98-PC

IEEE International Performance, Computing, and Communications Conference

Conference Proceedings
IPCCC 1998

Tempe/Phoenix, Arizona, U.S.A.
February 16 – 18, 1998

Institute of Electrical and Electronic Engineers

IEEE Communications Society

TABLE OF CONTENTS

*Paper was not available at time of printing

PERFORMANCE COMPARISON OF THREE ALTERNATIVES OF DISTRIBUTED MULTIDATABASE SYSTEMS: A GLOBAL QUERY PERSPECTIVE

Chung-Min Chen, Wei Sun and Naphtali Rishe

High Performance Database Research Center
School of Computer Science
Florida International University
Miami, FL 33199
E-mail: chungmin@cs.fiu.edu

ABSTRACT

Diversity and evolution in database applications often result in a multidatabase environment in which corporate data are stored in multiple, distributed data sources, each managed by an independent database management system. One of the essential functions of a multidatabase system is to provide inter-database access: the capability of evaluating global queries that require access to multiple data sources. This paper compares three common relational multidatabase approaches: the *federated* approach, the *gateway* approach, and the *middleware* approach from the perspective of global query performance. In particular, we examine their architectural impact on the applicability of pipelined query processing techniques and load balancing. We present a performance comparison based on a detailed simulation. The study suggests that the middleware approach, which is the most cost-effective solution among the three, provides better or comparable performance to the other two approaches.

I. INTRODUCTION

A *multidatabase system* is a collection of interconnected database systems (or data sources). Each data source is governed by an independent database management system (DBMS). Most of the time these DBMSs operate autonomously, but from time to time they need to cooperate with each other on answering *global queries* – queries that require access to more than one data sources. Multidatabase systems are common in today's enterprise-level information systems, often a result of certain application requirements or other strategic considerations [7]. Multidatabase systems are often configured in a distributed, client-server computing architecture: each componenet DBMS is installed and operates on a dedicated *server* machine, whereas database applications are run on *client* machines. All servers and clients are interconnected through a communication network. The servers retrieve data from the databases in response to client requests.

There are three common alternatives to establish a multidatabase system: the "federated" approach, the "gateway" approach, and the "middleware" approach [1]. Both the federated and gateway approaches use an inter-database-access-enabling DBMS to handle global queries. The only difference is that the federated approach adds a dedicated DBMS, whereas the gateway approach extends one of the existing servers to include such capability. The middleware, in contrast, is merely a software that coordinates the DBMS servers for collaborative activities. A middleware is *not* a fully loaded DBMS and must rely on the component DBMS servers to evaluate global queries. Middleware products are appealing to many mulitdatabase users as they are less expensive and more portable than the other two alternatives. However, there has been some doubt about the use of middleware due to a performance concern: the lack of an internal DBMS for efficient global query processing.

This paper examines the impact of the architectural differences among the three aforementioned multidatabase system approaches on the performance of global queries. We investigated this issue by analyzing the implications on applicability of pipeline, communication overhead, disk IO overhead, and load balancing. In particular, we seek answers to the following questions: (1) Is the performance of the middleware

This research was supported in part by NASA (under grants NAGW-4080, NAG5-5095, and NRA-97-MTPE-05), NSF (CDA-9711582, IRI-9409661, and HRD-9707076), ARO (DAAH04-96-1-0049 and DAAH04-96-1-0278), DoI (CA-5280-4-9044), NATO (HTECH.LG 931449), and State of Florida.

[1]The terms "federated", "gateway", and "middleware" have been used broadly, sometimes ambiguously, in the literature to mean the kinds of software that enable inter-database access. Our use of these terms in this article does not necessarily reflect exactly what is interpreted by others.

approach comparable to that of the gateway approach ? (2) Does the federated approach, at the cost of an additional server, achieve a performance gain (with respective to the middleware and gateway approaches) in proportion to the cost ?

