Automatic Deployment of Application-Specific Metadata and Code in MOCHA

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

Abstract

Database middleware systems require the deployment of application-specific data types and query operators to the servers and clients of a distributed database system. Existing middleware solutions rely on developers and system administrators to port and manually install all this application-specific functionality to all sites in the system. This approach cannot scale to an environment in which there are hundreds of data sources, such as those accessed by the Web and even more custom-tailored applications, since the complexity and the cost involved in maintaining a code base system-wide are enormous. This paper describes a novel metadata-driven framework designed to automate the deployment of all application-specific functionality used by a middleware system. We used Java and XML to implement this framework in MOCHA, a middleware system that was developed at the University of Maryland. We first present the kind of services, metadata elements and software tools used in MOCHA to automate code deployment. Then, we describe how the features of MOCHA simplify the administration and reduce the management cost of a middleware system in a large scale environment.

1 Introduction

Database middleware systems, such as database gateways and mediator systems, are used to integrate heterogeneous data sources dispersed over a computer network. In order to achieve data integration, the middleware layer imposes a global data schema on top of the individual schema used by each source. Through this mechanism, the client applications been serviced by the middleware system are provided with a uniform view and uniform access interface to the data sets stored by each data source. The translation of the data items to the global schema is performed by either a wrapper or database gateway. Wrappers are used when integration is achieved through a mediator system, such as TSIMMIS [CGMH+94], DISCO [TRV96] or Garlic [RS97]. On the other hand, gateways are used when integration is realized by importing the data into a commercial DBMS, such as Oracle [Cor99] or Informix [Cor97]. Typically, these applications use a connectivity API such as ODBC or JDBC to extract the data from the sources. The wrapper or gateway can either be run on a machine near the data source (e.g. on the same Local Area Network) or at the site where the integration server runs.

A problem with the use of middleware systems is the deployment of the application-specific data types and operators necessary to implement the global schema used by the system. Since new applications and data

[†]Contact author. Phone: (301)-405-2714, Fax: (301)-405-6707

sources are added to the system as time progresses, the global schema must be changed to reflect these new additions. And since the data in each source must be translated from its native format into the middleware-level format specified by the global schema, new data types must be custom-built to represent these data in the newly introduced format. Notice that these data types will be used by the middleware, to hold the values been processed, and by the client application, to present the result values to the user. Moreover, all query operators that cannot be evaluated by the data sources will have to be implemented at and evaluated by the middleware system. Therefore, the scalability of the middleware system depends on how efficiently it can ingest and deploy all this new application-specific functionality to the clients and servers which are part of the system.

In our view, existing middleware solutions fail to provide adequate mechanisms to deploy new or updated functionality to the existing middleware infrastructure. Most systems use either C or C++ as the implementation language for middleware-level data types and operators. With this approach, the functionality has to be ported to several different hardware and operating system platforms, which can be a very slow and expensive process. In addition, the new code has to be **manually** installed into every machine in which a client, mediator, wrapper or gateway application can be expected to be run. Clearly, as the system grows with new applications, data sources and users, it becomes increasingly difficult and expensive to maintain the software base used throughout the system.

The objective of this paper is to present a novel metadata-driven framework used to automate the deployment of application-specific functionality in a middleware system. We have implemented this framework in MOCHA[1], a prototype database middleware substrate developed at the University of Maryland. MOCHA is based on the philosophy that all application-specific code should be automatically deployed by the middleware system itself. In MOCHA, this is realized by implementing the new functionality in Java classes, which are then shipped to the client applications and to remote servers from which data will be extracted. This feature of *automatic code deployment*, frees the administrators from having to perform system-wide installations of software. Instead, all the Java classes are stored into one or more *code repositories* from which MOCHA later retrieves and deploys them on a "need-to-do" basis.

MOCHA not only simplifies the administration effort needed to maintain the software for integrating data sources, but also provides efficient query services. In [RMR99] we showed that MOCHA leverages its ability to ship Java classes implementing query operators to execute them near the data source or near the client in an effort to reduce data movement over the network. Data filtering operators which produce smaller values are computed near the data source, while data inflating operators which expand their arguments are evaluated near the client. Using this query optimization framework, MOCHA provides substantial performance gains on both single-site and multi-site queries containing complex aggregates, predicates and projections.

In this paper, we describe the components in the architecture of MOCHA, and the main services, metadata elements and software tools necessary to support automatic code deployment. Since metadata and control must be exchanged between the components of MOCHA, we also present the exchange formats used for this purpose. These formats are based on the well-accepted XML standard for content exchange between networked applications. The remaining of this paper is organized as follows. Section 2 presents a brief description of XML and RDF, Internet standards used for metadata and control exchange in MOCHA. The

[1] MOCHA stands for **M**iddleware Based **O**n a **C**ode **SH**ipping **A**rchitecture.

architecture of MOCHA is described in section 3. In section 4 we discuss the metadata necessary to support automatic code deployment. Section 5 presents the entire code deployment and query processing cycle used by MOCHA. Implementation status and benefits of our approach are presented in section 6. Related work is briefly described in section 7. Finally, our conclusions are given in section 8.

2 Overview of XML and RDF

In this section we briefly review XML and RDF, two technologies we use to build the framework for automatic deployment of application-specific code.

2.1 XML

The Extensible Markup Language (XML) [Con98], is a W3C [2] standard for data exchange over the Internet. XML is a markup language derived from SGML, but with a much simpler structure. XML is designed to *encode* the content in a document, and make it "machine-readable". In this regard, XML is very different from HTML, which is designed to *present* the content in a document on a Web browser. Figure 1 depicts XML data encoding a personalized phone book. As we can observe from the figure, XML data is organized as a series of *elements* delimited by tags. In this example the tags are `PhoneBook`, `Address`, `Name` and `Phone`. Each XML element either encloses another XML element or a datum encoded as a string. Thus, in XML the schema information and the data are all integrated in the same document. This arrangement is what makes XML documents machine-readable, or *self-describing*, since applications can parse the XML document and find the tags enclosing the data they need to process.

```
<PhoneBook>
  <Address>
    <Name>John Smith</Name>
    <Phone>(301)-403-0500</Phone>
  </Address>
  <Address>
    <Name>Adams Morgan</Name>
    <Phone>(999)-201-8931</Phone>
  </Address>
</PhoneBook>
```

Figure 1: An XML Phone Data.

