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

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

Nick Roussopoulos
Department of Computer Science
University of Maryland, College Park
nick@cs.umd.edu

Abstract

We present MOCHA, a new self-extensible database middleware system designed to interconnect distributed data sources. MOCHA is designed to scale to large environments and is based on the idea that some of the user-defined functionality in the system should be deployed by the middleware system itself. This is realized by shipping Java 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 source sites while executing *data-inflating* operators near the client's site. The Volume Reduction Factor is a new and more explicit metric introduced in this paper to select the best site to execute query operators and is shown to be more accurate than the standard selectivity factor alone. MOCHA has been implemented in Java and runs on top of Informix and Oracle. We present the architecture of MOCHA, the ideas behind it, and a performance study using scientific data and queries. The results of this study demonstrate that MOCHA provides a more flexible, scalable and efficient framework for distributed query processing compared to those in existing middleware solutions.

1 Introduction

Database middleware systems are used to integrate collections of data sources distributed over a computer network. Typically, these type of systems follow an architecture centered around a *data integration server*, which provides client applications with a uniform view and access mechanism to the data available in each source. Such an uniform view of the data is realized by imposing a global data model on top of the local data model used by each source. There are two main choices for deploying an integration server: a commercial database server or a mediator system. In the first approach, a commercial database server is configured to access a remote data source through a *database gateway*, which provides an access method mechanism to the remote data. In the second approach, a mediator server specially designed and tailored for distributed query processing is used as the integration server. The mediator utilizes the functionality of *wrappers* to access and translate the information from the data sources into the global data model. In both of these existing types of middleware solutions the user-defined, application-specific data types and query operators defined under the global data model are contained in libraries which must be linked to the clients, integration servers, gateways or wrappers deployed in the system. There are numerous examples of systems that follow this architecture, some of which are Oracle8i [Ora99], Informix Universal Server [Inf97], TSIMMIS [CGMH$^+$94], DISCO [TRV96] and Garlic [RS97].

Most of the research on database middleware systems carried out during the past years has focused on the problems of translation and semantic integration of the distinct data collections. In this paper, however, we deal with two important problems which have received little attention from the research community: (1) the deployment of the application-specific functionality[1], and (2) the efficient processing of queries with user-defined operators. These are critical problems since they affect the scalability, ease-of-use, efficiency and evolution of the system. Nevertheless, we are not aware of any work that has effectively addressed the first issue, and the second one is just beginning to receive more attention from the community [RS97, HKWY97, GMSvE98, MS99].

In order to effectively address these two important issues we have designed and implemented **MOCHA** (**M**iddleware Based **O**n a **C**ode **SH**ipping **A**rchitecture), a novel database middleware system designed to interconnect hundreds of data sources. MOCHA is built around the notion that the middleware for a large-scale distributed environment should be *self-extensible*. A self-extensible middleware system is one in which new application-specific functionality needed for query processing is deployed to remote sites in **automatic fashion** by the middleware system itself. In MOCHA, this is realized by shipping Java code containing new capabilities to the remote sites, where it can be used to manipulate the data of interest. A major goal behind this idea of automatic code deployment is to fill-in the need for application-

*This research was sponsored by DOD/Lucite Contract CG9815.

[1]These are complex data types and query operators not generally provided by general purpose commercial systems, but rather custom-built for a particular application by third-party developers.

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
MOD 2000, Dallas, TX USA
© ACM 2000 1-58113-218-2/00/05 . . .$5.00

specific processing components at remote sites that do not provide them. These components are migrated on demand by MOCHA from site to site and become available for immediate use. This approach sharply contrasts with existing solutions in which administrators need to manually install all the required functionality throughout the system.

MOCHA capitalizes on its ability to automatically deploy code in order to provide an efficient query processing service. By shipping code for query operators, MOCHA can generate efficient plans that place the execution of powerful *data-reducing* operators (”filters”) on the data sources. Examples of such operators are aggregates, predicates and data mining operators, which return a much smaller abstraction of the original data. On the other hand, *data-inflating* operators that produce results larger that their arguments are evaluated near the client. Since in many cases, the code being shipped is much more smaller than the data sets, automatic code deployment facilitates query optimization based on data movement reduction, which can greatly reduce query execution time. Again, this is very different from the existing middleware solutions, which perform expensive data movement operations since either all data processing occurs at the integration server, or a data source evaluates only those operators that exist **a priori** in its environment [RS97].

In this paper we describe a prototype implementation of MOCHA which has been operational at the University of Maryland since the early Spring of 1998. MOCHA is currently been considered as a middleware solution for the NASA Earth Science Information Partnership (ESIP) Federation. MOCHA is written in Java and supports major database servers, such as Oracle and Informix, file servers and XML repositories. We argue that MOCHA provides users with a more flexible, scalable and efficient framework to deploy new application-specific functionality than those used in existing middleware solutions. Our experiments show that when compared with the processing schemes used in previous 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 3-1 in the case of projections, predicates, and distributed joins. 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 further describes the shortcomings in existing middleware solutions and motivates the need for MOCHA. Section 3 presents the architecture of MOCHA and our solutions to the problems presented in section 2. Section 4 describes the proposed query processing framework for MOCHA. Section 5 contains a performance study of the MOCHA prototype. Finally, our conclusions are presented in section 6.

2 Motivation for MOCHA

Given the immense popularity of the World Wide Web, the continuous growth of the Internet, and the ever increasing number of corporate intranets, database middleware will be required to interconnect hundreds of data sites deployed over these networks. The data sets stored by many of these sites will be based on complex data types such as images, audio, text, geometric objects and even programs. Given this scenario, we argue that middleware solutions for these kind of environments will be successful only if they can provide: (1) scalable, efficient and cost-effective mechanisms to deploy and maintain the application-specific functionality used throughout the system, and (2) adequate query processing capabilities to efficiently execute the queries posed by the users. We argue that the existing middleware solutions fall short from providing adequate support for these two requirements, and we now proceed to justify this argument.

2.1 Deployment of Functionality

In order to deploy new application-specific functionality into a system based on mediators, or database servers (using either gateways or a client/server scheme), the data structures, procedures, and configuration files that contain the implementation of the new types and query operators are collected into libraries which must be installed at each site where a participating integration server, gateway, wrapper or client application resides. This is the scheme followed by Oracle 8i [Ora99], Informix [Inf97], Predator [SLR97], Jaguar [GMSvE98], TSIMMIS [CGMH+94], DISCO [TRV96] and Garlic [RS97]. We argue that as the number of sites in the system increases, such an approach becomes impractical because of the complexity and cost incurred in maintaining the software throughout the system.