There has been much research work on heterogeneous or multidatabase systems that focused on such issues as architectural design, schema integration, query optimization, and transaction processing (e.g., [6], [11], [1] and [10]). None of them, to the best of our knowledge, has addressed the comparative performance issue related to the three models described above. Recent examples of federated multidatabase systems include DATAPLEX [4] and DB Integrator [9]. Many relational DBMS vendors have also supported their own DBMS products with gateway access to other DBMSs (e.g., ORACLE's Open Gateways and Sybase's OmniConnect). Some early research prototypes, for example, Mermaid [13] and MDAS [5], can be considered middleware systems. Recently, middleware products from third-party vendors have been emerging in the market (e.g., Intersolv's DataDirect ODBC Driver and Information Builders's EDA/SQL). The most recent industry development on multidatabase architectures and APIs can be found in [8].

II. THREE MULTIDATABASE APPROACHES

A. Federated Approach

Figure 1 shows a federated multidatabase system. In this configuration, a separate federal DBMS (FDBMS) server with inter-database access capability is added to the system. The FDBMS has its own relational data storage system and query evaluation engine, and must provide, for each type of DBMS it supports, a software module that performs necessary SQL dialect translation and data format conversion. Local queries – queries that access data in a single data source – are processed at the respective DBMS server. The FDBMS server is used exclusively for processing global queries. It decomposes a global query into a number of remote sub-queries and a local *assembly sub-query*. The remote sub-queries must be expressed in SQL format and sent to the respective DBMSs for execution. The assembly sub-query, which is executed at the FDBMS server, collects the results from remote sub-queries and merges them into a final result. The outlined arrows in the figure indicate the data flow directions in the execution of a global query. Having its own relational database engine, the FDBMS may speed up the assembly sub-query by pipelining the input data streams from other servers with the local assembling operations. This avoids the need to store the input data streams explicitly on the local disk first.

B. Gateway Approach

Figure 2 shows the gateway approach for a multidatabase system. In this setting, one of the existing DBMS servers in the system is enhanced with the inter-database access capability. This DBMS, called a *gateway* DBMS (GDBMS), assumes two roles: as an autonomous DBMS server that continues to process local queries, and as a federal DBMS that handles global queries. From the view of global queries, the internal workings of the GDBMS is similar to that of a FDBMS. The difference between the two is configurational rather than architectural. Just like the federated approach, the gateway approach may also take advantage of pipelined query processing at the GDBMS server. In addition, global queries which involve tables stored in the GDBMS will not need to send these table across the network to a remote site. This is a potential performance gain over the federated approach which requires all tables to be staged at the FDBMS. The main problem with the gateway approach is that the GDBMS may become the bottleneck as it is overloaded with global and local queries.

C. Middleware Approach

A *middleware* is a software that supports global queries in a multidatabase system by relying only on the processing power of the component DBMSs. Figure 3 shows a multidatabase system using a middleware. The middleware, though drawn separately, can be run at any of the DBMS server machines. It accepts and generates, for each global query, an execution schedule that contains a number of sub-queries (to be executed at various DBMS servers) and necessary data transfers between the servers. The middleware is responsible for routing the sub-queries to the servers, translating SQL dialects as needed, converting data from one format to another, and coordinating data exchanges between the servers.

A middleware can only interact with the database servers through a high-level query interface (typically a call-level SQL). And since it does not have its own relational DBMS engine, pipelined global processing is not applicable. Rather, temporary tables must be created in some of the servers to hold intermediate results or remote tables. In comparison to the federated and gateway approaches, the need to hold data in temporary tables incurs extra IO overhead and increases the load on server disks. However, the middleware approach has a better load balancing nature since the workload of global queries is equally distributed among all DBMS

