MARIPOSA: A WIDE-AREA DISTRIBUTED DATABASE SYSTEM*

Michael Stonebraker, Paul M. Aoki, Avi Pfeffer°, Adam Sah,
Jeff Sidell, Carl Staelin† and Andrew Yu‡

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720-1776
`mariposa@postgres.Berkeley.EDU`

Abstract

The requirements of wide-area distributed database systems differ dramatically from those of LAN systems. In a WAN configuration, individual sites usually report to different system administrators, have different access and charging algorithms, install site-specific data type extensions, and have different constraints on servicing remote requests. Typical of the last point are production transaction environments, which are fully engaged during normal business hours, and cannot take on additional load. Finally, there may be many sites participating in a WAN distributed DBMS.

In this world a single program performing global query optimization using a cost-based optimizer will not work well. Cost-based optimization does not respond well to site-specific type extension, access constraints, charging algorithms, and time-of-day constraints. Furthermore, traditional cost-based distributed optimizers do not scale well to a large number of possible processing sites. Since traditional distributed DBMSs have all used cost-based optimizers, they are not appropriate in a WAN environment, and a new architecture is required.

We have proposed and implemented an economic paradigm as the solution to these issues in a new distributed DBMS called Mariposa. In this paper, we present the architecture and implementation of Mariposa and discuss early feedback on its operating characteristics.

---

* This research was sponsored by the Army Research Office under contract DAAH04-94-G-0223, the Advanced Research Projects Agency under contract DABT63-92-C-0007, the National Science Foundation under grant IRI-9107455 and Microsoft Corp.

° Author's current address: Department of Computer Science, Stanford University, Stanford, CA 94305.

† Author's current address: Hewlett-Packard Laboratories, M/S 1U-13 P.O. Box 10490, Palo Alto, CA 94303.

‡ Author's current address: Illustra Information Technologies, Inc., 1111 Broadway, Suite 2000, Oakland, CA 94607.

1. INTRODUCTION

The Mariposa distributed database system addresses a fundamental problem in the standard approach to distributed data management. We argue that the underlying assumptions traditionally made while implementing distributed data managers do not apply to today's wide-area network (WAN) environments. We present a set of guiding principles that must apply to a system designed for modern WAN environments. We then demonstrate that existing architectures cannot adhere to these principles because of the invalid assumptions just mentioned. Finally, we show how Mariposa can successfully apply the principles through its adoption of an entirely different paradigm for query and storage optimization.

Traditional distributed relational database systems that offer location-transparent query languages, such as Distributed INGRES [STON86], R* [WILL81], SIRIUS [LITW82] and SDD-1 [BERN81], all make a collection of underlying assumptions. These assumptions include:

- *Static data allocation*: In a traditional distributed DBMS, there is no mechanism whereby objects can quickly and easily change sites to reflect changing access patterns. Moving an object from one site to another is done manually by a database administrator and all secondary access paths to the data are lost in the process. Hence, object movement is a very ‘‘heavyweight’’ operation and should not be done frequently.
- *Single administrative structure*: Traditional distributed database systems have assumed a query optimizer which decomposes a query into ‘‘pieces’’ and then decides where to execute each of these pieces. As a result, site selection for query fragments is done by the optimizer. Hence, there is no mechanism in traditional systems for a site to refuse to execute a query, for example because it is overloaded or otherwise indisposed. Such ‘‘good neighbor’’ assumptions are only valid if all machines in the distributed system are controlled by the same administration.
- *Uniformity*: Traditional distributed query optimizers generally assume that all processors and network connections are the same speed. Moreover, the optimizer assumes that any join can be done at any site, e.g., all sites have ample disk space to store intermediate results. They further assume that every site has the same collection of data types, functions and operators, so that any subquery can be performed at any site.

These assumptions are often plausible in local area network (LAN) environments. In LAN worlds, environment uniformity and a single administrative structure are common. Moreover, a high speed, reasonably uniform interconnect tends to mask performance problems caused by suboptimal data allocation.

In a wide-area network environment, these assumptions are much less plausible. For example, the Sequoia 2000 project [STON91b] spans 6 sites around the state of California with a wide variety of hardware and storage capacities. Each site has its own database administrator, and the willingness of any site to perform work on behalf of users at another site varies widely. Furthermore, network connectivity is not uniform. Lastly, type extension often is available only on selected machines, because of licensing restrictions on proprietary software or because the type extension uses the unique features of a particular hardware architecture.

As a result, traditional distributed DBMSs do not work well in the non-uniform, multi-administrator WAN environments of which Sequoia 2000 is typical. We expect an explosion of configurations like Sequoia 2000 as multiple companies coordinate tasks, such as distributed manufacturing, or share data in sophisticated ways, for example through a yet-to-be-built query optimizer for the World Wide Web.

As a result, the goal of the Mariposa project is to design a WAN distributed DBMS. Specifically, we are guided by the following principles, which we assert are requirements for non-uniform, multi-administrator WAN environments:

- *Scalability to a large number of cooperating sites*: In a WAN environment, there may be a large number of sites which wish to share data. A distributed DBMS should not contain assumptions that will limit its ability to scale to 1000 sites or more.
- *Data mobility*: It should be easy and efficient to change the "home" of an object. Preferably, the object should remain available during movement.
- *No global synchronization*: Schema changes should not force a site to synchronize with all other sites. Otherwise, some operations will have exceptionally poor response time.
- *Total local autonomy*: Each site must have complete control over its own resources. This includes what objects to store and what queries to run. Query allocation cannot be done by a central, authoritarian query optimizer.
- *Easily configurable policies*: It should be easy for a local database administrator to change the behavior of a Mariposa site.

Traditional distributed DBMSs do not meet these requirements. Use of an authoritarian, centralized query optimizer does not scale well; the high cost of moving an object between sites restricts data mobility; schema changes typically require global synchronization; and centralized management designs inhibit local autonomy and flexible policy configuration.

One could claim that these are implementation issues, but we argue that traditional distributed DBMSs *cannot* meet the requirements defined above for fundamental architectural reasons. For example, any distributed DBMS must address distributed query optimization and placement of DBMS objects. However, if sites can refuse to process subqueries, then it is difficult to perform cost-based global optimization. In addition, cost-based global optimization is "brittle" in that it does not scale well to a large number of participating sites. As another example, consider the requirement that objects must be able to move freely between sites. Movement is complicated by the fact that the sending site and receiving site have total local autonomy. Hence the sender can refuse to relinquish the object, and the recipient can refuse to accept it. As a result, allocation of objects to sites cannot be done by a central database administrator.

Because of these inherent problems, the Mariposa design rejects the conventional distributed DBMS architecture in favor of one that supports a microeconomic paradigm for query and storage optimization. All distributed DBMS issues (multiple copies of objects, naming service, etc.) are reformulated in microeconomic terms. Briefly, implementation of an economic paradigm requires a number of entities and mechanisms. All Mariposa clients and servers have an account with a network bank. A user allocates a **budget** in the currency of this bank to each query. The goal of the query processing system is to solve the query within the allotted budget by contracting with various Mariposa processing sites to perform portions of the query. Each query is administered by a **broker,** which obtains bids for pieces of a query from various sites. The remainder of this section shows how use of these economic entities and mechanisms allows Mariposa to meet the requirements set out above.

The implementation of the economic infrastructure supports a large number of sites. For example, instead of using centralized metadata to determine where to run a query, the broker makes use of a distributed advertising service to find sites that might want to bid on portions of the query. Moreover, the broker is specifically designed to cope successfully with very large Mariposa networks. Similarly, a server can join a Mariposa system at any time by buying objects from other sites, advertising its services and then bidding on queries. It can leave Mariposa by selling its objects and ceasing to bid. As a result, we can achieve a highly scalable system using our economic paradigm.

Each Mariposa site makes storage decisions to buy and sell fragments, based on optimizing the revenue it expects to collect. Mariposa objects have no notion of a home, merely that of a current owner. The current owner may change rapidly as objects are moved. Object movement preserves all secondary indexes, and is coded to offer as high performance as possible. Consequently, Mariposa fosters data mobility and the free trade of objects.

Avoidance of global synchronization is simplified in many places by an economic paradigm. Replication is one such area. The details of the Mariposa replication system are contained in a separate paper [SIDE95]. In short, copy holders maintain the currency of their copies by contracting with other copy holders to deliver their updates. This contract specifies a payment stream for update information delivered within a specified time bound. Each site then runs a ‘‘zippering’’ system to merge update streams in a consistent way. As a result, copy holders serve data which is out of date by varying degrees. Query processing on these divergent copies is resolved using the bidding process. Metadata management is another, related area that benefits from economic processes. Parsing an incoming query requires Mariposa to interact with one or more **name services** to identify relevant metadata about objects referenced in a query, including their location. The copy mechanism described above is designed so that name servers are just like other servers of replicated data. The name servers contract with other Mariposa sites to receive updates to the system catalogs. As a result of this architecture, schema changes do not entail any synchronization; rather such changes are ‘‘percolated’’ to name services asynchronously.

Since each Mariposa site is free to bid on any business of interest, it has total local autonomy. Each site is expected to maximize its individual profit per unit of operating time and to bid on those queries that it feels will accomplish this goal. Of course, the net effect of this freedom is that some queries may not be solvable, either because nobody will bid on them or because the aggregate of the minimum bids exceeds what the client is willing to pay. In addition, a site can buy and sell objects at will. It can refuse to give up objects, or it may not find buyers for an object it does not want.

