Optimizing Queries in Distributed and Composable Mediators

Vanja Josifovski, Timour Katchaounov, Tore Risch
Laboratory for Engineering Databases
Linköping University, Sweden

{vanja, timka, torri}@ida.liu.se

4th Conf. on Cooperative Information Systems, CoopIS'99, Edinburgh, Scotland, 1999

Abstract

The mediator-wrapper approach to integrate data from heterogeneous data sources has usually been centralized in the sense that a single mediator system is placed between a number of data sources and the applications. As the number of data sources increases, the centralized mediator architecture becomes a bottleneck. This paper presents an architecture for composable and distributed mediator servers, defined in terms of other mediator servers. The modularity of composable mediators allows to build larger systems of distributed mediators integrating many data sources, without the need to maintain a global schema. Composable mediators furthermore provide data independence by allowing locality of changes in both submediators and data sources. However, a problem with a distributed and composable mediator architecture is that the query performance may degrade as the number of mediators increases. We describe some challenges for processing queries in this type of environment, and propose a distributed query decomposition algorithm that eliminates some of the overhead of logical mediator composition. For certain mediator compositions it produces distributed query plans whose inter-mediator data flow is optimal with respect to the query, but is different from the logical interdependencies between the involved mediators. Experimental results show that this strategy improves the query performance and allows increase of the number of mediators without query performance degradation.

1 Introduction

The wrapper-mediator approach for integration of data from heterogeneous data sources has been used in several projects [13, 25, 11]. This approach divides a data integration system into two functional units. The wrapper provides access to the data in the *data sources* using a *common data model* (CDM), and a common query language. The mediator provides a semantically coherent CDM representation of the combined data from the wrapped data sources, built using reconciliation primitives. Usually the data sources are distributed to several sites and accessed over some computer network. The mediator provides transparent access to the combined data from the data sources through queries to the mediating view The user/programmer does not need to make individual interfaces to each data source.

Current mediator systems and prototypes [13, 25, 11, 21, 5] are centralized systems where a single mediator server integrates data through a number of wrappers. Although indicated in some system architecture overviews, to the best of our knowledge no system allows many distributed mediator servers to interoperate. An original goal for mediator architectures [26] was that mediators should be relatively simple abstractions of modules of data and that larger systems of mediators should be composed through these primitive mediators. By making mediators servers composable and modular by allowing some mediators to act as wrappers for other mediators, it would be possible to scale the data integration process in the sense that more complex systems of data sources can be integrated than through a central integration. Composable mediators would allow for conceptual modeling of mediators without detailed knowledge of the definitions of other mediators and data sources, through modular design. As for other complex systems, modularity is essential for building large systems of mediators. Furthermore, modularity also increases the data independence between applications, mediators, and data sources by allowing for changes in lower level mediators and data sources without changes in higher level systems. When many mediator servers become available on the computer networks composability will be required for designing new distributed mediator servers in terms of the existing ones.

The design of composable and distributed mediator servers introduces, however, some new challenges to be addressed in this paper. For example, a naive implementation of several levels of mediators as black boxes, as with CORBA technology [24], would often cause significant performance overhead. While on a conceptual level it can make the modeling task easier, such black box treatment of mediators would prohibit extensive query processing over submediators. There is a need to minimize the overhead of the mediator composition hierarchy. CORBA-like technologies furthermore provides only object-instance oriented communication primitives while efficient query execution requires bulk-oriented inter-mediator communication.

We have developed a distributed mediator system, AMOS*II* , in which federations of mediator servers, acting as virtual object-oriented (OO) databases, can intercommunicate. Each mediator server in the federation has full OO query processing and cost-based optimization capabilities. It exports to other systems interfaces having capabilities for i) exchanging meta-data, ii) processing OO queries, iii) estimating query costs, and iv) bulk-oriented exchange of data. The user can post OO queries to any mediator server and the involved mediators in the federation will interoperate to produce the result as quick as possible.

In such a distributed mediator hierarchy the logical composition of mediators needs not necessarily be the same as the optimal data flow through the network of mediators for answering a query. Often it is favorable to do as much data selection as possible in the data sources and the mediators close to them. If a query needs data from only a single data source it is better to bypass all intermediate mediators which would then do no further filtration. It should furthermore be noted that the performance also depends on the speed of the links between the nodes in the federation and on the computers involved. A good distributed mediator query optimizer should take into account local and shipping costs to produce an optimal query execution plan distributed over the mediators.

The query optimization task in AMOS*II* is distributed over the distributed mediator servers. For a given mediator query a distributed query processing algorithm produces a distributed execution plan with optimized data flow that eliminate much of the overhead of composed mediation. Each local AMOS*II* optimizer knows the local access costs and can ask other mediators and data sources about their access costs. No mediator has total knowledge about all costs.

The query optimizer is thus modular in the sense that it does not work in a central environment where one mediator system has all knowledge needed for query processing. This eliminates the need for a centralized directory of schema and optimization information that might become a bottleneck when the number of mediator servers increase. Instead, in the proposed framework, each local query optimizer exchanges meta-information and costs with the other query optimizers in the federation.

We have done some experiments showing promising results for our distributed mediator query optimization techniques. The experiments show that the distributed query decomposition algorithm can produce better distributed inter-mediator plans than if data is joined through a central mediator. Furthermore, the reported results show that the distributed query decomposition algorithm produces plans that allow for increasing the number of involved servers with minimal increase of the query processing time. The experiments also show that, for a given query, different optimal distributed query execution plans sometimes need to be produced depending one the communication speeds between mediator nodes. For example, a personal mediator may reside in a portable computer and different execution plans are optimal when communicating with the federation over a telephone line than when the computer is connected to the LAN.

The paper is organized as follows. Section 2 introduces the terminology and the features of the AMOS*II* system. In Section 3 the query decomposition and the distributed compilation are described. Section 4 presents experimental results showing the benefits of the proposed strategies. The conclusions are presented in Section 5.

