Distributed and Parallel Databases, 11, 307–336, 2002

Query Decomposition for a Distributed Object-Oriented Mediator System

VANJA JOSIFOVSKI* vanja@almaden.ibm.com
TORE RISCH tore.risch@dis.uu.se
Uppsala Database Laboratory, Computing Science Department, Uppsala University, Uppsala, 751 05 Sweden

Recommended by: Marek Rusinkiewicz

Abstract. The mediator-wrapper approach to integrate data from heterogeneous data sources has usually been centralized in the sense that a single mediator system is placed between a number of data sources and applications. As the number of data sources increases, the centralized mediator architecture becomes an administrative and performance bottleneck. This paper presents a query decomposition algorithm for a distributed mediation architecture where the communication among the mediators is on a higher level than the communication between a mediator and a data source. Some of the salient features of the proposed approach are: (i) exploring query execution schedules that contain data flow to the sources, necessary when integrating object-oriented sources that provide services (programs) and not only data; (ii) handling of functions with multiple implementations at more than one mediator or source; (iii) multi-phase query decomposition using a combination of heuristics and cost-based strategies; (iv) query plan tree rebalancing by distributed query recompilation.

Keywords: query decomposition, middleware, data integration, distributed databases

1. Introduction

An important factor of the strength of a modern enterprise is its capability to effectively store and process information. As a legacy of the mainframe computing trend in the recent decades, large enterprises often have many isolated data repositories used only within a portion of the organization. The number of such isolated repositories increases even today due to organizational reasons. The inability of these systems to interoperate and provide the user with an unified view of the data and the resources of the whole enterprise is a major obstacle in taking the corporate structures to the next level of efficiency.

The *wrapper-mediator* approach [29] divides the functionality of a data integration system into two units. The wrapper provides access to the resources using a *common data model* and a *common query representation*. The mediator provides a coherent view of the enterprise resources according to the needs of the user, and processes the queries posed by the user. The query processing is one of the key technical challenges in a design and implementation of a mediator system. The queries are usually specified in a declarative query language and decomposed into query execution plans (schedules) to be executed in the mediator or in the sources by a process named *query decomposition*.

*Present address: IBM Almaden Research Center, San Jose, CA 95120, USA.

It has been identified in previous research on mediator systems [6, 11, 14, 22] that a distributed mediation architecture is necessary in order to avoid administrative and performance bottlenecks while providing modularity, scalability and reuse of specification in the integration process. The mediation in such a framework is performed by a set of distributed cooperating software components/mediators. While the distribution is inherit to a mediation environment due to the distribution of the data sources, a distributed mediation framework introduces interaction between the mediators that is on a higher level than the interaction between the mediators and the data sources. A mediator in such a network does not treat the other mediators as just another type of a data source.

Although distributed mediation has been identified as a promising research direction, most of the reported prototypes (with an exception of [22]) have centralized architectures [6, 13]. In this paper we present the query decomposition process in the AMOS*II* mediator system. In addition to being a distributed, the object-oriented (OO) approach used in AMOS*II* allows for integration of sources that provide operations (programs, logic) as opposed to the traditional assumption that the sources store only data. This leads to query execution schedules that perform shipment of data to the sources,[1] as opposed to the approaches where the mediator first retrieves the relevant data from the sources and then composes the requested result.

Another feature of the approach described in this paper is that it allows the user to define operations executable in more than one source or mediator. Some *source types* have certain basic operations implemented in every data source instance (e.g. comparison operators in relational databases). When more than one source of the same type is attached to the same mediator there is a choice where to execute such operations. The source types are organized in a hierarchy for reuse of the specifications of the basic operations.

An important issue in the design of a query decomposer for a mediator system is the division of the functionality between the wrapper and the mediator software components. The decomposer, being a central part of the mediator, should provide the facilities that are commonly needed for integration of data sources, so these features need not be re-implemented in each wrapper. On the other hand, since the query decomposer is a complex piece of software, it should have as simple design as possible in order to be maintainable and have satisfactory performance. The work presented in this paper offers a trade-off where the decomposer has simple but effective facilities for handling sources with different capabilities. The intent is to provide facilities to handle most of the data sources using very simple wrappers and an efficient decomposition process. The sources with more specific requirements can be handled by implementation of a more elaborate wrapping software layer. Although this does not allow for modeling the capabilities of the data sources to details as provided by other approaches (e.g. [28]) that are based on context-free grammars and rules, it provides faster query decomposition, leading to more efficient query processing while having simple wrapper definitions for most of the commonly encountered types of data sources.

The query decomposition process described in this paper is performed in several heuristic and cost based phases that can be grouped into three stages:

– *Query fragment* generation. This phase breaks the query into a set of fragments each executed at a single source/mediator.

– Cost based scheduling of the execution of the query fragments. Different data flows are explored. The fragments are compiled in the wrappers/data sources to obtain execution cost estimates for different data flow patterns.
– Rebalancing of the left-deep tree generated by the previous stages, by a distributed query fragment (re)compilation at the participating mediators and sources.

The final distributed execution plans operate over *bulks* of data rather than over single instances.

The paper proceeds as following. Section 2 presents an overview of the basic features of the AMOS*II* system. Related work is surveyed in Section 3. The central part of the paper is Section 4 which presents the query decomposition in AMOS*II* . Section 5 summarizes the paper.

2. Overview of the AMOS*II* system

As a platform for our research we used the AMOS*II* mediator database system [9] developed from WS-Iris [21]. The core of AMOS*II* is an open light-weight and extensible DBMS. For good performance, and since most the data reside in the data sources, AMOS*II* is designed as a main-memory DBMS. Nevertheless, it contains all the traditional database facilities, such as a recovery manager, a transaction manager, and a OO query language named AMOSQL [9]. An AMOS*II* server provides services to applications and to other AMOS*II* servers.

Figure 1 illustrates the three layer architecture of AMOS*II* where the mediator is placed between the top level containing applications, and the bottom level containing various kinds of data sources. Each mediator server has a *kernel* containing the basic DBMS facilities that is extensible through *plug-ins* that are program modules written in some regular programming languages. AMOS*II* currently supports plug-ins written in C, Java and Lisp.

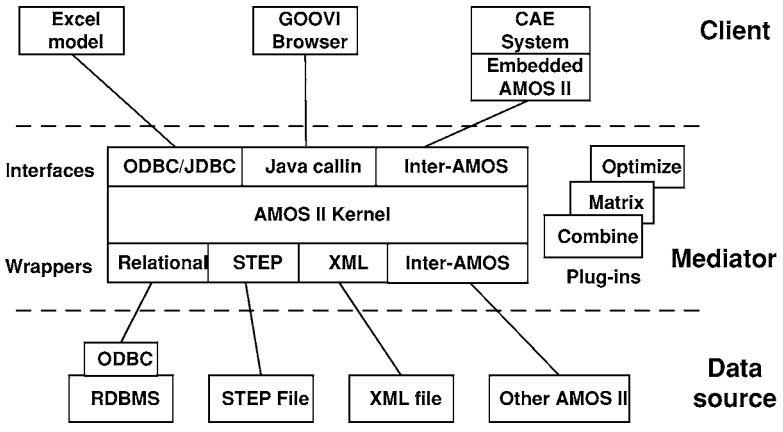

Figure 1. AMOS*II* architecture.

In order to access data from external data sources AMOS*II* mediators may contain one or several wrappers that interface and process data from external data sources. The wrappers are implemented by program modules in a mediator server having specialized facilities for query processing and translation of data from a particular kind external data sources. It contains both interfaces to external data repositories and knowledge how efficiently to translate and process queries involving accesses to a particular kind of external data sources. More specifically the wrappers perform the following functions:

– *Schema importation*: the explicit or implicit schema information from the sources is translated into a set of AMOS*II* types and functions.
– *Query translation*: object calculus is translated into equivalent query language expressions referencing functions in the plug-ins that use API calls to invoke functions in the sources.
– *OID generation*: when OIDs are required for the data in the sources, the query language expressions to be executed in the source are augmented with code or API calls to extract the information needed to generate these OIDs.

Analogously, different types of applications require different interfaces to the mediator layer. For example, there are call level interfaces allowing AMOSQL statements to be embedded in the programming languages Java, C, and Lisp. Figure 1 illustrates three such *call-in* interfaces. It is even possible to closely embed AMOS*II* with applications, e.g. a Computed Aided Engineering (CAE) system [25]. The AMOS*II* kernel is then directly linked with the application.

The kernel of AMOS*II* can be also extended with plug-ins for customized query optimization, data representations (e.g. matrix data), and specialized algorithms that are needed for integrating data from a particular application domain. Through the plug-in features of AMOS*II*, domain oriented algorithms can easily be included in the system and made available as new query language functions in AMOSQL.

The data model of AMOS*II*, also used as a common data model for data integration, is an OO extension of the DAPLEX [27] functional data model. It has three basic constructs: *objects*, *types* and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types making the object an *instance* of those types. The set of all instances of a type is called the *extent* of the type. The types are organized in a multiple inheritance, supertype/subtype hierarchy. If an object is an instance of a type, then it is also an instance of all the supertypes of that type. Object attributes, queries, methods, and relationships are modeled by functions. The non-literal types are divided into *stored*, *derived* and *proxy* types. The instances of the *stored* types are explicitly stored in the mediator and created by the user. The extent of a *derived* type is a subset of the extents of one or more supertypes specified through a declarative query over the supertypes. The *proxy* types represent objects stored in other AMOS*II* servers or in some of the supported kinds of data sources. The derived and proxy types are described in greater detail in [15].

The functions in AMOS*II* are divided by their implementations into four groups. The extent of a *stored* function is physically stored in the mediator (c.f. object attributes). *Derived* functions are implemented by queries in the query language AMOSQL (c.f. views

and methods). *Foreign* functions are implemented in some other programming language, e.g. C++ or Java (c.f. methods). To help the query processor, a foreign function can have associated cost and selectivity functions. The *proxy* functions are implemented in other AMOS*II* servers.

