data Engineering Bulletin*, 20(3):38–45, 1997.

[44] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, second edition edition, 1999.

[45] M. Tamer Özsu and Bin Yao. Building component database systems using CORBA. pages 207–236, 2001.

[46] Tore Risch, Vanja Josifovski, and Timour Katchaounov. Functional Data Integration in a Distributed Mediator System. In *The Functional Approach to Data Management*. Springer-Verlag, 2003.

[47] Manuel Rodriguez-Martinez and Nick Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. *SIGMOD Record*, 29(2):213–224, May 2000.

[48] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys (CSUR)*, 22(3):183–236, 1990.

[49] Anthony Tomasic, Louiqa Raschid, and Patrick Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[50] Jeffrey D. Ullman. Information Integration Using Logical Views. In *6th International Conference on Database Theory - ICDT '97*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer, January 1997.

[51] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[52] Vasilis Vassalos and Yannis Papakonstantinou. Expressive Capabilities Description Languages and Query Rewriting Algorithms. *Journal of Logic Programming*, 43(1):75–122, April 2002.

[53] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38–49, March 1992.

[54] Gio Wiederhold and Michael Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, Sept.-Oct. 1997. also in IEEE Intelligent Systems.

[55] Ling-Ling Yan, Rene J. Miller, Laura M. Haas, and Ronald Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *ACM SIGMOD Conference*, May 2001.

[56] Ramana Yerneni, Chen Li, Hector Garcia-Molina, and Jeffrey D. Ullman. Computing Capabilities of Mediators. In *Proceedings ACM SIGMOD International Conference on Management of Data (SIGMOD 1999)*, pages 443–454. ACM Press, June 1999.

Paper C:

Timour Katchaounov, Vanja Josifovski, and Tore Risch. Scalable view expansion in a peer mediator system. In *Eighth International Conference on Database Systems for Advanced Application, (DASFAA'03)*, pages 107–116, IEEE Computer Society, March 2003.

Scalable View Expansion in a Peer Mediator System

Timour Katchaounov
Uppsala University
Timour.Katchaounov@it.uu.se

Vanja Josifovski
IBM Almaden Research Center
vanja@us.ibm.com

Tore Risch
Uppsala University
Tore.Risch@it.uu.se

Abstract

To integrate many data sources we use a peer mediator framework where views defined in the peers are logically composed in terms of each other. A common approach to execute queries over mediators is to treat views in data sources as 'black boxes'. The mediators locally decompose queries into query fragments and submit them to the data sources for processing. Another approach, used in distributed DBMSs, is to treat the views as 'transparent boxes' by importing and fully expanding all views and merge them with the query. The black box approach often leads to inefficient query plans. However, in a peer mediator framework full view expansion (VE) leads to prohibitively long query compilation times when many peers are involved. It also limits peer autonomy since peers must reveal their view definitions. We investigate in a peer mediator framework the tradeoffs between none, partial, and full VE in two different distributed view composition scenarios. We show that it is often favorable with respect to query execution and sometimes even with respect to query compilation time to expand those views having common hidden peer subviews. However, in other cases it is better to use the 'black box' approach, in particular when peer autonomy prohibits view importation. Based on this, a hybrid strategy for VE in peer mediators is proposed.

1 Introduction

There has been substantial interest in using the mediator/wrapper approach for integrating heterogeneous data [14, 25, 11, 22, 7]. Most mediator systems integrate data through a central mediator server accessing one or several data sources through a number of 'wrapper' interfaces that translate data to a *common data model* (CDM). However, one of the original goals for mediator architectures [27] was that mediators should be relatively simple autonomous distributed software modules that encode domain-specific knowledge about data and share abstractions of that data with higher layers of mediators or applications. Composite mediators would then be defined in terms of other mediators and data sources through a high-level declarative language.

Compositionality of mediators allows to reuse available distributed resources on the Internet and to create new value-added mediation services in terms of existing

ones, while the autonomy of the sources and mediators is preserved. In the observable future it is most likely that data integration will be mostly a manual task. In order to scale integration to multiple autonomous sources, it is important that this task can be distributed among many parties with varying domain knowledge. We believe that a mediator architecture based on compositions of autonomous mediators is necessary to build large-scale data integration systems that are easy to tailor to existing infrastructure.

This paper investigates what are the implications of logical composition of distributed mediators on query compilation and execution performance and proposes a query processing technique suitable for the efficient execution of queries over composite mediators.

For our implementation we use the AMOS II peer mediator system [24]. To achieve modularity and distribution each mediator is an autonomous object-relational DBMS with its own query processor, storage, and catalog. AMOS II peers share many of the characteristics of peer-to-peer systems. AMOS II peers are autonomous because there is no global schema or global coordinator. Every mediator peer can act both as a client and a server to any number of other mediators. AMOS II peers communicate over the Internet via query compilation, query costing, view expansion and query execution requests in order to cooperatively process queries over composite mediators.

Mediator composition is based on a multidatabase query language that allows mediator peers to transparently access views, tables, and functions from remote mediators or data sources [23]. Logical composition of mediators is achieved when *multidatabase views* are defined in terms of views, tables, and functions in other mediators or data sources. Multidatabase views make groups of mediator peers and data sources appear to the user as a single virtual database.

There are two traditional approaches to implement distributed information systems. The first is the *black box* approach where distributed modules communicate with each other through some protocol without revealing the implementation of the services they export. This is the approach used in CORBA based systems and web services based on SOAP [3] and WSDL [4]. In the AMOS II peer mediator architecture the black box approach is equivalent to not to expand external views at all. It is common knowledge that this may lead to suboptimal query execution plans *(QEPs)* because of missed optimization opportunities.

On the other end is the *full view expansion* (*transparent box*) approach in distributed DBMS, where all views are expanded and merged with the query [21], independent of the location of the base tables and views that are used in a view definition. This 'reveals' to the query compiler the information 'hidden' in the view definitions which allows for better QEPs. Full view expansion could also remove unnecessary access to mediator peers. However, in a large scale peer mediator system using a cost-based query optimizer, full view expansion leads to prohibitively high compilation cost. Furthermore, full view expansion can only be made when permitted by the peer, to respect its autonomy.

We generalize both approaches and treat external mediator views as *grey boxes*, that is, when multidatabase views are defined in terms of other multidatabase views some of the view definitions are revealed to remote clients that query the views. We do this through a new query compilation technique for peer mediators, *distributed selective view expansion (DSVE)*. In DSVE, for better overall performance, mediators

control the level of transparency of the mediator peers by selectively expanding some multidatabase views.

To analyze the performance of DSVE we implemented two data integration scenarios scaled to up to 19 distributed AMOS II mediators with up to 12 commercial RDBMS data sources. As reference points we use the black box and the full view expansion approaches. We investigated the performance for both reference approaches under varying level of transparency and with respect to both query compilation and execution times. The analysis shows that DSVE can support the logical composition of mediators with little overhead and that this approach is superior to both black and transparent box approaches.

The rest of the paper is organized as follows. Section 2 investigates related work. Section 3 introduces the scenarios that are used throughout the paper. Section 4 describes the principles of DSVE and Section 5 investigates its performance followed by summary and future work in Section 6.

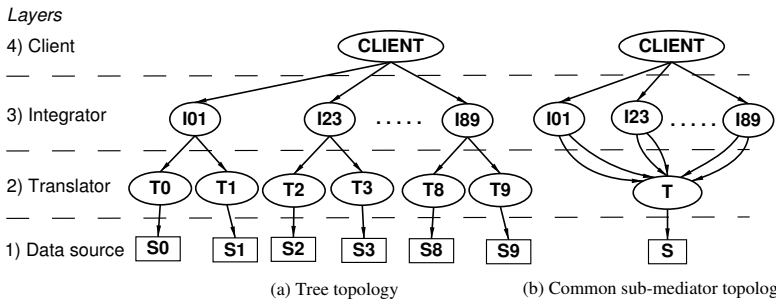
2 Related work

Distributed databases [1, 21] have complete global schemas describing on what sites different (fractions of) tables are located, while peer mediators do not have complete knowledge of meta-data from all mediators and data sources. Full expansion of all possible views in a distributed system with many nodes may be very costly. In [20] a restricted view expansion strategy for the System R* distributed database [5] is briefly mentioned but not evaluated.

To the best of our knowledge, there is no other study of the effects of a varying degree of view expansion in a distributed mediator or database system. No other mediator system (e.g. [14, 25, 7, 8, 22, 19]) use distributed view expansion.

The peer *peer data management system* (PDMS) architecture in [13] differs from ours by having a centralized catalog and therefore it is closer to a DDBMS. That work concentrates on data placement for PDMS. In [2] a data model suitable for PDMS is presented. Neither of the PDMS works studies query processing performance. Based on the similarity of PDMS with our peer mediator architecture, our results are readily applicable to the PDMS architecture.

Peer-to-peer (P2P) systems and *web services* have addressed the creation of large-scale integrated systems on the Internet. P2P systems, e.g Gnutella [12] and Freenet [10], address the problem of large scale sharing and replication of simple information objects such as files. P2P systems provide simple keyword search capabilities and do not support high-level abstractions as views. Most of the work on large-scale composition of distributed systems on the Internet is performed in the context of web services [6]. Problems related to composition of services are usually investigated from the perspective of workflow composition [26]. Our focus is on data integration and not on workflow/process composition. Web services are based on the SOAP [3] and WSDL [4] standards which provide no means for view definition exchange. Thus current proposals for composed web services treat wrapped DBMS views as black boxes.



(a) Tree topology (b) Common sub-mediator topolog

Figure 1: Logical compositions of mediators

3 Mediator composition scenarios

Having a potentially unlimited number of ways to compose mediators, we implemented for our study two scenarios that are simple enough to analyze the performance implications of view expansion in a peer mediator system. Our choice of scenarios assumes that data integration is performed with no global control or knowledge. Users define peer mediator views in terms of views in other mediators without knowing how those remote views are defined.

The integration scenarios are implemented using the AmosQL query language [24]. In this paper we define the scenarios in terms of equivalent SQL statements. Remote views defined in other mediators are referenced as $view@server$.

When participating in a logical composition, AMOS II peers can play several roles. *Translators* wrap different kinds of data sources and translate their data into the common data model (CDM) of AMOS II . *Integrators* reconcile conflicts and overlaps between similar real-world entities modeled differently in different sub-mediators [15, 16]. Users and applications can pose queries to any AMOS II peer, called the *client mediator* for the queries.

Scenarios. In the first scenario (Figure 1(a)) suppliers store information about parts in a RDBMS. Each supplier uses a translator that exports a view of the data. Several independent part resellers integrate information from the suppliers and present an integrated view hiding their information sources from their clients. A potential customer runs a mediator client that poses queries to the resellers' integrators.

In the second scenario (Figure 1(b)) the information about parts from all suppliers is stored in a single relational database. Each supplier has a single translator exporting the parts of that supplier. Each of the part resellers then exports an integrated view of the suppliers as in the first scenario. In a system with a global catalog such a scenario would look very artificial since the client mediator would discover in advance that all integrators access the same source of information. However in a peer system, this knowledge is not readily available. We assume that the integrators did not want to disclose their information source.

From the mediator client the two scenarios are equivalent and queries posed to the resellers' mediators would return exactly the same result. The differences are 'hidden' inside the view definitions of resellers' and suppliers' mediators.

Logical view integration graphs. To describe properties of mdiator compositions we define a *logical view integration graph (LVIG)* as a directed acyclic connected graph where vertices represent mediator peers or data sources and each directed arc represents the relationship *'is defined in terms of'* between a multidatabase view in one mediator and a view or table in another peer. Mediators are represented as ovals and data sources as rectangles. An LVIG represents a high-level view of the logical composition of mediators and data sources. Many distributed QEPs can be generated to compute the result of a query with the same LVIG.

The LVIGs of the two scenarios on Figure 1 differ in the topology of their LVIGs. Based on that we will name the first one as the *TREE* scenario and the second one as the Common Sub-Mediator (*CSM*) scenario.

3.1 Definitions of the mediators

The mediators and sources in the two scenarios are divided into four layers based on their roles:

The data source layer contains data stored in RDBMS. In the $TREE$ scenario the data for ten part suppliers is stored in different relational database tables, $PART$, each stored in its own DBMS S_i with the following schema:

```
CREATE TABLE part
(pnum integer not null,
 pname char(16) not null,
 quality integer, primary key(pnum))
```

In the CSM scenario all data about parts is stored in one relational database S in a single $PART$ table having one more column - a supplier id - and a composite key consisting of the part number and the supplier id. To simplify it is assumed that the same 'real' part has the same key $pnum$ in every relational source.

The translator layer consists of mediators providing views over the $PART$ tables. The translators T_i and T access the source data through an ODBC wrapper [9]. The translators could be hosted by independent application service providers or data source owners. In the $TREE$ scenario there is one translator T_i per relational source S_i. In the CSM scenario the single relational source S is wrapped by the translator T. In addition, in T each part supplier has a view, $part_i$, that selects parts from that supplier.

The integrator layer defines reconciliation views over the $part$ views defined in the translator layer. All integrator views are defined through the template below, where $[i]$ and $[j]$ are replaced by the indexes of the integrated translators for the $TREE$ scenario, and the indexes of the $PART$ tables for the CSM scenario, respectively. Each scenario uses only one of the two $FROM$ clauses.

```
CREATE VIEW part@I[ij] as
SELECT p0.pnum, p0.pname,
       combine_quality(p0.quality, p1.quality) AS quality
/* TREE scenario: */
  FROM part@T[i] p0, part@T[j] p1
/* CSM scenario: */
  FROM part[i]@T p0, part[j]@T p1
WHERE p0.pnum = p1.pnum;
```

In the $TREE$ scenario each mediator I_{ij} integrates information about parts from two translators T_i and T_j in the first $FROM$ clause. The $pnum$ attribute of the view is defined as the $pnum$ property of one of the joined tables. The $quality$ property is defined by the user-defined $combine\_quality$ function that encapsulates the knowledge of how to combine part qualities from different sources.

The integrators I_{ij} in the CSM scenario combine views of parts from the same part suppliers as in the $TREE$ scenario. However all the views $part_i$ in the translator T are defined in terms of the same relational table $PART$ in S as reflected by the second $FROM$ clause of the template.

From the mediator client both scenarios are indistinguishable as they export exactly the same views. Nevertheless the sources of information of the integrators differ.

Finally, *the top layer* has one mediator $CLIENT$ through which users pose queries to the $part$ views defined in the integrators I_{ij}. Depending on the remote views referenced in a query the corresponding LVIG may look different. The LVIGs on Figure 1 correspond to queries that reference all five available integrators.

To investigate multidatabase view expansion with respect to the number of participating mediator peers we use a class of test queries over a varying number of $part@I_{ij}$ views. A sample query over the $part@I01$ and $part@I23$ views defined in the integrators $I01$ and $I23$ is shown in Figure 2. The $quality\_part$ query states *what are the high-quality parts known to the* $I01$ *and* $I23$ *integrators*, where the $quality$ property ranges from 1 to 10.

```
select p1.pname
from part@I01 p1, part@I23 p2
where p1.quality >= 7 and
      p2.quality >= 7 and
      p1.pnum = p2.pnum;
```

Figure 2: Query $quality\_parts$ over I01 and I23

The $quality\_parts$ query is scaled by adding more $part@I_{ij}$ views from other integrators through equi-joins on the $pnum$ attribute and inequality predicates on each $quality$ attribute.

4 Multidatabase view Expansion

First the black box approach to process queries over multidatabase views is described, followed by a discussion of its potential problems. To remedy the major deficiency of the black box approach, poor QEP quality, we describe how to extend the mediator query processor with a general mechanism for exchanging view definitions between the mediator peers. In its simplest form this mechanism is equivalent to full view expansion. After discussing the advantages and problems of full view expansion we describe what is needed to achieve the best of both worlds - a generalized approach to multidatabase view expansion that allows the query optimizer of each mediator peer to explore the full range of possibilities between no and full view expansion.

4.1 Processing multidatabase views as black boxes

Queries in AMOS II are parsed and rewritten [18, 9, 15, 16] into a typed predicate calculus representation, ObjectLog [18], extending Datalog with predicate type signatures. In this paper we use SQL notation. For local queries rewritten calculus expressions are transformed by a cost-based query optimizer into an optimized object algebra expression [18, 9] which is interpreted to produce the query result. For multidatabase queries, before the query optimization phase, the calculus representation of the query is decomposed into multidatabase subqueries. At each mediator peer its cost-based optimizer generates optimized QEPs for the each of the subqueries. The query decomposition is performed in two main stages [17]: heuristic-based *predicate grouping* and cost-based *subquery optimization*.

The predicate grouping groups the query predicates into one or more composite predicates (subqueries). The result is one or more subqueries per each remote peer. After the predicate grouping phase the query in Figure 2 is divided into two subqueries (views) $SQ@I01$ and $SQ@I23$ that consist of predicates from $I01$ and $I23$ (Fig. 3 and 4).

