# Evaluation of Join Strategies for Distributed Mediation

Vanja Josifovski*, Timour Katchaounov, and Tore Risch

Uppsala Database Laboratory, Uppsala University, Sweden,
vanja@us.ibm.com, timour.katchaounov@dis.uu.se, tore.risch@dis.uu.se

**Abstract.** Three join algorithms are evaluated in an environment with distributed main-memory based mediators and data sources. A streamed ship-out join ships bulks of tuples to a mediator near a data source, followed by post-processing in the client. An extended streamed semi-join in addition builds a main-memory hash index in the client mediator. A ship-in algorithm materializes and joins the data in the client mediator. The first two algorithms are suitable for sources that require parameters to execute a query, as web search engines and computational software, and the last is suitable otherwise. We compare the execution times for obtaining all and the first N tuples, and analyze the percentage time spent in subsystems, varying the network communication speed, bulk size, and data duplicates. The join algorithm leads to orders of magnitude performance difference in different mediation environments.

## 1 Introduction

Integration of data from sources with varying capabilities has been intensively studied by the database community in the recent decade. The Amos II system [8, 9,17] uses the *wrapper-mediator* paradigm to integrate data from several sources. One of the salient features of Amos II is a distributed architecture where a number of interconnected *mediator servers* cooperate in providing the users and the applications with the required view of the data in the sources. We believe that a distributed mediator architecture is needed because it is unrealistic to assume that a single mediator server can be deployed in an enterprise composed of multiple organizational units. When many mediator servers become available on the network, composability will be required for designing new distributed mediator servers in terms of the existing ones, thus reusing mediation specifications. Multiple mediators will also alleviate the performance bottleneck problems that appear when all the queries are handled by a single mediator.

Having some of the basic assumptions different from the classical database systems, query processing in a distributed mediator system requires some novel strategies and solutions. One of the major reason for this is the different cost model in this environment. The I/O and CPU costs used in the traditional query optimization [14] are largely insignificant here compared to the cost of accessing

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

A. Caplinskas and J. Eder (Eds.): ADBIS 2001, LNCS 2151, pp. 308–322, 2001.
© Springer-Verlag Berlin Heidelberg 2001

data in external sources. While new cost models have been developed for use in mediator frameworks with centralized architecture [18], no experimental results are reported using a distributed mediator framework. In this work we quantify empirically the relations among the different costs in a wrapper-mediator environment, as for example, the network cost and the data source access costs.

Traditional data integration systems [11,16] send all data to the mediator for joining. Such 'ship-in' methods do not allow for integration of 'non-database' data sources that require some input, since it is not possible to ship the programming logic from these systems into the mediator. Also they are not good for top-N queries where only a first few tuples are retrieved.

Three join algorithms for a distributed mediation environment are presented and analyzed. An outer collection, generated as an intermediate result of a previous computation, is joined with an inner collection produced from a data source. Two *ship-out* algorithms ship data *toward* the sources. In these algorithms, intermediate result tuples are shipped to the sources where they are used as parameters to precompiled query fragments (subqueries or function calls) of the original query. The first algorithm is an order-preserving semi-join which is suitable when there are no duplicates in the outer collection. The second algorithm uses a temporary hash index of possibly limited size to reduce the number of accesses to the data sources. It is suitable when there are duplicates in the outer collection. Both ship-out algorithms are streamed [6] and the data is shipped between the mediator servers in bulks that contain several tuples to avoid the message set-up overhead. Finally, for comparison, a ship-in algorithm is analyzed, which is suitable when the sources cannot accept parameterized queries and when the data retrieved from the sources is small enough to be stored in a temporary main-memory index in the mediator.

The algorithms are evaluated in an environment with an ODBC data source and a mediator server running on Windows NT platforms, connected by ISDN and LAN. Substantial performance gains were measured (up to factor 100) when using our framework over an ISDN connection to access a relational database server, as compared to accessing the relational database with ODBC directly from the client, since bulk oriented join processing between the mediators minimizes ISDN message traffic and eliminates all expensive remote ODBC calls.

## 2 Background

As a platform for the work in this paper we use the Amos II mediator database system [8,9,17]. The core of Amos II is an open light-weight and extensible DBMS. It is a distributed mediator system where both the mediators and wrappers are fully functional Amos II servers, communicating over the Internet. For good performance, and since most the data reside in the data sources, each Amos II server is designed as a main-memory DBMS.