The AMOSQL query language is similar to OQL and based on OSQL [23] with extensions of multi-way foreign functions, active rules, late binding, overloading, etc. For example, assuming three functions *parent*, *name* and *hobby* defined over the type *person*, the query on the left below retrieves the names of the parents of the persons who have 'sailing' as a hobby:

```
                                create function sap(parent p)
                                  -> string as
select p, name(parent(p))       select name(parent(p))
from person p                   from person p
where hobby(p) = 'sailing';     where hobby(p) = 'sailing';
```

On the right above a derived function is defined that retrieves the names of the parents of a person if the person has 'sailing' as a hobby. The difference between the query and the function is that in the function the variable *p* is *bound*, i.e. given a specific value, while in the query it is *unbound* (i.e. free, ranging over the whole *Person* extent). A vector denoting the binding used for each argument and result variable during a function invocation is named a *binding pattern*. A derived function can be executed with different binding patterns, granted that the functions in its body are executable with the used binding pattern [21]. The following query demonstrates how the function defined above can be used to retrieve the children of 'John Doe' having 'sailing' as a hobby. Note that in this case the result of the function is bound while the argument is not bound:

```
select p
from person p
where sap(p) = 'John Doe';
```

The query processing in AMOS*II* (figure 2), first translates the AMOSQL queries into a type annotated *object calculus* representation. For example, the result of the calculus generation phase for the first query above is given by the following calculus

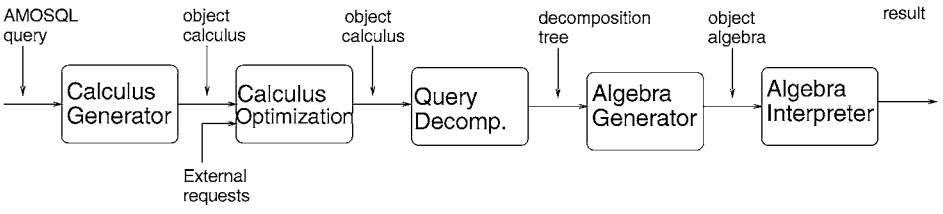

Figure 2. Query processing in AMOS*II*.

expression:

$$\{p, nm \mid$$
$$p = Person_{nil \rightarrow person}(\ ) \wedge$$
$$d = parent_{person \rightarrow person}(p) \wedge$$
$$nm = name_{person \rightarrow charstring}(d) \wedge$$
$$\text{`}sailing\text{'} = hobby_{person \rightarrow charstring}(p)\}$$

The first predicate in the expression is inserted by the system to assert the type of variable p. It defines the variable p to be member of the result of the extent function for the type *Person*. In case of a derived type, the extent function contains a query defining the extent in terms of predicates over the supertypes. The extent function can be used to generate the extent of a type, as well as to test if a given instance belongs to a type. Therefore, a predicate containing a reference to an extent function is called a *typecheck predicate*.

In the second processing phase, the calculus optimizer expands the derived functions bodies (including the derived types extent functions) and applies type-aware rewrite rules to reduce the number of predicates [7, 17].

After the rewrites, queries operating over data outside the mediator are decomposed into single-site query fragments to be executed in different AMOS*II* servers and data sources. This *query decomposition* process is the main topic of this paper. The algebra generation phase translates the formalism used during the query decomposition into an executable object algebra representation, accepted by the object algebra interpreter.

An interested reader is referred to [17] for more detailed description of the data integration framework of AMOS*II*, and to [7, 9, 10, 15, 21] for more comprehensive descriptions of the query processing phases preceding the query decomposition.

3. Related work

The work presented in this paper is related to the areas of data integration and distributed query processing. This section references and briefly overviews some representative examples of projects in these areas. A more elaborate comparison of the AMOS*II* system with other data integration systems is presented in [15].

One of the first attempts to tackle the query optimization problem in a distributed databases environment was done within the System R* project [3]. In that project an exhaustive, centrally performed query optimization is used to find the optimal plan. Due to the problem size, in our work we use a search strategy that performs an exhaustive strategy over only a portion of the whole search space. This method is combined with several heuristic-based phases that improve the plan and reduce the optimization time.

As opposed to the distributed databases, where there is a centralized repository containing meta-data about the whole system, the architecture described in this paper consists of autonomous systems, each storing only locally relevant meta-data. Most of the mediator frameworks reported in the literature [4, 6, 8, 12, 13, 22, 24, 28] propose centralized query compilation and execution coordination. In the Pegasus project [6] it is indicated that a distributed mediation framework is a promising research direction, but to the extent of our

knowledge no results in this area are reported. A centralized query processor in Pegasus identifies portions of the query tree that can be evaluated in a single source and converts them into *Virtual Tables* (VTs). It then generates a left-deep query tree having VTs or locally stored tables as leaves and internal nodes representing operators of extended relational algebra. Finally, re-balancing is performed over the left-deep tree based on the associativity and commutativity properties of the join and cross-product operators. In AMOS*II* we propose a fully distributed query compilation and tree re-balancing where more than one participating mediators can compile multi-source portions of the submitted query.

In DISCO [28], the query processing is performed over plans described in a formalism called universal abstract machine (UAM) that contains the relational algebra operators extended with primitives for executing parts of query plans in the wrappers. The mediator communicates with the wrapper by using a grammar describing the operator sequences accepted by the wrapper. It can also (in some cases) ask for the cost of a particular operator sequence. This method is more elaborate than the method for the description of data source capabilities in AMOS*II*, but it is more complex and time-consuming, due to the combinatorial nature of the problem of constructing the sub-plans executed in the wrappers.

The Garlic system [13] has also a centralized wrapper-mediator architecture. Decomposition in Garlic is achieved by performing rule-based transformation of the initial algebra query representation that can shift a part of the plan to be executed in the wrapper/source. As opposed to the approach presented in this paper, the query processor in Garlic does not have any knowledge about the capabilities of the individual sources. Therefore, in order to find out if a sub-query is executable in a data source the query processor must ask the corresponding wrapper. This approach combined with an exhaustive enumeration leads to a large number of sub-query compilation requests submitted to the wrapper. Common subexpression analysis is not used to reduce the amount of data retrieved from the source. Garlic supports a join implementation named *BindJoin* that sends parameters to the sources, but only on a single tuple level. Finally, Garlic does not store data locally and does not support functions that are implemented in more than one data sources, as described in this work.

The DIOM project [26], proposes a distributed framework for integration of relational data sources where the relational operators can be executed either in the mediators or in the data sources. The query optimization strategy used in DIOM first builds a join operator query tree (schedule) using a heuristic approach, and then assigns execution sites to the join operators using an exhaustive cost-based search. AMOS*II*, on the other hand, performs a cost-based scheduling and heuristic placement. Furthermore, the compilation process in DIOM is centrally performed, and there is no clear distinction between the data sources and the mediators in the optimization framework. Finally, in DIOM only the join operator can be scheduled to be executed at more than one site, lacking the flexibility of the multiple implantation functions mechanism in AMOS*II* where this applies to arbitrary user-defined operators and functions.

4. Query decomposition in AMOS*II*

Given a query over multiple data sources, the goal of the query decomposition is to determine the portions of the original query, *query fragments*, executed at each of the sites, and

to schedule their executions to produce the query result. As noted earlier, AMOS*II* is a distributed mediator system. This implies a decomposition framework that allows cooperation of a number of distinct AMOS*II* servers during the query processing. While distribution is present in any mediation framework due to the distribution of the data sources, the distributed mediator framework in AMOS*II* introduces another higher level of interaction among the AMOS*II* servers. In other words, an AMOS*II* server does not treat another AMOS*II* server as just another data source. More specifically, if we compare the interaction between an AMOS*II* server and a wrapper (and through it with a data source), and the interaction between two AMOS*II* servers, there are two major differences:

- AMOS*II* can accept compilation and execution requests for query fragments over data in more than one data source, as well as data stored in the local AMOS*II* database. The wrapper interfaces accept fragments that are always over data in a single data source.
- AMOS*II* supports materialization of intermediate results used as input to locally executed query fragments and generated by a fragment in another AMOS*II* server. A wrapper provides only *execute* functionality for queries to the data source. By contrast, the query execution interface of AMOS*II* provides a *ship-and-execute* functionality that first receives and stores locally an intermediate result, and then executes a query fragment using it as an input.

4.1. Motivation and overview

Two query execution plans are equivalent if, for any state of the database, they produce the same result. Equivalent plans can have very different execution times (costs). In order to avoid unnecessary large execution times, during the query decomposition process many different, but equivalent plans are considered. To illustrate the magnitude of the number of equivalent query execution plans for a query over multiple data sources, we use a query plan representation as n-ary operator trees. A straight forward query decomposition would estimate the cost of each plan tree and then select the cheapest plan. Note that such an exhaustive method is not used in AMOS*II*; it is used here solely to demonstrate the enormous search space of the optimization problem. Each tree node in such an operator tree contains a simple predicate from the query calculus expression, and is assigned a data source for execution. Some predicates can be executed at more than one data source. A tree is executed by first executing the root node children, then shipping the results to the execution site (data source) where the root node is assigned, and finally executing the root node predicate. Since in a non-trivial case the number of possible n-ary trees with p nodes is exponential to p, and the number of different site assignments is exponential to the number of predicates executable at more than one data source d, the total number of trees is $O(a^p \cdot s^d)$, where a is a constant and s is the number of sites involved.

This estimate shows that an exhaustive search of the whole search space is not feasible. Therefore the decomposition strategy presented in this paper is based on a combination of cost-based search strategies and heuristic rules that lower the number of the examined alternatives. The aim is not to find an optimal plan. Instead, we strive to to produce a "reasonably" efficient plan for most of the queries, while exploring only a limited portion of the search space.

The description of the query decomposition in this section assumes conjunctive predicate expressions as input. The query decomposer handles disjunctions in two ways, depending on the origin of the predicates in the disjuncts:

– *Single source disjunctions* contain predicates that are executed at a single data source. They are treated as a single predicate with the cost, the selectivity and the binding pattern induced form the disjuncts.
– *Multiple source disjunctions* are handled by normalization into disjunctive normal form. The decomposer then processes each disjunct in the normalized query separately.

From a parsed and flattened query calculus expression the query decomposer produces a *query decomposition tree*. The parsing and the flattening of multi-database queries is not different from queries over local data [21]. The query decomposition process is performed in four phases as illustrated in figure 3.

