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

Many new database applications require very large volumes of data. Mariposa is a data base system under construction at Berkeley responding to this need. Mariposa objects can be stored over thousands of autonomous sites and on memory hierarchies with very large capacity. This scale of the system leads to complex query execution and storage management issues, unsolvable in practice with traditional techniques. We propose an economic paradigm as the solution. A query receives a budget which it spends to obtain the answers. Each site attempts to maximize income by buying and selling storage objects, and processing queries for locally stored objects. We present the protocols which underlie the Mariposa economy.

1. INTRODUCTION

[STON94a] presents 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.

The goals of Mariposa are manifold:

(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, and in a scalable manner. We consider the possibility of distributed databases with as many as 10,000 sites (eg. a group of retailers sharing sales data).

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.

† Author's current address: Enterprise Computing Group, Microsoft Corp., 1 Microsoft Way, Redmond, WA 98052.

° Author's current address: Hewlett-Packard Laboratories, P.O. Box 10490, Palo Alto, CA 94303.

(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.** Storage hierarchies will be used to manage very large databases in the future. In Mariposa, we treat movement between the storage hierarchies conceptually as moving objects between sites in a computer network. There is one **logical** Mariposa site per storage device.

(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.

(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]. 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. In Mariposa, we insist on supporting both tactics, the choice should be made by the query optimizer (depending

on the locality of reference).
(6) **Flexible support for copy management.** When an object-oriented database system moves data from a server to a client, it keeps a redundant **copy** of the affected storage object in the client cache, yielding **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 is to support both transient and permanent copies of storage fragments within a single framework.
(7) **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.
(8) **Easily modified policy decisions.** In Mariposa, different sites might want to implement different policies for evicting fragments to tertiary memory. 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.

1.1. Resource Management with Microeconomic Rules

To deal with the 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.

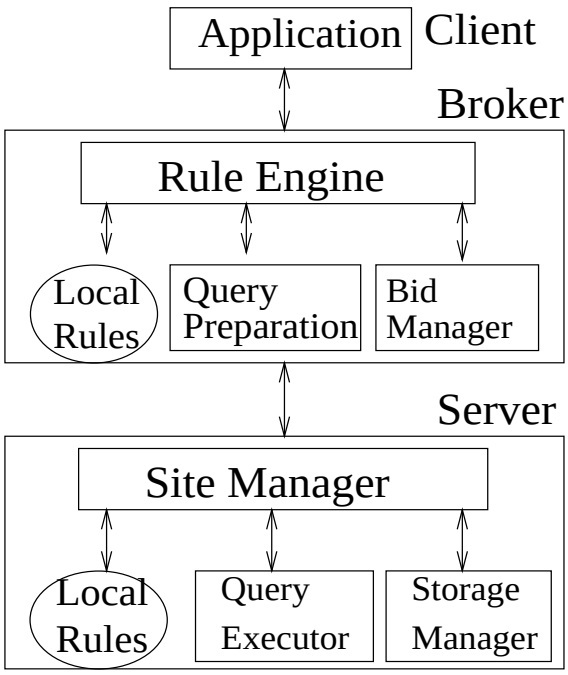

Figure 1. Mariposa entities.

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 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 [STON94a].

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 name service, storage management, and query execution. In general, clients find likely contractors, solicit bids for a given piece of work, and then select the winning bid. In Mariposa, brokers manage the bidding and query execution on behalf of clients, and clients direct brokers using budgets.

Each query, Q, has a **budget,** $B(t)$, which can be used to solve the query. The budget is a non-increasing (possibly non-linear) function of time, which represents the value that 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 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, $Q_1, \ldots, Q_n$, and $B(t)$. 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 **bid** order protocol, or a **purchase order**. If possible, the broker will choose a set of winning bids $B_{i,j}$, such that the aggregate cost C and aggregate delay D is less than the bid curve $B(D)$.

Every contract has a **penalty clause,** which the contractor must abide by if 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.

Using the bid protocol, the broker conducts a bidding process for each subquery by soliciting bids for the subquery (or a data structure representing it) from possible contractors and then selecting a winning bid. 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 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 requires many (expensive) messages. Most queries will not be computationally demanding enough to justify this level of overhead.

The **purchase order** 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 faster protocol.

3.1. Bid Acceptance

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 which make finding the best bid collection difficult: subquery parallelism and the combinatorial search space. The aggregate delay is not necessarily the sum of the delays D_i for each subquery Q_i, since there is often parallelism within the query plan. For example, a subquery can be run for each fragment of a class in parallel, or certain nodes in the query plan can be **pipelined.** Also, the number of possible bid collections grows exponentially with the number of processing steps in the query plan. For example, if there are 10 processing stages and 3 viable bids for each one, then the broker can evaluate each of the 3^{10} bid possibilities.

Given a collection of parallel subqueries, 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 bid collection: 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.

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.

