Data Replication in Mariposa

Jeff Sidell, Paul M. Aoki, Sanford Barr, Adam Sah, Carl Staelin † Michael Stonebraker and Andrew Yu ‡

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
mariposa@postgres.Berkeley.EDU [1]

Draft of: June 8, 1995

Abstract

The Mariposa distributed data manager uses an economic model for managing the allocation of both storage objects and queries to servers. In this paper, we present extensions to the economic model which support replica management, as well as our mechanisms for propagating updates among replicas. We show how our replica control mechanism can be used to provide consistent, although potentially stale, views of data across many machines without expensive per-transaction synchronization. We present a rule-based conflict resolution mechanism, which can be used to enhance traditional time-stamp serialization. We discuss the effects of our replica system on query processing for both read-only and read-write queries. We further demonstrate how the replication model and mechanisms naturally support name service in Mariposa.

1 Introduction

In this paper we describe replica management in the Mariposa distributed data base management system [STON94a]. There has been considerable research devoted to studying replica management, resulting in a wide variety of proposed solutions. In Mariposa we have adopted an economic framework for managing data objects and query processing [STON94b]. This paper describes how we propose to integrate replica management and replica control into this framework. In addition to describing the design of our replication system, we discuss the impact of replication on query

---

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

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

[1]This research was sponsored by the Army Research Office under grant DAAH04-94-G-0223, the Advanced Research Projects Agency under grant DABT63-92-C-0007, and Microsoft Corp.

optimization and query processing semantics. Finally, we show how our replication system supports name service without additional mechanisms.

A Mariposa system consists of a collection of sites which provide storage, query processing, and name service. More than one service can be provided by one site. The goal of Mariposa is to provide the following features:

- *Scalability*: Our goal is for Mariposa to scale to $10,000$ sites. This makes it necessary for Mariposa sites to operate autonomously and without global synchronization. Database activities such as class creation, updates, deletions, fragmentation and data movement must happen without notifying any central authority.
- *Fragmentation*: Every Mariposa table (class) is horizontally partitioned into a collection of **fragments** which together store the instances of the table. The collection of fragments can be **structured** (partitioned by predicate-based distribution criteria) or **unstructured** (partitioned randomly or in round-robin order).
- *Data movement*: Fragments can move from one site to another without quiescing the database. Data movement makes it possible for Mariposa sites to offload data objects, resulting in load balancing and better system throughput.
- *Flexible copies*: Copies can enhance data availability and provide faster query processing through parallel execution. However, the cost of maintaining the consistency of a large set of replicas can be prohibitive if conventional techniques (e.g., two-phase commit) are used. Mariposa provides a replica-management system that avoids the expensive synchronization requirements of conventional replica systems without sacrificing transaction serializability. Copies are at the granularity of fragments.
- *Rule-based system management*: The behavior of Mariposa sites is controlled by a production rule system, provided as part of the Rush scripting language [SAH94]. Rules are of the form *on* **event** *where* **condition** *do* **action**. The rule system is intended as a flexible tool to implement policy governing buying and selling data, pricing for query processing, etc.

In [STON94b], we expressed query processing in terms of the following economic framework. A user presents a query to a **broker** module. Along with the query, the user specifies a **bid curve**. The bid curve is in two dimensions, *cost* and *delay*, and specifies the maximum payment for a query to be answered within a given delay. Some users will be willing to pay more for a fast answer, resulting in a bid curve with a steep slope. Others will be willing to wait, but have a maximum price they are willing to pay, resulting in a flatter bid curve. The broker accepts the query and the bid curve, constructs an initial query plan using standard single-site query optimization techniques, and then decomposes the resulting plan into subqueries. The broker sends out the subqueries to **bidder** sites, asking them to respond with bids indicating how much they will charge to process the subquery and how long it will take. The bids are of the form `(price, delay)` indicating that the bidder will charge a fixed price to process the subquery and that the work will be completed within the given delay. Each bid can be thought of as a point in the same space as the bid curve. The broker selects a set of bids that (1) corresponds to a set of subqueries that can be assembled into the final query and (2) has an aggregate price and delay farthest under the bid curve. If no set of bids exists that is under the bid curve, the query cannot be run. When the bids have all been received, the broker notifies the winners and the query is processed at those sites.

In this paper, we extend our execution model in two major ways in order to support replicas. The Mariposa copy management model specifies that update transactions performed at one site are allowed to commit. Their changes are then sent to other copy holders within a time bound. This asynchronous replication implies that there will be some loss of consistency between copies. As seen in Figure 1, Mariposa provides an asynchronous replication mechanism in which a **resolver** module at each site arbitrates the application of concurrent updates at that site. The resolver is rule-based, allowing application-level control over the resolution of conflicting updates. Applications that wish to see stable data must limit their access to quiesced data that has passed through the resolver. Applications that are willing to gain concurrency by reading unstable data that may be rolled back may do so.

This architecture has two major implications. First, because copies are out of date by varying degrees, we introduce a **staleness** factor, indicating how out of date a site's copy is. Second, we define semantics that allow applications to understand the implications of reading unquiesced data.

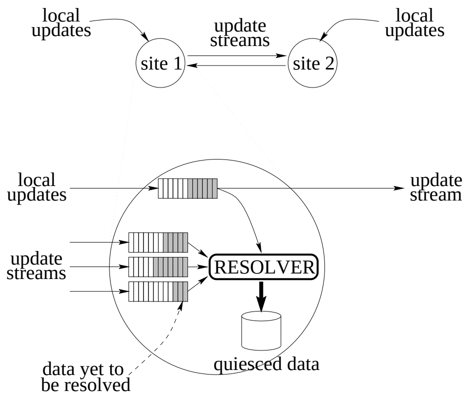

Figure 1: Asynchronous replication architecture.

The remainder of the paper is structured as follows: In Section 2, we cover related work. Section 3 and Section 4 present the basic replication management and control mechanism. Section 5 presents a strategy for query optimization and execution that generalizes and modifies the proposal in [STON94b]. Section 6 describes how the same replica control mechanisms and strategies can be used for name service. Finally, Section 7 summarizes the key points of the paper.

