UPPSALA UNIVERSITY
Department of Information Science

# Distributed View Expansion in Composable Mediators

Timour Katchaounov

Vanja Josifovski

Tore Risch

Division of Computer Science
Research Report 2000:2

Uppsala University
Department of Information Science
P.O. Box 513
SE-751 20 Uppsala, Sweden

ISSN 1403-7572

# Distributed View Expansion in Composable Mediators

Timour Katchaounov, Vanja Josifovski and Tore Risch
Uppsala Database Laboratory
Uppsala University
Sweden

*first_name.last_name*@dis.uu.se

**Abstract**

Data integration on a large scale poses complexity and performance problems. To alleviate the complexity problem we use a modular approach where many heterogeneous and distributed data sources are integrated through composable mediators. Distributed mediators are defined as object-oriented (OO) views defined in terms of views in other sub-mediators or data sources. In order to minimize the performance penalty of the modular approach we have developed a distributed expansion strategy for OO views where view definitions are selectively imported from sub-mediators. The performance of the approach is analyzed showing significant improvements over a naive strategy without distributed view expansion. In the naive strategy the sub-mediators do not export their view definitions but only execute queries and provide query costing information. We analyze performance gains with respect to network overhead, intra-mediator overhead, and utilization of data source query capabilities. The analysis shows that the distributed view expansion can support modularity through distributed and composable mediators with little overhead.

## 1 Introduction

There has been substantial interest in using the mediator/wrapper approach for integrating heterogeneous data [15, 32, 13, 27, 8]. Most mediator systems integrate data through a single mediator server accessing one or several data sources through a number of 'wrapper' interfaces that translate data to a common data model (CDM). However, one of the original goals for mediator architectures [33] was that each mediator should be a relatively simple modular abstraction of the integration of some particular kind of data. Larger systems of mediators would then be defined through these primitive mediators by composing

new mediators in terms of other mediators and data sources. Different mediator servers distributed on the network would define different logical views of data. Such a modular logical composition of mediators allows to overcome complexity problems of data integration on a large scale with many data sources and mediators involved. However, very few projects have used a distributed mediator architecture, e.g. [22], and there is little work on implementation issues of distributed mediators.

This paper investigates some query processing issues in a distributed mediator system, AMOS II [28]. Distributed mediators are composed as object-oriented (OO) views in terms of views in other sub-mediators or data sources. The views make the distributed mediators appear to the user as a single virtual database consisting of a number of types (classes) and functions (methods, attributes). However, unlike regular OO systems the extents of these types and functions are not explicitly stored in a database but are derived, through an OO multi-database query language, from data in the underlying data sources and other OO mediators [9, 17, 19]. Even though such an architecture addresses the complexity problems of data integration it also has some performance problems which we have addressed:

- In an architecture with many layers of intermediate mediator servers there might be performance problems because of overhead in the communication between, and computation inside, the distributed mediator servers. We minimize some of this overhead through distributed query optimization [16, 18].
- The mediators are autonomously maintained units having their own schemas and local databases. Unlike distributed databases, the distributed mediators therefore do not have any central schema and each mediator server has only limited knowledge about the structure of other mediators. This makes it difficult to find an optimal distributed query execution plan. In our approach the distributed mediator servers communicate with other known mediator servers to import some of the schema information, such as some OO view definitions. It will be shown that this can significantly improve query performance.
- We use an OO common data model (CDM) which involves query processing over objects, types and functions rather than just relational tables. The OO query language has mediating primitives to define the virtual database layers. We have developed several query optimization methods for efficient processing of such OO queries and views [21, 11, 9, 17, 19] which will not be elaborated here.

In [18] we described how to decompose distributed queries and then re-balance the decomposed query execution plans to minimize the communication overhead by generating

an optimized data flow pattern between the distributed mediator servers. In that strategy the sub-mediators did not export their view definitions but only executed queries and provided query costing information. Such a strategy can be suboptimal when there are more than two mediator layers. In this paper the importance is analyzed of a method based on distributed view expansion (DVE) to minimize the penalty of several mediator server levels. The combination of DVE, query decomposition and rebalancing is shown to significantly improve query performance. Compared to a black box treatment of data sources, as in, e.g., CORBA based systems [31], our approach improves the performance by importing some (but not all) information from other mediators and data sources; we treat the data sources as *grey* rather than black boxes. Often this completely eliminates the access to intermediate mediator layers. The method can drastically reduce query execution time when information from several hidden sub-mediators can be combined. The performance improvements are due to more selective queries, smaller data flows between the servers, and fewer servers involved in the data exchange.

For the analysis we have implemented a simple scenario of distributed mediators using our AMOS II research prototype and DB2 based data sources. We show significant performance gains over the non-DVE approach in [18] and that DVE is necessary when integrating data from large databases. The performance gains are analyzed with respect to network overhead, intra-mediator overhead, and utilization of data source query capabilities. The analysis shows that the distributed view expansion can support modularity through distributed and composable mediators with little overhead.

