MOCHA: A Self-Extensible Middleware Substrate for Distributed Data Sources*

Manuel Rodríguez-Martínez
Department of Computer Science
University of Maryland
College Park, Maryland 20742
`manuel@cs.umd.edu`

Nick Roussopoulos
Institute for Advanced Computer Studies &
Department of Computer Science
University of Maryland
College Park, Maryland 20742
`nick@cs.umd.edu`

Abstract

This paper describes MOCHA, a self-extensible middleware substrate designed to interconnect data sources distributed over a computer network. MOCHA is designed to scale to large environments and is based on the idea that the functionality in the system should be deployed by the middleware itself. This is realized by shipping the code implementing either advanced data types or tailored query operators to remote data sources and have it executed remotely. Optimized query plans push the evaluation of powerful *data-reducing* operators to the data sites while executing *data-inflating* operators at the client's site. The Volume Reduction Factor is a new cost metric introduced to select the best site to execute query operators and is shown to be more accurate than the standard selectivity factor. MOCHA has been implemented in Java and runs on top of the Informix Universal Server. In this paper we present the architecture of MOCHA, the ideas behind it, and a performance study using data and queries from the Sequoia 2000 Benchmark. The results of this study demonstrate that MOCHA not only provides a flexible and scalable framework but also substantially improves query performance in contrast to traditional middleware solutions.

---

*This work was supported by the DOD Lucite Contract CG9815.

1 Introduction

Mediator-based systems provide middleware for integrating collections of data sources distributed over a computer network. These systems provide users with a unified and uniform view of an otherwise heterogeneous collection of data items. Typically, the client application contacts a *mediator* element that provides the basic query processing services in the system. The mediator utilizes the functionality of *wrappers* to gain access to the information stored in each data site. The wrapper element extracts information from a data site and transforms this data from the local data model to a global data model that is specified by the mediator. This global data model can either be relational, object-oriented or object-relational, depending on the needs of the targeted applications. The data types defined under the global model are contained in libraries that must linked to the mediator, wrappers and client applications. Usually, mediators are run close to the client application, on what is called the *integration site*. The wrappers can be run either on the integration site or on the site where the data resides. Systems such as TSIMMIS [CGMH$^+$94], DISCO [TRV96] and Garlic [RS97] follow this architecture.

In the light of recent technological advances in connectivity it seems necessary to reconsider whether the existing middleware solutions can provide adequate support for emerging distributed applications. With the continuous growth of the Internet and the ever increasing number of corporate Intranets and Extranets, middleware will be required to interconnect hundreds or more of data sites deployed on these networks. Many of these sites will contain very large data sets with complex data types such as images, geometric objects, and even programs. More and more of these data sets will be manipulated through an extensible object-relational database server through which it is possible to add new user-defined types and operators (predicates, projections and aggregates). Furthermore, with the emergence of the World Wide Web as the de facto user interface for networked applications, end-users will expect and demand a middleware solution that efficiently integrates with Web-based clients.

Evidently, in order for middleware-based systems to succeed in these kind of environments they must:

- provide scalable, efficient and cost-effective mechanisms to deploy and maintain the functionality used to implement the complex data types and operators in the system.
- provide adequate query processing capabilities to efficiently execute the queries posed by the users.
- easily integrate with Web-based technologies, such as Java, and allow Web clients become "first class" citizens.

Current middleware solutions do not satisfy the requirements mentioned above. We argue that the current mechanisms used to deploy functionality cannot scale to an environment with hundreds or more of data sites. Furthermore, we show that, in many important cases, the query processing framework currently provided by mediator systems is not flexible enough to provide an efficient query processing service. In fact, in many occasions the worst possible plan is used to execute a query. Although current mediator systems provide interoperability among several heterogeneous data sites, and, in many cases, can be tied up with a Web-based interface, none of the above mentioned requirements are satisfied and therefore, they cannot provide a viable solution for large scale distributed systems.

To correct these problems, we propose MOCHA (**M**iddleware Based **O**n a **C**ode S**H**ipping **A**rchitecture), a middleware substrate designed to interconnect hundreds of data sites. MOCHA is based on the notion that the middleware for a large scale distributed environment should be *self-extensible*. A self-extensible middleware system is capable of deploying new functionality to remote sites in the system in an autonomous fashion. This can be realized by *shipping* the code containing new capabilities to the remote sites, and having them added to their environments. The shipped code can be either tailored for a given complex data type, or for a user-defined query operator such as a filtering predicate, a generalized projection, or an aggregate function. One goal behind this notion of autonomous code deployment is to fill-in the need for application-specific processing components at remote sites. These processing components can migrate from site to site and become available for immediate use.

Furthermore, autonomous code deployment is fundamental in achieving *efficient query processing*. By shipping code for query operators, we can generate plans that place the execution of powerful data-reducing operators (data "filters") on the sites where data resides. Examples of such "filters" include aggregates, data mining and image processing operators, which return a much smaller abstraction of the original data. On the other hand, data-inflating operators that generate results larger than their arguments should be evaluated at the integration site or the client site. Since in many cases, the code is a lot smaller than the data sets, autonomous code deployment facilitates query optimization based on data movement reduction, which in turn can have significant impact on query execution time.

In this paper we describe a prototype implementation of MOCHA written in Java and running on top of the Informix Universal Server (IUS). We argue that a self-extensible middleware substrate such as MOCHA is easier to maintain than traditional middleware solutions. We introduce a new metric for measuring the volume reduction in data movement gained by pushing operators to the data sources. We will then show through experiments that when compared with the processing schemes in current middleware solutions, the query processing framework proposed in MOCHA can substantially improve query performance by a factor of 4-1, in the case of aggregates and predicates, or 3-1 , in the case of projections. These experiments were carried out on the MOCHA prototype, using data and queries from the Sequoia 2000 Benchmark [Sto93].

The remainder of this paper is organized as follows. Section 2 amplifies the shortcomings of existing middleware solutions and motivates the need for MOCHA. Section 3 discusses the architecture of MOCHA and our solutions for the problems discussed in the section 2. Section 4 contains a performance study of the MOCHA prototype. In section 5, related work is briefly discussed. Finally, our conclusions are presented in section 6.

2 Motivating Problems

There are two major obstacles in designing and implementing large scale distributed database applications: deployment of functionality and efficient processing of queries.

2.1 Deployment of Functionality

We argue that the mechanisms for adding (user-defined) functionality to existing mediator-based systems cannot scale to hundreds of sites, and is certainly unthinkable for an even larger scale environment such as, for example, the Internet. In mediator-based systems, in order to extend system functionality, the classes, structures, methods and configuration files that contain the implementation of new data types, and query operators are collected into libraries which must be installed at each site where a mediator, wrapper or client resides. We contend that as the number of sites in the system increases, the current scheme becomes impractical because of the complexity and cost involved in maintaining the software throughout the entire system. We describe these problems with an example from the Sequoia Benchmark [Sto93].