To illustrate this point, consider an Earth Science application that manipulates data stored and maintained in sites distributed across the United States. Suppose there is one data site per state, which holds data scientists gathered from specific regions within that state. As part of its global schema, the system contains the following relation:

$$\texttt{Rasters}(\texttt{time} : \texttt{Integer}, \texttt{band} : \texttt{Integer}, \\ \texttt{location} : \texttt{Rectangle}, \texttt{image} : \texttt{Raster});$$

Table `Rasters` stores raster images containing weekly energy readings gathered from satellites orbiting the Earth. Attribute `time` is the week number for the reading, `band` is the energy band measured, `location` is the rectangle covering the region under study and `image` is the raster image itself.

To implement the schema for relation `Rasters` in existing middleware solutions, it would be necessary to install the libraries containing the code, mostly C/C++ code, for the `Rectangle` and `Raster` data types on each site where a client, integration server, wrapper or gateway interested in using table `Rasters` resides. This translates into at least fifty installations for the wrappers and gateways, plus as many more as are necessary for the integration servers and clients. The administrators of the system will have to access all these sites and manually perform these installations. Moreover, it is often the case that the functionality has to be *ported* to different hardware and operating system platforms. As a result, developers must invest extra effort in making the functionality work **consistently** across platforms. Furthermore, sci-

entists are frequently experimenting with new or improved methods to compute properties about their data, such as averages on the amount of energy absorption. Therefore, many users will not be satisfied with some of the code that has been already deployed, and will require for existing operators to be upgraded or replaced by new ones. Thus, it becomes necessary to have a scalable and efficient mechanism to keep track of and maintain the deployed code base in the system. Clearly, the logistics in such a large-scale system are formidable for an approach based on manual installations and upgrades of application-specific functionality to be feasible.

2.2 Efficient Query Processing

In the context of large-scale distributed environments, it is unrealistic to assume that every site has the same query execution capabilities. Therefore, many existing middleware solutions use a query processing framework in which most operators in a query are completely evaluated by the integration server [Ora99, Inf97, CGMH+94, TRV96]. The wrappers and gateways are mainly used to extract data items from the sources, and translate them into the middleware schema for further processing at the integration site. This approach for query processing is often called *data shipping* [FJK96] because data is moved from the source to the query processing site. In the alternative approach, called *query shipping* [FJK96], one or more of the query operators are evaluated at the data sources, and only the results are sent back to the integration server. A third approach, called *hybrid shipping* [FJK96], combines query and data shipping, and is shown to be the superior alternative. Garlic [RS97] exploits this latter approach in a powerful framework in which the evaluation of some of the query operators is "pushed" to the data source. However, this framework is somewhat limited since it can only be applied to those operators that are **already implemented** at the data source, and the integration server must evaluate any remaining operator(s) in the query. Thus, in all these middleware solutions, query operator placement is severely restricted by the availability of the code implementing the operators used in a query. In many situations, this can easily lead to worst-case scenarios in which a query plan dictates the transfer of very large amounts of data (megabytes or more!) over the communications network, making data transmission a severe performance bottleneck.

We illustrate this point with the Earth Science application introduced in section 2.1. Let the data site in the State of Maryland contain 200 entries in table `Rasters`. For 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 at the State of Virginia poses the following query for the data site at the State of 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. Function `AvgEnergy()` returns a 8-byte double precision floating point number, so the size of the records in the result is just 28 bytes. Clearly, the best way to execute this query is to let the data source at Maryland read every tuple from table `Rasters`, evaluate function `AvgEnergy()`, evaluate the qualification clause, and perform all the projections in the query, returning only the final results to the integration site in Virginia. Notice that even if all tuples in table `Rasters` satisfy the qualification clause, data movement still is negligible since 200 tuples $\times$ 28 bytes is approximately 5KB! However, this efficient approach is feasible **only if** the code for `AvgEnergy()` exists at the Maryland data site. Otherwise, this query must be evaluated by first shipping the attributes `time`, `location` and `image`, contained in every tuple in table `Rasters` from the data site in Maryland to the integration site in Virginia, and then performing the operations as mentioned before. Now, consider the cost of the operation just described. The system is moving roughly 200MB worth of data over a wide area network, an operation that will take minutes or even hours to complete since the bandwidth available to an application in most wide area links is very limited, often under 1Mbps. Clearly, the lack of adequate query processing functionality can easily lead to poor performance because a middleware system might be **restricted** to use an inefficient query processing strategy simply because the functionality required to use a superior one is not available where it is needed. Since it is highly improbable that all the functionality needed to process every query posed will be available everywhere, existing solutions are of limited use in large-scale environments. Notice that although many systems indeed provide the capabilities to manually add the code for user-defined types and operators into the wrappers [CGMH+94], gateways [Inf97], or remote data servers [SLR97, GMSvE98, MS99], this approach simply cannot scale for the reasons already discussed in section 2.1. Thus, the query processing framework used in existing solutions is limited and ill-suited for a large-scale environment in which users have diverse needs for processing data.

3 MOCHA Architecture

We have designed MOCHA around two fundamental principles that will enable it to overcome the shortcomings of previous middleware solutions. The first principle is that all application-specific functionality needed to process any given query is to be delivered by MOCHA to all interested sites in **automatic** fashion, and this is realized by shipping the Java classes containing the required functionality. The second principle is that each query operator is to be evaluated at the site that results in minimum data movement. The goal is to ship the code and the computation of operators around the system in order to minimize the effects of the network bottleneck. We argue that this framework provides the foundation for a more scalable, robust and efficient middleware solution. Figure 1 depicts the organization of the major components in the architecture for MOCHA. These

are the **Client Application**, the **Query Processing Coordinator (QPC)**, the **Data Access Provider (DAP)** and the **Data Server**. We now elaborate on the principles and design choices for MOCHA which form the basis for our arguments. A more detailed description can be found in [RMR00b].

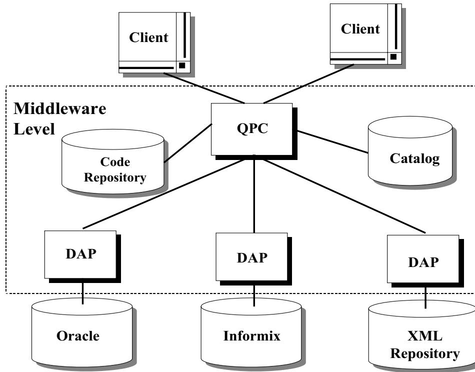


Figure 1: MOCHA Architecture

3.1 Client Application

MOCHA supports three kinds of client applications: 1) applets, 2) servlets, and 3) stand-alone Java applications. We expect applets to be the most commonly deployed clients in the system, used as the GUI for the users to pose queries against the data collections and visualize their results. Servlets can be used to interact with users that do not use applets with their Web browsers. The servlet receives the requests from the browser, interacts with MOCHA to query the data sources, and formats all results as either HTML or XML data. Finally, stand-alone Java applications will likely be used by administrators or software developers, who will need to carry out complex tasks such as system configuration and tuning, catalog management or distributed software debugging, to name a few. MOCHA provides a set of Java libraries with the APIs necessary for the client to easily interact with the system. These APIs contain all the required infrastructure to load the code containing the application-specific components necessary to manipulate the query results.