Some of the Amos II servers can be configured to wrap different kinds of data sources, e.g. ODBC compliant relational databases [4] or XML files [12]. Other servers reconcile conflicts and overlaps between similar real-world entities

modeled differently in different data sources, using the *mediation primitives* [8, 9,17] of the query language AmosQL .

Users and applications can pose OO queries to any Amos II server. We call the server(s) to which application queries are posed *client mediator(s)* for those queries. The other Amos II servers involved in answering a query are called *mediator servers.* The mediator servers may run on separate workstations and provide data integration, wrapping, and abstraction services through which different views are presented in different mediators. For example, in a mobile environment a portable computer could have a client mediator that integrates data represented by several mediator servers on a company LAN. A mediator server can have different types of data sources attached and access a number of other mediator servers.

The AmosQL query below contains a join and selection over the table $A$ at the source $DB1$, and $B$ at $DB2$, based on values of functions $fa$ and $fb$:

```
select res(b)
from A@DB1 a, B@DB2 b
where fa(a) = fb(b);
```

The query is issued in a client mediator over data that can be either directly stored in *DB1* and *DB2* or, if these are Amos II servers, retrieved from wrapped data sources. Strategies to execute this equi-join will be the focus of this paper.

The queries are rewritten by the optimizer to eliminate redundant computations. After the rewrites, queries operating over data outside the mediator are decomposed into distributed *query fragments*, executed in different Amos II servers and data sources. The decomposition uses heuristic and dynamic programming strategies in three stages [10]: query fragment generation, fragment placement and fragment scheduling. Each Amos II server uses a single-site *cost-based optimizer* to generate optimized execution plans for the query fragments. The fragments for other types of data sources are handled by the mediator if the source has no query processing capabilities, or by the source otherwise.

## 3 Algorithm Descriptions

While a naive data source interface provides only *execute* functionality for queries, Amos II also provides *bulked ship-out and execute* functionality where a remote Amos II server accepts and store tuples locally in main-memory, and then executes a query fragment using them as an input. When joining directly to a data source, the communication is directly with it and the processing is one tuple at the time for the ship-out algorithms, assuming that storing bulks of the intermediate results is not possible in data sources because of their autonomy.

### 3.1 Ship-Out Join Algorithms

In general, the ship-out algorithms can be described with the following steps:

1. preprocess and prepare the input collection for shipping
2. ship the input collection to a remote site
3. execute the query fragment over the collection at the remote site
4. return result of query fragment execution to the coordinating mediator
5. assemble the result collection to be emitted from the join

Steps 1, 4 and 5 are executed locally, while 2 and 3 are performed at another Amos II server by its join request handler.

The input collection is a table where some columns are used as parameters to the remote query fragment; other columns are passed through to the later post-processing in the mediator, or are assembled as parts of the query result.

A straight-forward implementation of a ship-out equi-join operator would ship the whole input bulk to the remote site, execute the remote query fragment on the bulk appending its result to the input, and then ship this result back. The first improvement of the naive strategy we propose is the *project-concat algorithm* (PCA) in Fig. 1[1]. It improves the naive strategy by the following two data transformations based on the semi-join algorithm [2]:

- The input bulk is projected over the data columns that are actually used in the remote query fragment, before shipping them there.
- After the query fragment is executed the result shipped back to the mediator contains only the relevant columns from the query fragment result.

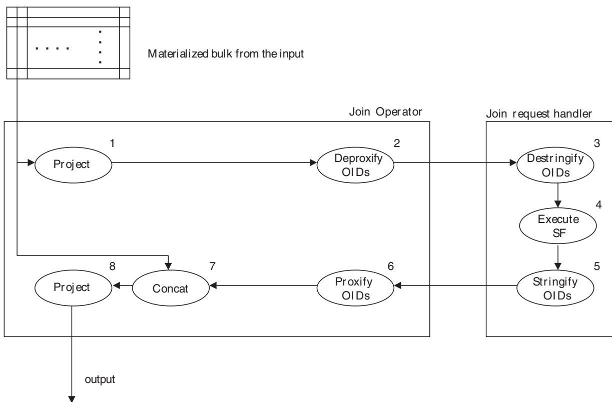
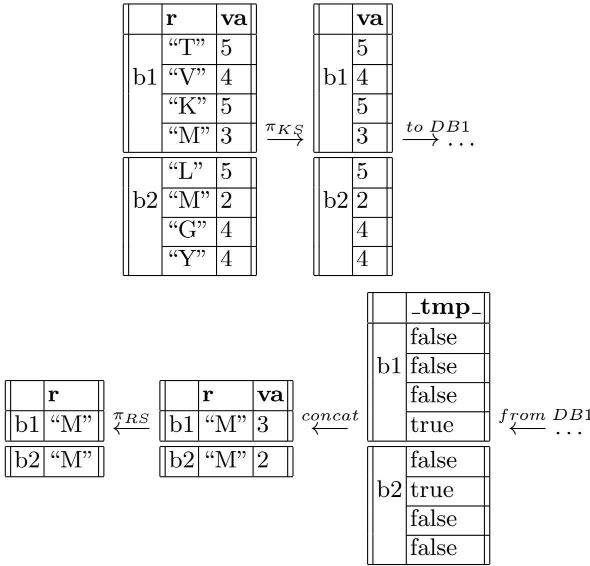



