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

Manuel Rodríguez-Martínez[†] Nick Roussopoulos
manuel@cs.umd.edu nick@cs.umd.edu
Institute for Advanced Computer Studies & Department of Computer Science
University of Maryland, College Park, MD 20742

To appear in the Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, May 2000

Abstract

This paper describes MOCHA, a new self-extensible database middleware system 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 some of the user-defined functionality in the system should be deployed by the middleware 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 data and queries from the Sequoia 2000 Benchmark. The results of this study demonstrate that MOCHA not only provides a flexible and scalable framework for distributed query processing but also substantially improves query performance in contrast to existing middleware solutions.

1 Introduction

The purpose of database middleware[1] systems is to integrate collections of data sources distributed over a computer network. Typically, these systems follow an architecture centered around a *data integration server*, which provides client applications with a uniform view and a uniform 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. This global model can either be relational, object-oriented or object-relational, depending on the needs of the targeted applications and the capabilities of the integration server. 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

[†]Contact author. Phone: (301)-405-2714, Fax: (301)-405-6707
[1]In this paper we will use the terms database middleware system and middleware interchangeably.

second approach, a mediator server specially designed and tailored for distributed data processing is used as the integration server. The mediator server utilizes the functionality of *wrappers* to gain access to the information stored in each data site. A wrapper extracts the information from a data source and translates it into the global data model specified by the mediator. In both of these existing 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. Usually, the integration server is run close to the client application, on what is called the *integration site*. The wrapper or gateway can be run either on the integration site (either as a function call or as a separate process), or on the site where the data source resides. 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 of database middleware which have received little attention from the research community: 1) the deployment of the application-specific functionality[2], 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 S**H**ipping **A**rchitecture) [RMR98a, RMR98b], 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-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 middleware solutions, in which administrators need to manually install all the required functionality throughout the entire 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 been shipped is much more smaller than the data sets, automatic code deployment facilitates query optimization based on data movement reduction. Notice that since network

[2]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.

bandwidth typically is the major performance bottleneck in distributed processing, our approach can reduce query execution time by minimizing the overall time spent on data transfer operations. 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 Spring of 1998[3]. MOCHA is currently been considered as a middleware solution for the NASA Earth Science Information Partnership (ESIP) facility at the University of Maryland. MOCHA is written in Java and supports major database servers, such as Oracle and Informix, file servers and even XML repositories. We argue that MOCHA provides 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, 3-1 in the case of projections and predicates, and close to 3-1 in the case of distributed joins. These experiments were carried out on the MOCHA prototype using realistic 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 query processing framework used in MOCHA. Section 5 contains a performance study of the MOCHA prototype. Finally, our conclusions for this paper are presented in section 6.

2 Motivation for MOCHA

Given the continuous growth of the Internet and the ever increasing number of corporate Intranets and Extranets, database middleware will be required to interconnect hundreds or more of data sites deployed over these type of 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. And since the World Wide Web has become the de facto user-interface for networked applications, end-users will demand a middleware solution that easily integrates Web-based clients to visualize these data sets. Given this scenario, we argue that middleware solutions for these kind of environments will be successful only if they can provide: a) scalable, efficient and cost-effective mechanisms to deploy and maintain the application-specific functionality throughout the entire system, and b) 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

[3]The actual implementation started in the Summer of 1997.

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 involved in maintaining the software throughout the entire 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:

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

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, scientists are frequently experimenting with new or improved methods to compute properties about the data, such as averages on the amount of infrared energy absorbed by the Earth's surface. Therefore, many users will not be satisfied with what 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 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. As a result, the existing middleware solutions follow a query processing approach in which most operators in a query are completely evaluated by the integration server [Ora99, Inf97, CGMH+94, TRV96]. The tasks for the wrappers and gateways include the extraction of the data items from the sources and their translation into the middleware schema for further processing at the integration site. In many situations, this scheme can 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 in the State of Virginia poses the following query for 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. Function `AvgEnergy()` returns a 8-byte double precision floating point number, so the size of the records in the result is just 28 bytes. Under the approach of query processing at the integration server, this query is evaluated by 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. Then, function `AvgEnergy()` is evaluated, the qualification clause in the query is tested and the projections are performed. This approach for query processing is called *data shipping* [FJK96], since the data is moved from the source to the site where the query was posed. 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[4]. On the other hand, if the query is processed at the data source then data movement is negligible. In a worst-case scenario in which all the tuples in table `Rasters` satisfy the qualification clause, 200 tuples $\times$ 28 bytes is approximately 5KB! In this alternative approach, called *query shipping* [FJK96] , the query is executed at the data source and only the final result is sent to the query site. However, this approach is feasible **only if** the necessary operators and query processing capabilities are available at the data source.

The work in [FJK96] advocates for a hybrid approach in which both data and query shipping are combined, in order to utilize the best alternative for the situation at hand. Unfortunately, query shipping is very difficult to realize in a large scale environment when user-defined types and operators are present in a query since these might not be available everywhere. 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. Other systems, in particular Garlic[RS97], attempt to push the evaluation of a query operator to a data site, but **only if** the operator exists *a priori* at the data site. Otherwise, the Garlic approach is not viable and processing must be done at the integration site. In our example, it would be necessary for function `AvgEnergy()` to be already implemented in the DBMS running at the data source before the query is even posed. Clearly, this is a sound technique, can be very powerful and ought to be exploited by every middleware solution, but its main limitation is that it is highly improbable that all the functions ever needed to evaluate queries in a large system will be present in each and every source. The end result is that in many important situations the system will be **restricted** to use an inefficient query processing strategy simply because the functionality required to use a superior strategy is not available where it is needed. Thus, the query processing framework used in existing solutions falls short from providing an efficient and scalable service for a large-scale environment in which users have diverse needs for data processing.

[4]Many wide area networks are built on top of Gigabit networks, such as ATM. However, this bandwidth must be shared by all the traffic, and as a result each connection only receives a fraction of the bandwidth that can be sustained by the network.

