AN ECONOMIC PARADIGM FOR QUERY PROCESSING AND DATA MIGRATION IN MARIPOSA

Michael Stonebraker, Robert Devine, Marcel Kornacker, Witold Litwin†, Avi Pfeffer, Adam Sah, and Carl Staelin†

Computer Science Div., Dept. of EECS
University of California
Berkeley, California 94720

Abstract

In this paper we explore query execution and storage management issues for Mariposa, a distributed data base system under construction at Berkeley. Because of the extreme complexity of both issues, we have adopted an underlying economic paradigm for both problems. Hence, queries receive a budget which they spend to obtain their answers, and each processing site attempts to maximize income by buying and selling storage objects and processing queries for locally stored objects. This paper presents the protocols which underlie this economic system.

1. INTRODUCTION

In [STON94] we presented the design of a new distributed database and storage system, called **Mariposa.** This system combines the best features of traditional distributed database systems, object-oriented DBMSs, tertiary memory file systems and distributed file systems. Moreover, in certain areas it alleviates common disadvantages of previous distributed storage systems.

The goals of Mariposa are eight-fold:

(1) **Support a very large number of sites.** Mariposa must be capable of dealing with several hundred **sites** (logical hosts) in a co-operating environment. For example, the Sequoia 2000 project [STON91, DOZI92] has around 200 sites with varying data storage needs and capabilities, mostly on the desktops of participating scientists. Other distributed databases may be substantially larger. For example, a group of cooperating retailers might want to share sales data. In the design of Mariposa, we consider the possibility of distributed databases with as many as 10,000 sites. The problems of data location, information discovery and naming issues must be dealt with in a scalable manner.

---

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

This research was sponsored by the National Science Foundation under grant IRI-9107455, the Defense Advanced Research Projects Agency under contract DABT63-92-C-0007, and the Army Research Office under grant DAAL03-91-G-0183.

(2) **Support data mobility.** Previous distributed database systems (e.g., [WILL81, BERN81, LITW82, STON86]) and distributed storage managers (e.g., [HOWA88]) have all assumed that each storage object had a fixed **home** to which it is returned upon system quiescence. Changing the home of an an object is a heavyweight operation that entails, for example, destroying and recreating all the indexes for that object.

In Mariposa, we expect data objects, which we call **fragments**, to move freely between sites in a computer network in order to optimize the location of an object with respect to current access requirements. Fragments are collections of records that belong to a common DBMS class, using the object model of the POSTGRES DBMS [STON91].

(3) **No differentiation between distributed storage and deep storage.** It is clear that storage hierarchies will be used to manage very large databases in the future. Hence, a storage manager must move data objects from tertiary memory to disk to main memory. In Mariposa, we insist that such movement be conceptually the same as moving objects between sites in a computer network. This will greatly simplify system software, but it will result in one Mariposa **logical** site per storage device, thereby increasing the number of sites which Mariposa must manage.

Also, since fragments are the object which moves between sites, it must be possible to adjust the size of a fragment by **splitting** it if it is too large or by **coalescing** it with another fragment of the same class if it is too small. The desirable fragment size will generally be storage device specific. For example, fragments which typically live on disk will be much smaller than fragments which typically reside on tertiary memory.

(4) **No global synchronization.** It must be possible for a site to create or delete an object or for two sites to agree to move an object from one to the other without notifying anybody. In addition, a site may decide to split or coalesce fragments without external notification. Therefore, any information about (e.g.) the location of an object may be out of date. As a result, Mariposa must base optimization decisions on perhaps **stale** data and the query executor must recover from inaccurate location information.

(5) **Support for moving the query to the data or the data to the query.** Traditional distributed database systems operate by moving the query from a client site to the site where the object resides, and then moving the result of the query back to the client [EPST78, LOHM86]. (Temporary copies of an object may be created and moved during query processing, but only the database administrator can change where an object resides.) This implements a "move the query to the data" processing scenario. Alternately, distributed file systems and object-oriented database systems move the data a storage block at a time from a server to a client. As such, they implement a "move the data to the query" processing scenario. If there is high locality of reference (as in [CATT92]) then the latter policy is appropriate because the movement cost can be amortized over several subsequent interactions. On the other hand, sending the query to the data is appropriate when low locality is observed. In Mariposa, we insist on supporting both tactics, and believe that the choice should be made at the discretion of the query optimizer.

(6) **Flexible support for copy management.** When an object-oriented database system moves data from a server to a client, it makes a redundant **copy** of the affected storage object. This copy lives in the client

cache until it is no longer worthy, and then any updates to the object are reflected back to the server. As a result, the caching of objects in client memory yields **transient** copies of storage objects. Alternately, traditional distributed database systems implemented (or at least specified) support for permanent copies of database relations [WILL81, BERN83, ELAB85]. Our goal in Mariposa is to support both transient and permanent copies of storage fragments within a single framework.

(7) **Support autonomous site decisions.** In a very large network, it is unreasonable to assume that any central entity has control over policy decisions at the local sites. Hence, sites must be **locally autonomous** and able to implement any local policies they please. This will include, for example, the possibility that a site will refuse to process a query on behalf of another site and will refuse to accept an object which a remote site wishes to evict from its storage. This policy is also the only appropriate one in **heterogeneous** distributed DBMSs, where foreign software may be running on each of the local sites. In this case, no assumptions can be made about its behavior.