Example 2.1 An Earth Science application manipulates data stored and maintained in sites distributed across the United States. There is one data site per state, and each site holds information that earth scientists gather from specific regions within the state. The system contains the following relational schema:

```
Polygons(landuse:Integer,location:Polygon);
Rasters(time:Integer,band:Integer,location:Rectangle,image:Raster);
```

Table `Polygons` contains polygons covering land regions of homogeneous type. Attribute `landuse` identifies the land type (i.e. city, forest, etc.) and attribute `location` contains the exterior vertices of the polygon. The second table, `Rasters`, stores raster images containing weekly energy readings gathered by satellites orbiting the Earth. Attribute `time` gives the week number for the reading, `band` gives the energy band measured, `location` contains the rectangle covering the region under study and `image` is the raster image itself. □

To implement the schema in Example 2.1 using the traditional middleware approach it would be necessary to install the libraries containing the code (most likely C/C++ code) for the `Polygon`, `Rectangle` and `Raster` data types on each site where a wrapper, mediator, or client can possibly run. This translates into at least fifty installations for the wrappers, plus as many more as are necessary for the mediators and client applications. In all likelihood, the developers or the administrators in charge of the distributed applications will have to access each site and manually perform these installations. Alternatively, each site might agree to implement the libraries using their local developers. Whatever method is used, coordinating the development process, the access to the server sites and guaranteeing that the functionality is implemented correctly by all developers appears to be a daunting task. Undoubtedly, the process will be slow and error prone due to the number of parties involved. As the number of involved sites increases, the administrators are faced with the problem of maintaining a large number of library versions and upgrades. They must keep track of where libraries have been installed, what versions of the libraries are in use at each site and where do upgrades must be performed.

To complicate matters further, it is often the case that the functionality has to be ported to different hardware and operating system platforms. This leads to a longer development cycle, since extra effort must be invested in making the functionality work consistently across platforms. The complexity and cost of this factor may not be evident until the implementation process is in full gear. Evidently, the logistics in such a large scale system are formidable for the conventional middleware approach.

2.2 Efficient Query Processing

The second problem with conventional middleware solutions is their rather rigid query processing framework. In the majority of the systems, the queries are completely evaluated by the mediator element. The only task for the wrappers is the extraction of the data values from the data sources and their mapping to the mediator values for further processing. In many situations, this framework can result in the use of the worst possible query plan to solve a query, namely fetch a-record-at-a-time and process it. We demonstrate this with the following example.

Example 2.2 In the data site for the State of Maryland, table `Rasters` contains 200 entries. In this table, attributes `time` and `band` are 4-byte integers, attribute `location` is a 16-byte record and attribute `image` is a 1MB byte array. A user in the State of Virginia poses the following query to the data site at Maryland:

```
SELECT  time, location, AvgEnergy(image)
FROM    Rasters
WHERE   AvgEnergy(image) < 100
```

This query will retrieve the time, location and average energy reading for all entries whose average energy reading is less than the constant 100. The result of function `AvgEnergy()` is a 8-byte double precision floating point number. □

To evaluate the query in Example 2.2, the conventional mediator system approach will ship the attributes `time`, `location` and `image`, contained in every tuple in table `Rasters`, from the data site in Maryland to the mediator site in Virginia. Then, the `AvgEnergy()` function is evaluated, the qualification clause in the query is tested and the projections are performed. This approach for query processing is often called *data shipping* [FJK96] since the data needed to solve the query is moved from its source to the site where the query has been posed. Now, consider the cost of the operation that we have just described. We are moving roughly 200MB worth of data over a wide-area network which will take several minutes or even hours to complete since the bandwidth available in most wide-area links is very limited (often under 1Mbps).

Notice, however, that the size of the result for the query of Example 2.2 will always be much smaller that the size of the data set from which it is derived. In the worst case scenario, all the tuples in table `Rasters` satisfy the qualification clause, but the result contains 200 tuples for a total size of 200x28 bytes, or approximately 5KB. Therefore, a better option is to move the evaluation of the qualification clause and the projections to the data source site, and then move the final result of 5KB to the mediator site, in just a few seconds to complete.

Unfortunately, pushing the evaluation of query operators to the data source site is not considered by most mediator systems since all of the query processing occurs at the mediator site. There are some systems that can provide a limited form of support for such a plan though. For example, Garlic [RS97] will push the evaluation of the qualification clause and the projections to the DBMS running at the data source, but this would only happen if function `AvgEnergy()` were already implemented in the DBMS. In general, however, it is highly improbable that all the functions ever needed to evaluate a query will be present in each data source. Similarly, TSIMMIS [CGMH$^+$94] provides the capabilities to embed such a function like `AvgEnergy()` into the wrappers. However, it is usually impractical to add all possible functions into all the wrappers in the system for the reasons discussed in section 2.1.

3 MOCHA Architecture

We have designed the architecture of MOCHA to correct the shortcomings of the existing middleware solutions. As mentioned before, MOCHA is built as self-extensible middleware, where code is deployed to the remote sites in the system in an autonomous fashion. In our view, this framework provides the foundation for a scalable middleware solution. We now elaborate on the principles and design choices for MOCHA.

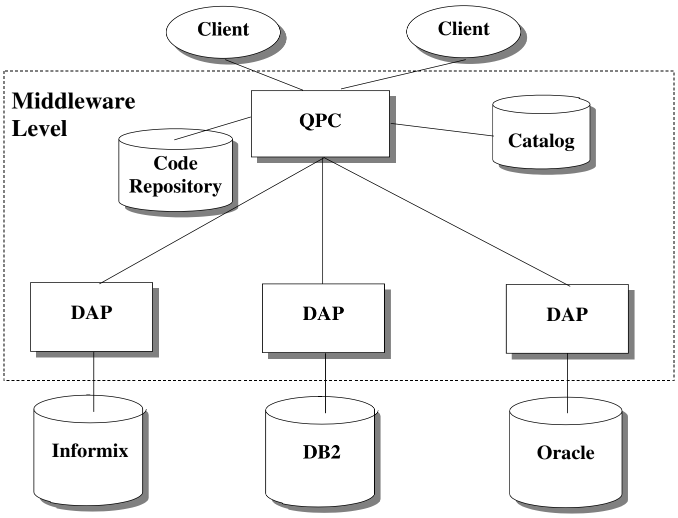


Figure 1: MOCHA Architecture

Figure 1 depicts the organization of the major components in MOCHA. We have implemented the MOCHA prototype using Java and it is based on the *distributed-objects* paradigm. The components in MOCHA are:

- **Client APIs** - These are Java-based libraries to be used in the client applications developed for MOCHA. These applications can either be stand-alone or applets that are loaded into a (Java-enabled) Web browser. Thus, MOCHA treats Web-based clients as "first class" citizens. The client APIs contain the functionality necessary to pose queries to MOCHA and retrieve their results. In addition, they contain the infrastructure used to load the code for the data types and operators needed to manipulate the results of the query. Notice that such data types and operators are delivered to the client application by MOCHA.
- **Query Processing Coordinator (QPC)** - This element controls the execution of the queries received from the client applications. It provides services such as query parsing, query optimization, query operator scheduling, query execution and monitoring of the entire execution

process. It also provides access to the meta-data in the system and to a repository containing code implementing system functionality. During query execution, this element is in charge of deploying all necessary system functionality to the client application and to those remote sites from which data will be extracted. Query execution is done by an executor module based on iterators[Gra93].

- **Data Access Provider (DAP)** - This element provides the QPC remote access to the data source, just like a wrapper does. But unlike most wrappers, DAP has query execution capabilities and runs on the data source site. Like in QPC, query execution is carried out in an executor module based on iterators. DAP extracts the data from the local source, converts it to the global model and either sends the raw data values to the QPC or performs filtering operations over the data (i.e. evaluates predicates or aggregates from the query) and sends the results back to the QPC for further processing. DAP receives from QPC the code for the data types and the operators needed to process each query. Clearly, it is possible for the DAP to cache the code so it can be reused several times without the need for repeated delivery.
- **Data Server** - This is the server that stores the data sets for the local site. This element can be a full-fledged database server, a Web server or even a file server providing access to a collection of flat files. In the MOCHA prototype, we used an object-relational database server since these are becoming the preferred choice for storing and manipulating data sets containing complex types.