2 Related Work

Rather than presenting an exhaustive survey of the extensive database and operating systems literature in the area of data replication, in this section we focus instead on presenting some of the major ideas on which our model builds.

The *number* and *placement* of replicated objects affects both the performance and the availability of a distributed database. The problem of optimizing this aspect of the physical database design is known as **replica management**. Classic work on replica management concentrated on the file allocation problem [CHU69, DOWD82] — that is, the problem of finding optimal static data layouts. More recent work has been done to develop techniques that adjust the data layout dynamically, such as learning algorithms [WOLF92, ACHA93].

Data replication entails maintaining physical and/or semantic consistency of the various copies. There has also been a tremendous amount of work on this problem, known as **replica control**. See [CHEN92] for a useful overview. Classic distributed data managers require that copies be kept fully consistent. Because of this, considerable effort has gone into improving the basic techniques for ensuring this kind of consistency, such as two-phase commit [ELAB85, SAMA93]. However, because the expense of synchronizing updates remains relatively high, work has also been done in constructing weak consistency models. These models typically place bounds on one or more **divergence parameters**. For example, some systems place bounds on time (temporal divergence control) [AGRA93] or the number of update transactions (value-based divergence control) [KRIS91]. Finally, other systems focus on flexible specification of divergence parameters. Both quasi-copies [ALON90] and epsilon serializability [PU91] permit value-based and temporal divergence control on the underlying data. Bayou [TERR94] takes a slightly different approach, providing various kinds of intuitive consistency guarantees for each user "session" by controlling data divergence as well as using information specific to a session's read/write dependencies.

In contrast to these well-defined semantics, most current commercial systems offer relatively informal, *ad hoc* consistency guarantees as the only alternative to costly full consistency mechanisms. Most systems apply updates in a transactionally consistent way for some granularity of the database. For example, Sybase provides transactional consistency for all data under a single

Replication Server. Oracle7 does so for single tables. However, they do not provide correspondingly powerful mechanisms for controlling target database updates. This means that a transaction that accesses replicas across the highest granularity of consistency may see data that was valid at different times. For example, if a Sybase table F is horizontally partitioned into F_1 and F_2 which are served by two different Replication Servers, a site that has replicas of F_1 and F_2 cannot be assured that a scan of its copy of the combined table F will produce a consistent result.

Although Mariposa can provide standard consistency guarantees using two-phase commit, it also supports temporal divergence between replicas. Mariposa provides consistency at the cost of greater staleness. We provide a transactionally consistent view of the data as of a time in the past. The services provided by the Mariposa storage manager permit a particularly natural kind of time-vintage/time-bounded [GARC82] query model.

3 Economic Replica Management

This section describes replica management in Mariposa: the mechanism by which a Mariposa site acquires, maintains and discards copies of an object. The copy mechanism we describe provides the basis for the discussion of read and read/write semantics in Section 5. Acquiring and maintaining a copy may be thought of as applying streams of updates from other copy holders, with associated processing costs. In the economic parlance of Mariposa, a site buys a copy from another site and negotiates to pay for update streams. In this section, we focus on the mechanism used by a site to buy a copy and contract for these update streams. We then describe how a Mariposa site discards a copy. Finally, we turn to the problem of splitting table fragments into smaller fragments or coalescing fragments into larger fragments.

3.1 Acquiring and Maintaining Copies—Update Streams

The process of buying a new copy is relatively simple. Assume a Mariposa site S_1 owns the only copy of fragment F. If another site, S_2, wishes to store a copy of F, then S_2 must perform the following steps:

1. *Negotiate for updates*: S_2 negotiates with S_1 to buy an **update stream** for F. This contract specifies an **update interval** T and an **update price** P. The update interval specifies that writes to F at S_1 will be forwarded to S_2 within T time units of transaction commitment at S_1. An update stream contains changes only from committed transactions. In this way, S_2 can be assured that its copy of F is out of date by an amount bounded by T plus the

maximum network delay plus the maximum time to install the update stream. In return for the update stream, S_2 will pay S_1 P dollars every T time units. See Section 4.1 for a discussion of the mechanism by which updates streams are generated.

2. *Negotiate reverse updates*: If S_2 wants to write to its copy of F, then it must also contract with S_1 to accept an update stream generated at S_2. In this case, there are two update intervals: $T_{1\rightarrow 2}$ and $T_{2\rightarrow 1}$, which are not necessarily the same. $T_{1\rightarrow 2}$ is the frequency with which S_1 updates S_2, and $T_{2\rightarrow 1}$ is the converse. In this case, the price P mentioned in step (1) above is the price paid by S_2 to S_1 for S_1 sending updates to S_2 and for S_1 receiving updates from S_2. See Section 3.1.1 for a discussion of pricing. See Section 4.2 for a discussion of resolution of conflicting updates.

3. *Construct an initial copy*: S_2 contracts with S_1 to run the query `SELECT * FROM F`. S_2 will install the result of this query and begin to apply the update stream generated by S_1. If S_2 is writing its copy of F, it starts to send updates to S_1 as well.

An easy generalization of this scheme is to allow a copy to be a select-project operation applied to a table. We think of this as requesting a copy of a view. In the above example, S_2 would specify a filter that corresponds to the view and send it to S_1 during step (1) above. S_1 would pass the change list through the filter before forwarding it to S_2.

To generalize the copy mechanism, if a site S_{n+1} wishes to purchase a copy of fragment F of which n copies already exist at sites $S_1 \ldots S_n$, $n > 1$, steps (1) and (2) above can be carried out between S_{n+1} and the other copy holders. In step (1), S_{n+1} will negotiate update streams with $S_1 \ldots S_n$. If S_{n+1} wishes to make writes to F, then in step (2), it can negotiate reverse update streams with $S_1 \ldots S_n$. Step (3) is the same regardless of the number of copy holders: the initial copy can be constructed from one site.