(8) **Mariposa policy decisions must be easily changeable.** One Mariposa environment might want to implement an LRU storage management policy for deciding which fragments to push from disk out to tertiary memory. A second site might want a totally different policy. It must be possible in Mariposa to easily accommodate such diversity. We expect policies to vary according to local conditions and our own experimental purposes.

To support this degree of flexibility, the Mariposa storage manager is **rule-driven**, i.e., it accepts rules of the form: *on* **event** *do* **action**. Events are predicates in a high performance, high level language we are developing, while actions are statements in the same language. Using this rule engine, we plan to encode solutions to the following issues:

- when to move a fragment between sites
- when to make a copy of a fragment at a site
- when to split a fragment
- when to coalesce two fragments
- where to process any node of a query plan
- where to find fragments in the network

1.1. Resource Management with Microeconomic Rules

To deal with the extreme complexity of these issues, the Mariposa team has elected to reformulate all issues relating to shared resources (query optimization and processing, storage management and naming services) into a microeconomic framework. There are several advantages to this approach over traditional solutions to resource management. First, there is no need for a central coordinator, because in an economy, every agent makes individual decisions, selfishly trying to maximize its utility. In other words, the decision process is inherently decentralized, which is a prerequisite for achieving scalability and avoiding a single point of failure. Second, prices in a market system fluctuate in accordance with the demand and supply of resources, allowing the system to dynamically adapt to resource contention. Third, everything can be traded in a computer economy, including CPU cycles, disk capacity and I/O bandwidth, making it possible to integrate queries, storage managers and name servers into a single market-based economy. The uniform treatment of these subsystems will simplify resource management algorithms. In addition, this will result in an efficient allocation of every available resource.

Using the economic paradigm, a query receives a **budget** in an artificial currency. The goal of the query processing system is to **solve** the query within the budget allotted, by **contracting** with various processing sites to perform portions of the query. Lastly, each processing site makes storage decisions to buy and sell fragments and copies of fragments, based on optimizing the revenue it collects. Our model is similar to [FERG93, WALD92, MALO88] which take similar economic approaches to other computer resource allocation problems.

In the next section, we describe the three kinds of entities in our economic system. Section 3 develops the bidding process by which a broker contracts for service with processing sites, the mechanisms to make the bidding system efficient, and demonstrates how our economic model applies to storage management. Section 4 details the pricing effect on fragmentation. Section 5 describes how naming and name service work in Mariposa. Previous work on using the economic model in computing is examined in Section 6.

2. DISTRIBUTED ENTITIES

In the Mariposa economic system, there are three kinds of entities: **clients**, **brokers** and **servers**. The entities, as shown in Figure 1, can reside at the same site or may be distributed across multiple sites. This section defines the roles that each entity plays in the Mariposa economy. In the process of defining each entity, we also give an overview of how query processing works in an economic framework. The next section will explain this framework in more detail.

Clients. Queries are submitted by user applications at a **client site**. Each query starts with a budget, $B(t)$, which pays for executing the query; query budgets form the basis for the Mariposa economy. Once a budget has been assigned (through administrative means not discussed here), the client software hands the query to a broker.

Brokers. The **broker**'s job is to get the query performed on the behalf of the client. A central goal of this paper is to describe how the broker expends the client's budget in a way that balances resource usage with query response time.

As shown in Figure 1, the broker consists of a *query preparation* module and a *bid manager* module that operate under the control of a *rule engine*. The query preparation module parses the incoming query, performing any necessary checking of names or authorization, and then prepares a **location insensitive** query processing plan. The bid manager coordinates the distributed execution of the query plan.

In order to parse the query, the query preparation module first requests **metadata** for each class referenced in the query from a set of **name servers**. This metadata contains the information usually required

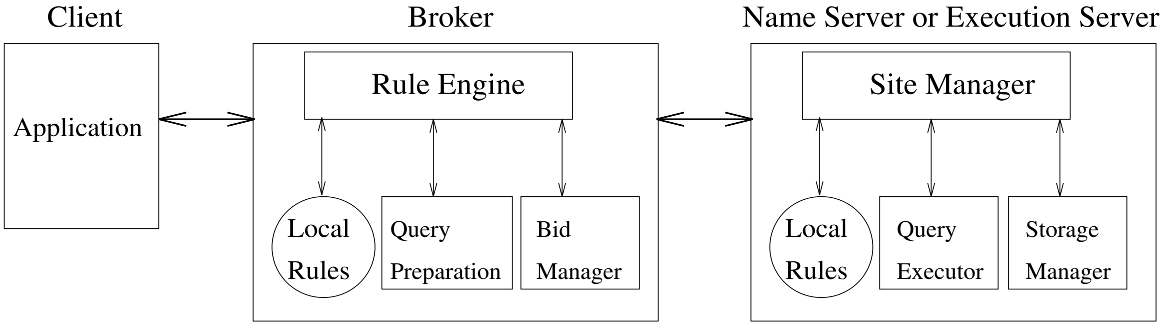

Figure 1. Mariposa entities.

for query optimization, such as the name and type of each attribute in the class and any relevant statistics. It also contains the location of each fragment in the class. We do not guarantee that this information, particularly fragment location, will be up-to-date. Metadata is itself part of the economy and has a price; the parser's 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 rules defined to prioritize these factors.