Both the DAP and the QPC are implemented as Java objects. MOCHA provides standard interfaces with the methods needed to interact with QPC and DAP. All the interactions between the clients and QPC, and between QPC and the DAPs are carried out through the Java Remote Method Invocation (RMI) mechanism. In the near future, we expect to add support for the also popular CORBA standard.

By using standard interfaces to interact with QPC and DAP, their implementations can be upgraded without affecting the other components in MOCHA. For example, a client application will not notice a new version of QPC as long as the new implementation uses the same interface as the previous implementation. Similarly, different implementations of the components can be used based on the privileges or needs of the users. A DAP can be implemented by two different Java classes, one with support for updates to the data in the local site and the other one limited to read-only operations. In this case, both implementations use the same interface, but threat the requests differently. Furthermore, the standard interfaces can be extended to provide new methods. For example, an extended interface for DAP may provide additional methods to create temporary files in the local site. MOCHA provides a flexible framework in which individual users can be offered with different “views” of the system depending on their needs or privileges. This flexibility is a necessary requirement for accommodating the needs of new applications without sacrificing those of existing applications.

3.1 Code Deployment

In MOCHA, code is deployed by shipping the compiled Java classes that contain the implementation of the advanced data types and query operators. To simplify this discussion, we assume that the implementation of each data type or operator is contained in only one Java class. In general, however, a data type might contain several other data types, and therefore, its implementation would span several classes. The same argument holds for the query operators.

Once a developer has written and tested the Java class containing the implementation of a data type or operator, he/she stores this class in a *code repository* (see Figure 1). Then, he/she ”registers” the new code by adding entries into the local catalog that contain, among other things, the name of the data type or operator and the name of the class where its implementation resides. Any other kind of relevant information, such a version number, users privileges, etc., is also added in this way. Once the code has been registered, it is ready for use by MOCHA. The data types and operators can be used in compiled and ad-hoc queries alike. The developer might have to publish the availability of the new functionality to the user community. This can be done by sending an electronic message to the users or by updating a Web page containing a description of the available types and operators in the system. Clearly, some form of repository management is needed.

When QPC receives a query, it generates a list with the data types and operators that are necessary to process the query at hand. The QPC accesses meta-data in the local catalog to map each data type and operators to its implementing class. Each class is then retrieved from the local code repository and prepared for distribution. It might be the case that some classes reside in remote repositories. In such case, the QPC accesses those remote repositories and retrieves all the classes that it needs[1]. Before the execution of the query starts, QPC ships the Java classes containing the required data types to the DAPs in the targeted data sites and to the client application. Then, the QPC sends the DAPs the classes containing the query operators that each DAP has to execute. We call this entire phase the *adaptation phase* since each party involved adapts to the needs of the query at hand. Once the adaptation phase is completed, QPC and the DAPs start processing the data and generating results. All the results are gathered by QPC and sent back to the client application for visualization purposes.

---

[1] Clearly, QPC must have the appropriate permissions to perform such operation.

It is important to point out that the adaptation phase occurs **on-line** and with no user involvement, as an autonomous process carried out completely by QPC. We emphasize the fact that it is not necessary to shutdown and restart any component in MOCHA before it can start using the new functionality received during the adaptation phase. Instead each of the QPC, DAPs and client application contains the necessary logic to load the Java classes into their run time systems and immediately start using them. Thus, each component in MOCHA can be extended at run time in a dynamic fashion.

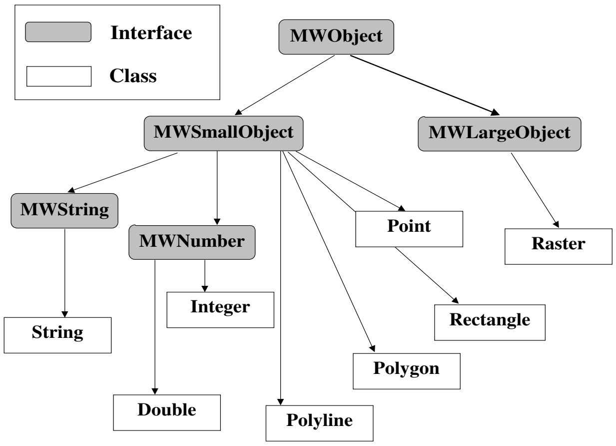


Figure 2: MOCHA Type System

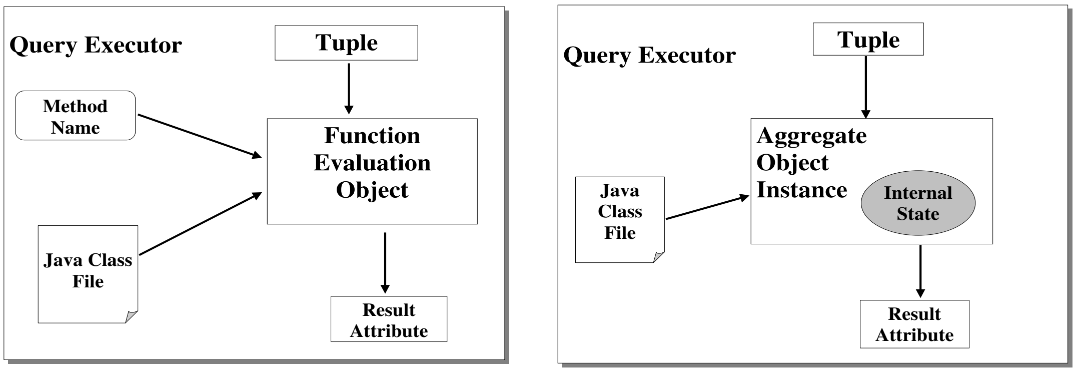
Organization of Data Types

In MOCHA, the attributes in a tuple are implemented as Java objects. In order for the QPC, DAPs and client applications to handle the classes for data types correctly, MOCHA provides a set of well-known type interfaces that must implemented in these Java classes. Figure 2 depicts the hierarchical structure of the type system for the current implementation of MOCHA. The gray rectangles represent the type interfaces and the white ones represent the Java classes that implement a given type. At the root of the hierarchy is the `MWObject` interface which is implemented by all the classes used for all data types. This interface identifies a Java object as been an instance of a type that can be manipulated by a component in MOCHA. In addition, this interface contains methods specifying how to transmit the object over the network. The `MWSmallObject` and `MWLargeObject` type interfaces extend `MWObject` and partition all types into two groups: large objects and small objects. Large objects are used for large sized types such as images, audio or text documents. Small objects are used for smaller types such as numbers, strings, points, rectangles, and so on. Additionally, interfaces for character and numeric based types are derived from `MWSmallObject`. Each of the Java classes for the data types must implement at least one of the interfaces below

`MWObject`. By doing so, the class inherits the features associated with the category represented by the interface.

Organization of Query Operators

MOCHA groups operators into two categories: 1) projections and predicates, and 2) aggregates. The complex functions in the projections and predicates are implemented as static methods[2] in a Java class. Figure 3(a) depicts how a function contained in a projection or in a predicate is evaluated in the executor module inside of either the QPC or a DAP. The name of the static method is included in the query plan created by QPC. The executor module uses the name of the static method and the Java class where it is defined to create a *function evaluation object.* This object executes the body of the method and hence the complex function in the query operator. The executor successively passes tuples to the function evaluation object and collects the resulting attributes for further processing or adds them into the result.