2 Data Integration with AMOS*II*

The AMOS*II* system has its roots in the workstation version of the Iris system, WS-Iris [18]. The core of AMOS*II* is an open, light-weight, and extensible database management system (DBMS). To achieve better performance, and because most of the data reside in the data repositories, AMOS*II* is designed as a main-memory DBMS. Nevertheless, it contains all the traditional database facilities, such as a recovery manager, a transaction manager, and a OO query language named AMOSQL [9]. An AMOS*II* server provides services to applications and to other AMOS*II* servers.

AMOS*II* is a distributed mediator system [26] where a number of mediator servers communicate over the Internet. Some of these servers can be configured as *translators* [7] which wrap different kinds of data sources, e.g. ODBC compliant relational databases [2] or XML files. We use the term translator since it is a fully fledged AMOS*II* system which can wrap more than one data source, contains a complete query processor, and supports semantic abstractions and conversions of the data in the data sources through OO views. A translator is thus also a mediator which provide a virtual OO database server layer that transparently translates data from some data sources.

Users and applications can pose OO queries to any AMOS*II* server. We call the server(s) to which some queries are posed *client mediator(s)* for those queries. The other AMOS*II* servers involved in answering a query are called *server mediators*. For example, in a mobile environment a portable computer could have a client mediator that integrates data from several server mediators on a company LAN. Such a scenario is assumed in our experiments below.

A client mediator can have various types of data sources and access a number of autonomous server mediators. As opposed to a distributed database environment where the data, meta-data and optimization information are available in a centralized repository, in an autonomous environment each server contains only portions of this data.

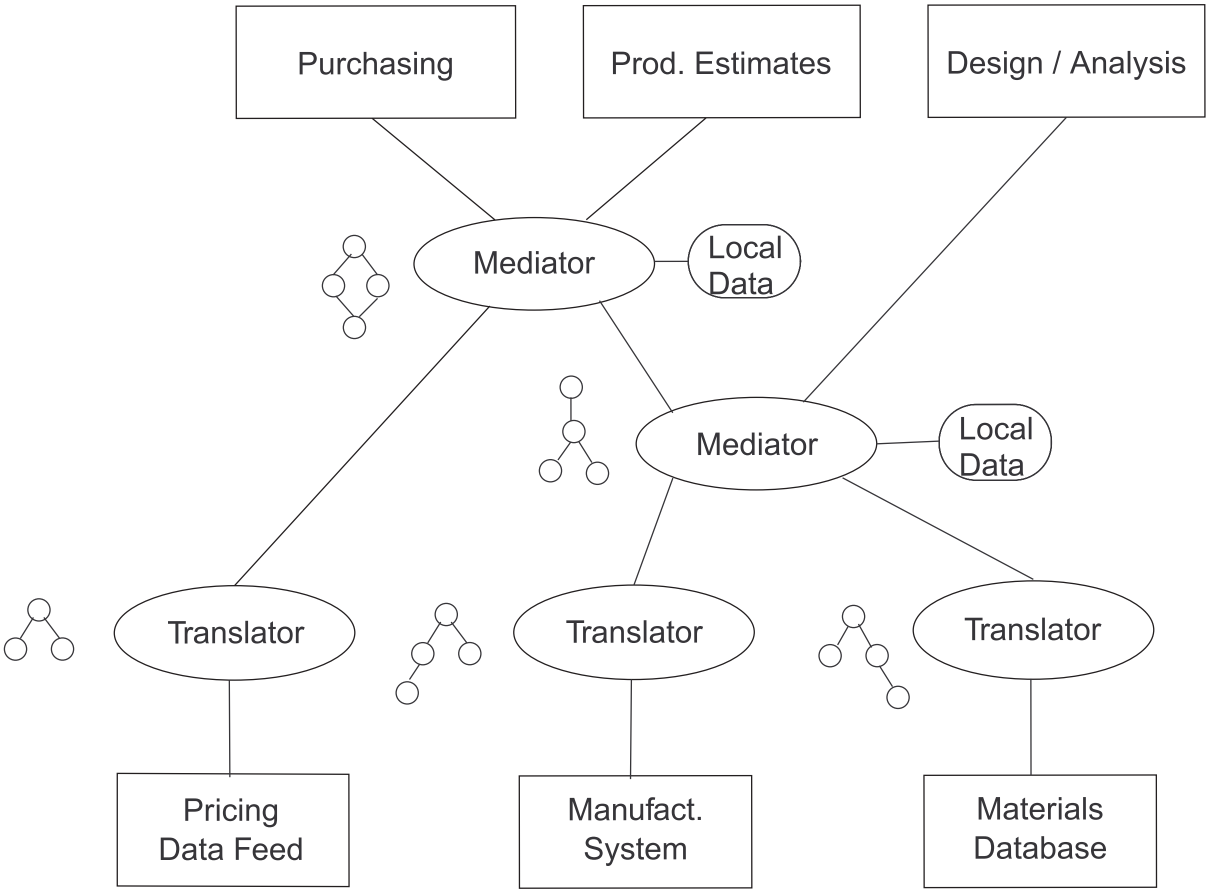


Figure 1: Interconnected AMOS*II* systems

Figure 1 illustrates the different roles that an AMOS*II* server can assume. In this example, applications access data stored in data sources through a collection of composed mediator servers. The servers may run on separate workstations and provide data integration, translation, and abstraction services through which different object view hierarchies are presented in the different mediators, as indicated in Figure 1. The mediator servers appear as virtual database servers having data abstractions, query interface, and other database functionality. AMOS*II* mediators are composable since a mediator server can regard other mediator servers as data sources. A single AMOS*II* server can also assume more than one role described in Figure 1 and serve more than one application simultaneously. Different interconnecting topologies can be used to connect mediator servers depending on the integration requirements of the environment. Here, a naive implementation where messages are passed between the several layers of composed mediator servers may have unacceptable performance. However, we have developed distributed mediator query optimization techniques that minimize the overhead of composing mediator servers, to be further elaborated in this paper.

The data model in AMOS*II* is an OO extension of the DAPLEX [22] functional data model. It has three basic constructs: *objects*, *types* (i.e. classes), and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types which makes the object *instances* of those types. The set of all instances of a type is called the *extent* of the type. Object properties and their relationships are modeled by functions.