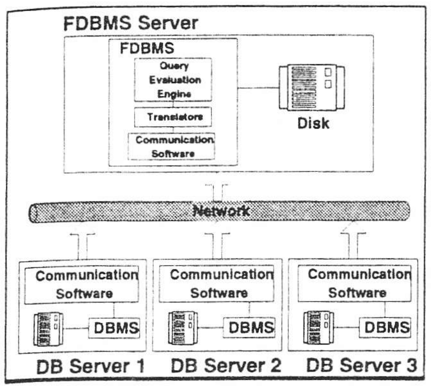

Figure 1: Federated Approach

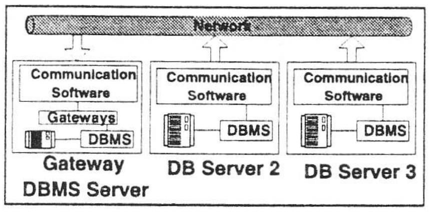

Figure 2: Gateway Approach

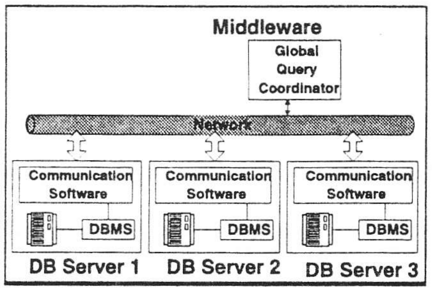

Figure 3: Middleware Approach

servers. Another merit of middleware has to do with the cost: being a "lightweight" software, a middleware is usually priced lower than the other two approaches, consumes relatively few resources, and possibly requires less administration.

III. MULTIDATABASE JOIN PROCESSING: PIPELINED VS. NON-PIPELINED

Global queries are indeed distributed *join* queries [12]. Distributed join processing in a multidatabase system differs from that in a traditional distributed database system in that the relations stored in autonomous DBMS servers are accessible only through SQL interfaces. The implication is that pipelined processing is rather limited in multidatabase systems. In the following, we describe variants of distributed join algorithms, with different degrees of pipelining, that are applicable in the three multidatabase alternatives and discuss their performance implications. We consider both hash-join and merge-join strategies [12], with focus on the former. In this study, we do not consider index-join strategies using *pre-existing* indices as the federated approach can not utilize any pre-existing index, neither do we consider nested-loop join strategy as it is much less efficient than hash-join. Throughout the discussion, we consider an equal-join query $r \bowtie_{r.A=s.A} s$ where r and s are relations located at different sites and A is the join attribute.

A. Hash-Join Algorithms

Figure 4 outlines a basic hash-join algorithm. The algorithm consists of two phases: the *partition* phase, and the *probe* phase. Without loss of generality, let s be the smaller relation. In the first phase, tuples in each of the relations are partitioned into "buckets" by applying a hash function, $h : \mathcal{D}_A \rightarrow \{1, 2, \ldots, m\}$, to the join attribute, where $\mathcal{D}_A$ is the domain of A and m, the number of buckets, is chosen so that a one-block output buffer can be allocated for each bucket. An output buffer is flushed out to the disk when it is full or when the partition is finished. In the probe phase, each pair of buckets $B_{r,i}$ and $B_{s,i}$ are compared for matching tuples. We assume that each bucket $B_{s,i}$ along with the hash index always fits in the memory.

When both relations are located at the same site (the case for a local join query), the partition phase accounts for one pass of read and one pass of write for both relations [2]; the probe phase accounts for another pass of read for both relations. We call this a ***loc-loc*** hash-join, indicating both operands are local relations. We will describe next the variants of hash-join algorithms that can be applied to global queries in each of the three multidatabase approaches. To facilitate the discussion, we call the DBMS server at which the final result is assembled the *join site*. A relation is a *local* relation if it is located at the join site, otherwise it is called a *remote* relation.

A.1. Federated Variants

In the federated approach, both operand relations of a global join are remote to the FDBMS server and thus must be accessed through the network. Since the FDBMS has its own query evaluation engine, the partition phase can be performed directly against the incoming data streams in a pipelined manner, without having to first store the data in temporary tables. In other words, the input of the remote relation from the network are overlapped with the partition task. Once the partition is done, the probe phase can proceed as usual. We call this a ***str-str*** hash-join, indicating that both join operands are in the form of data streams. The overall disk IO cost of a *str-str* hash-join is the same as

[2]Assuming all disk blocks, except the last one, allocated to the buckets are completely filled so the total size of the buckets is the same as the original size of the input relation.

```
for each $t_r$ in $r$ do    /* partition phase */
  $i = h(t_r.A)$;
  Insert $t_r$ into bucket $B_{r,i}$;
end
for each $t_s$ in $s$ do
  $i = h(t_s.A)$;
  Insert $t_s$ into bucket $B_{s,i}$;
end
for $i = 0$ to $m$ do    /* probe phase */
  Load $B_{s,i}$ and build an in-memory hash index
    for $B_{s,i}$ based on attribute $s.A$;
  Perform the join $B_{r,i} \bowtie B_{s,i}$ by probing, for each
    $t_r \in B_{r,i}$, the hash index to locate those tuples
    $t_s \in B_{s,i}$ such that $t_s.A = t_r.A$;
end
```

Figure 4: Basic Hash-Join Algorithm

that of a *loc-loc* hash-join. The former, however, incurs additional communication and CPU overhead.