3 MOCHA Architecture

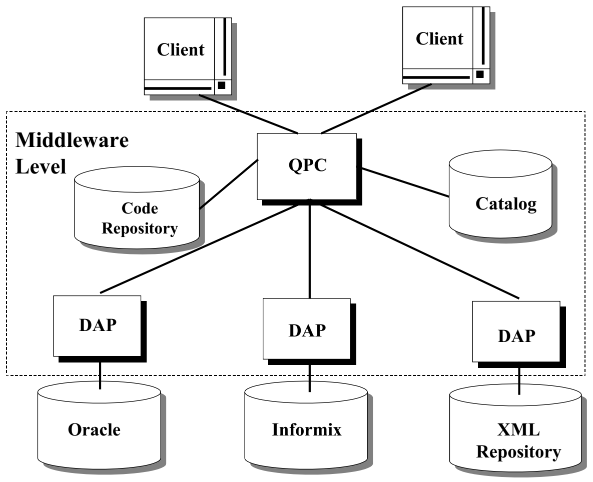


Figure 1: MOCHA Architecture

To overcome the shortcomings in previous middleware solutions, we have designed the architecture of MOCHA around two key principles. The first one is that all the application-specific functionality needed to process any given query is to be delivered by MOCHA to all interested sites in an **automatic** fashion. This is realized by shipping the Java classes containing the required functionality. In this aspect, MOCHA is a *network-centric* system, in which the data processing capabilities needed by the client or server sites can be dynamically obtained from the network, instead of requiring a manual installation by an administrator. 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 effect 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.

3.1 Client Application

To provide support for a wide spectrum of users, MOCHA supports three kinds of client applications: 1) applets, 2) serverlets 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. Figure 2 shows a screen shot of one of our demo client applets built for the NASA ESIP Facility at the University of Maryland. This applet allows users to extract and visualize AVHRR raster images from land regions, and polygons covering features such as cities, lakes or states. Now, since there might be users who prefer or require a purely HTML-based interface, or who prefer a non-Java client application, MOCHA provides the alternative of using a serverlet as entry point into the system. The serverlet receives the query requests and dynamically generates HTML documents encoding the query results obtained from MOCHA. Alternatively, all encoding can be done through XML, which can be converted into HTML as needed to present the query results. Finally, stand-alone Java applications will likely be used by users such as system administrators or software developers, who will need to carry out complex tasks such as system configuration and tuning, data cleaning and/or backups, catalog management and distributed software debugging, to name a few. MOCHA provides a set of Java libraries containing the APIs needed by the client to pose queries to the system and retrieve their results, send procedural commands, and request diagnostic information or metadata about a particular resource. In addition, the APIs contain the infrastructure needed to load the code containing the application-specific components necessary to manipulate the data produced by the queries. All of these components are automatically delivered to the client by MOCHA at run time.

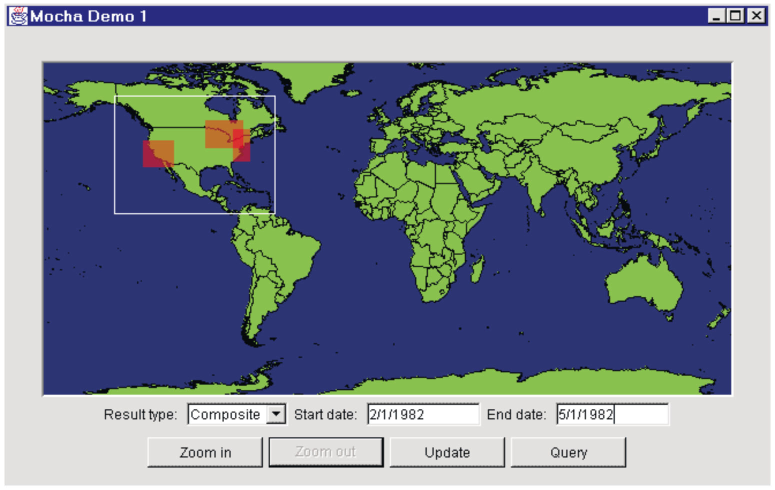


Figure 2: NASA ESIP Client

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. The QPC can be reached through a well-known *Uniform Resource Locator* (URL). 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 a 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. The internal components of the QPC are depicted in Figure 3. The Client API serves as the entry point to accept the requests from a client application. In MOCHA, the QPC offers three main data processing services. The first one provides access to distributed data sites which are modeled as object-relational sources. These sources can be full-fledged database servers, image servers, flat file repositories or any other kind of data source which can be modeled in a relational fashion. 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 preprocessed by the *SQL parser* in the QPC. The second data processing service provided by QPC

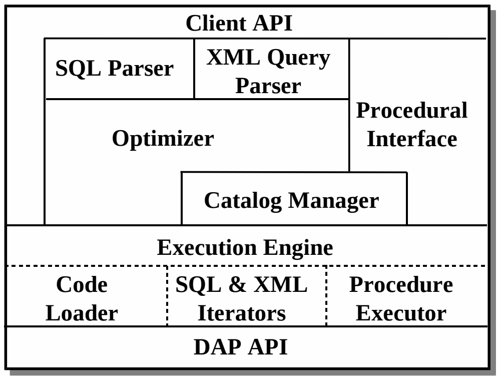


Figure 3: Organization of the QPC

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 parse XSL 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 channeled to gain access to the data in these sources. We would like to point out that it is entirely possible to add more functionality to the QPC in order to provide an object-oriented modeling abstraction on top of the data sources. However, the goal of our research is not data modeling, but rather the deployment and efficient use of the application-specific query processing components in middleware systems. Therefore, we chose to leave the implementation of object-oriented capabilities for future work.

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 exact structure in [RMR00]. The QPC uses the services of the *Catalog Manager* module to retrieve from the catalogs 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 *Code Loader* module in the execution engine is used to extract the required code from a code repository, and prepare it for deployment. All the communications with the remote sources occur through the facilities embedded in the DAP API.