We conclude this section with an overview of AMOS II's data model and query processing steps. Section 2 introduces the scenario based on the AMOS II data model that is used throughout the paper. Section 3 describes the principles of DVE and Section 4 investigates its performance. Section 5 investigates related work followed by summary and future work in Section 6.

## 1.1 AMOS II query processing

As our research platform we use the AMOS II mediator database system [28]. The core of AMOS II is an OO DBMS which is extensible and distributed. The mediators are implemented as fully functional AMOS II servers, communicating through TCP/IP. For good performance, and since most the data reside in the data sources, each AMOS II server is designed as a main-memory DBMS.

Some of the AMOS II mediators can contain interfaces that wrap different kinds of data sources, e.g. ODBC compliant relational databases [3] or XML files. The term

*translators* is used for mediators containing such wrappers. They translate data from one or several data sources to the CDM through the extensible OO query language AMOSQL [9].

Other mediators are used to intersect data and to reconcile conflicts and overlaps between similar real-world entities modeled differently in different sub-mediators [17, 19].

Users and applications can pose OO queries to any AMOS II server. We call the servers to which some queries are posed the *client mediators* for those queries.

The data model of AMOS II [28] is an OO extension of the DAPLEX functional data model [29]. The query language AMOSQL [12, 28] has its roots in OSQL [24]. It is similar the object part of SQL99 with extensions of multiple inheritance, multi-way foreign functions [21], multi-database queries [17, 19], foreign data source translators [9], late binding [12], etc.

The CDM is based on the three basic concepts of *objects*, *types*, and *functions*. Data is represented as objects which can be classified into one or more types. The *extent* of a type is the set of objects classified into that type. The types are organized in a multiple inheritance type hierarchy. The types can either be explicitly *stored* in the database or implicitly *derived* from the database schema and contents through the query language. Derived types are important for data integration [17, 19]. The functions define properties of objects, computations on objects, and relationships between objects. The extent of a *stored* function is explicitly stored in the mediator (c.f. object methods, tables); *derived* functions are implemented by AMOSQL queries (c.f. object methods, views); and *foreign* functions are implemented in some programming language, e.g. C or Java c.f. methods).

For example the following schema defines a type `part` having the attributes (stored functions) `pnum`, `name`, and `price`:

```
create type part;
create function pnum(part)->integer as stored;
create function name(part)->string as stored;
create function price(part)->string as stored;
```

The following AMOSQL query retrieves the name of the part being cheaper than 500:

```
select name(p)
from part p
where price(p)<500;
```

Queries in AMOS II are parsed and translated into a typed predicate calculus based representation, ObjectLog [21], which is then rewritten through a number of transformation

rules [21, 9, 17, 19]. The calculus representation of the above simple query after the transformations is:

$\{ V1 \mid$
$V1 = name_{part \to string}(p) \wedge$
$V2 = price_{part \to integer}(p) \wedge$
$V2 < 500\}$

The optimized calculus expressions are transformed by a cost-based query optimizer into an optimized object algebra expression [21, 9] which is finally interpreted to produce the query result.

For multi-database queries, before the query optimization phase, the calculus expressions operating over data outside the mediator are decomposed into distributed subqueries. The decomposition uses a combination of heuristic and dynamic programming strategies [16]. At each site the cost-based optimizer generates optimized execution plans for the subqueries. The decomposition is performed in four stages:

- *Subquery generation*: the query is broken into several subqueries.
- *Subquery placement*: In this phase subqueries are assigned sites where they will be further processed. While some of the subqueries are executable at only one data source, others can be executed in more than one data source or AMOS II server (e.g. comparison operators can be executed in any AMOS II server or relational database). The second type of subqueries are named multiple implementation functions, MIFs, (queries are also functions) and this phase assigns such subqueries to one or more sites.
- *Subquery scheduling*: the queries are scheduled for execution in the different AMOS II servers and the inter-site data flow between the servers is determined. Only left deep query execution schedules are explored during this stage.
- *Rebalancing*: a distributed re-compilation is applied to re-balance the subquery execution schedule. The generated schedules can contain "sidewise" data flows between servers that are on the same level in the mediator hierarchy [18].

An interested reader is referred to [28, 12] for a more detailed description of the AMOS II system and to [21, 9, 11, 16, 17, 18, 19] for more on its query processing.

# 2 Multi-layered mediators scenario

The rest of this paper is based on a simple multi-layered scenario where several mediators share a common sub-mediator. While in our framework it is possible to compose much

more complicated networks of mediators, we base our choice of example scenario on the following:

- It uses a kind of logical composition we believe will appear when separate autonomous entities compose mediators in order to share information.
- The third mediator level contains information that is hidden from the client mediator.
- The query posed to the client mediator combines overlapping information hidden in the third level.
- The example allows us to vary the selectivity of subqueries to investigate the scalability of the approach.