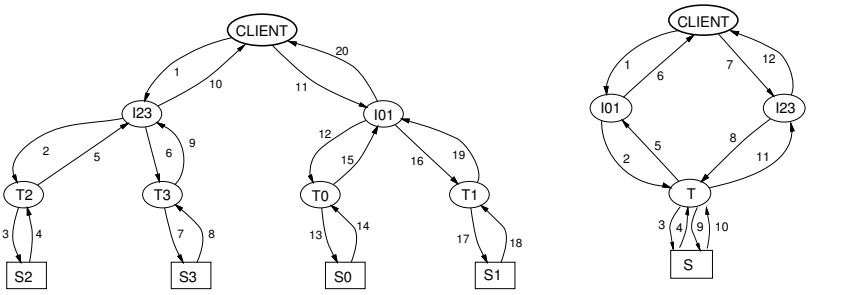
```
create view SQ@I01 as
select p0.pnum, p0.pname
from part@I01 p0
where p0.quality > 7;

create view SQ@I23 as
select p0.pnum
from part@I23 p0
where p0.quality > 7;
```

Figure 3: Subqueries after predicate grouping

```
select s0.pname
from SQ@I01 s0, SQ@I23 s1
where s0.pnum = s1.pnum;
```

Figure 4: $quality\_parts$ after grouping

The *subquery optimization* phase decides on the execution order of the subqueries which determines the data flow between the mediators. The two subqueries $SQ@I01$ and $SQ@I23$ are sent for compilation and costing to the integrators $I01$ and $I23$ to determine variable bindings and execution order for the subqueries. Based on the binding and cost information an executable plan is produced for the query in the client mediator and the subqueries in their respective mediators. These optimized plans for given binding patterns are saved in the mediator databases. The same process is applied recursively for subqueires that are themselves multidatabase queries in their respective mediators. Notice that the client mediator does not 'know' (and does not have to know) that $part@I01$ and $part@I23$ are actually views.



a) Treee topology b) Common sub-mediator topology

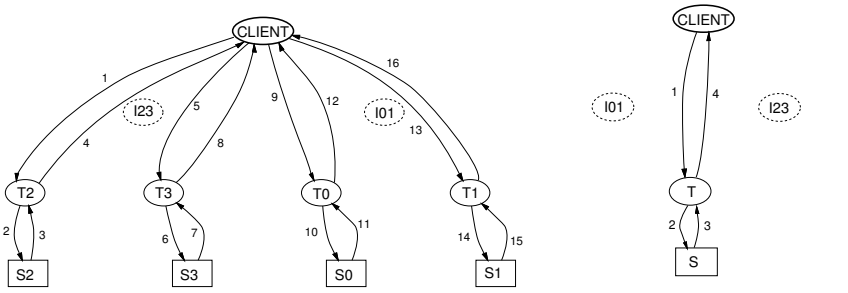
Figure 5: DDFGs for query $quality\_parts$ generated by the black box approach

Distributed data flow graphs. A useful tool to understand distributed QEPs is a graph that represents the flow of data during the execution of a multidatabase query. A *distributed query execution data flow graph (DDFG)* is a directed connected graph where the vertices in the graph represent mediator peers or data sources. There are two kinds of edges with respect to each vertex: *call* edges are out-edges that represent remote subquery execution requests (with optional parameters), *data* edges are the in-edges of a vertex representing the incoming flow of tuples that correspond to each request. All edges are numbered according to their execution order. DDFGs reflect only the distribution aspects of a query execution plan. Many DDFGs may correspond to a single multidatabase query.

For the $quality\_parts$ example query the black box approach to distributed query compilation described above generates DDFGs similar to those in Figure 5. All other DDFGs corresponding to the same query are different only in the order the nodes from the same layer are accessed. As one may expect the DDFGs on Figure 5 are very similar to the LVIGs for the same query in Figure 1. Thus the black box approach to query compilation produces QEPs that follow the logical view composition topology.

Advantages and disadvantages of the black box approach. Treating remote views as black boxes has some advantages. When remote views are not expanded a multidatabase view definition is often smaller and refers to fewer mediators than the expanded one. All the compilation effort spent to generate plans for the remote views can be reused because AMOS II stores precompiled parameterized views as functions that can be directly invoked. Therefore we can expect better compilation times when no views are expanded. Another advantage is that the integrators do not have to reveal their view definitions to the client mediator. This respects the autonomy of the mediators and the black box approach may be the only possible one if a peer mediator doesn't reveal view definitions to other mediators.

The main disadvantage of the black-box approach is that it can lead to suboptimal QEPs. In the context of a peer mediator system sub-optimality can be due to several reasons. A QEP may not be able to make use of hidden existing indexes in other mediators or sources. Similarly it is not possible to increase the selectivity of subqueries by merging predicates from remote views in different mediators. As in Figure 5 intermediate mediators are accessed despite that their view definitions do not access any



a) Treee topology b) Common sub-mediator topology

Figure 6: DDFGs for query $quality\_parts$ generated by the full view expansion approach

local data. In deep mediator networks this may result in considerable network overhead and unnecessary load on mediators. In the case of queries with LVIGs having TREE topology the distributed subquery scheduler at each mediator peer has fewer options for distributed join ordering. For queries with LVIGs having CSM topology a client cannot detect that more than one of its sub-mediators access data from the same source as in the scenario on Figure 5(b).

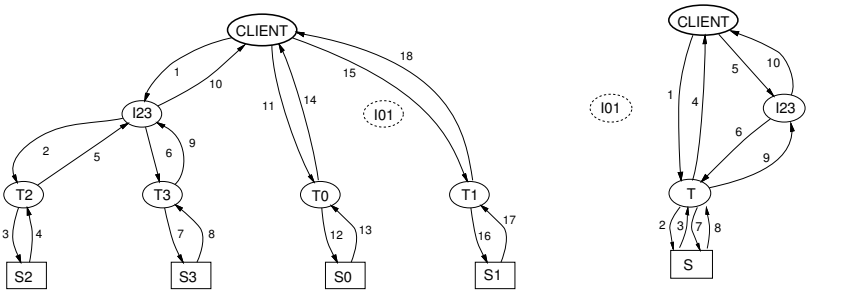
4.2 Full expansion of multidatabase views

To solve the problems of the black box approach described in Section 4.1 a logical step is to follow the approach employed in modern DBMSs (distributed or not) - to fully expand all view definitions. For the $quality\_parts$ example query this implies that the definitions of the view $part$ in the integrators $I01$ and $I23$ should be revealed to the client mediator.

After collecting the expanded definitions of all the remote subqueries, the subqueries in the original query are replaced by their expanded definitions and all predicates are grouped into subqueries. The query processing continues with the cost-based subquery optimization phase in the same way as in the black box approach.

Figure 6 shows some possible DDFGs for the $quality\_parts$ query from Figure 2 after performing full view expansion. The $CLIENT$ mediator eliminates all redundant mediators (dotted circles). In the the CSM scenario in Figure 6(b) the view definitions at the two integrators are combined in a single query together with the query predicates and the translator T is accessed only once. When supported by the data sources the combined predicates can be pushed to the sources which may further improve performance.

While full view expansion is very promising in terms of potential benefits in execution time, the cost to compile queries over fully expanded views may be prohibitively high. An expanded remote view definition may reveal that it has been defined through many mediators thus resulting in an explosion of the number of peers the query optimizer must consider. For example if we scale our scenario to ten integrators, each of them having an integrated view over ten translators, full view expansion of a query over the ten integrators will lead to a distributed query involving a hundred peers. At



a) Treee topology b) Common sub-mediator topology

Figure 7: DDFGs for query $quality\_parts$ generated by the $dsve_1$ strategy

the same time each of the subqueries of the distributed plan may contain many join predicates. As there is no global catalog in a peer mediator system the query optimizer must execute a remote cost estimate request for every query fragment that can be executed in a remote peer. This may result in a high cost of getting the cost. Finally due to incorrect cost estimates typical in a distributed mediator system the optimizer might still produce sub-optimal QEPs.

Finally, full view expansion does not respect mediator and source autonomy by forcing all mediators to reveal their view definitions. This makes full VE unsuitable for integration of data from independent information providers.

4.3 Selective expansion of multidatabase views

A natural idea is to generalize the processing of multidatabase views so that the query processor adapts itself to the query being compiled, the logical composition topology of the multidatabase views being queried, and the autonomy requirements of each mediator peer. Such a general approach should combine the good sides of both the black box and the full view expansion approaches: reasonable query compilation cost, good query execution performance, and respect of site autonomy.

We have implemented such a generalized mechanism in the AMOS II mediator system, named *distributed selective view expansion (DSVE)*. It allows to selectively expand only some of the multidatabase views. DSVE is generic in the sense that it allows various strategies to be used to select which of the subqueries in a multidatabase query should be view expanded. In particular, when no remote views are expanded DSVE is reduced to the black box approach, and when all subqueries are expanded DSVE is equivalent to the full view expansion approach. We use the term *partial view expansion (partial VE)* for all other DSVE strategies.

To achieve good performance DSVE's view selection strategy should expand views if it leads to high QEP quality improvement without dramatically increasing the optimization time for the expanded query. The DSVE strategy should scale well over the number of remote views. To preserve the autonomy of the mediator peers the strategy used in DSVE should require as little information as possible to be imported from mediator peers.

To investigate the tradeoffs between compilation time and QEP quality with vary-

ing number of expansions, we start with a family of simple strategies where each strategy performs a fixed number, $NExp$, of expansion requests per query. When $NExp$ is equal or bigger than the total number of subqueries, DSVE is equivalent to full view expansion. If $NExp = 0$ DSVE reduces to the black box approach. Let us denote each of these strategies as $DSVE\_N$. Figure 7 shows the resulting DDFG after the compilation of the $quality\_parts$ test query when the $DSVE\_1$ strategy was used to expand the view in integrator $I01$. In the following section we perform a set of experiments where we vary the $DSVE\_N$ strategy for both the TREE and CSM mediator composition scenarios from Section 3.

5 Experimental evaluation

The experimental goals are: *i)* quantify tradeoffs between no, full and partial VE; *ii)* test hypothesis that DSVE may lead to best overall performance; *iii)* understand properties of a DSVE strategy with good overall performance.

In all experiments we execute scaled versions of the test query $quality\_parts$ in Figure 2 for both scenarios. This allows us to include the topology of the LVIG of the query as a parameter in the experiments. We investigate the scalability of view expansion by varying number of expanded views and the number of integrators joined by the test query.

5.1 Experimental setup

We used three 600 MHz Dell Optiplex GX1 computers with 512 MB RAM running Windows 2000 interconnected by a fast 100 Mbit LAN. Each of the mediator layers (client, integrator, translator) run on separate computers. The query compiler of AMOS II generated synchronous QEPs allowing us to run several mediators on the same computer without any interference. During the experiments it was ensured that each of the nodes preallocates enough RAM to complete the experiment without swapping. All translators accessed a DB2 RDBMS through an ODBC wrapper. The $PART$ tables in the DB2 databases were populated with synthetic data, all with the same number of rows and even distribution of all join columns. All join columns of the $PART$ tables were indexed.

5.2 Compilation tradeoffs

First measurements investigate how the compilation time for a multidatabase query over multidatabase views depends on the number of view expansions for varying number of integrator views. Figures 8 and 9 show this dependency for LVIGs with TREE and CSM topology. Each point in the graphs corresponds to one compilation experiment. There is one curve per fixed number of expansions. Points with the same x-axis (same number of integrators) correspond to the same query compiled with different number of view expansions. The curves in the graphs partially coincide when the number of expansions are equal to or more than the total number of integrators. While our experiments were performed for all possible numbers of expansions between none and full, for clarity we removed some the experimental curves that do not change our conclusions.

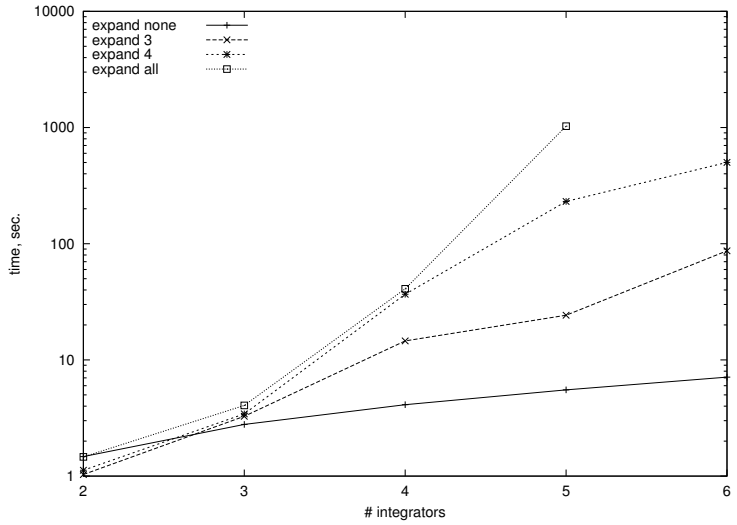


Figure 8: Query compilation times for different DSVE strategies, $TREE$ topology

The compilation cost of a multidatabase query is distributed among the components of the query compiler: the local query compiler and the distributed query optimizer. Both optimizer components use dynamic programming (DP) to find the optimal executable order of subqueries and the predicates in subqueries. Therefore query compilation cost depends exponentially both on the number of remote sub-queries and the number of predicates per sub-query.

Figure 8 shows experimental results for queries with TREE topology LVIGs. The y-axis of the graph is in logarithmic scale because of high value ranges. As expected, the more expansions are performed, the longer compilation time. Full VE expansion leads to exponentially growing compilation time and for 5 integrators it is 186 times more than with no VE. For 6 integrators and full VE (curve $expandall$) the experiment could not complete in 10000 seconds. Two factors contribute to the exponential behavior of full VE: *i)* DP is used to find optimal execution order of the remote subqueries; *ii)* in our scenario each expansion of a view on the integrator level reveals two more views from the translator level, thus increasing the distributed query optimizer search space. All other strategies result in compilation times between the two naive strategies: black box (curve $expandnone$) and full VE (curve $expandall$).

The experiments for queries with CSM LVIG topology (Figure 9) uncover completely different behavior than with TREE topology. The total time to compile the worst case of 5 integrators is 257 times less than with the TREE topology. Contrary to the common belief that the more views are expanded, the higher compilation cost, here we observe the opposite behavior up to 5 integrators: the more views are expanded, the less compilation time. This unexpected result is due to savings both in the local and the distributed query optimizer components. When expanded, the views on the integrator level reveal that they are defined in terms of the same mediator, the translator CSM on Figure 1(b). After all expanded views are merged and their predicates are grouped into a single subquery (executed at the translator T) it is simplified

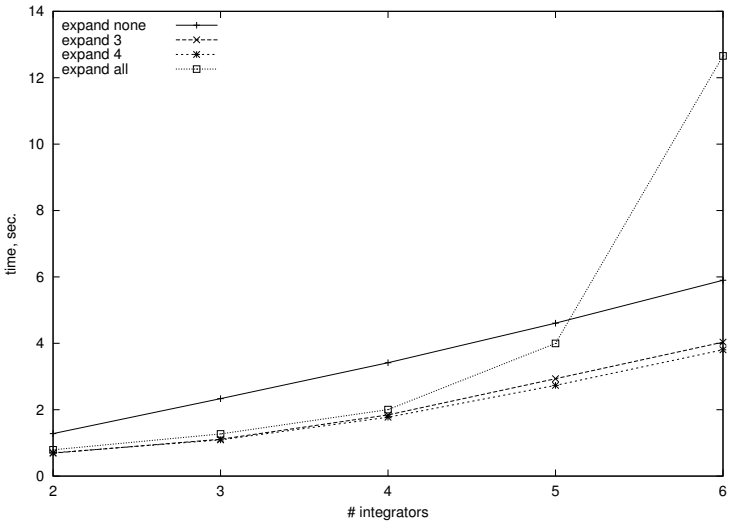


Figure 9: Query compilation times for different DSVE strategies, CSM topology

by query rewrites (Section 4.2). As a result the distributed sub-query optimizer at the client mediator has fewer predicate groups to optimize (just one) while the number of predicates for the local optimizer does not grow. After the number of integrators grows over 5, full VE leads to slower compilation time due to the large number of relational sources being accessed by the large subquery resulting from the view merge. This subquery is compiled in the translator T and increases the compilation time there.

We conclude that the more distinct sub-views are revealed by VE, the higher is compilation cost, and the dependency is exponential in the worst case. Furthermore if DP is used for query optimization full VE becomes too expensive when it results in more than 9 to 10 distinct sub-views. Finally, expansion of views with a common sub-mediator does not increase compilation time dramatically, and in some cases it may result in lower compilation time.

5.3 Execution plan quality

The next step in our evaluation of VE is to check two hypotheses made earlier: *i)* the more views are expanded the better the quality of the resulting QEP and *ii)* partial VE leads to sufficiently good plans with low compilation costs. Figure 10 represents the execution time of the test query $quality\_parts$ in Figure 2 scaled to 5 integrator views where all $PART$ tables contain 6000 tuples. The test query is precompiled for both LVIG topologies ($TREE$ and CSM) with varying number of view expansions resulting in different QEPs. The number of expansions varies between 0 (black box) and 5 (full VE). The quality of the QEPs is evaluated by measuring their actual running time.

For both topologies we observe improvement in the QEP quality in Figure 10 when the number of expansions grows. This confirms assumption *i)*. Notice that full VE improves the plan quality in the $TREE$ topology with only 24% while in the

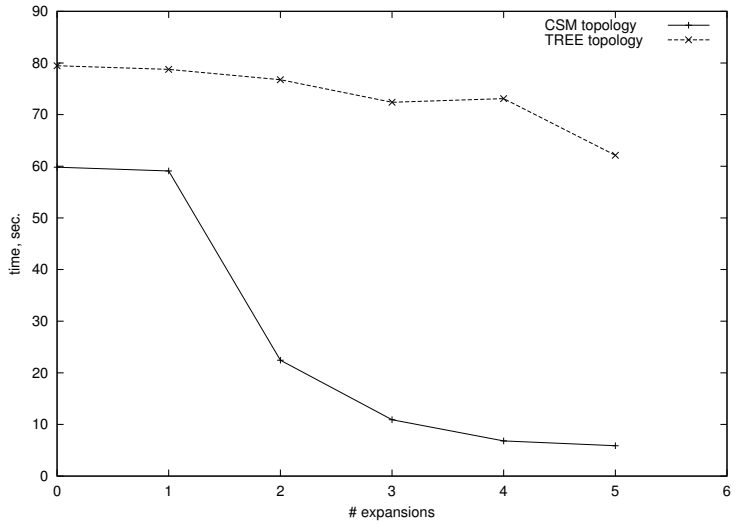


Figure 10: Plan quality for 5 integrators

CSM topology the improvement is 10 times.

Table 1 compares the ratio between the relative time to compile a query with varying number of expansions and the corresponding relative quality improvement for the experiment on Figure 10. The table consists of two similar parts, one for the test query being compiled and run against a LVIG with $TREE$ topology, and one for the CSM toplogy. For each topology the first row [abs. comp.] shows the absolute times to compile the query, the next row [rel. comp.] shows the compilation times relative to the time for 0 view expansions. The row [improvement] shows the ratio of the execution time with no expanded views (as a worst case) to all execution times from Figure 10. Finally the row [rel. cost for improvement] shows the ratio between the [rel. comp.] cell and the [improvement] cell which is an estimate of how much did it cost in compilation time to achieve an improvement in the quality of the QEP.