The types are divided into *stored*, *derived*, *proxy*, and *integration union* types, where the instances of *stored* types are explicitly stored locally in AMOS*II* and created by the user, the instances of *derived* types [14] are derived through a declarative query from the instances of one or more *constituent* supertypes, the instances of *proxy* types represent objects stored in other AMOS*II* servers or in some of the supported types of data sources, and the instances of *integration union types* (IUTs) [14] are defined as unions of instances representing the same real-world entity in different data sources. Even though the IUTs are outside the scope of this paper, the features presented in the experiments reported in this work are directly connected with the processing of queries over the IUTs, which require outer-join based operations transformed into a set of select-project-join queries [15].

The proxy, derived and integrated union types are the core of the integration framework in AMOS*II* . Composition of such types provide means for resolving a wide spectrum of semantic heterogeneities between the data and meta-data in the sources. Queries over the OO views are transformed into queries over data in multiple data sources. The OO view mediation framework is described in [14, 15].

The AMOS*II* functions are divided by their implementations into three groups. The extent of a *stored* function is physically stored in the database. *Derived* functions are implemented in the query language AMOSQL. *Foreign* functions are implemented in some other programming language, e.g. Java, Lisp or C++. Each foreign function can have several associated access paths and, to help the query processor, each access path has an associated cost and selectivity function [18].

The AMOSQL query language is based on the OSQL [19] language with extensions of mediation primitives, multi-directional foreign functions [18], overloading, late binding [8], active rules [23], etc. It contains data modeling as well as querying constructs. The general syntax for AMOSQL queries is:

```
select <result>
```

```
   from <type declarations for local variables>
   where <condition>
```

For example, the following query retrieves the names of the parents of all persons having 'sailing' as hobby:

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

Figure 2, presents an overview of the query processing in AMOS*II* . The first five steps, also called *query compilation* steps, translate the body of a query expressed in AMOSQL to a query execution plan which is stored with the query.

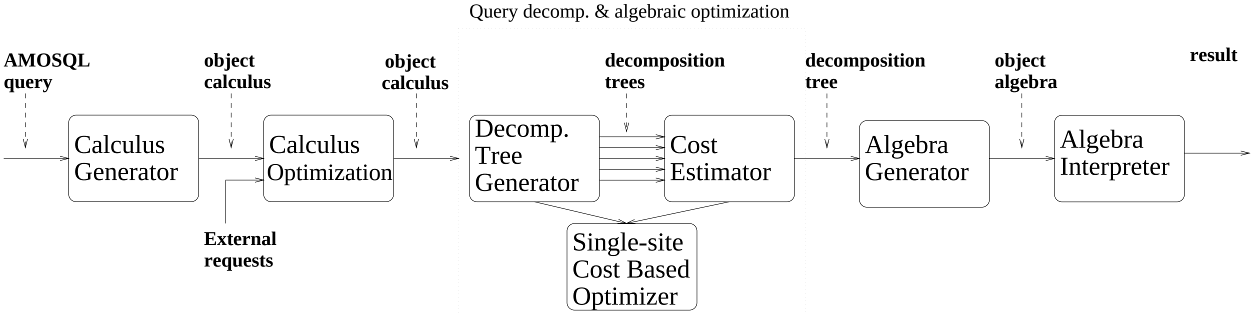


Figure 2: Query processing in AMOS*II*

From the parsed query tree, AMOS*II* first translates the AMOSQL queries into a type annotated *object calculus* representation [15].

Next, the calculus optimizer applies rewrite rules to reduce the size of the query [15].

After the rewrites, queries operating over data outside the mediator are decomposed into distributed subqueries expressed in an *object algebra*, to be executed in different AMOS*II* servers and data sources. The decomposition uses a combination of heuristic and dynamic programming strategies. At each site, a single-site *cost-based optimizer* generates optimized execution plans for the subqueries.

An interested reader is referred to [9] for a more detailed description of the AMOS and AMOS*II* system and to [18, 7, 8, 14, 15] for more on its query processing.

3 Query Plan Distribution

The distributed mediator framework of AMOS*II* allows cooperation of a number of distinct mediators on a query processor level. While distribution is present in any mediation framework due to the distribution of the data sources, the distributed mediator server framework introduces a higher level of interaction among the mediator systems. In other words, a client mediator does not treat another AMOS*II* server as just another data source. More specifically, if we compare the interaction between a centralized mediator system and a wrapped data source and the interaction between two AMOS*II* servers, there are two major differences:

- An AMOS*II* server can accept compilation and execution requests for general queries accessing data in more than one source. The wrapper interfaces accept subqueries that are always over data in a single data source.
- AMOS*II* supports materialization of intermediate results to be used as input to locally executed subqueries, generated by query decomposition in another AMOS*II* server. A wrapper provides *execute* functionality for queries to the data source. The query execution interface of AMOS*II* , on the other hand, provides *ship-and-execute* (SAE) functionality, that can first accept and store locally an intermediate result, and then execute a subquery using it as an input.

These two features influence the design of both the query decomposer and the run-time support for query execution. Techniques based on these features to achieve improved query performance are presented in this section. In the remaining of the section, first we overview the basic decomposition algorithm, and then present a method to improve the resulting query execution schedules by taking advantage of the features described above.

3.1 Query Decomposition

The query decomposition phase [14] of the query processing in AMOS*II* is invoked whenever a query is posed over data from more than one data source. The input of the query decomposition is a query calculus expression operating over imported (proxy) and local stored types. The output is an executable algebra plan. The query decomposition process follows in 5 phases:

1. Predicate grouping
2. Execution site assignment
3. Execution schedule generation
4. Object algebra generation

The rest of this subsection gives an overview of each of these phases. A more thorough description can be found in [14].

- *Predicate grouping.*

  This phase attempts to reduce the problem of finding a suboptimal execution plan by reducing the number of predicates. Predicates executed at the same data source are grouped into one or more composite predicates that are treated afterwards as single predicates. For each composite predicate, a temporary derived function is defined locally or at another AMOS*II* server. The following grouping heuristics is used:

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

  Within a composite predicate, the optimization is performed in the AMOS*II* server where this predicate is forwarded for execution.