The decomposition produces a query execution plan where query fragments are executed in various data sources having different query processing capabilities. This mismatch in the capabilities of the sources has a great impact on the query decomposition. Therefore, in order to give a basis for the detailed description of the decomposition phases

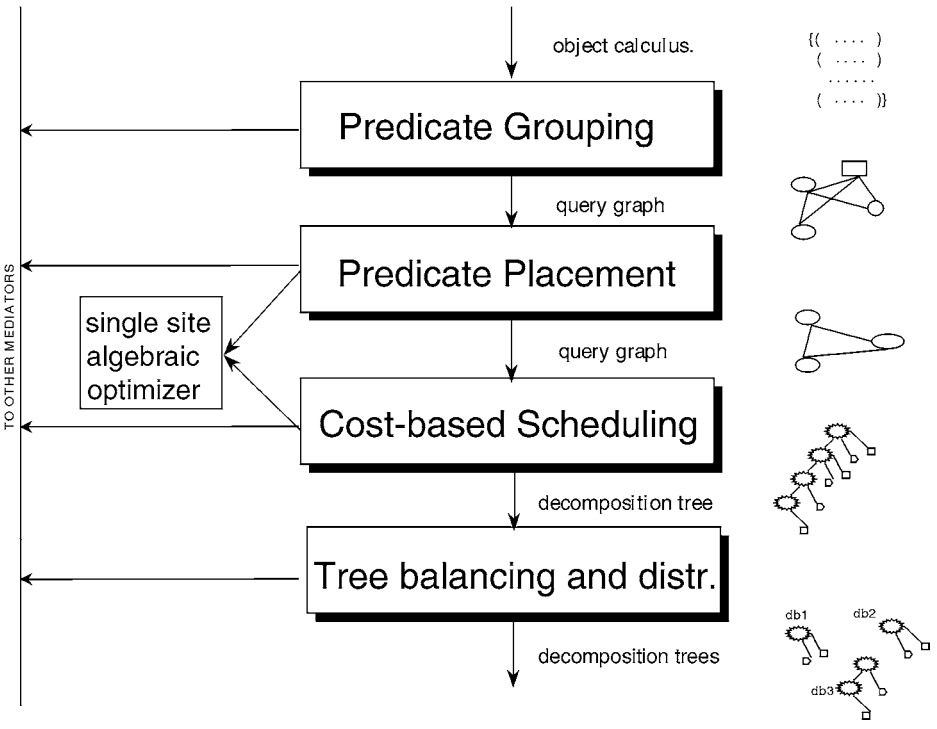

Figure 3. Query decomposition phases in AMOS*II*.

later in this section, we first present the AMOS*II* model for description of data source capabilities.

4.2. Describing the data source capabilities

The functions used in an AMOSQL query can be divided into two categories: functions implemented and executable in exactly one data source, *single implementation functions* (SIFs); and functions that are implemented and can be executed in more than one data source, *multiple implementations functions* (MIFs).

The user-defined local functions as well as the proxy functions are SIFs. For example, if a function $name_{Person \rightarrow string}$ is defined over the instances of the type *Person* in a source named *EMPLOYEE_DB*, then the implementation of this function is known only in that mediator and therefore it can be executed only there. An example of MIFs are the comparison operators (e.g. $<$, $>$, etc.) executable in AMOS*II* servers, relational databases, some storage managers, etc. The MIFs can also be user-defined. Since, in our framework, each user-defined type is defined in only one data source, a MIF may take only literal-typed parameters. A framework that would support replicated user-defined types and MIFs taking user-defined type parameters would require that the state (values) of the instances of these types is shipped between the mediators, in order to be used at the data source where the MIF is executed. In the framework presented here, only OIDs and the needed portions of the state of the instances are shipped among the mediators and the data sources. Extending the integration framework to handle replicated user-defined types is one of the topics of our current research. To the extent of our knowledge, such features are not supported by any of the mediator prototypes for data integration reported in the literature.

Depending on the set of MIFs that a data source implements, the sources are classified into several *Data Source Kinds* (DSKs). Inversely, the set of MIFs associated with a DSK is a *generic capability* of this DSK; the DSK *supports* the MIF. Besides a generic capability, each data source instance can also have *specific capability* defined by the user-defined types and single implementation functions exported to AMOS*II*. To simplify the presentation, in the rest of this section we use the term *capability* to refer to to the generic capability of a DSK.

In order to reuse the capability specifications, the DSKs are organized in a hierarchy where the more specific DSKs inherit the capability of the more general ones. This hierarchy is separate from the AMOS*II* type hierarchy and is used only during the query decomposition as described below. Figure 4 shows an example of an AMOS*II* DSK hierarchy. All DSK hierarchies are rooted in a node representing data sources with only the basic capability to execute simple calculus predicates that invokes a single function/operator in this source and returns the result to the mediator. Data sources of this kind cannot execute MIFs. At the next capability levels, DSKs are defined that have the capability to perform arithmetic, comparison and join operations. The arithmetic and comparison DSKs are defined using the usual set of operators, shown in the figure.

The two kinds join capabilities *single collection join* and *general join*, are not specified using MIFs as for the other DSK capabilities. In the calculus used in AMOS*II*, the equi-joins are represented implicitly by a variable appearing in more than one query predicate.

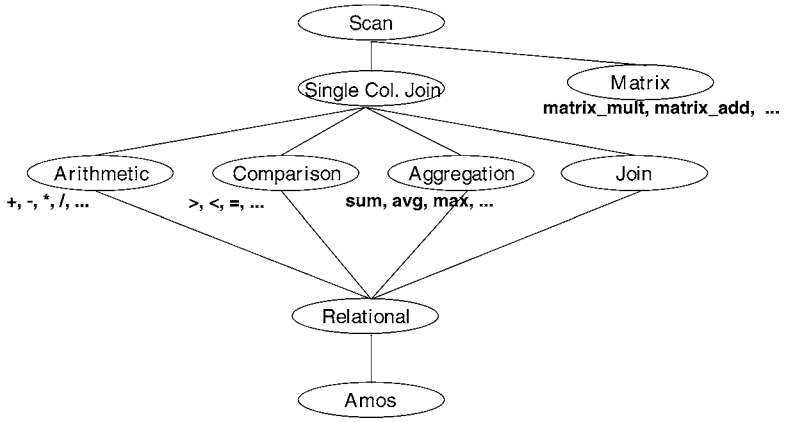

Figure 4. Data source capabilities hierarchy.

Accordingly, a wrapped data source with a join capability can process several predicates connected by common variables as a single unit. Based on the properties of the commonly used data sources, there is a need to distinguish between two kinds of join capabilities. First, there are sources that are capable of combining and evaluating conditions over only a single data collection in the source (e.g. a table in a storage manager). These kinds of sources are defined using a DSK that inherits only the *single collection join* capability. An example of such a data source is a storage manager storing several data tables, each represented in the mediator as a proxy type. Each table can be scanned with associated primitive conditions. The conditions to a single table can be added together. However, operations over separate tables need to be submitted separately. Therefore, for each table, the MIF operators are grouped together with the proxy type typecheck predicate, and submitted to the wrapper. One grouped predicate is submitted for each different collection. A system with such features fits the capability description of the *comparison* DSK in figure 4.

The general join capability is inherited by DSKs representing sources capable of processing joins over multiple collections (e.g. relational database sources). The decomposer sees each collection as a proxy type. It combines the operations over several proxy types into a single query fragment that is sent to join capable data sources.

New DSKs are defined by inserting them into the DSK hierarchy. For example a DSK representing a software capable of matrix operations is named *Matrix*, and placed under the DSK hierarchy root node in figure 4. This implies that matrix sources supports the execution of one operation at a time. A DSK allowing combining of matrix operations would have been defined as a child of one of the join DSKs.

4.3. Predicate grouping

The predicate grouping phase attempts to reduce the optimization problem by lowering the number of the predicates considered in the optimization. In this phase, if possible, the

individual predicates executed at a given data source are grouped into one or more query fragments, each represented by a composite predicate. The query fragments are treated afterwards as a single predicate. Each query fragment is optimized in the wrapper or the data source where it is forwarded for execution. Each fragment is parameterized with the calculus variables appearing both in the associated composite predicate and the rest of the query.

Two major challenges arise in grouping the query predicates into fragments:

- *Grouping heuristic*: an exhaustive approach to the grouping would not reduce the query optimization problem. A heuristic approach must be used.
- *Grouping of the MIF predicates*: how to group the predicates that reference multiple implementation functions from DSKs.

Since exploring all the possible groupings of the predicates into query fragments is not feasible AMOS*II* applies the following two grouping rules:

- Joins are pushed into the data sources whenever possible.
- Cross-products are avoided.

Using these two heuristic rules results into good query plans under certain assumptions about the input queries. The first rule is based on the observation that usually the result of a join is not orders of magnitude bigger than the operands, and that the sources that support joins might take advantage of auxiliary structures (e.g indeces) to perform the join faster than the mediator. By favoring this approach we also avoid storing intermediate results in the mediator. Sources for which it can be a priori determined that this properties does not hold can be declared of a DSK that does not have the general join capability which will prevent joins in the source. The second rule favors performing cross product operations in the mediator instead of in the sources. Assuming that the time to complete two separate accesses to the data source will not be substantially larger than the time to complete one, this approach should always lead to a better query execution plan.

The grouping process in AMOS*II* uses a *query fragment graph* that is similar to the query graphs in the centralized database query processors. Each graph node represents a query fragment. Initially each fragment composes of a single query predicate. Fragments containing one or more common variables are connected by an edge. Each edge is labeled with the variables it represents. The variables labeling the edges connecting a fragment with the rest of the graph represent the *parameters* of the fragment.

Query fragments composed of predicates referring to only SIFs are named *SIF fragments*. Analogously, the fragments composed of MIF predicates are *MIF fragments*. All fragments in the graph are assigned to a site.[2] The SIF fragments always contain SIF references from a single site. Accordingly, each SIF fragment is assigned to its corresponding site. The MIF fragments are assigned to a site in the later decomposition phases. Each fragment is also associated with a DSK. SIF fragments are given the DSK of the site where they are assigned. The MIF fragments are attributed DSK based on the functions referenced in the fragment. Initially each fragment contains only one calculus predicate, referencing only one MIF

function. Therefore the fragment can be unambiguously attributed the DSK containing the predicate function in its capability set.