For the $TREE$ topology the last row [rel. cost for improvement] shows that the more views we expand the more costly it is to improve the quality of the QEP while at the same time from row [improvement] we can see that even with full view expansion (5 expansions) we achieve only minor improvement of 1.28 times (22%) for which it took 177.9 times longer (983.6 seconds) to compile the query. In this case a good tradeoff is to perform partial expansion of 3 integrator views which takes only 4.4 times longer (24.3 sec.) to achieve 1.1 times (9%) improvement. Therefore in the case of a $TREE$ topology partial VE produces a better plan with relatively low cost, while full VE leads to prohibitively high cost for plan improvement which confirms assumption *ii)*. We can also notice that even with no VE at all the resulting QEP is pretty good.

The compilation and execution of the test queries in the CSM topology exposes radically different behavior. Partially expanding 3 integrator views improves the plan quality 5.5 times where the compilation time is 60% of the time for the non-expanded case. Therefore assumption *ii)* is true in the case of CSM topology as well. Full VE

number of expansions	0	1	2	3	4	5
TREE						
abs. comp. (sec.)	5.5	5.7	9.5	24.3	231.3	983.6
rel. comp.	1	1.04	1.7	4.4	41.8	177.9
improvement	1	1.01	1.04	1.1	1.1	1.3
rel. cost for improvement	1	1.03	1.66	3.99	38.38	139.01
CSM						
abs. comp. (sec.)	4.6	4.3	3.5	2.9	2.7	3.8
rel. comp.	1	0.9	0.8	0.6	0.6	0.8
improvement	1	1.01	2.7	5.5	8.8	10.2
rel. cost for improvement	1	0.91	0.28	0.12	0.07	0.08

Table 1: Compilation cost vs quality

in this case leads to 10.2 times improvement in the quality of the QEPs which requires less time (only 80%) than with no VE.

The conclusions are that in the general case partial VE produces sufficiently good plans with relatively low compilation cost. If we know that we are compiling a query over views with a $TREE$ topology of the LVIG, the compilation cost can be radically reduced by not expanding any views at all without sacrificing the quality of the QEP. By contrast, when compiling queries against views with CSM topology, full VE can lead to radical improvements in the quality of the QEPs with very low compilation cost.

6 Conclusions and future Work

We proposed a new approach, *distributed selective view expansion (DSVE)*, to process compositions of multidatabase views in a peer mediator system. In DSVE, some of the views defined in remote mediators are selectively expanded to balance between query compilation time and QEP quality for best overall performance. To minimize the number of expansion requests and to allow optimizations of the expanded remote views DSVE uses predicate grouping to combine query predicates into subqueries. We present a performance study of DSVE with respect to its scalability over the number of remote views both for query compilation and query execution. As a reference we use two traditional approaches, the black box and the full VE approach which are special cases of DSVE.

The experiments show that neither of the two reference approaches (black box and full VE) is suitable for a peer mediator system, because none of them performs well in all cases. Contrary to the common belief that VE is always beneficial, our experiments show that it is not favorable to always perform full VE because in some cases it leads to very high compilation costs without radical improvements in query execution time. In LVIGs with $TREE$ topology VE increases the number of views directly visible to a client node, and given that cost estimates are highly unreliable in a peer mediator system, this often results in suboptimal plans. Therefore VE for $TREE$-like LVIGs

defeats its own purpose - to improve the quality of the QEPs. On the contrary, more view expansions for queries with CSM-like LVIGs result in compilation times orders of magnitude lower than in a $TREE$-like LVIG, while the quality of the plans improves up to 12 times. In the case of CSM-like LVIG topologies VE can drastically reduce the query execution time when information from several hidden sub-mediators can be combined. The topology of the LVIG of a multidatabase query plays a crucial role in the VE process. For $TREE$ topologies the best strategy is to expand only few of the remote views while for the CSM topology all (or almost all) views should be expanded.

The performance improvements of DSVE in processing queries over logically composed mediators are due to more selective queries, smaller data flows between the servers, fewer servers involved in the query execution, while spending relatively little effort in query compilation. Our performance study shows that DSVE allows for efficient query processing in logically composed mediators.

We are currently designing a view expansion strategy for DSVE that selects for expansion the views most likely to lead to an improved QEP with low compilation cost. Such a strategy should utilize the information hidden in the topology of the LVIG to leverage the common view definitions for better plans and lower compilation cost. A DSVE strategy should also evaluate the potential number of remote subqueries it will produce for the distributed optimizer and take into account the total number of predicates per subquery to reduce the distributed and local query compilation costs.

References

[1] P. M. G. Apers, A. R. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, January 1983.

[2] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. *Proc. 5th Intl. Workshop on the Web and Databases, WebDB 2002*, Madison, Wisconsin, June 2002.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. *Simple object access protocol (SOAP) 1.1*. W3C Note, http://www.w3.org/TR/SOAP/, May 2000.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, http://www.w3.org/TR/wsdl, March 2001.

[5] D. Daniels, P. G. Selinger, L. M. Haas, B. G. Lindsay, C. Mohan, A. Walker, and P. F. Wilms. An introduction to distributed query compilation in R*. *Proc. of the 2nd Intl. Symposium on Distributed Data Bases*, 291–309, Berlin, September 1982. North-Holland Publishing Company.

[6] *IEEE Data Engineering Bulletin*. Special issue on infrastructure for advanced E-services. 24(1), March 2001.

[7] W. Du and M. Shan. Query processing in Pegasus. In O. Bukhres, A. Elmagarmid (eds.):*Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Pretince Hall, Englewood Cliffs, 1996.

[8] C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Multidatabase Query Optimization. *Distributed and Parallel Databases*, Kluwer, 5(1), 77-114, 1997.

[9] G. Fahl and T. Risch. Query processing over object views of relational data. *VLDB Journal*, 6(4):261–281, 1997.

[10] *Freenet.* http://freenet.sourceforge.com/.

[11] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, April 1997.

[12] *Gnutella*. http://www.gnutella.com/.

[13] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? *Proc. 4th Intl. Workshop on the Web and Databases, WebDB 2001*, 31–36, June 2001.

[14] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. *Proc. 23rd Intl. VLDB Conf.*, 276–285, Athens, Greece, August 1997.

[15] V. Josifovski and T. Risch. Functional query optimization over object-oriented views for data integration. *Journal of Intelligent Information Systems*, 12(2-3):165–190, 1999.

[16] V. Josifovski and T. Risch. Integrating heterogenous overlapping databases through object-oriented transformations. *Proc. of 25th Intl. VLDB Conf.*, 435–446, Edinburgh, Scotland, UK, September 1999.

[17] V. Josifovski and T. Risch. Query decomposition for a distributed object-oriented mediator system. *Distributed and Parallel Databases*, 11(3):307–336, May 2002.

[18] W. Litwin and T. Risch. Main memory oriented optimization of oo queries using typed Datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):517–528, 1992.

[19] L. Liu and C. Pu. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *Distributed and Parallel Databases*, 5(2):167–205, April 1997.

[20] G.M.Lohman, C.Mohan, L.M.Haas, D.Daniels, B.G.Lindsay, P.G.Selinger, and P.F.Wilms. Query processing in R*. In W.Kim, D.S.Reiner, and D.S.Batory (eds.): *Query Processing in Database Systems*, 31–47. Springer Verlag, 1985.

[21] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[22] K. Richine. *Distributed query scheduling in DIOM*. Tech. report TR97-03, Comp. Sc. Dept., Univ. of Alberta, 1997.

[23] T. Risch and V. Josifovski. Distributed data integration by object-oriented mediator servers. *Concurrency and Computation: Practice and Experience*, 13(11):933–953, 2001.

[24] T. Risch, V. Josifovski, and T. Katchaounov. *Amos II concepts*. http://www.csd.uu.se/∼udbl/amos/doc/, 2000.

[25] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. on Knowledge and Data Engineering*, 10(5):808–823, 1998.

[26] *VLDB Journal*. Special issue on E-services. 10(1), 2001.

[27] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.

Paper D:

Vanja Josifovski, Timour Katchaounov, and Tore Risch. Optimizing queries in distributed and composable mediators. In *Proceedings of the Fourth IFCIS International Conference on Cooperative Information Systems, CoopIS'99*, pages 291–302, IEEE Computer Society, September 1999.

Optimizing Queries in Distributed and Composable Mediators

Vanja Josifovski, Timour Katchaounov and Tore Risch
Laboratory for Engineering Databases
Linköping University
Linköping, Sweden
{vanja, timka, torri}@ida.liu.se

Abstract

The mediator-wrapper approach to integrate data from heterogeneous data sources has usually been centralized in the sense that a single mediator system is placed between a number of data sources and the applications. As the number of data sources increases, the centralized mediator architecture becomes a bottleneck. This paper presents an architecture for composable and distributed mediator servers, defined in terms of other mediator servers. The modularity of composable mediators allows to build larger systems of distributed mediators integrating many data sources, without the need to maintain a global schema. Composable mediators furthermore provide data independence by allowing locality of changes in both submediators and data sources. However, a problem with a distributed and composable mediator architecture is that the query performance may degrade as the number of mediators increases. We describe some challenges for processing queries in this type of environment, and propose a distributed query decomposition algorithm that eliminates some of the overhead of logical mediator composition. For certain mediator compositions it produces distributed query plans whose inter-mediator data flow is optimal with respect to the query, but is different from the logical interdependencies between the involved mediators. Experimental results show that this strategy improves the query performance and allows increase of the number of mediators without query performance degradation.

1 Introduction

The wrapper-mediator approach for integration of data from heterogeneous data sources has been used in several projects [13, 25, 11]. This approach divides a data integration system into two functional units. The wrapper provides access to the data in the *data sources* using a *common data model* (CDM), and a common query language. The mediator provides a semantically coherent CDM representation of the combined data from the wrapped data sources, built using reconciliation primitives. Usually the

data sources are distributed to several sites and accessed over some computer network. The mediator provides transparent access to the combined data from the data sources through queries to the mediating view The user/programmer does not need to make individual interfaces to each data source.

Current mediator systems and prototypes [13, 25, 11, 21, 5] are centralized systems where a single mediator server integrates data through a number of wrappers. Although indicated in some system architecture overviews, to the best of our knowledge no system allows many distributed mediator servers to interoperate. An original goal for mediator architectures [26] was that mediators should be relatively simple abstractions of modules of data and that larger systems of mediators should be composed through these primitive mediators. By making mediators servers composable and modular by allowing some mediators to act as wrappers for other mediators, it would be possible to scale the data integration process in the sense that more complex systems of data sources can be integrated than through a central integration. Composable mediators would allow for conceptual modeling of mediators without detailed knowledge of the definitions of other mediators and data sources, through modular design. As for other complex systems, modularity is essential for building large systems of mediators. Furthermore, modularity also increases the data independence between applications, mediators, and data sources by allowing for changes in lower level mediators and data sources without changes in higher level systems. When many mediator servers become available on the computer networks composability will be required for designing new distributed mediator servers in terms of the existing ones.

The design of composable and distributed mediator servers introduces, however, some new challenges to be addressed in this paper. For example, a naive implementation of several levels of mediators as black boxes, as with CORBA technology [24], would often cause significant performance overhead. While on a conceptual level it can make the modeling task easier, such black box treatment of mediators would prohibit extensive query processing over submediators. There is a need to minimize the overhead of the mediator composition hierarchy. CORBA-like technologies furthermore provide only object-instance oriented communication primitives while efficient query execution requires bulk-oriented inter-mediator communication.

We have developed a distributed mediator system, AMOS II, in which federations of mediator servers, acting as virtual object-oriented (OO) databases, can intercommunicate. Each mediator server in the federation has full OO query processing and cost-based optimization capabilities. It exports to other systems interfaces having capabilities for i) exchanging meta-data, ii) processing OO queries, iii) estimating query costs, and iv) bulk-oriented exchange of data. The user can post OO queries to any mediator server and the involved mediators in the federation will interoperate to produce the result as quick as possible.

In such a distributed mediator hierarchy the logical composition of mediators needs not necessarily be the same as the optimal data flow through the network of mediators for answering a query. Often it is favorable to do as much data selection as possible in the data sources and the mediators close to them. If a query needs data from only a single data source it is better to bypass all intermediate mediators which would then do no further filtration. It should furthermore be noted that the performance also depends on the speed of the links between the nodes in the federation and on the computers involved. A good distributed mediator query optimizer should take into account local and shipping costs to produce an optimal query execution plan distributed over the

mediators.

The query optimization task in AMOS II is distributed over the distributed mediator servers. For a given mediator query a distributed query processing algorithm produces a distributed execution plan with optimized data flow that eliminates much of the overhead of composed mediation. Each local AMOS II optimizer knows the local access costs and can ask other mediators and data sources about their access costs. No mediator has total knowledge about all costs.

The query optimizer is thus modular in the sense that it does not work in a central environment where one mediator system has all knowledge needed for query processing. This eliminates the need for a centralized directory of schema and optimization information that might become a bottleneck when the number of mediator servers increase. Instead, in the proposed framework, each local query optimizer exchanges meta-information and costs with the other query optimizers in the federation.

We have done some experiments showing promising results for our distributed mediator query optimization techniques. The experiments show that the distributed query decomposition algorithm can produce better distributed inter-mediator plans than if data is joined through a central mediator. Furthermore, the reported results show that the distributed query decomposition algorithm produces plans that allow for increasing the number of involved servers with minimal increase of the query processing time. The experiments also show that, for a given query, different optimal distributed query execution plans sometimes need to be produced depending on the communication speeds between mediator nodes. For example, a personal mediator may reside in a portable computer and different execution plans are optimal when communicating with the federation over a telephone line than when the computer is connected to the LAN.

The paper is organized as follows. Section 2 introduces the terminology and the features of the AMOS II system. In Section 3 the query decomposition and the distributed compilation are described. Section 4 presents experimental results showing the benefits of the proposed strategies. The conclusions are presented in Section 5.

2 Data Integration with AMOS II

The AMOS II system has its roots in the workstation version of the Iris system, WS-Iris [18]. The core of AMOS II is an open, light-weight, and extensible database management system (DBMS). To achieve better performance, and because most of the data reside in the data repositories, AMOS II is designed as a main-memory DBMS. Nevertheless, it contains all the traditional database facilities, such as a recovery manager, a transaction manager, and a OO query language named AMOSQL [9]. An AMOS II server provides services to applications and to other AMOS II servers.

AMOS II is a distributed mediator system [26] where a number of mediator servers communicate over the Internet. Some of these servers can be configured as *translators* [7] which wrap different kinds of data sources, e.g. ODBC compliant relational databases [2] or XML files. We use the term translator since it is a fully fledged AMOS II system which can wrap more than one data source, contains a complete query processor, and supports semantic abstractions and conversions of the data in the data sources through OO views. A translator is thus also a mediator which provide a virtual OO database server layer that transparently translates data from some data

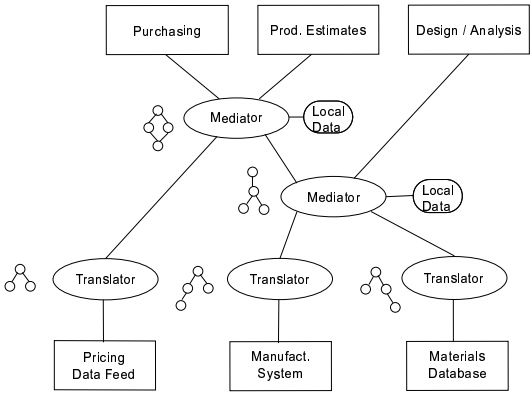


Figure 1. Interconnected AMOS II systems

sources.

Users and applications can pose OO queries to any AMOS II server. We call the server(s) to which some queries are posed *client mediator(s)* for those queries. The other AMOS II servers involved in answering a query are called *server mediators*. For example, in a mobile environment a portable computer could have a client mediator that integrates data from several server mediators on a company LAN. Such a scenario is assumed in our experiments below.

A client mediator can have various types of data sources and access a number of autonomous server mediators. As opposed to a distributed database environment where the data, meta-data and optimization information are available in a centralized repository, in an autonomous environment each server contains only portions of this data.

Figure 1 illustrates the different roles that an AMOS II server can assume. In this example, applications access data stored in data sources through a collection of composed mediator servers. The servers may run on separate workstations and provide data integration, translation, and abstraction services through which different object view hierarchies are presented in the different mediators, as indicated in Figure 1. The mediator servers appear as virtual database servers having data abstractions, query interface, and other database functionality. AMOS II mediators are composable since a mediator server can regard other mediator servers as data sources. A single AMOS II server can also assume more than one role described in Figure 1 and serve more than one application simultaneously. Different interconnecting topologies can be used to connect mediator servers depending on the integration requirements of the environment. Here, a naive implementation where messages are passed between the several layers of composed mediator servers may have unacceptable performance. However, we have developed distributed mediator query optimization techniques that minimize the overhead of composing mediator servers, to be further elaborated in this paper.

The data model in AMOS II is an OO extension of the DAPLEX [22] functional data model. It has three basic constructs: *objects*, *types* (i.e. classes), and *functions*. Objects model entities in the domain of interest. An object can be classified into one

or more types which makes the object *instances* of those types. The set of all instances of a type is called the *extent* of the type. Object properties and their relationships are modeled by functions.

The types are divided into *stored*, *derived*, *proxy*, and *integration union* types, where the instances of *stored* types are explicitly stored locally in AMOS II and created by the user, the instances of *derived* types [14] are derived through a declarative query from the instances of one or more *constituent* supertypes, the instances of *proxy* types represent objects stored in other AMOS II servers or in some of the supported types of data sources, and the instances of *integration union types* (IUTs) [14] are defined as unions of instances representing the same real-world entity in different data sources. Even though the IUTs are outside the scope of this paper, the features presented in the experiments reported in this work are directly connected with the processing of queries over the IUTs, which require outer-join based operations transformed into a set of select-project-join queries [15].

The proxy, derived and integrated union types are the core of the integration framework in AMOS II. Composition of such types provide means for resolving a wide spectrum of semantic heterogeneities between the data and meta-data in the sources. Queries over the OO views are transformed into queries over data in multiple data sources. The OO view mediation framework is described in [14, 15].

The AMOS II functions are divided by their implementations into three groups. The extent of a *stored* function is physically stored in the database. *Derived* functions are implemented in the query language AMOSQL. *Foreign* functions are implemented in some other programming language, e.g. Java, Lisp or C++. Each foreign function can have several associated access paths and, to help the query processor, each access path has an associated cost and selectivity function [18].

