Mariposa: A New Architecture for Distributed Data

Michael Stonebraker, Paul M. Aoki, Robert Devine, Witold Litwin and Michael Olson

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

Abstract

We describe the design of Mariposa, an experimental distributed data management system that provides high performance in an environment of high data mobility and heterogeneous host capabilities. The Mariposa design unifies the approaches taken by distributed file systems and distributed databases. In addition, Mariposa provides a general, flexible platform for the development of new algorithms for distributed query optimization, storage management, and scalable data storage structures. This flexibility is primarily due to a unique rule-based design that permits autonomous, local-knowledge decisions to be made regarding data placement, query execution location, and storage management.

1. INTRODUCTION

There have been four approaches to the management of distributed data that have been extensively investigated by the research community or commercial products, namely:

- distributed database systems
- client-server distributed file systems
- deep store file systems
- object-oriented database systems

The characteristics of each approach can be summarized as indicated in the first column of Figure 1. First, each approach has some fundamental unit of storage allocation, protection and naming. Hence, decisions must be made as to whether such storage objects have a fixed **home** and how movement of storage objects from one site in a computer network to another should be controlled. Second, all four approaches cache data at sites remote from their home. This requires decisions with respect to the size of the objects to be cached, the duration for which objects may be cached, the mechanism for cache management, and whether objects change representation when cached. Lastly, each approach implicitly defines a model of query execution. When a query appears at site A which requires data at site B, there are two possible query execution strategies. The data at site B can be moved to site A for processing, leading to a strategy of **move the data to the query**. Alternately, the query can be sent from site A to site B, thereby implementing a **move the query to the data** strategy.

Having defined the important architectural aspects of a distributed data manager, we now turn to exploring the columns of Figure 1 as they apply to existing distributed data management systems.

We first consider the approach taken by distributed database management systems. Several distributed DBMS prototypes were developed during the early 1980s, such as R* [WILL81], SDD-1 [BERN81], SIRIUS-DELTA [LITW82], and Distributed INGRES [STON86a]. All extended single-site DBMSs to manage relations that were spread over the sites in a computer network in a "seamless" manner. In order to do so, researchers developed techniques for handling distributed query optimization [EPST78, SELI80, BERN81], distributed transactions [SKEE81, LIND83, BERN81], and (sometimes) multiple copies of data [STON79]. Commercial systems based on these techniques are now available from several relational DBMS vendors.

Architecturally, the distributed database approach looks quite different from the other approaches. Each distributed DBMS assumes a "shared nothing" architecture [STON86b] and has the basic processing philosophy of "move the query to the data." As a result, these systems allocate data to sites in a computer network, and this allocation remains in place until it is subsequently changed by a database administrator. Hence, data objects have a fixed home and queries are moved to the home of the data for processing. If a data object must be moved in order to

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.

Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant #1243. Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

	Distributed DBMS	Distributed file system	Deep store file system	OODB	Mariposa
Unit of storage (object)	fragment	file	file	class	fragment
Fixed object home	yes	yes	yes	yes	no
Site control	human	human	human	human	internal rule system
Object cached	fragment	block	segment	block	fragment
Length of caching	one query	procedurally controlled	procedurally controlled	procedurally controlled	internal rule system
Caching policy	n/a	LRU	typically weighted LRU	LRU	defined with rules
Object changes representation when cached?	no	no	no	yes	yes or no
Entity moved	query	data	data	data	query or data

Figure 1. Architectural alternatives for distributed data.

perform a join, then a temporary copy of the data is made which is valid only for the duration of join processing. Moreover, all secondary indexes must be rebuilt on these temporary copies. Several systems proposed decomposing relations into a collection of **fragments** whose composition is controlled by **distribution criteria**. To the best of our knowledge, no system actually implemented fragments; however, in theory, it is the unit of storage allocation.

A second approach to distribution is that taken by client-server file systems such as Andrew [HOWA88] and the NFS-oriented commercial offerings. In such systems, a file (or collection of files) is the unit of storage allocation and has a unique home on some server. Relatively small, fixed-size blocks from the file are brought to a client site upon access, thereby implementing a strategy of "moving the data to the query." Blocks are cached on a client site until they are no longer worthy. Most file systems implement a **hard-coded** cache manager, typically utilizing a least-recently-used (LRU) eviction strategy.

A third approach to distribution is that of tertiary memory, or **deep store**, file systems. Systems in this category include Inversion [OLSO93], Highlight [KOHL92], and commercial offerings such as Epoch [EPOC92] and UniTree [GENE91]. Such systems typically offer a seamless caching service for a tertiary memory system. Some products require that the disk cache and the deep store be on the same physical machine. However, most allow the tertiary memory device to be remote from the disk cache. In this case, the granules that are moved from tertiary memory to or from disk are typically either whole files or large fixed- or variable-length **segments**. Each segment has a unique home and does not change format when it is moved. Lastly, like distributed file systems, deep store file systems implement a "move the data to the query" strategy.

A fourth approach to distribution is implemented by the commercial object-oriented DBMS vendors as well as by assorted research prototypes. All assume a client-server environment and move disk blocks back and forth between the client and the server. Data objects have a specific home, and these systems implement a "move the data to the query" strategy.

Notice in Figure 1 that distributed file systems and deep store file systems have great commonality. Except for the size of the object cached at client sites, they look essentially the same. Both move objects only under user control (by deleting and recreating the file), cache objects with a hard-coded procedural cache manager, and move the data to the query as a query execution strategy. An OODB also has much in common with both of these systems. Except for converting representations when a block is cached at a client site, an OODB looks architecturally like a distributed file system. On the other hand, a distributed DBMS implements a strategy of moving the query to the data and caches fragments at another site when required to do so during query execution but only for the duration of the query. In a distributed DBMS there is no notion of long-term caching or moving the data to the query.

The Mariposa architecture has the characteristics indicated in the last column of Figure 1. The next section describes the motivations and objectives behind this architecture.

2. MARIPOSA OBJECTIVES

The fundamental objective of Mariposa is to **unify** the (up until now) disparate approaches of distributed DBMSs, caching-based distributed file systems, deep-store file systems and object servers. Mariposa distributes data over a collection of **sites** that are connected by some form of local- or wide-area network. Each site has a storage device, which for purposes of this discussion, we will assume to be main memory, disk, or a robotic tertiary storage device.

When more than one storage device is attached to the same CPU, we will assume that this aggregate forms two or more logical Mariposa sites. Although the reader might expect many storage devices to be attached to a given CPU, we expect the contrary to be the case in practice for the following reasons. First, tertiary memory devices are extremely expensive and slow. As such, it is reasonable to drive each (six figure in cost) tertiary memory device with a private (four figure in cost) CPU. Moreover, tertiary memory can be on the other end of a LAN or high-speed WAN from its disk cache, without adverse performance implications. As a result, disk caching for tertiary memory will often be on a remote machine. Main memory caching for disks will usually be local but could be remote. Local caching from one device to another will entail a **move** from one Mariposa logical site to another, even though no actual networking traffic takes place.