XML is a fully extensible language, and the ability of programmers to add new tags to XML is one of its most important assets. XML can be customized with new tags that express the data schema for many applications and provide a mechanism for data exchange specific to these applications. The structure of an XML document can be validated by the applications by using a *Document Type Descriptor* (DTD). These are grammars which describe the valid structure of a particular XML document. All the above features in XML have caught the attention of major software vendors, which are now targeting XML as the standard for data interchange used by their products.

2.2 RDF

The Resource Description Framework (RDF) [Con99b] is an extension of XML designed to provide metadata interoperability between applications. RDF provides a standard mechanism to encode and exchange metadata about any entity of interest to any given application. Each object been described is termed a *resource* and is uniquely identified by a *Uniform Resource Identifier* (URI) [For98].

[2] W3C stands for World Wide Web Consortium, which is the body that directs the efforts to standardize Web-related technologies.

```
<?xml version=``1.0'' ?>
<rdf xmlns = ``http://w3.org/TR/1999/PR-rdf-syntax-199901105#''
      xmlns:DC = ``http://purl.org/DC#''>
      <Description about = ``http://www.umd.edu/report.html'' >
         <DC:Title>Annual Report</DC:Title>
         <DC:Creator>John Mote</DC:Creator>
         <DC:Date>05-01-99</DC:Date>
         <DC:Subject>UMCP, University, Government</DC:Subject>
      </Description>
</rdf>
```

Figure 2: An RDF example

RDF metadata is organized as a set of properties types and values encoded in XML, as shown in Figure 2. In this example, a report with URI `http : //www.umd.edu/report.html` is been described. The `rdf` and `Description` tags are introduced by RDF to identify the XML elements that contain metadata. The attributes `xmlns` and `xmlns : DC` are used to identify the *namespaces* for the tags used in the document. XML supports a *namespace* feature [Con99a] which is used to give a specific context to the tags contained in XML documents. Each namespace used in an XML document is uniquely identified by a URI. In Figure 2, `xmlns` gives the namespace for the RDF tags (`rdf` and `Description`), and `xmlns : DC` gives the namespace for the Dublin Core tags, which are those that begin with the `DC` : prefix. The Dublin Core is a standard set of metadata identifiers used to describe electronic documents, such as those stored in digital libraries. The metadata shown in Figure 2 indicates the title, author, creation date and general description of the annual report on the status of the University of Maryland. Clearly, RDF-encoded metadata can be readily used by an application to discover the information necessary to find documents of interest to the user, and such documents might reside on the Web, a database server or in the file system of a particular workstation.

3 MOCHA Architecture

In this section, we describe the principal components in the architecture of MOCHA. We have implemented a prototype for MOCHA using the Java programming language, and we have built the system around two fundamental principles. First, all the code which implements data types and query operators is automatically and seamlessly deployed by MOCHA to the clients and servers in the system. Second, all query operators that are evaluated by the middleware layer are scheduled for execution at the site that results in minimum data movement over the network.

Figure 3, on page 5, depicts the components in the architecture of MOCHA. At the top of the architecture is the **Client Application**, which provides the user with the Graphical User Interface (GUI) to pose queries to the system and visualize the result. In most cases, we expect the client to be an applet loaded into a Web browser, but it is also possible to use a Java stand-alone application. The client connects to the **Query Processing Coordinator** (QPC) and sends to it all queries posed by the user. The QPC is a server application which provides the basic query processing services in the system, and also takes care of deploying all application-specific code necessary to process a query. The client connects to the QPC by means of a *Uniform Resource Locator* (URL). The main services provided by QPC are: a) query parsing, b) metadata management, c) query optimization, d) code deployment, e) query execution, and f) error management.

In order to access the wealth of information stored in a particular data source, the QPC connects to the **Data Access Provider** (DAP) associated with the source. The DAP is a server application which extracts data from a source on behalf of the QPC. For each data source, there is at least one DAP, and each DAP in the system can be located by QPC through a URL. There are two essential services provided by a DAP: a) data translation, and b) query execution. The DAP extracts requested items from the data source, and translates them from the local schema used by the source into the global schema used by QPC. Also, the DAP is capable of executing query operators that generate new abstractions from the data. In particular, the DAP is designed to execute those operators that filter out the data sets (e.g. a predicate) to produce smaller values. For this reason, the DAP should be run at the data source site or in close proximity to it (e.g. on another host in the same LAN). The QPC delivers all the code for the data types and operators used by each DAP. Similarly, all results produced by each DAP are sent to QPC for further processing until the final answer to the query is fabricated.

The final component in the MOCHA architecture is the **Data Server**, which is the server application that provides storage for the data sets stored and manipulated by each data source. Each DAP in the system must be configured to run on top of a particular Data Server. MOCHA can support a wide variety of data servers, including database servers, XML repositories, Web servers and file servers. Clearly, the architecture of MOCHA provides the foundation for a very flexible, scalable and well-organized middleware solution to integrate a wide range of data sources.