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 sense, the DAP might seem very similar to a wrapper or a gateway. However, the DAP has an extensible query execution engine, capable of loading and using application-specific code that is obtained from the network with the help of the QPC. Since a DAP is run at the data source 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. To the best of our knowledge, none of the existing solutions has implemented this unique approach. Figure 4 shows the internal organization of a DAP. Like a QPC, a DAP can be reached via a URL. 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, which includes the kind of task to be performed (i.e. a query plan), the code that must be loaded into the run time system 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 and XML query requests, or prepares a procedural call to execute operations such as reading a data file from a file system, requesting a Web page or some other type of command. Notice that the iterators to access a source are built on top of Java standard packages such as JDBC, DOM (for XML repositories), Java Native Interface (JNI) and I/O file routines. Once the DAP has extended its query execution capabilities, it starts to carry out its data processing tasks. The data retrieved are first mapped into the middleware schema, and then filtered 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 in order to generate a final result.

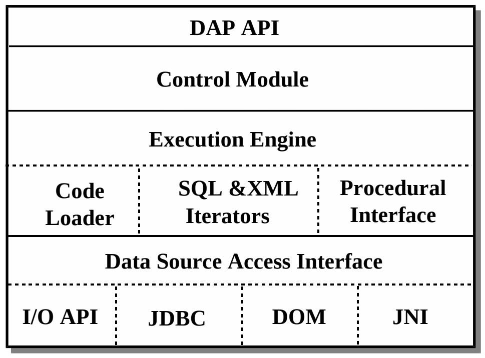


Figure 4: DAP Organization

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 and even a file server providing access to flat files. In the current MOCHA prototype implementation, 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 the definitions of views defined over the data sources, user-defined data types, user-defined operators, and 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). In MOCHA, URIs are of the form: `mocha://<host>/<source|repository>/<name>`. Keyword `mocha` identifies the URI as a one used by MOCHA. The `<host>` portion identifies the host for a data source or code repository, and `<source|repository>` indicates the specific name of a data source (i.e. database name) or code repository. Finally, the `<name>` part gives the particular name for the resource. For example, the URI for function `AvgEnergy()` is structured as: `mocha://cs1.umd.edu/EarthScience/AvgEnergy`, and this denotes that the function belongs to the `EarthScience` repository located in host `cs1.umd.edu`.

The metadata for each resource is specified in a document encoded with the Resource Description Framework (RDF). RDF is 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. For example, the RDF file for function `AvgEnergy()` indicates the name of the Java class implementing it, the type of arguments expected and its return type. The interested reader is referred to [RMR00] for more details about the syntax of the metadata files.

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. In general, however, a data type can contain several other types, and therefore, its implementation can span several classes. The same argument holds for query operators. In addition, we base our discussion on query requests, but all the ideas also apply to procedural requests as well.

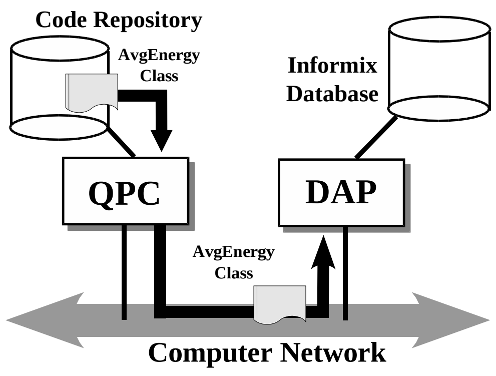


Figure 5: Shipping code for `AvgEnergy()`

When a system administrator needs to incorporate a new or updated data type or query operator into the system, he/she first stores the class for that type or operator into a well-known *code repository* (see Figure 5). 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 and its RDF file. Other relevant information, such as version number, user privileges, etc., can be added this way. After the code has been registered, the new functionality is ready for use in either compiled or ad-hoc queries issued by the users. Therefore, the administrator might have to periodically inform the users about new functionality added into the system. This can be achieved by sending an electronic message or updating a Web page describing the available new code. Clearly, some kind of repository management is needed.

Turning now to the automatic deployment of code, it all starts when QPC receives a 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 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 5 depicts how the class `AvgEnergy.class`, which implements function `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. As mentioned before, all results are gathered by QPC and sent back to the client for visualization purposes.

It is important to emphasize that the code deployment phase occurs **on-line** and with no human involvement, as an automatic process carried out completely by the QPC, using the metadata in the system catalog. It is not necessary 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 contains the necessary logic to load the classes into their Java run time systems and immediately start using them. Therefore, the capabilities of each element in MOCHA can be **extended** at run time in a dynamic fashion. 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 6 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 data type for use in MOCHA. This interface specifies the methods to be used to read and write each data value into the network. In addition, it specifies a method used to update the value in a Java object from the value of another object of equivalent type. The rationale for these methods will be discussed in sections 3.9.1 and 3.9.2. 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`.