Let us consider the following data analysis scenario: an analyst wants to make a survey of the price/quality distribution of car parts. S/he is using a client mediator $C$ on her/his notebook to integrate data from two mediators hosted by two independent entities: $P$ provides price information and $Q$ provides information about quality of parts. Both $Q$ and $P$ happen to integrate data from the same warehouse data source - a mediator $T$ hosted by a third entity which acts as a translator on top of a relational database $R$.

The logical composition of mediators is shown in Figure 1. Notice that the arrows mean "defined in terms of", i.e. they define the logical composition of the mediators.

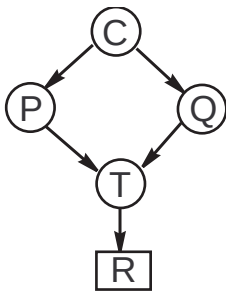


Figure 1: Logical composition of mediators

All the actual data is stored in the RDBMS $R$ in a $PARTS$ table with the following structure:

```
CREATE TABLE parts (
pnum     INTEGER NOT NULL,
name     CHAR(16) NOT NULL,
quantity INTEGER,
quality  INTEGER,
```

```
price    REAL,
PRIMARY KEY(pnum))
```

In the translator $T$ the PARTS table is automatically translated into a corresponding type `parts@R` [9] with the derived functions `pnum`, `name`, `quantity`, `quality`, and `price`.

The two integrators $P$ and $Q$ import the `parts@T` type from the translator $T$ and use derived types [17] to define Object-Oriented views over the schema exported by the translator. Mediator $P$ has the following definition:

```
create derived type part_price
subtype of part@T p;

create function pnum(part_price p) -> integer
as select part@T.pnum(p);

create function name(part_price p) -> charstring
as select part@T.name(p);

create function price(part_price p) -> real
as select part@T.price(p);
```

The `@` notation is used for referencing types and functions in other mediators. The type names in front of the function name is used to explicitly resolve the function name and avoid ambiguity. The derived type `part_price` has the derived functions (methods) `pnum`, `name`, and `price` imported from $T$. The 'quality' mediator has the following analogous definition:

```
create derived type part_quality
subtype of part@T p;

create function pnum(part_quality p) -> integer
as select part@T.pnum(p);

create function name(part_quality p) -> charstring
as select part@T.name(p);

create function quality(part_quality p) -> integer
as select part@T.quality(p);
```

The client mediator $C$ imports the two derived types `part_price` and `part_quality` from $P$ and $Q$, respectively. The analyst poses the following query, used as a running example, to the client mediator $C$:

```
select name(p)
   from part@P p, part@Q q
   where price(p) >= :price_low and
         price(p) < :price_high and
         quality(q) >= :quality_low and
         quality(q) < :quality_high and
         pnum(p) = pnum(q);
```

The query retrieves the names of all parts in a certain price and quality range parameterized with the variables `:price_low`, `:price_high`, `:quality_low`, and `:quality_high`. The objects representing the same part in the two integrated databases are matched using their part numbers `pnum`.

# 3 Distributed View Expansion

This section describes the mechanism for view definition exchange and expansion in a hierarchy of AMOS II servers using the mediator composition example in Figure 1. The discussion is based on the example scenario from the previous section.

For the example client query the calculus generator generates the following ObjectLog expression, where intermediate variables have been renamed for improved readability, as the AMOS II system uses automatically generated names.

$$
\begin{aligned}
&\{\ p\_low\ \ p\_high\ q\_low\ \ q\_high\ \ name \mid \\
&\ \ name = name_{part@P \rightarrow charstring}(p\_part)\ \wedge \\
&\ \ price1 = price_{part@P \rightarrow real}(p\_part)\ \wedge \\
&\ \ price2 = price_{part@P \rightarrow real}(p\_part)\ \wedge \\
&\ \ pn = pnum_{part@P \rightarrow integer}(p\_part)\ \wedge \\
&\ \ quality1 = quality_{part@Q \rightarrow integer}(q\_part)\ \wedge \\
&\ \ quality2 = quality_{part@Q \rightarrow integer}(q\_part)\ \wedge \\
&\ \ pn = pnum_{part@Q \rightarrow integer}(q\_part)\ \wedge \\
&\ \ price1 >= p\_low\ \wedge \\
&\ \ price2 < p\_high\ \wedge \\
&\ \ quality1 >= q\_low\ \wedge \\
&\ \ quality2 < q\_high\}
\end{aligned}
$$

This calculus expression is divided by the decomposer into two sub-expressions and a set of MIFs. Each sub-expression represents a sub-query: one containing the predicates representing functions from the "price" mediator $P$ (sub-query $P$ below) and a second one from the "quality" mediator $Q$ (sub-query $Q$ below). The third group of predicates contains predicates that can be executed in any of the mediators (MIFs below). They are later placed by the decomposer in one of the two sub-queries. The expression head of each of the subqueries contains all the variables that the subquery has in common with the rest of the query, or that are part of the result of the whole query.