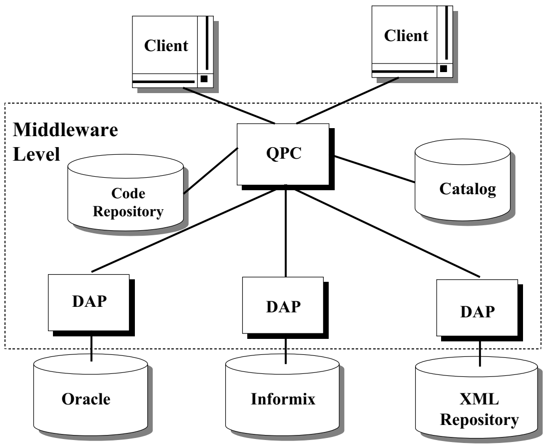

Figure 3: MOCHA Architecture

4 Publishing Resources

In this section we use an example application to describe the capabilities incorporated in MOCHA to publish resources such as tables, query operators and data types. For simplicity, we assume that the system follows the relational model. The capabilities for publishing resources are built on top of RDF and therefore, each resource is identified by a URI. The exact structure of such URI must be chosen by the system administrator, and should follow the conventions specified in [For98]. In the examples presented in this paper we will use two simple conventions. First, the URI for a relation will be of the form:

$$\texttt{mocha}:<\texttt{host}>/<\texttt{database}>/<\texttt{table}>.$$

The keyword `mocha` is used as a reminder that the resource been published will be used by MOCHA. The `host` component specifies the domain name or IP address of the machine hosting the data source. Similarly, the `database` part gives the name of the targeted database space, and `table` is the name of the table been published. The second convention is for data types and operators. For these resources, their URI is of the form :

$$\texttt{mocha}:<\texttt{host}>/<\texttt{repository}>/<\texttt{object}>.$$

In this instance, `host` is the domain name or IP address of the machine hosting the code repository containing the Java class for a type or operator. The `repository` component indicates what code repository must be accessed to find the Java class associated with the resource. Finally, the `object` part gives the user-specified name of the resource being published.

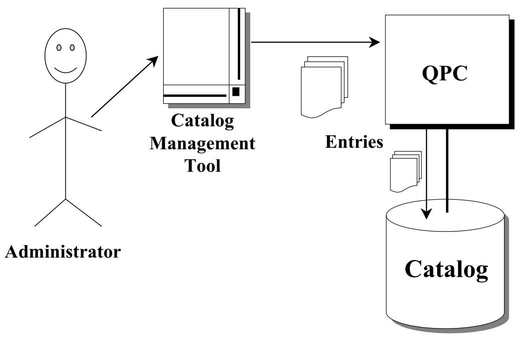

Figure 4: Catalog Management

MOCHA uses the URI for a resource as search key into the catalog to find the metadata for the resource. The metadata is contained in a RDF text document, with a schema specific to MOCHA. In this schema, all tags contain the prefix `mocha :` , which identifies the MOCHA namespace[3]. For each resource, the administrator uses an utility application program to add the metadata entry, of the form $(URI, RDF\ File)$, into the catalog table specific to the type of resource. Each entry is sent to the QPC and then added to the catalog, as illustrated by Figure 4.

4.1 Motivating Application

Consider an Earth Science application used to manipulate 2D satellite images and surface maps. The data sets needed by this application are maintained in two separate data sources. The first data source is an Oracle database server, containing a relation named `Maps`. This database server runs on a host located in the Geography Department at the University of Maryland. Relation `Maps` stores maps from different locations in the State of Maryland, and has the following schema:

$$\texttt{Maps}(\texttt{name} : \texttt{char}(20), \texttt{location} : \texttt{Rectangle}, \texttt{map} : \texttt{blob});$$

Attribute `name` is the name of a region, `location` is the bounding box for that region and `map` is the surface map for the region. This table is available for access by all users.

The second data source is an Informix database server, which contains a relation named `Rasters`. This server is hosted by a workstation in the Computer Science Department at the University of Maryland. Table `Rasters` contains satellite AVHRR images containing weekly energy readings from the surface of the State of Maryland. There is one year worth of observations stored in relation `Rasters`, and the schema for this relation is a follows:

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

In this case, attribute `week` gives the week number in which the image was made, `band` represents the energy band measured, `location` gives the bounding box for the region under study and `image` is the AVHRR image itself. This table is also available to all users.

Our example application mainly performs two tasks. First, it computes the composite of all AVHRR images for a given location within a specific time frame. The SQL query to accomplish this task is:

[3]The URI for this namespace is `http : //www.cs.umd.edu/users/manuel/MOCHA/`.

```
SELECT    location, Composite(image, band)
FROM      Rasters
WHERE     week BETWEEN t1 AND t2
GROUP BY  location
```

We will identify this query throughout the rest of this paper as $Q1$. The function `Composite()` used in $Q1$ is a user-defined aggregate, which generates an image that is the composite of a set of AVHRR images.

The second task performed by our application is to overlay AVHRR images on top of maps. Specifically, all those images containing an energy reading larger than an user-specified value `X`, are overlay on top of the map for the region to which the image belongs. In SQL, this task is specified as follows:

```
SELECT  M.name, R.week, R.location, Overlay(R.image, M.map)
FROM    Rasters R, Maps M
WHERE   Equal(R.location, M.location)
AND     Energy(R.image) > X
```

This second query will be identified as $Q2$, and it computes a join between relations `Rasters` and `Maps`. Tuples are joined based on whether they have a common `location` attribute, which is determined by function `Equal()`. The average amount of energy in an image is computed by function `Energy()` and this value is represented as a double precision floating point number. Finally, function `Overlay()` creates a new image by overlaying an AVHRR image on top of a map. All three functions used in $Q2$ are user-defined. Given this scenario, we now discuss how to configure MOCHA to provide support for our Earth Science application.

4.2 Tables

The first resources that must be made available to MOCHA are the tables to be used by the application. For each table, metadata indicating its name, the database in which it is stored, the columns names and the middleware types needed to represent each column must be added to the catalog. This information will enable MOCHA to access each table, retrieve its tuples, project one or more of its columns and translate each column value into a middleware data type.