A.2. Gateway Variants

There are two variants applicable in the gateway approach: the *str-str* and the ***str-loc*** variants. The former, which has been described above, applies when both operand relations are remote to the GDBMS server; the latter applies when only one relation is remote. Similar to the *str-str* variant, the *str-loc* variant may overlap the transfer of the remote relation with the partition task in a pipeline. The IO cost of a *str-loc* hash-join remains the same as that of a *loc-loc* hash-join. The *str-loc* variant will require additional communication and CPU costs over the *loc-loc* variant, but less than those incurred by the *str-str* variant (which access two remote relations).

A.3. Middleware Variants

In the middleware approach, the join site always hosts one of the operand relations. This means that for any global query, only one relation needs to be accessed through the network. However, since a middleware has no control of and can only interact with the DBMS at the join site through a high-level SQL interface, the remote relation must be sent to the join site and *fully imported* into a temporary database table before an SQL query can be issued to perform the join. We call this variant a ***imp-loc*** hash-join. Compared to the other variants, a *imp-loc* hash-join requires an additional pass of write and read for the imported relation. Though the middleware approach is unable to take advantage of pipelined processing, it may produce a better load balance since all componenet DBMS servers are potential join sites for global queries. Table 1 summarizes the

```
repeat until the end of either relation is reached
  Collect the next set of tuples $B_r$ from $r$, and the next
    set of tuples $B_s$ from $s$ such that $t_r.A = t_s.A$, for all
    $t_r \in r$ and $t_s \in s$;
  Perform a Cartesian product $B_s \times B_r$, add the output
    tuples to the result;
end
```

Figure 5: Basic Merge-Join Algorithm

	local queries	global queries
FED	*loc-loc*	*str-str*
GAT	*loc-loc*	*str-str*, *str-loc*
MID	*loc-loc*	*imp-loc*

Table 1: Applicable Join Algorithm Variants

hash-join variants that are applicable in the three multidatabase approaches (FED, GAT, and MID stands for federated, gateway, and middleware approaches, respectively). Among them, *loc-loc* is the most efficient one, followed by *str-loc*, with *str-str* and *imp-loc* trailing with a tradeoff between network and disk IO overhead.

B. Merge-Join Algorithms

When both relations are physically sorted based on the join attribute, merge-join could be more efficient than other join strategies. Figure 5 outlines the basic merge-join algorithm. The algorithm scans the relations sequentially, locating and brining into memory the next groups of tuples, B_r and B_s, from both relations that hold the same value on the join attribute. It then performs a Cartesian product between these two sets of tuples and add the composite tuples to the result. Variants of merge-join algorithm for different multidatabase approaches can be reasoned in a similar way to those described in the hash-join strategy and coincide with Table 1.

IV. SIMULATION MODEL

We have implemented a simulation package in C to evaluate the global query performance of the three multidatabase architectures based on the distributed join algorithms described in the previous section. To lay a foundation for a fair performance comparison, the simulation model assumes that (1) all server machines have equivalent processing power (same CPU speed and IO access time), and (2) all DBMSs are equally "intelligent" in the sense that, for each given query, they will select the same join strategy (hash-join vs. merge-join). Figure 6 shows a *closed queuing network* simulation model which consists of a number of DBMS server

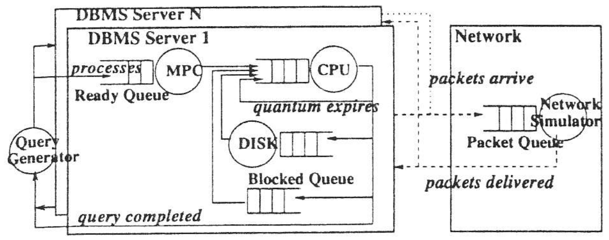

Figure 6: Simulation Model

modules and a network module.