The AMOSQL query language is based on the OSQL [19] language with extensions of mediation primitives, multi-directional foreign functions [18], overloading, late binding [8], active rules [23], etc. It contains data modeling as well as querying constructs. The general syntax for AMOSQL queries is:

```
select <result>
   from <type declarations for local variables>
   where <condition>
```

For example, the following query retrieves the names of the parents of all persons having 'sailing' as hobby:

```
select p, name(parent(p))
  from person p
  where hobby(p) = 'sailing'
```

Figure 2 presents an overview of the query processing in AMOS II. The first five steps, also called *query compilation* steps, translate the body of a query expressed in AMOSQL to a query execution plan which is stored with the query.

From the parsed query tree, AMOS II first translates the AMOSQL queries into a type annotated *object calculus* representation [15].

Next, the calculus optimizer applies rewrite rules to reduce the size of the query [15].

After the rewrites, queries operating over data outside the mediator are decomposed into distributed subqueries expressed in an *object algebra*, to be executed in different

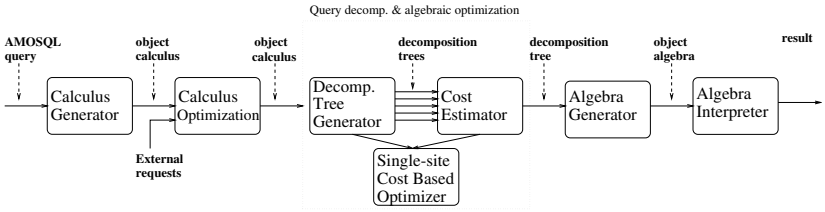


Figure 2. Query processing in AMOS II

AMOS II servers and data sources. The decomposition uses a combination of heuristic and dynamic programming strategies. At each site, a single-site *cost-based optimizer* generates optimized execution plans for the subqueries.

An interested reader is referred to [9] for a more detailed description of the AMOS and AMOS II system and to [18, 7, 8, 14, 15] for more on its query processing.

3 Query Plan Distribution

The distributed mediator framework of AMOS II allows cooperation of a number of distinct mediators on a query processor level. While distribution is present in any mediation framework due to the distribution of the data sources, the distributed mediator server framework introduces a higher level of interaction among the mediator systems. In other words, a client mediator does not treat another AMOS II server as just another data source. More specifically, if we compare the interaction between a centralized mediator system and a wrapped data source and the interaction between two AMOS II servers, there are two major differences:

- An AMOS II server can accept compilation and execution requests for general queries accessing data in more than one source. The wrapper interfaces accept subqueries that are always over data in a single data source.

- AMOS II supports materialization of intermediate results to be used as input to locally executed subqueries, generated by query decomposition in another AMOS II server. A wrapper provides *execute* functionality for queries to the data source. The query execution interface of AMOS II, on the other hand, provides *ship-and-execute* (SAE) functionality, that can first accept and store locally an intermediate result, and then execute a subquery using it as an input.

These two features influence the design of both the query decomposer and the run-time support for query execution. Techniques based on these features to achieve improved query performance are presented in this section. In the remaining of the section, first we overview the basic decomposition algorithm, and then present a method to improve the resulting query execution schedules by taking advantage of the features described above.

3.1 Query Decomposition

The query decomposition phase [14] of the query processing in AMOS II is invoked whenever a query is posed over data from more than one data source. The input of the query decomposition is a query calculus expression operating over imported (proxy) and local stored types. The output is an executable algebra plan. The query decomposition process is divided in 4 phases:

1. Predicate grouping
2. Execution site assignment
3. Execution schedule generation
4. Object algebra generation

The rest of this subsection gives an overview of each of these phases. A more thorough description can be found in [14].

- *Predicate grouping.*

  This phase attempts to reduce the problem of finding a suboptimal execution plan by reducing the number of predicates. Predicates executed at the same data source are grouped into one or more composite predicates that are treated afterwards as single predicates. For each composite predicate, a temporary derived function is defined locally or at another AMOS II server. The following grouping heuristic is used:

  – Joins are pushed to the data sources whenever possible
  – Cross-products are avoided

  Within a composite predicate, the optimization is performed in the AMOS II server where this predicate is forwarded for execution.

- *Site assignment (group placement).*

  This phase uses cost-based heuristics to make the final decision which composite predicate is executed where, eventually replicates some of the predicates, and assigns execution sites to those predicates that can be executed at more than one site (e.g. θ-joins specified by comparison operators). The output of this phase is a query graph where all the nodes are assigned to some site.

- *Cost-based execution scheduling.*

  In order to translate the query graph from the previous phase into an executable query plan, the query processor must decide on the order of execution of the predicates in the graph nodes, and on the direction of data shipping between the nodes.

  Execution schedules for distributed queries in AMOS II are represented by *decomposition trees* (DcTs). Each DcT node describes one data cycle through a client mediator. Fig. 3 illustrates one such cycle. In a cycle, the following steps are performed:

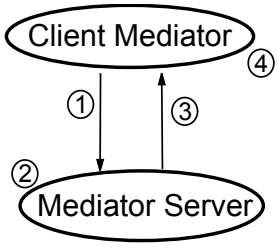


Figure 3. Query processing cycle, described by a decomposition tree node

1. Materialize intermediate results in a source where they are to be processed.
2. Execute a subquery function with the materialized data as input
3. Ship the results back to the client mediator
4. Execute one or more subquery functions defined in the client mediator (post-processing).

Each DcT node stores information about the first three steps in a structure called *ship-and-execute* (SAE) structure. The last step is described by a post-processing structure. The result of a cycle is always materialized in the client mediator. A sequence of cycles can represent an arbitrary execution plan.

As the space of all execution plans is exponential in the number of participating databases, we examine only a subset of the family of left-deep decomposition trees by using dynamic programming and heuristics to prune the search space. The outcome of this phase is an executable left-deep decomposition tree. Being central to our discussion, we elaborate more on decomposition tree generation in the next subsection.

- *Algebra generation.*

  The input to this phase is an executable decomposition tree, which is translated into equivalent sets of inter-calling local object algebra plans.

3.2 Tree Balancing and Distribution

The query decomposition algorithm as presented above, produces an initial *centralized* execution plan. This plan is similar to the execution schedules produced in other distributed and multidatabase systems, e. g. [3, 13, 17], where the query compilation and execution is a centralized process, managed by a coordinator for distributed databases, or by a single client mediator. All inter-site result assembling operations (equi-joins) are performed in the client mediator (coordinator).

This type of plans suffer from heavy involvement of the central client mediator and high network traffic between the client and server mediators. Furthermore these plans might contain redundant operations in which intermediate results are shipped from one server mediator to another, passing through the client mediator.

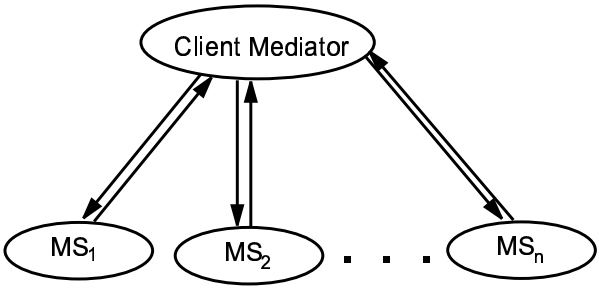


Figure 4. Class of centralized data flow patterns in AMOS II

In order to eliminate these problems, an additional query decomposition phase is introduced after the cost based scheduling, that improves the centralized left-deep execution schedule produced by the cost based scheduling. It uses a distributed compilation process to translate the input schedule into a plan distributed over all the participating servers which communicate the intermediate results directly to each other.

As noted above, in the execution schedules represented by the initial left-deep decomposition trees of the centralized plans, all data shipped between any of the server mediators, always passes through the client mediator. A graphical representation of the data flow patterns generated by these plans is shown in Fig. 4.

The composition of mediators is designed using semantic considerations, as opposed to trying to distribute load or other performance considerations. It may introduce considerable performance problems, mostly due to transmission costs between (possibly) many layers of mediators. In many cases, it might be cheaper if different server mediators can exchange data directly, independently from the client mediator. This is even more true in the cases of non-homogeneous execution environments, for example when the client mediator accesses the server mediators over a slow communication channel.

In the rest of this section, we present a novel distributed query decomposition technique, named *Tree Distribution*, for tree balancing in an environment of distributed, autonomous mediator systems. It extends the query decomposition process in AMOS II in order to explore the richer space of execution plans allowing direct communication among the server mediators involved in a query. It distributes not only the execution, but also the decomposition of the query plans among different servers. Experimental results show substantial performance improvement over the plans before the tree distributions.

The class of plans in Fig. 4 are transformed, when favorable, into plans using direct communication between different server mediators participating in the query without passing data through the client mediator. The algorithm uses random hill-climbing with a complexity that is linear in the size of the decomposition tree. Although this approach does not enumerate all the possible plans, in our experience it suffices for the typical mix of queries posed to a client mediator. The gains are especially apparent when the client mediator is hosted on a computer connected to the server mediators via a slow line.

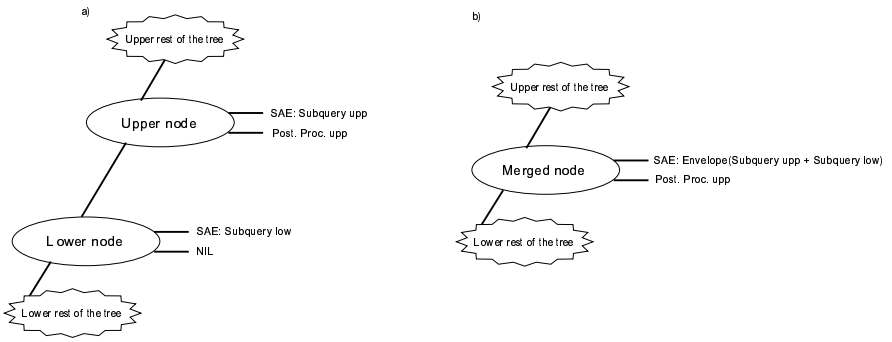


Figure 5. Decomposition tree node merger operation a) before the merge b) after the merge

3.3 Tree Distribution

Each node of the left-deep tree generated by the cost-based decomposition phase describes one processing cycle of Fig. 3 above. The data flow presented in Fig. 4 shows that the centralized left-deep plans generate data flow composed of several individual cycles between the client and server mediators (SM). Note that, in presence of OO server mediators, this strategy is more general than the strategy used in some other multidatabase systems (e.g. [20, 12, 17]) where the joins are performed in the client mediator system over data retrieved from the participating wrapped data sources. The latter does not allow for mediation of OO sources that access not only stored data, but also contain programs executed in the data source (e.g. image analysis, matrix operations). In such cases, it is impossible to retrieve the program logic from the source and therefore it is necessary to ship intermediate results to the source in order to execute the programs using the shipped data as input. From this aspect, the strategy is similar, but more efficient than the *bind-join* strategy in [13] since we use bulk shipping rather than instance (tuple) shipping.

The main idea of the tree distribution algorithm is to transform the centralized tree by a series of *node merge* operations. A node merge aims to eliminate the data flow through the client mediator and is applied over two consecutive decomposition tree nodes (a lower and an upper node) such that the lower node does not specify post-processing operations in the client mediator (step 4 in Fig. 3). The absence of the post-processing operations in the lower node means that the data is streamed unchanged from the server mediator participating in the lower node cycle (e.g SM_0), through the client mediator, to the server mediator participation in the upper node cycle (e.g. SM_1).

The node merge operation produces a new node that substitutes the two merged node in the DcT, as shown in Fig. 5. The new node has the same post-processing operations as the upper node. The ship-and-execute (SAE) operations of the new node are performed by a *envelope function* defined and compiled at one of the two mediator servers participating in the processing cycles described by the merged nodes. The body (predicate) of the envelope function is made by conjuncting the bodies of the functions specified in the SAE structures of the merged nodes. When the envelope

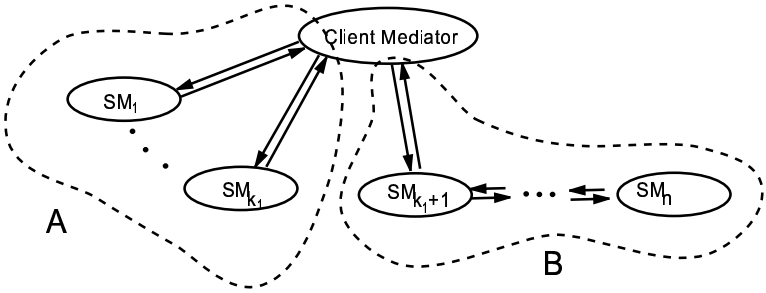
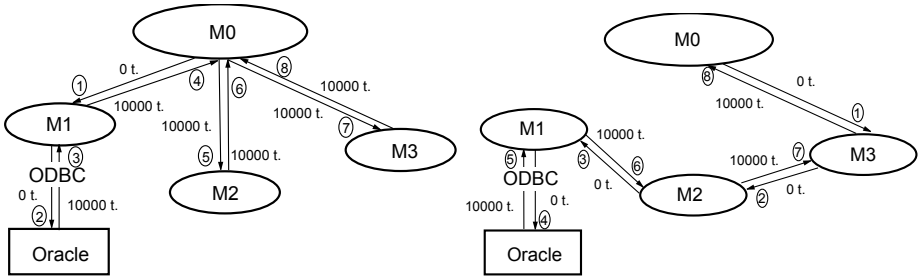


Figure 6. Class of distributed data flow patterns in AMOS II

function is compiled at one of the server mediators participating in the query, the generated execution plan contains a processing cycle that ships data *sideways* between the two server mediators, eliminating the involvement of the client mediator. The merged node describes a cycle where the envelope function is invoked in one of the server mediators, and the result of its invocation is shipped back to the client mediator. Note that each envelope function is a derived function (view) over data in more than one server mediator, and therefore the distributed query compiler generates a new decomposition tree for it at the server where it is compiled. After repeated recursive application of node merge operations the query execution plan is described by a set of decomposition trees stored in both the client mediator and the participating server mediators. Since these trees are generated by compilation at the server mediators, the client mediator does not need any optimization information used in the compilation of the envelope functions.

The the merge operation is applied at a random qualifying point in the tree. If the new tree has lower execution time than the original, then it is used instead of the original. The process continues until no beneficial merge operations are performed. The maximum number of merges during this process is $2(n-1)$, where n is the number of nodes in the input decomposition tree. In its final variant the input tree might become distributed between $n-2$ server mediators and the client mediator.

The family of execution plans, generated by this algorithm have the general data flow pattern of Fig. 6, where all communicating servers can be classified into two types of groups - groups containing servers which exchange data only with the client mediator (A in Fig. 6) and groups of servers that communicate directly with each other in a sequential manner (group B in Fig. 6). In the plans generated by the transformations described above, groups of the both type are interchanged. The introduction of type B groups into the plans, achieved by the proposed transformation, allows for better performance in a common case when a network of server mediators, connected by fast connections is accessed by a remote client mediator through a slow (modem or mobile) line.



(a) Centralized plan (b) Distributed plan

Figure 7. Dataflow graphs for *Q1*, 10000 tuples of size 100 bytes

4 Experimental Evaluation

This section presents the results from a comparison of the centralized plan with the distributed plan produced by the Tree Distribution algorithm for a class of queries to a client mediator.

4.1 Experimental environment

The performance experiments were made in two different environments. In both cases we had up to four server mediator, named M1, M2, M3, and M4, respectively, running on the same computer, and a client mediator, named M0, executing as a client on a remote computer, to which the queries were posed. All data was stored in an Oracle 8 relational data source, and server mediator M1 was acting as a translator through an ODBC wrapper. The Oracle server was running on the same computer as the four server mediators. The choice to run the server mediators M1-M4 on the same computer was made in order to simplify the experimental setting. We use local TCP/IP communication also between servers running on the same Windows NT workstation, and we measured that the local TCP/IP performance is actually about 10-15% slower than inter-computer TCP/IP communication over our 100 Mbit LAN. Furthermore, in our experiments we do not explore asynchronous server intercommunication which makes only one system run at the time. Thus this setting is roughly equivalent to the case when the translators are running on different computers on the LAN. It was ensured that all participating AMOS II systems fit into main memory, and no swapping occurred during the experiments.

The hardware used during the experiments was Compaq Professional Workstations with 200 MHz Pentium Pro CPUs, 128 MB RAM, and a 100 Mbit LAN card, running Windows NT Workstation 4.0.

In the first set of experiments the client mediator was connected to the server mediators through the LAN, while the second set of experiments were performed with the client mediator running on a remote NT workstation connected to our LAN over a 128 Kbit ISDN line. This corresponds to the situation where the client mediator resides in a portable computer which is sometimes connected over a LAN and sometimes remotely over a slower connection.

4.2 Queries and query plans

For the experiments we used a synthetic database with tables having varying number of tuples. Two types of precompiled queries were used in the experiments. In the examples we will use AMOSQL syntax. The first query was used to measure scale-up properties of the tree distribution algorithm as the mediator query spans more servers. In order to do this, we compiled three similar queries, such that the first one - *Q1'* involved two server mediators (M1,M2), the next one *Q1"* involved three server mediators (M1,M2,M3), and *Q"'* was executed over all four servers M1-M4. In each mediator *Mi* a function

$$process@Mi(charstring\ str, integer\ sel) \rightarrow charstring$$

was defined, which was simulating processing of data in *Mi* by selecting $sel\%$ of it's incoming data. For the discussed experiments we choose 100% selectivity ($sel = 100$) for all $process@Mi$ functions. In order to extend a query to involve server *Mi*, a call to the corresponding function $process@Mi$ was added to the query. Server *M1* wrapped the relational data source, and the *DATA* column of a relational table *EMPLOYEE* was accessed by the foreign AMOSQL function $data(emp) \rightarrow charstring$. As an example of the three queries, we show *Q1"*, which involves the client mediator *M0*, and three other server mediators, M1,M2,M3:

```
select s3
from string d,
     string s1,
     string s2,
     string s3,
     employee@M1 e
where d = data(e) and
      s1 = process@M1(d, 100) and
      s2 = process@M2(s1, 100) and
      s3 = process@M3(s2, 100);
```

Each of the three queries was compiled once using only the centralized query decomposition technique, and once using also the tree distribution algorithm. Correspondingly, different execution plans were produced by the different decomposition techniques. In the example case of Q1", the centralized decomposition produced a tree-like data flow graph, shown in Fig. 7a, while the distributed algorithm generated the L-shaped data flow graph, shown in Fig. 7b. Each directed arc of the data flow graphs is marked by the number of tuples sent in the corresponding direction. The numbers in the ovals show the order of execution. Considering that each tuple has size of 100 bytes, the total amount of data sent over the network in case a) is 50000, while in case b) it is only 30000. Even this simple consideration gives us a hint that case b) will be considerably more beneficial than case a).