After successful parsing, the broker prepares a query execution plan. This is a two-step process. First, a conventional query optimizer along the lines of [SELI79] generates a **single site** query execution plan by assuming that all the fragments are merged together and reside at a single server site. Second, a plan fragmentation module uses the metadata to decompose the single site plan into a **fragmented query plan**, in which each restriction node of a single site plan is decomposed into K subqueries, one per fragment in the referenced class. This parallelizes the single site plan produced from the first step. The details of this fragmentation process are described in [STON94].

Finally, the broker's bid manager attempts to solve the resulting collection of subqueries, $Q_1, \ldots, Q_K$, by finding a processing site for each one such that the summation of the subquery costs of C and a total delay of T fit the budget for the entire query. If sites cannot be found to solve the query within the specified budget, it will be aborted. Locally defined rules may affect how the subqueries are assigned to sites.

Decomposing query plans in the manner just described greatly reduces optimizer complexity. Signs that the resulting plans may not be significantly suboptimal appear in [HONG91], where a similar decomposition is studied. Decomposing the plan before distributing it also makes it easier to assign portions of the budget to subqueries.

Servers. **Server sites** provide a processor with varying amounts of persistent storage. Individual server sites **bid** on individual subqueries in a fashion to be described in Section 3. Each server responds to queries issued by a broker for data or metadata. Server sites can join the economy, by advertising their presence, bidding on queries and buying objects. They can also leave the economy by selling all their data and ceasing to bid.

Storage management, the second focus of the Mariposa economic model, is directed by each server site in response to events spawned by executing client's queries and by interaction with other servers.

3. THE BIDDING PROCESS

Mariposa uses an economic bidding process to regulate the storage management as well as the execution of queries. Using a single model for computation and storage simplifies the construction of distributed systems. In this section we describe how to select the bid price and how to find servers that are likely bidders.

Each query, Q, has a **budget,** $B(t)$, which can be used to solve the query. The budget is a decreasing function of time, which represents the value that the user gives to the answer to his query at a particular time, t. Hence, a constant function represents a willingness to pay the same amount of money for a slow answer as for a quick one, i.e., the user does not value quick response. A steeply declining function indicates the contrary. Cumulative user budgets are controlled by administrative means that are beyond the scope of this paper.

The broker handling a query, Q, receives a query plan containing a collection of subqueries, $Q1, \ldots, Qn$, and $B(t)$ which specifies the most amount of money the client is willing to pay for a given service time. Each subquery is a one-variable restriction on a fragment, F, of a class, or a join between two fragments of two classes. The broker tries to solve each subquery, Q_i, using either an **expensive** protocol or a **cheap** protocol. In the remainder of this section we discuss these two protocols and the conditions under which each is used.

3.1. The Expensive Bidding Protocol

Using the expensive protocol, the broker conducts a bidding process for each subquery by sending the subquery (or a data structure representing it) to a collection of **possible bidders.** These bidders can be identified in several different ways, as we will discuss in the next section. Once the broker has received bids from the possible servers, it must choose a winning collection of bids.

Each **bid** consists of a triple: (C_i, D_i, E_i) which is a proposal to solve the subquery, Q_i, for a cost, C_i, within a delay, D_i, after receipt of the subquery, noting the fact that the bid is only valid until a specified expiration date, E_i. The way that a site arrives at a bid will be discussed in a later section.

In order to bid on a subquery, Q_i, a site must possess the fragment(s) referenced by the subquery or a copy of them. If the site does not have each referenced fragment, then it must be willing to **buy** the missing ones. Buying a fragment entails contacting the current **owner** of the fragment, and either:

(1) buying the fragment from the owner, in which case there continues to be a single copy of the fragment, or

(2) purchasing a **copy** of the fragment, in which case the owner remains the same, but there is an additional copy.

Setting the price of fragments and copies is the subject of a later section.

The broker receives a collection of zero or more bids for each subquery. If there is no bid for some query, then the broker must either contact additional possible bidders, agree to perform the subquery itself, or notify the user that the query cannot be run. If there is one or more bids for each subquery, then the broker must ascertain if the entire query can be processed within the budget allocated, and if so, must choose the winning bids.

The broker must choose a collection of bids with aggregate cost C and aggregate delay D such that the aggregate cost is less than or equal to the cost requirement $B(D)$. It is possible that several collections of bids may meet the minimum price/performance 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: $difference = B(D) - C$. Note that this can have a negative value, if the cost is above the bid curve.

The goal of the broker is to choose the collection of bids which solves the query with a maximum value of *difference*. However, the broker's job is complicated by the parallelism possible in the query plan. A given subquery can be run for each fragment of a class in parallel. Also, a given join can be run in parallel for each of the pairs of fragments, one from each class. Lastly, certain nodes in the query plan can be **pipelined** into subsequent nodes, and hence, there is no need to **synchronize** between the nodes. In other cases, a subsequent node cannot be started until the last of the parallel subqueries has finished from the previous step. In this case the delay is determined by the slowest of the parallel tasks, and lowering the delay of any other task will not affect the total response time.