Finally, Mariposa provides powerful mechanisms for specifying the behavior of each site. Sites must decide which objects to buy and sell and which queries to bid on. Each site has a **bidder** and a **storage manager** that make these decisions. However, as conditions change over time, policy decisions must also change. Although the bidder and storage manager modules may be coded in any language desired, Mariposa provides a low level, very efficient embedded scripting language and **rule system** called Rush [SAH94a]. Using Rush, it is straightforward to change policy decisions; one simply modifies the rules by which these modules are implemented.

The purpose of this paper is to report on the architecture, implementation, and operation of our current prototype. Preliminary discussions of Mariposa ideas have been previously reported in [STON94a, STON94b]. At this time (June 1995), we have a complete optimization and execution system running, and we will present performance results of some initial experiments.

In the next section, we present the three major components of our economic system. Section 3 describes the bidding process by which a broker contracts for service with processing sites, the mechanisms that make the bidding process efficient, and the methods by which network utilization is integrated into the economic model. Section 4 describes Mariposa storage management. Section 5 describes naming and name service in Mariposa. Section 6 presents some initial experiments using the Mariposa prototype. Section 7 discusses previous applications of the economic model in computing. Finally, Section 8 summarizes the work completed to date and the future directions of the project.

2. ARCHITECTURE

Mariposa supports transparent fragmentation of tables across sites. That is, Mariposa clients submit queries in a dialect of SQL3; each table referenced in the `FROM` clause of a query could potentially be decomposed into a collection of table **fragments**. Fragments can obey range- or hash-based distribution criteria which logically partition the table. Alternately, fragments can be unstructured, in which case records are allocated to any convenient fragment.

Mariposa provides a variety of fragment operations. Fragments are the units of storage that are bought and sold by sites. In addition, the total number of fragments in a table can be changed dynamically, perhaps quite rapidly. The current owner of a fragment can **split** it into two storage fragments whenever it is deemed desirable. Conversely, the owner of two fragments of a table can **coalesce** them into a single fragment at any time.

To process queries on fragmented tables and support buying, selling, splitting, and coalescing fragments, Mariposa is divided into three kinds of modules as noted in Figure 1. There is a **client program** which issues queries, complete with bidding instructions, to the Mariposa system. In turn Mariposa contains a **middleware** layer and a **local execution** component. The middleware layer contains several query preparation modules, and a **query broker.** Lastly, local execution is composed of a **bidder,** a **storage manager,** and a local **execution engine.**

In addition, the broker, bidder and storage manager can be tailored at each site. We have provided a high performance rule system, Rush, in which we have coded initial Mariposa implementations of these modules. We expect site administrators to tailor the behavior of our implementations by altering the rules present at a site. Lastly, there is a low-level utility layer that implements essential Mariposa primitives for communication between sites. The various modules are shown in Figure 1. Notice that the client module can run anywhere in a Mariposa network. It communicates with a middleware process running at the same or a different site. In turn, Mariposa middleware communicates with local execution systems at various sites.

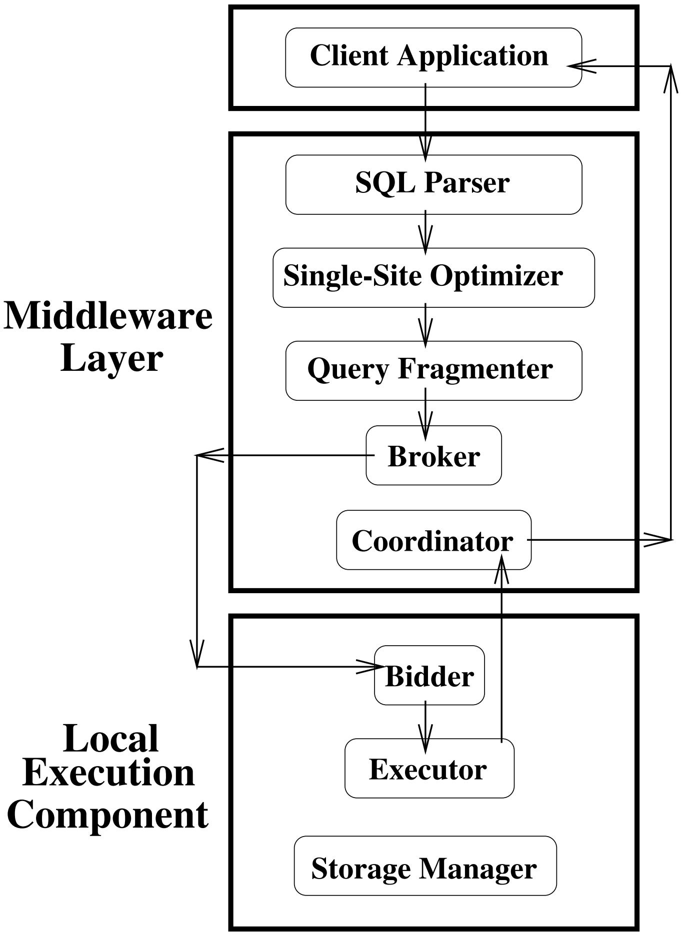

Figure 1. Mariposa architecture.

This section describes the role that each module plays in the Mariposa economy. In the process of describing the modules, we also give an overview of how query processing works in an economic framework. Section 3 will explain this process in more detail.

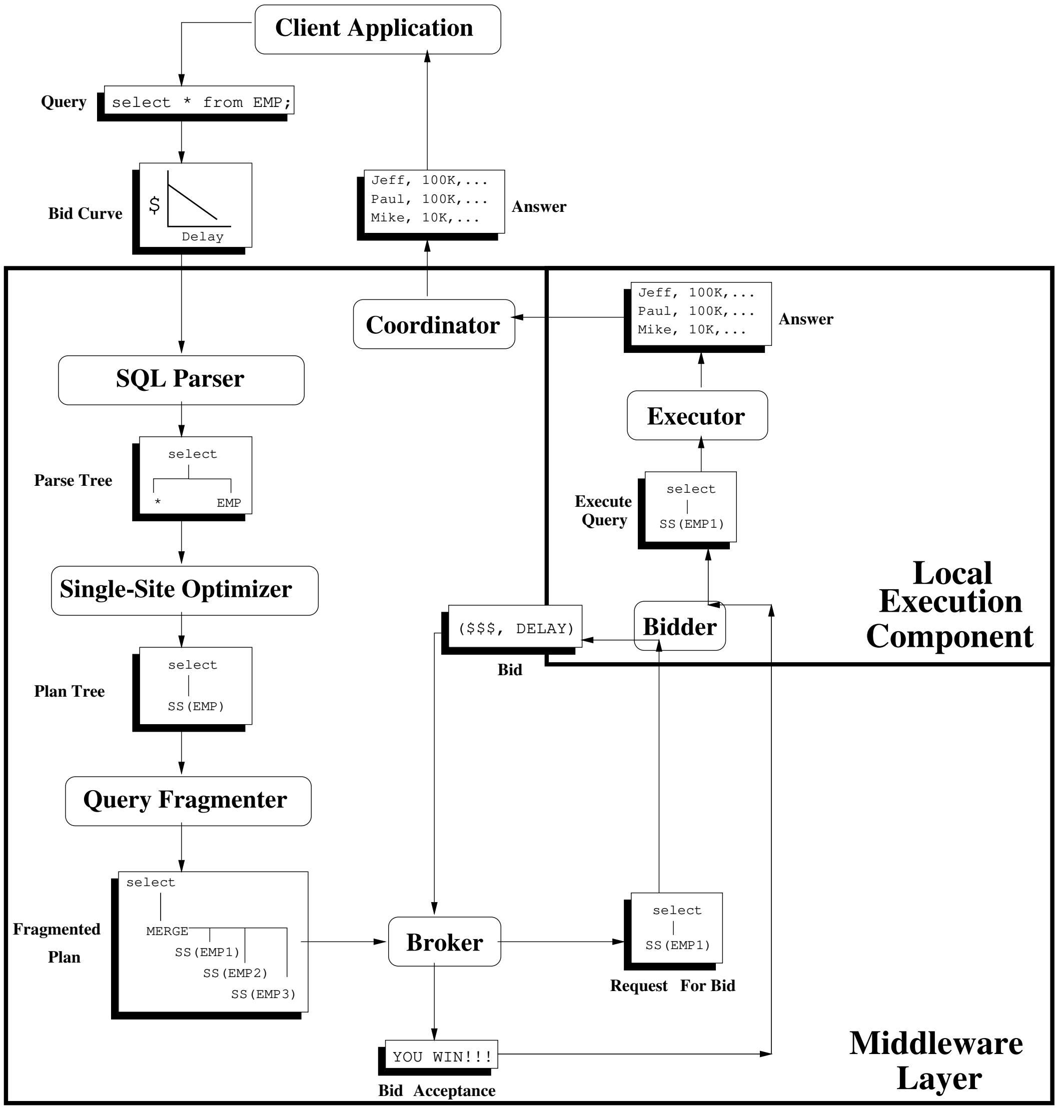

Figure 2. Mariposa communication.

Queries are submitted by the client application. Each query starts with a budget $B(t)$ expressed as a **bid**curve The budget indicates how much the user is willing to pay to have the query executed within time t. Query budgets form the basis of the Mariposa economy. Figure 2 includes a bid curve indicating that the user is willing to sacrifice performance for a lower price. Once a budget has been assigned (through administrative means not discussed here), the client software hands the query to Mariposa middleware.

Mariposa middleware contains an SQL parser, single-site optimizer, query fragmenter, broker, and coordinator module. The broker is primarily coded in Rush. Each of these modules is described below. The communication between modules is shown in Figure 2.

The parser parses the incoming query, performing name resolution and authorization. The parser first requests **metadata** for each table referenced in the query from some name server. This metadata contains information including the name and type of each attribute in the table, the location of each fragment of the table. and an indicator of the staleness of the information. Metadata is itself part of the economy and has a price. The choice of name server is determined by the desired quality of metadata, the prices offered by the name servers, the available budget, and any local Rush rules defined to prioritize these factors.