Mariposa supports the following data model. A Mariposa database consists of **instances** of objects in named **classes**, each containing a collection of **attributes** of specific data types. Each class is divided into a collection of **fragments**, which are the unit of storage in the system. Storage sites may split or coalesce fragments as desired. Fragments can optionally have a **distribution criteria** which controls the logical composition of instances in the fragment. Fragments do not have a specific home and can move freely within the network. Alternately, one can think of a Mariposa fragment having a current location which may rapidly change. For example, if a fragment is being accessed frequently, one would expect its location to be moved from tertiary memory to disk, or even to main memory. When activity abates, its location would migrate to slower storage. In a wide-area network environment, a fragment might move to the East Coast during the morning hours and then migrate westward as the day progressed and the access patterns gradually shifted from east to west.

Each Mariposa site is **locally autonomous**, i.e., the site's database administrator (DBA) fully controls object storage at that site. The DBA does this by specifying a set of **rules** in a language to be described presently. For example, the DBA must specify rules that control the eviction of fragments from his site when storage is overloaded. Each site can decide locally what fragment to evict and where to try to send it. As we will see, a designated recipient need not accept the offered fragment, which will lead to the evicting site having to try multiple recipients. It is not necessary that the rules in place at the various sites be consistent. For example, it is reasonable for a tertiary memory system in San Francisco to overflow to a tertiary memory system in New York and vice versa. In this way, each site is the overflow for the other and a logical loop exists in the rule-base.

Mariposa has several objectives with respect to fragment movement. First, movement should be very fast. For example, this is particularly true of movements from tertiary memory to magnetic disk. Mariposa should be no slower than a deep-store file system in performing this operation. Second, fragment movement should have low processing overhead. When moving a fragment from magnetic disk to main memory, it is sometimes desirable to **convert** the object from a disk-oriented representation to a main memory-oriented representation. Current OODBMSs perform pointer-swizzling and some remove disk blocks as object containers during this conversion process. Of course, such conversions are very expensive and are only justified if the objects in the block are going to be accessed several times before being moved to a different storage manager. In some cases, conversion of data representations cannot be avoided when moving from one storage manager to another. This occurs, for example, when one moves from a machine with "big-endian" integers to "little-endian" integers. In general, however, conversion should be performed only when necessary. Finally, when a fragment is moved, one wants to preserve the secondary indexes for the fragment. Unlike current distributed DBMSs, which lose indexing information when they move a fragment, it is an explicit goal of Mariposa to preserve access paths upon data movement.

The Mariposa approach to these objectives is to designate one representation which each storage manager must support, the so-called **canonical** representation, and a second optional **private** representation. As such, fast fragment movement can be accomplished by moving the canonical representation from one storage manager to another. On the other hand, a locally optimal representation can be optionally supported, in which case moving a fragment from one site to another requires conversion from the sender's private format to canonical representation and then into the recipient's private format. Of course, the expected number of accesses must justify this conversion cost.

In Mariposa, it is possible to **move** a fragment from one site to another or to **cache** a fragment at a second site. In this case, a **copy** of the fragment has been constructed, and Mariposa keeps track of a varying number of copies of a fragment. Of course, we require a copy update algorithm, which is discussed in a later section.

Lastly, Mariposa has a **query optimizer** which must generate a plan for solving a query. Obviously, the Mariposa query optimizer must contend with the standard distributed query optimization problem, on which much has been previously written. However, Mariposa must deal with the following additional difficulties.

First, no site has perfect knowledge of global database state. Previous systems have generally assumed (implicitly or explicitly) perfect global knowledge of the data layout, database schema and other factors critical to traditional cost-based query optimizers; data layout was assumed to be either static or slowly-changing, and the use of invalid metadata in query optimization was either not permitted or assumed to be an infrequent event that could safely result in reoptimization. In Mariposa, sites can move, split, merge or change the schema of data fragments without notifying other sites. Consider the implications of the fact that fragments may be in fairly rapid motion between sites. This means that optimizer has (at best) imperfect information about the location of fragments. Although it is possible to “freeze” a fragment at a given site, we believe that this interferes with local autonomy at that site, and is undesirable. Without a freeze operation, the optimizer cannot be assured of the location of fragments and must therefore produce query plans that can be adapted to changes in fragment location.

Second, the optimizer has the option of moving the query to the site where it thinks the data resides or bringing the data to the site of the query. The choice of which strategy to employ is an optimization decision. Although it is self-evident that an individual query is always optimally solved by moving the query to the data, if the optimizer takes a broader view and examines the likely query stream that will be seen by a site, it is possible that moving the data will be a better option. Hence, the Mariposa optimizer considers both possibilities.

Third, should Mariposa decide to move the data to the query, it must choose which copy to move or decide to make a new copy, again leading to an increase in complexity.

Fourth, we assume that Mariposa sites can be **heterogeneous**, that is, they can have differing speeds, capacities and capabilities. For example, it may not be possible to execute all operations at all sites because of limitations on storage capacity. In addition, in an **extensible** DBMS environment, the local DBMSs at each site may have different type or method libraries. Full site autonomy implies that each site may add or remove new abstract data types, functions, operators, access methods and any other system extensions at any time. Furthermore, sites may do this without consulting or notifying any other site. It is possible to implement mechanisms that permit the propagation of definitions and methods to each site in a user-transparent fashion. However, such mechanisms would have to be made capable of dealing with substantial differences in machine architecture and system software. In addition, not all user extensions make sense on all architectures (e.g., functions implemented on vector or massively-parallel supercomputers must be run on particular machines in order to run efficiently). Finally, and most importantly, *requiring* sites to accept extensions (so that all sites are capable of running any portion of any query) would violate our assumptions about site autonomy. Hence, Mariposa does not guarantee the ability of all sites to process a given portion of a given query.

As a result, the major aspects of Mariposa are:

(1) a rule system comprising an engine and rule set

(2) fragment movement algorithms to allow mobile data

(3) a query optimizer and execution engine

(4) an approach to multiple copies for this environment

In the next four sections, we discuss each of these issues in detail.

3. THE MARIPOSA RULE SYSTEM

Mariposa contains a rule processing subsystem to drive query processing, manage storage allocation, and provide name resolution. Every Mariposa site runs a rule manager. It watches for events of interest and, when one is detected, takes the associated action. By using a rule processor to control these functions, Mariposa makes it easy to experiment with different policies. Conventional query processing and file management systems compile policy into the system, making it extremely difficult to change later. Mariposa allows processing and allocation strategies to be changed dynamically at any site in the network. The administrator need only express the new policy in a high-level rule language.

Mariposa uses a production rule system in which every rule is of the form:

on *event* do *action*

Events are delivered to the rule system by the storage manager (see section 4), the query processor (section 5), or by other outside agents. If the event is recognized, its action is executed. Both events and actions take a number of parameters, as shown below, which may be passed to other events and actions or used to take conditional actions.

The set of legal events is extensible, so that an administrator may define new events to support new policies. Mariposa implements a collection of built-in types of events to support query processing, storage allocation and name service. The built-in events are described below.

The action portion of a rule is written in a general purpose language such as Tcl [OUST90], a scripting language with escapes to C subroutines. In addition, there is a collection of built-in **primitive** actions.

In this section, we describe the events and actions that Mariposa requires as built-ins; additional user-defined actions and events can be defined for each site.