During step (1), the times $T_{i\rightarrow n+1}$ negotiated between site S_{n+1} and the other sites should all be equal. If one of the update intervals were greater than the others, S_{n+1}'s copy would always be out of date by the amount of time specified by the longer interval. Therefore it only makes sense to have the update intervals for the streams going to one site be the same. We think of this time interval as the *staleness* of the data at the site, since the quiesced data is at least that much out of date. We denote the staleness of fragment F at site i as $St(i, F)$. Note that the values of $St(i, F)$ and $St(j, F)$ for two sites i and j are not necessarily equal.

If every site that buys a copy of a fragment performs steps (1) and (2) above with every other copy holder, the result will be $n(n-1)$ one-way update streams. However, during step (1), site S_{n+1} can negotiate an update contract with as few as one other copy holder, S_i, relying on that site to forward updates from the other copy holders. The type and number of update contracts negotiated by a site will affect how out of date its copy will be. For example, suppose S_{n+1} were to purchase an update stream from S_i alone and S_i's staleness were $St(i, F)$. S_{n+1} negotiates an update interval from S_i of $T_{i \to n+1}$. S_{n+1}'s staleness is $St(i, F) + T_{i \to n+1}$. Forwarding in this manner decreases the number of contracts, and therefore the network traffic, for read-only copies.

Step (2) above cannot be modified in this way. If S_{n+1} wishes to make writes to F, it must negotiate update streams with all n sites. Otherwise, S_{n+1} would affect how out of date other copy holders' copies were without the copy holders' consent.

3.1.1 Update Stream Pricing

Mariposa sites share the common goal of being profitable. It is our belief that by mimicking the behavior of economic entities, acceptable system performance and response time can be maintained in the face of varying workloads without sacrificing site autonomy. An analysis of the pricing of update streams reinforces this belief. We first restrict the discussion to one writer site with read-only copies at other sites, then expand the analysis to include multiple writers.

Suppose site S_1 owns a fragment F and another site S_2 wishes to buy a read-only copy of F, with a stream update interval of T time units. S_1 may lose a portion of its revenue from processing queries involving F if S_2 underbids it. In order to guarantee maintenance of its revenue stream, S_1 can examine the average revenue collected from read queries involving F during a time period equal to T and set the update price to that amount. S_2 now pays S_1 an amount equal to what S_1 would have made from processing queries involving F anyway.

In order to make a profit, S_2 must generate more revenue by processing queries involving F than it pays to S_1. If S_1 and S_2 bid on exactly the same queries, then on average S_2 must charge more than S_1, since it is already paying S_1 an amount equal to S_1's revenue from F. Since S_2 is charging more than S_1, it will only be given work if it processes queries faster than S_1, reducing user response time. If S_2 does process queries which otherwise would have gone to S_1, it will have reduced the load on S_1, increasing system throughput. If S_2 has negotiated update streams so that its staleness is less than S_1's, then it can bid on queries that S_1 cannot.

If S_2 does not make a profit from F, it may choose to renegotiate its update contract with S_1. Presumably, S_1 may be willing to do so, since it is processing queries involving F as well as receiving update revenue from S_2. In this way, S_1 and S_2 can work iteratively towards a balanced load on queries involving F. We can also assume that S_2 would not have requested to buy a copy of F unless there were sufficient activity involving F.

Now suppose S_2 wants to make writes to F. S_1 will calculate an update price based on read and write queries, rather than just read queries. If there is a significant number of writes, then this price will be quite a bit higher than that for a read-only copy. Consequently, S_2 will have to charge a lot more for queries involving F (whether read or write). The analysis of read-only copies holds for read-write copies as well: namely, S_2 can only make a profit by processing reads and writes faster than S_1, since it is charging more, thereby reducing user response time and potentially increasing system throughput.

3.1.2 Monopolies and Price Control

Another issue concerns the possible refusal of S_1 to enter into an update contract for F or to insist on a prohibitive price for it, thereby establishing a monopoly on queries involving F. S_1 could then charge exorbitant prices for these queries, resulting in extremely high profitability. If F is involved in lots of queries, it is natural that another site would want to buy a copy in order to attract some of the business. There are two factors that limit the price S_1 can demand for a copy of F:

First, a user can always buy the query `SELECT * FROM F` and then periodically update this copy. Pricing queries in Mariposa is complicated. However, no matter what pricing policy is used by a site, the price cannot exceed what users are willing to pay, as expressed in the bid curve. Since the queries used to acquire and maintain a copy can all be performed by a user, then collectively they set the upper bound on what a site can charge for a copy. Put differently, the receiver can always "do it manually," thereby putting a ceiling on the price of a copy.

Second, suppose site S_1 owns F and has more business than it can handle. It will either raise its price or its delay estimate for queries involving F until demand is decreased to its capacity. S_1 may be able to collect more money by selling a copy to S_2. If S_1's cost per query is Q and its price is P, then it earns a profit of $P - Q$. If S_2 has a lower cost structure and can process queries for $Q' < Q$, then it will be desirable for S_1 to sell a copy to S_2. S_2 will process queries that S_1 could not, either because its price or its delay (or both) were too high. S_2 will also pay S_1 for its update stream. This will result in higher profits for both sites.

3.1.3 Discarding Copies

If a site no longer wishes to maintain a copy, it has a number of options.

- *drop* its copy. That is, stop paying for its update streams, delete the fragment and stop bidding on queries involving the fragment.

- *sell* the copy. The site can try to sell its update streams to someone else, presumably at a profit. If so, then the seller must inform all the creators of update streams to redirect them to the buyer.

- *stop updating*, That is, stop paying for its update streams but don't delete the fragment. The fragment will become more and more out of date as updates are made at other sites. If the fragment is split or coalesced, the fragment will essentially become a view. This view is unlikely to be very useful, since it is unlikely that queries over the relation will correspond exactly to the view. Therefore, doing nothing is a possible but not very effective action.

We designate one copy of each fragment to be the **master** copy. We assume that all other copies can be freely deleted, but the master copy must be retained until sold. This will prevent the last copy of a fragment from being deleted. In addition, the notion of a master fragment is needed to support systematic splitting and coalescing of fragments, a subject which we address now.