The grouping of the query fragment graph is performed by a series of *fusion* operations that fuse two fragments into one. The result of a fusion is a new fragment representing the conjunction of the predicates of the fused fragments. The new fragment is connected to the rest of the graph by the union of the edges of the fused fragments. To preserve the unambiguous assignment of DSKs to MIF fragments, only MIF fragments associated with the same DSK are fused. Furthermore, the DSK of the fused MIF fragments must have at least a single collection join capability for a fusion to be applicable. SIF fragments are fused only with other SIF fragments associated with the same site, according to the following conditions, based on the site capabilities:

- *Site without join capability*: Fragments to be executed at this kind of sites are not fused and contain only a single predicate. The system will add typecheck predicates for all the variable used in the predicate, to provide typing information that might be needed for translation and function resolution in the wrapper.
- *Single collection joins site*: Two fragments that are to be executed at a site capable of only single collection joins fused if they represent operations over the same collection in the source, represented by a proxy type in the query.
- *General join site*: Two connected SIF query fragments executed at such a site are always fused.

Assuming a query fragment graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$, where $\mathcal{N} = \{n_1 \ldots n_k\}$ is a set of nodes (query fragments), and $\mathcal{E} = \{(n_1, n_2) : n_1, n_2 \in \mathcal{N}\}$ is a set of the edges between the nodes, the predicate grouping algorithm can be specified as follows:

while $\exists\ (n_i, n_k) \in \mathcal{E} : n_i$ *and* n_k *satisfy the fusion conditions* **do**
$n_{ik} := \mathit{fuse}(n_i, n_k)$;
$\mathcal{E} := \mathcal{E} - \{(n_i, n_k)\}$
$\mathcal{E} := \mathcal{E} \cup \{(n_{ik}, n_m) : (\exists (n_l, n_m) \in \mathcal{E} : n_l = n_i \vee n_l = n_k) \vee$
$(\exists (n_m, n_l) \in \mathcal{E} : n_l = n_i \vee n_l = n_k)\}$;
$\mathcal{E} := \mathcal{E} - \{(n_i, n_m)\} - \{(n_m, n_i)\} - \{(n_k, n_m)\} - \{(n_m, n_k)\}$;
$\mathcal{N} := \mathcal{N} - \{n_i, n_k\} \cup \{n_{ik}\}$;
end while

After each fusion, the fused fragments are replaced in the graph by the new fragment, and all the edges to the original fragments are replaced by edges to the new fragment. The *fuse* operation conjuncts the query fragments and adjusts the other run-time information stored in the fragment node (e.g. typing and variable information) to reflect the newly created query fragment. The algorithm terminates when all possible fragment fusions are performed.

After performing all the possible fusions, the graph contains fragments that are to be submitted to the data sources as a whole. However, this is not the final grouping. The grouping is performed again after the MIF fragments are assigned sites (to be discussed below). Note that MIF fragments of different DSKs are not grouped together at this stage.

At this stage the graph contains query fragments composed of either only MIF predicates of a same DSK, or only SIF predicates.

The following example, used as a running example through the rest of this section, illustrates the grouping process. The query below on the left contains a join and a selection over the type A in the source $DB1$, and the type B in the source $DB2$. Two functions are executed over the instances of these types: $fa_{A\rightarrow int}(\ )$ in DB1, and $fb_{B\rightarrow int}(\ )$ in DB2. The calculus generated for this query is shown on the right:

```
select res(A)
from A@DB1 a, B@DB2 b
where fa(a) + 1 < 60 and
fa(a) < fb(b);
```

$$
\begin{aligned}
&\{\, r \mid \\
&\quad a = A_{nil\rightarrow A}(\ ) \wedge \\
&\quad b = B_{nil\rightarrow B}(\ ) \wedge \\
&\quad va = fa(a) \wedge \\
&\quad vb = fb(b) \wedge \\
&\quad va1 < 60 \wedge \\
&\quad va1 = plus(va, 1) \wedge \\
&\quad va < vb \wedge \\
&\quad r = res(a)\ \}
\end{aligned}
$$

The example query is issued in an AMOS*II* mediator and is over data stored in the data sources *DB1* and *DB2*. In the example, we will assume that these two sources have *Join* capability (e.g. relational databases or AMOS*II* servers). In the initial query fragment graph, shown in figure 6(a), each node represents a single-predicate query fragment. The nodes (query fragments) are numbered with the rank of the predicates in the above calculus expression. In figure 6(a), the predicates are shown beside each graph node. The nodes are also labeled with their assigned site, or with "MIF" if they represent a MIF fragment. The graph edges are labeled with the variables that connect the fragments.

Figure 6(b) shows the result of the grouping phase. The fragments n_8, n_1 and n_3 are all assigned to the site *DB1* and make a connected subgraph; therefore they are fused into the fragment:

$$a = A_{nil\rightarrow A}(\ ) \wedge va = fa(a) \wedge r = res(a)$$

The same applies for n_4 and n_2 at *DB2*. Although, n_6 is connected with both n_5 and n_7, these MIF fragments cannot be fused because they correspond to different DSKs: *arithmetic* and *comparison* respectively.

4.4. MIF fragments execution site assignment

The graph produced by the previous phase contains SIF fragments with an assigned execution site, and MIF fragments that are still not assigned to a site. In order to generate query fragments for the individual data sources, the next step is to assign execution sites to the MIF fragments. A MIF fragment can be executed at any site known to the mediator that is capable of performing the operations (functions) referenced in the fragment. Furthermore, it could be beneficial to assign (replicate) a MIF fragment to more than one execution site where it is combined and translated, together with the other query fragments assigned to

this site into an execution plan that is cheaper to execute. Because of the declarative nature of the query fragments, their replication does not change the result of the query execution. Any assignment of execution site(s) to a MIF fragment yields a correct and executable query schedule; the difference is only in the costs (execution times) of the generated plans.

Searching the space of possible site assignments using an exhaustive strategy would require examining every combination of known sites as execution sites for each MIF fragment. This would require performing full query optimization for each alternative using backtracking, resulting ultimately in an algorithm with an exponential complexity. To avoid this expensive process, we tackle the site assignment problem for MIF fragments by using a heuristic approach aided, in certain cases, with partial cost estimates.

The heuristic used in AMOS*II* is based on analysis of the execution costs affected by the placement of a MIF fragment at a site. These costs are:

- The cost of the execution of the MIF fragment at the assigned site.
- The change of the execution costs of the other query fragments assigned at the same site (due to additional applicable optimization).
- The intermediate results shipment cost.

The first cost varies due to different speeds of the sites in the federation. The cost of the execution of other predicates can change when a MIF fragment is combined with a SIF fragment placed at the same site, as for example, when the newly assigned MIF fragment contains a selection condition that reduces the SIF fragment execution time in the data sources. Finally, this kind of a selection will also influence the size of the intermediate results.

In order to simplify the placement problem, we recognize several different sub-cases and in each one examine only some of the above costs. In each case, the following goals are pursued in the order they are listed:

1. Avoid introducing additional cross-site dependencies among the fragments, caused by having a single calculus variable appearing at multiple sites. These dependencies often lead to costly distributed equi-joins by shipment of intermediate results among the sites.
2. Place each MIF fragment so that it has common variables with one or more SIF fragments, in order to facilitate possible rewrites that can reduce the cost of accessing the data sources and reduce the intermediate results sizes.
3. Reduce the execution time for the individual MIF fragments.
4. When it is not possible to assign a site to a MIF fragment on the basis of the previous three criteria, if possible, execute the predicate in the mediator where the query was submitted.

The placement algorithm does not attempt to satisfy these goals simultaneously, but rather tries to satisfy one at the time in the order they are listed above.

While the set of the possible execution sites for a MIF predicate is fixed (all the sites having the required capability), and does not depend on the order in which the MIF fragments are placed, the placement heuristics presented below uses the already placed fragments as a source of information for placing the next fragment. Different placement order can therefore

provide different available information at fragment placement time. When considering a site for an execution of a query fragment, the available information is larger when more of the other MIF fragments that will be also placed at the same site are already placed. To maximize the information available during the fragment assignment, the query fragments requiring more capability are placed before the fragments requiring less capability. A fragment that requires less capability is always assigned to a site that is also considered when a fragment requiring more capability is assigned. Hence, if a fragment requiring less capability is assigned before a fragment requiring more capability, the placement heuristics can use the information in the former to decide on the placement of the later. This is not always true in the opposite direction, because the fragment requiring less capability might be assigned to a site that does not have the capability to process the fragment requiring more capability, making the information gained by the former placement unusable during the consequent placement of the later. Therefore, the fragments are placed in an order induced from the partial order of their capabilities where fragments with more capabilities are placed before the fragments with a subset of their capabilities.

After a MIF fragment is placed at an execution site, the new graph is regrouped by the grouping algorithm in order to group the newly assigned fragment with the fragments already assigned to the chosen site.

The site assignment process works as follows. First, each calculus variable that labels an edge in the graph is assigned the set of sites where it appears, i.e. the set of sites of the fragments that are connected by a graph edge labeled with this variable. Since this variables represent parameters for the query fragment, this set is referred to as *parameter site set*. Next, each of the MIF fragments is processed. For each fragment, first an intersection of the site sets of the fragment parameters is computed. This intersection represents the sites that operate over the same data items as the MIF fragment.

Figure 5 shows five sub-cases of the placement problem, distinguished by the properties of the parameter's site sets intersection and the query fragment. The rest of this section examines each of the cases in greater detail.

Case 1: Singleton site sets intersection. If the intersection is not empty and contains only one site, then the fragment is assigned to that site. This allows the optimizer to devise a strategy where no intermediate results are shipped between the mediators when the query

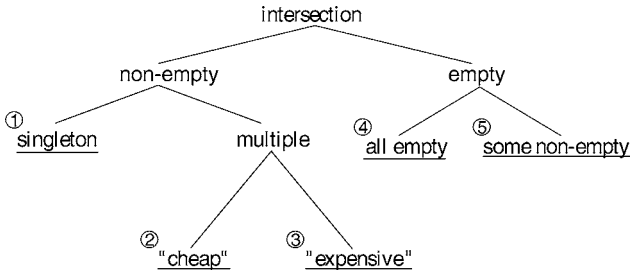

Figure 5. MIF fragment clasification using parameter site sets and execution costs.