(a) Predicates & Projections (b) Aggregates

Figure 3: Operator Evaluation in MOCHA

Aggregates, on the other hand, are implemented as Java objects. Figure 3(b) shows the overall scheme. MOCHA provides an interface called `Aggregate` that specifies three methods that must implemented in all classes used to create aggregate operators: `Reset()`, `Update()` and `Summarize()`. For each aggregate operator in a query, the query executor creates one aggregate object for each of the different groups encountered during the aggregation process. The internal state in the aggregate object is first set into an initial state by calling `Reset()`. Then, the query executor successively calls `Update()` to modify the internal state in the aggregate using the next tuple at hand and the current state in the aggregate. Once the tuples are exhausted, the final aggregate value is extracted

---

[2] These are methods that can be accessed directly from the Java class file, without having to create an object-instance from the class.

from the internal state contained in each aggregate object by calling `Summarize()`. The aggregate value is then added into the result.

Benefits of MOCHA

MOCHA leverages on self-extensibility to scale to hundreds of data sites. First of all, by implementing the data types and operators in Java classes, we avoid the problems of porting the code to different platforms. The Java code is compiled just once and deployed to each site as needed. Again, this code deployment occurs with no user involvement. Since Java is readily available for most computing platforms, it is neither difficult nor expensive to set up a Java-based environment to run MOCHA.

By organizing the code into repositories and shipping it from there to the remote sites in the system, it becomes easier to distribute new functionality, keep track of versions and apply updates. Recall the situation in Example 2.1. Using MOCHA, we can have the classes for types `Polygons`, `Rectangle` and `Raster` replicated into each of the code repositories for each data site in each state. Notice, that it is even possible to have a global repository where the most common data types and operators used throughout the entire system are kept. Each site can replicate the portions of this global repository that better fit the needs of their local users. In addition, each site can have its own types and operators used by the applications specific to the local site. Clearly, the tasks for the administrators are greatly reduced since they only have to deal with a few well-known sites where the code resides. Upgrades and new functionality are simply added into each repository, and from there they are deployed as needed by the system. Therefore, MOCHA provides a flexible framework that keeps administrative tasks simple, and yet it does not restricts any site in their choice of functionality. Users are free from having to search for code with the right functionality for their applications, since MOCHA delivers it as needed. This methodology is by far superior to the approach where libraries must be installed at any site where a client, wrapper or mediator element is to be run.

Another benefit of MOCHA is the ease with which new functionality can be added to the clients, QPCs and DAPs. All data types and operators are manipulated in generic fashion through the well-known interfaces. Thus, the client, QPC or DAP does not needs to understand all of the internals of a particular type or operator. For example, the module in the QPC that transmits the data values over the network handles every type as a `MWObject`, and calls the appropriate routines in this interface to send/receive each value. There is no need to “hard code” special routines to deal with each type or operator. Instead, the Java class implementing each operator or data type contains enough functionality for the QPC, DAP or client application to manipulate it.

3.2 Query Processing Framework

MOCHA is designed to leverage on its ability to ship code in order to generate query plans that minimize the amount of data movement during query processing. MOCHA pushes the evaluation of *data-reducing* operators to the DAPs running on the data sources and the execution of *data-inflating* operators to the QPC.

In MOCHA, *data-reducing* operators are those operators that reduce either the number of tuples in the result or the size of those tuples. Under this category we include:

- predicates with low selectivity factors, which filter out unnecessary tuples from the result
- predicates whose arguments are large-sized attributes, but these attributes are not part of the result
- projections that map very large-sized attributes into scalars or smaller-sized values.
- aggregates that map sets of tuples into distilled values

In the Example 2.2 of page 4, the projection operator `AvgEnergy(image)` is as a data-reducing operator since it maps a 1MB image into a 8-byte floating-point number. Data-reducing operators are commonly found in applications related to areas such as Data Mining, OLAP and Scientific Computing. In their queries, the *raw data values* are not needed for the result, but rather a distilled abstraction of them.

On the other hand, *data-inflating* operators are those operators that inflate the data values and present them in many forms, projections, rotations, sizes and levels of detail. We can find many data-inflating operators in applications related to Computer Graphics, Solid Modeling, Earth Science, and Multimedia Databases.

Example 3.1 A user from Virginia issues the following query to the data site in Maryland:

```
SELECT  time, location, IncrRes(image, 2X)
FROM    Rasters
```

This query retrieves all the images from the `Rasters` table in Maryland, but the function `IncrRes()` increases the resolution of an image by a factor 2X before presenting it to the user.□

The projection `IncrRes(image,2X)` in Example 3.1 is a data-inflating operator since it synthesizes an image that is four times larger than the original one. Other kinds of data-inflating operators are those used to visualize the same data value from many perspectives. One example is an operator that rotates an image by a certain degree θ, but does not change the size of the image. Another example in this class is an operator that allows a user to visualize a three-dimensional solid object from different orientations (i.e. top, bottom or sideways) which are given by the user in an interactive fashion. In these data-inflating operators the same data value is repeatedly transformed and, therefore, these transformations are more efficiently done at the client's site.

Clearly, a query can have an arbitrary combination of data-reducing and data-inflating operators. Therefore, it is necessary to evaluate several alternative query plans to find the one with the minimal data movement. Typically, a query optimizer evaluates the alternatives using an estimate of the execution time as a cumulative value derived from the time spent performing I/O, computation and network-related operations [ML86, GHK92]. As noted in [HS93], estimating the cost of these components in the presence of user-defined functions in the operators can be a rather complicated task since it is difficult to estimate how many I/O operations a function will do or how long will its computation take. In MOCHA, however, we introduce a new metric called the *volume reduction factor*, which measures network transmission cost due to data movement which is the dominant factor in distributed systems.

Definition 3.1 The *Volume Reduction Factor*, VRF, for a query plan P to answer query Q over a relation R is defined as

$$VRF(P) = \frac{VDT}{VDA} \qquad (0 \leq VRF(P) < \infty),$$

where VDT is the total volume of data transmitted over the network by P and VDA is the total volume of data from R that is retrieved to answer Q using P. Both VDT and VDA are measured in bytes.

Note that data-reducing operators give VRF values less or equal than 1 while data-inflating ones give a value greater than one. The volume reduction factor, (referred to as VRF from now on) serves as a measurement of the cost of a plan. Plans with VRF close to zero are better since the amount of data they transmit is small, and these plans are executed at the DAPs. On the other hand, any plan with a VRF greater than one, such as one with a data-inflating operator, is executed at the QPC. Therefore, in MOCHA, plans containing data-inflating operators are executed using the traditional data shipping approach. The plans containing data-reducing operators are executed using a new policy named *code shipping.*

Definition 3.2 The *code shipping* query processing policy specifies that a query operator and its code are to be shipped to and executed by the DAP that runs at the data source.

4 Performance Evaluation

To validate the overall design choices and our performance expectations for MOCHA , we have implemented prototypes for DAP and QPC. Using these prototypes, we study the performance of MOCHA and show:

- the feasibility of using Java to implement data types and query operators
- that VRF is better than straight selectivity for a cost-based optimizer
- the benefits obtained by using code shipping in query processing
- the need to use data shipping for data-inflating operators

In the MOCHA prototype, DAP contains the necessary logic to load Java classes containing types and operators, extract data from a source and execute parts of a query plan. Similarly, QPC contains the logic to coordinate query execution, ship Java classes full with system functionality and execute parts of a query plan. At this moment, however, we have not completed the implementation of the cost-based query optimizer for the QPC although the major building blocks are in place. In MOCHA, not only do we have to deal with the placement of complex predicates [HS93, CS96] but also with complex projections and aggregates, thus making the optimization process more complicated. Therefore, it is essential to understand the behavior of the system in order to adopt good strategies for the optimizer. The performance study shown below is a first step towards understanding the implementation behavior of MOCHA.

