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 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

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

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 shipped one tuple at a time to the query peer. The feasibility of these strategies depends

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

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'.

```
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.

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 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 subplans, 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 subsection 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 traditional 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 stand-alone 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 directly 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 experimentally. 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 *Functional Approach to Computing with Data*. 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.