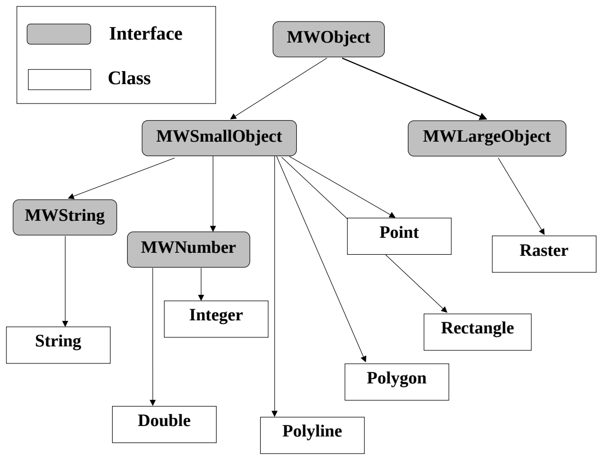


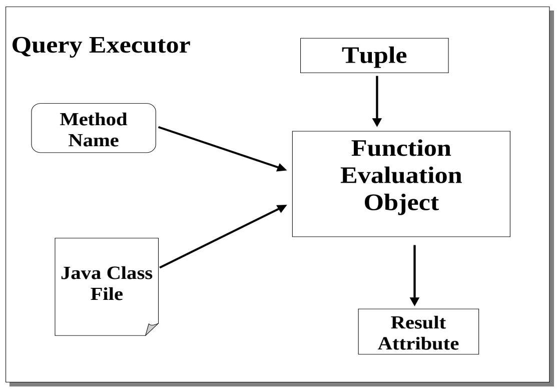
Figure 6: MOCHA Type System

3.8 Organization of Query Operators

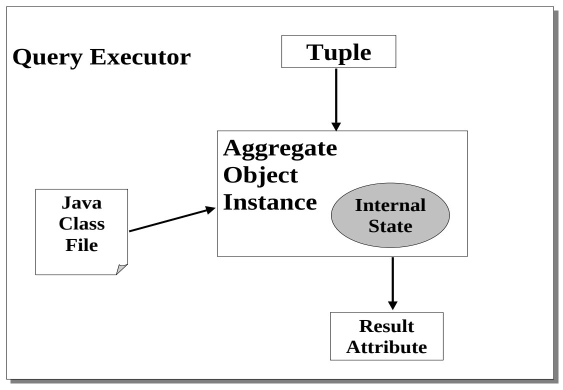
MOCHA groups query operators into two categories: 1) projections and predicates, and 2) aggregates. The complex functions present in projections and predicates are implemented as static methods in a Java class[5]. Figure 7(a) shows how such functions are evaluated in the executor module contained by either QPC or a DAP. The query plan created by QPC indicates the class and the method 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

[5]These are methods that can be evaluated by the Java run time system without first creating a new object from a class.

object and collects the resulting attributes for further processing or adds them into the result.



(a) Predicates & Projections



(b) Aggregates

Figure 7: Operator Evaluation in MOCHA

Aggregates, on the other hand, are implemented as Java objects, as shown in Figure 7(b). MOCHA provides interface `Aggregate` which specifies three methods to be implemented by any class used to create an aggregate: `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 object by calling `Summarize()`.

3.9 Implementation Issues

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

3.9.1 Memory Management

One of the major selling points for Java is the fact that most of the memory management is done by the Java run time system, the so called Java Virtual Machine (JVM). Contrary to languages such as C or C++, Java programmers do not need to worry about all the intricate details regarding the allocation and deallocation of objects, alignment of structures around word boundaries, and the much dreaded pointer arithmetic. Unfortunately, these advantages are often offset by programming practices in which objects are excessively created, and then left to the garbage collector for clean up 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 tuple is read from the 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 synchronized methods in the JVM, which are very expensive as they involve context switches between the thread that allocates memory and the user's thread(s). 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. The net effect is an implementation which spends most of its time managing memory, instead of computing the results of the query! Therefore, in MOCHA we adopted an aggressive policy of object preallocation and reuse. 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 `Next()` method in the iterator. Thus, our implementation only creates objects once and continuously reuses them during the course of query processing. This is the rationale behind the `resetValue()` method in the `MWObject` interface. This method allows the system to reuse objects already allocated to store the values of the columns in the tuples been manipulated by MOCHA.

3.9.2 Communications

Java provides the Remote Method Invocation (RMI) package for communications, which is similar to CORBA, and its main benefits are that it takes care of all the internals of communications and provides an interface based on method calls to remote object instances. In our initial implementation of MOCHA, we used RMI as the communication mechanism between the DAP and the QPC. 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. On the one hand, RMI relies on stubs and skeletons (like RPC and CORBA) to generate the remote method calls. We found the serialization protocol to be unstable, occasionally sending the wrong signals to the receivers, thus causing exceptions when the stubs attempted to unmarshall the data been exchanged. On the other hand, we found that at the receiver's end, multiple objects were allocated each time a tuple was read, and as we already discussed in section 3.9.1, such an approach is simply too inefficient. We dealt with these problems by building our own communications infrastructure on top of the network sockets provided by Java. This certainly required more effort on our part, but we felt it was an essential task in order to make communications in MOCHA stable, reliable and efficient. As result of this decision, we needed to incorporate methods `readNetwork()` and `writeNetwork()` in the `MWObject` interface in order to provide a generic mechanism to transmit the data values across the network. Obviously, this puts more work on the programmer since he/she has to worry about how to "serialize" their types over the network. Nevertheless, we believe that this task is not an onerous one since our interfaces simply require a field by field write or read operation on a network stream.

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. Notice that 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 is 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, and access to resources such as printers. Also notice 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. Thus, care must be taken to avoid a situation in which, for example, every call to a 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.

3.10 Benefits of the MOCHA Architecture

MOCHA provides a very flexible framework to deploy application-specific functionality over a large-scale environment. By implementing the functionality in Java classes, we avoid the problems associated with porting code to different platforms. All classes are compiled just once and automatically delivered by MOCHA to the sites that need them. 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. Also, by organizing the code into repositories, from which it is deployed to remote sites, it becomes easier to add new functionality, keep track of versions and apply updates. The tasks of the administrators are greatly reduced since they have to deal with just one or a few well-known sites where all the code resides. Upgrades and new functionality are simply added to the appropriate code repository, and from there, they are deployed by MOCHA as needed. This methodology is by far superior to the approach where libraries must be installed at any site where a client, wrapper, gateway or integration server is to be run. Another important benefit of MOCHA is the ease to manipulate new functionality added into the system. The use of well-known interfaces allows QPC, DAP and the client to manipulate any type or operator without understanding all of its internal details. For example, the data transmission module in QPC handles every type as a `MWObject`, and calls the appropriate methods defined in this interface to exchange the data values over the network. Similarly, all aggregates are manipulated through the `Aggregate` interface. Thus, there is no need to "hard code" special routines to deal with each type or operator. Instead, each class customizes the methods in the appropriate interfaces to work according with its intended behavior.

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, and the applications aggressively compete to obtain a fraction of it. To overcome this problem, MOCHA attempts to move the computation to the data whenever it is cost-effective. Notice that problems with heavy user loads on remote servers can be alleviated with replication, caching or even by upgrading to better server hardware, given that CPUs, memory and disks are becoming more powerful and less expensive. Networks, on the other hand, are not as easy and cost-effective to upgrade, and even an upgrade might only be feasible in small local area environments. Also notice that retrieving the "raw" data from an already loaded server to do the processing at the client adds further load to the server, and it might end up been worse than filtering the data first. In short, MOCHA takes the pragmatic approach of first optimizing for the common case in which the network is the bottleneck.