To model this possible parallelism, we assume that the query can be decomposed into disjoint processing steps. All the subqueries in each processing step are processed in parallel, and a processing step cannot begin until the previous one has been completed. Rather than consider bids for individual subqueries, we consider collections of bids for each processing step.

Given such a collection, the estimated delay to process the entire collection 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, there is an optimal collection of bids: the one with the cheapest cost. This is formed by choosing the least expensive bid for each subquery that can be processed within the given delay. By "coalescing" parallel bid collections and considering them as a single (aggregate) bid, the broker may reduce the bid acceptance problem to a simpler problem of choosing one bid (from among a set of aggregated bids) for each sequential step.

It is obviously feasible to perform an exhaustive search and consider all possible viable collections of bids. For example, if there are 10 processing stages and 3 viable collections for each one, then the broker can evaluate each of the 3^{10} bid possibilities, and choose the one with the maximum difference. For all but the simplest queries referencing classes with a minimal number of fragments, this strategy 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 two algorithms that determine how to do this. Although they cannot be shown to be optimal, we believe in practice they will demonstrate good results. A detailed evaluation and comparison against more complex algorithms is planned to test this hypothesis.

The first 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.

Begin by considering the bid collection with the smallest delay for each processing step. Compute the cost gradient for each unused collection. A trial solution with total cost C and total cost D is generated. Now, consider the processing step with the unused collection with the maximum cost gradient. If this collection 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. Recalculate all the cost gradients for the processing step involved in the substitution, and continue making substitutions until there are none which increase the *difference*.

The second algorithm takes the budget of the entire query and the structure of the query plan to produce a **subbudget** for each subquery. The subbudget for a subquery q is a scaled down version of the budget function for the entire query: $B_q(t) = C_q \times B(t/D_q)$ where C_q and D_q represent the fraction of the cost and time resources allocated to the subquery.

After bids have been received, a set of viable collections of bids is produced for each processing stage as described above. The various processing stages are then considered independently from each other. For every collection of bids, we compute the *difference* of each bid from the subbudget function for its subquery, and then add these values together to obtain a *difference* value for the collection. The collection with the greatest *difference* value is chosen for each processing stage, even if it happens to be negative. It is possible that the entire query can be solved within budget even if a certain processing stage cannot.

The values of C_q and D_q are produced as follows. Each subquery comes from the optimizer annotated with an estimate of total resource R needed to compute that subquery. All subqueries in a processing stage are allocated the same fraction of the total time, proportion to the maximum value of R for that stage. Every subquery is initially allocated a fraction of the total cost in proportion with its value of R.

However, since some subqueries are allocated more time than they need (because they run in parallel with slower subqueries), the fraction of the cost allotted to them can be scaled down accordingly.

The budgeting algorithms select a set of bids with total cost $C*$ and total delay $D*$. If the resulting solution is feasible, i.e., $C* < B(D*)$ then the broker accepts the winning bids, and they become **contracts,** which the bidder must honor. If $(C*, D*)$ is not feasible, then the broker has failed to find an acceptable solution, and a message should be sent to the user rejecting the query.

Every contract has a **penalty clause,** which the contractor must abide by in case, he does not deliver the result of the subquery within the time allotted. The exact form of this penalty is not important in the model.

3.2. The Cheap Bidding Protocol

The expensive bidding process is fundamentally a two-phase protocol. In the first phase, the broker sends out a request for bids, to which processing sites respond. During the second phase, the broker notifies processing sites whether they won or lost the bid. This protocol therefore requires many (expensive) messages. Most queries will not be computationally demanding enough to justify this level of overhead. Hence, there is a need for a cheaper alternative, which should be used the vast majority of the time.

The **cheap** bidding protocol simply sends each subquery to one processing site. This site would be the one thought most likely to win the bidding process, assuming there were one. This site simply 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. Using the cheap protocol, there is some danger of failing to solve the query within the allotted budget. As will be seen in the next section, the broker does not always know the cost and delay that the chosen processing site will bill him for. However, this is the risk which must be taken to get a cheaper protocol.

In the next section we turn to policy mechanisms that will help to make either of the two protocols as efficient as possible.

3.3. Finding Likely Bidders

Using either the expensive or the cheap protocol from the previous section, a broker must be able to effectively identify one (or more) sites who are likely to want to process a subquery. In this section we indicate several mechanisms whereby a broker can obtain the needed information. Our mechanisms can use several popular information dissemination algorithms, including: **yellow pages**, **posted prices**, **advertisements**, **coupons**, and **bulk purchase contracts**. The increasing levels of restriction for the algorithms is shown in Table 1.