Figure 5 shows the RDF metadata for table `Rasters`. The URI for this table is specified by the `about` attribute in the RDF `Description` tag. Property `mocha : Table` gives the name of the relation, and property `mocha : Owner` gives the e-mail address of its owner. Connectivity information is provided by property `mocha : Database`. This element specifies the URL of the DAP associated with the data source (i.e. the Informix Server) and the name of the database space in which relation `Rasters` is stored. In this case, the DAP is located at URL `cs1.umd.edu : 8000`, and table `Rasters` is contained in the `EarthSciDB` space. Each of the columns in `Rasters` is described in the `mocha : Columns` property, which contains a sequence of column descriptions. Each description is delimited by the `li` tag, and for each column, property `mocha : Column` indicates the column name, `mocha : Type` gives the name of the middleware type used to represent its values and the URI for this data type is specified by the `mocha : URI` property. Once this information is added to the catalog, table `Rasters` is ready to be used in queries posed to the QPC.

```
<Description about =
  ''mocha://cs1.umd.edu/EarthSciDB/Rasters'' >
  <mocha:Table> Rasters </mocha:Table>
  <mocha:Owner> manuel@cs1.umd.edu </mocha:Owner>
  <mocha:Database> cs1.umd.edu:8000/EarthSciDB
  </mocha:Database>
  <mocha:Columns>
    <Seq>
      <li parseType = ''resource'' >
        <mocha:Column> week </mocha:Column>
        <mocha:Type> MWInteger </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/BaseTypes/MWInteger
        </mocha:URI>
      </li>
      <li parseType = ''resource'' >
        <mocha:Column> band </mocha:Column>
        <mocha:Type> MWInteger </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/BaseTypes/MWInteger
        </mocha:URI>
      </li>
      <li parseType = ''resource'' >
        <mocha:Column> location </mocha:Column>
        <mocha:Type>Rectangle</mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/EarthScience/Rectangle
        </mocha:URI>
      </li>
      <li parseType = ''resource'' >
        <mocha:Column>image</mocha:Column>
        <mocha:Type>Raster</mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/EarthScience/Raster
        </mocha:URI>
      </li>
    </Seq>
  </mocha:Columns>
</Description>
```

Figure 5: Metadata for table `Rasters`

4.3 User-Defined Operators

As mentioned in section 3, query operators can be executed by the QPC or the DAP, and each of these two components contains an extensible query execution engine with an iterator-based machinery for data processing. Since each operator is dynamically imported into the execution engine, the metadata must provide enough information to instantiate the operator. In particular, the kind of operator, the number and type of arguments, and the expected result type must be thoroughly described for the execution engine module. In MOCHA, query operators are divided into two categories: complex functions and aggregates. We discuss the metadata structure of these two types separately.

4.3.1 Complex Functions

Complex functions are used in complex predicates and projections contained in queries. A complex function is implemented in a static method defined in a Java class[4]. As depicted in Figure 6, the execution engine uses the name of the static method and the class defining this method to create a *Function Evaluation Object*, which takes care of executing the body of the method. This Function Evaluation Object is based on the *Java Reflection Mechanism*, which is similar to the *function pointer* abstraction used in C. As tuples are read from the data source, the columns used as arguments to the function are extracted and passed to the Function Evaluation Object. Then, the body of the function is executed and the result is further processed or added to the final result.

Figure 7 shows the metadata for function `Equal()`, which is used in query $Q2$ of our example application (see section 4.1). Property `mocha : Function` gives the name of the function and also identifies the metadata block as one for a complex function. Function `Equal()` is defined in class `Geometry.class` and implemented by the static method `Equal`, as indicated by the `mocha : Class` and `mocha : Method` properties, respectively.

[4] Static methods are those methods whose body can be executed without first creating an object instance from the class in which the method is defined.

```
<Description about =
  ''mocha://cs1.umd.edu/EarthScience/Equal'' >
  <mocha:Function> Equal </mocha:Function>
  <mocha:Class> Geometry.class </mocha:Class>
  <mocha:Method> Equal </mocha:Method>
  <mocha:Repository> cs1.umd.edu/EarthScience
  </mocha:Repository>
  <mocha:Arguments>
    <Seq>
      <li parseType = ''resource'' >
        <mocha:Type> Rectangle </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/EarthScience/Rectangle
        </mocha:URI>
      </li>
      <li parseType = ''resource'' >
        <mocha:Type> Rectangle </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/EarthScience/Rectangle
        </mocha:URI>
      </li>
    </Seq>
  </mocha:Arguments>
  <mocha:Result>
    <mocha:Type> MWBoolean </mocha:Type>
    <mocha:URI>
    mocha:cs1.umd.edu/BaseTypes/MWBoolean
    </mocha:URI>
  </mocha:Result>
  <mocha:Creator> manuel@cs1.umd.edu </mocha:Creator>
</Description>
```

Figure 7: Metadata for function `Equal`()

Property `mocha : Repository` contains the URL for the code repository containing class `Geometry.class`. This repository is named `EarthScience` and resides on host `cs1.umd.edu`.

The arguments to function `Equal`() are the two rectangles to be tested for equality. The metadata for these arguments are contained in a `mocha : Arguments` property. Like in the case for the columns in a table, the arguments are specified using the `Seq` construct. For each argument, the name of its type is given in property `mocha : Type`, and `mocha : URI` gives the corresponding URI for this type. In similar fashion, property `mocha : Result` is used to describe the return type of the function. In this case, the result is a boolean value, whose type name and type URI are described by properties `mocha : Type` and `mocha : URI`, respectively. Finally, the person who implemented this function is identified with his/her e-mail address in `mocha : Creator`.

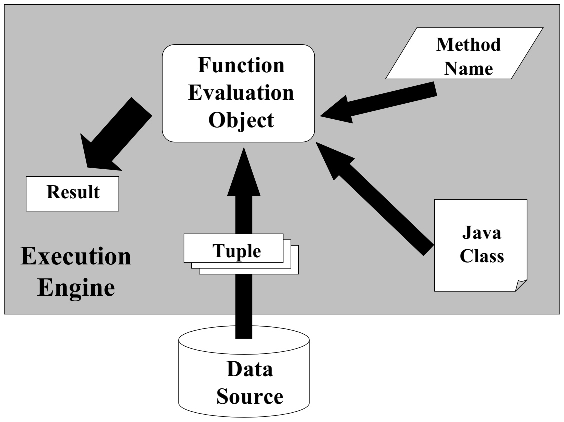

Figure 6: Complex Function Organization

4.3.2 Aggregates

In MOCHA, an aggregate operator is implemented as an instance of a Java class, as shown in Figure 8. Such class must implement the `Aggregate` standard interface provided by MOCHA. This interface defines three methods which are used by the execution engine to evaluate the aggregate operator: `Reset`(), `Update`() and `Summarize`(). The execution engine will create an aggregate object for each of the different groups formed during the aggregation process, and each object is first initialized through a call to method `Reset`(). As tuples are read form the source, method `Update`() is repeatedly called to update the internal state in the aggregate. This update is done based on the existing internal state in the aggregate object and the argument attributes