sub-query $P$:

$\{\ price1\ \ price2\ \ pn\ \ name|$

$name = name_{part@P \rightarrow charstring}(p\_part) \wedge$

$price1 = price_{part@P \rightarrow real}(p\_part) \wedge$

$price2 = price_{part@P \rightarrow real}(p\_part) \wedge$

$pn = pnum_{part@P \rightarrow integer}(p\_part)\}$

sub-query $Q$:

$\{\ quality1\ \ quality2,\ pn|$

$quality1 = quality_{part@Q \rightarrow integer}(q\_part) \wedge$

$quality2 = quality_{part@Q \rightarrow integer}(q\_part) \wedge$

$pn = pnum_{part@Q \rightarrow integer}(q\_part)\}$

MIFs:

$price1 >= p\_low$

$price2 < p\_high$

$quality1 >= q\_low$

$quality2 < q\_high$

This does not mean that the values of all these variables will be calculated in this subquery. The variables represent both the input and the result of the subquery. The input variables are bound during execution to the intermediate results generated by the other two expressions. Which variables in the subquery expressions head are bound and which are free is determined by the order of the subquery execution generated by the query decomposer.

To allow modeling of the mediator hierarchies only with concern to the logical model of the enterprise relieving the user from performance considerations, intermediate servers must share information with the client mediator.

In the example above, this implies that the derived type definitions in the *price and* quality mediators $P$ and $Q$ should be sent to the client mediator. Note that if any of $P$ or $Q$ decides to change the source of their data or add more data sources to their integrated

view, no logical changes have to be made at the client mediator. Although this is a simple example, it illustrates how modularity of change is achieved in an environment without a global meta-data repository.

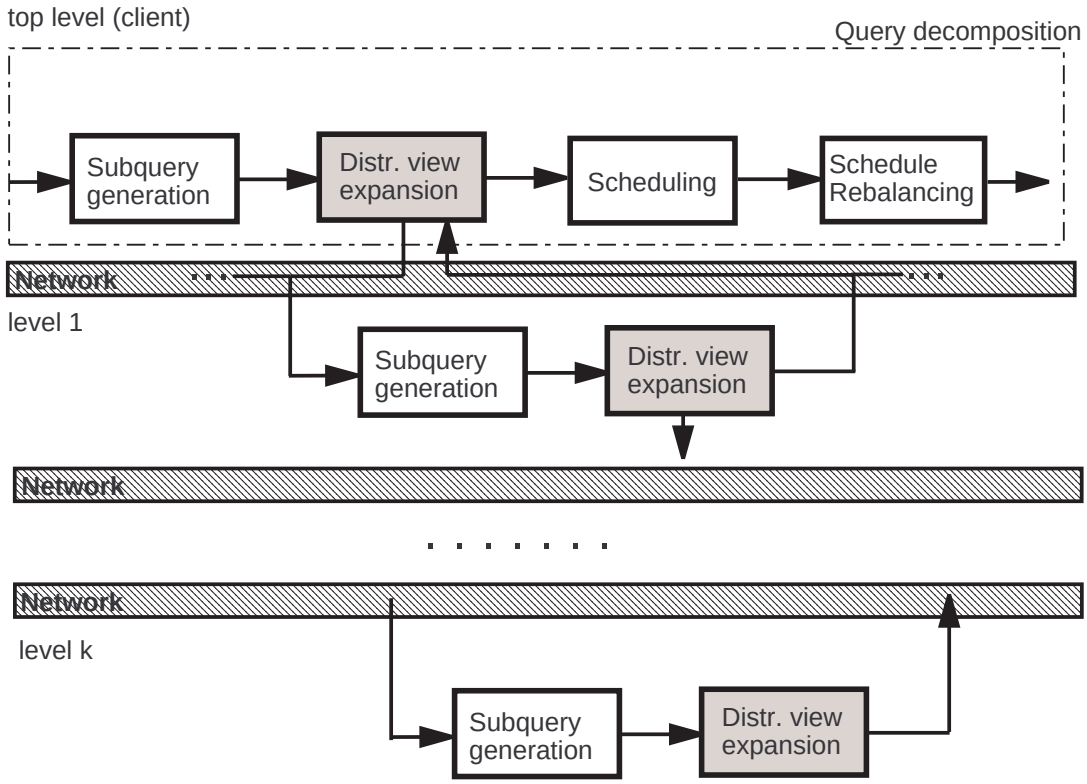


Figure 2: Distributed query expansion process

The view definitions are internally represented as calculus expressions. Figure 2 illustrates the extended query processing with distributed view expansion. The view expansion is placed after the subquery generation in order to avoid processing of individual predicates. Furthermore the grouped predicates are compiled and processed together at the receiving mediator, allowing optimizations through query rewrites. In the distributed view expansion phase (Figure 2), the client mediator sends an expansion request to each server where a subquery is to be executed and an expanded definition of the subquery is retrieved. The subqueries are communicated between the servers in form of calculus expressions. At the server accepting the subquery expansion request, the subquery processing starts with calculus optimization and continues in the same manner as with other queries until the distributed view expansion phase. In this phase, if the subquery where all the derived functions and types have been substituted with their definitions/extent