3.1. Built-in Events

Mariposa contains the following built-in events:

```
discover-class(class)
discover-fragment(fragment)
export-fragment(fragment)
import-fragment(fragment)
receive-rule(event, parameters, action, site)
receive-query(query, site)
receive-fragment(fragment, site)
receive-event(event, site, arguments)
```

The **discover-class** event is asserted by the query processing system when it needs to find information about some class in the database. For example,

```
discover-class(EMP)
```

returns the metadata for the EMP class, so that the query processor can formulate a plan to satisfy a user query over EMP. The metadata includes the class schema, a list of fragments composing the class, and the presumed location of each fragment. The metadata returned may be out of date. In this case, subsequent attempts to use the data as described may fail, and a new attempt to discover the current metadata must be made.

Consider the following rules that define how to locate a class:

```
on discover-class (*) do
  if (islocal($1)) then
    return(local-lookup($1))
  else
    return(deliver-event(discover-class,
                         nameserver, $1))

on discover-class (EMP) do
  return(deliver-event(discover-class,
                       big-server, EMP))
```

These rules have the following effect: Requests for metadata on the EMP class always go to the machine *big-server*. For all other classes, the local system is searched to see if the class is stored locally. If so, the metadata is found on this site. Otherwise, the request is passed on to the central name server.

The first rule implements centralized name service. The second uses explicit knowledge about the way the database is distributed to manage the EMP class. Other rules could be programmed to implement other sorts of name service.

The events

```
discover-fragment(fragment)
export-fragment(fragment)
import-fragment(fragment)
```

can be used to implement a name service and to control the location of the named fragment. The **discover-fragment** event for fragments works exactly as it does for classes. The **import-fragment** and **export-fragment** events control fragment placement. **Import-fragment** is asserted when the local site wants to fetch a copy of the fragment to store locally. **Export-fragment** is asserted when some fragment must be moved off of local storage.

This rule will cause the local site to overflow to a designated back-up site.

```
on export-fragment(fragment) do
  send-fragment($1, my-back-up)
```

A more sophisticated rule would be:

```
on export-fragment(fragment) do
  if size($1) < 1000
    send-fragment($1, my-small-back-up)
  else
    send-fragment($1, my-large-back-up)
```

The event

```
receive-rule(event, parameters, action, site)
```

is asserted whenever any site, whether local or remote, attempts to define a rule at the local site. This event permits the local Mariposa administrator to control the degree of local autonomy he is willing to surrender. The first three parameters correspond to the parts of the rule being defined, and the *site* parameter is the name of the site that is attempting to define the new rule. Hence, acceptance of an outside rule may be conditional on both the contents and the origin of the rule.

The **receive-query** event is automatically asserted when some other site tries to send a query to the local site. The action part of this event can control whether or not the query is processed here or passed on to some other site. For example, the rule

```
on receive-query (*, *) do
  if (data-is-local($1)) then
    return(execute($1))
  else
    return(send-query($1, data-site($1))
```

arranges for queries to be executed at the site storing the data that they touch. (In this example, the data-site function discovers the location of some fragment referenced in the query.)

More sophisticated query-acceptance rules must be defined in order to handle other processing constraints. These may include load balance constraints (CPU load, availability of sufficient storage and buffer space to execute a query, etc.) as well as data-dependency constraints (e.g., lack of the user-defined types or access methods required to process a given query). There can also be events that are triggered when an event is asserted at a site or when a fragment arrives at a site. This cascading of events provides a powerful way to structure rules.

3.2. Built-in Actions

In addition to writing special-purpose action routines in Tcl, the user may invoke any of the following built-in actions to handle an event.

```
send-rule(event, parameters, action,
          site-list)
send-fragment(fragment, site-list)
split-fragment(fragment, split-predicate)
merge-fragment(fragment, fragment)
send-query(query, site)
get-fragment(fragment, site)
deliver-event(event, site-list, arguments)
```

The **send-rule** action is the mechanism used to register a new rule in the rule-base at all of the sites in the network that must know about them. The *event* and *parameters* arguments specify the event to be detected. The *action* argument is code to execute when the rule is detected. The sites in *site-list* may choose to accept or reject the rule, depending on its origin or on other criteria. This permits Mariposa to support arbitrary degrees of local autonomy at each site in the network.

The **send-fragment** action attempts to export the fragment to each of the sites in *site-list* in turn. Since no site is required to accept the fragment, failure at one site means that the next site should be tried. Once the fragment is accepted by some site in the list, the action terminates. If no site accepts the fragment, then the action has failed; other steps must be taken to get rid of the fragment.

The **send-query** action passes the query to the named site for processing. Again, the site can refuse. However, it will be more common that it will pass the query on if it does not want to execute it.

The **get-fragment** action contacts the named site, and asks it to hand over the named fragment. The site may deliver the fragment or not.

The **deliver-event** action delivers the named event to each of the sites in *site-list* in turn. This is the technique used in Mariposa to do cooperative distributed computing; one site requests another site to do work on its behalf by sending it the appropriate event. This mechanism is similar to a remote procedure call service.

3.3. Extensibility

The set of events is user-extensible to permit experimentation with different policies for query processing and distributed data management. For example, the user might define an event **makespace** which would be triggered by an overfull condition of local storage. The action part of this rule would implement some policy for choosing a fragment to evict. Here is the complete rule that implements an LRU replacement algorithm to a specific back-up site.

```
on makespace() do
  send-fragment(oldest-local-fragment(),
                my-back-up-site)
```

The new makespace event will be asserted, and the least-recently accessed fragments exported, until the local site is no longer overfull.

3.4. Rule System Execution

Mariposa has been designed to allow individual sites in the network to choose the degree of autonomy they wish to retain. A site can enforce strict local autonomy by refusing to allow remote sites to define rules at the local site. Alternately, a site may allow remote sites to supply it with rules for its rule-base. This surrenders local autonomy. A combined strategy is also possible. Any site can accept or reject the incoming rule based on its origin or on other criteria.

It is possible for multiple rules in the rule-base to be applicable in any given situation. For example, there may be two different rules that can be activated when a fragment must be exported. The following **resolution** strategies are feasible in such situations:

- run randomly chosen rules until one succeeds
- run all applicable rules in random order
- run rules in priority order until one succeeds
- run all rules in priority order

In Mariposa, the services provided by the rule system are most appropriate for a policy that runs rules until one succeeds. For example, exporting a fragment will succeed at some point, obviating the need to run any more rules to make further export attempts. Also, priority order will give more determinism to Mariposa than random order.

The Mariposa rule engine implements a simple priority scheme that allows a fixed number of priority levels (16). Only one rule may apply on a given object at any priority level. Mariposa guarantees this by refusing to store a new rule if another rule at the same priority level already exists that conflicts with the new rule.

4. FRAGMENTS

As previously noted, each Mariposa class is composed of one or more **fragments**, each containing:

- the metadata for the class
- the data instances for the fragment
- any secondary indexes for the fragment

A fragment is fully self-describing so that it can be moved from site to site without dangling dependencies. Hence, all index records are maintained and stored with the fragment's data records.

In this section, we explain how Mariposa implements fragments and operations on fragments in a way that permits fast and efficient migration. However, before we do so, we must discuss some of the issues involved in identifying data objects and their impact on the efficiency of accessing such objects.

Every Mariposa fragment has a unique fragment identifier (FID). Splitting or coalescing fragments produces new FIDs for the resulting fragments. Every fragment records as part of its metadata the FIDs that describes its **lineage** (that is, its history of splits). The merge operation combines the lineage information from the fragments being merged.

Every instance of a class is identified by its record identifier (RID) that is unique within a fragment. In addition, every class has a class identifier (CID). The triple (CID, FID, RID) is globally unique, and constitutes a Mariposa **object identifier (OID)**.