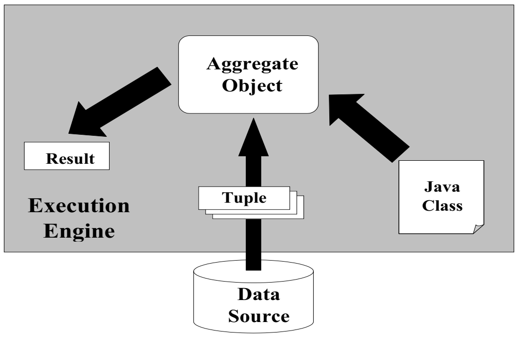

Figure 8: Aggregate Organization

from the next tuple read. Once all tuples have been ingested, the result in the aggregate object is obtained by calling method `Summarize()`.

```
<Description about =
  ''mocha://cs1.umd.edu/EarthScience/Composite'' >
  <mocha:Aggregate> Composite </mocha:Aggregate>
  <mocha:Class> Composite.class </mocha:Class>
  <mocha:Repository> cs1.umd.edu/EarthScience
  </mocha:Repository>
  <mocha:Arguments>
    <Seq>
      <li parseType = ''resource'' >
        <mocha:Type> Raster </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/EarthScience/Raster
        </mocha:URI>
      </li>
      <li parseType = ''resource'' >
        <mocha:Type> MWInteger </mocha:Type>
        <mocha:URI>
        mocha:cs1.umd.edu/BaseTypes/MWInteger
        </mocha:URI>
      </li>
    </Seq>
  </mocha:Arguments>
  <mocha:Result>
    <mocha:Type> Raster </mocha:Type>
    <mocha:URI>
    mocha:cs1.umd.edu/EarthScience/Raster
    </mocha:URI>
  </mocha:Result>
  <mocha:Creator> manuel@cs1.umd.edu </mocha:Creator>
</Description>
```

Figure 9: Metadata for aggregate `Composite()`

Figures 9 shows the metadata for aggregate `Composite()` used in $Q1$. The structure of the metadata is essentially the same as that for the complex functions, with only two minor differences. First, the name of the aggregate is given by property `mocha : Aggregate`. Secondly, property `mocha : Method` is not needed, since the aggregate will be manipulated through the three well-known methods defined in the `Aggregate` interface. As we can see from the figure, the aggregate is defined in class `Composite.class`, which is stored in repository `EarthScience`. The aggregate receives two arguments, an AVHRR image and the energy band measured in the image, and it returns an AVHRR image, which is the composite of all the images processed.

4.4 User-Defined Data Types

From the previous sections, we have seen that most resources depend heavily on data types. In MOCHA, data types are implemented in Java classes, and the type system is organized in a hierarchy shown in Figure

10. The root element is the MWObject interface which identifies a Java class as implementing a data type. In addition, this interface defines the methods necessary to transmit object instances across the network.

Two interfaces are directly derived from MWObject, namely MWSmallObject and MWLargeObject. Interface MWSmallObject must be implemented by classes used for small-sized types such as strings, numbers, points, rectangles, etc. This interface defines methods to read the values from the data source, convert the content of a type to a Java String and perform tests for equality. The semantics for character-based types are embedded in the MWString interface, and in similar fashion, interface MWNumber contains those for numeric types. On the other side of the spectrum, large objects such as images, videos and text documents are supported through the MWLargeObject interface. This interface provides an abstraction based on files to support large objects read from the data source and manipulated by the components in MOCHA. Every class used to implement a data type must either implement MWSmallObject, MWLargeObject or an interface derived from one of these.

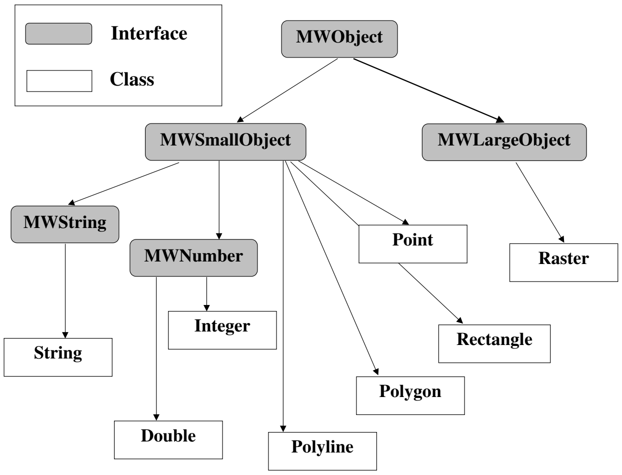

Figure 10: Organization of Data Types

Our example application handles AVHRR images, and Figure 11 presents the metadata for the data type Raster used to represent them. The RDF property mocha : Type indicates that Raster is the name of the type for the images. This type is defined in class Raster.class, as indicated by property mocha : Class. Like in our previous examples, the code repository, in this case EarthScience, is given by the property mocha : Repository and the developer by property mocha : Creator. Since QPC needs to optimize the queries posed by the user, the size (or at least an approximation) of the attributes accessed by the query must be available to the optimizer to estimate the cost of transferring such attributes over the network. This is provided with property mocha : Size, which indicates that the AVHRR images are 1MB in size.

```
<Description about =
  ``mocha://cs1.umd.edu/EarthScience/Raster'' >
  <mocha:Type> Raster </mocha:Type>
  <mocha:Class> Raster.class </mocha:Class>
  <mocha:Repository> cs1.umd.edu/EarthScience
  </mocha:Repository>
  <mocha:Size> 1MB </mocha:Size>
  <mocha:Creator> manuel@cs1.umd.edu </mocha:Creator>
</Description>
```

Figure 11: Metadata for data type Raster

5 Query Execution Architecture

In this section we describe the query execution architecture used in MOCHA. Query execution is a process divided in six phases: 1) Query Request, 2) Resource Discovery, 3) Query Validation, 4) Query Optimization, 5) Code Deployment and 6) Data Processing. We describe each of these phases during the execution of $Q2$.