functions contains itself subqueries in other AMOS II mediators, it recursively expands the subqueries. The process might span several levels, and, assuming acyclic type importation, it terminates when no expansions are performed (the next subsection discusses strategies for selective view expansion).

After collecting the expanded definitions of all the subqueries executed in other AMOS II servers, the next phase in the view expansion combines again all these expressions into one in order to be regrouped by the subsequent optimization phases. The benefits from this "one step back" in the query processing are the following:

1. Calculus based rewrites can be performed at the client to eliminate overlap in the calculation among the different servers.
2. Each subquery to be executed at another AMOS II server could be expanded into an expression containing functions defined in multiple AMOS II servers. These expressions might in turn have sub-expressions that are executed at a same AMOS II server. Putting them together in a single predicate allows for optimizations that eliminate overlap or achieve a better execution strategy.
3. The expanded expressions returned from the DVE may contain MIF predicates that could be combined/replicated with predicates at different data sources where they can act as selections, reducing the query execution time and the intermediate result sizes.
4. A richer space of data flow patterns will be considered by the query decomposer.

## 3.1 Selective view expansion

Having explained the extension of the query processor needed to perform the distributed view expansion, we turn our attention to the problem of deciding when to perform expansion of the views. The relative cost of the expanded and unexpanded execution schedules of a query depend on several factors. While, as noted previously, the view expansion can eliminate some spurious computation and data shipment, it could also introduce some extra costs in the optimization.

An exhaustive algorithm determining which views to expand would need to fully compile the query for each combination of the views at the intermediate nodes. To reduce the complexity of this problem each node decides locally if the view is to be expanded. Only locally available data is used in this process.

When a subquery is sent to an AMOS II server, is is expanded and compiled into an executable plan by the local optimizer. As a part of the compilation process, if the

subquery is expanded into a multi-database query, the query decomposer produces a series of sub-subqueries. At this point a budget based decision procedure is used to control the depth of view expansion as follows:

- The client starts with initial budget and sends view expansion requests to each server.
- The initial budget is distributed evenly between the servers and each server starts query processing with it's share of the budget.
- Each time view expansion has to be performed, the corresponding mediator uses it's budget to send view expansion requests to its sub-mediators.
- If the budget is not enough to request expansion of all external views, then the sub-mediators are ordered by the number of participating sub-submediators in each external view (this information is provided together with the view (function) signatures, and is locally available), and the views with higher number of mediators are given preference for expansion.
- No expansion requests are sent (correspondingly no budget is wasted) to mediators without sub-mediators.
- View expansion stops when a server has exhausted its budget or no more view expansions can be done (a sub-query consists only of local predicates).

The main goal of this approach is to favor view expansion in cases of deep mediator hierarchies, and prohibit view expansion explosion when the number of direct sub-mediators is too large.

# 4 Experimental Evaluation

In order to analyze the performance of our approach we tested our distributed view expansion strategy in our scenario by integrating data sources through three layers of mediators, composed as follows:

1. The top layer is a client mediator $C$ that accepts user queries to object views integrating data from the next layer server mediators. The client mediator was running on a Dell Inspiron 366 MHz notebook with 128 MB of RAM.
2. The middle layer runs the two mediators $P$ and $Q$ integrating views of data from the third layer. The middle layer mediator servers are run on a Dell Gx1 workstation with 600 MHz CPU and 512 MB RAM.

3. The third layer runs the translator mediator $T$ wrapping a DB2 database $R$. The translator accesses the DB2 RDBMS through an ODBC wrapper [3]. The translators and the DB2 database server run on a Dell Poweredge server with two 500 MHz CPUs and 512 MB RAM.

All network communication used a fast 100 Mbit LAN. In order to simplify the experimental setting, the second layer mediators were both executing on the same computer. This does not influence the measurements in our case, as our prototype system uses synchronous communication (i.e. no two mediators execute at the same time) and it was ensured that enough memory is allocated for each process so that no swapping occurred.

The PARTS table in the relational database is populated with synthetic data, where the total number of rows is 50000, the *price* column ranges between 1 and 100 and the *quality* column ranges between 1 and 10. Both price and quality are evenly distributed and indexed.