4.1 Experimental Setup

We have implemented the prototypes for both QPC and DAP as Java distributed objects. All data types and operators were implemented in classes containing 100% Java code. We used Sun's Java Developers Kit 1.2beta4 to implement all of the Java classes. As our data server, we used the Informix Universal Server (IUS), version 9.12. A *datablade* was used to implement the local data model for the schema described in section 4.2. To provide connectivity between Informix and the DAP, we developed a JDBC-like library with support for complex types. This library was integrated with the client libraries provided by Informix using the facilities provided by the Java Native Interface (JNI).

In our experimental setup, we configured the QPC to use query plans in two modes: code shipping and data shipping. This permits the study of each kind of operator under each of the policies. In each experiment, we measured the execution time of a query from the time QPC starts deploying code to the time the last tuple is processed and added to the result. We do not include the time needed by the QPC to forward the results to the client application since it should be the same for both code and data shipping methods. The execution time is divided into four components: **DB Time**, **CPU Time**, **Net Time** and **Misc Time** as shown in Table 1. The Net Time component includes network transmission time and software overhead. Network transmission time is the time it takes to transmit all the bits contained in the data values from the DAP to the QPC. Software overhead is time spent in the JDBC-like routines that must be called to carry out the data transfer.

Component	**Meaning**
DB Time	Time spent reading tuples from data source by DAP.
CPU Time	Time spent evaluating all complex query operators.
Net Time	Time spent sending data from DAP to QPC.
Misc Time	Time spent on all initialization and cleanup tasks.

Table 1: Execution Time Components

In addition to the execution time, we measured the amount of data that is manipulated by each of the policies during the execution of each query. Specifically, we measured the volume of data accessed (VDA), the volume of data transmitted (VDT) and the volume of data in the query result. For each alternative, the VRF is computed from the measured VDA and VDT.

In all the experiments discussed in this paper, we ran the QPC on a Sun Ultra SPARC 60, running Solaris 2.6. This machine had 128MB of main memory. Similarly, the DAP and the IUS were run on a Sun Ultra SPARC 1 running Solaris 2.5. This second machine had 256 MB of main memory. Both machines were connected to a 10Mbps Ethernet network. The Informix Server stored all data using raw partitions on a 9 GB Seagate Cheetah Ultra SCSI hard disk. We ran all experiments late at night when both machines and the network were unloaded. The values reported as execution times are averages obtained from five independent measurements. For all experiments, the Java Virtual Machines (JVM) used to run the DAP and the QPC were limited to use a maximum of 96MB of memory. In addition, the Java Just-In-Time (JIT) compiler in each Virtual Machine was activated, thus at run time, all Java classes were compiled to native code before been executed.

4.2 Benchmark Data and Queries

We used the Sequoia 2000 Benchmark [Sto93] to test our implementation of MOCHA. Sequoia contains data sets describing geographic information about the Earth. We used the data sets contained in the regional version of the benchmark, which correspond to the state of California.

```
Points(name:String,location:Point);  Polygons(landuse:Integer,location:Polygon);

Graphs(identifier:Integer,          Raster(time:Integer,band:Integer,
     location:Polyline);                 location:Rectangle, data:Raster
                                         lines:Integer,sample:Integer);
```

Figure 4: Schema for Benchmark Data

Figure 4 contains the schema used for the experimental procedure. The description of the schema is a follows:

- `Points`- A collection of geographic locations represented with their name and their coordinates. There are 62556 tuples for a total size of 4.3 MB.
- `Polygons`- This data set is composed of polygons enclosing regions of land. For each polygon, the type of land enclosed (e.g. forest, city, etc.) and an array with its exterior vertices are stored. This data set contains 77643 entries for a total size of 18.8 MB.
- `Graphs`- This collection is composed of graphs representing drainage networks. Each network is represented by an identifier and a collection of line segments that make up the graph. The data set contains 201650 entries for a total size of 31MB.
- `Rasters`- This is a set of raster images made from weekly energy readings of the Earth's surface gathered by satellites. For each entry, we have the time (week number), the energy band measured, the bounding box for the region, the raster image itself, the number of lines in the image and the number of samples (pixels) per line. Each sample is a 16-bit integer. Originally, each image was 8MB in size, but we partitioned their bounding boxes into 8 sub-regions, each containing a sub-image of 1MB. We then picked 200 entries corresponding to all the readings for the first five weeks, for a total size of 200MB.

Our benchmark queries are shown in Table 2. We derived these queries from the ones in Sequoia by adding and combining several new complex operators. All the functions in these queries were written as 100% Java code. We classified these queries into categories with different execution characteristics and compared their execution in MOCHA using both code shipping and data shipping. In all their execution time graphs, code shipping is labeled as CS, data shipping is labeled as DS and execution time is measured in seconds (secs).

4.3 Queries with Complex Aggregates

Aggregates are perhaps the most powerful of all the data-reducing operators since they decrease both the number of tuples in the query result and the size of those tuples. For this category we used queries Q_1, Q_2, Q_3 and Q_4 from Table 2. Queries Q_1 and Q_2 are similar in structure since

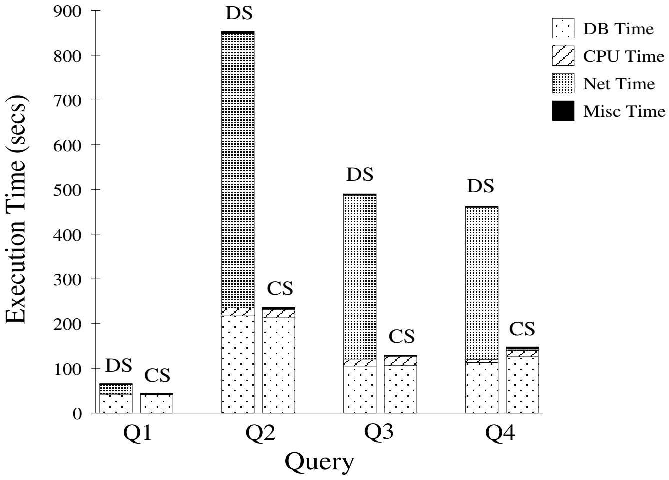
Q_1: SELECT MinBox(location) FROM Points;	Q_2: SELECT TotalArcLength(graph) FROM Graphs;
Q_3: SELECT landuse, TotalArea(location), TotalPerimeter(location) FROM Polygons GROUP BY landuse;	Q_4: SELECT location, CompositeImage(Clip(data,lines, samples,WIN)) FROM Rasters GROUP BY location;
Q_5: SELECT time, band, location, Clip(data,lines,samples,WIN) FROM Rasters;	Q_6: SELECT indentifier, ArcLength(graph) FROM Graphs;
Q_7: SELECT time, band, location, IncrRes(data,lines,samples,2X) FROM Rasters;	Q_8: SELECT time,location FROM Rasters WHERE AvgEnergy(data) < N;
Q_9: SELECT identifier FROM Graphs WHERE NumVertices(graph) > N AND ArcLength(graph) > S;	Q_{10}: SELECT landuse, location FROM Polygons WHERE Perimeter(location) > N;

Table 2: Benchmark Queries

they both map an entire data set into a single value, but Q_2 operates on a larger data set. In Q_1 a bounding box for all the points in table `Points` is computed, while Q_2 computes the total length of all the drainage networks in `Graphs`. Queries Q_3 and Q_4 contain GROUP BY clauses, with Q_4 having a complex data type as the grouping attribute. Q_3 computes the total area and the total perimeter for all the polygons in each category contained in `Polygons`. In Q_4, the aggregate operator `CompositeImage()` computes an image which is the composition of all the images in `Rasters` for each location. Before this composition is made, each image is clipped to a region specified by parameter WIN, which selects a piece five times smaller than the original image.