DBMS Server Modules Each DBMS server module contains a CPU and a disk device, both associated with a process queue. Processes are scheduled for CPU based on round-robin and for the disk device based on FCFS. To avoid performance degradation caused by thrashing, the server restricts the number of concurrent processes by maintaining a *multiprogramming control queue* (MPC queue). A newly arriving process must enter the MPC queue before it can be admitted to compete for the resources.

The execution of a query is simulated in terms of processes. While a local query spawns only one process at a database server, a global join query needs to spawn processes at more than one database servers. We model a process as a state transition diagram (STD) that contains a detailed schedule of the operations to be performed. Each state specifies an operation and the queue (CPU, disk, or blocked queue) at which the operation is to be performed. The time a process spends in a queue depends on the length of the queue and the service time to complete the operation. Typically, a process would go through a number of CPU and disk IO cycles before completion. Due to space restriction, we refer the readers to [3] for more details on the STD model.

Network Module The communication between the database servers is based on a message-passing model with two *blocking* primitives: `send()` and `receive()`. Data are divided into and transmitted in *packets*. A process calling `send()` will enter a packet into the packet queue and be temporarily blocked until the packet is delivered. Similarly, a call to `receive()` will not return control until a packet is received by the calling process. The network processor and the packet queue simulate the latency caused by the network protocol stack, physical bandwidth restriction, and packet contention. In our experiments, we use the same size for packets and disk blocks. The pipelined processing in variants *str-str* and *str-loc* is performed at the granularity of data blocks.

Query Workload A random query generator is used to produce the query stream workload. The generator allows one to specify the values for such parameters as relation sizes, join selectivities, and percentages of local and global queries. All database servers prefer merge-join (if applicable) to hash-join because the former is more efficient. When a query is completed, a new query is generated immediately and enters the system. The number of concurrent queries allowed in the system, called the *concurrency level* (CL), is specified as a parameter. Since the system is arranged as a closed queuing network, by varying the concurrency level we are able to observe the scalability of the performance with respect to system loads.

System Parameter	Value
number of DBMS servers	3
concurrency level	1 - 70
server CPU MIPS	100 MIPS
avg. disk access time	10 ± 2 msec
network transfer rate	10 Mbits/sec
block/packet size	4KBytes

Operation	# of instr./data block
read	3500
write	7000
send, receive	3500
data conversion	20000
merge	10000
hash	15000
probe	18000

Query Parameter	Value
global/local query	20%/80%
merge-join/hash-join	20%/80%
relation sizes	200-250; 100-200 blocks
result size	40-150 blocks

Table 2: Default Parameter Values

V. SIMULATION RESULTS

Table 2 shows the default values of the parameters used in the experiments. We used three autonomous database servers in all experiments. In the case of the federated approach, a fourth FDBMS server is added. The CPU costs of the various operations (in terms of number of instructions executed per data block) are estimated based on the implementation of a client-server heterogeneous DBMS prototype [2]. The default query stream consists of 80% local queries and 20% global queries. Similarly, 20% of all queries are joined on sorted attribute and use merge-join strategy; the other 80% use hash-join strategy. We believe this is the norm in a multidatabase environment where most queries are local and most relations are not sorted on join attributes.

A. Effect of Query Loads

Figure 7 shows the average throughputs of the three approaches versus the concurrency level (CL). The average throughput is calculated by dividing the number of queries completed (including global and local queries) by the total elapsed time, measured in terms of number of queries per minute (QPM). For each point of observation, a sufficient number of queries (200-1000) were run so that the system reaches a stable state with a 95% confidence on the average throughput before finishing. As can been seen, the throughputs of all approaches level off when the number of concurrent queries reach to 20. GAT and MID yield close performance. When the load is mild ($10 <$ CL < 30), MID outperforms GAT because of better load balancing. However, when the load is heavy and the system becomes critically IO-bound (CL ≥ 40), GAT surpasses MID because it needs fewer IO operations for global queries (due to the applicability of pipelined processing). The FED approach, using an additional DBMS server for global queries, produces better performance than the other two in all cases. The improvement, however, is less than linear with the cost: the *throughput per server* produced by FED ($\frac{18 \text{ QPM}}{4 \text{ servers}}$) is less than those produced by GAT and MID (which are greater than $\frac{15 \text{ QPM}}{3 \text{ servers}}$).