The first mechanism is similar to the concept of the phone company **yellow pages**. Specifically, a server can advertise that it offers a specific service using this mechanism. The yellow pages can be implemented as a broadcast facility by which a server alerts all brokers of the capability or it can be a single

	Describe Service	Specifies Price	Has an Expiration	Limits Quantity or Use
yellow pages	X			
posted prices	X	X		
advertisements	X	X	X	
coupons & bulk	X	X	X	X

Table 1. Likely Bidder Dissemination Algorithms.

data base that is queried by brokers as needed. Using this mechanism, a server advertises the fact that it desires transactions which 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 resulting information. The server specific field(s) allows a server to add any other items of information it deems appropriate. We will see a use for this field in the name service discussion in the next section. A server can issue a new yellow pages advertisement at any time without explicitly revoking a previous one. In keeping with the characteristics of current yellow page advertisements, no prices are allowed. A server advertises in the yellow pages style by promulgating the following data structure:

(class-name, server-identifier, date, server-specific field(s))

We now turn to a second facility, which supports posting **current prices.** Here, a server is allowed to post the prices on specific kinds of transactions. This is analogous to a supermarket which posts the prices of specific goods in its window. This construct requires the notion of a **query template**, which is a query with parameters left unspecified, for example:

```
SELECT param-1
FROM EMP
WHERE NAME = param-2
```

A server can post the current price by specifying the data structure:

(query-template, server-identifier, price, delay, server-specific-field(s))

This alerts brokers that the indicated server currently processes queries which fits the template for the indicated price with the specified delay. Of course, the server does not need to guarantee that these terms will be in effect when a broker later tries to make use of the server. In effect, these are current prices and can be changed with no advance notice.

A third mechanism is an **advertisement,** which is the following data structure:

(query-template, server-identifier, price, delay, expiration-date, server-specific-field(s))

An advertisement is the same construct as a current price, differing only by the fact that the server must guarantee the terms until a specified expiration-date. Hence, the server is specifying the current prices and additionally guaranteeing not to raise them until a specified time. Obviously, a server takes some risk when it places an advertisement because substantial demand may be forthcoming as a result of the advertisement, which the server will be unable to meet. If so, it will be overwhelmed and be forced to pay heavy penalties. There is no way to lower demand by raising prices until the expiration-date of the advertisement has passed.

A fourth notion is a somewhat safer alternative and makes use of **coupons.** These are advertisements to process a query matching a template for a specific price until an expiration date. However, they have two characteristics not shared by advertisements. First, the coupon can place a limit on the number of queries that can be accommodated at the given price and delay. This is analogous to a grocery store coupon which says "limit one per customer." The second characteristic is that coupons can limit the brokers to which they apply. This is analogous 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. Consequently, a coupon is an advertisement with the extra fields:

(quantity, broker-list)

The quantity field indicates the number of queries that can be run prior to the expiration-date by a customer before the coupon becomes invalid. The broker-list field indicates who can make use of the coupon. Coupons can be used to advertise an attractive price without subjecting a processing site to the danger of being swamped. However, coupons also have the property that they expire at a specific expiration-date, and therefore cannot be used as the basis for a long term relationship between a broker and a processing site.

A variation of coupons is a mechanism which supports long term relationships is the notion of **bulk purchase contracts.** These can be considered as special coupons where the processing site sets up a contract with a broker to provide a specific quantity of queries to be processed within a specific interval of time. At contract expiration, the broker receives another coupon for the same quantity of queries good for an interval of time of the same length. These **periodic** coupons continue until a specified termination date. In this way, the broker can process a specific sized workload during each time interval. This is analogous to a travel agent which books 10 seats on each sailing of a cruise ship. The broker presumably receives a good price, in exchange for using the server in bulk. We allow bulk purchases to optionally be **guaranteed,** 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 a broker requires a way to solve large numbers of quite similar queries.

Another variation on the coupon-based bulk purchase contracts is load-based bulk contracts. A broker buys the right to have a specific number of queries outstanding at any one time for a given interval of time. The notion is that transaction processing environments tend to have a steady load, so a broker can purchase the right to keep a given load on a server.

A broker will decide potential bidders by using some or all of the above mechanisms. In addition, we also expect a broker to remember sites who have bid successfully for previous queries. Presumably the broker will include such sites in the bidding process, thereby generating a system which learns over time what processing sites are appropriate to which 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.

3.4. 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 resource so as to maximize its revenue income per unit time.

In order for sites to trade fragments, they have to have some means of calculating the (expected) value of the fragment for each site. Consequently, some access history is kept with each fragment so sites may evaluate the past activity of the fragment, and use this information to predict future fragment activity. The access history is useful to foreign sites if and only if the units quantifying past activity are common across all machines in the system.

For each fragment which the site stores, it maintains the **size** of the fragment plus its **revenue history:**

(query, qualifying-records, time-since-last-query, revenue, delay, I/O-used, CPU-used)

Here, the first field is the actual query which was processed, while the second is the number of records which qualified. Field three is the relative time since the previous query in the revenue history and the fourth field encodes the revenue collected. Actual response time is captured in the fifth field while the sixth and seventh are normalized versions of the CPU and I/O resources used, expressed in site-independent units.

To estimate the revenue that a site would receive if it owned a particular fragment, the site must assume that accesses 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 **buy** any fragment(s) referenced by the subquery. 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 size of the fragment to be purchased. To the extent that storage is not full, then lesser (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:

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

In this calculation, the buyer will obtain the value of the new fragment but lose the value of the fragments which it must evict. Moreover, it will **sell** the evicted fragments, and receive some price for them.