With the full bidding protocol, the broker receives a collection of zero or more bids for each subquery. If there is no bid for some query or no collection of bids meets the client's minimum price 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: $\mathit{difference} = B(D) - C$. Note that this can have a negative value, if the cost is above the bid curve.

For all but the simplest queries referencing classes 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 two heuristic 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 bid. A trial solution with total cost C and total cost D is generated. Now, consider the processing step for the unused bid with the maximum cost gradient. 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. 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. This algorithm is presented in more detail in [STON94b].

3.2. Finding Contractors

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 may process a subquery. There are several mechanisms whereby a broker can obtain the needed information, including: yellow pages, posted prices, advertisements, coupons, and bulk purchase contracts.

Using **yellow pages**. a server advertises that it offers a specific service, such as 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. 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, and server-specific field(s).

A server is allowed to post the prices on specific kinds of transactions, 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 query-template, server-identifier, price, delay, and server-specific-field(s). 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.

Advertisements are similar to current price mechanism, except that the server must guarantee the terms until a specified expiration-date. Obviously, a server takes some risk when it places an advertisement which generates more demand than the server can meet, forcing it to pay heavy penalties.

Coupons are advertisements that include a limit on the number of queries that can be executed under the terms of the advertisement, similar to supermarket coupons. Coupons may also limit the brokers who are eligible to redeem them, 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.

Bulk purchase contracts are renewable coupons which allow a broker to negotiate cheaper prices with a server in exchange for guaranteed (pre-paid) service. This is analogous to a travel agent which books 10 seats on each sailing of a cruise ship. 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 brokers solve large numbers of similar queries.

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.

3.3. 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. Some access history is kept with each fragment so sites may predict future activity based on the history.

For each fragment which the site stores, it maintains the **size** of the fragment plus 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 **buy** any fragment(s) referenced by the subquery, 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 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:

offer price = value of fragment -
value of alternate fragments + 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 latter item is problematic to compute. A plausible assumption is that the 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 received price. Hence, it will be willing to sell if:

offer price > value of fragment -
value of alternate fragments + 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, only read transactions contribute to the revenue collected by a copy since update transactions are always directly to the owner of a fragment. The buyer of a copy has to estimate the copy's expected revenue solely from the owner's revenue history. If there are $N-1$ secondary copies already and the owner currently has $\frac{1}{N}$th of the read operations, the new copy could plausibly assume that it will get $\frac{1}{N+1}$ of each of the read revenue streams, or equivalently, $\frac{N}{N+1}$ of the owner's read revenue stream. The buyer has to compute a *copy offer price* from this value. Second, the copies will have to perform updates but will receive no revenue for their effort. These updates consume extra network resources and the price has to reflect that. Lastly, the buyer must also 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:

copy offer price > update tax + network tax

The selling site can calculate the update tax from the revenue history and the network tax from the revenue history and the copy consistency algorithm.

Sites may sell fragments at any time, for any reason. For example, decommisioning 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 + received price

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.

3.4. 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.

One strategy is to simply let market pressure correct for 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, small fragments have high processing overhead for queries, and economies of scale would result by coalescing it with another fragment in the same class into a single larger fragment.

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 fetch the number of fragments, NUM_C, in that class from a name server, and, assuming that all fragments are equal-sized, can compute the expected delay 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 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.

3.5. 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 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. If the site possesses all the referenced fragments, the quoted price is simply the billing rate multiplied by the expected resources to perform the query. If the site does not possess all the 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. The bid might also be adjusted to take into account of amount of recent business and current **load**.

The delay it will promise 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. After adjusting for the current load, it can then estimate the expected delay (the D in the bid).

The expiration date on a bid should be assigned by a site after considering how much risk it is willing to take. A long expiration date incurs the risk of honoring lower out-of-date prices while a too early one runs the risk of the broker not being able to use the bid because of inherent delays in the processing engine.

A site might also consider declining to bid on queries referencing low value fragments. The query will then 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.

4. 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. 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.

4.1. 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.

4.2. 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.

4.3. 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 as well as the name server's rate of update.

5. 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.

[KURO89] present a solution to the file allocation problem that makes use of microeconomic principles, but is based on a cooperative, not competitive environment.

[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 bid a completion time, which they estimate 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 client then sends that task to the server with the lowest completion time.

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.

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 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. 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. We expect that global updates of metadata will suffer from a scalability problem, sacrificing the advantages of the decentralized nature of microeconomic decisions.

More detailed comparisons of Mariposa with other systems can be found in [STON94b].

6. 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.

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, Canada, Aug. 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.

[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., Skeen, D. and Cristian, F., "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," *Proc. 4th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, Portland, OR, Mar. 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, Mar. 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, Alberta, Canada, Aug. 1986.

[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 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 [HUBE88].

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

[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, Jun. 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.

[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.

[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," Sequoia 2000 Technical Report 94/49, University of California, Berkeley, CA, Apr. 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.