The simplified cost model for evaluating data flow patterns between databases presented here gives us some insight of the possible benefits of the tree distribution algorithm. Later on, in Sec. 4.3 we present experimental confirmations of our expectations.

The second group of experiments used query *Q2*. The major difference between the group of queries *Q1* and query *Q2* is, that in *Q2* we introduced a function defined

in the client mediator $process@M0(charstring\ str, integer\ sel) \rightarrow charstring$, which restricts data retrieved from the relational data source. In this case we chose 10% selectivity for the $process@M0$ call:

```
select s2
from string d, string s1, string s2, string m,
     employee@M0 e
where d = data(e) and
      s1 = process@M1(d, 100) and
      m  = process@M0(s1, 10) and
      s2 = process@M2(m, 100);
```

4.3 Experimental results

This subsection presents the results from measurements of execution times for queries *Q1* and *Q2* in the two environments described in section 4.1. During the measurements, each query was executed four times, and the average of the last three measurements was taken.

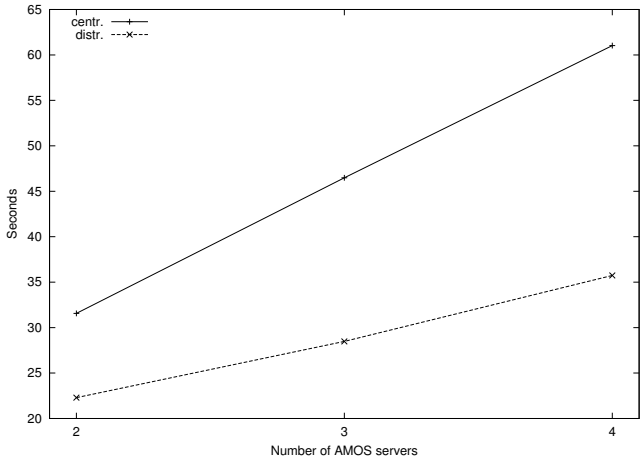
Figure 8a compares the performance of the centralized execution plans for queries Q1', Q1", and Q1"' with the corresponding distributed plans produced by the tree distribution algorithm. As expected the distributed plans generated by the tree distribution algorithm are significantly faster than the centralized plans. In particular, the performance improvements for Q1', Q1", and Q1"' are 33%, 47%, and 49%, respectively. Thus, as the queries span more servers the performance improvement of the tree distribution algorithm increases. In this case the tree distribution algorithm produces plans that scale better since they send less data between the servers and the client mediator M0 than the centralized plans.

In Fig. 8a the connection between M0 and the other servers uses a fast LAN. If a slower connection is used the performance gains will be larger, as shown in Fig. 8b where M0 is connected to the server mediators through a slower ISDN connection. The performance gains are here 50%, 80%, and 86%, respectively. The execution time is virtually constant, independent on the size of the query, since the amount of data shipped between the client and server mediators is constant with the distributed plan.

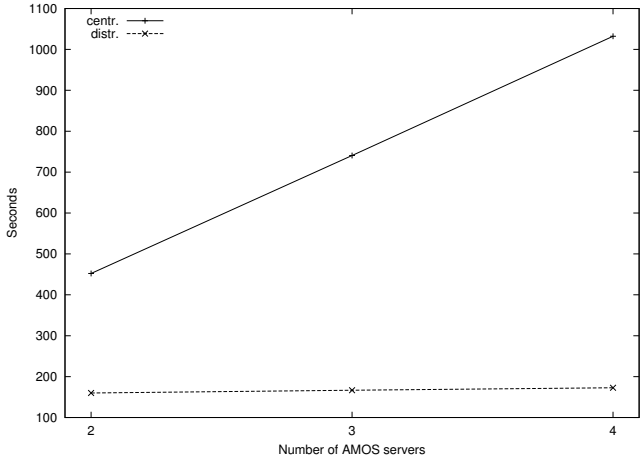
The latter measurements correspond to a mobile client mediator connected to the server mediators, while the former measurements could be when the same client mediator is docked directly to the company LAN. In these examples the rebalanced plans are always better, since all selections are in the server mediators M1-M4 and there are no selections in the client mediator M0. If there were selections in M0, as in Q2, it could be possible that it would be favorable to use the centralized plan, since the selection in M2 could restrict the number of shipped tuples. To evaluate this we made some tests with query Q2, having a selection in M0.

Fig. 9a compares the centralized plan produced for Q2 with the corresponding distributed plan. As expected, it shows that the centralized plan is better in this case.

In Fig. 9b the query Q2 is tested with an ISDN connection to M0. Because of the slower connection it is here more favorable to use a distributed plan, since the cost to ship data to M0 is higher than the costs of shipping many more tuples between the server mediators. However, in this case our tree distribution algorithm would produce



(a) LAN



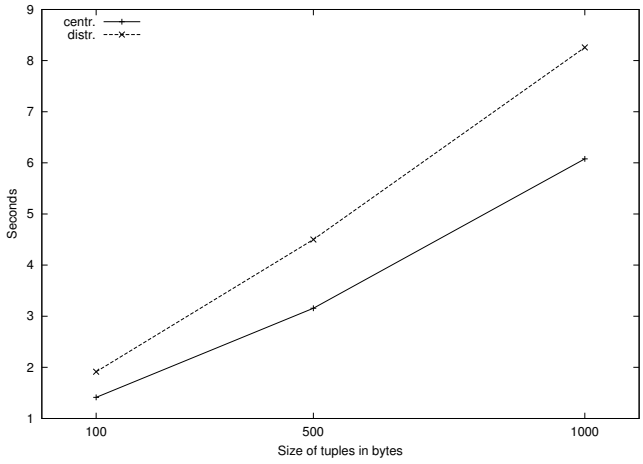
(b) ISDN

Figure 8. Execution times for *Q1"*, 10000 tuples of size 100 bytes

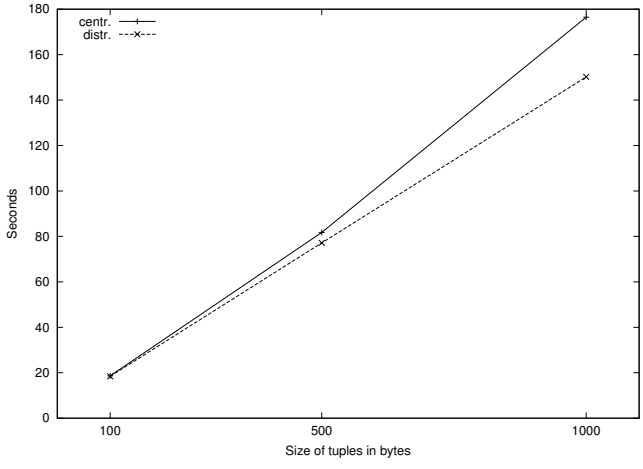
the suboptimal centralized plan, because no node merge would take place. We are investigating how the tree distribution algorithm can be generalized to handle this case too. Fig. 9a and 9b also illustrates that there are cases where different strategies are needed depending on the speed of the connection to M0. In a dynamic (e.g. mobile) mediator environment this would have to be taken into consideration. The system could here generate two different distributed plans and use one or the other depending on if the mobile client mediator is connected via LAN or ISDN. This is an area for further research.

5 Related Work

The research presented in this paper is related to the areas of data integration and distributed databases. This section references and briefly overviews some representa-



(a) LAN



(b) ISDN

Figure 9. Execution times for *Q2*, 1000 tuples, 2 servers

tive examples of projects in these areas, close to the work presented in this paper. A more elaborate comparison of the AMOS II system with other data integration systems is presented in [16].

One of the first attempts to tackle the query optimization problem in distributed databases was done within the System R* project [3]. In that project an exhaustive, centrally performed query optimization is made to find the optimal plan. Because of the problem size, AMOS II searches only a portion of the whole search space by an exhaustive search strategy. Other phases use heuristics to improve the plan and reduce the optimization time. The SDD-1 system [10] also uses a hill-climbing heuristics as in AMOS II to schedule "moves of relations" and "local processing actions" in that compose the distributed query execution schedule. Another classical work on query optimization in a distributed database environment is presented in [1]. In this approach, named AHY (Apers-Hevner-Yao), the system performs first local

processing over the relations, then it reduces the results by semi-joins, and finally composes the result at a central site named *evaluation site*. This is clearly different from AMOS II where joins are performed in different servers. All three approaches perform the query compilation in a single site, as opposed to the distributed query compilation in AMOS II.

As opposed to the distributed databases, where there is a centralized repository containing meta-data about the whole system, the architecture described in this papers consists of autonomous systems, each storing only locally relevant meta-data. Most of the mediator frameworks reported in the literature (e.g. [13, 25, 11]) propose centralized query compilation and execution coordination. In [5] it is indicated that a distributed mediation framework is a promising research direction, but to the extent of our knowledge no results in this area are reported. Within the same project a centralized query tree rebalancing is proposed [4].

In the DIOM project [21], the importance of the mediator composability is also recognized. A framework for integration of relational data sources is presented where the operations can be executed either in the mediator or in the data source. The query optimization strategy used first builds a join operator query tree (schedule) using a heuristic approach, and then assigns execution sites to the join operators using an exhaustive cost-based search. AMOS II , on the other hand, performs a cost-based scheduling and heuristic placement. Furthermore, the compilation process in DIOM is centrally performed, and there is no clear distinction between the data sources and the mediators in the optimization framework.

6 Summary and Future Work

We have given an overview of the architecture of the AMOS II mediator system where federations of distributed mediator servers can be composed by AMOS II servers. Each AMOS II server has DBMS facilities for query compilation, and exchange of data and meta-data with other AMOS II servers. OO views can be defined where data from several other mediator servers are abstracted, transformed, and reconciled.

The importance was reiterated of being able to logically compose systems of mediators without global meta-data knowledge in order to build large data integration systems.

It was shown how to decrease the overhead of logically composing mediator servers by a distributed query optimization technique called *Tree Distribution*. Here, a distributed compilation algorithm generates distributed execution plans where the optimized data flow is different from the logical mediator composition, and where each participating mediator interacts only with its neighbor mediator servers.

Performance measurements show that the Tree Distribution algorithm significantly improves query performance and also allows for scale-up with respect to the number of mediator servers involved in a query.

The performance improvements are particularly large in an environment with low bandwidth connections between a client mediator and a set of composed server mediators, as e.g. in a mobile environment where a portable computer is connected via ISDN or a regular phone line to mediator servers communicating via LAN.

We showed that if there are selections in the client mediator different query distri-

butions are optimal depending on the speed of the connection and the selectivity of the selections. Further research is ongoing to handle the larger class of distributed and multiple query plans required in this situation.

More work is also needed to deal with parallel execution plans, unreliable and sometimes disconnected connections from the client mediator, and deep mediator compositions.

References

[1] P. Apers, A. Hevner and S. Yao: Optimization Algorithms for Distributed Queries. *IEEE-TSE*, SE-9:1, 1983

[2] Silvio Brandani: Multi-database Access from Amos II using ODBC. In *Linköping Electronic Press*, Vol. 3, Nr. 19, Dec. 8th, 1998, *http://www.ep.liu.se/ea/cis/1998/019/*.

[3] D. Daniels et al.: An Introduction to Distributed Query Compilation in R*. In H. Schneider (ed) *Distribute Data Bases*, North-Holland, 1982

[4] W. Du, R. Krishnamurthy and M-C. Shan: Query Optimization in Heterogeneous DBMS. *18th Conf. on Very Large Databases (VLDB'92)*, Vancouver, Canada, 1992

[5] W. Du and M. Shan: Query Processing in Pegasus, *Object-Oriented Multi-database Systems*, O. Bukhres, A. Elmagarmid (eds.), Pretince Hall, Englewood Cliffs, NJ, 1996.

[6] G. Fahl, T. Risch, M. Sköld: AMOS - An Architecture for Active Mediators. *Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, Haifa, Israel, June 1993.

[7] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4), pp 261-281, November 1997.

[8] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions, *21st Conf. on Very Large Databases (VLDB'95)*, Zurich, Switzerland, 1995

[9] S. Flodin, V. Josifovski, T. Risch, M. Sköld and M. Werner: AMOS II User's Guide, available at *http://www.ida.liu.se/∼edslab*.

[10] N. Goodman, P. Bernstein, E. Wong, C. Reeve and J. Rothnie: Query Processing in SDD-1: A System for Distributed Databases. *ACM-TODS* 6:4, 1981

[11] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y.Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)* Vol 8 No. 2 117-132, Kluwer Academic Pulishers, The Netherlands,1997

[12] B. Finance, V. Smahi J. Fessy: Query Processing in IRO-DB, *Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)* pp.299-319, 1995

[13] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries accross Diverse Data Sources. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 276-285, Athens Greece, 1997

[14] V. Josifovski: Design, Implementation and Evaluation of a a Distributed Mediator System for Data Integration: the Story of AMOS II, Ph D Thesis, University of Linköping, Linköping, Sweden, June 1999

[15] V.Josifovski and T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration *Journal of Intelligent Information Systems (JIIS)* Vol 12 No. 2/3, Kluwer Academic Pulishers, The Netherlands, 1999.

[16] V.Josifovski and T.Risch: Comparison of AMOS II with Other Data Integration Projects. Available at http://www.ida.liu.se/ edslab/amosII_comp.pdf

[17] E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.

[18] W. Litwin and T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering* 4(6), pp. 517-528, 1992

[19] P. Lyngbaek et al: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[20] S. Nural, P. Koksal, F. Ozcan, A. Dogac: Query Decomposition and Processing in Multidatabase Systems. *OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis*, Montpellier, July 1996.

[21] K. Richine: Distributed Query Scheduling in DIOM. Tech. Report TR97-03, Computer Science Department, University of Alberta, 1997

[22] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), ACM Press, 1981.

[23] M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th International Conf. on Data Engineering (ICDE'96)*, (IEEE), New Orleans, Louisiana, Feb. 1996.

[24] R. Soley, C. Stone (eds.): Object Management Architecture. *John Wiley & Sons*, New York, 1995

[25] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *Transactions on Knowledge and Data Engineering* (TKDE) vol. 10 No. 5, pp 808-823, 1998

[26] G Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), Mar. 1992.

Paper E:

Vanja Josifovski, Timour Katchaounov, and Tore Risch. Evaluation of join strategies for distributed mediation. In *5th East European Conference on Advances in Databases and Information Systems, ADBIS 2001*, volume 2151 of *Lecture Notes in Computer Science*, pages 308–322, Springer-Verlag, September 2001.

Evaluation of Join Strategies for Distributed Mediation

Vanja Josifovski*, Timour Katchaounov and Tore Risch
Uppsala Database Laboratory, Uppsala University, Sweden,
vanja@us.ibm.com, timour.katchaounov@dis.uu.se, tore.risch@dis.uu.se

Abstract

Three join algorithms are evaluated in an environment with distributed main-memory based mediators and data sources. A streamed ship-out join ships bulks of tuples to a mediator near a data source, followed by post-processing in the client. An extended streamed semi-join in addition builds a main-memory hash index in the client mediator. A ship-in algorithm materializes and joins the data in the client mediator. The first two algorithms are suitable for sources that require parameters to execute a query, as web search engines and computational software, and the last is suitable otherwise. We compare the execution times for obtaining all and the first N tuples, and analyze the percentage time spent in subsystems, varying the network communication speed, bulk size, and data duplicates. The join algorithm leads to orders of magnitude performance difference in different mediation environments.

1 Introduction

Integration of data from sources with varying capabilities has been intensively studied by the database community in the recent decade. The Amos II system [8, 9, 17] uses the *wrapper-mediator* paradigm to integrate data from several sources. One of the salient features of Amos II is a distributed architecture where a number of interconnected *mediator servers* cooperate in providing the users and the applications with the required view of the data in the sources. We believe that a distributed mediator architecture is needed because it is unrealistic to assume that a single mediator server can be deployed in an enterprise composed of multiple organizational units. When many mediator servers become available on the network, composability will be required for designing new distributed mediator servers in terms of the existing ones, thus reusing mediation specifications. Multiple mediators will also alleviate the performance bottleneck problems that appear when all the queries are handled by a single mediator.

Having some of the basic assumptions different from the classical database systems, query processing in a distributed mediator system requires some novel

*Current address: IBM Almaden Research Center, San Jose, CA 95120, USA

strategies and solutions. One of the major reason for this is the different cost model in this environment. The I/O and CPU costs used in the traditional query optimization [14] are largely insignificant here compared to the cost of accessing data in external sources. While new cost models have been developed for use in mediator frameworks with centralized architecture [18], no experimental results are reported using a distributed mediator framework. In this work we quantify empirically the relations among the different costs in a wrapper-mediator environment, as for example, the network cost and the data source access costs.

Traditional data integration systems [11, 16] send all data to the mediator for joining. Such 'ship-in' methods do not allow for integration of 'non-database' data sources that require some input, since it is not possible to ship the programming logic from these systems into the mediator. Also they are not good for top-N queries where only a first few tuples are retrieved.

Three join algorithms for a distributed mediation environment are presented and analyzed. An outer collection, generated as an intermediate result of a previous computation, is joined with an inner collection produced from a data source. Two *ship-out* algorithms ship data *toward* the sources. In these algorithms, intermediate result tuples are shipped to the sources where they are used as parameters to precompiled query fragments (subqueries or function calls) of the original query. The first algorithm is an order-preserving semi-join which is suitable when there are no duplicates in the outer collection. The second algorithm uses a temporary hash index of possibly limited size to reduce the number of accesses to the data sources. It is suitable when there are duplicates in the outer collection. Both ship-out algorithms are streamed [6] and the data is shipped between the mediator servers in bulks that contain several tuples to avoid the message set-up overhead. Finally, for comparison, a ship-in algorithm is analyzed, which is suitable when the sources cannot accept parameterized queries and when the data retrieved from the sources is small enough to be stored in a temporary main-memory index in the mediator.