The first two items are easy to compute, while the third one is problematic. A plausible assumption is that the buying site can sell the alternate fragments for a price equal to their value to the selling site. If so, the price to bid will be:

$$\textit{offer price} = \textit{value of fragment}$$

However, it is not always prudent to make this assumption, and a more conservative assumption would be to assume that the price obtained for the alternate fragments is zero. In this case, the offer price would be:

$$\textit{offer price} = \textit{value of fragment} - \textit{value of alternate fragments}$$

Notice that the offer price need not be positive.

The potential seller of the fragment performs the following calculation. If it sells the fragment for the offer price, then it receives this value. In addition, it will avoid having to evict a collection of alternate fragments summing in size to the indicated fragment. Hence, it will be willing to sell if:

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

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

In any case, if the inequality is true, then the seller will transfer the fragment to the buyer, who assumes ownership of the fragment. If the inequality is not true, then the buyer might be willing to make a **copy** of the fragment, with ownership remaining with the seller.

If a copy is made, then several economic considerations must take place. First, the buyer only has access to the revenue stream for the owner of the fragment. This will have all the write transactions, but only a fraction of the read operations. If there are $N - 1$ secondary copies, then each of them will have processed some share of the read operations, and this revenue history is only accessible to the buyer if it finds all $N - 1$ secondary copies and obtains their revenue history, an expensive operation indeed. In any case, if the buyer is contemplating adding the Nth secondary copy, then it could plausibly assume that the owner currently has $\frac{1}{N}$th of the read operations. If it makes a copy, it could plausibly assume it will get $\frac{1}{N+1}$ of each of the revenue streams. Based on these two factors the buyer will compute a *copy offer price*.

Second, all update transactions must be directed to the owner of the fragment. Hence, the secondary copies will have to perform updates to their copies but will receive no revenue for their effort. This will impact the price the buyer is willing to pay for a copy because every update will consume extra network overhead. If copies are kept transactionally consistent, then a two-phase commit is required and extra messages are incurred. Even if the copy is kept up to date on a "best effort" basis, then extra messages are required to propagate the changes. Hence, the network manager must be compensated.

Lastly, the buyer must pay a "tax" to the owner to compensate him for the extra trouble of propagating updates onward. Hence, the seller will allow the buyer to make a copy if:

$$\textit{copy offer price} > \textit{update tax} + \textit{network tax}$$

If this inequality is true, then the buyer will make a copy of the seller's fragment, thereby increasing the number of secondary copies from $N - 1$ to N. The selling site can calculate the update tax by examining the revenue history. Moreover, the selling site can estimate the network tax by knowing the revenue history and the copy consistency algorithm.

A buyer can undertake the above methodology for a fragment at any time. it need not have a query in hand which requires the fragment. Hence, the buyer can "prefetch" fragments which it expects will be profitable in the future.

A possible improvement is to lease copies instead of selling them to the sites, as outlined in [FERG93]. The initial lease price is established along the same lines as a fixed offer price. The difference is that the lease is only valid for a particular period of time, the lease period, after which the lease contract has to be renewed. The primary site charges a lease renewal price to all sites holding copies; this renewal price can take, among other things, current system and network load into account. If, for instance, the load at the primary site suddenly increases, the renewal price will also increase, making it unprofitable for some sites to hold a copy. Thus, a fragment lease strategy allows a better dynamic adaptation of the degree of replication to the execution environment.

A site can also unilaterally decide to sell a fragment at any time. An extreme case occurs when the site wishes to "go out of business" and get rid of all its fragments. A site can decide that its storage is **over-full** according to its local rules that define policy and it needs room for importing other fragments at a later time. In this case, the site tries to **evict** the lowest value fragment. If the fragment is a copy of another fragment, then it simply deletes the fragment. Otherwise, it must try to **sell** it to another site.

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.

Each site examines the revenue history and decides how much to bid. Moreover, bids may be negative. The seller considers the highest bid and will **accept** the bid if:

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

Here, 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 received price. If the above inequality is true, then the seller will proceed with the sale to the buyer.

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.

3.5. Setting The Bid Price For Subqueries

When a site receives a subquery and is asked to bid, it must respond with a triple (C, D, E) as noted in an earlier section. Each site maintains a **billing rate** for each fragment, which is the revenue per unit of resources expended which it expects to charge to perform a query. To quote a price for a query, the site simply multiplies its billing rate by the expected resources which will be required to perform the query. However, this bid must be adjusted for two factors, which we now consider.

Each site also maintains for each fragment the revenue collected per unit time. We will call this the **collection rate** for the fragment. Furthermore, it maintains a history of the collection rate, and can calculate the **derivative** of the collection rate. If the derivative is positive then the site should raise its billing rate, while if the derivative is negative it should lower its rate. Specifically, it should perform the following calculation:

$$new\ billing\ rate = old\ billing\ rate \times (1 + W2 \times derivative)$$

Here, $W2$ is a weighting factor which indicates how heavily to weight the change in recent business.

The second consideration is the current load at the site. If the site is over-busy, then it should raise prices; it is is underutilized it should drop its prices. Hence, it should adjust its billing rate for its current load in the following way. Assume that the site keeps track of its current load in some units which it understands and then normalizes the data item so that it ranges between 0 and 1, with 0 indicating idleness and 1 indicating full utilization. Denote this quantity as the current site **load.** The site should adjust its bidding rate as follows:

$$actual\ billing\ rate = billing\ rate \times (W3 \times load)$$

Again, $W3$ is a weighting factor indicating how seriously to take current load in the current bid.