**Fig. 1.** Project-concat ship-out algorithm

[1] Amos II is object-oriented and steps 2, 3, 5, and 6 handle object identifier (OID) conversions, which are not further elaborated here.

**Table 1.** Example execution of equi-join using the project-concat algorithm



| | r | va |
|---|---|---|
| b1 | "T" | 5 |
| | "V" | 4 |
| | "K" | 5 |
| | "M" | 3 |
| b2 | "L" | 5 |
| | "M" | 2 |
| | "G" | 4 |
| | "Y" | 4 |

$\xrightarrow{\pi_{KS}}$

| | va |
|---|---|
| b1 | 5 |
| | 4 |
| | 5 |
| | 3 |
| b2 | 5 |
| | 2 |
| | 4 |
| | 4 |

$\xrightarrow{to\ DB1} \ldots$

| | _tmp_ |
|---|---|
| b1 | false |
| | false |
| | false |
| | true |
| b2 | false |
| | true |
| | false |
| | false |

$\xleftarrow{from\ DB1} \ldots$

$\xleftarrow{concat}$

| | r | va |
|---|---|---|
| b1 | "M" | 3 |
| b2 | "M" | 2 |

$\xleftarrow{\pi_{RS}}$

| | r |
|---|---|
| b1 | "M" |
| b2 | "M" |

The difference between PCA and the classical semi-join is in the use of order for matching the tuples from the joined collections.

The result of the join is assembled by a simple concatenation of the input and the result shipped back from the remote Amos II mediator or data source. Since the operations are order preserving, concatenation can be used instead of a more expensive join.

Table 1 illustrates an execution of PCA between the results of query fragments $QF1$ executed at $DB1$ and $QF2$ executed at $DB2$. The input is a collection of tuples with columns $va$ and $r$ produced by the execution of the fragment $QF2$, and a collection of tuples produced by the execution of $QF1$ containing $va$ values and keys of table $tB$. The fragments are joined over $va$ and the result is represented by column $r$. Since there are no result columns that are shipped back from DB1 to DB2, a boolean value is used to identify if the tuples produced by $QF1$ have a matching $va$ value in the tuples produced by $QF2$. We assume that the fragment at $DB1$ produces the following table:

| *va* | |
|---|---|
| **tB** | **va** |
| $ib_1$ | 4 |
| $ib_2$ | 5 |
| $ib_1$ | 6 |

where $ib_k$ denotes a key of $tB$. The example illustrates the execution over 2 bulks of size 4, named in the example as $b1$ and $b2$. In the example, first the projection strips the $r$ values from the input bulks since they are not used in

the join. Next, the bulks are shipped to $DB1$ where the query fragment $QF1$ is executed. The resulting set of boolean values is shipped back to the mediator. The concatenation shown in the example is a special case where the executed function does not return any data used later in the query processing. In this case, the concatenation of the returned boolean values and the input tuples actually filters the tuples for which the result is *true*. The final projection removes the *va* values to form the requested result.

The PCA has the advantage of improving the naive implementation, while preserving the simplicity of the processing. All operations have constant complexity per data item and therefore cheap to perform. Nevertheless, it is inefficient when there is a large percentage of duplicates in the input bulk(s), an expensive query fragment, and/or expensive communication between the servers involved.

The traditional *semi-join* algorithm (SJA) [2] improves the performance of the PCA when duplicates are involved. After projecting the input bulk over the columns used as input to the remote query fragment, SJA performs duplicate removal before shipping the data. When there is a large percentage of duplicates within the bulks, this reduces both the size of the shipped data and the number of executions of the remote query fragment. The result of the query fragment execution is shipped back to the calling server where, as in the previous algorithm, the shipped tuples are concatenated to the result of the query fragment invocation. Next, an equi-join is performed over the input bulk and the result of the concatenation. Here, because of the duplicate removal it is not possible to match the tuples by their rank in the bulk.