5.1 Query Request

In our example, the client application is a Java applet that is loaded from a Web server into a Java-enabled Web browser. The client application first opens a connection, using a URL, to the QPC for which it has been configured to communicate with. In this case, the QPC is hosted by the Computer Science Department at the University of Maryland. Once the connection with the QPC has been established, the client sends a query request to the QPC. This request consists of the SQL string for query $Q2$. The client then waits for the QPC to signal that the query request has been successful, and if so, the client waits for the results to arrive. If an error condition is signaled by QPC, the client presents the error message to the user. All these tasks are performed through the Java client APIs provided by MOCHA.

5.2 Resource Discovery

Upon receiving a query request from the client, the QPC must determine from the query what tables must be accessed, what software must be deployed, and what kind of computational resources (e.g. memory space) must be allocated to process the query. These tasks are carried out by searching **metadata** in the catalogs under the control of the QPC.

For $Q2$, QPC first parses its SQL string and extracts the following information: 1) the name of the tables, namely `Rasters` and `Maps`; 2) the attributes from the records in each table to be manipulated, in this case:

- `time`, `location` and `raster` for table `Rasters`
- `name`, `location` and `map` for table `Maps`;

and 3) the user-defined functions used in the query, which are `Equal()`, `Energy()` and `Overlay()`.

The next task for QPC is to access the catalogs to find the metadata for these resources. In turn, this task is divided in two steps. In the first step, QPC accesses an aliases table that contains the mapping between "common" names for the tables or operators and their corresponding URI. For example, for table `Rasters`, there is an entry of the form:

(`Rasters`, `mocha : cs1.umd.edu/EarthSciDB/Rasters`).

In the second step, QPC formulates a query to the catalog to extract the metadata for each resource, using the resource's URI as the search key. QPC searches for metadata in the following order: 1) Tables, 2) Operators and 3) Data Types. In $Q2$, QPC first finds the metadata for tables `Rasters` and `Maps`. These metadata are immediately processed to extract all pertinent information about the tables (i.e. the URL for the DAP associated with each data source). In addition, QPC extracts from the metadata the URIs of the data types used to represent the columns in each table, and stores these URIs in a list.

After the metadata for the tables have been ingested, the QPC searches the metadata for functions `Equal()`, `Energy()` and `Overlay()`. Like before, QPC processes the metadata for these functions and extracts the URIs for all the data types used in the functions and stores them in the list previously created. Finally, the QPC iterates over the list of URIs for data types and finds the metadata for each data type necessary to process query $Q2$. All the metadata found by this process is kept in memory for use during the remaining four phases of query execution. Clearly, after this phase the QPC has gathered enough information to fully interpret the structure and expected behavior of the resources that it is about to utilize.

5.3 Query Validation

After gathering the metadata necessary to interpret query $Q2$, QPC validates the query to guarantee its correctness. First, QPC checks whether tables `Rasters` and `Maps` exist and if they can be accessed by the user who posed query $Q2$. Next, QPC finds if attributes `R.time`, `R.location`, `R.raster`, `M.name`, `M.location` and `M.map` appear in the records stored in `Rasters` and `Maps`. Finally, QPC determines if the arguments to functions `Energy()`, `Equal()` and `Overlay()` are of the correct types. If no error is discovered during the validation process, the query request is accepted for evaluation and the QPC signals the client application. If QPC finds any error during the validation process, it creates a message explaining the causes of the error. This message is sent to the client application to inform the user about the error condition and have her/him take a corrective action.

Coming back to our example, we can see that the validation of query $Q2$ yields no error. Tables `Rasters` and `Maps` exist and are accessible by all users. All attributes used in the query are all valid since they are defined as part of the schema for tables `Rasters` and `Maps`. Function `Equal()` receives as arguments the `location` attributes in each of the records from the tables to be joined, and these are the correct arguments for this function. Similarly, function `Energy()` receives as argument an AVHRR image, which is the expected argument type. Finally, `Overlay()` receives its expected arguments, an AVHRR image and a map. QPC is now ready to move to the next step: finding the best plan to process $Q2$.

5.4 Query Optimization

In MOCHA, query optimization is based on the principle that code is less bulky and far more efficient to ship than data. Therefore, MOCHA capitalizes on the migration of code and the "plug & play" feature of the Java platform. This is a novel and unique approach in query optimization, since code deployment is incorporated with other well-known techniques for finding the plan which **minimizes** the execution time needed to process a query, and the amount of data transferred between QPC and the DAPs. In MOCHA, operators are shipped to and executed at the site which results in minimum data movement. Operators that reduce or filter the data sets to produce a smaller abstraction, called *data-reducing*, are computed by the DAPs associated with the data sources. For example, predicate `Energy(R.raster)` $>$ `X` in $Q2$ is data-reducing since it removes unnecessary tuples from table `Rasters`. Similarly, aggregate `Composite(raster, band)` in $Q1$ (see section 4.1) is data-reducing since it maps a sets of AVHRR images into just one image.

On the other hand, operators that increase the size of the data sets are called *data-inflating* and are evaluated by the QPC. For example, suppose that an additional projection operator named `IncrRes(R.raster, 2X)` is added to $Q2$, and this operator increases the resolution of each image by a factor 2X. This new projection is data-inflating since it generates a new AVHRR image with twice the resolution and four times the size of the original. The details of query optimization in MOCHA can be found in [RMR99].

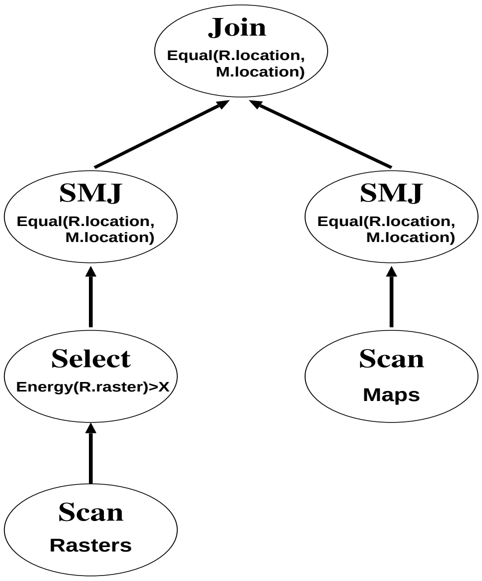

Figure 12: Plan for $Q2$.