3.2 Query Processing Coordinator (QPC)

The Query Processing Coordinator (QPC) is the middle-tier component that controls the execution of all the queries and commands received from the client applications. QPC provides services such as query parsing, query optimization, query operator scheduling, query execution and monitoring of the entire execution process. QPC also provides access to the metadata in the system and to the repository containing the Java classes with application-specific functionality needed for query processing. During query execution, the QPC is responsible for deploying all the necessary functionality to the client application and to those remote sites from which data will be extracted.

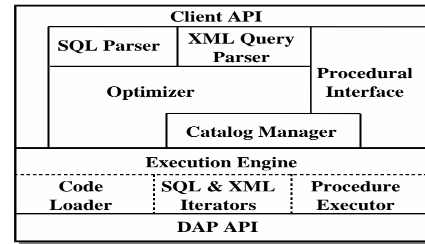


Figure 2: Organization of the QPC

The internal components of the QPC are depicted in Figure 2. The Client API serves as the entry point to accept the requests from a client application. The QPC offers three main data processing services. The first one provides access to distributed data sites which are modeled as object-relational sources. QPC provides the infrastructure to perform operations such as distributed joins and transactions over these sources. The requests for these kind of services are encoded as SQL queries, which are first pre-processed by the *SQL parser* in the QPC. The second data processing service provided by QPC allows users to directly query the content of XML repositories. XML is rapidly becoming a very important technology and we felt that MOCHA should support native access to XML repositories without the burden of first mapping them to another data model. We are currently developing the infrastructure that will enable the QPC to process queries over the XML repositories. Finally, since many sources, such as Web servers or file systems, do not provide a query language abstraction, the QPC provides a procedural interface through which operations such as HTTP requests, ftp downloads or proprietary file system access requests can be issued to access these data sources.

One of the most important components of the QPC is its query optimizer, which generates the best strategy to solve the queries over the distributed sources. The optimizer follows a dynamic programming model for query optimization similar to those in System-R [SAC$^+$79] and R* [ML86]. We will defer further details on query optimization until section 4. For now, it suffices to say that the plan generated by the optimizer explicitly indicates which are the operators to be evaluated by the QPC and those to be evaluated at the remote data sites. In addition, the plan indicates which Java classes need to be dynamically deployed to each of the participants in the query execution process. All plans are encoded and exchanged as XML documents, and the interested reader can find examples of their structure in [RMR00a]. The QPC uses the services of the *Catalog Manager* module to retrieve from the catalog all relevant metadata for query optimization and code deployment. Section 3.5 briefly describes the organization of this catalog. The QPC also contains an extensible query execution engine based on iterators [Gra93]. There are iterators to perform local selections, local joins, remote selections, distributed joins, semi-joins, transfers of files and sorting, among others. The execution engine also provides a series of methods used to issue procedural commands (i.e. ftp requests) and to deploy the application-specific code. The

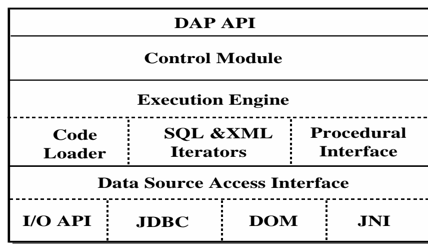


Figure 3: DAP Organization

Code Loader module in the execution engine is used to extract the required code from the code repository, and prepare it for deployment. The DAP API provides the facilities to communicate with the remote data sources.

3.3 Data Access Provider (DAP)

The role of a Data Access Provider (DAP) is to provide the QPC with a uniform access mechanism to a remote data source. In this regard, the DAP might seem similar to a wrapper or a gateway. However, the DAP has an extensible query execution engine capable of loading and using application-specific code obtained from the network with the help of the QPC. Since a DAP is run at the data source site or in close proximity to it, MOCHA exploits this capability to push down to the DAP the code and computation of certain operators that "filter" the data been queried, and minimize the amount of data sent back to the QPC. This is a feature unique to MOCHA. Figure 3 shows the internal organization of a DAP. Query and procedural requests issued by the QPC are received through the DAP API, and routed to the *Control Module*, where they are decoded and prepared for execution. Each request contains information for the *Execution Engine* in the DAP that includes the kind of task to be performed (i.e. a query plan), the code that must be loaded, and the access mechanism necessary to extract the data. The execution engine first calls the *Code Loader* to load the required application-specific code, which is delivered to the DAP by the QPC through a mechanism that will be described in section 3.6. Then, it creates iterators for SQL or XML query requests, or prepares a procedural call to execute operations such as reading a file from a file system, requesting a Web page, etc. Notice that the iterators to access a source are built on top of standard Java packages such as JDBC, DOM (for XML repositories), Java Native Interface (JNI) and the I/O routines. Once the DAP has extended its query execution capabilities, it retrieves the data from the source, maps them into the middleware schema, and then filters them with the operators (if any) specified by the QPC in the query plan. The DAP then sends back to the QPC all values that it produced so they can be further processed to generate final results.

3.4 Data Server

The Data Server is the server application that stores the data sets for a particular data site. This element can be a full-fledged database server, a Web server or even a file server providing access to flat files. In the current MOCHA prototype, we provide support for object-relational database servers such as Informix and Oracle8i, XML repositories and file systems, since these are among the most commonly used servers to store the emerging complex data sets.

3.5 Catalog Organization

Query optimization and automatic code deployment are driven by the metadata in the catalog. The catalog contains metadata about views defined over the data sources, user-defined data types, user-defined operators, and any other relevant information such as selectivity of various operators. The views, data types and operators are generically referred to as "resources" and are uniquely identified by a Uniform Resource Identifier (URI). The metadata for each resource is specified in a document encoded with the Resource Description Framework (RDF), an XML-based technology used to specify metadata for resources available in networked environments. In MOCHA, for each resource there is a catalog entry of the form (`URI`, `RDF File`), and this is used by the system to understand the behavior and proper utilization of each resource. The reader is referred to [RMR00a] for more details about the structure of the URIs, the RDF metadata schema and other catalog management issues in MOCHA.

3.6 Automatic Code Deployment

In MOCHA, deploying code with application-specific components is done by shipping the compiled Java classes containing the implementation of data types and query operators. To simplify this discussion, we assume that each type or operator is entirely defined in only one Java class; but in general, their implementation can span several classes. When a system administrator needs to incorporate a new or updated data type or query operator into the system, he/she first stores the Java class for that resource into a well-known *code repository* (see Figure 4). Next, the administrator registers the new type or operator by adding entries into the system catalog that indicate the name of the type or operator, its associated URI, its RDF file, and any other relevant information such as version number, user privileges, etc. Once the code has been registered, the new functionality is ready for use in the queries posed by the users.