Database systems commonly use more than one method of object identification as a tradeoff between efficiency and convenience. Physical OIDs (e.g., record addresses) allow fast access but they do not allow the record to be moved because that would invalidate the address. Logical OIDs (e.g., surrogates) have poor performance even if a hash table is used to map them to addresses. They do allow data to be moved but they typically require two disk accesses, though in the best case it is possible to reduce it to a single access.

A logical OID can be translated into a physical pointer, or **swizzled**, to gain performance in some cases. This operation is typically performed by OODBMSs when moving an object into memory from disk. The disadvantages of swizzling reduce its usefulness. Swizzling is not worthwhile if the overhead of doing so exceeds the time saved by using the faster pointers. Even if the data is read-only, the pointers must be modified through swizzling. Finally, the swizzled pointers must be **reverse-swizzled** when writing the object back out to disk. The ideal method of moving data fragments would therefore avoid (or defer) swizzling.

4.1. Fragment Location

Mariposa locates a fragment and its metadata using directives programmed in the Mariposa rule language of Section 3. Although it is possible to implement a global name server for the entire network, we expect that each site will store some amount of extra metadata and then rely on an intelligent search to find missing information.

When a site receives a fragment, it is responsible for storing the metadata in a collection of local **system catalogs** that can be efficiently searched by the local rule system. These catalogs contain the following information:

- fragment description
- index information
- protection information
- integrity constraints

While a fragment remains at a site, it is the responsibility of the site to ensure that the local copy of the fragment's metadata is correct. When a fragment leaves a site, that site has the option of discarding its copy of the metadata. A site can keep metadata for objects it no longer stores, but must recognize that this information might become obsolete. Hence, part of a rule to register a newly-received fragment might look like:

```
on import-fragment(fragment) do
  update-local-catalog($1)
```

Similarly, a site might mark an exported fragment's metadata as being possibly out of date in the following way:

```
on export-fragment(fragment) do
  set-forwarding-local-catalog($1)
```

4.2. Fragment Migration

Efficient fragment migration is central to the performance of the entire system in the same way that buffer management is critical to a conventional database system. Fragments are exported from a site if storage is overfull or if the administrator invokes a rule requesting that it be exported (as might be the case before shutting down a node). Fragments are imported if the query optimizer uses a “move the data to the query” strategy. Both actions are programmed using the Mariposa rule language. As noted earlier, fragments can migrate at any time, and no permission need be obtained to make storage allocation decisions.

A simple way to migrate a fragment is to first disable all activity to it so that there are no portions of it pinned or locked. However, waiting until all activity has quiesced for fragments will seriously impede the storage manager's ability to dynamically deal with over-allocation of space. We therefore develop algorithms that can move fragments which are not quiesced. Similarly, we remove the restriction that all fragments must remain whole so that the storage manager can split a fragment at any time if the fragment grows too large. The resulting two (or more) fragments can be independently accessed and migrated. All storage level operations such as splits, merges and moves are transparent at the application level.

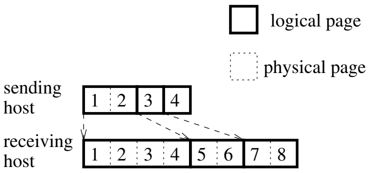
There are two additional problems that must be overcome during fragment migration: conversion of the fragment's data representation and dealing with migration of fragments during updates.

4.2.1. Mariposa Canonical Data Representation

Fragments are converted into the Mariposa canonical representation (MCR) while being migrated between sites. MCR is the common interchange representation to permit migration among heterogeneous sites. Every Mariposa site recognizes a collection of registered base data types that the site can convert to and from MCR. When defining new data types, the definer must specify an **import** and an **export** function as part of the registration process to perform the conversions. The exporting migrater invokes the data export functions to convert the records into the MCR format and the importing migrater invokes the import functions. Numerous optimizations are possible. For example, it is possible to adopt a “receiver makes right” style of conversion to take advantage of configurations that are likely to have homogeneous systems. Additionally, if a fragment is exported to a site that will serve as a repository, it will not always be necessary to first convert the data to MCR format.

Converting the records may result in no change to the page's size if the converted records do not consume all of the internal free space. Hence, if exporting does not change the length of any records, then blocks can be read and transferred efficiently to the network. Similarly, if importing does not change record lengths, then pages in MCR format can be directly stored on a recipient storage device.

Indexes present a special problem, because pointers in an index are physical record identifiers. On the target system, these pointers may not be valid. The next section indicates how Mariposa efficiently maintains indexes during migration.



(a) change in page layout due to a change in data size (identical physical page sizes)

sending host

logical page #	physical page #
1	1
2	3
3	4

receiving host

logical page #	physical page #
1	1
2	5
3	7

(b) translation tables

Figure 2. Translation tables for migration sites.

4.2.2. Index Conversion

Every Mariposa fragment has an associated translation table to translate the logical page pointers to the actual addresses. When the fragment is created or migrated the translation table is filled with the physical addresses of each of the pages. This table is used to locate individual pages whether they are in memory, on disk, or even not yet received over the network. This technique is different from pointer swizzling where pointers are modified in-place. Translation tables are more flexible than swizzling because they can be demand-filled and it avoids the re-swizzling costs if the fragment is moved, merged, or split.

A two-level strategy of page pointers and record pointers is used. The RID is composed of a page and record number that are individually translated when the fragment is migrated. All page pointer deferences use the local translation table to locate the page and then uses information stored on that page to get the byte offset of the record.

Figure 2 shows the contents of translation tables which give the conversion between the logical page pointer and the actual page pointer on the sending and receiving hosts.

All page pointers can remain as logical pointers when migrated, i.e., they remain the same regardless of their current storage location and can use the table to be correctly translated to the current physical value. Alternately, the pointers can be swizzled in place and a flag set to indicate whether the pointer is logical or physical. Translation tables have the clear advantages of changing the pointer value in only one spot (even if there are multiple pointers) and of avoiding an update to all data pages in order to write the swizzled pointer values, but also have the disadvantage of requiring an additional level of indirection. We intend to further compare the performance and complexity levels of both schemes.

4.3. Migration When Active

The Mariposa goal of allowing migration at any time means that a fragment can be migrated even while it is being concurrently updated by a different transaction. If the conservative policy of requiring total quiescence before migration was used, the power of migration would be enormously diminished.

There are a number of alternatives for implementing fragment movement, each permitting varying degrees of concurrency and freedom of movement. We consider the three alternatives of simple locking, sectional locking, and incremental migration.

The simplest possible method for fragment movement consists of waiting until all access to the fragment has been completed and then performing the entire movement. One merely obtains a write-lock on the entire fragment, copies it to the new storage site, then deletes the copy at the old storage site. This method is very easy to implement but permits no concurrency, i.e. no access is permitted while the fragment is being moved, and the move cannot begin until exclusive access can be obtained for the entire fragment.

One can trivially improve the simple locking algorithm by obtaining write-locks on **sections** of the fragment as they become available. As the sections are locked, they can be copied to the remote site. Once the entire fragment has been copied (and therefore write-locked), the old copy can be deleted. This scheme permits some read and write access to the fragment to occur concurrently with the fragment movement.