The SJA benefits from avoiding shipping duplicate entries over the network and executing the query fragment for them, but only for duplicates within a single bulk and with the added costs of the two additional phases of duplicate removal and equi-join.

To avoid duplicates over different bulks, the algorithm in Fig. 2, SJMA (semi-join with materialized index algorithm) extends SJA by saving the index built up for the bulks of the outer collection between executions for different bulks. The shipped data is passed through an additional anti-join over the set already pruned from duplicates and the temporary index. If a tuple is in the index, it has already been processed in some of the previous bulks. The remaining tuples are shipped to the remote site for query fragment execution as before. Next, new entries are added to the index from the returned result. Finally, a join between the input bulk and the index is performed as in the SJA. A comparative execution of SJMA in the same scenario as for the PCA example is presented in Table 2. Here, the second bulk is reduced to one tuple before shipping to $DB1$, since the anti-join eliminates the two tuples present in the first bulk.

The size of the index in SJMA is proportional to the number of distinct tuples in the outer collection. The algorithm can be used as a filter even in the case when the whole index is too big to fit in the memory. When the memory limit is reached, new entries replace old entries using some replacement criteria.

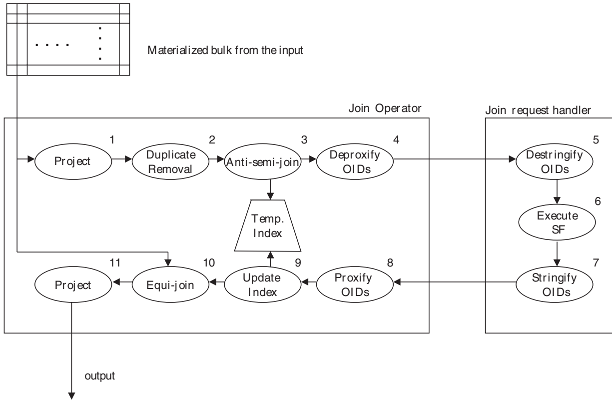
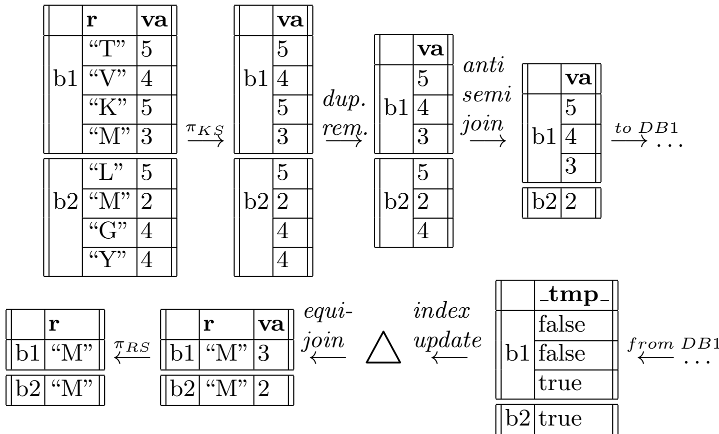



**Fig. 2.** Streamed semi-join with a temporary index

**Table 2.** Example execution of the semi-join with materialized index algorithm



| | r | va |
|---|---|---|
| b1 | “T” | 5 |
| | “V” | 4 |
| | “K” | 5 |
| | “M” | 3 |
| b2 | “L” | 5 |
| | “M” | 2 |
| | “G” | 4 |
| | “Y” | 4 |

$\xrightarrow{\pi_{KS}}$

| | va |
|---|---|
| b1 | 5 |
| | 4 |
| | 5 |
| | 3 |
| b2 | 5 |
| | 2 |
| | 4 |
| | 4 |

*dup. rem.* $\longrightarrow$

| | va |
|---|---|
| b1 | 5 |
| | 4 |
| | 3 |
| b2 | 5 |
| | 2 |
| | 4 |

*anti semi join* $\longrightarrow$

| | va |
|---|---|
| b1 | 5 |
| | 4 |
| | 3 |
| b2 | 2 |

$\xrightarrow{to\ DB1} \ldots$

$\xleftarrow{from\ DB1} \ldots$

| | _tmp_ |
|---|---|
| b1 | false |
| | false |
| | true |
| b2 | true |

*index update* $\longleftarrow$ $\triangle$ *equi-join* $\longleftarrow$

| | r | va |
|---|---|---|
| b1 | “M” | 3 |
| b2 | “M” | 2 |

$\xleftarrow{\pi_{RS}}$

| | r |
|---|---|
| b1 | “M” |
| b2 | “M” |

SJMA does not add substantially to the cost of the SJA, while it offers the possibility for performance improvements. In fact, it reduces to the SJA in the case when the whole input is contained in only one bulk.