Figure 12 shows the plan to be used by MOCHA for $Q2$. In MOCHA, the join node is evaluated by the integration server, namely QPC, and the scan nodes by the data sources. The inputs for the join operator, are two semi-join (SMJ) operators. The left input is the semi-join `Rasters`⋉`Maps`, which is computed by the DAP for the Informix Server, and the right input is the semi-join `Maps`⋉`Rasters`, which is computed by the DAP for the Oracle Server. Notice that before computing the semi-join `Rasters`⋉`Maps`, the DAP filters relation `Rasters` with predicate `Energy(R.raster)` $>$ `X`. Once the join between relations `Rasters` and `Maps` is completed, function `Overlay()` is evaluated and all the projections are taken.

5.5 Code Deployment

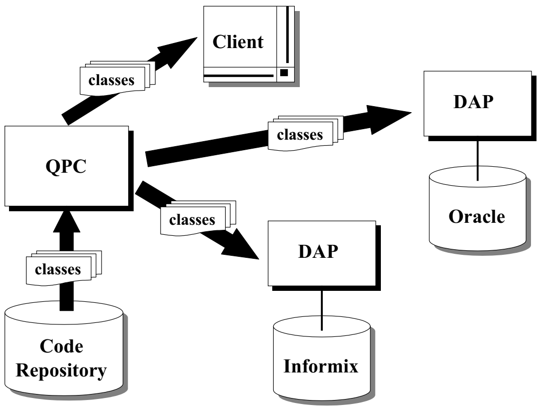

Figure 13: Automatic Code Deployment

Once the QPC has determined the plan P, to solve the query at hand, in this case $Q2$, its next task is the automatic deployment of all the classes that implement each of the data types and operators used in the query. In MOCHA, this process is called the *code deployment phase*. As illustrated in Figure 13, QPC retrieves each class for its code repository and ships it to the other components that require it. The client application and the DAPs will receive only those classes that each requires, as specified in the operator schedule contained in the query plan P.

procedure MOCHA_DeployCode:
/* deploys classes for resources R */

1) **for each** $i \in R$ **do**
2) $\quad entry = findEntry(i, M)$
3) $\quad repository = getRepository(entry)$
4) $\quad name = getClassName(entry)$
5) $\quad class = getClassFile(name, repository)$
6) $\quad S = getTargetSites(i, P)$
7) $\quad$ **for each** $j \in S$ **do**
8) $\qquad shipRDF(entry, j)$
9) $\qquad shipClass(class, j)$

Figure 14: Code Deployment Algorithm.

Figure 14 presents the algorithm used by QPC to deploy all the Java classes. The algorithm receives three input parameters: 1) R - a list with the URIs for the operators and data types used in the query, 2) M - a structure containing all the metadata for the query, and 3) P - the execution plan for the query. The algorithm iterates over the list of resources R as follows. First, the entry with the metadata for the current URI i is fetched from M. Next, the name of the repository containing the Java class implementing resource i is found. In step (4), the name of the Java class for resource i is determined. With this information the algorithm uses step (5) to retrieve the Java class file for resource i from the code repository. Then, the set S of all sites which require the class for resource i is determined from the query plan P by calling function $getTargetSites()$ in step (6). Having found the target sites, the algorithm iterates over S, and ships the metadata and Java class file for resource i to each site. Notice that in step (8) the metadata is converted to RDF format and then transmitted to the target site. Once the site receives the class file, it loads it into the Java Virtual Machine, and the resource becomes available for use. Notice that this entire process has been completely done by the QPC and totally driven by the metadata retrieved from the catalog. There is no human involvement of any kind, and therefore the functionality has been automatically deployed by MOCHA.

For query $Q2$, the code deployment phase unfolds as follows. The classes for the data types used in columns `R.week`, `R.location` and `R.raster` are ship to the DAP for the Informix server. QPC also ships to it the classes for functions `Equal()` and `Energy()`. Next, QPC ships the classes for the types used in columns `M.name`, `M.location` and `M.map` to the DAP for the Oracle Server. In addition, this DAP receives the class for function `Equal()`. Finally, QPC ships the classes for the columns projected in the query result to the client application. Notice that since QPC has found all the functionality for the query, it can simply load all the Java classes it needs into its run time system. In the case of $Q2$, it needs all the classes for the columns in each table, plus the classes for functions `Equal()` and `Overlay()`. Once each component has extended its query execution capabilities, the query is ready to be solved.

5.6 Data Processing

Once the plan to process the query is chosen and all the necessary code has been deployed, the QPC will the start the execution of the query. QPC sends to the DAPs the sub-plan(s) that each DAP must execute and request each DAP to create an iterator to evaluate the sup-plan(s). In MOCHA, sub-plans are also encoded as XML documents, since this approach frees the developer of the DAP from the inconvenience of learning the specific protocol and APIs used to exchange the plans between the QPC and the DAPs. Also, this approach provides interoperability for control exchange. The XML document for a plan is simply parsed at the DAP and converted to the data structure that better fits the needs of the DAP been developed.

```
<plan>
  <tables>
    <table>Rasters R</table>
  </tables>
  <columns>
    <column>R.week</column>
    <column>R.location</column>
    <column>R.raster</column>
  </columns>
  <constants>
    <const> X </const>
  </constants>
  <operators>
    <operator> Energy </operator>
  </operators>
  <projections>
    <projection>
      <colname>R.week</colname>
    </projection>
    <projection>
      <colname>R.location</colname>
    </projection>
    <projection>
      <colname>R.raster</colname>
    </projection>
  </projections>
  <restrictions>
    <cluase>
      <GT>
        <arg>
          <operator>Energy</operator>
          <arg>
            <column>R.raster</column>
          </arg>
        </arg>
        <arg>
          <const> X </const>
        </arg>
      </GT>
    </clause>
  </restrictions>
</plan>
```