For the evaluation we use the client mediator query which was executed with various parameter choices to vary the selectivity of the subqueries in $P$ and $Q$. The parameters were chosen so that the selectivity of the subqueries over $P$ and $Q$ were symmetric in all cases. The parameters were adjusted so that the resulting query had the selectivities 0.01, 0.25, 0.75 and 1 of all tuples in the table `parts`.

The test query was precompiled with and without view expansion and the dependency of the query execution time on the query selectivity was measured in both cases. Figure 3 compares the execution times for the test query with varying selectivity with and without DVE. (The 'Hash Join' curve is explained later). Distributed view expansion improves the execution time between 144 times (selectivity 0.01) and 20 times (selectivity 1).

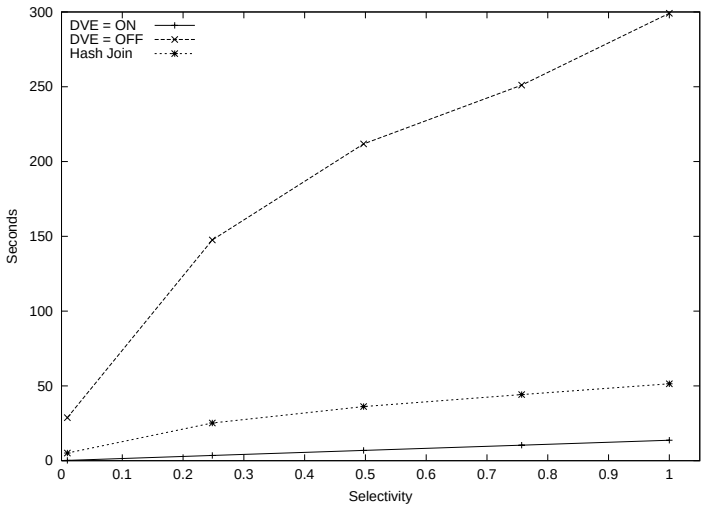


Figure 3: Execution times with/without view expansion

The data flow of the generated execution plan without distributed view expansion is shown in Figure 4A. The main problem here is that the query decomposer was unable to detect i) that the data in sub-mediators $P$ and $Q$ actually came from the same source $T$ and ii) that the mediators $P$ and $Q$ could be bypassed as in the optimal query plan in Figure 4B, generated with view expansion. The arrows in the figures indicate function calls and transmission of query parameters in the forward direction and flow of tuples in the backward direction. The function parameters are shipped in bulks to minimize the transmission overhead [16]. The number next to each arrow shows the order in which mediators and the data source communicate. The dotted circles depict those mediators that were eliminated by the DVE.

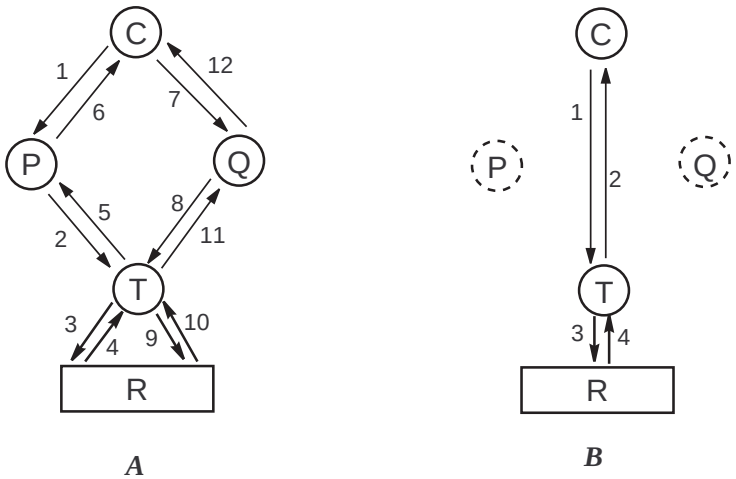


Figure 4: Data flow patterns

Without distributed view expansion the system sends two separate subqueries to the translator $T$ which are in turn translated into the queries shown in Figure 5 and executed in the relational database. The queries are parameterized and precompiled (prepared) by the RDBMS. The first query in Figure 5, selecting parts in a price range, is first executed in the RDBMS as the values of the `:price_low` and `:price_high` parameters are transported from $C$ to $T$ (steps 1, 2) into $R$ (step 3). The result is transported back to $C$ in an upward data flow from the relational database through mediator $T$ (step 4) and $P$ to $C$ (steps 5,6). Its result tuples are then transported from $C$ down through $Q$ to $T$ together with the `:quality_low`, and `:quality_high` parameters (steps 7,8). The translator $T$ will probe, through the second query in Figure 5, which tuples in the selected price range are also in the quality range[1] (steps 9,10). A bitmap of the tuples passing this test is then sent back to $C$ to indicate what selected tuples passed the quality test (steps

[1]We probe by testing if the query returns -1 and then immediately closing the scan

```
SELECT P.NAME, P.PNUM
FROM PART P
WHERE P.PRICE >= ? AND P.PRICE < ?

SELECT -1
FROM PART P
WHERE P.QUALITY >= ? AND
      P.QUALITY < ? AND
      P.PNUM = ?
```

Figure 5: Translated SQL queries, no DVE

```
SELECT P.NAME, P.PNUM
FROM PART P
WHERE P.QUALITY >= ? AND P.QUALITY < ?
```

Figure 6: Translated SQL query, Hash-join

11,12). This kind of rather inefficient strategy generates a fully streamed plan requiring limited memory in each mediator and no sorting.