Figure 8 shows the average response time for global queries. The response time of a query is measured as the elapsed time between the time the query is submitted and the time the query is completed. In all configurations, the response time increases almost linearly with the number of concurrent queries. The FED approach constantly yields the best response time due to the additional dedicated FDBMS server for global queries. The MID configuration, thanks to its load balancing nature, produces shorter response time than the GAT approach (which is more likely to develop a contention at the gateway server) under moderate to heavy loads (CL ≥ 15).

B. Effect of Network Bandwidth

Today's multidatabase systems are not necessarily confined to a local-area-network. The database servers could be connected through a wide-area-network such as the Internet or other types of proprietary networks that bear a lower effective data transfer rate (typically in the range of tens to hundreds of kilobits per second). On the other hand, emerging network technologies continue to improve the data transfer rate (e.g. ATM at 155 Mbps and Fast Ethernet at 100 Mbps). Figure 9 compare the throughputs (under a workload of CL = 20) over a wide spectrum of effective network data transfer rates. The middleware approach outperforms the other two when the effective network data

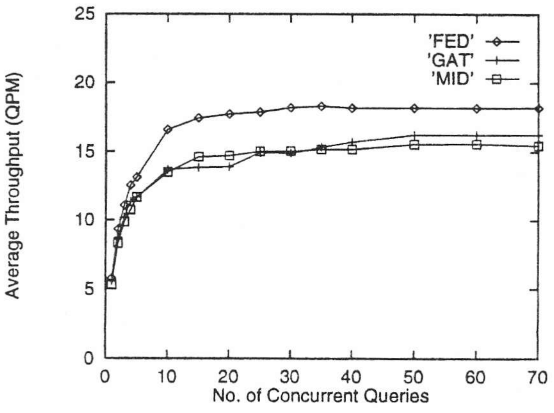

Figure 7: Comparison of Average Throughput

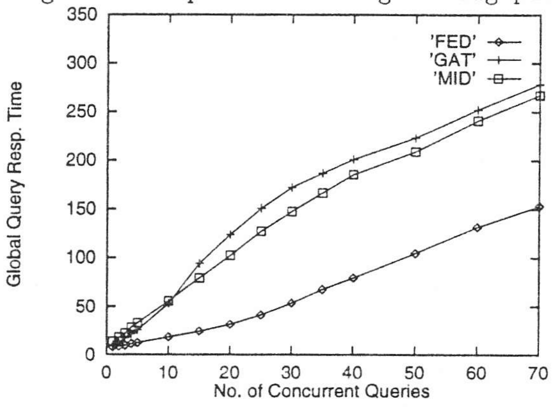

Figure 8: Comparison of Global Query Resp. Time

transfer rate is relatively low (≤ 1 Mbps). This is because MID requires less data transfer (one relation per global query) than the other two approaches (two relations for FED and $1 + \frac{N-1}{N}$ relations for GAT, where N is the number of database servers). When the effective network data transfer rate increases, the system bottleneck shifts from the network to the IO. Eventually the system becomes IO-bound and the throughputs level off. The FED performs the best under such a condition as it has an additional disk to share the loads of global queries. The performance gain, again, is less than linear scale-up to the cost.