The algorithms are evaluated in an environment with an ODBC data source and a mediator server running on Windows NT platforms, connected by ISDN and LAN. Substantial performance gains were measured (up to factor 100) when using our framework over an ISDN connection to access a relational database server, as compared to accessing the relational database with ODBC directly from the client, since bulk oriented join processing between the mediators minimizes ISDN message traffic and eliminates all expensive remote ODBC calls.

2 Background

As a platform for the work in this paper we use the Amos II mediator database system [8, 9, 17]. The core of Amos II is an open light-weight and extensible DBMS. It is a distributed mediator system where both the mediators and wrappers are fully functional Amos II servers, communicating over the Internet. For good performance, and since most the data reside in the data sources, each Amos II server is designed as a main-memory DBMS.

Some of the Amos II servers can be configured to wrap different kinds of data sources, e.g. ODBC compliant relational databases [4] or XML files [12]. Other servers reconcile conflicts and overlaps between similar real-world entities modeled differently in different data sources, using the *mediation primitives* [8, 9, 17] of the query language AmosQL .

Users and applications can pose OO queries to any Amos II server. We call the server(s) to which application queries are posed *client mediator(s)* for those queries. The other Amos II servers involved in answering a query are called *mediator servers*. The mediator servers may run on separate workstations and provide data integration, wrapping, and abstraction services through which different views are presented in different mediators. For example, in a mobile environment a portable computer could have a client mediator that integrates data represented by several mediator servers on a company LAN. A mediator server can have different types of data sources attached and access a number of other mediator servers.

The AmosQL query below contains a join and selection over the table A at the source $DB1$, and B at $DB2$, based on values of functions fa and fb:

```
select res(b)
from A@DB1 a, B@DB2 b
where fa(a) = fb(b);
```

The query is issued in a client mediator over data that can be either directly stored in *DB1* and *DB2* or, if these are Amos II servers, retrieved from wrapped data sources. Strategies to execute this equi-join will be the focus of this paper.

The queries are rewritten by the optimizer to eliminate redundant computations. After the rewrites, queries operating over data outside the mediator are decomposed into distributed *query fragments*, executed in different Amos II servers and data sources. The decomposition uses heuristic and dynamic programming strategies in three stages [10]: query fragment generation, fragment placement and fragment scheduling. Each Amos II server uses a single-site *cost-based optimizer* to generate optimized execution plans for the query fragments. The fragments for other types of data sources are handled by the mediator if the source has no query processing capabilities, or by the source otherwise.

3 Algorithm Descriptions

While a naive data source interface provides only *execute* functionality for queries, Amos II also provides *bulked ship-out and execute* functionality where a remote Amos II server accepts and store tuples locally in main-memory, and then executes a query fragment using them as an input. When joining directly to a data source, the communication is directly with it and the processing is one tuple at the time for the ship-out algorithms, assuming that storing bulks of the intermediate results is not possible in data sources because of their autonomy.

3.1 Ship-out Join Algorithms

In general, the ship-out algorithms can be described with the following steps:

1. preprocess and prepare the input collection for shipping
2. ship the input collection to a remote site
3. execute the query fragment over the collection at the remote site
4. return result of query fragment execution to the coordinating mediator
5. assemble the result collection to be emitted from the join

Steps 1, 4 and 5 are executed locally, while 2 and 3 are performed at another Amos II server by its join request handler.

The input collection is a table where some columns are used as parameters to the remote query fragment; other columns are passed through to the later post-processing in the mediator, or are assembled as parts of the query result.

A straight-forward implementation of a ship-out equi-join operator would ship the whole input bulk to the remote site, execute the remote query fragment on the bulk appending its result to the input, and then ship this result back. The first improvement of the naive strategy we propose is the *project-concat algorithm* (PCA) in Figure 1[1]. It improves the naive strategy by the following two data transformations based on the semi-join algorithm [2]:

- The input bulk is projected over the data columns that are actually used in the remote query fragment, before shipping them there.
- After the query fragment is executed the result shipped back to the mediator contains only the relevant columns from the query fragment result.

The difference between PCA and the classical semi-join is in the use of order for matching the tuples from the joined collections.

The result of the join is assembled by a simple concatenation of the input and the result shipped back from the remote Amos II mediator or data source. Since the operations are order preserving, concatenation can be used instead of a more expensive join.

Table 3.1 illustrates an execution of PCA between the results of query fragments $QF1$ executed at $DB1$ and $QF2$ executed at $DB2$. The input is a collection of tuples with columns va and r produced by the execution of the fragment $QF2$, and a collection of tuples produced by the execution of $QF1$ containing va values and keys of table tB. The fragments are joined over va and the result is represented by column r. Since there are no result columns that are shipped back from DB1 to DB2, a boolean value is used to identify if the tuples produced by $QF1$ have a matching va value in the tuples produced by $QF2$. We assume that the fragment at $DB1$ produces the following table:

va	
tB	**va**
ib_1	4
ib_2	5
ib_1	6

[1]Amos II is object-oriented and steps 2, 3, 5, and 6 handle object identifier (OID) conversions, which are not further elaborated here.

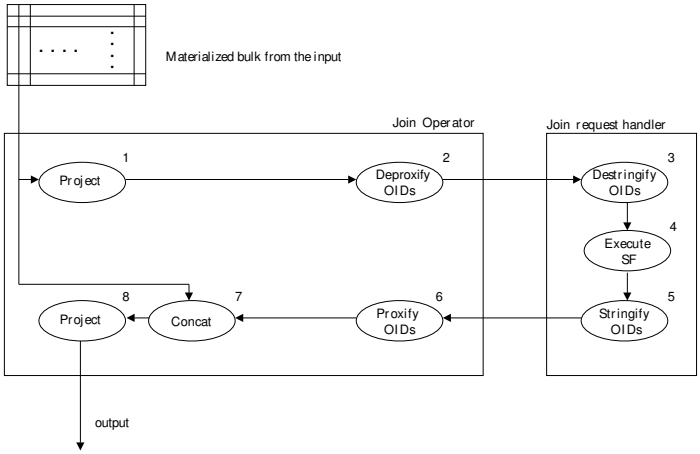


Figure 1: Project-concat ship-out algorithm

where ib_k denotes a key of tB. The example illustrates the execution over 2 bulks of size 4, named in the example as $b1$ and $b2$. In the example, first the projection strips the r values from the input bulks since they are not used in the join. Next, the bulks are shipped to $DB1$ where the query fragment $QF1$ is executed. The resulting set of boolean values is shipped back to the mediator. The concatenation shown in the example is a special case where the executed function does not return any data used later in the query processing. In this case, the concatenation of the returned boolean values and the input tuples actually filters the tuples for which the result is *true*. The final projection removes the *va* values to form the requested result.

The PCA has the advantage of improving the naive implementation, while preserving the simplicity of the processing. All operations have constant complexity per data item and therefore cheap to perform. Nevertheless, it is inefficient when there is a large percentage of duplicates in the input bulk(s), an expensive query fragment, and/or expensive communication between the servers involved.

The traditional *semi-join* algorithm (SJA) [2] improves the performance of the PCA when duplicates are involved. After projecting the input bulk over the columns used as input to the remote query fragment, SJA performs duplicate removal before shipping the data. When there is a large percentage of duplicates within the bulks, this reduces both the size of the shipped data and the number of executions of the remote query fragment. The result of the query fragment execution is shipped back to the calling server where, as in the previous algorithm, the shipped tuples are concatenated to the result of the query fragment invocation. Next, an equi-join is performed over the input bulk and the result of the concatenation. Here, because of the duplicate removal it is not possible to match the tuples by their rank in the bulk.

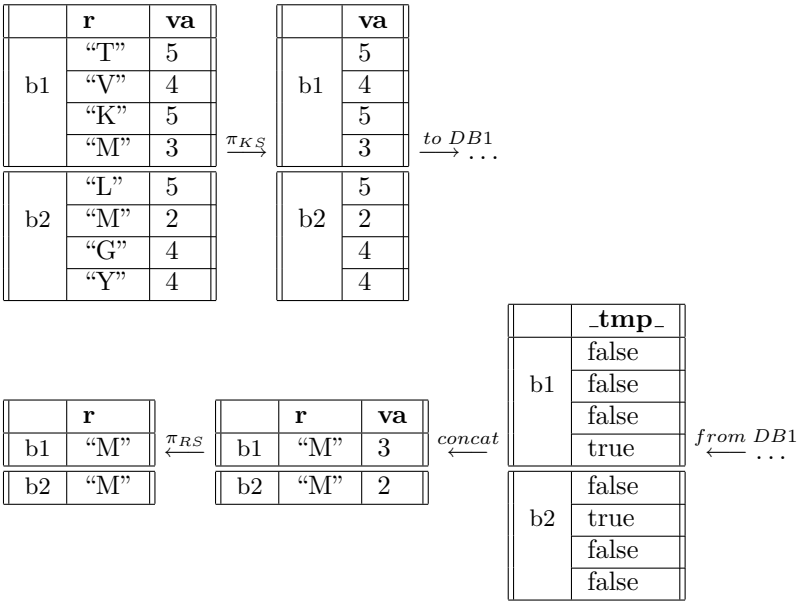


Table 1: Example execution of equi-join using the project-concat algorithm

The SJA benefits from avoiding shipping duplicate entries over the network and executing the query fragment for them, but only for duplicates within a single bulk and with the added costs of the two additional phases of duplicate removal and equi-join.

To avoid duplicates over different bulks, the algorithm in Figure 2, SJMA (semi-join with materialized index algorithm) extends SJA by saving the index built up for the bulks of the outer collection between executions for different bulks. The shipped data is passed through an additional anti-join over the set already pruned from duplicates and the temporary index. If a tuple is in the index, it has already been processed in some of the previous bulks. The remaining tuples are shipped to the remote site for query fragment execution as before. Next, new entries are added to the index from the returned result. Finally, a join between the input bulk and the index is performed as in the SJA. A comparative execution of SJMA in the same scenario as for the PCA example is presented in Table 2. Here, the second bulk is reduced to one tuple before shipping to $DB1$, since the anti-join eliminates the two tuples present in the first bulk.

The size of the index in SJMA is proportional to the number of distinct tuples in the outer collection. The algorithm can be used as a filter even in the case when the whole index is too big to fit in the memory. When the memory limit is reached, new entries replace old entries using some replacement criteria.

SJMA does not add substantially to the cost of the SJA, while it offers the possibility for performance improvements. In fact, it reduces to the SJA in the case when the whole input is contained in only one bulk.

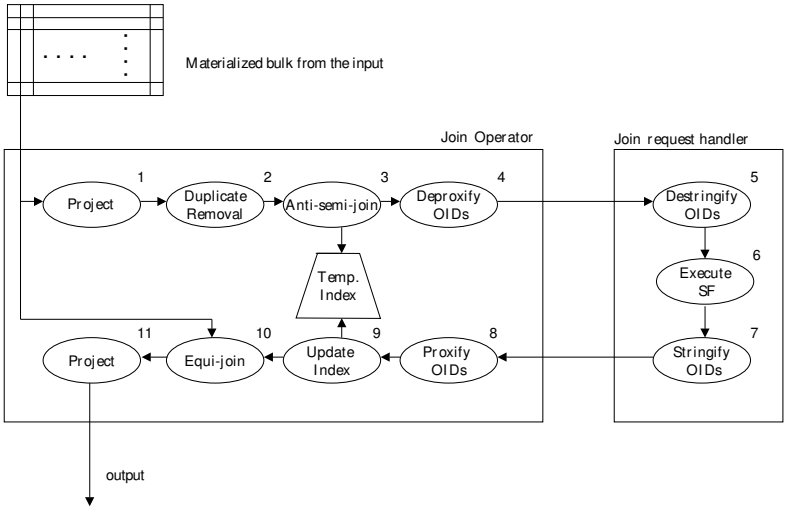


Figure 2: Streamed semi-join with a temporary index

3.2 Ship-in Join Method

Unlike the previous two algorithms where the remote query fragment is executed using parameters from the tuples of the intermediate result, with the ship-in join method no intermediate result is shipped to the remote site. Consequently, the query fragment is executed without parameters. This has two effects:

- Since the remote query fragment is executed once only, it may reduce the number of accesses to the data source.

- The result size may increase since instead of a semi-join of the query fragment result and the intermediate result, the whole query fragment result is sent to the client to be joined there.

While the reduction of the data source accesses may improve the performance, the increased volume of the data shipped and stored in the mediator are the possible performance disadvantages of this algorithm. The algorithm is inapplicable when the query fragment result is too big for the mediator resources. This is also the case when the query fragment contains predicates representing methods/programs in the data source that require parameters to be supplied from the mediator. With the ship-out method, when there are sufficient resources, the materialized index can persist between the execution of the algorithm for different bulks, reducing further the query processing time. This case corresponds to hash join algorithms where an index is built for the inner relation.

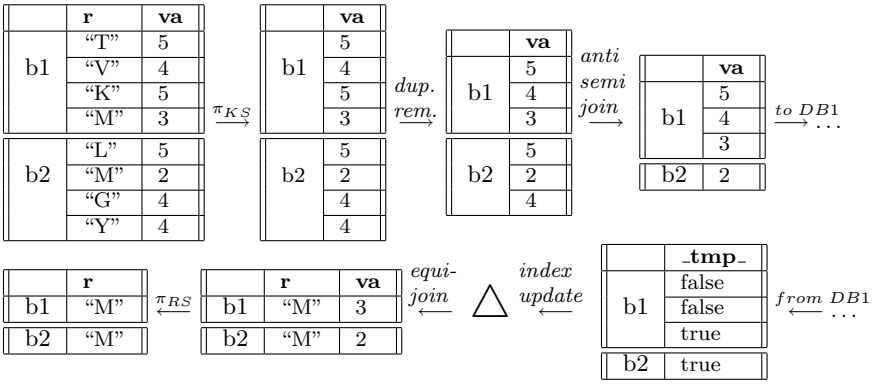


Table 2: Example execution of the semi-join with materialized index algorithm

4 Performance Measurements

In the two scenarios used in the experiments the data source was an ODBC data source. We performed experiments using both Microsoft Access ODBC and IBM DB2 ODBC drivers with no significant differences in conclusions. Where not specifically indicated, the measurements use the Access ODBC driver.

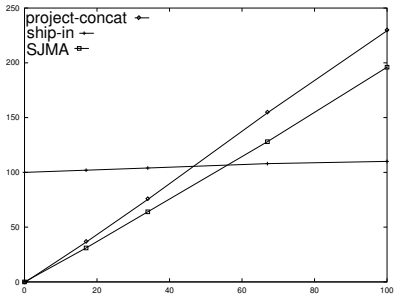
In the first scenario, we deployed an Amos II server at the same workstation as the source. This server wrapped the source and exported it to the client mediator running on another Windows NT workstation. We present test results using this scenario and two different network connection speeds between the workstations: a 115Kb ISDN connection over the public telephone network in Sweden; and a 100Mb departmental LAN. We also varied the speed of the workstation that hosted the client mediator. In one experiment we used a 233 MHz, 32Mb RAM PC, and in the other a 600 MHz, 256Mb RAM PC. In the second scenario the data source was accessed directly from the client mediator through the ISDN network connection using DB2's ODBC interface. In this case the joins are executed one tuple at a time. We also compared the effects of different bulks sizes on the query execution time.

The inner collection is obtained from a table stored in the ODBC data source. The table consisted of three columns: an integer primary key ID, and two textual columns A and B of fixed length strings with sizes 10 and 250. The outer collection was stored in the client mediator, where it simulated an intermediate result. During the execution, the outer collection is bulked and streamed into the join algorithm one bulk at the time. Both the outer and inner collection had the same attributes.

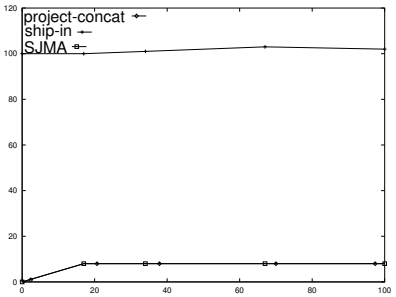
Figure 3 shows the results of the execution of the three join algorithms from the previous section using a 233 MHz Windows NT workstation as a client and an ISDN connection to the server computer. The X axes in the graphs show the sizes of the outer collection in percentage of the size of the inner that always contains 30000 tuples; the Y axes marks query execution times in seconds. The outer collection is scaled from 17% to same the size as the inner. In these experiments the outer collection contained 20% duplicates. Each tuple of the

outer matches exactly one tuple of the inner. The graph on the left compares the execution times for a complete evaluation of the join operation. The graph in the middle compares the times to emit the first 1024 tuples. This coincides with he bulk size used to execute the query. The graph on the right compares the SJMA with PC for different percentages of duplicates in join columns of the collections.

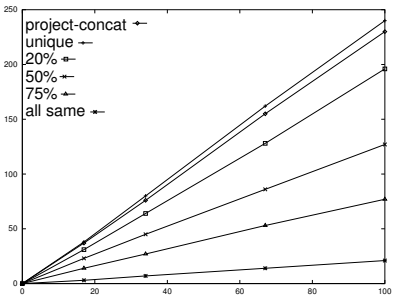
We first analyze the execution times for the complete join operation. Since the inner collection has constant size, the time spend in the Amos II server of the inner and the network time are constant for the execution of the ship-in algorithm. The only increase of execution time is noted in the client: from 8 seconds for a 5000 tuple outer collection, to 16 seconds for a 30000 tuple outer collection. This is due to the increase of the number of index searches. Nevertheless, this increase in negligible in comparison to the total query execution time.



(a) Whole result



(b) First 1024 tuples



(c) SJMA with different percentage duplicates

Figure 3: Execution times when varying the outer collection size, ISDN, 233 MHz PC

	Time distribution			
	Client	Server	Source Access	Net.
Ship-in	10%	1%	4%	85%
Ship-out, PC	5%	3%	43%	49%
Ship-out, SJMA	7%	3%	42%	48%

Table 3: **Query execution time distribution, ISDN, 233 MHz PC**

The ship-out algorithms show performance that is linear to the size of the inner collection, outperforming the ship-in algorithm until the outer is about 50% of the inner. SJMA performs better than the PC algorithm. Figure 3c compares the algorithms for different percentages of duplicates. The PC algorithm performs exactly the same, regardless of the data distribution. SJMA improves as the number of duplicates of the join columns increases. Note that even without duplicates, the performance difference of these two algorithms is small. This shows that in main-memory based mediator systems, the penalty of the additional steps of the SJMA is low.