A third alternative that permits even more concurrency is that of incremental migration. This scheme employs the same idea as fuzzy dump algorithms or process migration schemes: the system does not lock any data, but instead copies clean (unmodified) pages to the new site first, followed by dirty pages. As additional pages are dirtied, they are also copied over. Finally, when the algorithm reaches a given threshold, all access is frozen, the remainder of the pages are copied and the old copy deleted. The incremental scheme may be more difficult to implement but permits unrestricted read and write access to the old copy of the fragment while movement occurs.

Allowing access to the old copy of the fragment to continue while migration occurs has an additional pitfall. A query (scan) may be accessing data and then discover that it cannot read a portion of the fragment. In this case the query must be halted and eventually restarted on the remote machine. This is not as hard as process migration, since the state of a database scan is far more easily encapsulated than a general operating system process, but nevertheless constitutes a significant complication. Each fragment contains a **fragment state** field that is updated when the fragment begins migration.

Mariposa will explore the performance of the three fragment migration algorithms to determine whether or not the gains in concurrency are worth the increased complications of the latter methods.

4.4. Migrating Logs

While the data can be freely moved, the support of active transactions is more problematical. Conventional databases use a write-ahead log to maintain information needed for undoing or redoing the effect of updates when there is a crash. Because the fragment may be moved incrementally to the new node and since in a distributed system each node can independently fail, there is a period when updates can occur on two nodes. Multiple updates to the same logical fragment can lead to a vastly complicated recovery manager. We argue that a simpler no-log approach is required to support the migration of fragments so that crash recovery is always a locally controlled operation.

Because fragments are migrated a block at a time, the log must be on the same granularity so that the log can be moved in conjunction with the data blocks. Mariposa will examine multiversioning algorithms that convert record updates into inserts so that failure recovery is simpler.

4.5. Fragment Lineage

Fragments can be recursively split into smaller fragments in response to changing system configurations or query execution patterns. To cause a split of fragment, the rule action **split_fragment($1, split-predicate)** is invoked where *split-predicate* specifies how the fragment is split. For example, a fragment could be split into ranges of an attribute based on the attributes value or it could be split according to some hash partitioning scheme. Fragmentation follows the Mariposa philosophy of allow all decisions to made locally. Since the split algorithm operates a locally greedy heuristic that proceeds based on local information, it may not produce the "best" set of fragments for the entire distributed system.

After a fragment is split, it is important that the fragment parts can be merged later to reform the original fragment if needed. A fragment's **lineage information** stores the history of splits for each fragment and guides the subsequent merges so that the operation can be performed by a site other than the one that split the fragment. The lineage of splitting fragments is logically tree-shaped but it can be represented as a simple log that is associated with each fragment since a fragment only needs to know about its parent and immediate child fragments. Merging can be constrained to only the reverse of the split operation so that only occur fragments at the same depth can be merged. In the most general case, any fragments of the same class can be merged if the strict hierarchical split lineage does not need to be preserved. Mariposa will not constrain the possible split and merge choices.

5. THE MARIPOSA QUERY PROCESSOR

In order to satisfy our demands for site autonomy and fully-distributed computation, we take a new approach to query optimization and query processing. We describe, in turn, the mechanisms by which Mariposa handles query parsing (initial preprocessing), query optimization and query execution. Finally, we briefly discuss how the Mariposa run-time system deals with schema changes.

5.1. Query Parsing

We assume the following model for query parsing and initial query processing. A query is delivered to Mariposa by a user at some site. This site becomes the **home site** for the query, responsible for both parsing the query and delivering the answer to the user.

In order to parse a query, the parser must find the metadata for each class C referenced in the query. The query processor at the parsing site asserts the event

```
discover-class(C)
```

to the rule system for each C, and the rule system returns whatever metadata it can obtain for C using the mechanisms described in Section 4.2.

In general, the information on-hand for all fragments of all classes referenced in the parse tree may be arbitrarily out of date. Since this includes the current location of fragments, the class schema and all class statistics, this has a critical impact on query optimization (as will be described below).

5.2. Query Optimization Issues

In addition to the standard problems associated with query optimization, the Mariposa query processor must deal with some additional problems and restrictions caused by the distribution and mobility of data fragments and the extensibility of the data manager. Some of these problems, previously discussed in Section 2, include:

- locating and processing data whose position is not fixed
- deciding on data movement vs. query movement
- replication
- load/capacity heterogeneity between computers
- type/method heterogeneity (extension library inconsistency)

Others include:

- harnessing the parallelism made possible by fragmentation
- permitting maximal distribution and autonomy during assignment of work to sites

We describe our solution to these problems in the subsections that follow.

5.3. The Query Processing Strategy

The Mariposa query optimization and query processing systems must operate within the constraints just described. We argue that traditional approaches to query processing do not have the flexibility required for operation in this environment and present our own query processing architecture that addresses this problem.

Traditional cost-based query optimizers, including nearly all optimizers found in commercial products, have been based on resource-usage models and exponential-complexity dynamic-programming search algorithms similar to those developed in System R [SELI79]. Extension of these traditional optimizers to handle distributed database systems, as in R* [SELI80, MACK86], is straightforward and produces optimal plans. However, this approach has two key disadvantages. First, the exponential complexity of the search space makes the use of such optimizers impractical in very large distributed systems. Second, the query plans generated by such optimizers explicitly specify the location of each data fragment to be accessed and the host on which each query processing operation (e.g., joins) should run. Since data can move frequently in Mariposa, re-optimization of invalidated query plans would quickly become very costly.

In order to reduce the sensitivity of the optimization process to changes in database configuration, Mariposa considers conventional optimization factors (such as join order) separately from distributed system factors (such as data layout and execution location). Hence, the Mariposa query optimization algorithm is a multi-pass process. We now describe our algorithm.

5.3.1. Three-Phase Optimization

Our basic approach is to generate query plans at compile-time that completely disregard the current location of the data, and make decisions with respect to the degree of parallel execution, data movement and query execution sites at execution-time. Although the initial use of "single-site" optimization may often generate suboptimal plans, as pointed out in [MACK86], we believe that other strategies are impractical in our environment. Other strategies require an arbitrarily large amount of current global knowledge regarding fragment composition and location. This use of standard optimizer technology as a basis for subsequent refinement is similar to that proposed in [HONG91].

In the discussion that follows, the word **site** generally indicates the Mariposa query processing engine running on a particular computer. The words **query plan** and **query tree** are used interchangably. As with many query processing systems, Mariposa implements query execution plans as trees of nodes corresponding to the set of query processing operations (scans, joins, etc.). Record-based dataflow occurs along the branches. Such trees can be trivially decomposed into subtrees, each of which describes the execution of a portion of the original query (subquery). These trees can be encapsulated for transmission between hosts.

Briefly, the Mariposa query optimizer on the home site conducts a first optimization pass that *compiles* the query string into an initial query plan without considering distribution. Immediately before execution, the home site optimizer *parallelizes* the result of this first pass. At execution-time, Mariposa *selects sites* on which to run each query processing operator (scans, joins, etc.) in a distributed, top-down manner. Sites make decisions to accept or reject responsibility for handling portions of a query by asserting the *receive-query* event in their rule system. As sites accept responsibility for running portions of a query, they also assume responsibility for selecting sites to run the portions of their own subqueries.

A slightly more detailed sketch of the algorithm is as follows:

(1) *Compilation*: This phase takes a query parse tree as its input and produces a binary query plan tree.