In MOCHA, the *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 a data-reducing operator 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 data 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. One can interpret code shipping 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, the *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. Thus, the projection `IncrRes(image, 2X)` is a data-inflating operator 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. Examples of these include 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.

We now turn to the discussion of the cost of the user-defined operators. In each case, cost is defined as computation time plus data transmission time. We first consider operators evaluated at the DAP using code shipping. Given an input relation R, an operator Ω, a set argument attributes $A_1, ..., A_n$, and a sustained network bandwidth NB between the QPC and the DAP, the cost of Ω is defined as:

$$Cost(\Omega) = \begin{cases} |R| * (comp(\Omega) + (\frac{1}{NB} * size[\Omega(A_1, ..., A_n)])), & \text{if } \Omega \text{ is a projection} \\ |R| * comp(\Omega) + (G * \frac{1}{NB} * size[\Omega(A_1, ..., A_n)]), & \text{if } \Omega \text{ is an aggregate} \\ |R| * comp(\Omega) + SF_\Omega * \sum_{j=1}^{k} Cost(B_j), & \text{if } \Omega \text{ is a predicate} \end{cases}$$

where $comp(\Omega)$ is the per-tuple cost of evaluating Ω, and $size(\alpha)$ is the size of an attribute α. Thus, when

Ω is a projection, its total cost is the time spent computing and transmitting the values $\Omega(A_1, ..., A_n)$.Notice that for a simple projection of an attribute A, $comp(\Omega)$ is equal to zero. If Ω is an aggregate, then its total cost is the time spent computing the values $\Omega(A_1, ..., A_n)$ over the input relation, plus the time spent transmitting the values for each of the G different groups generated by the query. Finally, whenever Ω is a predicate, its cost is the time evaluating it over relation R, plus the product of its selectivity SF_Ω times the sum of the cost of each of the $B_1, ..., B_k$ projected attributes in the resulting tuples, assuming none of them is an aggregate. In all cases, computation time and size of a result attribute are determined from metadata stored in the catalog. Now we turn to the discussion of the case where an operator is evaluated at QPC. For this case, it is necessary to move all the required attributes from each tuple in the input relation R from the remote data site to QPC, and the overall cost is: $Cost(\Omega) = |R| * (comp(\Omega) + \frac{1}{NB} * \sum_{i=1}^{n} size(A_i))$. Thus, for every operator, its cost is equal to the time spent computing it, plus the time spent fetching its arguments from the source.

Having discussed how to estimate the cost of each complex operator in a query, we move now to the presentation of the proposed optimization algorithm for MOCHA. But first, 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 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 5.4-5.7.

Figure 8(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 the expressions $\sigma_g(A)$ and $\pi_f(\sigma(B))$. Step (2) builds an initial plan with two nodes, one for the QPC (a QPC node), and one for 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 and the operators to be evaluated by the data source. This initial plan is then modified in step (3) to add the user-defined operators that the middleware must execute. Figure 8(b) shows the algorithm used to place complex operators, given an input relation R and plan P. First, the set of complex operators $\mathcal{O}$ that

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

(a) System R-style Optimizer

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

(b) Operator Placement

Figure 8: MOCHA Optimization Algorithm

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 that also is a QPC node. Then, each operator Ω in the set $\mathcal{O}$ is placed in 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. Finally, the complex predicates added to each node are sorted based on increasing value of the metric: $rank(p) = (SF_p - 1)/Cost(p)$, where SF_p is the selectivity of 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 9. 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 8(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. The final join plan for our example is shown on the right hand side of Figure 9. 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. In MOCHA, not only do we have

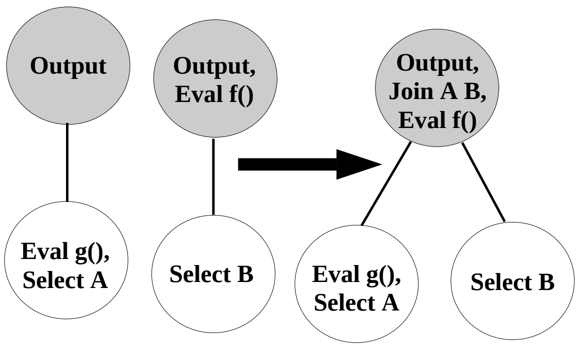


Figure 9: Optimization in MOCHA

to deal with the placement of complex predicates [HS93, CS96] but also deal with complex projections and aggregates as well, thus making the optimization process more complicated. 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 Experimental Setup

The current implementation of MOCHA was made using Sun's Java Developers Kit 1.2, and all middleware types and operators were implemented in classes containing 100% Java code. We used the Informix Universal Server as our data server and installed into it a *datablade* to provide support at the DBMS level for the types in the schema described in section 5.2. To provide connectivity between Informix and the DAP, we developed a JDBC-like library with support for complex types. This library translates the attributes from their native DBMS representation into their corresponding middleware representation. This library was integrated with the client libraries provided by Informix using the functionality of the Java Native Interface (JNI). In all the experiments discussed in this paper, we ran QPC on a Sun Ultra SPARC 60 running Solaris 2.6 and with 128MB of memory. For the experiments in sections 5.4 through 5.6, one DAP and Informix ran on a Sun Ultra SPARC 1 running Solaris 2.5 and with 256MB of memory. For the experiment in section 5.7, we added to this setup a third machine to run a second pair of DAP and Informix server. This machine was a Sun Ultra SPARC 5 running Solaris 2.6 and with 128 MB of memory. Both Informix servers stored all their data using raw partitions on two 9GB Seagate Ultra SCSI disks, one disk per machine. All machines were connected to a 10Mbps Ethernet network, and this choice of network was made mainly to obtain reproducible results. In practice, however, we expect MOCHA to be run on wide area environments over which the available bandwidth would be much smaller, and therefore, MOCHA's benefits much more pronounced.