The parser hands the query, in the form of a parse tree, to the **single-site optimizer**. This is a conventional query optimizer along the lines of [SELI79]. The single-site optimizer generates a single-site query execution plan. The optimizer ignores data distribution and prepares a plan as if all the fragments were located at a single server site.

The **fragmenter** accepts the plan produced by the single-site optimizer. It uses location information previously obtained from the name server to decompose the single site plan into a **fragmented query plan**. The fragmenter decomposes each restriction node in the single site plan into subqueries, one per fragment in the referenced table. Joins are decomposed into one join sub-query for each pair of fragment joins. Lastly, the fragmenter groups the operations that can proceed in parallel into query **strides.** All subqueries in a stride must be completed before any subqueries in the next stride can begin. As a result, strides form the basis for intraquery synchronization. Notice that our notion of strides does not support **pipelining** the result of one subquery into the execution of a subsequent subquery. This complication would introduce sequentiality within a query stride and complicate the bidding process to be described. Inclusion of pipelining into our economic system is a task for future research.

The **broker** takes the collection of fragmented query plans prepared by the fragmenter and sends out requests for bids to various sites. After assembling a collection of bids, the broker decides which ones to accept and notifies the winning sites by sending out a **bid acceptance**. The bidding process will be described in more detail in Section 3.

The broker hands off the task of coordinating the execution of the resulting query strides to a **coordinator.** The coordinator assembles the partial results and returns the final answer to the user process.

At each Mariposa server site there is a local execution module, containing a **bidder,** a **storage manager,** and a local execution engine. The **bidder** responds to requests for bids and formulates its bid price and the speed with which the site will agree to process a subquery based on local resources such as CPU time, disk I/O bandwidth, storage, etc. If the bidder site does not have the data fragments specified in the subquery, it may refuse to bid or it may attempt to buy the data from another site by contacting its storage manager.

Winning bids must sooner or later be processed. To execute local queries, a Mariposa site contains a number of local execution engines. An idle one is allocated to each incoming subquery to perform the task at hand. The number of executors controls the multiprocessing level at each site, and may be adjusted as conditions warrant. The local executor sends the results of the subquery to the site executing the next part of the query or back to the coordinator process.

At each Mariposa site there is also a **storage manager,** which watches the revenue stream generated by stored fragments. Based on space and revenue considerations, it engages in buying and selling fragments with storage managers at other Mariposa sites.

The storage managers, bidders and brokers in our prototype are primarily coded in the rule language Rush. Rush is an embeddable programming language with syntax similar to Tcl [OUST94] that also includes rules of the form:

```
on <condition> do <action>
```

Every Mariposa entity embeds a Rush interpreter, calling it to execute code to determine the behavior of Mariposa.

Rush conditions can involve any combination of primitive Mariposa events, described below, and computations on Rush variables. Actions in Rush can trigger Mariposa primitives and modify Rush variables. As a result, Rush can be thought of as a fairly conventional forward-chaining rule system. We chose to implement our own system, rather than use one of the packages available from the AI community, primarily for performance reasons. Rush rules are in the ''inner loop'' of many Mariposa activities, and as a result, rule interpretation must be very fast. A separate paper [SAH94b] discusses how we have achieved this goal.

Mariposa contains a specific inter-site protocol by which Mariposa entities communicate. Requests for bids to execute subqueries and to buy and sell fragments can be sent between sites. Additionally, queries and data must be passed around. The main messages are indicated in Table 1. Typically, the outgoing message is the action part of a Rush rule, and the corresponding incoming message is a Rush event at the recipient site.

3. THE BIDDING PROCESS

Each query Q has a **budget** $B(t)$ that can be used to solve the query. The budget is a non-increasing function of time that represents the value the user gives to the answer to his query at a particular time t. Constant functions represent a willingness to pay the same amount of money for a slow answer as for a quick one, while steeply declining functions indicate that the user will pay more for a fast answer.

Actions (messages)	Events (received messages)
Request_bid	Receive_bid_request
Bid	Receive_bid
Award_Contract	Contract_won
Notify_loser	Contract_lost
Send_query	Receive_query
Send_data	Receive_data

Table 1. The main Mariposa primitives.

The broker handling a query Q receives a query plan containing a collection of subqueries, $Q_1, \ldots, Q_n$, and $B(t)$. Each subquery is a one-variable restriction on a fragment F of a table or a join between two fragments of two tables. The broker tries to solve each subquery, Q_i, using either an **expensive bid protocol** or a cheaper **purchase order protocol**.

The expensive bid protocol involves two phases: in the first phase, the broker sends out requests for bids to bidder sites. A bid request includes the portion of the query execution plan being bid on. The bidders send back bids that are represented as triples: (C_i, D_i, E_i). The triple indicates that the bidder will solve the subquery Q_i for a cost C_i within a delay D_i after receipt of the subquery, and that this bid is only valid until the expiration date, E_i.

In the second phase of the bid protocol, the broker notifies the winning bidders that they have been selected. The broker may also notify the losing sites. If it does not, then the bids will expire and can be deleted by the bidders. This process requires many (expensive) messages. Most queries will not be computationally demanding enough to justify this level of overhead. These queries will use the simpler **purchase order** protocol.

The purchase order protocol sends each subquery to the processing site that would be most likely to win the bidding process if there were one; for example, one of the storage sites of a fragment for a sequential scan. This site receives the query and processes it, returning the answer with a **bill** for services. If the site refuses the subquery, it can either return it to the broker or pass it on to a third processing site. If a broker uses the cheaper purchase order protocol, there is some danger of failing to solve the query within the allotted budget. The broker does not always know the cost and delay which will be charged by the chosen processing site. However, this is the risk that must be taken to use this faster protocol.

3.1. Bid Acceptance

All subqueries in each stride are processed in parallel, and the next stride cannot begin until the previous one has been completed. Rather than consider bids for individual subqueries, we consider collections of bids for the subqueries in each stride.

When using the bidding protocol, brokers must choose a winning bid for each subquery with aggregate cost C and aggregate delay D such that the aggregate cost is less than or equal to the cost requirement $B(D)$. There are two problems that make finding the best bid collection difficult: subquery parallelism and the combinatorial search space. The aggregate delay is not the sum of the delays D_i for each subquery Q_i, since there is parallelism within each stride of the query plan. Also, the number of possible bid collections grows exponentially with the number of strides in the query plan. For example, if there are 10 strides and 3 viable bids for each one, then the broker can evaluate each of the 3^{10} bid possibilities.

The estimated delay to process the collection of subqueries in a stride is equal to the highest bid time in the collection. The number of different delay values can be no more than the total number of bids on subqueries in the collection. For each delay value the optimal bid collection is the least expensive bid for each subquery that can be processed within the given delay. By coalescing the bid collections in a stride and considering them as a single (aggregate) bid, the broker may reduce the bid acceptance problem to the simpler problem of choosing one bid from among a set of aggregated bids for each query stride.

With the expensive bid protocol, the broker receives a collection of zero or more bids for each subquery. If there is no bid for some subquery or no collection of bids meets the client's minimum price and performance requirements ($B(D)$), then the broker must solicit additional bids, agree to perform the subquery itself, or notify the user that the query cannot be run. It is possible that several collections of bids meet the minimum requirements, so the broker must choose the best collection of bids. In order to compare the bid collections, we define a *difference* function on the collection of bids: $\textit{difference} = B(D) - C$. Note that this can have a negative value, if the cost is above the bid curve.

For all but the simplest queries referencing tables with a minimal number of fragments, exhaustive search for the best bid collection will be combinatorially prohibitive. The crux of the problem is in determining the relative amounts of the time and cost resources that should be allocated to each subquery. We offer a heuristic algorithm that determines how to do this. Although it cannot be shown to be optimal, we believe in practice it will demonstrate good results. Preliminary performance numbers for Mariposa are included later in this paper which support this supposition. A more detailed evaluation and comparison against more complex algorithms is planned in the future.

The algorithm is a **greedy** one. It produces a trial solution in which the total delay is the smallest possible, and then makes the greediest substitution until there are no more profitable ones to make. Thus a series of solutions are proposed with steadily increasing delay values for each processing step. On any iteration of the algorithm, the proposed solution contains a collection of bids with a certain delay for each processing step. For every collection of bids with greater delay a **cost gradient** is computed. This cost gradient is the cost decrease that would result for the processing step by replacing the collection in the solution by the collection being considered, divided by the time increase that would result from the substitution.

The algorithm begins by considering the bid collection with the smallest delay for each processing step and compute the total cost C and the total delay D. Compute the cost gradient for each unused bid. Now, consider the processing step that contains the unused bid with the maximum cost gradient, B'. If this bid replaces the current one used in the processing step, then cost will become C' and delay D'. If the resulting *difference* is greater at D' than at D, then make the bid substitution. That is, if $B(D') - C' > B(D) - C$, then replace B with B'. Recalculate all the cost gradients for the processing step that includes B', and continue making substitutions until there are none that increase the *difference*.

Notice that our current Mariposa algorithm decomposes the query into executable pieces, and then the broker tries to solve the individual pieces in a heuristically optimal way. We are planning to extend Mariposa to contain a second bidding strategy. Using this strategy, the single-site optimizer and fragmenter would be bypassed. Instead, the broker would get the entire query directly. It would then decide whether to decompose it into a collection of two or more ‘‘hunks’’ using heuristics yet to be developed. Then, it would try to find contractors for the hunks, each of which could freely subdivide the hunks and subcontract them. In contrast to our current query processing system which is a ‘‘bottom up’’ algorithm, this alternative would be a ‘‘top down’’ decomposition strategy. We hope to implement this alternative and test it against our current system.