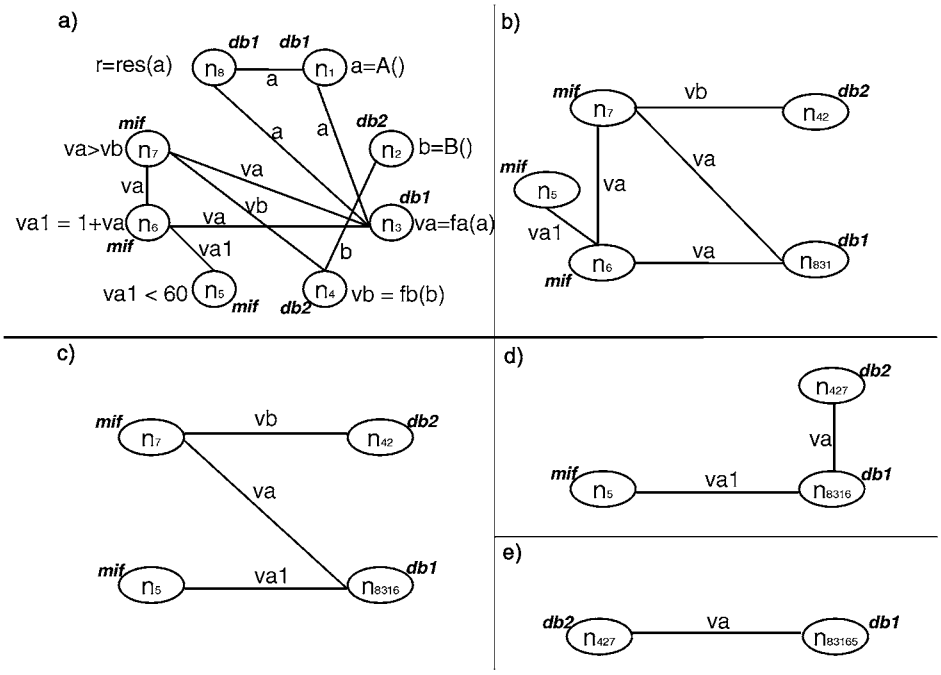

Figure 6. Query graph grouping sequence for the example query.

fragment is executed. All the parameters can be produced locally at the chosen site. Placing the query fragment at a site where only a subset of the needed parameters can be produced implies that the missing parameters must be shipped in before the fragment is executed. An example of a Case 1 placement is shown in figure 6(b) where fragment 6 is connected only by the variable *va* to fragment 831. This fragment is assigned to the same site as 831, i.e. *DB1*. After the grouping of the graph the result is as presented in figure 6(c).

Cases 2 and 3: Several sites in the parameter site sets intersect. In this case, MIF fragments are placed on the basis of their execution costs when all the parameters are bound. If a fragment has a cost lower than a predetermined low constant threshold, then the it is considered to represent a cheap selection. The query fragment is therefore replicated, placing one copy at each of the sites in the intersection. This strategy is particularly useful for query processing in multi-database environments. In a classical distributed database environment, it would suffice to execute the selection at only one site. The query processor could then ship the intermediate result to the other sites, and use this already reduced set of instances as the inner set in the joins. That strategy is not possible when data sources do not support materialization of intermediate results. Thus, the selections should be pushed into all the applicable data sources to reduce the processing times in the sources, as well as minimizing proxy object generation in the wrappers associated with these sources.

Case 4: All parameter site sets empty. A parameter has an empty site set if it appears only in MIF fragments that have not yet been placed. If all site sets of the fragment parameters are empty, assuming a connected query graph, we can conclude that all the neighboring fragments are also unplaced. In order to obtain more information for the placement of such fragments, the placement is postponed and the fragment is omitted. Omitted fragment are processed after processing the rest of the fragments. If all MIF fragments have all parameter site sets empty, the first fragment is placed for execution in the mediator where the query is issued, if possible. Otherwise, it is placed at the site where it will be executed fastest, i.e. at the most powerful site.

Assuming, that the site assignment proceeds in the same order as the nodes are numbered, in the situation shown in figure 6(b) the algorithm will attempt to place n_5. Since n_5 is connected to only MIF fragments, its parameter site sets intersection is empty. Thus, n_5 is skipped as described above, and considered again when the rest of the MIF fragments are placed. The graph at this point is presented in figure 6(d). Now, the site set of the parameter *va1* is $Aset_{va1} = \{DB1\}$ since n_5 is connected to n_{8316} at *DB1*, by an edge labeled *va1*. Fragment n_5 is therefore placed at *DB1*. After the grouping, the final query graph is shown in figure 6(e).

Case 5: Non-empty site sets with empty intersection. In this last case, we consider placing a query fragment having a non-empty intersection of its parameters' site sets, but not all of the sets are empty. In other words, there is no site that can produce all the parameters needed for the execution of the fragment, while all of the parameters are present at at least one site. The placement process in this case is based on a simplified cost estimate. The estimate calculation takes into account only the query fragments that are neighbors in the query graph to the currently placed fragment. Moreover, the cost estimate is calculated by taking into account only the graph edges of the currently processed query fragment. Another simplification of the problem is that this kind of query fragments are placed at exactly one site. Since no site contains all the data needed for the execution of the query fragment, the missing data must be shipped to the execution site from other sites. By placing the query fragment at one site, we avoid plans where the parameters for the MIF fragment are shipped to multiple sites.

For each of the possible placements, the sum of the execution costs of the predicates in the neighboring query fragments and the necessary data shipment is estimated. The predicate is placed at a site where this cost is the lowest. To describe the calculation of this cost, let the set of the neighboring fragments be $N = \{n_1^{s_{n_1}}, \ldots, n_l^{s_{n_l}}\}$; the set of sites these fragments are placed at $S = \{s_1, \ldots, s_m\}$, $m \leq l$; the currently placed fragment parameter variables $A = \{a_1, \ldots, a_k\}$; and finally, the corresponding parameter site set of each of these variable: $As = \{aSet_1, \ldots, aSet_k\}$.

The execution cost of all the query fragments at site s, assuming that each fragment is executed over *BS* (bulk size), tuples is defined as the sum of the costs of the execution costs of the individual query fragments:

$$exec\_cost(s, BS, A) = \sum_{j=1\ldots l, s=s_j} cost\left(n_j^{s_{n_j}}, BS, A\right)$$

The *cost* function returns the cost of executing a fragment with the parameters in the set A unbound (i.e. emitted as a result), so that they can be consumed by the currently placed fragment. In calculating the estimate, the number of input tuples is fixed to a predetermined constant BS that denotes the size in tuples of the bulks of intermediate results shipped between the AMOS*II* servers during the streamed execution. This number is used since the size of the results cannot be precisely estimated before all the fragments are placed and scheduled for execution. Using such a constant value for the estimates provides a good basis for comparison, however, it is important that this constant is larger than 1 in order to correctly estimate the effect of techniques as sub-query materialization in queries containing nested sub-queries. In such cases, the query processor might decide to materialize the sub-query result and use it in the processing of the whole input. The cost of the materialization is amortized over the processing of all the input tuples and therefore:

$$cost\left(n_j^{s_{n_j}}, BS, A\right) \neq BS \cdot cost\left(n_j^{s_{n_j}}, 1, A\right)$$

Nested sub-queries are common in the system-generated functions for support of the AMOS*II* OO view system spanning over multiple mediators and data sources [16], making this kind of cost estimate necessary.

When a site s is chosen for a query fragment, the grouping algorithm is applied to the neighbor subgraph nodes placed at this site. The sum of the execution costs of the all fragments at this site is denoted with $pa\_exec\_cost(s, BS)$. Assuming that a subset A_l of the parameter set A is produced locally at s while the rest of the parameters $A_t = A - A_l$ are shipped from the neighboring fragments, the execution cost estimate can be expressed as a sum of the new cost at the site where the fragment was placed and the unchanged costs at the other sites:

$$ece(s) = pa\_exec\_cost(s, BS) + \sum_{i=1\ldots l, s_i \neq s} exec\_cost(s_i, BS, A_t)$$

To obtain a complete cost estimate, besides the execution cost estimate, we need to compute an estimate for the intermediate results shipping cost. Here, we assume that each of the missing parameters in A_t is shipped to the site s from the cheapest possible alternative. The cost of shipping the parameter $a_i \in A_t$ from a site r where it is produced by the query fragments in N to a site s where it is consumed is:

$$\begin{aligned} tec(a_i, N, S) = {} & BS \cdot selectivity(N, A_t) \cdot sizeof(type(a_i)) \cdot WB_{rs} \\ & + selectivity(N, A_t) \cdot WI_{rs} \end{aligned}$$

Where WB_{rs} is the weight of the cost of the network link between the sites r and s per byte and WI_{rs} is the message initiation cost (we assume that each message contains BS tuples); $selectivity(N, A_t)$ returns the selectivity of the query fragments in N with all parameters in A_t unbound; *sizeof*( ) returns a size of a given tuple of types; and *type*( ) returns a tuple of data types for a given tuple of variables. The unit of cost in AMOS*II* is the time to execute a single access in the main-memory storage manager. The network weights WB_{rs} and WI_{rs}

represent the ratio between this unit and the time to ship one byte between the sites R and S and to initiate a message respectively. The parameter shipping cost can be expressed as:

$$tec(S) = \sum_{a_i \in A_t} \min_{n_i \in N} tec(a_i, N, S)$$

The complete cost estimate for placing the fragment at the site s is:

$$ce(s) = ece(s) + tec(s)$$

The fragment is assigned to the site so such that

$$\forall s \in S \; ce(so) \leq ce(s)$$

Although all the possible site assignments produce a correct execution plan, the cost estimate calculation can fail for some sites because some of the query fragments might not be executable with the incomplete binding patterns used to calculate the estimate. Such sites are ignored in the assignment process. In a rare case, it is possible that all the estimate computations fail. In this case, an arbitrary site is chosen from the set of sites capable of handling the query fragment.