The home site constructs a **locally** optimal plan by running a local optimizer over the query, assuming in its cost calculations that all data is local to the home site. At this point, the query execution plan tree is simply a standard binary operator tree. This step fixes such items as join order and the application of join and restriction clauses.

As mentioned above, the local optimizer builds query execution plans using the metadata provided by the rule system. The fact that this metadata may be out of date has two possible effects. First, inaccurate class statistics (e.g., number of records, overall size, selectivity information) will have exactly the same effect as they would on a non-distributed optimization — the plan produced may deviate from the optimal plan. In this case, however, the plan is still usable. Second, the class schema may be incorrect (i.e., attributes may have been added). In this case, the plan is, in general, not usable. This run-time system can easily detect this situation using schema version numbers and abort the execution of the plan by notifying the home site.

Note that this first phase can consist of any optimization method proposed for single-site database management systems that produces (or can be modified to produce) a binary query operator tree. Recent research demonstrates that bushy query plan trees permit substantial gains in performance in both serial (e.g, [IOAN91]) and parallel (e.g., [GANG92]) database systems. We intend to experiment with a number of different optimization strategies (tree topology families) to determine which method provides the the best results in the Mariposa environment.

(2) *Parallelization*: This phase takes a binary query operator tree as input and produces a non-binary query plan tree.

Immediately prior to execution, the home site determines the degree of intra-operator parallelism required for various subtrees of the plan tree and inserts **collector** nodes throughout the plan. Collector nodes serve two purposes. First, they coordinate the execution of the portion of the query plan tree that lies immediately below them. Second, they consolidate the results produced by the query plan tree nodes immediately below them. For example, a collector node may have several child scan nodes. The results from those scans are fed into the collector node and piped up to the next level in the query plan as a single stream.

The method by which collector nodes are generated will be described later. For now, it suffices to say that this phase fixes the degree of parallelism used throughout the plan tree and introduces the nmechanism (collector nodes) by which Mariposa controls parallel execution.

(3) *Site Selection and Execution*: This phase takes a parallelized, non-binary query plan tree, distributes the tree nodes among the various Mariposa sites and executes it.

Again, the actual mechanism by which the query operator nodes are distributed will be described later. However, the selection of a site to execute a particular node is simple in concept. The initial selection of a site to handle a particular scan or join node will be determined by the location of the required data fragment, though other sites may be considered. Any site considered must be able to conform to a number of **constraints**. As discussed previously, these constraints include both load-balance considerations as well as data dependencies. These constraints will be tested when a site's rule engine receives a *receive-query* event; if all of the applicable constraints are not met, the site refuses the query and another site must be found. If no site will accept a query, the query must be aborted.

Since site-selection occurs in a top-down fashion, execution can begin immediately after sites have been selected for the bottom-level operator nodes.

In the remainder of this section, we show how this three-phase approach applies to single-class scans and two-class joins. We then present the general algorithm for generating query plans for multi-class join queries.

5.3.2. Single-Variable Scans

The most basic query processing operation is the single-variable query (single-class scan). However, in Mariposa, a simple scan over a single class may become very complex because the data may be distributed over several sites and the plan may be executed using several parallel threads. Mariposa uses collector nodes to control both the flow of results and thread execution.

The local optimization phase will simply generate a single scan node. For clarity, we will not consider such issues as the use of indexes or sorting of results in this example.

During the second phase, the optimizer turns the single scan node into a collection of parallel scans whose execution is coordinated by a collector node. In general, a scan will be on f different fragments. Such a scan will have f parallel threads of execution for the scan, one per fragment. Hence, the query plan tree changes from a single node to a f-way tree — one collector node with f child scan nodes.

In the third phase, the Mariposa query processing engines cooperate in distributing the query plan nodes among various sites. The query's home site must first find a site (most likely the home site itself) to accept responsibility for executing the coordinator node. This site must, in turn, find f sites (possibly non-distinct) to accept the f scan subqueries. Since these f sites will be the various sites that actually store the required fragments, they can be found by (1) asserting a *discover-class* event to determine the number of fragments and their respective FIDs and then (2) asserting a *discover-fragment* event for each of the fragments. Execution begins once each of these bottom-level nodes have been assigned to Mariposa sites.

When execution begins, the collector node must make the following decision for each scan thread: the collector can either import the associated fragment to its own site, executing the scan locally (data to the query), or it can execute the scan on the site currently storing the data (query to the data). In the end, of course, all of the scan *results* must be collected on the site on which the collector node resides (after which the results are delivered to the user at the home site). However, the location at which Mariposa performs the actual fragment scan may or may not be that collector site.

In the single-class scenario, then, the key question is how the run-time system makes the import data/export query decision for each of the fragments. The remainder of this subsection describes how this decision is made.

Each Mariposa site keeps two sets of access statistics. First, each site S keeps a **reference history** for each local fragment (fragments stored at S). This history consists of two time-decaying reference counts. One reference count is for purely local accesses (scans requested by S) and the other is for remote accesses (scans requested by all other sites other than S). In other words, the latter reference count will be incremented when some site decides to "move the query" to the given fragment. Second, each site S keeps a time-decaying reference history for fragments that it has accessed on sites other than itself. Hence, this reference count is incremented when S decides to "move a query" to a remote data fragment.

The import/export decision is made as follows. At execution-time, the query processor at the collector node's site S_c asserts an *import-fragment* event in its local rule system. The local rule system delivers this event to the remote rule system along with the local reference history (if any) for that fragment. At this point, there are two possibilities, depending on whether S_c had any local reference history for the fragment:

(1) *No local history available*: If S_c has not made any (recent) previous queries on the fragment under consideration then the import request will fail — the remote site will hold onto the fragment because there is no compelling reason to move the data to the query. In this case, the query processor at S_c passes the current query to the site that stores the fragment using *send-query*.

(2) *Local history available*: The storage site compares S_c's reference count to its own local reference count for the fragment. If S_c's reference count exceeds the local count by a (tunable) threshold, then the import request will succeed (that is, the remote site will initiate a *send-fragment* action).

This approach has the benefit of simplicity and flexibility. The decision to import the data is a joint decision of the query processor and the storage rule systems on the respective sites and depends on factors. Furthermore, the decision factors can be tuned by changing weighting parameters and policies within the rule system. In addition, because the data/query movement decisions are be made at execution-time, plans generated by this method are never invalidated by out-of-date fragment location information.

Mariposa assumes that all network connections have equivalent bandwidth and latency. This is not to say that all hosts must be on the same local area network or a set of completely homogeneous networks, but rather that the networks are all "fast" in the sense discussed in [MACK86], such that network bandwidth does not overwhelmingly dominate query processing costs. If we can make the simplifying assumption that hosts are essentially equidistant, then importing a fragment from a given remote host costs the same as importing the fragment from any other remote host. This is why Mariposa only considers fragment access to be "local" or "remote" and does not distinguish between remote hosts.

5.3.3. Two-Class Joins

The basic join operation is the simple two-class join. Mariposa implements both nested-loop and hashed join methods. In both cases, the first optimization phase produces a standard two-way join plan: a join node with two scan nodes as children.

The second phase replicates the query plan nodes to reflect the number of fragments into which each class is divided, producing a multi-way tree. This tree will look different for the two join methods, as described below.