Figure 5(a) shows the execution time for each query under code shipping and the traditional data shipping approach. In this category of queries, code shipping outperforms data shipping in all cases. As can be observed from the figure, all other times but the Net Time are roughly the same. However, the total time for Q_2, Q_3, Q_4 was roughly 4 times smaller in code shipping(CS) than in data shipping(DS). For query Q_1, code shipping was 50% faster than data shipping.

The performance gain is due to the reduction of data movement. Thus, MOCHA's pushing data-reducing operators to the data source, is very effective for queries with aggregates. This is also seen from the volumes of data transmitted, as shown in Figure 5(b). Notice the large savings in data transmission (the VDT) gained by using code shipping to evaluate the aggregates at the data source. In each case, the DAP only transfers a small fraction of the data that it would otherwise transfer using data shipping. As can be seen from Figure 5(b), for each of the queries the best processing alternative indeed has the smallest VRF. Therefore, the VRF is an excellent metric for



Query	Policy	VDA	VDT	Volume in Result	VRF
Q_1	CS	4.3MB	16B	16B	$4x10^{-6}$
	DS	4.3MB	4.3MB	16B	1
Q_2	CS	31MB	8B	8B	$2x10^{-7}$
	DS	31MB	31MB	8B	1
Q_3	CS	18.8MB	740B	740B	$4x10^{-5}$
	DS	18.8MB	18.8MB	740B	1
Q_4	CS	200MB	1.6MB	1.6MB	0.008
	DS	200MB	200MB	1.6MB	1

(a) Execution Times

(b) Data Volumes

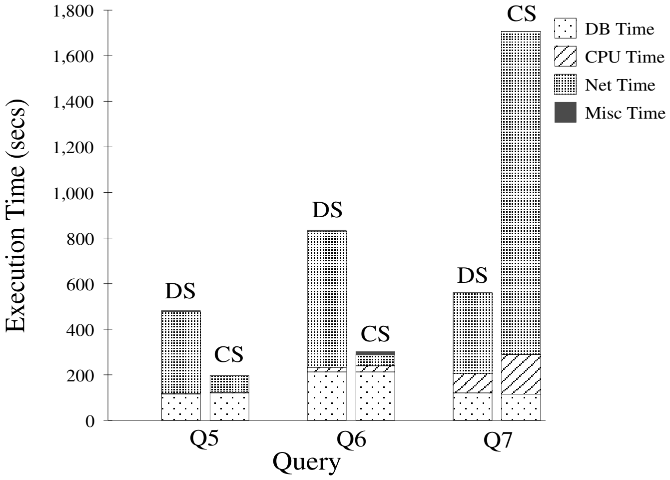
Figure 5: Performance for Q_1, Q_2, Q_3, Q_4

choosing the best plan for queries containing aggregates.

4.4 Queries with Complex Projections

The next query category contains complex functions in the projection operators, which we term as *generalized* complex projections. We measured their effect on the volume of data transmitted with queries Q_5, Q_6 and Q_7 from Table 2. Queries Q_5 and Q_6 contain data-reducing projection operators. In Q_5, each image in table `Raster` is clipped into an image whose size is determined by the clipping box `WIN`. We chose `WIN` so as to generate an image five times smaller than the original. Query Q_6 maps each graph (an array of line segments) in table `Graphs` into a double precision floating point number, which represent the length of a drainage network. On the other hand, Q_7 contains a data inflating operator implemented by the function `IncrRes()` which transforms each image into a new image with twice the resolution of the original one.

As shown in Figure 6(a), code shipping is the better option for executing queries Q_5 and Q_6. By moving the execution of the complex projections to the DAP, the execution time for Q_5 is reduced from 482 secs to 192 secs. Similarly, the execution time for Q_6 is improved from 835 secs to 302 secs. In both cases, code shipping improves performance by a factor of 3. However, code shipping is bad for Q_7 since this query contains a data-inflating operator. The evaluation of the projection `IncrRes(data,lines,sample,2X)` at the DAP, results in the transmission of tuples from the DAP to QPC that are four times larger than those sent under data shipping. Thus, the network cost is increased by a factor of 4. In this case Q_7 runs for a total execution time of 1710 seconds as



Query	*Policy*	*VDA*	*VDT*	*Volume in Result*	*VRF*
Q_5	CS	200MB	40MB	40MB	0.20
	DS	200MB	200MB	40MB	1
Q_6	CS	31MB	2.3MB	2.3MB	0.07
	DS	31MB	31MB	2.3MB	1
Q_7	CS	200MB	800MB	800MB	4
	DS	200MB	200MB	800MB	1

(a) Execution Times

(b) Data Volumes

Figure 6: Performance for Q_5, Q_6 and Q_7

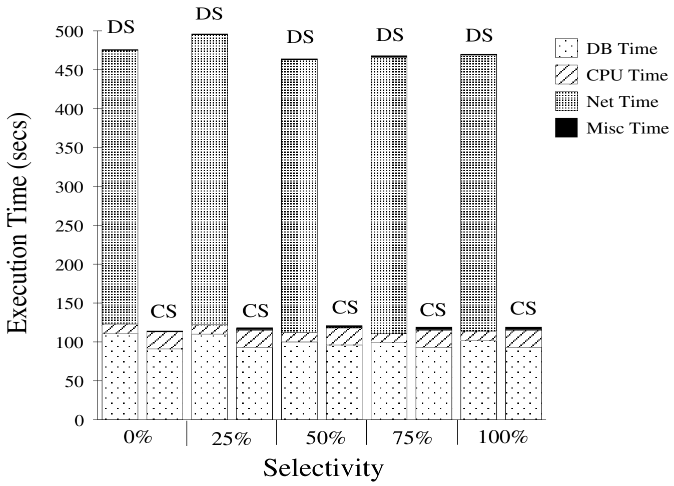
opposed to 562 seconds for data shipping.

We would like to emphasize the accuracy of the VRF in selecting the best plan. As shown in Figure 6(b), for queries Q_5 and Q_6, the VRF for code shipping is a lot smaller than that for data shipping and thus results in much better performance. In contrast, for Q_7 the VRF for code shipping is 4 and results in approximately four times slower executions than data shipping.

4.5 Queries with Complex Predicates

The last category of queries contains complex predicates. Early evaluation of predicates can reduce the volume of data in the query result by either reducing the number of tuples or eliminating (large) attributes from the original data set. For this category, we used queries Q_8, Q_9 and Q_{10} from Table 2 on page 17. Queries Q_8 and Q_9 are similar in the sense that their qualification clauses use attributes that will not be part of the query result. In query Q_8, the function `AvgEnergy()` computes a weighted average of the energy reading contained in each sample in the raster image. In query Q_9, the qualification clause compares the number of vertices and the length of the graph for each drainage network against two constants. For each of the queries, we varied the selectivity of the qualification clause to study the behavior of code shipping and data shipping under different selectivity cases.