3.2. Finding Bidders

Using either the expensive bid or the purchase order protocol from the previous section, a broker must be able to identify one or more sites to process each subquery. Mariposa achieves this through an advertising system. Servers announce their willingness to perform various services by posting **advertisements**. Name servers keep a record of these advertisements in an **Ad Table**. Brokers examine the Ad Table to find out which servers might be willing to perform the tasks they need. Table 2 shows the fields of the Ad Table.

In practice, not all these fields will be used in each advertisement. The most general advertisements will specify the fewest number of fields. Table 3 summarizes the valid fields for some types of advertisement.

Using **yellow pages**, a server advertises that it offers a specific service (e.g., processing queries that reference a specific fragment). The date of the advertisement helps a broker decide how timely the yellow pages entry is, and therefore how much faith to put in the information. A server can issue a new yellow pages advertisement at any time without explicitly revoking a previous one.

In addition, a server may indicate the price and delay of a service. This is a **posted price** and becomes current on the start-date indicated. There is no guarantee that the price will hold beyond that time and, as with yellow pages, the server may issue a new posted price without revoking the old one.

Ad Table Field	Description

query-template	A description of the service being offered. The query template is a query with parameters left unspecified. For example,
	`SELECT param-1`
	`FROM EMP`
	indicates a willingness to perform any SELECT query on the EMP table, while
	`SELECT param-1`
	`FROM EMP`
	`WHERE NAME = param-2`
	indicates that the server wants to perform queries that perform an equality restriction on the NAME column.

server-id	The server offering the service.
start-time	The time at which the service is first offered. This may be a future time, if the server expects to begin performing certain tasks at a specific point in time.
expiration-time	The time at which the advertisement ceases to be valid.
price	The price charged by the server for the service.
delay	The time in which the server expects to complete the task.
limit-quantity	The maximum number of times the server will perform a service at the given cost and delay.
bulk-quantity	The number of orders needed to obtain the advertised price and delay.
to-whom	The set of brokers to whom the advertised services are available.
other-fields	Comments and other information specific to a particular advertisement.

Table 2. Fields in the Ad Table.

Ad Table Field	Type of Advertisement				
	Yellow Pages	Posted Price	Sale Price	Coupon	Bulk Purchase
query-template	√	√	√	√	√
server-id	√	√	√	√	√
start-date	√	√	√	√	√
expiration-date	-	-	√	√	√
price	-	√	√	√	√
delay	-	√	√	√	√
limit-quantity	-	-	-	√	-
bulk-quantity	-	-	-	-	√
to-whom	-	-	-	*	*
other-fields	*	*	*	*	*

Key: - = null, √ = valid, * = optional

Table 3. Ad Table fields applicable to each type of advertisement.

Several more specific types of advertisements are available. If the expiration-date field is set, then the details of the offer are known to be valid for a certain period of time. Posting a **sale price** in this manner involves some risk, as the advertisement may generate more demand than the server can meet, forcing it to pay heavy penalties. This risk can be offset by issuing **coupons**, which, like supermarket coupons, place a limit on the number of queries that can be executed under the terms of the advertisement. Coupons may also limit the brokers who are eligible to redeem them. These are similar to the coupons issued by the Nevada gambling establishments, which require the client to be over 21 and possess a valid California driver's license.

Finally, **bulk purchase contracts** are renewable coupons that allow a broker to negotiate cheaper prices with a server in exchange for guaranteed, pre-paid service. This is analogous to a travel agent who books 10 seats on each sailing of a cruise ship. We allow the option of guaranteeing bulk purchases, in which case the broker must pay for the specified queries whether it uses them or not. Bulk purchases are especially advantageous in transaction processing environments, where the workload is predictable, and brokers solve large numbers of similar queries.

Besides referring to the Ad Table, we expect a broker to remember sites that have bid successfully for previous queries. Presumably the broker will include such sites in the bidding process, thereby generating a system that learns over time which processing sites are appropriate for various queries. Lastly, the broker also knows the likely location of each fragment, which was returned previously to the query preparation module by the name server. The site most likely to have the data is automatically a likely bidder.

3.3. Setting The Bid Price For Subqueries

When a site is asked to bid on a subquery, it must respond with a triple (C, D, E) as noted earlier. This section discusses our current bidder module and some of the extensions that we expect to make. As noted earlier, it is coded primarily as Rush rules and can be changed easily.

The **naive** strategy is to maintain a **billing rate** for CPU and I/O resources for each site. These constants are be set by a site administrator based on local conditions. The bidder constructs an estimate of the amount of each resource required to process a subquery for objects that exist at the local site. A simple computation then yields the required bid. If the referenced object is not present at the site, then the site declines to bid. For join queries, the site declines to bid unless one of of following two conditions are satisfied:

- it possesses one of the two referenced objects.
- it had already bid on a query, whose answer formed one of the two referenced objects.

The time in which the site promises to process the query is calculated with an estimate of the resources required. Under zero load, it is an estimate of the elapsed time to perform the query. By adjusting for the current load on the site, the bidder can estimate the expected delay. Finally, it multiplies by a site-specific safety factor to arrive at a promised delay (the D in the bid). The expiration date on a bid is currently assigned arbitrarily as the promised delay plus a site-specific constant.

This naive strategy is consistent with the behavior assumed of a local site by a traditional global query optimizer. However, our current prototype improves on the naive strategy in three ways.

First, each site maintains a billing rate on a per-fragment basis. In this way, the site administrator can bias his bids toward fragments whose business he wants and away from those whose business he does not want. The bidder also automatically declines to bid on queries referencing fragments with billing rates below a site-specific threshold. In this case, the query will have to be processed elsewhere, and another site will have to buy or copy the indicated fragment in order to solve the user query. Hence, this tactic will hasten the sale of low value fragments to somebody else.

Our second improvement concerns adjusting bids based on the current site load. Specifically, each site maintains its current load average by periodically running a UNIX utility. It then adjusts its bid based on its current load average as follows:

$$\textit{actual bid} = \textit{computed bid} \times \textit{load average}$$

In this way, if it is nearly idle (i.e., its load average is near zero), it will bid very low prices. Conversely, it will bid higher and higher prices as its load increases. Notice that this simple formula will ensure a crude form of load balancing among a collection of Mariposa sites.

Our third improvement concerns bidding on subqueries when the site does not possess any of the data. As will be seen in the next section, the storage manager buys and sells fragments to try to maximize site revenue. In addition, it keeps a **hot list** of fragments it would like to acquire but has not yet done so. The bidder automatically bids on any query which references a hot list fragment. In this way, if it gets a contract for the query, it will instruct the storage manager to accelerate the purchase of the fragment, which is in line with the goals of the storage manager.

In the future we expect to increase the sophistication of the bidder substantially. We plan more sophisticated integration between the bidder and the storage manager. We view hot lists as merely the first primitive step in this direction. Furthermore, we expect to adjust the billing rate for each fragment automatically, based on the amount of business for the fragment. Finally, we hope to increase the sophistication of our choice of expiration dates. Choosing an expiration date far in the future incurs the risk of honoring lower out-of-date prices. Specifying an expiration date that is too close means running the risk of the broker not being able to use the bid because of inherent delays in the processing engine.

Lastly, we expect to consider network resources in the bidding process. Our proposed algorithms are discussed in the next subsection.

3.4. The Network Bidder

In addition to producing bids based on CPU and disk usage, the processing sites need to take the available network bandwidth into account. The network bidder will be a separate module in Mariposa. Since network bandwidth is a distributed resource, the network bidders along the path from source to destination must calculate an aggregate bid for the entire path and must reserve network resources as a group. Mariposa will use a version of the Tenet network protocols RTIP [ZHAN92] and RCAP [BANE91] to perform bandwidth queries and network resource reservation.

A network bid request will be made by the broker to transfer data between source/destination pairs in the query plan. The network bid request is sent to the destination node. The request is of the form: *(transaction-id, request-id, data size, from-node, to-node)*. The broker receives a bid from the network bidder at the destination node of the form: *(transaction-id, request-id, price, time)*. In order to determine the price and time, the network bidder at the destination node must contact each of the intermediate nodes between itself and the source node.

For convenience, call the destination node n_0 and the source node n_k. (See Figure 3.) Call the first intermediate node on the path from the destination to the source n_1, the second such node n_2, etc. Available bandwidth between two adjacent nodes as a function of time is represented as a **bandwidth profile**. The bandwidth profile contains entries of the form *(available bandwidth,* t_1*,* t_2*)* indicating the available bandwidth between time t_1 and time t_2. If n_i and n_{i-1} are directly-connected nodes on the path from the source to the destination, and data is flowing from n_i to n_{i-1}, then node n_i is responsible for keeping track of (and charging for) available bandwidth between itself and n_{i-1} and therefore maintains the bandwidth profile. Call the bandwidth profile between node n_i and node n_{i-1} B_i and the price n_i charges for a bandwidth reservation P_i.

The available bandwidth on the entire path from source to destination is calculated step by step starting at the destination node, n_0. Node n_0 contacts n_1 which has B_1, the bandwidth profile for the network link between itself and n_0. It sends this profile to node n_2, which has the bandwidth profile B_2. Node n_2 calculates $min(B_1, B_2)$, producing a bandwidth profile that represents the available bandwidth along the path from n_2 to n_0. This process continues along each intermediate link, ultimately reaching the source node.