(1) *Nested-loop join*: A nested-loop join can be naturally decomposed into f_{outer} parallel joins, where f_{outer} is the number of fragments in the outer join class. This, in turn, essentially turns into f_{outer} separate scans of the inner join class. The collector nodes for the inner scans are merged with the nested-loop query plan nodes and perform the join with the records in the outer join class fragment. Hence, there will be f_{outer} results at f_{outer} collector nodes, which are then collected into a final result at the home site.

Although it would be possible to implement nested-loop join using a single parallel inner scan (with a single collector node), the unit of import would then be the entire inner scan result. By replicating the parallel inner scans, the collector nodes can choose to import individual fragments of the inner class.

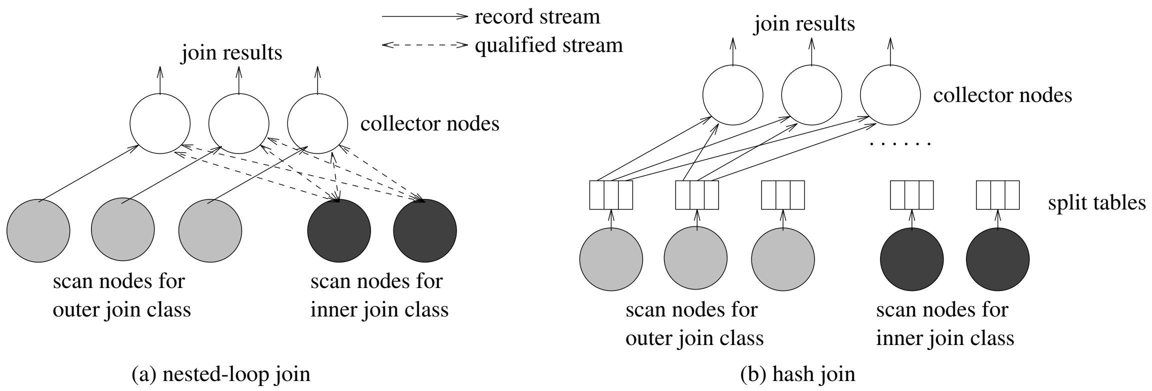


Figure 3. Join method control and dataflow structure.

(2) *Hash join*: Since we cannot assume that any given hashed access method will exist on all of the fragments, we assume that hash join executes as two separate stages: hashing the classes into b buckets (using split tables to route records to the correct bucket site) and then performing joins of corresponding buckets in parallel. This requires b collector nodes, which are again merged with the hash-join query plan nodes, for the results of the b subjoins. For the purposes of our discussion, we assume $b = f_{outer}$.

Figure 3 shows the relationships between the fragment scan nodes and the collector nodes. In these examples, $f_{outer} = 3$, so there are three fragments in the outer join class, three collector nodes, and three streams of results. Although each node is shown as a separate entity, any or all of the nodes may be co-located on the same host. Hence, the flow of records from one node to another may be local rather than over an actual network connection.

During the third phase, Mariposa distributes the nodes among sites for execution. Just as with single-variable queries, the home site selects other sites to execute the collector nodes. The sites on which the collector nodes execute then find sites that will accept its child scan nodes. Finally, each collector site S_c coooperates with the storage sites to determine whether fragments will be imported to S_c or the scan will be run on the storage site. Figure 4 shows how a nested-loop join might be executed on four sites.

This method of producing query plans for parallel two-way joins suggests a very natural method for producing plans for parallel multi-class joins.

5.3.4. Multi-Class Joins

Our algorithm for decomposing query trees with multiple joins is both fully distributed and compatible with site autonomy. Having filled in details on the handling of one and two variable queries, we expand on our previous discussion of parallelization and site assignment.

As in the last two subsections, the local optimization phase produces a simple binary query plan tree.

When the parallelization phase begins, the home site determines a *threading value* for the left-deep portion of its query tree. This threading value, t, is the number of parallel threads that will be executed for each join along the left-deep portion of the tree. That is, each of those joins will produce t streams of result records. This means that every join along the left-deep portion of the tree will have the same number of fragments in its outer join class (or input streams from the join immediately below it). We give t the value of f_{outer} of the leftmost bottom join node.

A value for t must be determined for each left-deep path in the query tree. In bushy trees, there may be more than one left-deep path.

The home site then parallelizes the joins and scans. Parallel joins and scans are generated as described in the previous two subsections. Again, along a given left-deep path, each join will have the same number of outer join fragments.

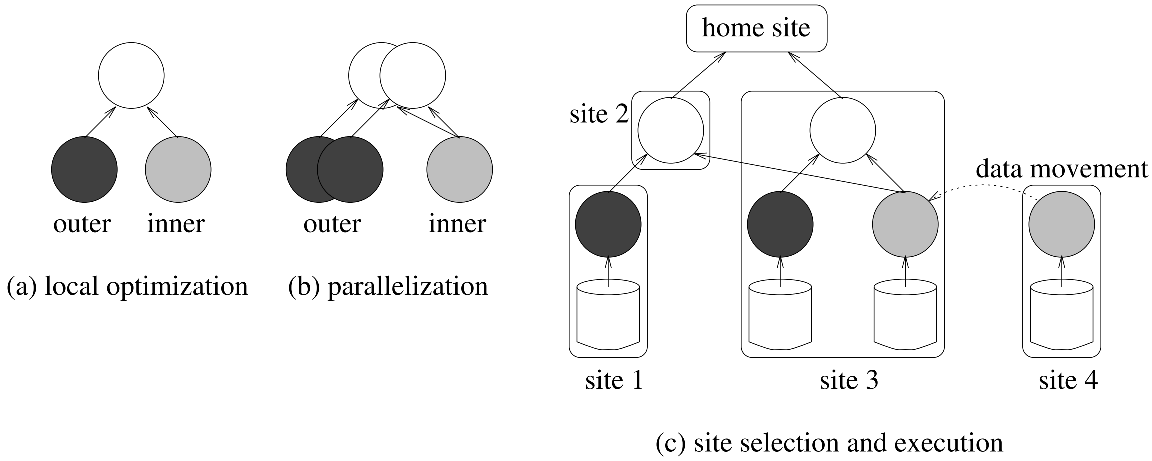


Figure 4. Query optimization and execution.

During the third phase, the home site takes the query plan tree and begins distribution of its nodes. The parallel query plan for a multi-way join will have a set of collector nodes at the top; the home site must find sites that will execute the top-level collector nodes. These sites must then find additional sites that will execute the child nodes as described in the last two subsections. This process continues recursively until all nodes are assigned to sites, and execution begins when all nodes are in place. Decisions to import fragments are also made at this time.

It should be noted that the choice of a single value for t in a given left portion of a tree is completely arbitrary. Setting t to be the same at each level in a left-deep subtree allows us to avoid writing results to disk after a join since we can trivially pipe the results to the next join (unless the result is used as an inner join class, in which case it must be written to disk if the join is a nested-loop). On the other hand, parallelism can be increased by allowing more than one thread on an outer join class fragment. Parallelism can be decreased by having threads manage more than one fragment.

5.4. Handling Schema Changes

The query processing algorithms described above must be able to deal with arbitrary

fragment splits
fragment combinations
schema modifications

that may occur between the time that the local metadata was created and execution-time. We propose to deal with these problems in the following manner.