5.2 Benchmark Data and Queries

We used the Sequoia 2000 Benchmark [Sto93] to test our implementation of MOCHA. Sequoia contains real data describing geographic information about the Earth. Typically, these data sets are stored at different sites, and the applications manipulating them are inherently distributed. We used the regional version of the benchmark, which corresponds to the State of California. The schema we used is as follows:

• $\texttt{Points}(\texttt{name} : \texttt{String}, \texttt{location} : \texttt{Point})$ - geographic locations represented by their name and coordinates. This table had 62,556 tuples worth 4.3MB.

• $\texttt{Polygons}(\texttt{landuse} : \texttt{Integer}, \texttt{location} : \texttt{Polygon})$ - polygons enclosing regions of land. Each tuple stores the type of land been enclosed (i.e. forest, city, etc.) and an array with the exterior vertices of the polygon. This table had 77,643 tuples worth 18.8MB.

• $\texttt{Graphs}(\texttt{identifier} : \texttt{Integer}, \texttt{graph} : \texttt{Polyline})$ - graphs representing drainage networks. Each tuple stores an identifier and an array with the line segments that make up the graph. This table had 201,650 tuples worth 31MB.

• $\texttt{Rasters}(\texttt{time} : \texttt{Integer}, \texttt{band} : \texttt{Integer}, \texttt{location} : \texttt{Rectangle}, \texttt{data} : \texttt{Raster}, \texttt{lines} : \texttt{Integer}, \texttt{samples} : \texttt{Integer})$ - satellite raster images made from weekly energy readings taken from regions of the Earth's surface. Each tuple stores the week number, the energy band measured, the rectangle enclosing the region, the raster image, the number of lines in the image and the number of samples (pixels) per line. Each sample is a 16-bit integer and each image is 1MB in size. This table had 200 tuples worth 200MB.

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 identifier
	FROM Graphs
	WHERE NumVertices(graph) > N
	AND ArcLength(graph) > S;
		Q_9: SELECT landuse, location
		FROM Polygons
		WHERE Perimeter(location) > N;

Q_{10}: 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 1: Benchmark Queries

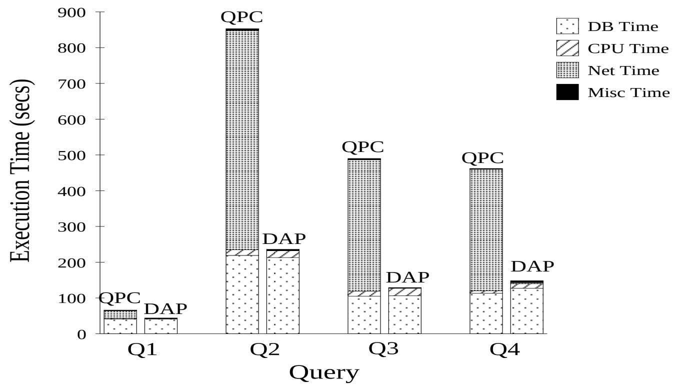
Our benchmark queries are shown in Table 1, and each one is discussed in the next sections. We derived these queries from the ones in Sequoia by adding and combining several new complex operators.

5.3 Experimental Methodology

The 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. Without code shipping, the middleware integration server has to evaluate all operators not available at the data sources. Thus, our goal is to showcase code shipping, not to further explore the query-data shipping tradeoffs already throughly discussed in [FJK96]. To show the benefits of MOCHA in this important and frequent situation, we configured QPC to use query plans that place all operators on either the QPC or the DAP. This permits the study of each kind of operator (data-inflating or data-reducing) under each approach. 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 policy. 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.

5.4 Queries with Complex Aggregates

For this category we used queries Q_1, Q_2, Q_3 and Q_4 from Table 1 on page 19. Queries Q_1 and Q_2 map an entire table into a single value; Q_1 finds a minimum bounding box for all points in table `Points`, while Q_2 computes the total length of all drainage networks in `Graphs`. Queries Q_3 and Q_4 contain GROUP BY clauses, with Q_4 having attribute `location` with complex type `Rectangle` as the grouping attribute. Query Q_3 computes the total area and total perimeter of all the polygons covering each type of landuse. In Q_4, for each of the regions present in table `Rasters`, function `CompositeImage()` generates an image which is the composition of all images stored for each particular region. Before calling this function, each image is clipped to a region specified by parameter `WIN`, which we set to select a piece five times smaller than the original.



(a) Execution Times

Query	Site	CVDA	CVDT	Volume in Result	CVRF
Q_1	DAP	4.3MB	16B	16B	$4x10^{-6}$
	QPC	4.3MB	4.3MB	16B	1
Q_2	DAP	31MB	8B	8B	$2x10^{-7}$
	QPC	31MB	31MB	8B	1
Q_3	DAP	18.8MB	740B	740B	$4x10^{-5}$
	QPC	18.8MB	18.8MB	740B	1
Q_4	DAP	200MB	1.6MB	1.6MB	0.008
	QPC	200MB	200MB	1.6MB	1

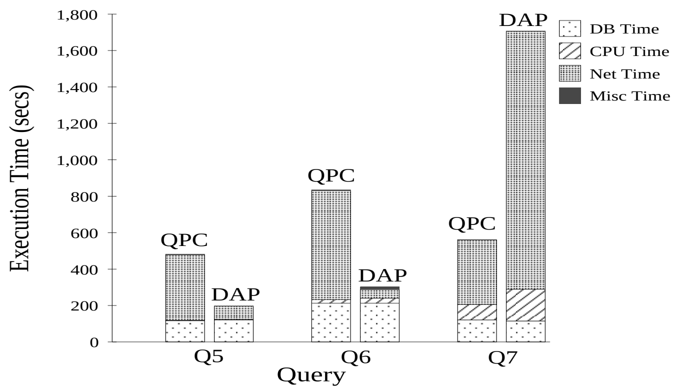
(b) Data Volumes

Figure 10: Performance for Q_1, Q_2, Q_3, Q_4

Figure 10(a) shows the execution time for each query, with the operators evaluated at each site: QPC or DAP. Clearly, evaluation at the DAP outperforms evaluation at QPC in all cases. For queries Q_2, Q_3, and Q_4, queries can be run 4 times faster at the DAP and in Q_1, there is a 50% improvement in performance. As can be observed from the figure, network cost (**Net Time**) is the dominant factor in terms of performance. In each case, the performance gain is achieved by capitalizing on code shipping to run the aggregates close to the data source and only send to the QPC a few result values. Thus, a middleware system that uses code shipping will be better equipped to deal with this type of queries than one based on the existing technologies. Figure 10(b) shows the large savings in the volume of data movement that can be obtained by using code shipping to evaluate the aggregates at the data site. Notice that in each case the best processing alternative indeed has the smallest $CVRF$ value. These results show that the VRF is an excellent metric for choosing the best operator placement in plans for queries with complex aggregates because it minimizes data movement.