The automatic deployment of code starts after QPC receives a new query request from a client. The first task for the QPC is to generate a list with the data types and operators needed to process the query. QPC then accesses the metadata in the catalog to map each type or operator into the specific class implementing it. Each class is then retrieved from the code repository by the QPC's code loader and prepared for distribution. Before the actual execution of the query starts, QPC distributes the pieces of the plan to be executed by each of the DAPs running on the targeted data sites. Afterwards, the QPC starts the *code deployment phase* in which it first ships the classes for the data types to the client and to the DAPs, and then ships the classes for the query operators to be executed by each DAP. Figure 4 depicts how the class `AvgEnergy.class`, which implements func-

tion `AvgEnergy()`, would be shipped to a remote DAP. Once the code deployment phase is completed, QPC signals each DAP to activate its piece of the query plan, and only then, QPC and the DAPs start processing the data and generating results, all of which are gathered by QPC and sent back to the client for visualization purposes.

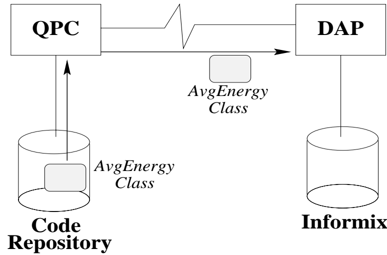


Figure 4: Shipping code for `AvgEnergy()`

It is important to emphasize that the code deployment phase occurs **on-line** as an automatic process carried out completely by the QPC without any human involvement. There is no need to restart any element in MOCHA before it can start using the functionality received during the code deployment phase. Instead, each of the QPC, DAPs and client application contain the necessary logic to load the classes into their Java run time systems and start using them immediately. Therefore, the capabilities of each element in MOCHA can be **extended** at run time in a dynamic fashion. Notice that the tasks for the administrators are simplified since they only need to deal with one or a few repositories where all the code resides. Upgrades or new functionality are simply added to the repository, and from there, they are deployed as needed by MOCHA. To the best of our knowledge, no other system implements this unique approach in which the middleware is self-extensible.

One very interesting issue that we are going to address in depth in the near future, is the possibility for a DAP to cache frequently used code so it can be reused many times without the need for repeated delivery. One simple solution is to have the QPC and DAP exchange information about the last known modification dates for the classes for types and operators already imported into a DAP. The DAP informs the QPC of all instances in which dates do not match, and the QPC only delivers the code for these cases.

3.7 Organization of Data Types

In MOCHA, the attributes in a tuple are implemented as Java objects. MOCHA provides a set of well-known Java type interfaces with the methods needed by the QPC, DAPs and client applications to handle the classes for data types correctly. Figure 5 shows the hierarchical structure of the type system for the MOCHA prototype. The dark rectangles represent type interfaces and the white ones represent Java classes for a particular type. At the root of the type hierarchy is the `MWObject` interface which identifies a class as one implementing a MOCHA data type, and also specifies the methods to be used to read/write each data value into the network.

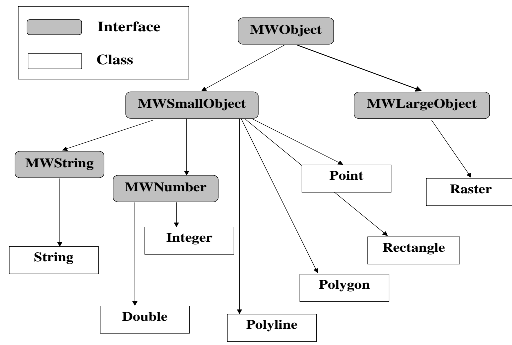
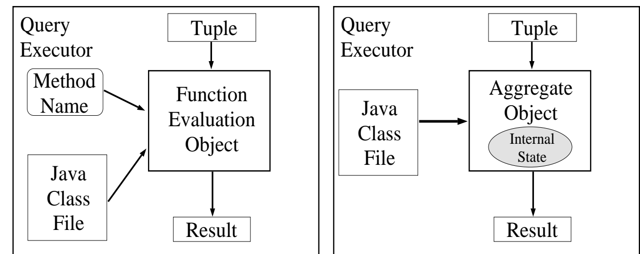


Figure 5: MOCHA Type System

The `MWLargeObject` and `MWSmallObject` interfaces extend `MWObject`, partitioning 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 or rectangles. Additionally, interfaces for character and numeric types are derived from `MWSmallObject`. Any new type added to the system must implement one of the interfaces below `MWObject`.

3.8 Organization of Query Operators

MOCHA groups query operators into two categories: 1) projections and predicates, and 2) aggregates. The complex functions present in projections and predicates are implemented as static methods in a Java class. Figure 6(a) shows how such functions are evaluated in the executor module contained by either QPC or a DAP. The query plan created by QPC indicates the class name and the method name associated with each function. The executor module uses this information to create a *function evaluation object*, which executes the body of the method and hence 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 6: Operator Evaluation in MOCHA

Aggregates are implemented as Java objects, as shown in Figure 6(b). The Java class for any aggregate operator implements the interface `Aggregate` provided by MOCHA. This interface specifies three methods that are used to compute the aggregate value: `Reset()`, `Update()` and `Summarize()`. For

each aggregate operator, the executor creates one aggregate object for each of the different groups found during the aggregation process. The internal state in an aggregate object is first set to an initial state by calling `Reset()`. Then, the executor successively calls `Update()` to modify the internal state in each object using the next tuple at hand and the current internal state. Once all the tuples are processed, the final value for each aggregate is extracted from the internal state in each aggregate object by calling `Summarize()`.

3.9 Implementation Issues

We now discuss three important implementation issues for MOCHA: memory management, communications over the network and security.

3.9.1 Memory Management

In Java, most of the memory management is done by the Java run time system, the so called Java Virtual Machine (JVM), and programmers do not need to worry about all the intricate low-level details regarding object allocation/deallocation. Unfortunately, these advantages are often offset by programming practices in which objects are excessively created, and then discarded after just one use! We found such practices in some JDBC drivers and proprietary Java database access APIs in which new objects are created to store column values each time a new tuple is read from the data source. Our experience with MOCHA proved that this paradigm is extremely inefficient for most database applications. The main reason is that object allocation involves calls to expensive synchronized methods in the JVM. Since possibly thousands of tuples are read from a data source during a query, the overhead of such calls has a devastating effect on performance. Moreover, as the number of objects allocated increases, the garbage collector might perform more work each time is called to dispose of the unused memory. Therefore, in MOCHA we adopted an aggressive policy of object preallocation and re-use. When an iterator is created by the execution engine, the constructor for the iterator creates one structure to read the columns from the database, and one structure to store the results to be returned by each call of the method `Next()` in the iterator. Thus, our implementation only creates these objects once and continuously re-uses them during the course of query processing.