3.2 Splitting and Coalescing Replicated Fragments

Having discussed the means by which the creation and deletion of new copies are controlled, we now turn to the ways in which replication affects the fragmentation of a class. We discuss this under replica management because a change in the degree of fragmentation of one replica has an effect on the structure and/or economic value of the other (replicated) collections of fragments.

In order to make the implementation of splitting and coalescing fragments more manageable, we assume that only one site is allowed to initiate splitting of a fragment. For simplicity, we assume that this is the site with the master copy. This site simply splits the fragment, thereby splitting its update streams. It then sends the split update streams on to each other copy. Each holder of a copy must take one of three actions when it receives a split in an incoming update stream:

- *split* its copy in a compatible way and thereby split its update streams into two pieces. Each such site continues to have a copy of the existing fragments.

- *drop* its copy. The site ceases to participate in replication.

- *do nothing*. The effects of failure to split a fragment replica are more or less identical to those of failing to update a fragment replica - the site holds an increasingly out of date view.

A site can initiate coalescing of two fragments only if it possesses both fragments and has the master copy of each one. It then starts sending out a coalesced change stream to all sites that got either change stream before. The site which receives a coalesced change stream can execute the same actions as for the splitting case, with the same results. If a copy holder only has one of the two fragments, and wishes to coalesce it with the other one, it can contract to buy the missing fragment from any of the sites who have copies.

4 Replica Control

We now turn to replica control mechanisms. In this section, we describe the mechanisms by which objects are physically replicated (i.e., how update streams are generated). We then discuss how Mariposa addresses the problem of conflicts due to concurrent reads and writes.

4.1 Replication Mechanisms

Each read-write copy accepts writes and must forward the changes on to the other copy holders within the contracted time intervals. Because write frequency varies from site to site, as does the update interval between sites, it seems reasonable to specify three different update propagation mechanisms: **triggers**, **side files** and **table scans**. We propose using these mechanisms as indicated in Table 1.

write frequency		
low .... *high*		
trigger	side file	table scan

Table 1: Choice of replication mechanism as a function of write frequency.

All three techniques take advantage of certain aspects of the Mariposa storage system, which we briefly discuss at this time. Mariposa uses the POSTGRES "no overwrite" storage system [STON87]. Each record has an object identifier (OID), a time stamp (TMIN) at which it becomes valid and another timestamp (TMAX) at which it ceases to be valid. An INSERT at a site S causes a new record to be constructed with an OID and TMIN field but not TMAX. A DELETE operation causes the TMAX field for an existing record to be added. Lastly, an UPDATE is a DELETE operation followed by an INSERT operation using the same OID. By keeping old, timestamped versions, the POSTGRES no-overwrite storage strategy makes it possible to run read queries as of a time in the past. Postgres also assigns a unique transaction identifier (XID) to each transaction, and a unique identifier to each operator (OP-ID). To be able to detect and correct both write-write and read-write conflicts, the update stream must contain:

```
(XID, {Read-Set}, {Write-Set})
```

A **read** (**write**) set is a list of tuple/attribute identifiers read (written) at the update site. The read sets will contain only the tuple/attribute identifiers. The write sets will also contain the OP-ID and list of operands, as well as other log entries:

```
(OID, OLD-VALUE, NEW-VALUE, OP-ID, OPERAND-LIST, TMIN, TMAX)
```

In Section 4.2 we present examples of both write-write and read-write conflicts and their resolution.

The first mechanism by which update streams are generated, triggers, takes advantage of the Postgres rule system [STON90]. A site with a copy of F using triggers as the update mechanism would install the rule:

```
CREATE RULE copyF AS ON UPDATE TO F DO
      [ INSERT INTO F1 SELECT current.*; ...  ];
```

Triggers are appropriate for sites with very low write frequency, since each update results in a rule firing and network communication with the other copy holders.

If there are relatively few writes, then a side file approach is preferable. Specifically, install a trigger at S_1 that makes the correct notations in a side file F-SF every time F is written. This notation includes the values for TMIN, TMAX, OID, etc. Now have S_1 run the following query every T time units, and send the result to S_2:

```
SELECT * FROM F-SF
```

When the copy has been safely installed at S_2, S_1 can delete the records in F-SF inserted since the last update:

```
DELETE FROM F-SF WHERE TMIN <= (last-update)
```

This scheme will forward the contents of every committed record within T time units. Of course, we need to "fake" the receiving site into installing records with pre-existing TMIN, TMAX, etc. This can easily be done with a carefully coded user defined function.

If there is high write volume, then we might want to avoid the inefficiency caused by the trigger system making a copy of each write in F-SF. Another alternative is for S_2 to wake up after each update interval and run the query

```
SELECT * FROM F [now - T, ]
```

which will find all the changed records with a single scan. This will avoid copying write records to F-SF but will require reading the entire table (ignoring indexing). If the update interval is long enough, this will be an attractive alternative. Roughly, the cost of the side file approach during each update interval, T, will be dominated by the cost of writes to F-SF:

$$cost_{sidefile} = \frac{cost_{write} \times nWrites}{T}$$

The cost of the sequential scan during an update interval is the cost of a read multiplied by the number of pages in the relation (ignoring indexing):

$$cost_{scan} = \frac{cost_{read} \times pages}{T}$$

Combining these two expressions, we arrive at a **write threshold**:

$$\text{write threshold} = \frac{cost_{read} \times pages}{cost_{write}}$$

We now have the following decision criteria for our replication mechanisms. If the update rate is extremely low, one can avoid the overhead of auxiliary side files and table scans by using the trigger mechanism. When the number of writes during the update interval is below the write threshold, it is more efficient to use the side file mechanism for updating. Otherwise it is better to use a sequential scan of the table.