The execution times for Q_8 under each selectivity value are shown in Figure 7(a). Code shipping outperforms data shipping in all cases, regardless of selectivity. On the average code shipping achieved a 4-1 improvement in performance. Figure 8(a) depicts a similar picture for Q_9, with performance improved by a factor of 3-1 for the first three selectivity values and a factor of 2-1 for the remaining ones. For these two queries, evaluating the predicate at the DAP results in large savings in data transmission because of the low value of VRF. Pushing the evaluation of the predicates in Q_8 and Q_9 to the DAP, avoids shipping attribute `data` in `Rasters` and attribute `graph` in `Graphs` respectively.



(a) Execution Times

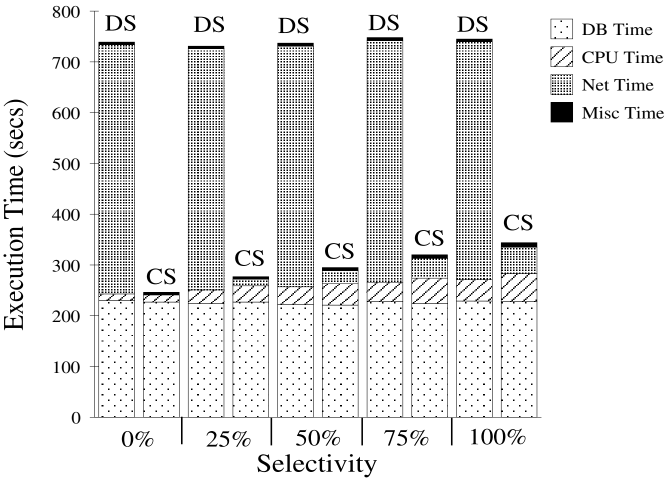
Selectivity	*Policy*	*VDA*	*VDT*	*Volume in result*	*VRF*
0	CS	200MB	0B	0B	0
	DS	200MB	200MB	0B	1
0.25	CS	200MB	1KB	1KB	$5x10^{-6}$
	DS	200MB	200MB	1KB	1
0.50	CS	200MB	2KB	2KB	$1x10^{-5}$
	DS	200MB	200MB	2KB	1
0.75	CS	200MB	3KB	3KB	$1x10^{-5}$
	DS	200MB	200MB	3KB	1
1	CS	200MB	4KB	4KB	$2x10^{-5}$
	DS	200MB	200MB	4KB	1

(b) Data Volumes

Figure 7: Performance for Q_8

In Q_{10}, all the attributes in the tuples in table `Polygons` appear in the result. In this query, the qualification clause compares the perimeter of each polygon against a constant value. As shown in Figure 9(a), for query Q_{10} the gain is in inverse proportion to the selectivity, with the biggest gains in the low selectivity situations. Code shipping outperforms data shipping in all but the last case with 100% selectivity, in which the network, computation, and data access costs are essentially the same in both alternatives but there is a one minute (almost 9%) overhead for code shipping due to initialization and cleanup. Note that the large step up in network cost from 0% to 10% selectivity is due to the nature of the `Polygons` data set and the qualification clause in Q_{10}. The tuples in table `Polygons` have variable size because of attribute `location`, which represents the vertices in the polygon and can have a variable number of them. Therefore, at low selectivity most of the largest polygons satisfy the qualification clause in Q_{10} and these polygons are the most expensive ones to transmit. As the selectivity increases, the increment in network cost is less dominant since only the smaller and less costly polygons are added to the result. Figures 7(b), 8(b) and 9(b) show again that the VRF is still an excellent metric for measuring the overall performance of of queries.

One important concept emerging from the performance results in this section is that selectivity

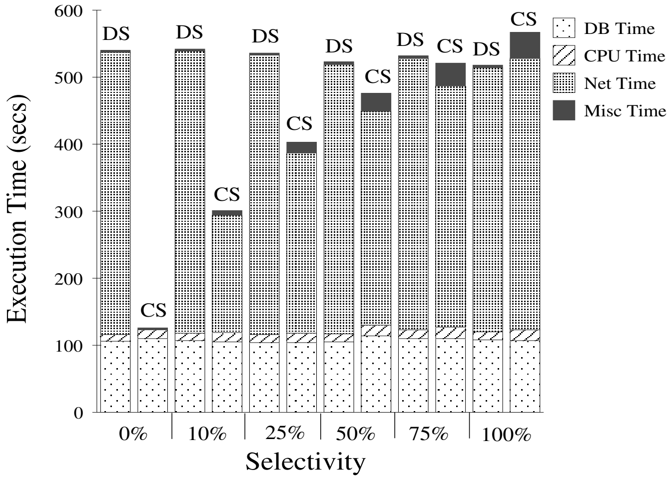


Selectivity	*Policy*	*VDA*	*VDT*	*Volume in result*	*VRF*
0	CS	31MB	0B	0B	0
	DS	31MB	31MB	0B	1
0.25	CS	31MB	200KB	200KB	0.01
	DS	31MB	31MB	200KB	1
0.50	CS	31MB	400KB	400KB	0.01
	DS	31MB	31MB	400KB	1
0.75	CS	31MB	600KB	600KB	0.02
	DS	31MB	31MB	600KB	1
1	CS	31MB	790KB	790KB	0.02
	DS	31MB	31MB	790KB	1

(a) Execution Times

(b) Data Volumes

Figure 8: Performance for Q_9



Selectivity	*Policy*	*VDA*	*VDT*	*Volume in result*	*VRF*
0	CS	18.8 MB	0 MB	0 MB	0
	DS	18.8 MB	18.8 MB	0 MB	1
0.10	CS	18.8 MB	6.3 MB	6.3 MB	0.34
	DS	18.8 MB	18.8 MB	6.3 MB	1
0.25	CS	18.8 MB	11.6 MB	11.6 MB	0.62
	DS	18.8 MB	18.8 MB	11.6 MB	1
0.50	CS	18.8 MB	14.9 MB	14.9 MB	0.79
	DS	18.8 MB	18.8 MB	14.9 MB	1
0.75	CS	18.8 MB	17.2 MB	17.2 MB	0.91
	DS	18.8 MB	18.8 MB	17.2 MB	1
1	CS	18.8 MB	18.8 MB	18.8 MB	1
	DS	18.8 MB	18.8 MB	18.8 MB	1

(a) Execution Times

(b) Data Volumes

Figure 9: Performance for Q_{10}

is no longer the best metric for cost estimation because it fails to take into account the volume of the transmitted data, as in Q_8 and Q_9. Consider the case of 50% selectivity in Q_9. From Figure 8(b), we can see that code shipping only moves 400KB (1% of the original data volume) from the DAP to QPC, not 15MB or half of the volume in the `Graphs` table. In fact, for Q_8 and Q_9 the amount of data transmitted is always much smaller than what would be expected if selectivity were used alone. On the other hand, selectivity might under estimate the actual amount of data transferred. This can be observed for Q_{10} in Figure 9(b), where in the 10% selectivity case, code shipping actually transmits 34% of the data volume in the `Polygons` data set. Evidently, for distributed processing VRF is much better metric for the cost of a plan than selectivity alone.