The naive strategy without DVE can be improved if we assume that $C$ is large enough to hold all selected tuples. Then a *ship-in* strategy can be used to perform a hash join in $C$. Figure 3 also compares a hash join without DVE with the naive streamed non-DVE algorithm, and the optimal algorithm above using DVE. The optimal algorithm is between 27 times (selectivity 0.01) and 4 times (selectivity 1) faster than the hash join. The resulting data flow pattern is the same as on Figure 4A, but query execution proceeds in the same way as in the previous example only from steps 1 to 6. After the first sub-query is executed its temporary result is stored in a hash-index in $C$. Execution continues with steps 7 and 8 where only the `:quality_low`, and `:quality_high` parameters are sent to $T$, which in turn executes the translated query in Figure 6. This query retrieves all parts within a certain quality range (steps 9,10), and sends them to $C$ (steps 11,12) to be joined with the temporary result of the first sub-query.

A third alternative is to use merge sort in the mediator, but this could involve sorting in the RDBMS. That strategy would in any case have similar performance figures to the hash join.

By contrast query optimization with distributed view expansion will only submit a single query to the translator $T$ translated to the query in Figure 7 to the relational database. The query decomposer combined with a distributed query plan rebalancing algorithm [18] has eliminated all accesses to mediators $P$ and $Q$ and the data flow will

```
SELECT P.NAME
FROM PART P1, PART P2
WHERE P1.QUALITY >= ? AND P1.QUALITY < ? AND
      P2.PRICE >= ? AND P2.PRICE < ? AND
      P1.PNUM = P2.PNUM
```

Figure 7: Translated SQL queries, with DVE

pass directly from $T$ to $C$ as illustrated in Figure 4B.

Without view expansion the execution plan in Figure 4A follows the paths of the logical decomposition shown in Figure 1 which is shown to be very inefficient in this case.

By contrast, distributed view expansion removes all intermediate mediator layers in the distributed execution plan, thus reducing significantly the number of messages and processing time in the mediator servers. The main gain from DVE is when a client accesses more than one middle layers which in turn share a common sub-mediator, as in the example. The view expansion creates then a subquery that combines all sub-queries (compare Figures 5 and 7) from the middle layer mediators. This leads to two major gains:

- Selections can be pushed down to the data sources. In the example, the selections over both $P$ and $Q$ are combined and pushed to $R$.
- The total selectivity of the query is reduced. In the example, if the selections over $P$ and $Q$ are not combined with DVE, the selectivity of each sub-query through $P$ and $Q$, respectively, is an order of magnitude larger than the combined query.

The disadvantage with distributed view expansion is that query optimization is slower. If there are many mediator layers involved in a query this means that the expanded query will be large and distributed over many mediators. Though optimizing such a large expanded query would create an optimal execution plan, it would be very slow to optimize incurring much CPU and communication time. An interesting future work would be to investigate strategies for an adaptive DVE.

In order to gain deeper understanding where most of the time was spent during query processing, we measured execution time distribution between time spent in the mediators, time spent in network transmission, and time spent in executing ODBC calls. As a typical example Table 1 shows execution time distribution for the example query with selectivity 1%. Executing the same query with other selectivities showed similar distribution patterns. One may observe that savings in time are achieved in every component of the mediator composition (mediators, network, data source). The relatively high network

time in the streamed plan without DVE is due to the fact that resulting tuples from the first subquery in Figure 5 are probed one by one by the mediator $Q$ (because of the streamed nature of the algorithm), thus incurring a number of messages corresponding to the number of tuples. The high number here reflects the high network setup cost per message. The number of ODBC calls is one for the first sub-query plus an execution of the probing query for each tuple.

If we consider the hash-join variant, it executes only 2 ODBC calls each of them retrieving 10% of the database. As far as communication in AMOS II is bulk-oriented, the network cost of shipping approximately 20% of the database is very low. Notice that the hash-join variant requires that one of the subqueries is materialized in the client, which may require large amounts of memory.