### 3.2 Ship-In Join Method

Unlike the previous two algorithms where the remote query fragment is executed using parameters from the tuples of the intermediate result, with the ship-in join method no intermediate result is shipped to the remote site. Consequently, the query fragment is executed without parameters. This has two effects:

- Since the remote query fragment is executed once only, it may reduce the number of accesses to the data source.

– The result size may increase since instead of a semi-join of the query fragment result and the intermediate result, the whole query fragment result is sent to the client to be joined there.

While the reduction of the data source accesses may improve the performance, the increased volume of the data shipped and stored in the mediator are the possible performance disadvantages of this algorithm. The algorithm is inapplicable when the query fragment result is too big for the mediator resources. This is also the case when the query fragment contains predicates representing methods/programs in the data source that require parameters to be supplied from the mediator. With the ship-out method, when there are sufficient resources, the materialized index can persist between the execution of the algorithm for different bulks, reducing further the query processing time. This case corresponds to hash join algorithms where an index is built for the inner relation.

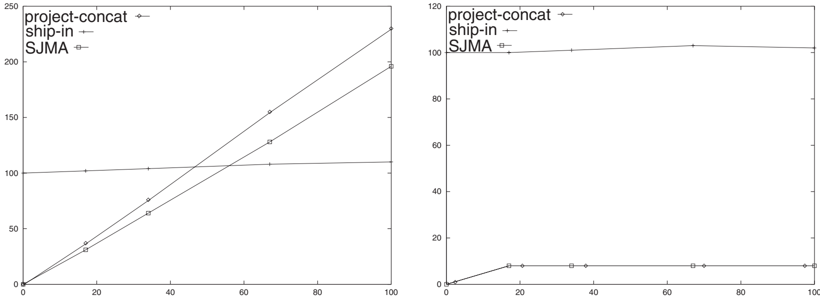
## 4 Performance Measurements

In the two scenarios used in the experiments the data source was an ODBC data source. We performed experiments using both Microsoft Access ODBC and IBM DB2 ODBC drivers with no significant differences in conclusions. Where not specifically indicated, the measurements use the Access ODBC driver.

In the first scenario, we deployed an Amos II server at the same workstation as the source. This server wrapped the source and exported it to the client mediator running on another Windows NT workstation. We present test results using this scenario and two different network connection speeds between the workstations: a 115Kb ISDN connection over the public telephone network in Sweden; and a 100Mb departmental LAN. We also varied the speed of the workstation that hosted the client mediator. In one experiment we used a 233 MHz, 32Mb RAM PC, and in the other a 600 MHz, 256Mb RAM PC. In the second scenario the data source was accessed directly from the client mediator through the ISDN network connection using DB2's ODBC interface. In this case the joins are executed one tuple at a time. We also compared the effects of different bulks sizes on the query execution time.

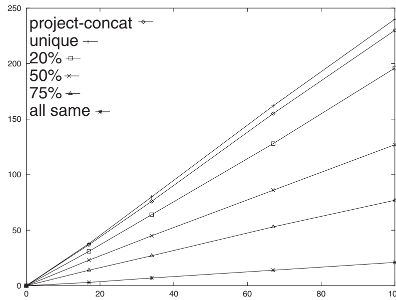
The inner collection is obtained from a table stored in the ODBC data source. The table consisted of three columns: an integer primary key $ID$, and two textual columns $A$ and $B$ of fixed length strings with sizes 10 and 250. The outer collection was stored in the client mediator, where it simulated an intermediate result. During the execution, the outer collection is bulked and streamed into the join algorithm one bulk at the time. Both the outer and inner collection had the same attributes.

Figure 3 shows the results of the execution of the three join algorithms from the previous section using a 233 MHz Windows NT workstation as a client and an ISDN connection to the server computer. The X axes in the graphs show the sizes of the outer collection in percentage of the size of the inner that always contains 30000 tuples; the Y axes marks query execution times in seconds. The outer collection is scaled from 17% to same the size as the inner. In these experiments




(a) Whole result

(b) First 1024 tuples




(c) SJMA with different percentage duplicates

**Fig. 3.** Execution times when varying the outer collection size, ISDN, 233 MHz PC

the outer collection contained 20% duplicates. Each tuple of the outer matches exactly one tuple of the inner. The graph on the left compares the execution times for a complete evaluation of the join operation. The graph in the middle compares the times to emit the first 1024 tuples. This coincides with he bulk size used to execute the query. The graph on the right compares the SJMA with PC for different percentages of duplicates in join columns of the collections.