In order to determine the complexity of the cost estimate calculation we can observe that the terms used in the equations above can all be obtained either from the system catalogue (e.g. *sizeof*( ) function and the network weights), or from compilation of the fragments in the subgraph (the *cost*( ) and *selectivity*( ) functions). The maximum number of compilations needed to obtain this data is $2l$, where l is the number of neighboring fragments of the fragment being placed in the query graph. This estimate is based on the observation that each neighboring fragment is compiled twice: once for the case when the fragment is placed at the same site with the neighboring fragment, and once when it is placed elsewhere. Normally, the queries posed to the mediator have connected query graphs, implying that $l \leq n$, n being the number of sites involved in the query. Hence, the cost of the site assignment will usually not be larger than $2n$ single site query fragment compilations, some of which might be reused in the latter decomposition phases. We also note that n here does not represent all the sites involved in the query, but rather the sites that operate over the parameters of the placed fragment.

In figure 6(c) the fragment n_7 represents an example of case 5 placement problem. The example illustrates the problem of the placement of the join condition $va < vb$. Figure 6(d) shows the graph after placing n_7 at $DB2$.

A more elaborate example of this case is illustrated by the query graph shown in figure 7(a). The upper right side of the figure shows the sets of fragments, sites, parameters and parameter site sets used in the calculations of the estimates. There are three sites involved with a total of 4 fragments. Assuming *Join* capability, the resulting grouped graphs for each placement alternative are shown in the figure 7(b–d). Depending on the costs and selectivities of the fragments the optimizer chooses one of these alternatives.

This concludes the description of the query decomposition phases that produce the query fragments sent to the individual data sources. The following sections will present a method

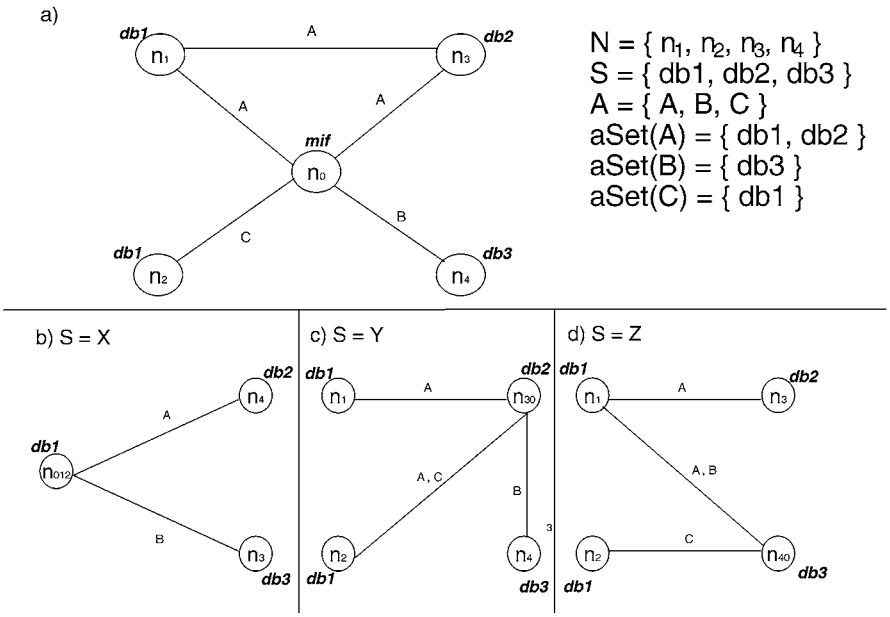

Figure 7. Case 5 example and the possible outcomes.

for scheduling the execution of these fragments and assembling the result to the query from the results of the query fragments.

4.5. Cost-based scheduling

The result of the first two query decomposition phases is a query fragment graph where each fragment is assigned an execution site. The fragments are connected by edges representing equi-joins over the values of common variables. To translate this query graph into an executable query plan, the query processor must decide on the order of the execution of the query fragments. This order influences the data flow between the sites. The query processor builds an *execution schedule* to describe the execution order and the flows between the sites.

The query fragments are represented by system generated derived functions with a signature based on the data types of the parameters. When a fragment is executed in a source that is not an AMOS*II* mediator, the derived function is defined in the mediator that wraps the source. The bodies of derived function representing non AMOS*II* query fragments are generated by the corresponding wrappers. Usually, these call foreign functions implemented in C, Java or Lisp that access the data source and perform the requested operations. For example, the relational wrapper implemented within the AMOS*II* project creates an SQL statement from the object calculus, and then invokes the foreign function *sql* [1] that passes an SQL statement to an ODBC data source. The *sql* function is called from within

the calculus expression generated for the query fragment and is contained in the derived function body. In addition to this expression, the function body can contain expressions for generating OIDs based on the data retrieved from the relational database.

Examining all the possible execution schedules is not feasible for larger queries. To illustrate the hardness of this problem, we examine the alternative execution schedules for the query example from the previous subsection. The final query graph for this query contains two query fragments, each at one of the two participating sites. The definition of the derived functions representing the query fragments are as follows (ignoring the source specific translations):

in DB1:

$QFdb1_{type\_va \rightarrow boolean}(va) \Longleftrightarrow$
{
$b = B_{nil \rightarrow B}(\ ) \wedge$
$vb = fb(b) \wedge$
$va < vb\}$

in DB2:

$QFdb2_{type\_r,type\_va \rightarrow boolean}(r, va) \Longleftrightarrow$
{
$a = A_{nil \rightarrow A}(\ ) \wedge$
$va = fa(a) \wedge$
$va1 = plus(va, 1) \wedge$
$va1 < 60 \wedge$
$r = res(a)\}$

These two function definitions are as the query fragments would be executed with all their parameters unbound. Such binding patterns are used because the real binding patterns are at this time still unknown. They are determined later in the scheduling process and the functions are recompiled to generate the plans for their execution with the correct binding. During the scheduling, a query fragment can be recompiled more than once using different binding patterns, depending on the placement in the currently generated schedule. Since a derived function recompilation does not perform the calculus optimization, it is beneficial to define these functions once and recompile them possibly multiple times.

The execution of the example query begins by evaluating one of the functions at one of the sites. Next, the other fragment function is evaluated and the results shipped to a join and materialization capable site, where an equi-join over the variable *va* is performed. Although, in general, any join capable site could join the intermediate results, in this work we consider only the sites where a query fragment is evaluated. Under this assumption, only one of the intermediate results is to be shipped, to the site where the other is produced. Therefore, at the site of the second query fragment, we can either first evaluate the second query fragment and join the results, or use the join attributes in the tuples of the materialized intermediate result.[3] For example, if *QFdb*1 is executed first and the resulting *va* values are shipped to *DB*2, we could either first execute the function *QFdb*2, and match the resulting tuples with the materialized values of *va*, or invoke *QFdb*2 with the values of the parameter *va* bound to the values in the shipped set. In order to determine the optimal schedule, the query processor must calculate and compare the costs of the different strategies. The cost calculation depends on the execution cost and the selectivity of each of the query fragments, and the cost of shipping data among the systems.

This analysis illustrates that the number of alternatives is large even in a simple example where the query is decomposed in only two query fragments as above. Hence, the strategy

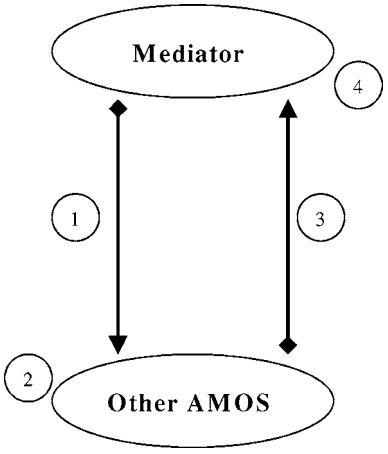

Figure 8. A query processing cycle described by a decomposition tree node.

described in this section searches only a portion of the search space of possible execution plans. The plan chosen by this search is then improved using additional heuristic described in the next section.

The generated execution schedules are described by *decomposition trees* (DcTs). Each DcT node describes one data cycle through the mediator. Figure 8 illustrates one such cycle. In a cycle, the following steps are performed:

1. Materialize successively portions of the intermediate results in an AMOS*II* server where they are to be processed.
2. Execute a remote query fragment and join the result with the intermediate result produced in step 1.
3. Ship the results back to the mediator.
4. Execute one or more local query fragments in the mediator.

The result of a cycle is always materialized in the mediator. A sequence of cycles can represent an arbitrary execution plan. Not all cycle steps are required in every DcT node.

The intermediate results used as an input in the cycle are represented recursively by a list of child DcT nodes, the *materialization list*. In order to simplify the query processing, currently the tree building algorithm considers at this stage only materialization lists with one element (left-deep trees), and therefore the intermediate result always has the form of a single flattened table.

Steps 1 through 3, which involve communication with an another AMOS*II* server, are performed by the an distributed equi-join operator that performs an equi-join between the intermediate result generated from the materialization list and the result of the execution of an remote query fragment, represented by a remote derived function [18]. Each DcT node contains the necessary information generated during the query compilation that describes the remote query fragment and the way it is invoked. More specifically this description

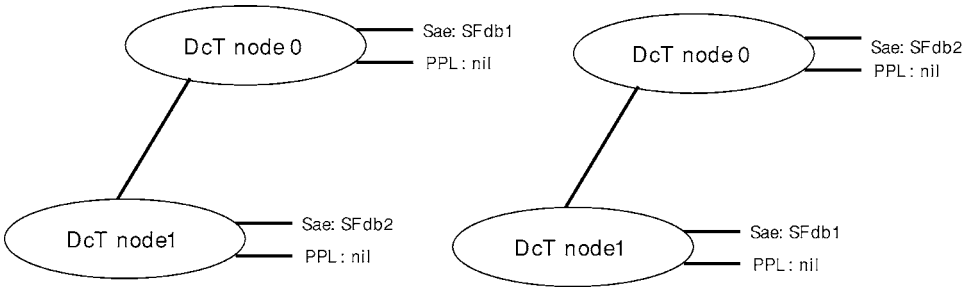

Figure 9. Two decomposition trees for the example query.

consists of the following items:

– proxy OID for the remote derived function representing the query fragment
– parameter bindings and typing information
– execution cost and selectivity estimates of the query fragment for a given binding

Step 4 is specified by a list of locally executed query fragments, *post-processing list*. These query fragments are represented by locally defined derived functions and are executed over the result of the equi-join operation specified by the previous steps. The order of the post-processing list determines the sequence of applications of the fragments, determined by the optimizer.