5.5 Queries with Complex Projections

We used queries Q_5, Q_6 and Q_7 from Table 1, on page 19, to measure the effect of complex projections on the volume of data transmitted. Queries Q_5 and Q_6 contain data-reducing projections; in Q_5, 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. Query Q_6 maps each graph in table `Graphs` into a double precision floating point number representing the length of a drainage network. On the other hand, Q_7 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.



(a) Execution Times

Query	*Site*	*CVDA*	*CVDT*	*Volume in Result*	*CVRF*
Q_5	DAP	200MB	40MB	40MB	0.20
	QPC	200MB	200MB	40MB	1
Q_6	DAP	31MB	2.3MB	2.3MB	0.07
	QPC	31MB	31MB	2.3MB	1
Q_7	DAP	200MB	800MB	800MB	4
	QPC	200MB	200MB	800MB	1

(b) Data Volumes

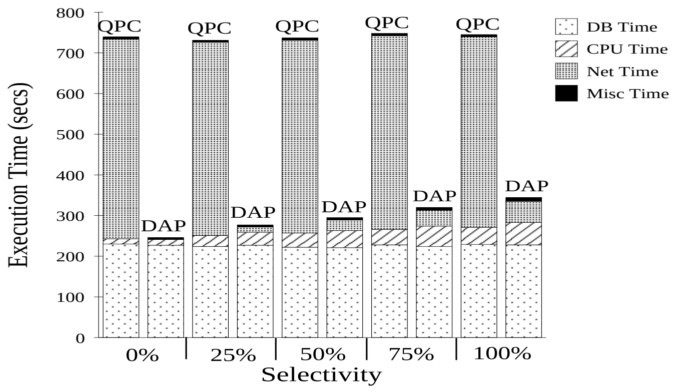
Figure 11: Performance for Q_5, Q_6 and Q_7

Figure 11(a) demonstrates that operator evaluation at the DAP is the best option for evaluating Q_5 and Q_6, as it results in a decrease in execution time from 482 secs to 192 secs for query Q_5, and, from 835 secs to 302 secs for Q_6. In both cases, the use of code shipping improves performance by a factor of 3. However, it is totally inadequate to run Q_7 at the DAP since this query contains a data-inflating operator. For this case, Q_7 runs for 1710 secs as opposed to 562 secs when QPC evaluates it. 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, therefore, 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 11(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 value for the $CVRF$. Thus, these results confirm that VRF can be used to select the best plan for queries with complex projections. Clearly, MOCHA will be well suited for processing queries with very different characteristics.

5.6 Queries with Complex Predicates

For this category, we used queries Q_8 and Q_9 from Table 1 on page 19, and in each case, we varied the selectivity of the predicates in order to study the behavior of each execution alternative under different selectivity values. In query Q_8, the qualification clause compares the number of vertices and length of each drainage network graph against two constants. The execution times for Q_8 under each selectivity value are shown in

Figure 12(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_8 to the DAP, the system avoids shipping the large-sized attribute `graph` over the network, thus providing substantial performance gains.



(a) Execution Times

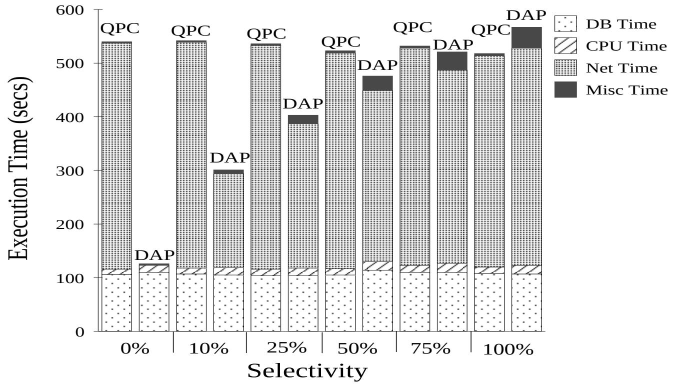
Selectivity	*Site*	*CVDA*	*CVDT*	*Volume in result*	*CVRF*
0	DAP	31MB	0B	0B	0
	QPC	31MB	31MB	0B	1
0.25	DAP	31MB	200KB	200KB	0.01
	QPC	31MB	31MB	200KB	1
0.50	DAP	31MB	400KB	400KB	0.01
	QPC	31MB	31MB	400KB	1
0.75	DAP	31MB	600KB	600KB	0.02
	QPC	31MB	31MB	600KB	1
1	DAP	31MB	790KB	790KB	0.02
	QPC	31MB	31MB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

(b) Data Volumes

Figure 12: Performance for Q_8

In Q_9, all attributes in the tuples of table `Polygons` appear in the result. For this case, the qualification clause compares the perimeter of each polygon against a constant. As shown in Figure 13(a), the performance gain is in inverse proportion to the selectivity value. Execution at the DAP outperforms its QPC counterpart in all but the last case with 100% selectivity, in which all components are essentially the same for both alternatives, but there is a one minute (almost 9%) overhead for code shipping due to initialization and cleanup. Note that the large increment in network cost from 0% to 10% selectivity is due to the nature of table `Polygons`. In this table, attribute `location`, which represents the vertices in the polygon, has variable size. Therefore, at low selectivity most of the largest polygons are the only ones satisfying the qualification clause in Q_9, which in turn are the most expensive ones to transmit. As the selectivity increases, the increment in network cost is less pronounce since only the smaller and less costly polygons are been added to the result.