We first analyze the execution times for the complete join operation. Since the inner collection has constant size, the time spend in the Amos II server of the inner and the network time are constant for the execution of the ship-in algorithm. The only increase of execution time is noted in the client: from 8 seconds for a 5000 tuple outer collection, to 16 seconds for a 30000 tuple outer collection. This is due to the increase of the number of index searches. Nevertheless, this increase in negligible in comparison to the total query execution time.

**Table 3.** Query execution time distribution, ISDN, 233 MHz PC

| | Time distribution | | | |
|---|---|---|---|---|
| | Client | Server | Source Access | Net. |
| **Ship-in** | 10% | 1% | 4% | 85% |
| **Ship-out, PC** | 5% | 3% | 43% | 49% |
| **Ship-out, SJMA** | 7% | 3% | 42% | 48% |

The ship-out algorithms show performance that is linear to the size of the inner collection, outperforming the ship-in algorithm until the outer is about 50% of the inner. SJMA performs better than the PC algorithm. Figure 3c compares the algorithms for different percentages of duplicates. The PC algorithm performs exactly the same, regardless of the data distribution. SJMA improves as the number of duplicates of the join columns increases. Note that even without duplicates, the performance difference of these two algorithms is small. This shows that in main-memory based mediator systems, the penalty of the additional steps of the SJMA is low.

Table 3 shows the portions of the time spent in the individual system components. The data source access time includes the time spent in the ODBC interface and the data source. The main portion of the execution of the ship-in algorithm executed over ISDN is spent on shipping the inner to the client side, which was consistently around 85% of the query execution time. We can also note that, due to the main-memory architecture of Amos II, the index build time in the client is relatively small, around 5% of the whole execution time. The first tuple is not emitted until the index for the inner is finished, which is after 95% of the processing time. This makes this algorithm unsuitable for top-N queries.

The ship-out algorithms spend less time on the network, but more in accessing the data source. They also emit the first tuple much faster than the ship-in algorithm (Fig. 3b). The experiments show here the time to emit the first 1024 tuples. When the bulking factor is less than 10, the first tuple is emitted after less than a millisecond. Furthermore, the bulking factor also determines the smoothness of the flow of the results. Smaller bulking factor will allow smoother flow of the results to the application.

Table 4 compares the effect of the distributed Amos II architecture for the ship-out algorithms. First we used SJMA to access a remote IBM DB2 data

**Table 4.** Direct access to an ODBC source and through Amos II servers

| | outer/inner | | | |
|---|---|---|---|---|
| **Inner size/outer size** | 17% | 33% | 66% | 100% |
| **through Amos II, all tuples** | 58 | 115 | 245 | 358 |
| **ODBC direct, all tuples** | 2769 | 5059 | 8552 | 12799 |
| **through Amos II, B=1024, first tuple** | 14 | 15 | 15 | 15 |
| **through Amos II, B=1, first tuple** | 0.7 | 0.72 | 0.68 | 0.71 |
| **ODBC direct, first tuple** | 1.1 | 0.9 | 1.2 | 1.04 |

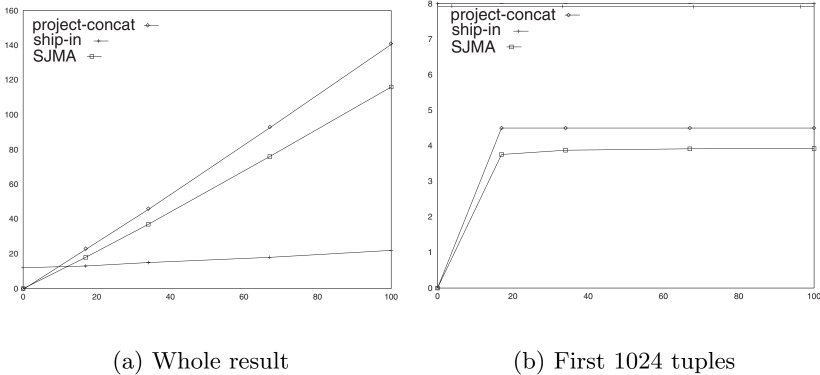
**Table 5.** Query execution time distribution, 100Mb LAN, 233MHz PC