4.2 Conflict Resolution

If more than one site is allowed to write copies of a fragment, then the possibility of **conflicts** is introduced. In Mariposa, each copy is a **version** of an object, and processing the update stream is equivalent to a version merge. If there are multiple copies of a fragment, each site must perform a version merge operation upon receiving an update stream from another site. Mariposa provides **rule-based** conflict resolution. Rules may be defined to provide traditional timestamp-order serialization. However, the Mariposa rule system allows a more flexible resolution of update conflicts which may better reflect the needs of the underlying application.

Update conflict events specify the conflicting sites and the fragment. Conditions can specify the scope of the particular rule, thereby providing fine- or coarse-grained control over conflict resolution. For example, an event could be `UPDATE-CONFLICT(`S_1, S_2, F`)`. A condition such as `(EMP.name = 'Bob')` has narrow scope—one tuple, while an empty condition would broaden the scope to include all update conflicts between S_1 and S_2. The action clause could specify that one or the other transaction be rolled back, in effect declaring one site or one user the winner. Or it could specify that both transactions be rolled back and mail sent to a responsible party. Rules can also be used to enhance timestamp resolution, for example by notifying two salespeople who have ordered the last of an item, and automatically generating a back-order letter to be sent to the customer.

In the examples that follow, sites S_1 and S_2 each have a copy of fragment F. Both sites can perform updates to F. Update streams have been negotiated between the two sites. Each site has an associated staleness, $St(i, F)$ as discussed in Section 3.1. We define two read/write transactions, X_1 and X_2 occurring at sites S_1 and S_2, respectively. Since there are two writers, there is a possibility of conflicting updates. The update streams described in Section 4.1 contain enough data for each site to detect both write-write and read-write conflicts. A write-write conflict occurs when the write set of one transaction and the write set of another transaction have a non-empty intersection. A read-write conflict occurs when the read set of one transaction has a non-empty intersection with the write set of another. As mentioned in Section 4.1, in order to detect both write-write and read-write update conflicts, it is necessary to include the read set and write set of every transaction in the update stream. To detect and correct write-write conflicts requires only the write set. In Mariposa, we propose to permit turning off read-write conflict detection and correction. If read-write conflict detection is turned off for one fragment, it will be turned off for all copies of that fragment. This will provide enhanced system performance at the risk of creating an inconsistent database state.

4.2.1 Examples of Conflict Resolution

The following two transactions, X_1 and X_2, are an example of a write-write conflict if Bob works in Accounting:

X_1:
```
UPDATE emp SET salary=15000
 WHERE dept = 'Accounting';
```
X_2:
```
UPDATE emp SET salary=12000
 WHERE name = 'Bob';
```

The update streams for X_1 and X_2 would contain the following information for the tuple corresponding to Bob, if read-write conflict detection were turned off:

X_1: (XID_1, {OID_{Bob}, salary}, 10000, 15000, write, 15000, $TMIN_1$, $TMAX_1$)

X_2: (XID_2, {OID_{Bob}, salary}, 10000, 12000, write, 12000, $TMIN_2$, $TMAX_2$)

The following rule will resolve the conflict in timestamp order:

```
CREATE RULE resolve_{1,2} AS ON CONFLICT(S_1, S_2, F) DO
      [
      INSERT X_1 INTO F AS OF TIME TMIN_1
      INSERT X_2 INTO F AS OF TIME TMIN_2
      ];
```

If the user determined that updates to salaries made by senior management override those made by department managers, the following rule would resolve the conflicting updates noted above correctly:

```
CREATE RULE resolve_{1,2} AS ON CONFLICT(S_1, S_2, F) DO
      [
      if (USER_2 = MANAGER*(USER_1))
         INSERT X_2 INTO F AS OF TIME TMIN_2
      else
         INSERT X_1 INTO F AS OF TIME TMIN_1
      ];
```

The two transactions below are an example of a read-write conflict, if Bob works in Accounting:

```
X_1:  UPDATE emp SET salary=salary*1.1
       WHERE dept = 'Accounting';
X_2:  UPDATE emp SET salary=salary*1.5
       WHERE name = 'Bob';
```

The update streams for Bob's tuple would look like:

X_1: $(XID_1$, $\{OID_{Bob}$, dept, salary$\}$, $\{OID_{Bob}$, salary$\}$, 10000, 11000, multiply, 1.1, $TMIN_1$, $TMAX_1)$

X_2: $(XID_2$, $\{OID_{Bob}$, name, salary$\}$, $\{OID_{Bob}$, salary$\}$, 10000, 15000, multiply, 1.5, $TMIN_2$, $TMAX_2)$

Notice that `salary` is in both the read and write sets of each transaction. The following rule would serialize the transactions correctly, regardless of the order of arrival of the update streams:

```
CREATE RULE resolve_{1,2} AS ON CONFLICT(S_1, S_2, F) DO
    [
    if (TMIN_1 < TMIN_2)
        INSERT X_1 INTO F AS OF TIME TMIN_1
        if (READ-SET_1 ∩ WRITE-SET_2 ≠ ∅)
            re-run query over READ-SET_1 ∪ WRITE-SET_2,
    else
        INSERT X_1 INTO F AS OF TIME TMIN_1
    ];
```

Using rules to perform conflict resolution is not without its complications. Clearly, all the rules dealing with update conflicts to a relation must be the same at all sites that hold copies. Furthermore, the rules must be commutative—that is, the outcome of the rules must be the same regardless of the order in which they are fired. Finally, the complexity of rule systems presents a problem: Rules could conflict, for example, if S_1 and S_2 make conflicting updates to EMP and DEPT and there is a rule for EMP favoring S_1, while the rule for DEPT favors S_2. Rules could violate transactional consistency: if S_1 and S_2 make conflicting updates to EMP and DEPT, which are stored at different sites, say S_3 and S_4, there is presently no mechanism to guarantee that the rules will result in a consistent database state. This is a subject for further study.

5 Query Processing

The non-transactional replication mechanisms described above will lead to divergence of the contents of fragment copies. In this section, we elaborate on this divergence and describe the effect it has on our economic query processing model. We first discuss the effects of temporal divergence—different data stalenesses at different sites—on our original query processing algorithms. Finally, we show how we modify traditional transactional semantics in order to increase the ability of applications to read and write replicated data that may not be consistent between copies.