3.9.2 Communications Over the Network

In our initial implementation of MOCHA we used the Java Remote Method Invocation (RMI) mechanism for the communications between the DAPs and the QPC. RMI is very similar to CORBA since it provides a communications interface based on method calls to remote object instances. RMI certainly made our implementation easy and elegant, but it proved too unstable and slow, specially when tuples containing complex and large types, such as images, where exchanged. RMI relies on stubs and skeletons to generate the remote method calls, and we found the data serialization protocol to be unstable, occasionally sending incorrect signals to the receivers, thus causing exceptions when the stubs attempted to unmarshall the data been exchanged. We also found that at the receiver's end, multiple objects were allocated each time a new tuple was read, an approach that is simply too inefficient, as we already discussed in section 3.9.1. We dealt with these problems by building our own communications infrastructure on top of the network sockets provided by Java. As result of this decision we needed to incorporate several methods in the `MWObject` interface to specify a generic mechanism to serialize the data in each tuple across the network. Although this required more effort on our part, we felt it was an essential task to guarantee that communications in MOCHA were stable, reliable and efficient.

3.9.3 Security

Since a client, QPC or DAP can dynamically load and execute compiled Java code, it is necessary to have a security mechanism to guarantee that the code does not executes dangerous operations on the host machines. In most object-relational engines, user-defined code is assumed to be "trusted", and it is the responsibility of the programmer to guarantee that the code is safe. In a large-scale environment, this kind of policy is unreasonable, and therefore, MOCHA leverages on the security architecture provided by Java. Administrators can configure the clients, QPC and DAPs to implement fine-grained security policies as supported by the `SecurityManager` class provided by Java. These policies include restrictions on the access to local file systems, allocation of network sockets, creation of threads, etc. Notice also that since the DAP is run as a process independent of the Data Server, a crash in a DAP will likely go unnoticed by the Data Server. It is important to realize, however, that security comes at the price of extra overhead. Each time an operation which the administrator defines as dangerous is attempted, a call to the security manager will be made to determine if the operation can proceed or not. Therefore, care must be taken to avoid a situation in which, for example, every call to an user-defined predicate triggers multiple calls to the security manager. In our view, security is a very important and complex issue in itself, deserving more careful exploration and done in close collaboration with the programming languages community, since the run time system must efficiently support the implementation of the security mechanisms.

4 Query Processing Framework

We have designed MOCHA to capitalize on its ability to ship code in order to generate query plans that minimize data movement. Following a cost-based approach, MOCHA pushes the evaluation of *data-reducing* operators to the DAPs running on the data sites and the evaluation of *data-inflating* operators to the QPC. The philosophy behind this scheme is that data movement typically is the major performance bottleneck in large-scale environments because network bandwidth is a shared resource, relatively expensive to upgrade, and the applications aggressively compete to obtain a frac-

tion of it. Notice that problems with heavy user loads on remote servers can be alleviated with replication, caching or even by upgrading to better server hardware since CPUs, memory and disks are becoming more powerful and less expensive. Therefore, MOCHA takes the pragmatic approach of first optimizing to minimize network costs.

In MOCHA, *data-reducing* operators are those operators that reduce the number and/or the size of the tuples in the result. Under this category we include: a) predicates with low selectivity, which filter out unnecessary tuples, b) predicates whose arguments are large-sized attributes that are not part of the result, c) projections that map large-sized attributes into scalars or smaller values, d) aggregates that map sets of tuples into a few distilled values and e) semi-joins which eliminate tuples that do not participate in a join. For example, the projection operator `AvgEnergy(image)` presented in section 2.2 is data-reducing because it maps a 1MB image into a 8-byte floating-point number. Whenever possible, data-reducing operators are evaluated by the DAPs, using a new query processing policy that we call **code shipping**. This policy specifies that a query operator and its code will be **shipped** to and executed by the DAP for a given data source. Code shipping can be viewed as query shipping enhanced with the capability to *materialize* the code for an operator remotely, as described in section 3.6.

On the other hand, *data-inflating* operators are those that inflate the data values and/or present them in many forms, projections, rotations, sizes and levels of detail. Recall the Earth Science application from section 2.1. Suppose a user from the State of Virginia now poses the following query to the data site in the State of Maryland:

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

This query retrieves all images from the table `Rasters` in Maryland, but function `IncrRes()` increases the resolution of each image by a factor 2X. Therefore, the projection `IncrRes(image, 2X)` is data-inflating since it synthesizes a new 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, for example, an operator that rotates an image by a certain degree θ without changing its size or one that allows a user to visualize a three-dimensional solid from different orientations (i.e. top, bottom or sideways). In these operators, the same data value is repeatedly transformed, and therefore, these transformations are more efficiently done near the client. For that reason, MOCHA executes data-inflating operators at the QPC using a data shipping strategy.

In MOCHA, each complex aggregate, predicate or projection operator Ω has an associated execution cost which has the general form:

$$Cost(\Omega) = CompCost(\Omega) + NetworkCost(\Omega)$$

Here $CompCost(\Omega)$ is the total cost of computing Ω over an input relation R. The $NetworkCost(\Omega)$ is total cost of data movement incurred while executing Ω on R. If Ω is evaluated at the DAP, then this component is the cost of sending to the QPC the results generated after applying Ω to all tuples in R. Otherwise, when Ω is evaluated at the QPC, this component is the cost of moving to the QPC each of the arguments to Ω in each of the tuples in R. The interested read can find more specific details and some of the exact cost formulas for each kind of operators in [RMR00b].

The cost of each operator Ω is used in the proposed optimization algorithm for MOCHA in order to find its proper execution placement. Before going into further details about the algorithm we need to introduce a new cost metric, the *Volume Reduction Factor* for an operator, which is used in the optimization process.

Definition 4.1 The **Volume Reduction Factor**, VRF, for an operator Ω over an input relation R is defined as:

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

where VDT is the total data volume to be transmitted after applying Ω to R, and VDA is the total data volume originally present in R.

Therefore, an operator Ω is *data-reducing* **if and only if** its VRF is less than 1; otherwise, it is *data-inflating*. In a similar fashion, we can define the *Cumulative Volume Reduction Factor* for a query plan P.

Definition 4.2 The **Cumulative Volume Reduction Factor**, $CVRF$, for a query plan P to answer query Q over input relations $R_1, ..., R_n$ is defined as:

$$CVRF(P) = \frac{CVDT}{CVDA} \qquad (0 \leq CVRF(P) < \infty),$$

where $CVDT$ is the total data volume to be transmitted over the network after applying all the operators in P to $R_1, ..., R_n$ and $CVDA$ is the total data volume in $R_1, ..., R_n$.

The intuition here is that the smaller the $CVRF$ of the plan, the less data it sends over the network, and the better performance the plan provides. This observation is validated by the results in sections 5.3-5.4.