| | Time distribution | | | |
|---|---|---|---|---|
| | Client | Server | Data source acc. | Net. |
| **Ship-in** | 67% | 7% | 22% | 4% |
| **Ship-out, PC** | 8% | 5% | 86% | 1% |
| **Ship-out, SJMA** | 12% | 5% | 82% | 1% |

source using DB2's ODBC interface over an ISDN connection. Due to the autonomy of the data sources we assume that it is not feasible to materialize intermediate results in the sources. Even if this was possible, due to the disk based nature of the DBMS, we could not expect a comparable execution time as with the main-memory storage used in Amos II. Therefore the join must be performed one tuple at a time over the remote ODBC. However, when the source is accessed through an Amos II server located on the same computer as the source, the join between the client and server mediators is executed in a bulked manner, using only the local ODBC connection between the server mediator and the source, leading to performance improvements of orders of magnitude.

The time to emit the first tuple when the bulking factor is 1024 is notably greater when the processing is done through an Amos II server. This actually represents how long it takes to emit the first 1024 tuples. If fewer tuples are required, a smaller bulking factor leads to better performance for the top-N queries when an intermediate Amos II server is used. Even when the bulking factor is 1 we can note that the use of an intermediate Amos II yields better performance than accessing the source directly, due to communication protocol differences. To achieve the best performance, the bulking factor should match the number of tuples required immediately.

Figure 4 and Table 5 illustrates join execution time on the same client computer connected with a 100Mb fast LAN to the data source. We can note that the curves have similar shapes, while the scale is different. The network cost is eliminated for almost all of the algorithms. In this executions most of the time is spend in the data source (parameterized and unparameterized query execution) and in the client for the ship-in algorithm (index build-up and join). We can also note that when the whole join result is required the ship-in algorithm outperforms the ship-out in almost all the cases. When the first-N tuples are required, however, the ship-out algorithms are more efficient. For the first 1024 result tuples the difference is about 50%. If the number of requested result tuples is smaller, the difference can be a couple of orders of magnitude. We have also varied the client computer from a workstation to a notebook. We noted that the return time for the first tuple is almost constant for the ship-out algorithms regardless of the power of the client computer. This can be explained by the fact that in the case of ship-out algorithms, the server uses the larger share of the workload than with the ship-in algorithms.




(a) Whole result (b) First 1024 tuples

**Fig. 4.** Join execution times for different outer collection sizes in percentage of the inner size, 100Mb LAN, client 233MHz PC

## 5 Related Work

The System R* project [14] is one of the first distributed database prototypes. In System R*, both ship-in and ship-out strategies are examined. In [15] a disk-based ship-in strategy (named ship-whole) is implemented with a disk based b-tree index. This type of implementation leads to considerably different results where the ship-out method always outperforms ship-in.

Disk-based semi-join algorithms are described in [1,2,5,14]. A sort-merge join, bloom filter semi-join, and sort-based semi-join are evaluated in [15] for a distributed database environment. A bloom filter phase can be added to the ship-out algorithms described in this paper. Nevertheless, this would incur additional query processing overhead and possibly shipping of some extra tuples of the inner collection. Bloom filter strategies cannot be used with sources that cannot enumerate the extent of the inner collection.

Most of the mediator frameworks reported in the literature (e.g. [7,16,19]) propose centralized query compilation and execution coordination. In [3] it is indicated that a distributed mediation framework is a promising research direction, but to the extent of our knowledge the results in this area are sketchy without experimental support. The protocols for execution of joins between data in different sources are in most cases based on retrieving the data from the sources and assembling the results in the mediator [16,19]. In the DIOM project [13], a distributed mediator system is presented where the query execution is performed in two phases: subquery execution and result assembly. The dataflow is only from the sources to the mediator.

The Garlic mediator system [7] is the only mediator system known to us that supports ship-out join strategies. The *bind join* in Garlic sends parameters to the sources as single tuples of values. In Amos II the data sources are also accessed one tuple at the time, but the distributed architecture allows for using bulked protocols over high latency lines between Amos II servers to avoid most

of the processing cost. A Garlic wrapper that has two components, one local and one remote, could achieve the benefits of the approach described in the paper. Finally, join methods where bulk shipping is combined with hashing are not applied in Garlic.

## 6 Summary and Conclusions

An efficient data integration system needs to be able to adapt to different environments by using different algorithms. The algorithms presented in this paper allow for balancing the workload between the client and the server, and for different network use patterns that give wide range of options over different hardware platforms.