With these adjustments, the site can bid on any query, referencing data it possesses. If the site does not possess all referenced fragments, then it must buy missing ones, and should only bid if it wishes to acquire the missing fragments using the process of the previous section.

To calculate the delay it will promise that it can make an estimate for the resources required to process the query. Under zero load, it can then estimate the elapsed time to perform the query. After adjusting for the current load, it can estimate an expected delay. The number D included in the bid can either be this expected delay or it can be the expected delay times a safety factor.

The expiration date on a bid should be assigned by a site after considering how much risk it is willing to take. An expiration date a long way in the future can be chosen, but the processing site incurs the risk that prices will rise in the interim, and it will be stuck honoring out of date prices. On the other hand, a too early expiration date runs the risk that the broker will not be able to use the bid because of inherent delays in the processing engine.

Since a site keeps its fragments in value order, it should consider declining to bid on queries referencing low value fragments. In this case, the query will have to be processed elsewhere, and another site will have to copy or buy the indicated fragment in order to solve the user query. Hence, this tactic will hasten the sale of low value fragments to somebody else.

Lastly, the site can refuse to process queries for a fragment and can refuse to sell the fragment. In this case, unless a second site is willing to make a copy of the fragment, then "livelock" will result for the fragment. In a system with total local autonomy, there is no way to prevent such an occurrence.

3.6. Spheres of Influence

So far, we have assumed that all sites are in the same "administrative moat," i.e., that each site participating in the economy uses the same currency. Most distributed systems do not obey this homogeneity assumption, and we must extend our economic model to deal with this reality.

As a result, we assume that each site has a collection of other sites, S, which form its **sphere of influence**. A sphere of influence exists for each site, and these sets of sites may be overlapping. We assume that each site executes the calculations in the above sections and considers the queries from sites in its sphere of influence at face value and applies a discount rate to queries from all other sites. This discount rate can range from 0, in which case all queries are considered equally, to 1, in which case the site considers that it receives no value for performing queries from other sites. The discount rate acts as a **tariff** on trades between different spheres. We allow non-symmetric tariffs to be defined. Depending on the discount rate, queries are accepted based on their calculated bid price as described earlier.

4. SPLITTING AND COALESCING FRAGMENTS

Mariposa sites must 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.

One strategy is to simply let market economics determine the sizes of fragments. Consider a fragment, F, which has high revenue. A second site would naturally want to make a copy of F and thereby divert some of the revenue. Facing this revenue loss, the first site might preemptively split the fragment into two pieces, selling one to another site In this way, the remaining smaller fragment becomes less attractive for copying.

On the other hand, if a fragment is too small, then the overhead of processing queries will be high, and economies of scale would result by coalescing it with another fragment in the same class. Hence, there are market pressures in Mariposa which will tend to correct for inappropriate fragment sizes.

If a 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 make use of the metadata returned from the name server to note the current number of fragments in the class, NUM_C. Moreover, if the broker assumes that all fragments are of equal size, then it can guess the expected delay, ED, which will result from the solution of the NUM_C subqueries which will be run. Furthermore, the broker can use the budget function to compute the amount of the expected feasible bid per site, which is:

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

In other words, if the query is feasible, then this is the amount of revenue which can realistically be given to each site. Now the broker can repeat the above collection of calculations for the class being decomposed into any particular number, NUM, of fragments. Lastly, it can calculate the number of fragments, NUM *, which maximizes 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 which 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 previously 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 make 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

Naming is an important component of distributed database and file systems. 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. Mariposa goals of mobile fragments and avoidance of global synchronization require that a more flexible naming service be used. We develop a decentralized naming facility that does not depend on a centralized authority for name registration or binding.

5.1. Names

Three types of names are used in Mariposa. First, **internal names** are the location-dependent names that are used to physically locate the fragment. Because these are low-level names that are defined by the implementation, no more description will be given in this section. Next, **full names** are the completely specified names that uniquely identify an object. A full name can be tied to any object regardless of location. Full names are not user specific and are location transparent so that when a fragment moves, the name does not have to converted. A full name can be used equally well from anywhere; this allows a query to move to a different site but still request the same object.

In contrast, **common names** are names that are sensible to a user. 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 from either parameters directly supplied by the user or from something in the user's environment. There exists an ambiguity in common names because different users can refer to different objects using the same name. Because common names are context dependent, they may even refer to different objects at different times.

One **name space** exists for all sites in a system. It is a single rooted tree of names. All full names are globally unique within the name space however the policy for selecting names is locally defined. So as not to constrain the later growth of the name space from the amalgamation of other name spaces, a non-fixed-root name space as suggested in [LAMP86] can be used to support upwards growth beyond the current root.

Finally, a **name context** is a set of names that are affiliated. This grouping is of names that are expected to share some feature such as they are often used together in an application (i.e., directory) or the names construct a more complex object (i.e., class definition). A programmer can define a name context for global use that everyone can access or a private 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 resources to make them unique such as birth site as 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 locate the name among other sites. If a broker fails to resolve a name using its local cache, it must ask one or more name servers.