Table 3 shows the portions of the time spent in the individual system components. The data source access time includes the time spent in the ODBC interface and the data source. The main portion of the execution of the ship-in algorithm executed over ISDN is spent on shipping the inner to the client side, which was consistently around 85% of the query execution time. We can also note that, due to the main-memory architecture of Amos II, the index build time in the client is relatively small, around 5% of the whole execution time. The first tuple is not emitted until the index for the inner is finished, which is after 95% of the processing time. This makes this algorithm unsuitable for top-N queries.

The ship-out algorithms spend less time on the network, but more in accessing the data source. They also emit the first tuple much faster than the ship-in algorithm (Figure 3b). The experiments show here the time to emit the first 1024 tuples. When the bulking factor is less than 10, the first tuple is emitted after less than a millisecond. Furthermore, the bulking factor also determines the smoothness of the flow of the results. Smaller bulking factor will allow smoother flow of the results to the application.

Table 4 compares the effect of the distributed Amos II architecture for the ship-out algorithms. First we used SJMA to access a remote IBM DB2 data source using DB2's ODBC interface over an ISDN connection. Due to the autonomy of the data sources we assume that it is not feasible to materialize intermediate results in the sources. Even if this was possible, due to the disk based nature of the DBMS, we could not expect a comparable execution time as with the main-memory storage used in Amos II. Therefore the join must be performed one tuple at a time over the remote ODBC. However, when the source is accessed through an Amos II server located on the same computer as the source, the join between the client and server mediators is executed in a bulked manner, using only the local ODBC connection between the server mediator and the source, leading to performance improvements of orders of magnitude.

	outer/inner			
Inner size/outer size	17%	33%	66%	100%
through Amos II, all tuples	58	115	245	358
ODBC direct, all tuples	2769	5059	8552	12799
through Amos II, B=1024, first tuple	14	15	15	15
through Amos II, B=1, first tuple	0.7	0.72	0.68	0.71
ODBC direct, first tuple	1.1	0.9	1.2	1.04

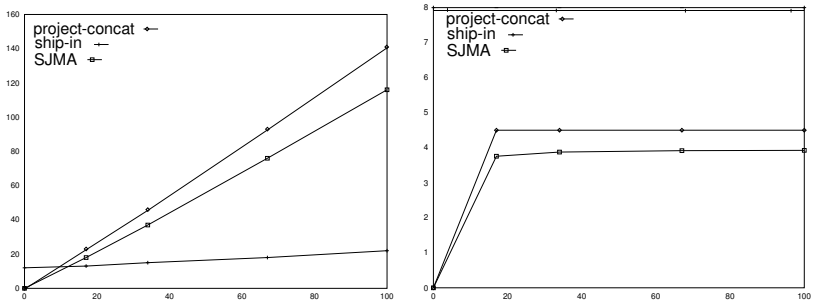
Table 4: **Direct access to an ODBC source and through Amos II servers**

The time to emit the first tuple when the bulking factor is 1024 is notably greater when the processing is done through an Amos II server. This actually represents how long it takes to emit the first 1024 tuples. If fewer tuples are required, a smaller bulking factor leads to better performance for the top-N queries when an intermediate Amos II server is used. Even when the bulking factor is 1 we can note that the use of an intermediate Amos II yields better performance than accessing the source directly, due to communication protocol differences. To achieve the best performance, the bulking factor should match the number of tuples required immediately.

Figure 4 and Table 5 illustrates join execution time on the same client computer connected with a 100Mb fast LAN to the data source. We can note that the curves have similar shapes, while the scale is different. The network cost is eliminated for almost all of the algorithms. In this executions most of the time is spend in the data source (parameterized and unparameterized query execution) and in the client for the ship-in algorithm (index build-up and join). We can also note that when the whole join result is required the ship-in algorithm outperforms the ship-out in almost all the cases. When the first-N tuples are required, however, the ship-out algorithms are more efficient. For the first 1024 result tuples the difference is about 50%. If the number of requested result tuples is smaller, the difference can be a couple of orders of magnitude. We have also varied the client computer from a workstation to a notebook. We noted that the return time for the first tuple is almost constant for the ship-out algorithms regardless of the power of the client computer. This can be explained by the fact that in the case of ship-out algorithms, the server uses the larger share of the workload than with the ship-in algorithms.

	Time distribution			
	Client	Server	Data source acc.	Net.
Ship-in	67%	7%	22%	4%
Ship-out, PC	8%	5%	86%	1%
Ship-out, SJMA	12%	5%	82%	1%

Table 5: **Query execution time distribution, 100Mb LAN, 233MHz PC**



(a) Whole result

(b) First 1024 tuples

Figure 4: Join execution times for different outer collection sizes in percentage of the inner size, 100Mb LAN, client 233MHz PC

5 Related Work

The System R* project [14] is one of the first distributed database prototypes. In System R*, both ship-in and ship-out strategies are examined. In [15] a disk-based ship-in strategy (named ship-whole) is implemented with a disk based b-tree index. This type of implementation leads to considerably different results where the ship-out method always outperforms ship-in.

Disk-based semi-join algorithms are described in [1, 2, 5, 14]. A sort-merge join, bloom filter semi-join, and sort-based semi-join are evaluated in [15] for a distributed database environment. A bloom filter phase can be added to the ship-out algorithms described in this paper. Nevertheless, this would incur additional query processing overhead and possibly shipping of some extra tuples of the inner collection. Bloom filter strategies cannot be used with sources that cannot enumerate the extent of the inner collection.

Most of the mediator frameworks reported in the literature (e.g. [7, 16, 19]) propose centralized query compilation and execution coordination. In [3] it is indicated that a distributed mediation framework is a promising research direction, but to the extent of our knowledge the results in this area are sketchy without experimental support. The protocols for execution of joins between data in different sources are in most cases based on retrieving the data from the sources and assembling the results in the mediator [16, 19]. In the DIOM project [13], a distributed mediator system is presented where the query execution is performed in two phases: subquery execution and result assembly. The dataflow is only from the sources to the mediator.

The Garlic mediator system [7] is the only mediator system known to us that supports ship-out join strategies. The *bind join* in Garlic sends parameters to the sources as single tuples of values. In Amos II the data sources are also accessed one tuple at the time, but the distributed architecture allows for using bulked protocols over high latency lines between Amos II servers to avoid most of the processing cost. A Garlic wrapper that has two components, one local

and one remote, could achieve the benefits of the approach described in the paper. Finally, join methods where bulk shipping is combined with hashing are not applied in Garlic.

6 Summary and Conclusions

An efficient data integration system needs to be able to adapt to different environments by using different algorithms. The algorithms presented in this paper allow for balancing the workload between the client and the server, and for different network use patterns that give wide range of options over different hardware platforms.

The experimental results showed that for a complete query answer the ship-in algorithms generally outperform the ship-out algorithms over fast networks. Over slow networks and with very slow sources, the ship-out algorithms can give orders of magnitude better performance than ship-in since ODBC over TCP/IP calls are executed one tuple at a time while bulks of tuples are shipped between the distributed mediators. For top-N queries where N is considerably smaller than the result size, the ship-out algorithms with bulking factor N give the best performance over all the range of hardware and network connections used in the experiments. These outperform the ship-in algorithms by a few orders of magnitude. Although the bulking factor greater than 1 provides benefits, too large bulk sizes lead to reduced query execution efficiency.

In our environment, where the index operations are main-memory based and relatively cheap, the penalty of SJMA (the Semi-Join with Materialized index Algorithm) is small and it always performs nearly as well, or better than PCA (the Project-Concat Algorithm). Nevertheless, PCA uses less memory and could be much more efficient in memory-limited mediators. A compromise between these two algorithm is the SJMA with a limited size temporary index that degenerates to a SJA when the temporary index size is 0. Finally, if simplicity of implementation is considered the PCA is the algorithm of choice.

Placing an mediator server close to the source allows for bulked execution of the protocols that might change the query execution time by orders of magnitude, especially in networks with high latency. In cases when the sources lack filtering capability, the mediator server can also locally filter the query fragment result and reduce the communication cost even more.

A topic of our current work is a strategy to dynamically select between the proposed algorithms during run-time. Statistics collected during the execution can be used to determine if the default choice was the best one. Another open issue is a method to determine the optimal bulking factor in a multi join query, by taking in account the tuple sizes, join selectivities and the buffer pool size.

References

[1] P. Apers, A. Hevner, and S. Yao: Optimization Algorithms for Distributed Queries. *IEEE Transactions on Software Engineering*, 9(1), 57-68, 1983

[2] P. Bernstein and D. Chiu: Using Semi-joins to Solve Relational Queries. *Journal of ACM* 28(1), 25-40, 1981

[3] W. Du and M. Shan: Query Processing in Pegasus, In O. Bukhres and A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*. Pretince Hall, 449-471, 1996.

[4] G. Fahl and T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, Springer, 6(4), 261-281, 1997.

[5] P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie Jr.: Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4), 602-625, 1981

[6] G. Graefe and W. J. MCKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. *12th Data Engineering Conf. (ICDE'93)*, 209-218, 1993.

[7] L. Haas, D. Kossmann, E.L. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *23th Intl. Conf. on Very Large Databases (VLDB'97)*, 276-285, 1997

[8] V.Josifovski and T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration. *Intelligent Information Systems (JIIS)* 12(2-3), Kluwer, 165-190, 1999.

[9] V.Josifovski and T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. *25th Intl. Conf. on Very Large Databases (VLDB'99)*, 435-446, 1999.

[10] V. Josifovski and T. Risch: Query Decomposition for a Distributed Object-Oriented Mediator System. To appear in *J. of Distribued and Parallel Databases*, Kluwer, 2001.

[11] E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, and M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.

[12] H. Lin, T. Risch and T. Katchanounov: Adaptive data mediation over XML data. To appear in *J. of Applied System Studies (JASS)*, Cambridge International Science Publishing, 2001.

[13] L. Liu and Calton Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Journal of Distributed and Parallel Databases* 5(2), 167-205, Kluwer Academic Pulishers, The Netherlands, 1997.

[14] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger and P. Wilms: Query Processing in System R*. In W. Kim, D. Reiner, D. Batory (eds.): *Query Processing in Database Systems*, Springer-Verlag, 1985.

[15] L. Mackert and G. Lohman: R* Optimizer Validation and Performance Evaluation for Distributed Queries. In M. Stonebraker (ed.): *Readings in Database Systems*, Morgan-Kaufmann, CA, 1988

[16] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac: Dynamic Query Optimization in Multidatabases. *IEEE Data Engineering Bulletin*, 20(3), 38-45, 1997.

[17] T. Risch and V. Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers. To appear in *Concurrency - Practice and Experience J.*, John Wiley & Sons, 2001.

[18] M. Roth, F. Ozcan and L. Haas: Cost Models DO MAtter: Providing Cost Information for Diverse Data Sources in Fededrated System. *25th Intl. Conf. on Very Large Databases (VLDB99)*, 599-610, 1999.

[19] A. Tomasic, L. Raschid and P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions in Knowledge and Data Engineering*, 10(5), 808-823, 1998

Paper F:

Timour Katchaounov, Tore Risch, and Simon Zürcher. Object-oriented mediator queries to internet search engines. In *Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 176–186, Springer-Verlag, September 2002.

Object-Oriented Mediator Queries to Internet Search Engines

Timour Katchaounov, Tore Risch, Simon Zürcher

Uppsala Database Laboratory, Department of Information Technology,
Uppsala University, Sweden

Abstract. A system is described where multiple Internet search engines (ISEs), e.g. Alta Vista or Google, are accessed from an Object-Relational mediator database system. The system makes it possible to express object-oriented (OO) queries to different ISEs in terms of a high level OO schema, the *ISE schema*. The OO ISE schema combined with the mediator database system provides a natural and extensible mechanism in which to express queries and OO views that combine data from several ISEs with data from other data sources (e.g. relational databases). High-level OO web queries are translated through query rewrite rules to specific search expressions sent to one or several wrapped ISEs. A generic ISE query function sends the translated queries to a wrapped ISE. The result of an ISE query is delivered as a stream of semantically enriched objects in terms of the ISE schema. The system leverages publicly available *wrapper toolkits* that facilitate extraction of structured data from web sources, and it is independent of the actual wrapper toolkit used. One such wrapper toolkit was used for generating HTML wrappers for a few well-known ISEs.

1. Introduction

To facilitate the combined access to data on the web with data from other databases, a system called ORWISE (Object-Relational Wrapper of Internet Search Engines) has been developed that can process queries combining data from different Internet search engines (ISEs) with data from regular databases and other data sources. The design of ORWISE leverages available *wrapper toolkits* to extract information from web pages. ORWISE has been implemented for three well-known search engines using a publicly available wrapper toolkit [31].

ORWISE is an extension to the database system Amos II [29], [30], that is based on the *wrapper-mediator approach* [34] for heterogeneous data integration. The core of Amos II is an extensible object-relational database engine having mediation primitives in a query language *AmosQL* similar to the OO parts of SQL-99 and ORWISE thus permits SQL-99 like queries that combine ISE results with data from other types of sources such as relational databases [10] and XML [23]. Amos II is suitable for collecting and processing results from ISEs because its purpose is to act as a fast mediator database which can manage meta-data of heterogeneous and distributed data sources and efficiently process queries to the sources.

The generalized *ISE wrapper manager* ORWISE, described in this paper, makes it possible to easily access one or several ISEs from Amos II using different *ISE*

wrappers for each engine. Combined with OO mediation facilities [4], [17], it allows to process OO database queries that combine data from several ISEs with data from conventional databases and other data sources. In difference to relational systems for web queries [14], the data produced by ORWISE is not just text strings but much more semantically rich object structures in terms of an OO schema for ORWISE, called the *ISE schema* (Internet search engine schema). The ISE schema describes capabilities and other properties of the search engines along with the structure of their results.

ISEs have some special problems compared to 'conventional' databases:

- *Semi-structured interfaces*: There are no standard interfaces to ISEs such as ODBC and JDBC. Web forms are used for specifying queries and other inputs to them. The result of an ISE query is a semi-structured web document containing not just the query result but also auxiliary text, banners, etc., which need to be filtered out from the query result.
- *Query languages*: ISEs do not have a standardized query language such as SQL but every ISE has its own query language with varying syntax and semantics.
- *Autonomy*: The content, structure and availability are totally controlled by the information supplier.
- *Evolution*: Internet sites tend to change very often. A system that accesses a site has to be very flexible.
- *Heterogeneity*: The data delivered by ISEs have varying structures and the system has to reconcile semantic differences.

In order to handle the above problems we need reliable and flexible interfaces to the ISEs, here termed *ISE wrappers,* which can programmatically fill and submit web forms and parse the structure of an ISE result document searching for predefined patterns. An ISE wrapper must be flexible enough to cope with small changes in the web sites.

To specify web source wrappers ORWISE utilizes wrapper toolkits to extract useful information from web pages. ORWISE is designed to be independent of the actual wrapper toolkits used. We investigated several of them to make sure that the system works with all of them. For our first implementation we chose W4F [31] to generate ISE wrappers for three search engines - Google (http://www.google.com), AltaVista (http://www.altavista.com), and Cora Research Paper Search (Cora) (http://cora.whizbang.com).

The ISE wrappers are connected to the system through a generic query language function called *orwise*, which is a foreign function (implemented in Java) overloaded for each search engine. It returns objects of an ISE specific type[1] that describes the retrieved query results. New ISE wrappers can dynamically be added to the system by creating a new subtype of the system type *SearchEngine* for each new ISE and then implementing some code (in Java) to interface its ISE wrapper. The overloading of the function *orwise* is used for facilitating the plug-in of new ISE wrappers.

Once a new search engine is connected to the *orwise* function it can be used in OO queries. Since the parameters for each implementation of *orwise* are search engine specific, such queries will be rather detailed with search engine specific parameters for, e.g., query strings, site names, etc. The system therefore provides

[1] We use the terms 'type' and 'class' as synonyms.

high-level query functions that can be used for any ISE and where queries are specified uniformly. For example, the function *webSearch* is defined for every search engine to specify OO queries to it in a search engine independent form. The high-level OO query expressions need to be transformed before the actual call(s) to *orwise* is issued. The approach in Amos II is to implement a *translator* module for each kind of data source (search engine, relational database, etc.). In the case of ISEs, the translators rewrite the high-level query into search engine query specifications containing calls to *orwise*. Since different search engines have different ways of specifying searches, they have different rewrite rules.

In summary, ORWISE provides i) the ISE schema for describing and querying data from any ISE, ii) a mechanism to specify search engine specific translators, and iii) facilities to allow different wrapper toolkits to be easily plugged into the system.

2. Related work

Many projects (e.g. [11], [16], [21], [27], [33]) use the mediator approach to data integration in general. The work presented here describes how an object-relational mediation framework [29] leverages upon an available wrapper toolkit to provide access to ISEs.

The use of object-relational approach in querying the structure of XML Web documents has been done, e.g., in [3], [8], [12], [23]. A query language standard for XML, XQuery [35], is being developed with which the contents of XML documents can be queried and new XML documents constructed. All major ISEs use HTML, not XML. General Web query languages for HTML are proposed in [19], [25]. These are general languages for querying well-formed Web documents and not directly suitable for defining embedded interfaces to ISEs.

By contrast *wrapper toolkits* [9], [13], [15], [18], [20], [22], [24], [31] specify programmatic interfaces to web sources handling both sending commands and extracting structured data from responses. They often include some advanced pattern matching language to extract data from Web documents as regular expressions operating on varying levels of granularity. With a wrapper toolkit a web source wrapper is defined by processing *wrapper specifications*, consisting of statements to connect to web sources and to detect the parts of the text to be extracted. They allow new wrappers to be specified much easier than with manual programming and the developers need not master a complex query language. A good overview of projects related to wrapper construction for Web sources can be found in [31].

A wrapper toolkit can be a *wrapper-generator* that generates code (e.g. Java) implementing a web source wrapper [1], [2], [24], [31]. It can also be a *wrapper-interpreter* where the web source wrapper is specified as commands, which are interpreted at run time [18], [15]. ORWISE is designed to work with both wrapper-generators and wrapper-interpreters. Web source wrappers represent data differently and are not sufficient themselves to combine data from Web sources and conventional databases. Therefore there is need for data mediation facilities along the lines of this paper.