- *Site assignment (group placement).*

  This phase uses cost-based heuristics to make the final decision which composite predicate is executed where, eventually replicates some of the predicates, and assigns execution sites to those predicates that can be executed at more than one site (e.g. θ-joins specified by comparison operators). The output of this phase is a query graph where all the nodes are assigned to some site.

- *Cost-based execution scheduling.*

  In order to translate the query graph from the previous phase into an executable query plan, the query processor must decide on the order of execution of the predicates in the graph nodes, and on the direction of data shipping between the nodes.

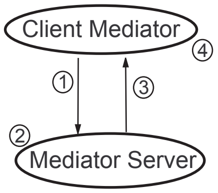


Figure 3: Query processing cycle, described by a decomposition tree node

Execution schedules for distributed queries in AMOS*II* are represented by *decomposition trees* (DcTs). Each DcT node describes one data cycle through a client mediator. Fig. 3 illustrates one such cycle. In a cycle, the following steps are performed:

1. Materialize intermediate results in a source where they are to be processed.
2. Execute a subquery function with the materialized data as input
3. Ship the results back to the client mediator
4. Execute one or more subquery functions defined in the client mediator (post-processing).

Each DcT node stores information about the first three steps in a structure called *ship-and-execute* (SAE) structure. The last step is described by a post-processing structure. The result of a cycle is always materialized in the client mediator. A sequence of cycles can represent an arbitrary execution plan.

As the space of all execution plans is exponential to the number of participating databases, we examine only a subset of the family of left-deep decomposition trees by using dynamic programming and heuristics to prune the search space. The outcome of this phase is an executable left-deep decomposition tree. Being central to our discussion, we elaborate more on decomposition tree generation in the next subsection.

- *Algebra generation.*

  The input to this phase is an executable decomposition tree, which is translated into equivalent sets of inter-calling local object algebra plans.

3.2 Tree Balancing and Distribution

The query decomposition algorithm as presented above, produces an initial *centralized* execution plan. This plan is similar to the execution schedules produced in other distributed and multidatabase systems, e. g. [3, 13, 17], where the query compilation and execution is a centralized process, managed by a coordinator for distributed databases, or by a single client mediator. All inter-site result assembling operations (equi-joins) are performed in the client mediator (coordinator).

This type of plans suffer from heavy involvement of the central client mediator and high network traffic between the client and server mediators. Furthermore these plans might contain redundant operations in which intermediate results are shipped from one server mediator to another, passing through the client mediator.

In order to eliminate these problems, an additional query decomposition phase is introduced after the cost based scheduling, that improves the centralized left-deep execution schedule produced by the cost based scheduling. It uses a distributed compilation process to translate the input schedule into a plan distributed over all the participating servers which communicate the intermediate results directly to each other.

As noted above, in the execution schedules represented by the initial left-deep decomposition trees of the centralized plans, all data shipped between any of the server mediators, always passes through the client mediator. A graphical representation of the data flow patterns generated by these plans is shown in Fig. 4. The composition of mediators is designed using semantic considerations, as opposed to trying to distribute load or other performance considerations. It may introduce considerable performance problems, mostly due

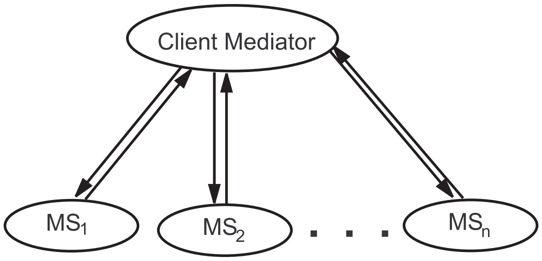


Figure 4: Class of centralized data flow patterns in AMOS*II*

to transmission costs between (possibly) many layers of mediators. In many cases, it might be cheaper if different server mediators can exchange data directly, independently from the client mediator. This is even more true in the cases of non-homogeneous execution environments, for example when the client mediator accesses the server mediators over a slow communication channel.

In he rest of this section, we present a novel distributed query decomposition technique, named *Tree Distribution*, for tree balancing in an environment of distributed, autonomous mediator systems. It extends the query decomposition process in AMOS*II* in order to explore the richer space of execution plans allowing direct communication among the server mediators involved in a query. It distributes not only the execution, but also the decomposition of the query plans among different servers. Experimental results show substantial performance improvement over the plans before the tree distributions.

The class of plans in Fig. 4 are transformed, when favorable, into plans using direct communication between different server mediators participating in the query without passing data through the client mediator. The algorithm uses random hill-climbing with a complexity that is linear to the size of the decomposition tree. Although this approach does not enumerate all the possible plans, in our experience it suffices for the typical mix of queries posed to a client mediator. The gains are especially apparent when the client mediator is hosted on a computer connected to the server mediators via a slow line.

3.3 Tree Distribution

Each node of the left-deep tree generated by the cost-based decomposition phase describes one processing cycle of Fig. 3 above. The data flow presented in Fig. 4 shows that the centralized left-deep plans generate data flow composed of several individual cycles between the client and server mediators (SM). Note that, in presence of OO server mediators, this strategy is more general than the strategy used in some other multidatabase systems (e.g. [20, 12, 17]) where the joins are performed in the client mediator system over data retrieved from the participating wrapped data sources. The latter does not allow for mediation of OO sources that access not only stored data, but also contain programs executed in the data source (e.g. image analysis, matrix operations). In such cases, it is impossible to retrieve the program logic from the source and therefore it is necessary to ship intermediate results to the source in order to execute the programs using the shipped data as input. From this aspect, the strategy is similar, but more efficient than the *bind-join* strategy in [13] since we use bulk shipping rather than instance (tuple) shipping.

The main idea of the tree distribution algorithm is to transform the centralized tree by a series of *node merge* operations. A node merge aims to eliminate the data flow through the client mediator and is applied over two consecutive decomposition tree nodes (a lower and an upper node) such that the lower node does not specify post-processing operations in the client mediator (step 4 in Fig. 3). The absence of the post-processing operations in the lower node means that the data is streamed unchanged from the server mediator participating in the lower node cycle (e.g SM_0), through the client mediator, to the server mediator participation in the upper node cycle (e.g. SM_1).