Figure 9 shows two trees generated for the example query. These trees illustrate the scheduling alternatives where the equi-join of the results of the execution of the two query fragments is performed at *DB*1 and *DB*2, respectively. Because we consider only left-deep trees, joins in the mediator are not considered at this stage. The trees also determine the relative order of the execution of the query fragments. The order of the operations given above implies that the trees are executed bottom-up. This in turn determines the execution binding pattern for each query fragment. The same query fragment in different trees can have different binding patterns and thus different execution costs. In the left DcT in figure 9 *QFdb*2 is executed with the variable *va* unbound, while in the tree on the right this variable is bound. If the function $fa(a)$ is expensive, or has high selectivity, then the execution of *QFdb*2 with *va* unbound can have a much higher cost than when *va* is bound. This cost variation combined with the cost variation of *QFdb*1 influences the cost of the whole tree.

The cost of an execution schedule represented by a DcT node is calculated recursively by adding the costs of the steps in figure 8 to the costs of the subtrees in the materialization list. The cost calculation depends on the algorithms used to implement the query processing cycle steps described in [18].

The left-deep DcTs are generated using a variation of the dynamic programming approach. The algorithm attempts to avoid generation of all the possible plans by keeping a sorted list of partial plans and adding to the list all the possible extensions of the cheapest one. When the cheapest plan is also a complete plan, then it is one of the plans with the lowest cost. This algorithm, used also for the singe-site is described in detail in [16].

We conclude the subsection with an observation that the described strategy is more general given OO data sources than the strategies used in some other multi-database systems, as for example, [8, 20, 24] where the joins are performed in the mediator system. Such strategies do not allow for mediation of OO sources that provide functions which are not stored, but rather performed by programs executed in the data source (e.g. image analysis, matrix operations). In this case, it is necessary to ship intermediate results to the source in order to execute the programs using the result tuples as an input. From this aspect, the strategy presented above generalizes and improves the *bind-join* strategy in [13].

4.6. Decomposition tree distribution

The scheduling phase described in the previous subsection produces a left-deep DcT representing a query execution schedule for the input query. Each DcT node describes a query processing step that involves passing data through the mediator. Some of the steps pass data from one data source to another, copying it through the mediator. In an environment consisting of a several AMOS*II* servers, it is desirable to design schedules where the superfluous data transfers and the involvement of the coordinating mediator are eliminated. In such a schedule, the participating mediators communicate directly during the execution of the query fragments. The result of the query fragment computation is then shipped to the coordinating mediator. For example, the trees in figure 9 describe plans in which the values of *va* are shipped from *DB*2 to the mediator and then to *DB*1, in the tree on the left, and vice versa in the tree on the right. It would be less costly if the mediator instructs *DB*2 to ship the values directly to *DB*1, or vice-versa.

In order to construct schedules that perform "sidewise" transfer of data, the DcT generated by the previous phase is restructured using a series of *node merge* operations, performed over two consecutive nodes, *lower* and *upper* respectively. The merge operation is applicable over two DcT nodes if both specify an equi-join with a remote fragment result and the lower node does not specify post-processing involving locally executed query fragments. In such a case, during the bottom up evaluation of the plan represented by the DcT, after the equi-join specified by the lower node the result is in the mediator. In the next query evaluation step specified by the upper DcT node, this intermediate result is shipped to the site where the remote query fragment of this node is executed, in order to perform the equi-join. If the sites are wrapped by different AMOS*II* servers, instead of shipping the intermediate results between the sites through the mediator, the plans generated by the merge operation will perform a direct shipping of the intermediate results between the mediators wrapping these sites. For example, the left tree in figure 9 describes a plan where *QFdb*2 is executed at *DB*2 and the result is shipped to the mediator. The upper node then ships the same result from the mediator further to *DB*1, where it is used in an equi-join.

In the case when the lower DcT node specifies locally executed query fragments are to be applied to the intermediate result before the next remote equi-join, the intermediate result needs to be shipped to the mediator and the merge operation is not applicable.

The node merge operation is shown in figure 10. Two consecutive nodes with the required properties are identified (figure 10(a)) and substituted with a single node. The new node has the post-processing list from the upper node and an remote fragment descriptor assembled

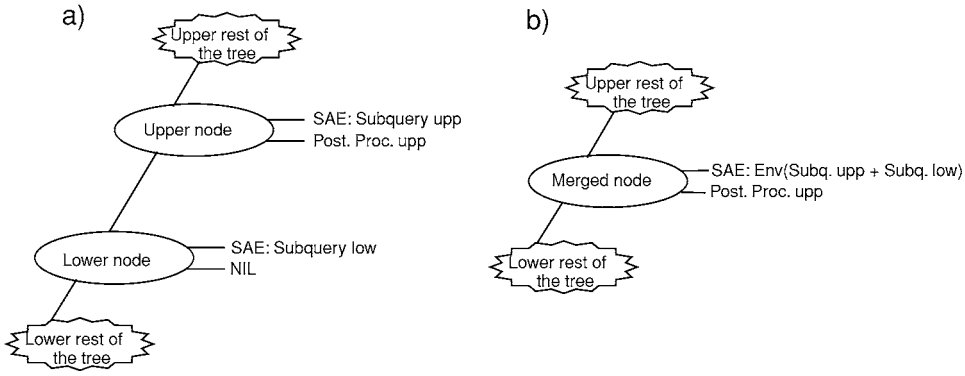

Figure 10. Node merge: a) the original tree b) the result of the merge operation.

from the remote fragment descriptors of the merged nodes. In order for the new tree to represent a correct query schedule, the derived function representing the two remote fragment should perform the same operations as the derived functions in the original nodes. Therefore, the function in the new node is a combination of the functions of the original nodes. Since these functions are remote, to avoid the unnecessary bypass of the data throughout the mediator, the new function is compiled and executed at the site where the original functions were to be executed. This is done by defining an *envelope derived function* that calls the two original functions representing the query fragments in the merged DcT nodes. The envelope function is compiled at both of the participating sites (or at the one site if both of the remote query fragments to be merged are executed at the same site) and the cheaper alternative is chosen. This, in turn, is compared with the cost of the original tree and if it has a lower cost, the modified tree is accepted instead of the original.

Figure 11 illustrates the possible data flows between the three AMOS*II* servers in the example from figure 9(a). In figure 11(a) the data flow of the execution of the original schedule is presented. The query execution starts by the mediator contacting *DB*2 to execute *QFdb*2 in step 1, and shipping across the results in step 2. Next, from the result of the previous step, the mediator sends the values of the parameter *va* to *DB*1 where *QFdb*1 is executed and the result is joined with the incoming set of *va* values. For each joined value of *va* a temporary boolean value is returned indicating which of the incoming *va* values joined with the result of the execution of *QFdb*2. Finally, after joining the result shipped in step 4 with the result of step 2, the mediator emits the values of *r* for which the temporary iteration variable *_tmp_* is *TRUE*.

This strategy would be very inefficient in cases when the set of *va* values is very large and the net links connecting the mediator with *DB*1 and *DB*2 are very slow (e.g. due to geographical dislocation). Also, note that with this strategy the *va* values are shipped twice.

The strategy illustrated in figure 11(b) is obtained by merging the nodes of the DcT in figure 9(a) and placing the envelope function at *DB*2. Here, the values of *va* are sent directly from *DB*2 to *DB*1, shipping them therefore only once. Figure 11(c) represents the execution

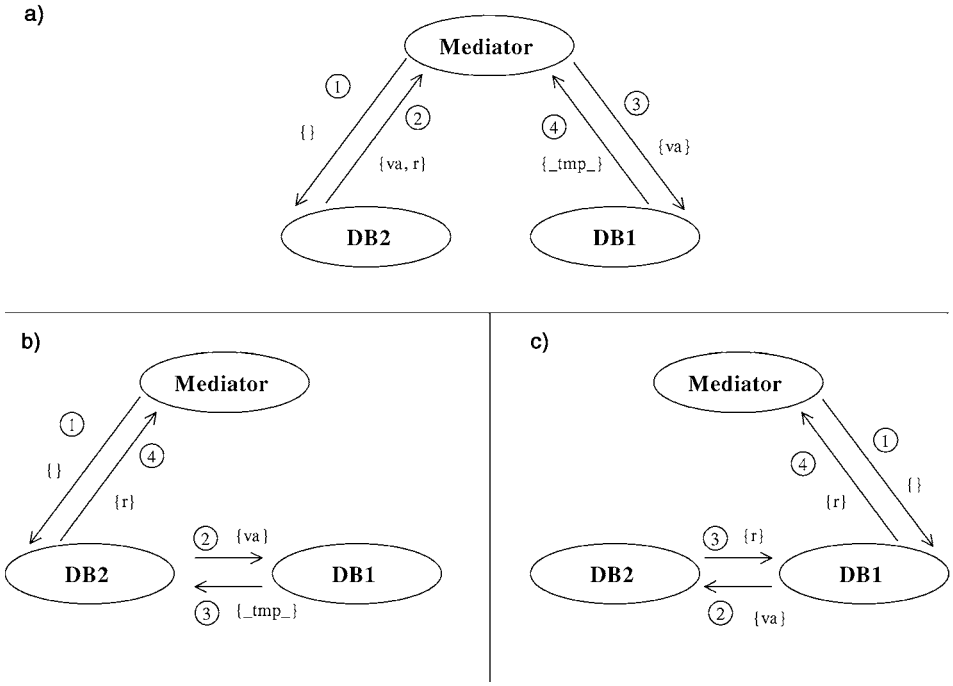

Figure 11. Execution diagrams of the decomposition tree of the example query before node merge and after.

strategy of the transformed DcT in figure 9(a) where the envelope function is placed at *DB*1. This strategy performs the best when *QFdb*1 has large selectivity and/or the network link between the mediator and *DB*2 is slow.