The experimental results showed that for a complete query answer the ship-in algorithms generally outperform the ship-out algorithms over fast networks. Over slow networks and with very slow sources, the ship-out algorithms can give orders of magnitude better performance than ship-in since ODBC over TCP/IP calls are executed one tuple at a time while bulks of tuples are shipped between the distributed mediators. For top-N queries where N is considerably smaller than the result size, the ship-out algorithms with bulking factor N give the best performance over all the range of hardware and network connections used in the experiments. These outperform the ship-in algorithms by a few orders of magnitude. Although the bulking factor greater than 1 provides benefits, too large bulk sizes lead to reduced query execution efficiency.

In our environment, where the index operations are main-memory based and relatively cheap, the penalty of SJMA (the Semi-Join with Materialized index Algorithm) is small and it always performs nearly as well, or better than PCA (the Project-Concat Algorithm). Nevertheless, PCA uses less memory and could be much more efficient in memory-limited mediators. A compromise between these two algorithm is the SJMA with a limited size temporary index that degenerates to a SJA when the temporary index size is 0. Finally, if simplicity of implementation is considered the PCA is the algorithm of choice.

Placing an mediator server close to the source allows for bulked execution of the protocols that might change the query execution time by orders of magnitude, especially in networks with high latency. In cases when the sources lack filtering capability, the mediator server can also locally filter the query fragment result and reduce the communication cost even more.

A topic of our current work is a strategy to dynamically select between the proposed algorithms during run-time. Statistics collected during the execution can be used to determine if the default choice was the best one. Another open issue is a method to determine the optimal bulking factor in a multi join query, by taking in account the tuple sizes, join selectivities and the buffer pool size.

## References


1. P. Apers, A. Hevner, and S. Yao: Optimization Algorithms for Distributed Queries. *IEEE Transactions on Software Engineering*, 9(1), 57-68, 1983
2. P. Bernstein and D. Chiu: Using Semi-joins to Solve Relational Queries. *Journal of ACM* 28(1), 25-40, 1981
3. W. Du and M. Shan: Query Processing in Pegasus, In O. Bukhres and A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems.* Pretince Hall, 449-471, 1996.
4. G. Fahl and T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, Springer, 6(4), 261-281, 1997.
5. P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie Jr.: Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4), 602-625, 1981
6. G. Graefe and W. J. MCKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. *12th Data Engineering Conf. (ICDE'93)*, 209-218, 1993.
7. L. Haas, D. Kossmann, E.L. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *23th Intl. Conf. on Very Large Databases (VLDB'97)*, 276-285, 1997
8. V.Josifovski and T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration. *Intelligent Information Systems (JIIS)* 12(2-3), Kluwer, 165-190, 1999.
9. V.Josifovski and T.Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. *25th Intl. Conf. on Very Large Databases (VLDB'99)*, 435-446, 1999.
10. V. Josifovski and T. Risch: Query Decomposition for a Distributed Object-Oriented Mediator System. To appear in *J. of Distribued and Parallel Databases*, Kluwer, 2001.
11. E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, and M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.
12. H. Lin, T. Risch and T. Katchanounov: Adaptive data mediation over XML data. To appear in *J. of Applied System Studies (JASS)*, Cambridge International Science Publishing, 2001.
13. L. Liu and Calton Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Journal of Distributed and Parallel Databases* 5(2), 167-205, Kluwer Academic Pulishers, The Netherlands, 1997.
14. G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger and P. Wilms: Query Processing in System R*. In W. Kim, D. Reiner, D. Batory (eds.): *Query Processing in Database Systems*, Springer-Verlag, 1985.
15. L. Mackert and G. Lohman: R* Optimizer Validation and Performance Evaluation for Distributed Queries. In M. Stonebraker (ed.): *Readings in Database Systems*, Morgan-Kaufmann, CA, 1988
16. F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac: Dynamic Query Optimization in Multidatabases. *IEEE Data Engineering Bulletin*, 20(3), 38-45, 1997.
17. T. Risch and V. Josifovski: Distributed Data Integration by Object-Oriented Mediator Servers. To appear in *Concurrency - Practice and Experience J.*, John Wiley & Sons, 2001.

18. M. Roth, F. Ozcan and L. Haas: Cost Models DO MAtter: Providing Cost Information for Diverse Data Sources in Fededrated System. *25th Intl. Conf. on Very Large Databases (VLDB99)*, 599-610, 1999.
19. A. Tomasic, L. Raschid and P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions in Knowledge and Data Engineering*, 10(5), 808-823, 1998