The node merge operation produces a new node that substitutes the two merged node

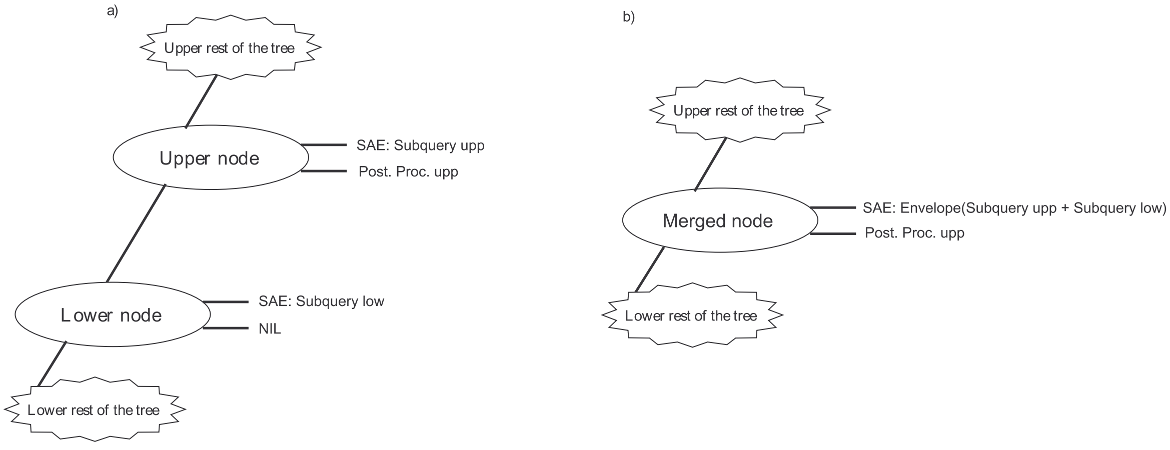


Figure 5: Decomposition tree node merger operation a) before the merge b) after the merge

in the DcT, as shown in Fig. 5. The new node has the same post-processing operations as the upper node. The ship-and-execute (SAE) operations of the new node are performed by a *envelope function* defined and compiled at one of the two mediator servers participating in the processing cycles described by the merged nodes. The body (predicate) of the envelope function is made by conjuncting the bodies of the functions specified in the SAE structures of the merged nodes. When the envelope function is compiled at one of the server mediators participating in the query, the generated execution plan contains a processing cycle that ships data *sideways* between the two server mediators, eliminating the involvement of the client mediator. The merged node describes a cycle where the envelope function is invoked in one of the server mediators, and the result of its invocation is shipped back to the client mediator. Note that each envelope function is a derived function (view) over data in more than one server mediator, and therefore the distributed query compiler generates a new decomposition tree for it at the server where it is compiled. After repeated recursive application of node merge operations the query execution plan is described by a set of decomposition trees stored in both the client mediator and the participating server mediators. Since these trees are generated by compilation at the server mediators, the client mediator does not need any of optimization information used in the compilation of the envelope functions.

The application of the merge operation is applied at a random qualifying point in the tree. If the new tree has lower execution time than the original, then it is used instead of the original. The process continues until no beneficial merge operations are performed. The maximum number of merges during this process is $2(n-1)$, where n is the number of nodes in the input decomposition tree. In it's final variant the input tree might become distributed between $n-2$ server mediators and the client mediator.

The family execution plans, generated by this algorithm have the general data flow pattern of Fig. 6, where all communicating servers can be classified into two types of groups - groups containing servers which exchange data only with the client mediator (A in Fig. 6) and groups of servers that communicate directly with each other in a sequential manner

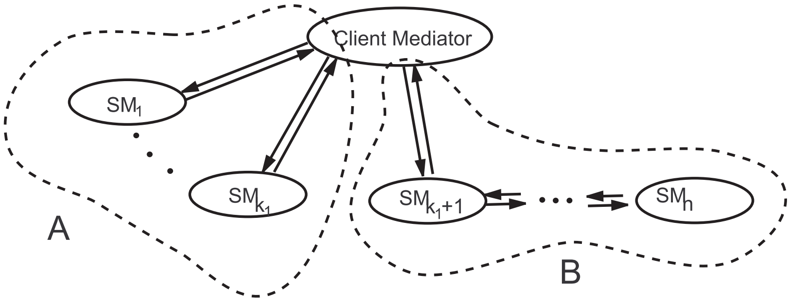


Figure 6: Class of distributed data flow patterns in AMOS*II*

(group B in Fig. 6). In the plans generated by the transformations described above, groups of the both type are interchanged. The introduction of type B groups into the plans, achieved by the proposed transformation, allows for better performance in a common case when a network of server mediators, connected by fast connections is accessed by a remote client mediator through a slow (modem or mobile) line.

4 Experimental Evaluation

This section presents the results from a comparison of the centralized plan with the distributed plan produced by the Tree Distribution algorithm for a class of queries to a client mediator.

4.1 Experimental environment

The performance experiments were made in two different environments. In both cases we had up to four server mediator, named M1, M2, M3, and M4, respectively, running on the same computer, and a client mediator, named M0, executing as a client on a remote computer, to which the queries were posed. All data was stored in an Oracle 8 relational data source, and server mediator M1 was acting as a translator through an ODBC wrapper. The Oracle server was running on the same computer as the four server mediators. The choice to run the server mediators M1-M4 on the same computer was made in order to simplify the experimental setting. We use local TCP/IP communication also between servers running on the same Windows NT workstation, and we measured that the local TCP/IP performance is actually about 10-15% slower than inter-computer TCP/IP communication over our 100 Mbit LAN. Furthermore, in our experiments we do not explore asynchronous server intercommunication which makes only one system run at the time. Thus this setting is roughly equivalent to the case when the translators are running on different computers on the LAN. It was ensured that all participating AMOS*II* systems fit into main memory, and no swapping occurred during the experiments.