Figure 7(a) shows the pseudo-code for the proposed System R-style optimizer for MOCHA. Consider, for example, the query: $\sigma_g(A) \bowtie \pi_f(\sigma(B))$, where predicate g is data-reducing and projection f is data-inflating. The algorithm first runs steps (1)-(3) to build selection plans for each of the expressions $\sigma_g(A)$ and $\pi_f(\sigma(B))$. Step (2) builds an initial plan with two nodes, one to be executed by the QPC (a QPC node), and one to be executed by the DAP (a DAP node) associated with the particular relation (A or B). At this point, the QPC node only has annotations that indicate the output to be returned, and the DAP node has annotations that indicate the attributes to be extracted. This initial plan is then modified in step (3) to add the user-defined operators that the middleware must execute. Figure 7(b) shows the algorithm used to place these complex operators, given as input a relation R and a plan P. First, the set of complex operators $\mathcal{O}$

```
procedure MOCHA_Optimizer(R_1, ..., R_n):
/* find best join plan */
1. for i = 1 to n do
2.    P ← selectPlan(R_i)
3.    optPlan(R_i) ← PlaceComplex(P, R_i)
4. for i = 2 to n do
5.    for all S ⊆ {R_1, ..., R_n} s.t. |S| = i do
6.       bestPlan ← dummy plan with infinite cost
7.       for all R_j, S_j s.t. S = {R_j} ∪ S_j do
8.          P ← joinPlan(optPlan(S_j), optPlan(R_j))
9.          P ← PlaceComplex(P, R_j)
10.         if cost(P) ≤ cost(bestPlan)
11.            bestPlan ← P
12.      optPlan(S) ← bestPlan
13. return optPlan({R_1, ..., R_n})
```

(a) System R-style Optimizer

```
procedure PlaceComplex(P, R):
/* find best operator placement */
1. O ← getComplexOps(P, R)
2. nDAP ← findDAP(P, R)
3. nQPC ← findAncestorQPC(P, nDAP)
4. for all Ω ∈ O do {
5.    if VRF(Ω) < 1
6.       insert(Ω, nDAP)
7.    else
8.       insert(Ω, nQPC)}
9. sortRank(nDAP)
10. sortRank(nQPC)
```

(b) Operator Placement

Figure 7: MOCHA Optimization Algorithm

that can be applied to the input relation is found with function $getComplexOps()$. Next, the DAP node used to access the relation R in plan P is found with function $findDAP()$. This DAP node is then used to find its nearest ancestor node in the plan P which also is a QPC node. Then, each operator Ω in the set $\mathcal{O}$ is placed at its best execution location based on its VRF value. Those operators with VRF less than 1 are placed at the DAP node, and the rest are placed at the QPC node. These heuristics serve to minimize the $CVRF$ of the plan P, and hence, its data movement and cost. In particular, they are used to produce plans with $CVRF$ less or equal to 1. Finally, the complex predicates added to each node are sorted based on increasing value of the metric: $rank(p) = (SF_p - 1)/CompCost(p)$, where SF_p is the selectivity of predicate p, as proposed in [HS93]. The result of this process on expressions $\sigma_g(A)$ and $\pi_f(\sigma(B))$ is shown on the left hand side of Figure 8. The gray nodes are QPC nodes, and the white ones are DAP nodes.

Once the single table access plans have been built, the algorithm in Figure 7(a) runs through steps (4)-(13) to explore all different possibilities to perform the join, incrementally building a left-deep plan in which a new relation R_j is added into an already existing join plan S_j for a subset of the relations. This task is done by function $joinPlan()$ in step (8). After the join plan is built, the algorithm again places complex operators in step (9). These are operators whose arguments come from more than one relation. Finally, the plan

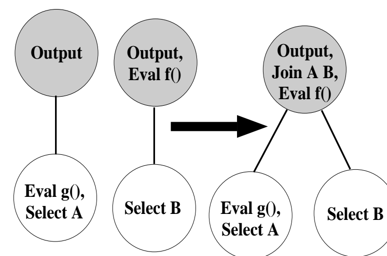


Figure 8: Optimization in MOCHA

P with smallest cost is selected. Here the cost of the plan includes the join cost and the evaluation costs of all complex operators. The final join plan for our example is shown on the right hand side of Figure 8. Notice that our algorithm is not exhaustive in terms of possible alternatives for complex operator placement (as is [CS96] for predicates). This is an intentional compromise done to avoid the extra combinatorial explosion of such an exhaustive search. At present, we have not completed the implementation of the cost-based query optimizer for the QPC although the major building blocks, such as query plans and search procedures, are in place. Since we have to deal not only with complex predicates, but also with complex projections and aggregates, we are exploring a series of pruning heuristics to reduce the search space of the optimizer, and speed up the optimization process.

5 Performance Evaluation

To validate our design choices and performance expectations for MOCHA, we benchmarked our prototype to characterize its behavior and show: a) the feasibility of using Java to implement types and operators, b) the benefits obtained by using code shipping, c) the need to use data shipping for data-inflating operators and d) that VRF is an accurate cost estimator for choosing the best query plan.

5.1 Benchmark Data and Queries

We used scientific data sets and queries from the Sequoia 2000 Benchmark [Sto93] to test our MOCHA prototype. Typically, these type of data sets are stored at different sites, and the applications manipulating them are inherently distributed. We used the regional version of the benchmark with data sets corresponding to the State of California, and Table 1 depicts the schema and other relevant information about these data sets. Similarly, Table 2 shows the queries used in our experiments, and each one is explained in the follow up sections. We derived these queries from the ones in Sequoia by adding and combining several new complex operators.

5.2 Experimental Methodology

We implemented the MOCHA prototype using Sun's Java Developers Kit 1.2, and all middleware data types and operators were implemented in classes containing 100% Java code. We used the Informix Universal Server and our own spatial *datablade* to provide support at the DBMS level for

Relation	Description	Cardinality	Size
`Polygons(landuse : Integer, location : Polygon)`	Polygons enclosing different types of land regions.	77,643	18.8MB
`Graphs(identifier : Integer, graph : Polyline)`	Graphs representing water drainage networks.	201,650	31MB
`Rasters(time : Integer, band : Integer, location : Rectangle, data : Raster, lines : Integer, samples : Integer)`	Satellite raster images made from weekly energy readings from Earth's surface regions. Each sample (pixel) represents a point on the surface region.	200	200MB

Table 1: Datasets

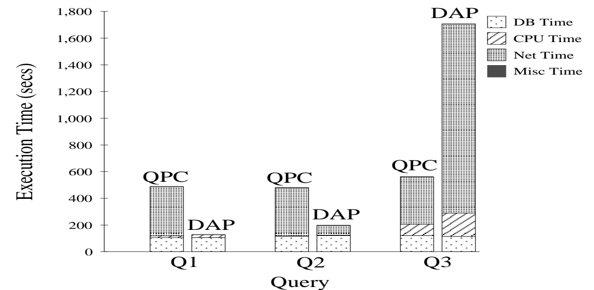
Q_1: SELECT landuse, TotalArea(location), TotalPerimeter(location) FROM Polygons GROUP BY landuse;	Q_2: SELECT time, band, location, Clip(data,lines,samples,WIN) FROM Rasters;
Q_3: SELECT time, band, location, IncrRes(data,lines,samples,2X) FROM Rasters;	Q_4: SELECT identifier FROM Graphs WHERE NumVertices(graph) > N AND ArcLength(graph) > S;
Q_5: SELECT R1.location, R1.time, R2.time, (AvgEnergy(R1.data) - AvgEnergy(R2.data)) FROM Rasters1 R1, Rasters2 R2 WHERE Equal(R1.location,R2.location);	