Executing the query with DVE combines the benefits of both streamed execution and hash-join in the client mediator, while at the same time providing considerably better performance with small memory requirements.

| | **Execution times for 1% selectivity** | | |
|---|---|---|---|
| **Time, sec.** | **no DVE, streamed** | **no DVE, Hash join** | **with DVE** |
| **Total** | 29.95 | 5.36 | 0.24 |
| **Mediator** | 3.77 | 1.20 | 0.11 |
| **Network** | 19.69 | 0.86 | 0.03 |
| **ODBC** | 6.50 (4964 calls) | 3.29 (2 calls) | 0.10 (1 call) |

Table 1: Execution times and time distributions for 1% selectivity

# 5 Related Work

This work is related to work on query optimization in distributed databases and mediators. Distributed databases [26, 6, 14, 1] have complete global schemas describing on what sites different (fractions of) tables are located, while distributed mediators do not have complete knowledge of meta-data from all mediators and data sources. Full opening of all possible views in a distributed system with many nodes may be very costly. System R* [6]uses such an exhaustive, centrally performed query optimization to find the optimal plan while SDD-1 [14] uses hill-climbing heuristics. In [23] a restricted view opening strategy for the System R* distributed database is briefly mentioned but not evaluated. Furthermore, modern computer architectures with fast computers, large main memories, and widely varying network speeds makes our results different from those of classical distributed databases.

Mediator systems are usually not distributed (e.g. [15, 32, 25]) and thus do not use our strategies. In [8] it is indicated that a distributed mediation framework is a promising research direction without reporting any results. The DIOM system [27] is also a distributed mediator system using distributed query scheduling similar to our decomposer. However, no evaluation of distributed view opening is reported.

# 6 Summary and Future Work

We described and evaluated a distributed view opening technique in composable mediators. We have shown that even in a simple mediator composition this approach leads to significant performance improvements, compared to a "black-box" approach to distributed query optimization. The main contribution of this work is that it shows that mediators may be logically composed to solve integration problems with very little execution overhead.

Though not addressed in this paper, we have also investigated performance problems in more complex composition scenarios. Preliminary results show that DVO will be beneficial to use to discover optimal data flow in a non-homogeneous network with different communication speeds between the mediator nodes, even though the query optimization time increases. In order to deal with communication speeds that vary over time more research needs to be done on adaptive decision procedures for DVE.

Another direction of our research is to expand the results of this paper to asynchronous communication model and explore parallel execution plans.

An issue not addressed in our current work is compilation time of large distributed queries. Our current experience shows that while mediator compositions of less than 10 mediators works reasonable with our framework, larger systems of distributed mediators require scalable and distributed compilation techniques.

# References


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

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

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

[4] O. Bukhres, A. Elmagarmid (eds.): Object-Oriented Multidatabase Systems. Pretince Hall, 1996.

[5] M. Garcia-Solaco, F. Saltor, M. Castellanos: Semantic Heterogeneity in Multidatabase Systems. In [4].

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

[7] U. Dayal, H. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Trans. on Software Eng.* 10(6), 1984.

[8] W. Du, M. Shan: Query Processing in Pegasus. in [4].

[9] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *VLDB Journal*, 1997.

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

[11] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions. *VLDB95*, Zürich, Switzerland, 1995

[12] S. Flodin, V. Josifovski, T.Katchaounov, T. Risch, M. Sköld and M. Werner: AMOS II User's Manual, available at *http://www.dis.uu.se/~udbl/amos/*, 1999.

[13] 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)* 8(2), 117-132, 1997

[14] 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

[15] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *VLDB97*, 276-285, Athens Greece, 1997

[16] V. Josifovski: Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration. Ph D. Thesis 582, Linköpings universitet, 1999. Available at *http://www.dis.uu.se/~udbl/publications.html.*

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

[18] V.Josifovski, T.Katchaounov, T.Risch: *Optimizing Queries in Distributed and Composable Mediators*, *4th Conference on Cooperative Information System (CoopIS'99)*, Edinburgh, Scotland, Sept. 1999.

[19] V.Josifovski, T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations, *25th VLDB Conf.*, Edinburgh, Scotland, Sept. 1999.

[20] E-P. Lim et al: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, 25(5), 553-562, 1995.

[21] W. Litwin, T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE TKDE*, 4(6), 517-528, 1992

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

[23] G.Lohman, C.Mohan, L.Haas, D.Daniels, B.Lindsay: Query Procesing in R*, in W.King, D.S.Reiner, D.S.Batory (eds.): *Query Processing in Database Systems*, Springer Verlag, 1985.

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

[25] 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.

[26] M.T.Özsu, P.Valduriez: *Principles of Distributed Database Systems*, 2nd edition, Prentice Hall, 1999.

[27] K.Richine: *Distributed Query Scheduling in DIOM*, Tech. report TR97-03, Computer Science Dept., University of Alberta, 1997.

[28] T.Risch, V.Josifovski, T.Katchaounov: AMOS II Concepts, available at *http://www.dis.uu.se/~udbl/amos/*.

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

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

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

[32] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE TKDE*, 10(5), 808-823, 1998

[33] G. Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 1992.