5.1 Time Validity of Copies

As defined in Section 4.1,the staleness of a fragment F at a site S_i is equal to the update interval S_1 has negotiated with the other update sites. That is, after a time period equal to $St(i, F)$ has elapsed, S_i has received and resolved all updates to F that happened at time $now - St(i, F)$ or earlier. To guarantee that it is reading consistent, stable data, a query to F at S_i must be run no later than $St(i, F)$.

That is, a query

```
SELECT ...  FROM F
```

becomes

SELECT ... FROM F [$now - St(i, F)$]

or a similar query "as of" a time earlier than $now - St(i, F)$ Otherwise, a query may give an answer based on data that is not valid, since potential conflicts from update sites have not been resolved.

5.2 Implications of Time Validity on Read Query Processing

Figure 2 shows how the broker in the original Mariposa model decided which bids to accept. Assume that we have a query plan that has been fragmented into two subqueries, A and B, that must be run sequentially. In Figure 2(a), the broker has received two bids for A and one bid for B. Figure 2(b) shows that the broker can construct two sets of bids that answer the complete query (i.e., it can combine $B1$ with either $A1$ or $A2$). However, when the two bids making up bid set 2 are added together, the sum is farther under the bid curve than that of bid set 1. In this case, the broker would select bid set 2, even though bid set 2 is expected to be completed more slowly than bid set 1.

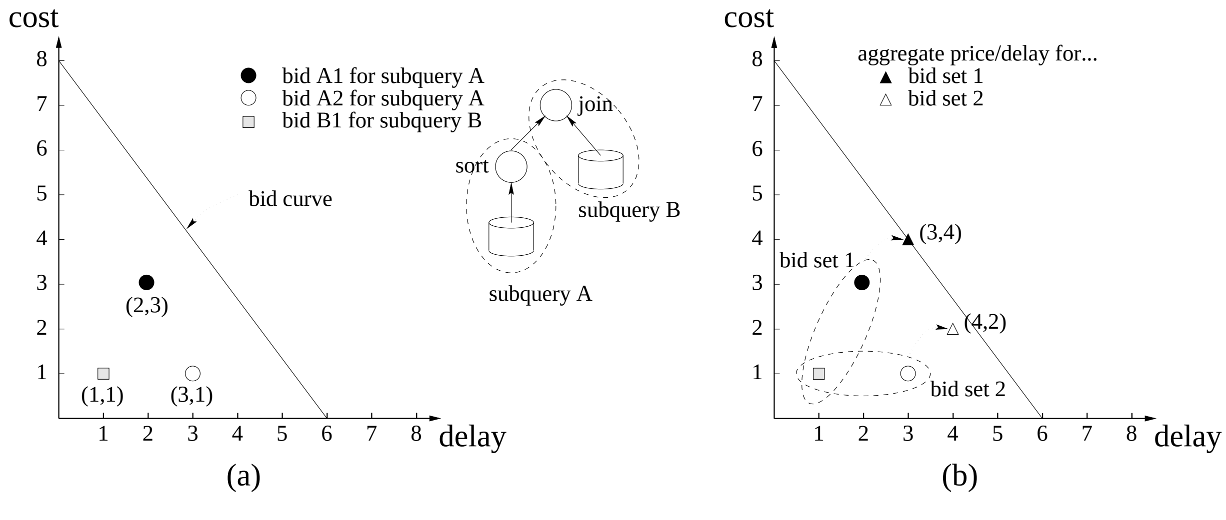

Figure 2: Assembling query plans using 2-D bid curves.

Because of the staleness of copies, we have expanded this model by adding staleness as the third bid parameter. The user specifies the bid curve by providing three non-collinear points $maxBid_1, maxBid_2, maxBid_3$, as well as three values $maxCost$, $maxDelay$, $maxStaleness$. The first three points are representative bids that define a plane limiting the space of acceptable bids.

This plane is shown in Figure 3(a). Acceptable bids are points that fall underneath the plane and whose values for cost, delay and staleness are less than the maximum values specified. The space of acceptable bids is a convex polyhedron with up to seven faces, as shown in Figure 3(b). The user also provides a boolean parameter $resolved$. If $resolved$ is true, then the user requires that the data used to process the query be resolved. If $resolved$ is false, then the user is willing to run the query over unresolved data, which may be rolled back.

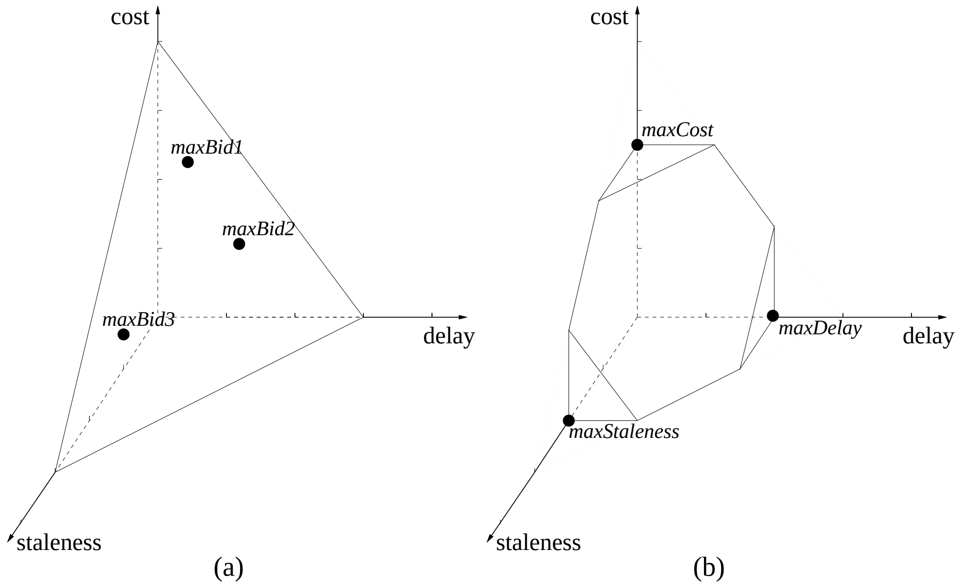

Figure 3: Determining the acceptable bid space.