The hardware used during the experiments was Compaq Professional Workstations with 200 MHz Pentium Pro CPUs, 128 MB RAM, and a 100 Mbit LAN card, running Windows NT Workstation 4.0.

In the first set of experiments the client mediator was connected to the server mediators through the LAN, while the second set of experiments were performed with the client mediator running on a remote NT workstation connected to our LAN over a 128 Kbit ISDN line. This corresponds to the situation where the client mediator resides in a portable computer which is sometimes connected over a LAN and sometimes remotely over a slower connection.

4.2 Queries and query plans

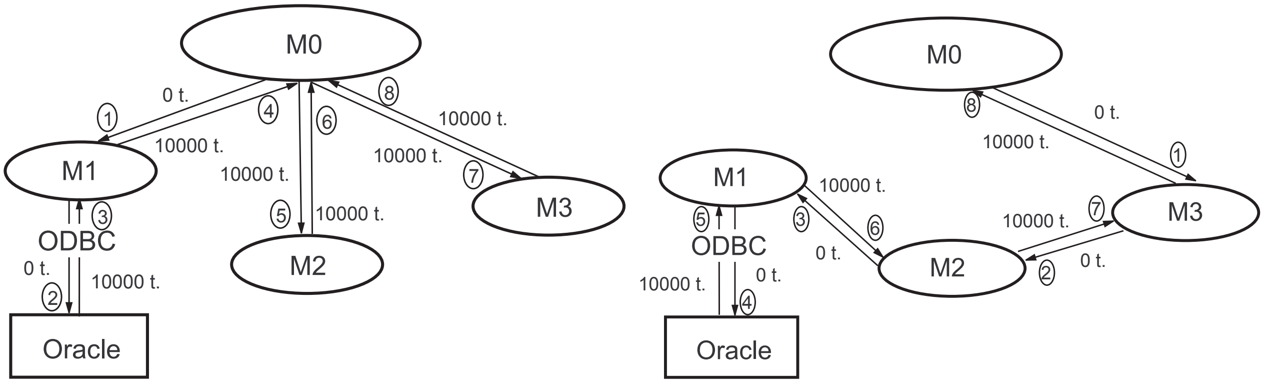
For the experiments we used a synthetic database with tables having varying number of tuples. Two types of precompiled queries were used in the experiments. In the examples we will use AMOSQL syntax. The first query was used to measure scale-up properties of the tree distribution algorithm as the mediator query spans more servers. In order to do this, we compiled three similar queries, such that the first one - *Q1'* involved two server mediators (M1,M2), the next one *Q1"* involved three server mediators (M1,M2,M3), and *Q"'* was executed over all four servers M1-M4. In each mediator *Mi* a function

$$process@Mi(charstring\ str, integer\ sel) \rightarrow charstring$$

was defined, which was simulating processing of data in *Mi* by selecting *sel*% of it's incoming data. For the discussed experiments we choose 100% selectivity ($sel = 100$) for all $process@Mi$ functions. In order to extend a query to involve server *Mi*, a call to the corresponding function $process@Mi$ was added to the query. Server *M1* wrapped the relational data source, and the *DATA* column of a relational table *EMPLOYEE* was accessed by the foreign AMOSQL function $data(emp) \rightarrow charstring$. As an example of the three queries, we show *Q1"*, which involves the client mediator *M0*, and three other server mediators, M1,M2,M3:

```
select s3
from string d,
     string s1,
     string s2,
     string s3,
     employee@M1 e
where d = data(e) and
      s1 = process@M1(d, 100) and
      s2 = process@M2(s1, 100) and
      s3 = process@M3(s2, 100);
```

Each of the three queries was compiled once using only the centralized query decomposition technique, and once using also the tree distribution algorithm. Correspondingly, different execution plans were produced by the different decomposition techniques. In the example case of Q1", the centralized decomposition produced a tree-like data flow graph, shown in Fig. 7a, while the distributed algorithm generated the L-shaped data flow graph, shown in Fig. 7b. Each directed arc of the data flow graphs is marked by the number of tuples sent in the corresponding direction. The numbers in the ovals show the order of execution. Considering that each tuple has size of 100 bytes, the total amount of data sent over the network in case a) is 50000, while in case b) it is only 30000. Even this simple consideration gives us a hint that case b) will be considerably more beneficial than case a).



(a) Centralized plan

(b) Distributed plan

Figure 7: Dataflow graphs for *Q1*, 10000 tuples of size 100 bytes

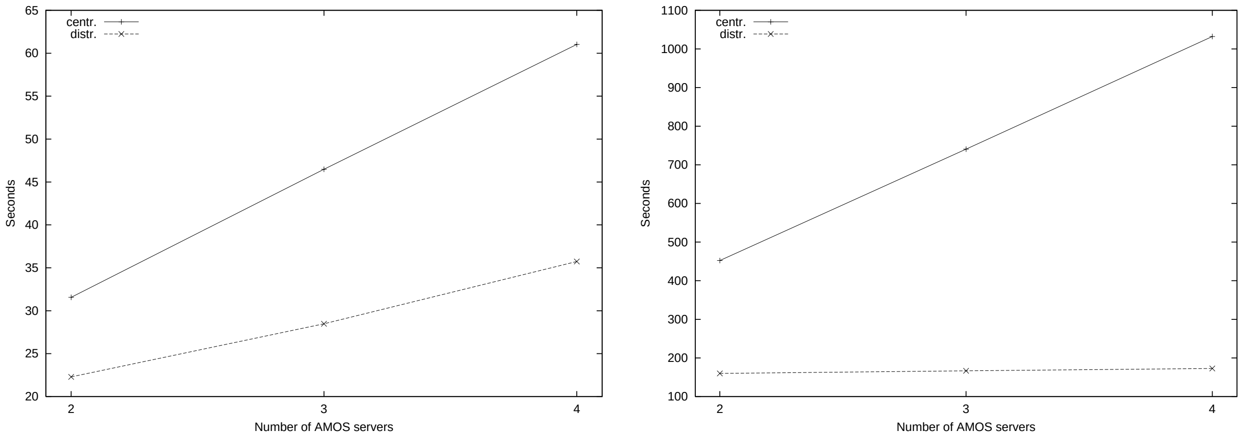
The simplified cost model for evaluating data flow patterns between databases presented here gives us some insight of the possible benefits of the tree distribution algorithm. Later on, in Sec. 4.3 we present experimental confirmations of our expectations.