Table 2: Benchmark Queries

the data sets described in section 5.1. To provide connectivity between Informix and the DAP, we developed a JDBC-like library with support for the retrieval of complex types. In all the experiments discussed in this paper, we ran QPC on a Sun Ultra SPARC 60 with 128MB of memory. For the experiments in section 5.3, one DAP and Informix ran on a Sun Ultra SPARC 1 with 256MB of memory. In addition, for the experiment in section 5.4 we added a Sun Ultra SPARC 5 with 128 MB of memory, to run a second pair of DAP and Informix server. All machines ran Solaris 2.6 and were connected to a 10Mbps Ethernet network, and this choice of network was made mainly to obtain reproducible results. But in practice, we expect MOCHA to be run on wide area environments over which the available bandwidth would be much smaller, making MOCHA's benefits even more pronounced.

The main objective of this study is to clearly show the substantial performance benefits provided by a system that uses code shipping, such as MOCHA, over one that lacks this capability, and must rely heavily on data shipping. We configured QPC to use query plans that place all operators on either the DAP (with code shipping) or the QPC (with data shipping), which permits the study of each operator under each strategy. In each experiment, we ran the query plan on the MOCHA prototype and measured execution time from the time QPC starts deploying code to the time it receives the last tuple in the result. We present these results using graphs, in which the x-axis shows the query been tested and the y-axis gives its execution time under each strategy. Execution time was divided into four components: 1) **DB Time**- time spent reading tuples from the data source by DAP; 2) **CPU Time**- time spent evaluating all complex operators; 3) **Net Time**- time spent sending data from a DAP to QPC; and 4) **Misc Time**- time spent on all initialization and cleanup tasks. The **Net Time** component includes both network transmission time and the communications software overhead. All values reported as execution time are averages obtained from five independent measurements. In addition, we measured the total volume of data accessed by each plan ($CVDA$), the total volume of data transmitted by each plan ($CVDT$) and the volume of data in the query result. In each case, the $CVRF$ for each plan was computed from these measured values. We ran all experiments late at night, when all machines and the network were unloaded.

5.3 Queries Over a Single Data Source



(a) Execution Times

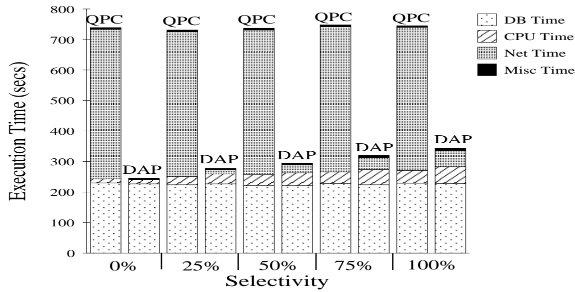
Query	Site	CVDA	CVDT	Volume in Result	CVRF
Q_1	DAP	18.8MB	740B	740B	$4x10^{-5}$
	QPC	18.8MB	18.8MB	740B	1
Q_2	DAP	200MB	40MB	40MB	0.20
	QPC	200MB	200MB	40MB	1
Q_3	DAP	200MB	800MB	800MB	4
	QPC	200MB	200MB	800MB	1

(b) Data Volumes

Figure 9: Performance for Q_1, Q_2 and Q_3

We used queries Q_1, Q_2 and Q_3 from Table 2 to measure the effect of complex aggregates and projections on the volume of data transmitted during query processing. Queries Q_1 and Q_2 contain data-reducing operators; query Q_1 is an aggregation query that computes the total area and total perimeter of all the polygons covering each type of land region. In Q_2, each image in table `Rasters` is clipped into an image whose size is determined by the clipping box `WIN`, which we chose so as to generate an image five times smaller than the original. On the other hand, Q_3 contains a data-inflating projection implemented by function `IncrRes()`. In this case, each image is transformed into a new image with twice the resolution and four times the size of the original. Figure 9(a) demonstrates that operator evaluation at the DAP with code shipping is the best option to execute Q_1 and Q_2, as it results

in performance improvements of 4-1 and 3-1, respectively. In each case, the performance gain is achieved by capitalizing on code shipping to run the data-reducing operators close to the data source and only send to the QPC a few result values. Figure 9(b) shows the large savings in the volume of data movement that can be obtained by using code shipping to evaluate these operators at the data site. However, code shipping is totally inadequate to run Q_3 at the DAP since this query contains a data-inflating operator. The evaluation of projection `IncrRes(data, lines, samples, 2X)` at the DAP results in the transmission of tuples four times larger than those sent when QPC executes it, and the network cost is increased by a factor of 4. Notice, however, that MOCHA will not use code shipping in this case, since any data-inflating operator will be evaluated at the QPC using data shipping. The results in Figure 9(b) emphasize the accuracy of the VRF in selecting the best operator placement, because in each case the best alternative is the one with the smallest $CVRF$.



(a) Execution Times

Selectivity	Site	CVDA	CVDT	Volume in result	CVRF
0	DAP	31MB	0B	0B	0
	QPC	31MB	31MB	0B	1
0.25	DAP	31MB	200KB	200KB	0.01
	QPC	31MB	31MB	200KB	1
0.50	DAP	31MB	400KB	400KB	0.01
	QPC	31MB	31MB	400KB	1
0.75	DAP	31MB	600KB	600KB	0.02
	QPC	31MB	31MB	600KB	1
1	DAP	31MB	790KB	790KB	0.02
	QPC	31MB	31MB	790KB	1

(b) Data Volumes

Figure 10: Performance for Q_4

Query Q_4 contains two complex predicates which compare the number of vertices and the total length of each drainage network against two constants. The execution times for Q_4 under various selectivity values are shown in Figure 10(a). As we can see, execution at the DAP outperforms execution at the QPC in all cases regardless of selectivity, with performance improved by a factor of 3-1 for the first three selectivity values and 2-1 for the remaining ones. By pushing the code and evaluation of the predicates in Q_4 to the DAP, the system avoids shipping the large-sized attribute `graph` over the network, as seen from Figure 10(b), thus providing substantial performance gains. Figure 10(b) shows that again the VRF is an accurate metric for determining the best plan for a query with complex predicates. One important concept emerging from the results in Figures 10(a)-10(b) is that an operator placement metric based on selectivity and result cardinality is not the best metric for cost estimation because it fails to take into account the volume of transmitted data, as happens in Q_4. Consider the case of 50% selectivity in Q_4. From Figure 10(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_4 the percentage of data transmitted is always much smaller than what would be expected if selectivity alone were used to make the estimate. The full set of results in [RMR00b] shows that selectivity might also under estimate the actual amount of data transferred. As a result, a query operator placement scheme based on selectivity and result cardinality might produce plans with poor performance for distributed queries because of these inaccuracies since these plans might transfer a larger (smaller) data volume than expected. In contrast, the VRF combines the selectivity information, cardinality and the size of the attributes in the tuples been transmitted to make a better estimate of the cost of a query operator and determine its proper placement.