5 Related Work

Distributed Database Systems have been the subject of extensive research for over twenty years. In many of the proposed systems, it has been assumed that the same data model and DBMS is used at every site. Such systems include R*[WDH+81], Distribute INGRES [Sto86], ADMS(+−)[RK86], and more recently, Mariposa[SAD+94, SAP+96]. Much of the research has focused on the study and development of efficient techniques for distributed join processing [ML86], dissemination of updates [DR94], data replication [SAS+96], data caching [Fra96], the study of client-servers architectures [DR92] and the study of query processing strategies [FJK96]. The trade-offs between the data shipping and query shipping strategies for join processing are discussed in [FJK96]. This work also discusses an alternative strategy called hybrid shipping that combines both data and query shipping. In MOCHA, the use of both code shipping and data shipping is similar to such hybrid approach.

Mediator-based systems are targeted at heterogeneous environments, where the data sources might follow different data models or run different database servers. Under this category we find systems such as TSIMMIS [CGMH+94], Pegasus [Aea91], DISCO [TRV96], and Garlic [RS97] Most of the research in these systems has been directed towards the development of translation schemes to map the data models in each data source into the global model used by the mediator. More recent work has addressed the issue of query optimization in heterogeneous environments [RS97, HKWY97].

The use of Java as the implementation language for used-defined functionality in a DBMS is been considered by many commercial products and research prototypes. The Jaguar Project at Cornell University is investigating many of the interesting issues raised by such an idea, in the context of the Predator client-server DBMS [SLR97]. In [GMSvE98] synthetic data and queries are used to analyze the performance impact of implementing user-defined functions in Java for use in Predator.

Finally, the work in [HS93, CS96] has studied the development of optimization techniques based on selectivity to handle queries with user-defined predicates, in the context of single-site databases.

6 Conclusions

In this paper we proposed MOCHA as an alternative middleware solution to the problem of integrating data sources distributed over a network. We have argued that the scheme in traditional mediator-based systems, where the functionality for data types and query operators is manually deployed is inadequate and will not scale to environments with hundreds or more of data sources. The high cost and complexity involved in having administrators installing and maintaining the necessary software libraries into every site in the system makes the traditional approach impractical. MOCHA, on the other hand, is a self-extensible middleware substrate written in Java, in which the functionality is automatically deployed on the sites that do not provide it. Code shipment in MOCHA is fully automatic with no user involvement and this reduces the complexity and cost of deploying functionality in large systems.

MOCHA offers data-reducing operators which are evaluated at the data sources and data-inflating operators that are evaluated near the client. Data shipping is used for the plans with data-inflating operators and a new policy named code shipping is used for the plans with data-reducing ones. The selection between code shipping and data shipping is based on a new metric, the Volume Reduction Factor, which measures the amount of data movement in queries used in distributed systems.

The MOCHA prototype implementation is written in Java and runs on top of the Informix Universal Server. We have carried out a performance study on MOCHA using data and queries from the Sequoia 2000 Benchmark. We have shown through experiments that selecting the right strategy and the right site to execute the operators can increase performance of queries by a factor of 4-1, in the case of aggregates and predicates, or 3-1, in the case of projections. These experiments also demonstrated that the volume reduction factor is a more accurate metric than the standard selectivity factor.

Future work on MOCHA includes a cost-based query optimizer, exchange protocols among multiple Java DAP stations, and storage and pipeline capability on DAPs. The last two are needed for multi-site interaction and exchange of data and control.

Acknowledgements We would like to thank Informix Software Inc., for making available the Informix Universal Server through the University Grant Program. We would also like to thank Björn Thór Jónsson, Kostas Stathatos, Yannis Kotidis, Alexandros Labrinidis, Mike Beynon and Nelson Padua-Pérez for many useful discussions, comments and suggestions on the contents of this paper. Kostas Stathatos coined the term *data-inflating* operators. This research work was supported by the DOD Lucite Contract CG9815.

References

[Aea91] R. Ahmed and et. al. The Pegasus Heterogeneous Multidatabase System. In *IEEE Computer*, pages 19–27, December 1991.

[CGMH+94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of IPSJ Conference*, Tokyo, Japan, 1994.

[CS96] Surajit Chaudhuri and Kyuseok Shim. Optimization of Queries with User-defined Predicates. In *Proc. 22nd VLDB Conference*, pages 87–98, Bombay, India, 1996.

[DR92] Alexios Delis and Nick Roussopoulos. Performance and Scalability of Client-Server Database Architectures. In *Proc. 18th VLDB Conference*, pages 610–623, Vancouver, British Columbia, Canada, 1992.

[DR94] Alexios Delis and Nick Roussopoulos. Management of Updates in the Enhanced Client-Server Dbms. In *Proc. 14th ICDCS Conferece*, pages 326–334, 1994.

[FJK96] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proc. ACM SIGMOD Conference*, pages 149–160, Montreal, Quebec, Canada, 1996.

[Fra96] Michael J. Franklin. *Client Data Caching.* Kluwer Academic Press, Boston, Massachusetts, 1996.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query Optimization for Parallel Execution. In *Proc. ACM SIGMOD Conference*, pages 9–18, San Diego, California, USA, 1992.

[GMSvE98] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Secure and portable database extensibility. In *Proc. ACM SIGMOD Conference*, pages 390–401, Seattle, Washington, USA, 1998.

[Gra93] Goetz Grafe. Query Evaluation Techniques for Large Databases. *ACM Computer Surveys*, 25(2):73–170, June 1993.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yans. Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd VLDB Conference*, pages 276–285, Athens, Greece, 1997.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. ACM SIGMOD Conference*, pages 267–276, Washington, D.C., USA, 1993.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. 12th VLDB Conference*, Kyoto, 1986.

[RK86] Nick Roussopoulos and Hyunchul Kang. Preliminary Design of ADMS+-: A Workstation-Mainframe Integrated Architecture for Database management systems. In *Proc. 12th VLDB Conference*, pages 355–364, Kyoto, Japan, 1986.

[RS97] Mary Tork Roth and Peter Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *23rd VLDB Conference*, Athens, Greece, 1997.

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R.A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM SIGMOD Conference*, pages 23–34, Boston, Massachusetts, USA, 1979.

[SAD+94] Michael Stonebraker, Paul M. Aoki, Robert Devine, Witold Litwin, and Michael Olson. Mariposa: A New Architecture for Distributed Data. In *Proc. 10th Int. Conference on Data Engineering*, pages 54–65, Houston, Texas, 1994.

[SAP+96] Michael Stonebraker, Paul M. Aoki, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 1996.

[SAS+96] Jeff Sidell, Paul M. Aoki, Adam Sah, Carl Staelin, Michael Stonebraker, and Andrew Yu. Data Replication in Mariposa. In *Proc. 12th ICDE Conference*, pages 485–494, New Orleans, Louisiana, 1996.

[SLR97] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proc. 23rd VLDB Conference*, pages 66–75, Athens, Greece, 1997.

[Sto86] Michael Stonebraker. The Design and Implementation of Distributed INGRES. In *The INGRES Papers*. Addison-Wesley, Reading, Massachusetts, 1986.

[Sto93] Michael Stonebraker. The SEQUOIA 2000 Storage Benchmark. In *Proc. ACM SIGMOD Conference*, Washington, D.C., 1993.

[TRV96] Anthony Tomasic, Louiqa Rashid, and Patrick Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *Proc. 16th ICDCS Conference*, Hong Kong, 1996.

[WDH$^+$81] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, Wilms P, and R. Yost. R*: An Overview of the Architecture. Technical Report RJ3325, IBM Almaden Research Center, San José, California, 1981.