The second group of experiments used query *Q2*. The major difference between the group of queries *Q1* and query *Q2* is, that in *Q2* we introduced a function defined in the client mediator $process@M0(charstring\, str, integer\, sel) \rightarrow charstring$, which restricts data retrieved from the relational data source. In this case we chose 10% selectivity for the $process@M0$ call:

```
select s2
from string d,
     string s1,
     string s2,
     string m,
     employee@M0 e
where d = data(e) and
      s1 = process@M1(d, 100) and
      m  = process@M0(s1, 10) and
      s2 = process@M2(m, 100);
```

4.3 Experimental results

This subsection presents the results from measurements of execution times for queries *Q1* and *Q2* in the two environments described in section 4.1. During the measurements, each query was executed four times, and the average of the last three measurements was taken.



(a) LAN

(b) ISDN

Figure 8: Execution times for *Q1”*, 10000 tuples of size 100 bytes

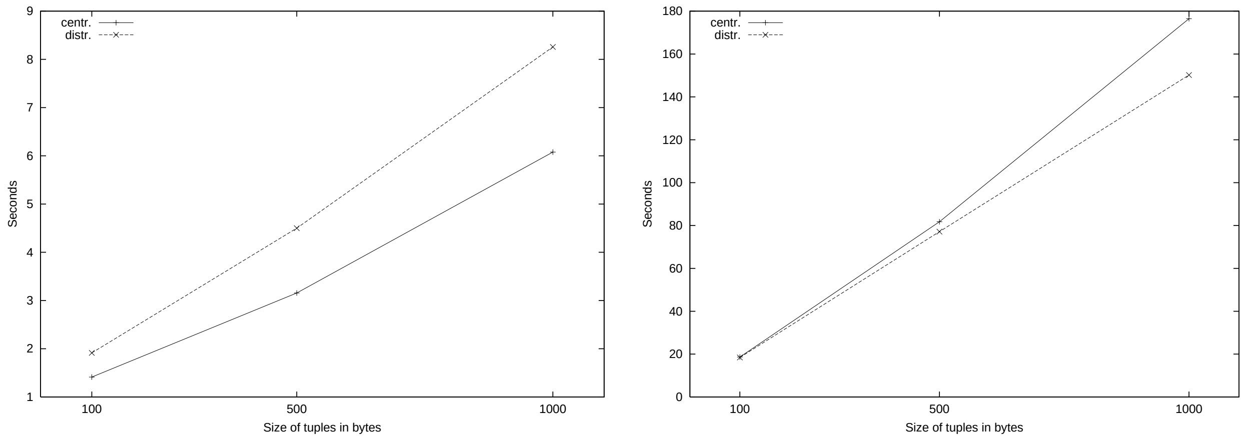
Figure 8a compares the performance of the centralized execution plans for queries Q1’, Q1”, and Q1”’ with the corresponding distributed plans produced by the tree distribution algorithm. As expected the distributed plans generated by the tree distribution algorithm are significantly faster than the centralized plans. In particular, the performance improvements for Q1’, Q1”, and Q1”’ are 33%, 47%, and 49%, respectively. Thus, as the queries span more servers the performance improvement of the tree distribution algorithm increases. In this case the tree distribution algorithm produces plans that scale better since they send less data between the servers and the client mediator M0 than the centralized plans.

In Fig. 8a the connection between M0 and the other servers uses a fast LAN. If a slower connection is used the performance gains will be larger, as shown in Fig. 8b where M0 is connected to the server mediators through a slower ISDN connection. The performance gains are here 50%, 80%, and 86%, respectively. The execution time is virtually constant, independent on the size of the query, since the amount of data shipped between the client and server mediators is constant with the distributed plan.

The latter measurements correspond to a mobile client mediator connected to the server mediators, while the former measurements could be when the same client mediator is docked directly to the company LAN. In these examples the rebalanced plans are always better, since all selections are in the server mediators M1-M4 and there are no selections in the client mediator M0. If there were selections in M0, as in Q2, it could be possible that it would be favorable to use the centralized plan, since the selection in M2 could restrict the number of shipped tuples. To evaluate this we made some tests with query Q2, having a selection in M0.

Fig. 9a compares the centralized plan produced for Q2 with the corresponding distributed plan. As expected, it shows that the centralized plan is better in this case.

In Fig. 9b the query Q2 is tested with an ISDN connection to M0. Because of the slower connection it is here more favorable to use a distributed plan, since the cost to ship data to M0



(a) LAN

(b) ISDN

Figure 9: Execution times for *Q2*, 1000 tuples, 2 servers

is higher than the costs of shipping many more tuples between the server mediators. However, in this case our tree distribution algorithm would produce the suboptimal centralized plan, because no node merge would take place. We are investigating how the tree distribution algorithm can be generalized to handle this case too. Fig. 9a and 9b also illustrates that there are cases where different strategies are needed depending on the speed of the connection to M0. In a dynamic (e.g. mobile) mediator environment this would have to be taken into consideration. The system could here generate two different distributed plans and use one or the other depending on if the mobile client mediator is connected via LAN or ISDN. This is an area for further research.

5 Related Work

The research presented in this paper is related to the areas of data integration and distributed databases. This Section references and briefly overviews some representative examples of projects in these areas, close to the work presented in this paper. A more elaborate comparison of the AMOS*II* system with other data integration systems is presented in [16].

One of the first attempts to tackle the query optimization problem in distributed databases was done within the System R* project [3]. In that project an exhaustive, centrally performed query optimization is made to find the optimal plan. Because of the problem size, AMOS*II* searches only a portion of the whole search space by an exhaustive search strategy. Other phases use heuristics to improve the plan and reduce the optimization time. The SDD-1 system [10] also uses a hill-climbing heuristics as in AMOS*II* to schedule "moves of relations" and "local processing actions" in that compose the distributed query execution schedule. Another classical work on query optimization in a distributed database environment is presented in [1]. In this approach, named AHY (Apers-Hevner-Yao), the system performs first local processing over the relations, then it reduced the results by semi-joins, and finally composes the result at a central site named *evaluation site.* This is clearly different from AMOS*II* where joins are performed in different servers. All three approaches perform the query compilation in at a single site, as opposed to the distributed query compilation in AMOS*II* .