When the bandwidth profile reaches the source node, it is equal to the minimum available bandwidth over all links on the path between the source and destination, and represents the amount of

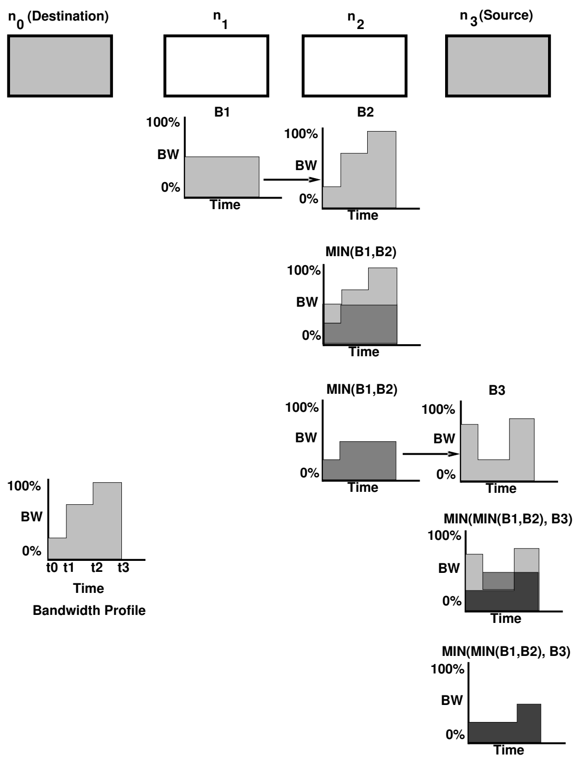

Figure 3. Calculating a bandwidth profile.

bandwidth available as a function of time on the entire path. The source node, n_k, then initiates a backward pass to calculate the price for this bandwidth along the entire path. Node n_k sends its price to reserve the bandwidth, P_k, to node n_{k-1}, which adds its price, and so on until the aggregate price arrives at the destination, n_0. Bandwidth could also be reserved at this time. If bandwidth is reserved at bidding time, there is a chance that it will not be used (if the source or destination is not chosen by the broker). If bandwidth is not reserved at this time, then there will be a window of time between bidding and bid award when the available bandwidth may have changed. We are investigating approaches to this problem.

In addition to the choice of when to reserve network resources, there are two choices for when the broker sends out network bid requests during the bidding process. The broker could send out requests for network bids at the same time that it sends out other bid requests, or it could wait until the single-site bids have been returned and then send out requests for network bids to the winners of the first phase. In the first case, the broker would have to request a bid from every pair of sites that could potentially communicate with one another. If P is the number of parallelized phases of the query plan, and S_i is the number of sites in phase i, then this approach would produce a total of $\sum_{i=2}^{P} S_i S_{i-1}$ bids. In the second case, the broker only has to request bids between the winners of each phase of the query plan. If $winner_i$ is the winning group of sites for phase i, then the number of network bid requests sent out is $\sum_{i=2}^{P} S_{winner_i} S_{winner_{i-1}}$.

The first approach has the advantage of parallelizing the bidding phase itself and thereby reducing the optimization time. However, the sites that are asked to reserve bandwidth are not guaranteed to win the bid. If they reserve all the bandwidth for each bid request they receive, this approach will result in reserving more bandwidth than is actually needed. This difficulty may be overcome by reserving less bandwidth than is specified in bids, essentially "overbooking the flight."

4. STORAGE MANAGEMENT

Each site manages a certain amount of storage, which it can fill with fragments or copies of fragments. The basic objective of a site is to allocate its CPU, I/O and storage resources so as to maximize its revenue income per unit time. This topic is the subject of the first part of this section. After that, we turn to the splitting and coalescing of fragments into smaller or bigger storage units.

4.1. Buying and Selling Fragments

In order for sites to trade fragments, they must have some means of calculating the (expected) value of the fragment for each site. Some access history is kept with each fragment so sites can make predictions of future activity. Specifically, a site maintains the **size** of the fragment as well as its **revenue history**. Each record of the history contains the query, number of records which qualified, time-since-last-query, revenue, delay, I/O-used, and CPU-used. The CPU and I/O information is normalized and stored in site-independent units.

To estimate the revenue that a site would receive if it owned a particular fragment, the site must assume that access rates are stable and that the revenue history is therefore a good predictor of future revenue. Moreover, it must convert site-independent resource usage numbers into ones specific to its site through a weighting function, as in [LOHM86]. In addition, it must assume that it would have successfully bid on the same set of queries as appeared in the revenue history. Since it will be faster or slower than the site from which the revenue history was collected, it must adjust the revenue collected for each query. This calculation requires the site to assume a shape for the average bid curve. Lastly, it must convert the adjusted revenue stream into a cash value, by computing the net present value of the stream.

If a site wants to bid on a subquery, then it must either **buy** any fragment(s) referenced by the subquery or subcontract out the work to another site. If the site wishes to buy a fragment, it can do so either when the query comes in (*on demand*) or in advance (*prefetch*). To purchase a fragment, a buyer locates the owner of the fragment and requests the revenue history of the fragment, and then places a value on the fragment. Moreover, if it buys the fragment, then it will have to evict a collection of fragments to free up space, adding to the cost of the fragment to be purchased. To the extent that storage is not full, then fewer (or no) evictions will be required. In any case, this collection is called the **alternate fragments** in the formula below.

Hence, the buyer will be willing to bid the following price for the fragment:

$$\mathit{offer\ price} = \mathit{value\ of\ fragment} - \mathit{value\ of\ alternate\ fragments} + \mathit{price\ received}$$

In this calculation, the buyer will obtain the value of the new fragment but lose the value of the fragments that it must evict. Moreover, it will **sell** the evicted fragments, and receive some price for them. The latter item is problematic to compute. A plausible assumption is that *price received* is equal to the value of the alternate fragments. A more conservative assumption is that the price obtained is zero. Note that in this case the offer price need not be positive.

The potential seller of the fragment performs the following calculation: The site will receive the offered price and will lose the value of the fragment which is being evicted. However, if the fragment is not evicted, then a collection of alternate fragments summing in size to the indicated fragment must be evicted. In this case, the site will lose the value of these (more desirable) fragments, but will receive the expected *price received*. Hence, it will be willing to sell the fragment, transferring it to the buyer:

$$\mathit{offer\ price} > \mathit{value\ of\ fragment} - \mathit{value\ of\ alternate\ fragments} + \mathit{price\ received}$$

Again, *price received* is problematic, and subject to the same plausible assumptions noted above.

Sites may sell fragments at any time, for any reason. For example, decommissioning a server implies that the server will sell all of its fragments.

To sell a fragment, the site conducts a bidding process, essentially identical to the one used for subqueries above. Specifically, it sends the revenue history to a collection of **potential bidders** and asks them what they will offer for the fragment. The seller considers the highest bid and will **accept** the bid under the same considerations that applied when selling fragments on request, namely if:

$$offered\ price > value\ of\ fragment - value\ of\ alternate\ fragments + price\ received$$

If no bid is acceptable, then the seller must try to evict another (higher value) fragment until one is found that can be sold. If no fragments are sellable, then the site must lower the value of its fragments until a sale can be made. In fact, if a site wishes to go out of business, then it must find a site to accept its fragments, and must lower their internal value until a buyer can be found for all of them.

The storage manager is an asynchronous process running in the background, continually buying and selling fragments. Obviously, it should work in harmony with the bidder mentioned in the previous section. Specifically, the bidder should bid on queries for remote fragments that the storage manager would like to buy, but has not yet done so. In contrast, it should decline to bid on queries to remote objects, in which the storage manager has no interest. The first primitive version of this interface is the "hot list" mentioned in the the previous section.

4.2. Splitting and Coalescing

Mariposa sites must also decide when to split and coalesce fragments. Clearly, if there are too few fragments in a class, then parallel execution of Mariposa queries will be hindered. On the other hand, if there are too many fragments, then the overhead of dealing with all the fragments will increase and response time will suffer, as noted in [COPE88]. The algorithms for splitting and coalescing fragments must strike the correct balance between these two effects.

At the current time, our storage manager does not have general Rush rules to deal with splitting and coalescing fragments. Hence, this section indicates our current plans for the future.

One strategy is to let market pressure correct inappropriate fragment sizes. Large fragments have high revenue and attract many bidders for copies, thereby diverting some of the revenue away from the owner. If the owner site wants to keep the number of copies low, it has to break up the fragment into smaller fragments, which have less revenue and are less attractive for copies. On the other hand, a small fragment has high processing overhead for queries. Economies of scale could be realized by coalescing it with another fragment in the same class into a single larger fragment.

If more direct intervention is required, then Mariposa might resort to the following tactic. Consider the execution of queries referencing only a single class. The broker can fetch the number of fragments, Num_C, in that class from a name server and, assuming that all fragments are the same size, can compute the expected delay (ED) of a given query on the class if run on all fragments in parallel. The budget function tells the broker the total amount that is available for the entire query under that delay. The amount of the expected feasible bid per site in this situation is:

$$expected\ feasible\ site\ bid = \frac{B(ED)}{Num_C}$$

The broker can repeat those calculations for a variable number of fragments to arrive at Num^*, the number of fragments to maximize the expected revenue per site.

This value, *Num* *, can be published by the broker along with its request for bids. If a site has a fragment that is too large (or too small), then in steady state it will be able to obtain a larger revenue per query if it splits (coalesces) the fragment. Hence, if a site keeps track of the average value of *Num* * for each class for which it stores a fragment, then it can decide whether its fragments should be split or coalesced.

Of course, a site must honor any outstanding contracts that it has already made. If it discards or splits a fragment for which there is an outstanding contract, then the site must endure the consequences of its actions. This entails either subcontracting to some other site a portion of the previously committed work or buying back the missing data. In either case, there are revenue consequences, and a site should take its outstanding contracts into account when it makes fragment allocation decisions. Moreover, a site should carefully consider the desirable expiration time for contracts. Shorter times will allow the site greater flexibility in allocation decisions.