In addition to the case of untranslatable names, there is a possibility of ambiguous resolutions when resolving a common name. For example, a common name of "EMP" may in multiple name contexts that a program is using such as "RESEARCH.EMP" and "DEVELOPMENT.EMP". 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 the rules. Some possible policies are "first match," as exemplified by the UNIX shell command search (path), or a policy of "best match" that seeks to choose more intelligently. 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 service responds to metadata queries in the same way as data servers execute regular queries. Consequently, the name service process uses the bidding protocol of Section 3 to interact with a collection of potential bidders. Mariposa expects there to be some number of name servers, and this collection may be dynamic as name servers are added to and subtracted from a Mariposa environment. The broker decides which name server to use based on economic considerations of cost and quality of service. A name server translates a common name into a full name by using a list of possible name contexts that the client passes. The context list can be like a UNIX path or the name server can use any default name contexts as defined with the rule system. These name servers are expected to use the advertising capabilities to find clients for their services.

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

As a result, name servers are differentiated on their **quality of service** regarding their price and the correctness of their information. For example, a name server which 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. We propose that name servers use the *server-specific-field* in the various advertising mechanisms in the previous section to indicate the quality of their answers to queries. 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 which it wishes to deal with.

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 rather than the name server's rate of update.

6. RELATED WORK

So far there are only a few systems documented in the literature which incorporate microeconomic approaches to deal with 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 currently best price and supplier for a needed 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, that also offer their services to brokers.

[KURO89] present 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 competing for resources to selfishly 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 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 convergies with multiple rounds of messages. His clearinghouses violate our constraint against single points of failure; hence, Mariposa name service can be though of as clearinghouses with only a partial list of possible suppliers. His optimality results are clearly invalidated by the possible exclusion of optimal bidders, suggesting 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 to our model 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 is available at every site that has perfect information about all the fragments in the network, 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.

7. CONCLUSIONS

We present a distributed microeconomic approach to deal with 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, expense of global synchronization, and requirement for supporting heterogeneous systems. Complexity is further increased by the presence of a dynamically changing environment, including time varying load levels for each site and the possibility of sites entering and leaving the system.

The economic model is a well studied model that can reduce scheduling complexity of distributed interactions by not seeking perfect globally optimal solutions. Instead, the forces of the market provide an “invisible hand” to guiding reasonably equitable trading of resources.

Mariposa’s economic model differs from the strict assumptions of the classic Walras model, and does so in ways that cannot guarantee pareto optimality (i.e., imperfect name service may exclude some possible contenders from bidding on a plan). Instead of seeking such optimality, we are attempting to model the Mariposa market to incent the selection of reasonably good solutions without incurring high overhead. In cases where our baseline model fails to incent the correct behavior, we are studying ways of applying external economic pressures (i.e., "taxing" certain behaviors) to adjust this model as needed.

At the present time the query preparation module is nearly complete and the Mariposa rule engine is beginning to work. We are now focused on implementing the low level support code, the complete broker and the site manager, and expect to have a functioning initial system by the end of 1994.

REFERENCES

[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.

[BERN83] Bernstein, P. and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 1983 ACM Symp. on Principles of Distributed Computing*, Montreal, Quebec, Aug. 1983.

[CATT92] Cattell, R. G. G. and Skeen, J., "Object Operations Benchmark," *ACM Trans. on Database Sys.* 17(1), Mar. 1992.

[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.

[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, Jun. 1988.

[DOZI92] Dozier, J., "How Sequoia 2000 Addresses Issues in Data and Information Systems for Global Change," Sequoia 2000 Technical Report 92/14, University of California, Berkeley, CA, Aug. 1992.

[ELAB85] El Abbadi, A. *et al.*, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," *Proc. ACM SIGMOD-SIGACT Symp. on Principles of Data Base Systems*, Apr. 1985.

[EPST78] Epstein, R. S., Stonebraker, M. and Wong, E., "Distributed Query Processing in Relational Database Systems," *Proc. 1978 ACM-SIGMOD Conf. on Management of Data*, Austin, TX, May 1978.

[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, 1993.

[HONG91] Hong, W. and Stonebraker, M., "Optimization of Parallel Query Execution Plans in XPRS," *Proc. 1st Int. Conf. on Parallel and Distributed Info. Sys.*, Miami Beach, FL, Dec. 1991.

[HOWA88] Howard, J. H., Kazar, M. L. Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N. and West, M. J., "Scale and Performance in a Distributed File System," *ACM Trans. on Comp. Sys.* 6(1), Feb. 1988.

[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.

[LAMP86] Lampson, B., "Designing a Global Name Service," *Proc. ACM Symp. on Principles of Distributed Computing*, Calgary, Canada, Aug. 1986.

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

[LOHM86] Mackert, L. F. and Lohman, G. M., "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, May 1986.

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

[MILL88] Miller, M. S. and Drexler, K. E., "Markets and Computation: Agoric Open Systems," in *The Ecology of Computation*, B.A. Huberman (ed.), North-Holland, 1988.

[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.

[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.

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

[STON94] 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.

[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.