As opposed to the distributed databases, where there is a centralized repository containing meta-data about the whole system, the architecture described in this papers consists of autonomous systems, each storing only locally relevant meta-data. Most of the mediator frameworks reported in the literature (e.g. [13, 25, 11]) propose centralized query compilation and execution coordination. In [5] it is indicated that a distribute mediation framework is a promising research direction, but to the extent of our knowledge no results in this area are reported. Within the same project a centralized query tree rebalancing is proposed [4].

In the DIOM project [21], the importance of the mediator composability is also recognized. A framework for integration of relational data sources is presented where the operations can be executed either in the mediator or in the data source. The query optimization strategy used first builds a join operator query tree (schedule) using a heuristic approach, and then assigns execution sites to the join operators using an exhaustive cost-based search. AMOS*II* , on the other hand, performs a cost-based scheduling and heuristic placement. Furthermore, the compilation process in DIOM is centrally performed, and there is no clear distinction between the data sources and the mediators in the optimization framework.

6 Summary and Future Work

We have given an overview of the architecture of the AMOS*II* mediator system where federations of distributed mediator servers can be composed by AMOS*II* servers. Each AMOS*II* server has DBMS facilities for query compilation, and exchange of data and meta-data with other AMOS*II* servers. OO views can be defined where data from several other mediator servers are abstracted, transformed, and reconciled.

The importance was reiterated of being able to logically compose systems of mediators without global meta-data knowledge in order to build large data integration systems.

It was shown how to decrease the overhead of logically composing mediator servers by a distributed query optimization technique called *Tree Distribution.* Here, a distributed compilation algorithm generates distributed execution plans where the optimized data flow is different from the logical mediator composition, and where each participating mediator interacts only with its neighbor mediator servers.

Performance measurements show that the Tree Distribution algorithm significantly improves query performance and also allows for scale-up with respect to the number of mediator servers involved in a query.

The performance improvements are particularly large in an environment with low bandwidth connections between a client mediator and a set of composed server mediators, as e.g. in a mobile environment where a portable computer is connected via ISDN or a regular phone line to mediator servers communicating via LAN.

We showed that if there are selections in the client mediator different query distributions are optimal depending on the speed of the connection and the selectivity of the selections.

Further research is ongoing to handle the larger class of distributed and multiple query plans required in this situation.

More work is also needed to deal with parallel execution plans, unreliable and sometimes disconnected connections from the client mediator, and deep mediator compositions.

References

[1] P. Apers, A. Hevner and S. Yao: Optimization Algorithms for Distributed Queries. *IEEE-TSE*, SE-9:1, 1983

[2] Silvio Brandani: Multi-database Access from Amos II using ODBC. In *Linköping Electronic Press*, Vol. 3, Nr. 19, Dec. 8th, 1998, *http://www.ep.liu.se/ea/cis/1998/019/.*

[3] D. Daniels et al.: An Introduction to Distributed Query Compilation in R*. In H. Schneider (ed) *Distribute Data Bases*, North-Holland, 1982

[4] W. Du, R. Krishnamurthy and M-C. Shan: Query Optimization in Heterogeneous DBMS. *18th Conf. on Very Large Databases (VLDB'92)*, Vancouver, Canada, 1992

[5] W. Du and M. Shan: Query Processing in Pegasus, *Object-Oriented Multidatabase Systems*, O. Bukhres, A. Elmagarmid (eds.), Pretince Hall, Englewood Cliffs, NJ, 1996.

[6] G. Fahl, T. Risch, M. Sköld: AMOS - An Architecture for Active Mediators. *Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, Haifa, Israel, June 1993.

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

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

[9] S. Flodin, V. Josifovski, T. Risch, M. Sköld and M. Werner: AMOS*II* User's Guide, available at *http://www.ida.liu.se/∼edslab.*

[10] N. Goodman, P. Bernstein, E. Wong, C. Reeve and J. Rothnie: Query Processing in SDD-1: A System for Distributed Databases. *ACM-TODS* 6:4, 1981

[11] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y.Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)* Vol 8 No. 2 117-132, Kluwer Academic Pulishers, The Netherlands,1997

[12] B. Finance, V. Smahi J. Fessy: Query Processing in IRO-DB, *Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)* pp.299-319, 1995

[13] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries accross Diverse Data Sources. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 276-285, Athens Greece, 1997

[14] V. Josifovski: Design, Implementation and Evaluation of a a Distributed Mediator System for Data Integration: the Story of AMOS*II*, Ph D Thesis, University of LinkÖping, Linköping, Sweden, June 1999

[15] V.Josifovski and T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration *Journal of Intelligent Information Systems (JIIS)* Vol 12 No. 2/3, Kluwer Academic Pulishers, The Netherlands, 1999.

[16] V.Josifovski and T.Risch: Comparation of AMOS*II* with Other Data Integration Projects. Available at http://www.ida.liu.se/ edslab/amosII_relwork.pdf

[17] E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.

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

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

[20] S. Nural, P. Koksal, F. Ozcan, A. Dogac: Query Decomposition and Processing in Multidatabase Systems. *OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis*, Montpellier, July 1996.

[21] K. Richine: Distributed Query Scheduling in DIOM. Tech. Report TR97-03, Computer Science Department, University of Alberta, 1997

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

[23] M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th International Conf. on Data Engineering (ICDE'96)*, (IEEE), New Orleans, Louisiana, Feb. 1996.

[24] R. Soley, C. Stone (eds.): Object Management Architecture. *John Wiley & Sons*, New York, 1995

[25] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *Transactions on Knowledge and Data Engineering* (TKDE) vol. 10 No. 5, pp 808-823, 1998

[26] G Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), Mar. 1992.