5.4 Queries Over Multiple Data Sources

For this category we used query Q_5, from Table 2, which performs a distributed join between two tables, `Rasters1` and `Rasters2`. The schema for these tables is the same as the one for table `Raster` but the images where reduced to 128KB in size, and there are only three locations common to both of these tables. Table `Rasters1` was stored on a Sun Ultra SPARC 1, which we call Site1, while `Rasters2` was stored on a Sun Ultra SPARC 5, which we call Site2. Q_5 joins all tuples that coincide on the `location` attribute, and projects the location, the week number for each reading and the difference in the average energy between the readings.

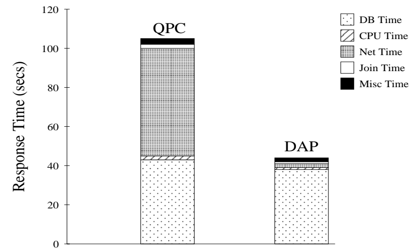


Figure 11: Execution Time for Q_5

Figure 11 shows the execution time for Q_5 for the alternatives in which complex operators in the join are executed at the QPC or at the DAP; the join itself is performed at the QPC. When complex operators must be executed at the QPC, attributes `time`, `location` and `data` in all tuples from each of the relations have to be moved to QPC. As tuples arrive, function `AvgEnergy()` is evaluated to perform the projections, and then the tuples are stored to disk,

from which they are later read to perform a hash join operation. In this figure, the **Join Time** component indicates the cost of accessing disk to perform this join operation. Notice that performance is dominated by the cost of transferring the images over the network. On the other hand, the join performs over two and a half times better when the DAP evaluates the complex operators. In this case, a 2-way semi-join can be performed by computing, at each DAP, the semi-joins `Rasters1`⋉`Rasters2` (at Site1) and `Rasters2`⋉`Rasters1` (at Site2), using the *complex predicate* in Q_5 for both of them. After each semi-join operation is performed, function `AvgEnergy()` is evaluated and all projections are taken. Thus, only attributes `time`, `location` and `AvgEnergy(data)` are moved from a DAP to QPC, where they are first materialized to disk and then joined. By using this strategy the network cost is minimized and the overall execution time of the query is substantially reduced. In terms of data movement, both plans access 30.6MB worth of data from the data sources, and produce a result of size 49.2KB. However, the first approach moves 30.6MB worth of data over the network, while the second approach only moves 3.8KB. This translates into a $CVRF$ values of 1 and 0.0001, respectively. Hence, experimental evidence confirms that the VRF can be used to determine the best operator placement in the plan for a distributed join.

6 Conclusions

We have proposed MOCHA as an alternative database middleware solution to the problem of integrating data sources distributed over a network. We have argued that the schemes used in existing middleware solutions, where user-defined functionality for data types and query operators is manually deployed, is inadequate and will not scale to environments with hundreds 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 such approach impractical. MOCHA, on the other hand, is a self-extensible middleware system written in Java, in which user-defined functionality is automatically deployed by the system to the sites that do not provide it. This is realized by shipping Java classes implementing the required functionality to the remote sites. 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 classifies operators as data-reducing ones, which are evaluated at the data sources, and data-inflating ones, which are evaluated near the client. Data shipping is used for data-inflating operators, and a new policy named code shipping is used for the 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 distributed queries. Our experiments with the MOCHA prototype show that selecting the right strategy and the right site to execute the operators can increase query performance by a factor of 4-1, in the case of aggregates, or 3-1, in the case of projections, predicates and joins. These experiments also demonstrated that the Volume Reduction Factor (VRF) is a more accurate cost metric for distributed processing than the standard metric based on selectivity factor and result cardinality because VRF also considers the volume of data movement.

References

[CGMH+94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proc. of IPSJ Conf.*, Tokyo, Japan, 1994.

[Inf97] Informix Corporation. Virtual Table Interface Programmer's Guide, September 1997.

[Ora99] Oracle Corporation. Oracle Transparent Gateways, 1999. http://www.oracle.com/gateways/html/transparent.html.

[CS96] S. Chaudhuri and K. Shim. Optimization of Queries with User-defined Predicates. In *Proc. 22nd VLDB Conf.*, pp. 87–98, Bombay, India, 1996.

[FJK96] M.J. Franklin, B.T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *Proc. ACM SIGMOD Conf.*, pp. 149–160, Montreal, Canada, 1996.

[GMSvE98] M. Godfrey, T. Mayr, P. Seshadri, and T. von Eicken. Secure and Portable Database Extensibility. In *Proc. ACM SIGMOD Conf.*, pp. 390–401, Seattle, WA, USA, 1998.

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

[HKWY97] L.M. Haas, D. Kossmann, E.L. Wimmers, and J. Yans. Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd VLDB Conf.*, pp. 276–285, Athens, Greece, 1997.

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

[ML86] L.F. Mackert and G.M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. 12th VLDB Conf.*, Kyoto,Japan, 1986.

[MS99] T. Mayr and P. Seshadri. Optimization of client-site user-defined functions. In *Proc. ACM SIGMOD Conf.*, Philadelphia, PA, USA, 1999.

[RMR00a] M. Rodríguez-Martínez and N. Roussopoulos. Automatic Deployment of Application-Specific Metadata and Code in MOCHA. In *Proc. 7th EDBT Conf.*, Konstanz, Germany, 2000.

[RMR00b] M. Rodríguez-Martínez and N. Roussopoulos. MOCHA: A Self-Extensible Database Middleware System For Distributed Data Sources. Technical Report UMIACS-TR 2000-05, CS-TR 4105, University of Maryland, January 2000.

[RS97] M.T. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *23rd VLDB Conf.*, Athens, Greece, 1997.

[SAC+79] P.G. 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 Conf.*, pp. 23–34, Boston, MA, USA, 1979.

[SLR97] P. Seshadri, M. Livny, and R. Ramakrishnan. The Case for Enhanced Abstract Data Types. In *Proc. 23rd VLDB Conf.*, pp. 66–75, Athens, Greece, 1997.

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

[TRV96] A. Tomasic, L. Rashid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of DISCO. In *Proc. 16th ICDCS Conf.*, Hong Kong, 1996.