Processing sites can bid based on the following criteria: if $resolved$ is false, any site can bid. If $resolved$ is true:

$$\begin{array}{lll} \text{if} & maxStaleness_{user} < St(i,F) & S_i \text{ cannot bid} \\ \text{if} & maxStaleness_{user} \geq St(i,F) & S_i \text{ can bid} \end{array}$$

Users can have a low tolerance for delay or staleness, in which case they should expect to pay more money for answers. Increased staleness will allow additional sites to bid on their queries and presumably improve the price. Increased delay will allow queries to run more slowly and perhaps improve the price. The job of the Mariposa broker is to solve the query as far under this bid curve as possible. The broker begins by sending out requests for bids (RFB's) to prospective bidder sites. Each RFB includes $maxStaleness$ and $resolved$. Each bidder site responds if it can bid. When the broker receives bids, it tries to select the group of bids farthest under the bid curve.

Each bid is a point in 3-space. A set of bids, therefore, is also a point in 3-space, the result of adding together the $price$ values, and taking the max over the $delay$ and $staleness$ values, of the constituent bids. Although the **greedy** algorithm shown below will not always give the optimal solution, we believe it will work well in practice. If the user has set *resolved* to false, then a simpler algorithm which only considers price and delay can be used.

$\delta_{max} = 0$
for $i = 1$ to $nSubQueries$
    $\beta_i.price = 0$
    $\beta_i.delay = \infty$
    $\beta_i.staleness = \infty$
for each bid, $B_i(1 \leq i \leq nSubQueries)$ do
        $price_{total} = 0$
        for $k = 1$ to $nSubQueries$
            if $k = i$
                $price_{total} = price_{total} + B_i.price$
            else
                $price_{total} = price_{total} + \beta_i.price$
        $delay_{max} = max(\max_k(\beta_k.delay), B_i.delay), 1 \leq k \leq nSubQueries, k \neq i$
        $staleness_{max} = max(\max_k(\beta_k.staleness), B_i.staleness), 1 \leq k \leq nSubQueries, k \neq i$
        $\delta' = \delta(price_{total}, delay_{max}, staleness_{max})$
    if $|\delta'| > |\delta_{max}|$
        $\beta_i = B_i$

where:

$$
\begin{aligned}
\delta() &= \text{distance of point to bid curve} \\
\delta_{max} &= \text{distance of best bid so far from bid curve} \\
nSubQueries &= \text{number of subqueries for which bids are being received} \\
\beta_i &= \text{best bid so far for subquery } i
\end{aligned}
$$

Once a broker has selected a group of processing sites, it makes bid awards to those sites. If a user has set *resolved* to true, then a query that is processed at more than one site must be run as of time $now - maxStaleBid$, as calculated in the above algorithm, at all the sites. If *resolved* is false, then each processing site must perform its subquery as of time now.

Recall that each bidder that bids on a query for which the user has specified transactional consistency must have a value of $St(i, F)$ that is less than that specified by the user in the bid curve or the site would not have bid. Therefore, every processing site that responds is able to run the query. However, this algorithm may err on the side of caution. Suppose the site with the largest value of $St(i, F)$ doesn't process its part of the query until the last step of the query. The query could possibly be run as of a more recent time than $now - maxStaleBid$.

One possible solution would be for the broker to use the delay estimates specified by the processing sites and calculate the maximum staleness as of the time a site would start processing its stage of the query, not as of the time the bid awards were sent out. This approach, while intuitively appealing, is probably impractical, since it depends on *extremely* accurate delay estimates from the processing sites.

Another approach is for the sites to keep track of the most stale tuple accessed during query processing. While this approach will give a more stale answer in some cases than the previous one would, it does not depend on making extremely accurate delay estimates, and is therefore much more practical.

5.3 Data Visibility for Read-Write Transactions

The conflict resolution technique described in Section 4.2 means that the effects of a transaction on a fragment that has multiple copies are not permanent until after the staleness period has passed. In this section, we consider the effects of this conflict resolution strategy on update transactions. Unlike read-only transactions, update transactions must make their writes as of time "now." Without locking all copies of a fragment, there will always be the possibility of two writes conflicting with one another. This leads to the question of whether unresolved writes should be made visible within a read-write transaction, and to other read-write transactions. We propose to allow a user to read his own unresolved writes within a transaction, but not read those of other transactions.

As noted in Section 3, a copy of a fragment at site S_i is resolved within time $St(i, F)$. If a transaction writes a value x and then uses the updated value in subsequent operations, there is a chance that another transaction S_j will also write x and be serialized before S_i, causing S_i to be re-run with the value of x written by S_j. However, allowing a transaction to read its own writes does not affect the probability of rollback. Nor does it increase the amount of bookkeeping required to roll it back, or the complexity of the resolver. Moreover, it is a feature that users are unlikely to be willing to live without.

Allowing a read-write transaction to see the effects of other update transactions' uncommitted writes would slightly decrease the probability of rollback, but at the cost of excessive bookkeeping. By seeing unresolved writes of n other transactions, an update transaction would decrease the number of possible conflicting updaters by n. However, in order to remain transactionally consistent, a transaction would have to run at exactly the sites that ran the transactions whose writes it is reading. For example, say the `EMP` table is at three sites, $S_{1...3}$ and the `DEPT` relation is at sites S_4 and S_5. Transactions X_1 and X_2 update both relations. If transaction X_1 updates `EMP` at site S_1 and `DEPT` at site S_4, then if X_2 sees X_1's unresolved updates, it must run at sites S_1 and S_4

also. Not only is keeping track of this information a bookkeeping nightmare, but this approach constrains the Mariposa execution model by restricting the sites at which a transaction can run.