5. NAMES AND NAME SERVICE

Current distributed systems use a rigid naming approach, assume that all changes are globally synchronized, and often have a structure that limits the scalability of the system. The Mariposa goals of mobile fragments and avoidance of global synchronization require that a more flexible naming service be used. We have developed a decentralized naming facility that does not depend on a centralized authority for name registration or binding.

5.1. Names

Mariposa defines four structures used in object naming. These structures (internal names, full names, common names and name contexts) are defined below.

Internal names are location-dependent names used to determine the physical location of a fragment. Because these are low-level names that are defined by the implementation, they will not be described further.

Full names are completely-specified names that uniquely identify an object. A full name can be resolved to any object regardless of location. Full names are not specific to the querying user and site and are location-independent so that when a query or fragment moves the full name is still valid. A name consists of components describing attributes of the containing table, and a full name has all components fully specified.

In contrast, **common names** (sometimes known as synonyms) are user-specific partially specified names. Using them avoids the tedium of using a full name. Simple rules permit the translation of common names into full names by supplying the missing name components. The binding operation gathers the missing parts either from parameters directly supplied by the user or from the user's environment as stored in the system catalogs. Common names may be ambiguous because different users may refer to different objects using the same name. Because common names are context dependent, they may even refer to different objects at different times.

Translation of common names is performed by functions written in the Mariposa rule/extension language, stored in the system catalogs, and invoked by the module (e.g., the parser) that requires the name to be resolved. Translation functions may take several arguments and return a string containing a Boolean expression that looks like a query qualification. This string is then stored internally by the invoking module when called by the name service module. The user may invoke translation functions directly, e.g., `my_naming(EMP)`. Since we expect most users to have a "usual" set of name parameters, a user may specify one such function (taking the name string as its only argument) as a default in the `USER` system catalog. When the user specifies a simple string (e.g., `EMP`) as a common name, the system applies this default function.

Finally, a **name context** is a set of affiliated names. Names within a context are expected to share some feature. For example, they may be often used together in an application (e.g., a directory) or they may form part of a more complex object (e.g., a class definition). A programmer can define a name context for global use that everyone can access or a private name context that is visible only to a single application. The advantage of a name context is that names do not have to be globally registered nor are the names tied to a physical resource to make them unique, such as the birth site used in [WILL81].

Like other objects, a name context can also be named. In addition, like data fragments, it can be migrated between name servers and there can be multiple copies residing on different servers for better load balancing and availability.

This scheme differs from another proposed decentralized name service [CHER89] that avoided a centralized name authority by relying upon each type of server to manage their own names without relying on a dedicated name service.

5.2. Name Resolution

A name must be resolved to discover which object is bound to the name. Every client and server has a name cache at the site to support the local translation of common names to full names and of full names to internal names. When a broker wants to resolve a name, it first looks in the local name cache to see if a translation exists. If the cache does not yield a match, the broker uses a rule-driven search to resolve ambiguous common names. If a broker still fails to resolve a name using its local cache, it will query one or more name servers for additional name information.

As previously discussed, names are unordered sets of attributes. In addition, since the user may not know all of an object's attributes, it may be incomplete. Finally, common names may be ambiguous (more than one match) or untranslatable (no matches). When the broker discovers that there are multiple matches to the same common name, it tries to pick one according to the policy specified in its rule base. Some possible policies are "first match," as exemplified by the UNIX shell command search (path), or a policy of "best match" that uses additional semantic criteria. Considerable information may exist that the broker can apply to choose the best match, such as data types, ownership, and protection permissions.

5.3. Name Discovery

In Mariposa, a name server responds to metadata queries in the same way as data servers execute regular queries, except that they translate common names into full names using a list of name contexts provided by the client. The name service process uses the bidding protocol of Section 3 to interact with a collection of potential bidders. The name service chooses the winning name server based on economic considerations of cost and quality of service. Mariposa expects multiple name servers, and this collection may be dynamic as name servers are added to and removed from a Mariposa environment. Name servers are expected to use advertising to find clients.

Each name server must make arrangements to read the local system catalogs at the sites whose catalogs it serves periodically and build a composite set of metadata. Since there is no requirement for a processing site to notify a name server when fragments change sites or are split or coalesced, the name server metadata may be substantially out of date.

As a result, name servers are differentiated by their **quality of service** regarding their price and the staleness of their information. For example, a name server that is less than one minute out of date generally has better quality information than one which can be up to one day out of date. Quality is best measured by the maximum staleness of the answer to any name service query. Using this information a broker can make an appropriate tradeoff between price, delay and quality of answer among the various name services, and select the one that best meets its needs.

Quality may be based on more than the name server's polling rate. An estimate of the real quality of the metadata may be based on the observed rate of update. From this we predict the chance that an invalidating update will occur for a time period after fetching a copy of the data into the local cache. The benefit is that the calculation can be made without probing the actual metadata to see if it has changed. The quality of service is then a measurement of the metadata's rate of update as well as the name server's rate of update.

6. MARIPOSA STATUS AND EXPERIMENTS

At the current time (June 1995), a complete Mariposa implementation using the architecture described in this paper is operational on Digital Equipment Corp. Alpha AXP workstations running Digital UNIX. The current system is a combination of old and new code. The basic server engine is that of POSTGRES [STON91a], modified to accept SQL instead of POSTQUEL. In addition, we have implemented the fragmenter, broker, bidder and coordinator modules to form the complete Mariposa system portrayed in Figure 1.

Building a functional distributed system has required the addition of a substantial amount of software infrastructure. For example, we have built a multithreaded network communication package using ONC RPC and POSIX threads. The primitive actions shown in Table 1 have been implemented as RPCs and are available as Rush procedures for use in the action part of a Rush rule. Implementation of the Rush language itself has required careful design and performance engineering, as described in [SAH94b].

We are presently extending the functionality of our prototype. At the current time, the fragmenter, coordinator and broker are fairly complete. However, the storage manager and the bidder are simplistic, as noted earlier. We are in the process of constructing more sophisticated routines in these modules. In addition, we are implementing the replication system described in [SIDE95]. We plan to release a general Mariposa distribution when these tasks are completed later this year.

The rest of this section presents details of a few simple experiments which we have conducted in both LAN and WAN environments. The experiments demonstrate the power, performance and flexibility of the Mariposa approach to distributed data management. First, we describe the experimental setup. We then show by measurement that the Mariposa protocols do not add excessive overhead relative to those in a traditional distributed DBMS. Finally, we show how Mariposa query optimization and execution compares to that of a traditional system.

6.1. Experimental Environment

The experiments were conducted on Alpha AXP workstations running versions 2.1 and 3.0 of Digital UNIX. Table 4 shows the actual hardware configurations used. The workstations were connected by a 10 Mbps Ethernet in the LAN case and the Internet in the WAN case. The WAN experiments were performed after midnight in order to avoid heavy daytime Internet traffic that would cause excessive bandwidth and latency variance.

The results in this section were generated using a simple synthetic dataset and workload. The database consists of three tables, R1, R2 and R3. The tables are part of the Wisconsin Benchmark database [BITT83], modified to produce results of the sizes indicated in Table 5. We make available statistics that allow a query optimizer to estimate the size of (R1 join R2), (R2 join R3) and (R1 join R2 join R3) as 1 MB, 3 MB and 4.5 MB, respectively. The workload query is an equijoin of all three tables:

	WAN				LAN			
Site	**Host**	**Location**	**Model**	**Memory**	**Host**	**Location**	**Model**	**Memory**
1	huevos	Santa Barbara	3000/600	96 MB	arcadia	Berkeley	3000/400	64 MB
2	triplerock	Berkeley	2100/500	256 MB	triplerock	Berkeley	2100/500	256 MB
3	pisa	San Diego	3000/800	128 MB	nobozo	Berkeley	3000/500X	160 MB

Table 4. Mariposa site configurations.

Table	Location	Number of Rows	Total Size
R1	site 1	50,000	5 MB
R2	site 2	10,000	1 MB
R3	site 3	50,000	5 MB

Table 5. Parameters for the experimental test data.

```
SELECT *
FROM R1, R2, R3
WHERE R1.u1 = R2.u1
  AND R2.u1 = R3.u1
```

In the wide area case, the query originates at Berkeley and performs the join over the WAN connecting U. C. Berkeley, U. C. Santa Barbara and U. C. San Diego.

6.2. Comparison of the Purchase Order and Expensive Bid Protocols

Before discussing the performance benefits of the Mariposa economic protocols, we should quantify the overhead they add to the process of constructing and executing a plan relative to a traditional distributed DBMS. We can analyze the situation as follows. A traditional system plans a query and sends the subqueries to the processing sites; this process follows essentially the same steps as the purchase order protocol discussed in Section 3. However, Mariposa can choose between the purchase order protocol and the expensive bid protocol. As a result, Mariposa overhead (relative to the traditional system) is the difference in elapsed time between the two protocols, weighted by the proportion of queries that actually use the expensive bid protocol.

To measure the difference between the two protocols, we repeatedly executed the three-way join query described in the previous section over both a LAN and a WAN. The elapsed times for the various processing stages shown in Table 6 represent averages over ten runs of the same query. For this experiment, we did not install any rules that would cause fragment migration and did not change any optimizer statistics. The query was therefore executed identically every time. Plainly, the only difference between the purchase order and the expensive bid protocol is in the brokering stage.