A series of node merge operations can produce longer data flow patterns that do not necessarily pass through the query issuing mediator. One feature of the trees produced by node mergers is that the envelope functions are themselves multi-database functions over data in multiple data sources. The compilation process creates a DcT for each envelope function at the remote site where they are defined. A repeated application of DcT node merging can break a DcT into a set of DcT located at multiple AMOS*II* servers. Hence, the process can be viewed as *DcT distribution*. Compared with the traditional query tree balancing [6] the node merge exhibits the following differences:

– Distributed compilation: node merging is a distributed process where envelope functions are compiled at nodes other than the mediator. This distributed compilation process is decentralized and does not need a centralized catalogue of optimization information that is a potential bottleneck when the number of mediators increases.
– Distributed tree: The resulting tree is not stored in one AMOS*II* server, but rather is spread over the participating servers that expose only an already compiled function for the query fragment sent by the coordinating mediator.

In a tree produced by the cost-based scheduling there might be more than one spot that qualifies for a merge operation. An important issue in applying node merging is where in the DcT to apply the the operation. Different sequences of merge operations can produce different results. The simplest solution to this problem is to perform an exhaustive application of all possible sequences of merge operations by backtracking. However, it is clear that this will require a large number of query fragment compilations and is therefore not suitable. An alternative is to use hill-climbing from a few randomly chosen positions and perform the process until no transformation can be made such that a cheaper tree is produced. The process can be guided by heuristics that prioritize DcT nodes where the transformation can be especially useful, and avoid merging nodes that are unlikely to produce a merged node with lower cost. An interested reader is referred to [15, 16, 19] for a more detailed description and an experimental evaluation of the DcT distribution.

5. Summary and conclusion

The distributed mediator architecture has been proposed to alleviate the administrative and performance bottleneck in the centralized wrapper-mediator architecture for data integration. Query decomposition is one of the central query processing phases in every mediator system. This paper presented a query decomposition algorithm used in the distributed mediator system AMOS*II* . The algorithm distinguishes between different kinds of data sources and the mediators that interact to provide an answer to a user query. The interaction between the mediators is on a higher level than the interaction between a mediator and a data source. More specifically, a mediator can accept compilation of a sub-query over more than one source and allow for shipping of intermediate results to be used as parameters to a sub-query. The presented algorithm first uses heuristics to determine the sub-queries for the individual data sources and mediators, then builds a left-deep schedule for execution of the sub-queries using a dynamic programming approach, and finally re-balances the schedule using a distributed query compilation.

Another feature of the approach presented in this paper is that it is object-oriented (OO) and allows integration of OO sources. When integrating OO sources that might encapsulate programs rather than data, it is necessary to consider execution schedules that ship data to the sources where it is used as input to the programs. This type of schedules are opposed to the schedules build by the traditional query decomposition approaches where the data is retrieved from the sources and then the query result is composed by post-processing operations in the mediator. The algorithm also considers the situation when several sources implement the same operation. In such cases, the execution site for the operation is determined using a partial cost calculation.

One important issue addressed in this paper is that of the division of the query processing facilities between the query decomposer and the wrappers. A simple query decomposer requires more complex wrapper implementations. A wrapper in such a case must be able to perform more sophisticated transformations in order to produce representation of the query fragment that is executable by the data sources. Furthermore, the same features might be needed and re-implemented in several wrappers. A more elaborate query decomposer, on the other hand, leads to a slower query decomposition and less maintainable code.

The design of the query decomposer described in this work aims to provide a functionality sufficient for easy integration of the majority of the data sources we have accounted for, while keeping the design as simple as possible. Compared to other approaches to the integration of heterogeneous data sources based on grammars and rules it allows for partitioning of the query into fragments without repeated probing for fragments that are executable in the data sources. Data sources that cannot be described by MIFs and join capability might require wrappers capable of restructuring the query fragments sent by the decomposer so it can be successfully translated into code executable in the data sources. In this process, the wrapper writer can use some externalized local query optimizer facilities to perform some commonly required tasks (e.g. enumeration, costing, sorting, compilation etc.), needed in wrappers for several data source types. Nevertheless, we believe that such cases are rare.

With an appropriate change of the heuristics used in the early phases, the algorithm can be adapted for use in environments where the services (programs) can move from one source to another. Another challenge is to extend the query decomposition to produce plans suitable for parallel execution. Finally, in order to extend the use of the AMOS*II* system to a dynamic environment, we are currently developing query processing and optimization techniques that deal with unavailability of the sources, and dynamically explore alternative paths to the required data and services.

A simplified version of the presented algorithm is implemented in the AMOS*II* system on a Windows NT platform.

Acknowledgments

The authors would like to thank Timour Katchaounov for his participation in the AMOS*II* project and the implementation of the rebalancing algorithm.

Notes

1. The terms “source” and “data source” are used interchangeably, although the sources might contain programs rather than data.
2. The term *site* is used to refer to both AMOS*II* servers and data sources of all other kinds. The terms *site assignment* and *query fragment placement* are used interchangeably.
3. This is a conceptual view of the execution; AMOS*II* has streamed execution model where the intermediate results are shipped in bulks.

References

1. S. Brandani, “Multi-database Access from Amos II using ODBC,” Linköping Electronic Press, vol. 3, no. 19, 1998. http://www.ep.liu.se/ea/cis/1998/019/.
2. M. Carey, L. Haas, J. Kleewein, and B. Reinwald, “Data access interoperability in the IBM database family,” IEEE Data Engineering, vol. 21, no. 3, pp. 4–11, 1998.
3. D. Daniels, P. Sellnger, L. Haas, B. Lindsay, C. Mohan, A. Walker, and P. Wilms, “An introduction to distributed query compilation in R*,” Distribute Data Bases, in H. Schneider (Ed.), North-Holland: Amsterdam, 1982.
4. U. Dayal and H. Hwang, “View definition and generalization for database integration in a multidatabase system,” IEEE Trans. on Software Eng., vol. 10, no. 6, 1984.

5. W. Du, R. Krishnamurthy, and M.-C. Shan, "Query optimization in heterogeneous DBMS," 18th Conf. on Very Large Databases (VLDB'92), Vancouver, Canada, 1992.
6. W. Du and M. Shan, "Query processing in pegasus," in Object-Oriented Multidatabase Systems, O. Bukhres and A. Elmagarmid (Eds.), Prentice Hall: Englewood Cliffs, NJ, 1996.
7. G. Fahl and T. Risch, "Query processing over object views of relational data," The VLDB Journal, vol. 6, no. 4, pp. 261–281, 1997.
8. B. Finance, V. Smahi, and J. Fessy, "Query processing in IRO-DB," Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95), 1995, pp. 299–319.
9. S. Flodin, V. Josifovski, T. Risch, M. Sköld, and M. Werner, AMOS*II* User's Guide, available at http://www.ida.liu.se/∼edslab.
10. S. Flodin and T. Risch, "Processing object-oriented queries with invertible late bound functions," 21st Conf. on Very Large Databases (VLDB'95), Zurich, Switzerland, 1995.
11. J. Fowler, B. Perry, M. Nodine, and B. Bargmeyer, "Agent-based semantic interoperability in infoSleuth," SIGMOD Record, vol. 28, no. 1, pp. 60–67, 1999.
12. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom, "The TSIMMIS approach to mediation: Data models and languages," Journal of Intelligent Information Systems (JIIS), vol. 8, no. 2, pp. 117–132, 1997.
13. L. Haas, D. Kossmann, E. Wimmers, and J. Yang, "Optimizing queries across diverse data sources," in 23th Int. Conf. on Very Large Databases (VLDB97), Athens Greece, 1997, pp. 276–285.
14. J. Hellerstein, M. Stonebraker, and R. Caccia, "Independent, open enterprise data integration," IEEE Data Engineering, vol. 22, no. 1, 1999.
15. V. Josifovski, "Design, implementation and evaluation of a distributed mediator system for data integration," Ph.D. Dissertation, Linköpings Universitet, Linköping, Sweden, 1999.
16. V. Josifovski, T. Katchaounov, and T. Risch, "Optimizing queries in distributed and composable mediators," in Proc. of 3rd Intl. Conf. on Cooperative Informational Systems (CoopIS99), Edinburgh, Scotland, Sept. 1999.
17. V. Josifovski and T. Risch, "Functional query optimization over object-oriented views for data integration," Journal of Intelligent Information Systems (JIIS), vol. 12, no. 2/3, 1999.
18. V. Josifovski, T. Risch, and T. Katchaounov, "Evaluation of join strategies for distributed mediation," in Proc. of Conf. on Advances in Database and Information Systems (ADBIS), Vilnius, Lithuania, 2001.
19. T. Katchaounov, V. Josifovski, and T. Risch, "Distributed view expansion in composable mediators," in Proc. of 4th Intl. Conf. on Cooperative Informational Systems (CoopIS2000), Haifa, Israel, 2000.
20. E.-P. Lim, S.-Y. Hwang, J. Srivastava, D. Clements, and M. Ganesh, "Myriad: Design and implementation of a federated database system," Software—Practice and Experience, vol. 25, no. 5, pp. 553–562, 1995.
21. W. Litwin and T. Risch, "Main memory oriented optimization of OO queries using typed datalog with foreign predicates," IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 6, pp. 517–528, 1992.
22. L. Liu and Calton Pu, "An adaptive object-oriented approach to integration and access of heterogeneous information sources," Journal of Distributed and Parallel Databases, vol. 5, no. 2, pp. 167–205, 1997.
23. P. Lyngbaek, "OSQL: A language for object databases," Technical Report, HP Labs, HPL-DTD-91-4, 1991.
24. S. Nural, P. Koksal, F. Ozcan, and A. Dogac, "Query decomposition and processing in multidatabase systems," OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis, Montpellier, July 1996.
25. K.Orsborn and T. Risch, "Next generation of O-O database techniques in finite element analysis," Intl. Conf. on Computational Structures Technology, Budapest, Hungary, Aug. 1996.
26. K. Richine, "Distributed query scheduling in DIOM," Technical Report TR97-03, Computer Science Department, University of Alberta, 1997.
27. D. Shipman, "The functional data model and the data language DAPLEX," ACM Transactions on Database Systems, vol. 6, no. 1, 1981.
28. A. Tomasic, L. Raschid, and P. Valduriez, "Scaling access to heterogeneous data sources with DISCO," Transactions on Knowledge and Data Engineering (TKDE), vol. 10, no. 5, pp. 808–823, 1998.
29. G. Wiederhold, "Mediators in the architecture of future information systems," IEEE Computer, vol. 25, no. 3, 1992.