If a host H receives a query Q for some fragment F, it first checks to see if it has F. If so, it processes the query normally. If it does not have F, then it looks in its log to see if it split or combined F. If it split F, then it constructs 2 queries, one for each fragment, and deals with each as an incoming query.

If H combined F with another fragment, it changes Q to the new enclosing fragment, F' and deals with $Q(F')$ as an incoming query. The only exception to the above rule is if the host has already issued $Q(F')$. In this case, it simply deletes the (now redundant) query. It can discover this fact by inspecting the fragment's history.

If a host receives an *import* command, then it responds as above by decomposing the command into two *import* commands (for a split) or one enclosing *import* command (for a combination).

The owner of a class can change the schema for a fragment of the class at any time. A log must be maintained of this action. In this case, when a query $Q(F)$ arrives, it must be tagged with a **schema id** and must include the raw query language command. If a mismatch occurs on schema ids, then the receiving site must recompile the query.

6. MULTIPLE COPIES

Mariposa supports a variable number of copies of each fragment, and builds on the concepts used traditionally by file systems. When a file block is moved from disk to main memory, the file system makes a copy of the block in main memory. The disk copy continues to exist but is not writable. Rather, the main memory copy can be read or written. If the main memory block is dirty at the time that it is discarded from the cache, then the file system overwrites the corresponding disk block, otherwise it discards the main memory copy. In this way, a temporary copy of a block is constructed and then discarded when no longer needed. In this section we generalize this concept for Mariposa.

When a fragment is moved from one site to another, the receiving site can declare the move to be one of three kinds:

(1) An actual data move. In this case the sending site can discard its copy once the receiving site has safely accepted the fragment. The fragment at the receiving site has the same status (read only or read/write) as it had on the sending site.

(2) A normal copy. In this case, the receiving copy marks its copy as "read only." The copy at the sending site continues to have the same status as previously.

(3) A reversed copy. In this case, the sending site retains its copy of the fragment but marks it "read only." The copy at the receiving site inherits the status previously held by the copy at the sending site.

In this way, exactly one copy of a fragment can be updated, and the remainder are read only. A site with a read-only copy can send a message to the site with the updatable copy to **exchange** privileges. In this way, updatability can be passed between copies.

When a read command is received, it can be directed to any fragment. When a write is received, it must be processed at the fragment with updatability status. Then, inside the transaction of the writer, all other copies must be interrogated. Specifically, the update must be sent to each site with a copy, and each can either accept the update and modify its copy or discard its copy.

It can be noted that this scheme is basically a primary copy algorithm [STON79] extended to allow the number of copies to be (perhaps rapidly) varying.

Again, the meta data must include the location of each copy, and it may be arbitrarily out of date. If a write of a fragment is sent to a site without the updatable copy, then the site can either import the updatability capability or it can export the command to another site. This decision can be made using the same techniques discussed earlier concerning whether to export the query or import the data. Lastly, the updating site must be able to find all copies of the fragment so it can propagate any update. It uses its guess for the locations of copies to send out update messages to these sites. If the site does not have the copy, it forwards the update onward to the site where it moved the fragment. Likewise, if the site has participated in the construction of an additional copy, then it splits the update message into two, processes one and passes the other one onward. If it deleted the copy, then it simply sends back a negative acknowledgement. Ultimately all copies are found and they process the update.

When the query optimizer decides to import a fragment, it must decide which site to import it from and whether to make a copy or perform a move. These decisions can be easily made by the action part of rules, constructed for this purpose.

7. CONCLUSIONS

We have presented the architecture of Mariposa, a prototype data management system that unifies the best features of distributed operating system and distributed database management system research. Unlike previous distributed data management systems, Mariposa manages object storage in an environment of high data mobility and highly heterogeneous system capabilities while retaining high performance. In addition, Mariposa's rule-based storage architecture preserves local site autonomy and gives site administrators considerable flexibility in specifying their storage policies.

Mariposa will serve as a "workbench" for analyzing various algorithms for database and file systems. Distributed query optimization has been identified as an area that will receive a strong emphasis and we will also examine how to build a system that has a rule system at its core. We have begun implementation of Mariposa as an extension of the POSTGRES next-generation DBMS [STON86c] and expect to be able to present results from a working prototype in approximately one year.

Future work remains in the areas of system robustness, distributed failure recovery, and performance assessment. The analysis of these important large system issues will be addressed during the construction of Mariposa.

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. Database Sys.*, 6(4), Dec. 1981.

[EPOC92] Epoch Systems, "EpochServ Software Release Notes (Rel. 5.1)," Doc. 64-001614 Rev 01, Epoch Systems, Inc., Westborough, MA, Sep. 1992.

[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.

[GANG92] Ganguly, S., Hasan, W. and Krishnamurthy, R., "Query Optimization for Parallel Execution," *Proc. 1992 ACM-SIGMOD Conf. on Management of Data*, San Diego, CA, June 1992.

[GENE91] General Atomics (DISCOS Division), "The UniTree Virtual Disk System: An Overview," DISCOS Technical Report, General Atomics Corp., San Diego, CA, 1991.

[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. Comp. Sys.*, 6(1), Feb. 1988.

[IOAN91] Ioannidis, Y. E. and Kang, Y. C., "Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization," *Proc. 1991 ACM-SIGMOD Conf. on Management of Data*, Denver, CO, May 1991.

[KOHL92] Kohl, J., Staelin, C. and Stonebraker, M., "Highlight: Using a Log-structured File System for Tertiary Storage Management," Sequoia 2000 Technical Report 92/16, University of California, Berkeley, CA, Nov. 1992.

[LIND83] Lindsay, B. G., Haas, L. M., Mohan, C., Wilms, P. F. and Yost, R. A., "Computation and Communication in R*: A Distributed Database Manager," *ACM Trans. Comp. Sys.* 2(1), Feb. 1984.

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

[MACK86] 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.

[OLSO93] Olson, M., "The Design and Implementation of the Inversion File System," *Proc. Winter 1993 USENIX Conf.*, San Diego, CA, Jan. 1993.

[OUST90] Ousterhout, J. K., "Tcl: An Embeddable Command Language," *Proc. Winter 1990 USENIX Conf.*, Washington, DC, Jan. 1990.

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

[SELI80] Selinger, P. G., and Adiba, M. E., "Access Path Selection in Distributed Data Base Management Systems," *Proc. 1980 Int. Conf. on Data Bases*, Aberdeen, Scotland, July 1980.

[SKEE81] Skeen, D., "Crash Recovery in a Distributed Database System," Ph.D. thesis, Dept. of EECS, Univ. of California, Berkeley, CA, 1981.

[STON79] Stonebraker, M., "Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Software Eng.* TSE-5(3), May 1979.

[STON86a] Stonebraker, M., "The Design and Implementation of Distributed INGRES," in *The INGRES Papers*, M. Stonebraker (ed.), Addison-Wesley, Reading, MA, 1986.

[STON86b] Stonebraker, M., "The Case for Shared Nothing," *IEEE Database Engineering* 9(1), Mar. 1986.

[STON86c] Stonebraker, M. R. and Rowe, L. A., "The Design of POSTGRES," *Proc. 1986 ACM-SIGMOD Conf. on Management of Data*, Washington, DC, Jun. 1986.

[WILL81] Williams, R., et al., "R*: An Overview of the Architecture," IBM Research Report RJ3325, IBM Research Laboratory, San Jose, CA, Dec. 1981.