Network	Stage	Time (s)	
		Purchase Order Protocol	Expensive Bid Protocol
LAN	parser	0.18	0.18
	optimizer	0.08	0.08
	broker	1.72	6.69
WAN	parser	0.18	0.18
	optimizer	0.08	0.08
	broker	4.52	14.08

Table 6. Elapsed times for various query processing stages.

The difference in elapsed time between the two protocols is due largely to the message overhead of brokering, but not in the way one would expect from simple message counting. In the purchase order protocol, the single-site optimizer determines the sites to perform the joins and awards contracts to the sites accordingly. Sending the contracts to the two remote sites involves two round-trip network messages (as previously mentioned, this is no worse than the cost in a traditional distributed DBMS of initiating remote query execution). In the expensive bid protocol, the broker sends out Request for Bid (RFB) messages for the two joins to each site. However, each prospective join processing site then sends out subbids for remote table scans. The whole brokering process therefore involves 14 round-trip messages for RFBs (including subbids), 6 round-trip messages for recording the bids and 2 more for notifying the winners of the two join subqueries. Note, however, that the bid collection process is executed in parallel because the broker and the bidder are multithreaded, which accounts for the fact that the additional cost is not as high as might be thought.

As is evident from the results presented in Table 6, the expensive bid protocol is not unduly expensive. If the query takes more than a few minutes to execute, the savings from a better query processing strategy can easily outweigh the small cost of bidding. Recall that the expensive protocol will only be used when the purchase order protocol cannot be. We expect the less expensive protocol to be used the majority of the time. The next subsection shows how economic methods can produce better query processing strategies.

6.3. Bidding in a Simple Economy

We illustrate how the economic paradigm works by running the three-way distributed join query described in the previous section repeatedly in a simple economy. We discuss how the query optimization and execution strategy in Mariposa differs from traditional distributed database systems and how Mariposa achieves an overall performance improvement by adapting its query processing strategy to the environment. We also show how data migration in Mariposa can automatically ameliorate poor initial data placement.

In our simple economy, each site uses the same pricing scheme and the same set of rules. The expensive bid protocol is used for every economic transaction. Sites have adequate storage space and never need to evict alternate fragments to buy fragments. The exact parameters and decision rules used to price queries and fragments are as follows:

Queries: Sites bid on subqueries as described in Section 3.3. That is, a bidder will only bid on a join if the criteria specified in Section 3.3 are satisfied. The *billing rate* is simply $1.5 \times \textit{estimated cost}$, leading to the following offer price:

$$\textit{actual bid} = (1.5 \times \textit{estimated cost}) \times \textit{load average}$$

$\textit{load average} = 1$ for the duration of the experiment, reflecting the fact that the system is lightly loaded. The difference in the bids offered by each bidder is therefore solely due to data placement (e.g., some bidders need to subcontract remote scans).

Fragments: A broker who subcontracts for remote scans also considers buying the fragment instead of paying for the scan. The fragment value discussed in Section 4.1 is set to $\frac{2 \times \textit{scan cost}}{\textit{load average}}$; this, combined with the fact that eviction is never necessary, means that a site will consider selling a fragment whenever

$$\textit{offer price} > \frac{2 \times \textit{scan cost}}{\textit{load average}}$$

A broker decides whether to try to buy a fragment or to pay for the remote scan according to the following rule:

```
on (salePrice(frag) <= moneySpentForScan(frag))
   do acquire(frag)
```

In other words, the broker tries to acquire a fragment when the amount of money spent scanning the fragment in previous queries is greater than or equal to the price for buying the fragment. As discussed in Section 4.1, each broker keeps a hot-list of remote fragments used in previous queries with their associated scan costs. This rule will cause data to move closer to the query when executed frequently.

This simple economy is not entirely realistic. Consider the pricing of selling a fragment as shown above. If *load average* increases, the sale price of the fragment decreases. This has the desirable effect of hastening the sale of fragments to off-load a busy site. However, it tends to cause the sale of hot fragments as well. An effective Mariposa economy will consist of more rules and a more sophisticated pricing scheme than that with which we are currently experimenting.

We now present the performance and behavior of Mariposa using the simple economy described above and the WAN environment shown in Table 4. Our experiments show how Mariposa adapts to the environment through the bidding process under the economy and the rules described above.

A traditional query optimizer will use a fixed query processing strategy. Assuming that sites are uniform in their query processing capacity, the optimizer will ultimately differentiate plans based on movement of data. That is, it will tend to choose plans that minimize the amount of base table and intermediate result data transmitted over the network. As a result, a traditional optimizer will construct the following plan:

(1) Move R2 from Berkeley to Santa Barbara. Perform R1 join R2 at Santa Barbara.

(2) Move the answer to San Diego. Perform the second join at San Diego.

(3) Move the final answer to Berkeley.

		steps					
		1	2	3	4	5	6
elapsed time (s)	brokering	13.06	12.78	18.81	13.97	8.9	10.06
	total	449.30	477.74	403.61	428.82	394.3	384.04
location of (site)	R1	1	1	1	1	3	3
	R2	2	2	1	1	3	3
	R3	3	3	3	3	3	3
revenue (per query)	site 1	97.6	97.6	95.5	97.2	102.3	0.0
	site 2	2.7	2.7	3.5	1.9	1.9	1.9
	site 3	177.9	177.9	177.9	177.9	165.3	267.7

Table 7. Execution times, data placement and revenue at each site.

This plan causes 6.5 MB of data to be moved (1 MB in step 1, 1 MB in step 2, and 4.5 MB in step 3). If the same query is executed repeatedly under identical load conditions, then the same plan will be generated each time, resulting in identical costs.

By contrast, the simple Mariposa economy can adjust the assignment of queries and fragments to reflect the current workload. Even though the Mariposa optimizer will pick the same join order as the traditional optimizer, the broker can change its query processing strategy because it acquires bids for the two joins among the 3 sites. Examination of Table 7 reveals the performance improvements resulting from dynamic movement of objects. It shows the elapsed time, location of data and revenue generated at each site by running the 3-way join query described in Section 6.1 repeatedly from site 2 (Berkeley).

At the first step of the experiment, Santa Barbara is the winner of the first join. The price of scanning the smaller table, R2, remotely from Santa Barbara is less than that of scanning R1 remotely from Berkeley; as a result, Santa Barbara offers a lower bid. Similarly, San Diego is the winner of the second join. Hence, for the first two steps, the execution plan resulting from the bidding is identical to the one obtained by a traditional distributed query optimizer.

However, subsequent steps show that Mariposa can generate better plans than a traditional optimizer by migrating fragments when necessary. For instance, R2 is moved to Santa Barbara in step 3 of the experiment and subsequent joins of R1 and R2 can be performed locally. This eliminates the need to move 1 MB of data. Similarly, R1 and R2 are moved to San Diego at step 5 so that the joins can be performed locally[1]. The cost of moving the tables can be amortized over repeated execution of queries that require the same data.

The experimental results vary considerably because of the wide variance in Internet network latency. Table 7 shows a set of results which best illustrate the beneficial effects of the economic model.

7. RELATED WORK

Currently, there are only a few systems documented in the literature that incorporate microeconomic approaches to resource sharing problems. [HUBE88] contains a collection of articles that cover the underlying principles and explore the behavior of those systems.

[MILL88] uses the term “Agoric Systems” for software systems deploying market mechanisms for resource allocation among independent objects. The data-type agents proposed in that article are comparable to our brokers. They mediate between consumer and supplier objects, helping to find the current best price and supplier for a service. As an extension, agents have a “reputation” and their services are brokered by an agent-selection agent. This is analogous to the notion of a quality-of-service of name servers, which also offer their services to brokers.

---

[1] Note that the total elapsed time does not include the time to move the fragments. It takes 82 seconds to move R2 to site 1 at step 3 and 820 seconds to move R1 and R3 to site 3 at step 5.

[KURO89] presents a solution to the file allocation problem that makes use of microeconomic principles, but is based on a cooperative, not competitive, environment. The agents in this economy exchange fragments in order to minimize the cumulative system-wide access costs for all incoming requests. This is achieved by having the sites voluntarily cede fragments or portions thereof to other sites if it lowers access costs. In this model, all sites cooperate to achieve a global optimum instead of selfishly competing for resources to maximize their own utility.

[MALO88] describes the implementation of a process migration facility for a pool of workstations connected through a LAN. In this system, a client broadcasts a request for bids that includes a task description. The servers willing to process that task return an estimated completion time and the client picks the best bid. The time estimate is computed on the basis of processor speed, current system load, a normalized runtime of the task and the number and length of files to be loaded. The latter two parameters are supplied by the task description. No prices are charged for processing services and there is no provision for a shortcut to the bidding process by mechanisms like posting server characteristics or advertisements of servers.

Another distributed process scheduling system is presented in [WALD92]. Here, CPU time on remote machines is auctioned off by the processing sites and applications hand in bids for time slices. This is is contrast to our system, where processing sites make bids for servicing requests. There are different types of auctions and computations are aborted if their funding is depleted. An application is structured into manager and worker modules. The worker modules perform the application processing and several of them can execute in parallel. The managers are responsible for funding their workers and divide the available funds between them in an application-specific way. To adjust the degree of parallelism to the availability of idle CPUs, the manager changes the funding of individual workers.

Wellman offers a simulation of multicommodity flow in [WELL93] that is quite close to our bidding model, but with a bid resolution model that converges with multiple rounds of messages. His clearinghouses violate our constraint against single points of failure. Mariposa name service can be thought of as clearinghouses with only a partial list of possible suppliers. His optimality results are clearly invalidated by the possible exclusion of optimal bidders. This suggests the importance of high-quality name service, to ensure that the winning bidders are usually solicited for bids.