C. Effect of Query Mix

Figure 10 shows the average throughput as a function of the percentage of global queries. The throughput of GAT declines as the frequency of global queries increases since it causes the gateway server to become the bottleneck, and leaves other servers under-utilized (by not having enough local queries to keep them busy). In the case of MID, the throughput is less sensitive to the change of global query load. This is largely attributed to MID's load balancing nature in processing global queries. In contrast, the FED approach is most

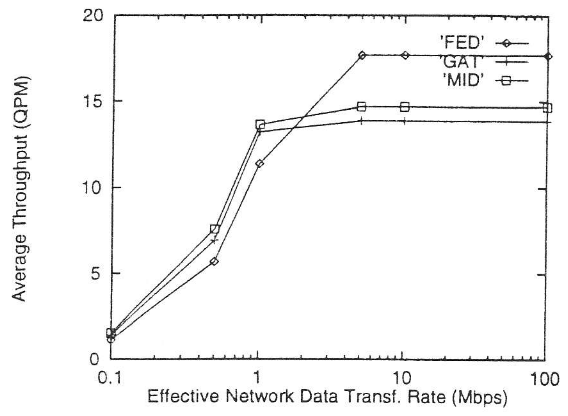

Figure 9: Effect of Network Bandwidth

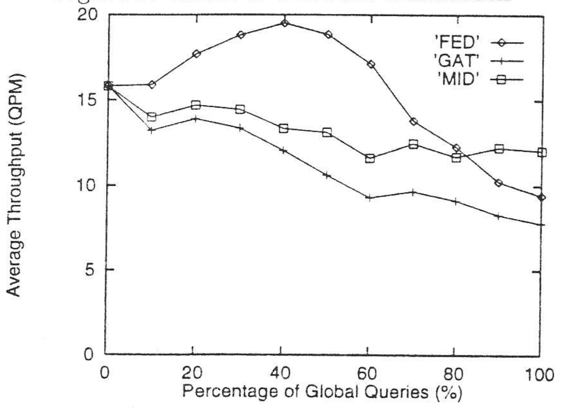

Figure 10: Effect of Global/Local Query Mix

sensitive to the global query load. The throughput increases initially as a result of better federal server utilization. It reaches a summit at which the resources of the federal database server is fully utilized. The throughput then starts to drop as a consequence of resource contention.

VI. CONCLUSIONS

In this paper we have examined the implications and compared the performance of three alternatives of distributed multidatabase systems from the perspective of global join queries. Our study has shown that, on a fair comparison ground, the middleware approach, though lacking its own query evaluation engine, produces comparable performance with the gateway approach, sometimes better. The middleware's lack of pipelined processing capability (and thus higher disk IO costs) is compensated by its better balanced servers for global query workloads and lower network overhead. The federated approach, at the expense of an additional DBMS server, is able to enhance the system throughput, but at a rate less than linear with the cost. Among the three, the middleware is the most cost-effective one if performance is not a dominant concern. In practice, the middleware approach has another advantage: it may utilize pre-existing indices in processing global queries, a technique that is not applicable in the federated approach and is limited in the gateway approach.

VII. REFERENCES

[1] Proc. of 1st Intl. Workshop on Interoperability in Multidatabase Systems. Kyoto, Japan, 1991.

[2] C.-M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer. In *Procs. of the 4th Intl. Conf. on Extending Database Technology*, 1994.

[3] C.-M. Chen, W. Sun, and N. Rishe. Evaluation of three multidatabase alternatives. Technical report, Florida International University, Miami FL, 1997.

[4] C.-W. Chung. DATAPLEX: An access to heterogeneous distributed databases. *Comm. of the ACM*, 33(1), 1990.

[5] B C Desai and R Pollock. MDAS: heterogeneous distributed database management system. *Information and Software Technology*, 1992.

[6] A. Gupta, editor. *Integration of information systems: bridging heterogeneous databases.* IEEE Press, 1989.

[7] A.R. Hurson, M.W. Bright, and S. Pakzad, editors. *Multidatabase systems : an advanced solution for global information sharing.* IEEE Press, 1994.

[8] K. North. Understanding multidatabase APIs and ODBC. DBMS and Internet Systems (*www.dbmsmag*), 1994.

[9] R. Pledereder et al. DB integrator: Open middleware for data access. *Digital Technical Journal*, 7(1), 1995.

[10] S. Ram. Heterogeneous distributed database systems. *IEEE Computer*, December 1991.

[11] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.

[12] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, 1996.

[13] Templeton et al. Mermaid: A front-end to distributed heterogeneous databases. *Proc. IEEE*, 1987.