The approach described above provides decreased response time, allowing a transaction to commit and return control to the application, but at the expense of being able to see the effects of a transaction until a resolver has run. We feel that in many cases this will be an attractive option compared to two-phase commit, which is effectively the converse. Two-phase commit sacrifices response time for transactional consistency and serializability.

6 Name Service

In Mariposa there are one or more **name servers**, whose job is to maintain a data base of records of the form:

(table-name, fragment-locations, other-information)

A site contacts a name server if it needs information on a table which is not present at its site. The name server responds with parsing, optimization and location information. There can be as many name servers as necessary, and each can manage only some specific part of the entire name space. For example, one name server might focus on personnel data, another on product data, etc.

In a large network it is prohibitively expensive to require name servers to be transactionally consistent with the data they are locating. Doing so would require that every site send a message to every name service every time it created, dropped or moved a fragment. Hence, the original Mariposa design [STON94a] assumed that name servers could have a specified quality-of-service, which was the degree of staleness they maintained. Name service therefore fits very naturally into the consistency model described in the previous sections.

To implement this notion of name service is straightforward on top of the replica management system we have presented. Specifically, each Mariposa site has a TABLES table and a COLUMNS table that form part of the system catalogs at each site. In addition, there are tables for index, type, function and inheritance information. A name server comes into existence and then negotiates with some collection of Mariposa sites to make a replica of a specific view on each local system catalog that contains objects of interest to the name server. The name service data base is then the union of these views. Moreover, the name server can specify the quality of service that it will support by setting up appropriate contracts for these replicas. The name server responds to ordinary queries to this union view using the normal Mariposa query mechanism. As such, there is essentially no custom code required to set up a name service; it is merely a general purpose Mariposa site managing a particular kind of replica.

If an object is sold, and thereby moved from one site to another, then the name servers are alerted within the appropriate delays. If a broker receives out of date information and subsequently sends a request for bid to a site which has recently sold the fragment to a second site, then the selling site can respond in one of 4 ways:

- *no bid*; object not here.
- try to *subcontract* the query to the receiving site.
- *forward* the request for bid on to the receiving site.
- *keep* the sold object around for a little while as it grows increasingly out of date. The seller can bid on queries whose range can be satisfied.

All are reasonable strategies, and a site can execute any one of them.

7 Conclusions

In this paper, we have shown how the economic mechanisms described in [STON94b] can be extended to support replicated fragments. Replica management falls neatly into the economic model. Sites buy and sell copies of fragments in response to changing activity. We have also defined replica control in terms of the economic model. The staleness of a site's data is inversely proportional to the amount it is willing to pay for update streams from other copy holders. We have described the mechanisms by which update streams are generated. We went on to describe a flexible rule-based conflict resolution mechanism, which can be used to enhance traditional timestamp based resolution. We described the effect of asynchronous replication management on query processing. Our model now accounts for staleness as well as price and delay. We described how read transactions can be processed to produce a consistent, stable view of the database by processing the queries as of a time in the past, once conflicting updates have been resolved. We place data visibility restrictions on read-write transactions, allowing a transaction to see its own writes within the transaction, but not the writes of other transactions. We believe this approach complements traditional two-phase commit. Finally, we have demonstrated how the Mariposa internal name service is a natural application of our replication scheme.

8 Acknowledgements

Sunita Sarawagi and Avi Pfeffer provided a great deal of useful input and criticism during this work. Rex Winterbottom and Robert Patrick have contributed to the Mariposa implementation.

References

[ACHA93] S. Acharya and S. B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Brown University, Providence, RI, Sept. 1993.

[AGRA93] D. Agrawal and S. Sengupta. Modular synchronization in distributed, multiversion databases: Version control and concurrency control. *IEEE Trans. on Knowledge and Data Eng.*, 5(1):126–137, Feb. 1993.

[ALON90] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an informational retrieval system. *ACM Trans. on Database Sys.*, 15(3):359–384, Sept. 1990.

[CHEN92] S.-W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical Report CUCS-006-92, Columbia Univ., New York, NY, 1992.

[CHU69] W. W. Chu. Optimal file allocation in a multiple computer system. *IEEE Trans. on Computers*, C-18(10):885–889, Oct. 1969.

[DOWD82] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *Computing Surveys*, 14(2):287–313, June 1982.

[ELAB85] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. *Proc. 4th ACM SIGACT-SIGMOD Conf. on Principles of Database Sys.*, pages 215–228, Mar. 1985.

[GARC82] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Trans. on Database Sys.*, 7(2):209–234, June 1982.

[KRIS91] N. Krishnakumar and A. J. Bernstein. Bounded ignorance in replicated systems. *Proc. 10th ACM SIGACT-SIGMOD Conf. on Principles of Database Sys.*, pages 63–74, May 1991.

[PU91] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. *Proc. 1991 ACM SIGMOD Conf. on Management of Data*, pages 377–386, May 1991.

[SAH94] A. Sah, J. Blow, and B. Dennis. An introduction to the Rush language. *Proc. 1994 Tcl Conf. (Tcl'94)*, pages 105–116, June 1994.

[SAMA93] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. *Proc. 9th Int. Conf. on Data Engineering*, pages 520–529, Apr. 1993.

[STON87] M. Stonebraker. The design of the POSTGRES storage system. *Proc. 13th Int. Conf on Very Large Data Bases*, pages 289–300, Sept. 1987.

[STON94a] M. Stonebraker, P. M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. *Proc. 10th Int. Conf. on Data Engineering*, pages 54–65, Feb. 1994.

[STON94b] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. *Proc. 3rd Int. Symp. on Parallel and Distributed Info. Sys.*, pages 58–67, Sept. 1994.

[STON90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. *Proc. 1990 ACM SIGMOD Conf. on Management of Data*, pages 281–290, June 1990.

[TERR94] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. *Proc. 3rd Int. Symp. on Parallel and Distributed Info. Sys.*, pages 140–149, Sept. 1994.

[WOLF92] O. Wolfson and S. Jajodia. An algorithm for dynamic data distribution. *Proc. 2nd Wksp. on the Management of Replicated Data*, pages 62–65, Nov. 1992.