A model similar to ours is proposed in [FERG93], where fragments can be moved and replicated between the nodes of a network of computers, although they are not allowed to be split or coalesced. Transactions, consisting of simple read/write requests for fragments, are given a budget when entering the system. Accesses to fragments are purchased from the sites offering them at the desired price/quality ratio. Sites are trying to maximize their revenue and therefore lease fragments or their copies if the access history for that fragment suggests that this will be profitable. Unlike our model, there is no bidding process for either service purchase or fragment lease. The relevant prices are published at every site in catalogs that can be updated at any time to reflect current demand and system load. The network distance to the site offering the fragment access service is included in the price quote to give a quality-of-service indication. A major difference between this model and ours is that every site needs to have perfect information about the prices of fragment accesses at every other site, requiring global updates of pricing information. Also, it is assumed that a name service, which has perfect information about all the fragments in the network, is available at every site, again requiring global synchronization. The name service is provided at no cost and hence excluded from the economy. We expect that global updates of metadata will suffer from a scalability problem, sacrificing the advantages of the decentralized nature of microeconomic decisions.

When computer centers were the main source of computing power, several authors studied the economics of such centers' services. The work focussed on the cost of the services, the required scale of the center given user needs, the cost of user delays, and the pricing structure. Several results are reported in the literature, in both computer and management sciences. In particular, [MEND85] proposes a microeconomic model for studies of queueing effects of popular pricing policies, typically not considering the delays. The model shows that when delay cost is taken into account, a low utilization ratio of the center is often optimal. The model is refined in [DEWA90]. The authors assume a nonlinear delay cost structure, and present necessary and sufficient conditions for the optimality of pricing rules that charges out service resources at their marginal capacity cost. Although these and similar results were intended for human decision making, many apply to the Mariposa context as well.

On the other hand, [MEND86] proposes a methodology for trading off the cost of incomplete information against data-related costs, and for constructing minimum-cost answers to a variety of query types. These results can be useful in the Mariposa context. Users and their brokers will indeed often face a compromise between complete but costly data and processing and a cheaper, but incomplete and partial one.

8. CONCLUSIONS

We present a distributed microeconomic approach for managing query execution and storage management. The difficulty in scheduling distributed actions in a large system stems from the combinatorially large number of possible choices for each action, the expense of global synchronization, and the requirement of supporting systems with heterogeneous capabilities. Complexity is further increased by the presence of a rapidly changing environment, including time-varying load levels for each site and the possibility of sites entering and leaving the system.

The economic model is well-studied and can reduce the scheduling complexity of distributed interactions because it does not seek globally optimal solutions. Instead, the forces of the market provide an "invisible hand" to guide reasonably equitable trading of resources.

We further demonstrated the power and flexibility of Mariposa through experiments running over a wide-area network. Initial results confirm our belief that the bidding protocol is not unduly expensive and that the bidding process results in execution plans that can adapt to the environment (such as unbalanced workload and poor data placement) in a flexible manner. We are implementing more sophisticated features and plan a general release for the end of the year.

ACKNOWLEDGEMENTS

We would like to thank Jim Frew and Darla Sharp of the Institute for Computational Earth System Science at the University of California, Santa Barbara, and Joseph Pasquale and Eric Anderson of the Department of Computer Science and Engineering of the University of California, San Diego for providing a home for the remote Mariposa sites and their assistance in the initial setup.

Mariposa has been designed and implemented by a team of students, faculty and staff that includes the authors as well as Robert Devine, Marcel Kornacker, Witold Litwin, Michael Olson, Robert Patrick and Rex Winterbottom. The presentation and ideas in this paper have been greatly improved by the suggestions and critiques provided by Sunita Sarawagi and Allison Woodruff.

REFERENCES

[BANE91] Banerjea, A. and Mah, B. A., "The Real-Time Channel Administration Protocol," *Proc. 2nd Int. Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991.

[BERN81] Bernstein, P. A., Goodman, N., Wong, E., Reeve, C. L. and Rothnie, J., "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Sys. 6*, 4, Dec. 1981, pp. 602-625.

[BITT83] Bitton, D., DeWitt, D. J. and Turbyfill, C., "Benchmarking Data Base Systems: A Systematic Approach," *Proc. 9th Int. Conf. on Very Large Data Bases*, Florence, Italy, Nov. 1983.

[CHER89] Cheriton, D. and Mann T. P., "Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," *ACM Trans. on Comp. Sys. 7*, 2, May 1989, pp. 147-183.

[COPE88] Copeland, G., Alexander, W., Boughter, E. and Keller, T., "Data Placement in Bubba," *Proc. 1988 ACM-SIGMOD Conf. on Management of Data*, Chicago, IL, June 1988, pp. 99-108.

[DEWA90] Dewan S., and Mendelson, H., "User Delay Costs and Internal Pricing for a Service Facility," *Management Sci. 36*, 12, Dec. 1990, pp. 1502-1517.

[FERG93] Ferguson, D., Nikolaou, C. and Yemini, Y., "An Economy for Managing Replicated Data in Autonomous Decentralized Systems," *Proc. Int. Symp. on Autonomous Decentralized Sys. (ISADS 93)*, Kawasaki, Japan, Mar. 1993, pp. 367-375.

[HUBE88] Huberman, B. A. (ed.), *The Ecology of Computation,* North-Holland, 1988.

[KURO89] Kurose, J. and Simha, R., "A Microeconomic Approach to Optimal Resource Allocation in Distributed Computer Systems," *IEEE Trans. on Computers 38*, 5, May 1989, pp. 705-717.

[LITW82] Litwin, W. *et al.*, "SIRIUS System for Distributed Data Management," in *Distributed Data Bases*, H. J. Schneider (ed.), North-Holland, Amsterdam, The Netherlands, 1982.

[LOHM86] Mackert, L. F. and Lohman, G. M., "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proc. 12th Int. Conf. on Very Large Data Bases*, Kyoto, Japan, Aug. 1986, pp. 149-159.

[MALO88] Malone, T. W., Fikes, R. E., Grant, K. R. and Howard, M. T., "Enterprise: A Market-like Task Scheduler for Distributed Computing Environments," in [HUBE88].

[MEND85] Mendelson, H. "Pricing Computer Services: Queueing Effects," *Comm. of the ACM 28*, 3, Mar. 1985, pp. 312-321.

[MEND86] Mendelson, H. and Saharia, A. N., "Incomplete Information Costs and Database Design," *ACM Trans. on Database Sys. 11*, 2, June 1986, pp. 159-85

[MILL88] Miller, M. S. and Drexler, K. E., "Markets and Computation: Agoric Open Systems," in [HUBE88].

[OUST94] Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, MA, 1994.

[SAH94a] Sah, A., Blow, J. and Dennis, B., "An Introduction to the Rush Language," *Proc. Tcl'94 Workshop*, New Orleans, LA, June 1994, pp. 105-116.

[SAH94b] Sah, A. and Blow, J., "A New Architecture for the Implementation of Scripting Languages," *Proc. USENIX Symp. on Very High Level Languages*, Santa Fe, NM, Oct. 1994, pp 21-38.

[SELI79] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G., "Access Path Selection in a Relational Database Management System," *Proc. 1979 ACM-SIGMOD Conf. on Management of Data*, Boston, MA, June 1979.

[SIDE95] Sidell, J., Aoki, P. M., Barr, S., Sah, A., Staelin, C., Stonebraker, M., and Yu, A., "Data Replication in Mariposa," Sequoia 2000 Technical Report 95-60, University of California, Berkeley, CA, June 1995.

[STON86] Stonebraker, M., "The Design and Implementation of Distributed INGRES," in *The INGRES Papers*, M. Stonebraker (ed.), Addison-Wesley, Reading, MA, 1986.

[STON91a] Stonebraker, M. and Kemnitz, G., "The POSTGRES Next-Generation Database Management System," *Comm. of the ACM 34*, 10, Oct. 1991, pp. 78-92.

[STON91b] Stonebraker, M., "An Overview of the Sequoia 2000 Project," Sequoia 2000 Technical Report 91/5, University of California, Berkeley, CA, Dec. 1991.

[STON94a] Stonebraker, M., Aoki, P. M., Devine, R., Litwin, W. and Olson, M., "Mariposa: A New Architecture for Distributed Data," *Proc. 10th Int. Conf. on Data Engineering*, Houston, TX, Feb. 1994, pp. 54-65.

[STON94b] Stonebraker, M., Devine, R., Kornacker, M., Litwin, W., Pfeffer, A., Sah, A. and Staelin, C. "An Economic Paradigm for Query Processing and Data Migration in Mariposa," *Proc. 3rd Int. Conf. on Parallel and Distributed Information Sys.*, Austin, TX, Sep. 1994, pp. 58-67.

[WALD92] Waldspurger, C. A., Hogg, T., Huberman, B., Kephart, J. and Stornetta, S., "Spawn: A Distributed Computational Ecology," *IEEE Trans. on Software Engineering 18*, 2, Feb. 1992.

[WELL93] Wellman, M. P. "A Market-Oriented Programming Environment and Its Applications to Distributed Multicommodity Flow Problems," *Journal of AI Research 1*, 1, Aug. 1993.

[WILL81] Williams, R., Daniels, D., Haas, L., Lapis, G., Lindsay, B., Ng, P., Obermarck, R., Selinger, P., Walker, A., Wilms, P. and Yost, R., "R*: An Overview of the Architecture," IBM Research Report RJ3325, IBM Research Laboratory, San Jose, CA, Dec. 1981.

[ZHAN92] Zhang, H. and Fisher, T. "Preliminary Measurement of the RMTP/RTIP," *Proc. Third International Workshop on Network and Operating System Support for Digital Audio and Video,* San Diego, CA, Nov. 1992.