In [26] it is shown that an OO query language indeed is very useful for specifying queries to text engines. Our work differs in that we propose leveraging upon using external wrapper toolkits, OO query rewrites, and the ISE schema. Furthermore, we explicitly model the capabilities of the search engines in the ISE schema, rather than

in the internals of the system. The WSQ/DSQ [14] project proposes an architecture where web searches are specified as SQL queries to two virtual relational tables. Their relational tables are inflexible for the purpose, compared to our ISE schema. The focus of the work in [5] is re-write rules and cost models for integrating text search with other queries. Those rewrite rules are applicable in our translator too.

To the best of our knowledge, no other project proposes a system that uses inheritance and overloading to model ISEs and their results on the conceptual level, while at the same time the implementation is independent of, and leverages existing wrapper toolkits. Another major difference to other projects is that our object-oriented ISE schema distinguishes between the search engine specific descriptions of documents and the actual documents. Furthermore, the ISE capabilities are modeled in the ISE schema too.

3. Scenario

We have implemented the scenario of Figure 1 to illustrate the functionality of the system. In the scenario, an Amos II mediator is used to process queries that combine data from a relational DB2 database through ODBC with three ISEs, AltaVista, Google and Cora. The access to the three Internet search engines uses the ORWISE wrapper, while the relational database is accessed through an ODBC-wrapper.

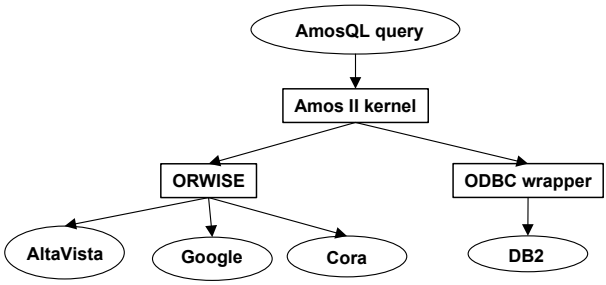


Fig. 1. Mediator scenario.

The relational database stores a table of employees that is mapped to the mediator type *Employee*, using the techniques for defining OO views of relational databases [10]. The following AmosQL query uses Google to find the names of those employees who are mentioned in some web page in the web site `'www.csd.uu.se'`:

```
SELECT DISTINCT given_name(e), family_name(e)
FROM Employee e, DocumentView d, Google ise
WHERE d = webSearch(ise, given_name(e)) AND
      d = webSearch(ise, family_name(e)) AND
      host(url(d))=' www.csd.uu.se');
```

The first two lines of the 'where' clause in the query retrieve the documents that contain given names and family names of employees in the relational database, while the last line restricts the search to only those persons whose names are found

by Google in web pages on the host `'www.csd.uu.se'`. Other text-related predicates such as `'near'` can also be added to refine the search. The type *DocumentView* represents descriptions of documents returned by an ISE and the type *Google* represents the wrapper for Google. The same query can be specified for Alta Vista by replacing the type *Google* with *AltaVista*. It is also possible to specify queries over several search engines by using the generic supertype *SearchEngine* instead of *AltaVista* or *Google*.

4. The ISE Schema

Queries to ISEs are posed in terms of the OO database schema on Figure 2. Inheritance and overloading are used to model heterogeneity of both ISEs and their results. Furthermore, we separate the description of results returned by ISEs from the documents themselves. Since Amos II has a functional data model [32], both type attributes and relationships between types are modeled by functions shown as think lines on Figure 2. For clarity, the overloaded function *orwise* is represented as an attribute of the subtypes of type *SearchEngine*. The core of the ISE Schema consists of three base types:

- *SearchEngine* – this type is used to categorize ISEs. It reflects the fact that search engines have different query capabilities and parameters. It has a subtype for each specific ISE normally with only one instance. The generic function *orwise* is overridden for each ISE to reflect their different semantics. Analogously each of them has a specific query rewrite function.
- *DocumentView* – objects of this type describe the results of a query to different ISEs. By introducing this type of objects we can distinguish between the documents themselves and the description of a document by an ISE. Document views often contain information about a document that is not part of the document itself and is imprecise or outdated. They may use different formats from the document itself; e.g. the Cora ISE returns HTML descriptors of PostScript documents. Differentiating between documents and views over documents allows for more precise queries.
- *Document* – describes document objects themselves. Subtypes of *Document* may describe document objects with different structure. The problem of querying structured documents is outside the scope of this work and has been addressed by other researchers [6], [28]. All this work can be easily reused in our system due to the flexibility of our OO data model.

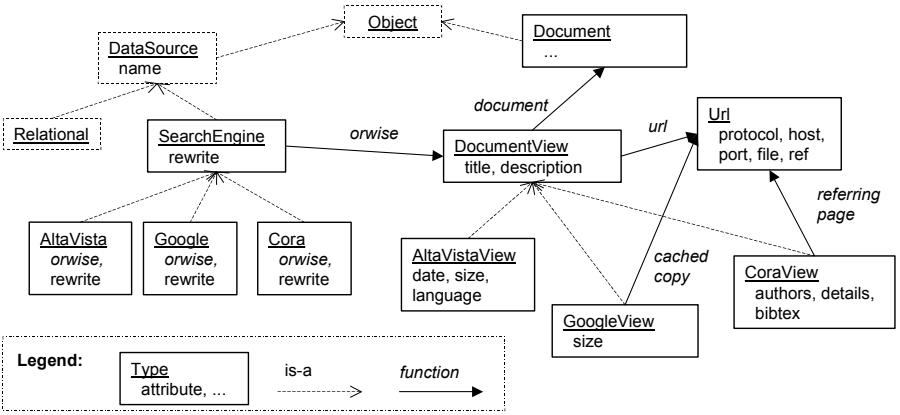


Fig. 2. The ISE schema

The two classes *DataSource* and *Relational* are part of the general Amos II meta-type hierarchy. The type *DataSource* serves as the base type of all meta-types for different kinds of data sources accessible through the mediator system. One such meta-type is *Relational*, which describes relational data sources. It has the function *sql*, analogous to the *orwise* function of *SearchEngine*. In our current implementation, the type *SearchEngine* has three subtypes for each of the wrapped search engines AltaVista, Google and Cora. Each of them defines its own version of the *orwise* function and specific rewrite rules. Correspondingly the type *DocumentView* has three subtypes: *AltaVistaView, GoogleView and CoraView*, where each of them has additional properties. For example, of the three ISEs only AltaVista returns the language of a document, while only Google may provide a locally cached copy of it's indexed documents, accessible through the function *cached_copy*. Finally, *Document* objects may be accessed and queried further through the *document* function of the type *DocumentView*. The type *Url* is an example of semantic enrichment of the ISE query results, as they return URLs as strings.

5. The ORWISE Architecture

Figure 3 shows the layered architecture of the system. The left part shows how ORWISE is interfaced with the Amos II kernel, while the right part shows the layers of ORWISE itself.
The architecture is designed to fulfill several requirements:

- It provides a uniform interface from the Amos II query processor to any ISE.
- It can use any existing general wrapper toolkits.

- It is independent of the wrapper toolkits used.
- It is possible to easily add a new ISE wrapper without any changes to the rest of the system.
- There is no need to modify the definitions of wrappers generated by wrapper-generators.

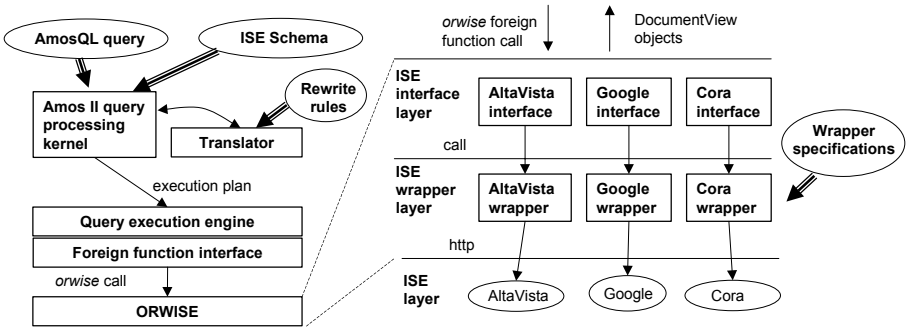


Fig. 3. System architecture.

The two layers *ISE interface* and *ISE wrapper* fulfill these requirements. This architecture permits any wrapper toolkit to be used and different kinds of wrapper toolkits can even be combined.

The ISE interface layer defines an interface between the Amos II kernel and the underlying ISE wrapper layer used for interfacing each search engine. The functionality common for every ISE wrapper, such as instantiating ISE specific *DocumentView* objects and emitting the result stream, is encapsulated in this layer. It is called by the query processor and it calls the ISE wrapper for the chosen search engine. The basic foreign function interface of Amos II allows new ISE interfaces to be dynamically added to a running system. The ISE wrappers are specified by some external wrapper toolkit(s) chosen for each particular search engine. Therefore, the functionality they expose can be very different and cannot be directly used by ORWISE. The ISE interface therefore must instantiate objects, convert strings to URL objects or numbers, etc.

The ISE wrapper layer consists of the modules specified through the wrapper toolkit. It forms and sends HTTP requests to an ISE server and then extracts the results from the so received HTML page. The input to a wrapper toolkit is a specification of request submission and data extraction rules for a web source. The chosen W4F [31] toolkit is a wrapper-generator, which generates Java classes per wrapped data source. In this case the layer consists of the generated code. For wrapper-interpreters the interpreter together with the specifications is the layer.

With this layered architecture, the following steps are needed to add a new search engine to ORWISE:

1. Design an ISE wrapper for the specific search engine by using a chosen wrapper toolkit. For example in the case of W4F this involves specifying the extraction rules in terms of the HEL extraction language from which a Java class is generated per each wrapped web source. By contrast, wrapper-interpreters are

directly called from the ISE interface layer using the wrapper specifications as parameters.
2. Create types in the mediator database as subtypes of *SearchEngine* and *DocumentView*.
3. Design an ISE interface module as the overloaded Amos II foreign function, *orwise*, calling the ISE wrapper module from step 1.

Once step 1-3 are completed the ISE is already queryable directly through *orwise*. However, the queries can be complex and very ISE dependent. Efficient and transparent queries to an ISE therefore requires an additional step:

4. Design the rewrite rules needed for the ISE to translate between, e.g., *webSearch* calls and the particular *orwise* calls.

6. Translating ORWISE Queries

Queries calling the *webSearch* function combined with other Web document related predicates are translated to an equivalent but more efficient query containing optimized calls to the function *orwise* overloaded for specific ISEs. The function *webSearch* could be defined as a query calling *orwise* without any translation. However such untranslated execution may be significantly less efficient. In our example, the Google query is translated to the following *orwise* query:

```
SELECT given_name(e), family_name(e)
FROM Employee e, DocumentView d, Google gse
WHERE d = orwise(gse, given_name(e) + ' ' +
family_name(e), 20, 'www.csd.uu.se', 'english');
```

where the signature of *orwise* is Google specific. Here *orwise* for Google takes the parameters *query*, *result size*, *language restriction*, and *host*. The function is defined as a foreign AmosQL function that calls the underlying ISE wrapper for Google. The example illustrates the semantic rewrite of the original query by the translator, where several calls to *webSearch* and *host* are combined into one call to Google's *orwise*. The translator also added the default specifications of '*english'* as language and that only the first 20 results should be returned. The result of *orwise* is a stream of *GoogleView* objects. The translator for each ISE knows how to generate optimized *orwise* calls with specific parameters expressing ISE supported capabilities.

As shown in the example, queries to a search engine will contain subqueries expressed using the specific query language of the ISE, which is usually different for different ISEs. In the example above the string "`given_name(e) + ' ' + family_name(e)`" is an example of the construction of a conjunctive query to Google (it uses AND by default). During query translation, there are possible query transformations that can dramatically improve performance and result quality. We have implemented some translator rules to show the usefulness of the system and can utilize other results in related areas [5], [6].

7. Summary

A flexible system for querying Internet search engines through an OO mediator database system was presented. The system has the following unique combination of features:

1. Data about both the search engine capabilities and the results they return were modeled in an OO *ISE schema* in a mediator database.
2. The ISE schema permits transparent queries to ISEs with different capabilities and result structures. The mediation facilities provide for processing heterogeneous queries that combine data from ISEs with data from other data sources.
3. New kinds of ISEs can be easily plugged in. The system assumes the ISEs are autonomous and outside the control of the query processor.
4. The system is designed to be independent of the wrapper toolkits used for specifying the ISE wrappers. Several such publicly available toolkits were evaluated to choose one for the implementation.
5. The query processor provides a mechanism to plug in OO search engine specific rewrite rules for translating OO queries into the parameterized *orwise* calls. The system is independent of the actual rewrite rules to utilize previous work in this area.

References

1. B. Adelberg: NoDoSe – A Tool for Semi-Automatically Extracting Structured and Semistructured Data from Text Documents, *SIGMOD 1998 Conference*: 283:294, 1998.
2. N. Ashish, C. Knoblock: Semi-automatic Wrapper Generation for Internet Information Sources. *CoopIS'97 Conference:* 160-169, 1997.
3. G. Arocena, A. Mendelzon: WebOQL: Restructuring Documents, Databases, and Webs. *In Proc. ICDE'98*, Orlando, 1998.
4. O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems.* Prentice Hall, 1996.
5. V. Christophides, S. Abiteboul, S. Cluet, M Scholl: From Structured Documents to Novel Query Facilities. *SIGMOD 1994 Conference*: 313-324, 1994.
6. S. Chaudhuri, U. Dayal, T. W. Yan: Join Queries with External Text Sources: Execution and Optimization Techniques. *SIGMOD 1995 Conference*: 410-422, 1995.
7. C. Chang, H. Garcia-Molina, A. Paepcke: Predicate rewriting for translating Boolean queries in a heterogeneous information system. *ACM Trans. on Information Systems*, 17(1), 1999.
8. Donald D. Chamberlin, Jonathan Robie, Daniela Florescu: Quilt: An XML Query Language for Heterogeneous Data Sources. *WebDB'2000*: 53-62, 2000.
9. A. Firat, S. Madnick, M. Siegel: The Caméléon Web Wrapper Engine, *First Workshop on Technologies for E-Services*, Cairo, 2000.
10. G. Fahl, T. Risch: Query Processing over Object Views of Relational Data*, The VLDB Journal* , 6(4), 261-281, 1997.
11. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Intelligent Information Systems (JIIS),* Kluwer, 8(2), 117-132, 1997
12. R. Goldman, J. McHugh, J. Widom: From Semisturctured Data to XML: Migrating the Lore Data Model and Query Language, *WebDB'99*, 1999.

13. J. Gruser, L. Raschid, M. Vidal, L. Bright: Wrapper Generation for Web Accessible Data Sources. *CoopIS'98 Conference*: 14-23, 1998
14. R. Goldman, J. Widom: WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. *SIGMOD 2000 Conference*: 285-296, 2000.
15. G. Huck, P. Fankhauser, K. Aberer, Erich J. Neuhold: Jedi: Extracting and Synthesizing Information from the Web. *CoopIS'98 Conference*: 32-43, 1998.
16. L. Haas, D. Kossmann, E. L. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *23rd Intl. Conf. on Very Large Databases (VLDB'97),* 276-285, 1997
17. V. Josifovski, T. Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, *25th Conference on Very Large Databases (VLDB'99),* 435-446, 1999.
18. T. Kistlera, H. Marais: WebL: a programming language for the Web. In *WWW7*, Brisbane, Australia, *http://www.research.digital.com/SRC/WebL/,* 1998.
19. D. Konopnicki, O. Shmueli. W3QS: A query system for the World Wide Web. *21st Conference on Very Large Databases (VLDB'95)*, 54–65, Zurich, Switzerland, 1995.
20. N. Kushmerick, D. Weld, R. Doorenbos: Wrapper Induction for Information Extraction. *IJCAI'97* Vol. 1: 729-737**,** 1997**.**
21. L. Liu, C. Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed and Parallel Databases,* Kluwer, 5(2), 167-205, 1997.
22. L. Liu, C. Pu, W. Han: XWRAP: An XML-Enabled Wrapper Construction System for Web Information Sources. *ICDE 2000:* 611-621, 2000.
23. H. Lin, T. Risch, T. Katchanounov: Adaptive data mediation over XML data. To be published in special issue on "Web Information Systems Applications" of *Journal of Applied System Studies (JASS)*, Cambridge International Science Publishing, 2001.
24. G. Mecca, P. Merialdo, P. Atzeni: ARANEUS in the Era of XML. *IEEE Data Engineering Bulletin*, Special Issue on XML, September, 1999.
25. A. O. Mendelzon, G. Mihaila, T. Milo: Querying the World Wide Web. *International Journal on Digital Libraries*, 1(1), 54-67, April 1997.
26. A. Paepcke: An Object-Oriented View Onto Public, Heterogeneous Text Databases. *Proceedings of the Ninth International Conference on Data Engineering (ICDE'93),* 1993.
27. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom: Querying Semistructured Heterogeneous Information. In *Deductive and Object-Oriented Databases, Proceedings of the DOOD'95 conference*, 1995, LNCS Vol. 1013, 319-344, Springer 1995.
28. D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom: Querying Semistructured Heterogeneous Information. In *Deductive and Object-Oriented Databases, Proceedings of the DOOD'95 conference*, 1995, LNCS Vol. 1013, 319-344, Springer 1995.
29. T. Risch, V. Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers, To be published in *Concurrency – Practice and Experience J.,* John Wiley & Sons, *http://www.csd.uu.se/~udbl/publ/concur00.pdf*, 2001.
30. T. Risch, V. Josifovski, T. Katchaounov: *Amos II Concepts*, *http://www.csd.uu.se/~udbl/amos/doc/amos_concepts.html*, 2000.
31. A. Sahuguet, F. Azavant: Building Intelligent Web Applications Using Lightweight Wrappers, *Data and Knowledge Engineering*, 36(3), 283-316, March, 2001.
32. D. W. Shipman: The Functional Data Model and the Data Language DAPLEX, *TODS,* 6(1), 140-173, 1981.
33. A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Date Engineering*, 10(5), 808-823, 1998
34. G. Wiederhold: Mediators in the architecture of future information systems, *IEEE Computer,* 25(3), 38–49, 1992.
35. XQuery: A Query Language for XML, W3C Working Draft, 15 February 2001, *http://www.w3.org/TR/xquery/*.