Figures 12(b) and 13(b) show again that the VRF is an accurate metric for determining the best plan for a query. One important concept emerging from the performance results in this section is that a 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_8 and Q_9. Consider the case of 50% selectivity in Q_8. From Figure 12(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 the percentage of data transmitted is always much smaller than what would be expected if selectivity alone were used to make the estimate. Furthermore, selectivity might under estimate the actual amount of data transferred. This can be observed for Q_9 in Figure 13(b), where in the 10% selectivity case, execution at the DAP with code shipping, actually transmits 34% of the data volume in the `Polygons` table. As a result, a query operator placement scheme based on selectivity and result cardinality, might produce plans with poor performance for distributed queries. On the other hand, the VRF combines the selectivity information, cardinality and the size of the



(a) Execution Times

Selectivity	*Site*	*CVDA*	*CVDT*	*Volume in result*	*CVRF*
0	DAP	18.8 MB	0 MB	0 MB	0
	QPC	18.8 MB	18.8 MB	0 MB	1
0.10	DAP	18.8 MB	6.3 MB	6.3 MB	0.34
	QPC	18.8 MB	18.8 MB	6.3 MB	1
0.25	DAP	18.8 MB	11.6 MB	11.6 MB	0.62
	QPC	18.8 MB	18.8 MB	11.6 MB	1
0.50	DAP	18.8 MB	14.9 MB	14.9 MB	0.79
	QPC	18.8 MB	18.8 MB	14.9 MB	1
0.75	DAP	18.8 MB	17.2 MB	17.2 MB	0.91
	QPC	18.8 MB	18.8 MB	17.2 MB	1
1	DAP	18.8 MB	18.8 MB	18.8 MB	1
	QPC	18.8 MB	18.8 MB	18.8 MB	1

(b) Data Volumes

Figure 13: Performance for Q_9

attributes in the tuples been transmitted in order to estimate the cost of a query operator and determine its proper placement. Therefore, for distributed processing, VRF is a much better metric for the cost of a plan.

5.7 Queries with Distributed Joins

For this category, we used query Q_{10} from Table 1 on page 19. This query performs a distributed join between two tables, `Rasters1` and `Rasters2`, containing raster images. The schema for these tables is the same as the one for table `Raster` (see section 5.2), but the images where reduced to 128KB in size, and only three locations are common to both tables, `Rasters1` and `Rasters2`. 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_{10} 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.

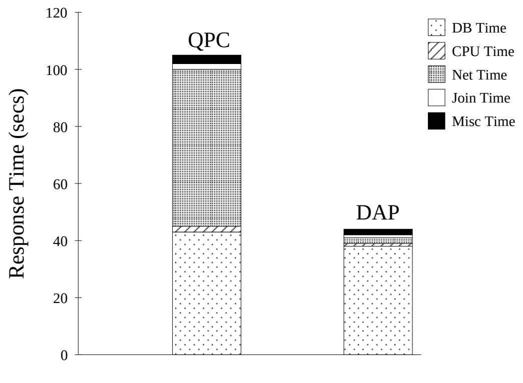


Figure 14: Execution Time for Q_{10}

Figure 14 shows the execution time for Q_{10} 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 two relations have to be moved to QPC. As the tuples arrive, the function `AvgEnergy()` is evaluated to perform the projections and then the tuples are stored to disk, from which they are later read to perform the join operation, which is done using a hash join method. In this figure, the **Join Time** component indicates the cost of accessing disk to manipulate the partial results been generated during join processing. 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 some of the operators. In this case, a 2-way semi-join is performed by computing, at each DAP, the semi-joins `Rasters1`$\ltimes$`Rasters2` (at Site1) and $\texttt{Rasters2} \ltimes \texttt{Rasters1}$ (at Site2), using the *complex predicate* in Q_{10} for both of them. After each semi-join operation is performed, function $\texttt{AvgEnergy}()$ is evaluated and all projections are taken. As a result, only attributes `time`, `location` and $\texttt{AvgEnergy}(\texttt{data})$ are moved from a DAP to QPC, where they are first materialized to disk and then joined. By doing this 2-way semi-join operation and evaluating the complex projections at the DAP, the network cost is minimized and the overall execution time 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

In this paper we have proposed MOCHA as an alternative middleware solution to the problem of integrating data sources distributed over a network. We have argued that the scheme used in existing solutions, where user-defined 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 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 the 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 queries used in distributed systems. The MOCHA prototype has been implemented with JDK 1.2 and extensively tested top of the Informix Universal Server. We have carried out a performance study on MOCHA using data and queries from the Sequoia 2000 Benchmark, and show 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, 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 Conference*, Tokyo, Japan, 1994.

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

[Ora99] Oracle Corporation. Oracle Transparent Gateways, 1999. URL: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 Conference*, pages 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 Conference*, pages 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 Conference*, pages 390–401, Seattle, Washington, USA, 1998.

[Gra93] G. 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 Conference*, pages 276–285, Athens, Greece, 1997.

[HS93] J.M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. In *Proc. ACM SIGMOD Conference*, pages 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 Conference*, Kyoto, 1986.

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

[RMR98a] M. Rodríguez-Martínez and N. Roussopoulos. An Architecture for a Mobile and Dynamically Extensible Distributed DBMS. Technical Report ISR-TR 98-10, CSHCN-TR 98-2, University of Maryland, January 1998.

[RMR98b] M. Rodríguez-Martínez and N. Roussopoulos. MOCHA: A Self-Extensible Middleware Substrate For Distributed Data Sources. Technical Report UMIACS-TR 98-67, CS-TR 3955, University of Maryland, October 1998.

[RMR00] M. Rodríguez-Martínez and N. Roussopoulos. Automatic Deployment of Application-Specific Metadata and Code in MOCHA. In *Proc. 7th EDBT Conference*, Konstanz, Germany, 2000.

[RS97] M.T. Roth and P. 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.

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

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

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