Figure 15: Encoding of Query Plan

Figure 15 illustrates the XML encoding for the Select node in the plan for query $Q2$ that was presented in section 5.4. The elements with tags `tables`, `columns`, `constants` and `operators` form the preamble of plan. This preamble and the metadata received from the QPC are used by the execution engine to create all the data structures necessary to build the local representation of the query plan that the DAP will use. The elements with tags `tables` and `columns` are used by the DAP to create a SQL string that will be passed to the Informix Server to extract all the columns from tuples in `Rasters` that will be processed. The constants used in the query are contained in element `constants` and the DAP will create object instances with the values for these constants. This task, however, will be postponed until the projection and restriction clauses are processed since the type for the constants will be inferred from these expressions. In the meantime, the DAP creates the appropriate object to evaluate each of the operators specified by the element with tag `operators`. Recall that the structure of these objects was discussed in sections 4.3.1 and 4.3.2. The attributes to be projected and passed to the next node in the query tree are given in the `projections` element. These attributes are extracted from tuples that satisfy the restriction clauses contained in element `restrictions`. The `restrictions` element is composed of conjunctions obtained from the `WHERE` clause of the query.

Once all the data structures are in place, data processing begins. As shown in Figure 16, each DAP uses the JDBC API to extract the tuples from the Informix and Oracle servers. The DAP for the Informix server filters all the tuples been read by using the expression $\texttt{Energy}(\texttt{R.raster}) > \texttt{X}$. All the tuples which satisfy this expression are then used for the semi-join $\texttt{Rasters} \ltimes \texttt{Maps}$. In this semi-join, the expression $\texttt{Equal}(\texttt{R.location}, \texttt{M.location})$ is used as the semi-join predicate. After this semi-join is computed the tuples with attributes `R.location`, `R.week` and `R.raster` are sent to the QPC and stored to disk. In a similar

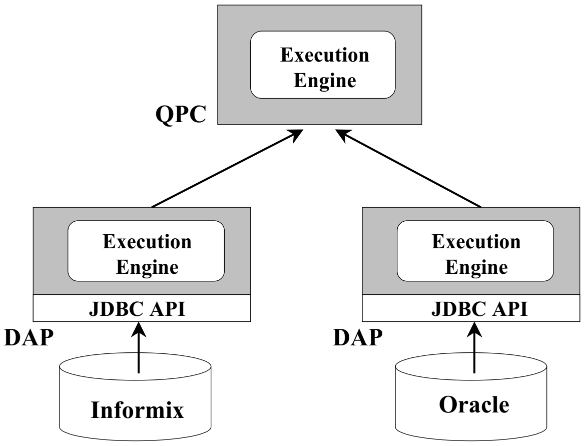

Figure 16: Data Processing

fashion, the DAP for the Oracle database server extracts the data from the database, computes the semi-join `Maps`$\ltimes$`Rasters` and sends the tuples with attributes `M.name`, `M.location` and `M.map` to the QPC, where they are materialized to disk. Finally, the QPC joins the two results received from the DAPs, performs the overlay operation using function `Overlay()`, and projects the attributes for the result. All results are then send to the client application for visualization purposes.

6 Discussion

In this section we describe the main benefits of MOCHA and the status of the current implementation of the system.

6.1 Benefits of MOCHA

In the MOCHA architecture there is a clear specification of the services to be provided by each component in the system. The most important benefits provided by MOCHA are:

1. **Middle-tier Solution** - In MOCHA, the client application does not connect to the data sources directly. Instead, the client leverages the services provided by the middle-tier software layer composed of the QPC and the DAPs. With this approach, clients can be kept as simple as possible, since there is no need to integrate into them the routines necessary to access each data source. This also makes clients easier and inexpensive to set up and maintain.

2. **Minimum Changes on Existing Servers** - MOCHA does not require data to be moved from an existing and adequate data server into a new and possibly different server just for the purpose of interoperability. Instead, the middle-tier layer is configured to provide connectivity and remote access to the data sets stored in the existing server. There is no need to perform costly and error-prone upgrades on the existing servers participating in the distributed system. Thus, existing software is reused and interoperability is still achieved.

3. **Extensibility** - New application-specific software implementing additional features, such as complex types, search operators and numeric aggregates, can be added to MOCHA after it has been deployed. The system is not static, rather it can be extended with additional functionality that is required to support the needs of new applications. Thus, the system can evolve to accommodate the changing requirements of users.

4. **Code Reusability** - MOCHA is implemented in the Java programming language and, as result, all software used in MOCHA is independent of the computer platform being used at any particular data center or client site. MOCHA can support a vast array of hardware platforms, ranging from desktop PCs to multi-processor workstations. Thus, there is no need to perform expensive and time-consuming ports of the software to different platforms. Instead, the software is written once, and then used anywhere in the system. Also, the cost of software maintenance can be significantly reduced since only one port of the software is needed.

5. **Automatic, Plug-&-Play Code Deployment** - MOCHA automatically and seamlessly deploys all the code implementing the application-specific functionality used to process the queries posed to the system. There is no need for the end-users or administrators to make system-wide installations of application-specific software, since MOCHA extracts all code from the code repositories and deploys it as needed and where it is needed.

6. **Efficient Query Evaluation** - MOCHA leverages automatic code deployment in order to provide an efficient query processing service based on data movement reduction. The code and the computation for each of the operators in a query are shipped to and performed at the site which results in minimum data movement over the computer network. This approach not only reduces the time it takes to solve a query, but also increases the query throughput of the system.

7. **XML-based Metadata** - Instead of creating yet another language to represent metadata, MOCHA utilizes the well-accepted XML and RDF standards and leverages the availability of their tools. A schema with the appropriate tags is provided by MOCHA to encode and exchange metadata between the components in the system.

8. **XML-based Control** - Rather than inventing a new control protocol and forcing developers to learn the data structures, formats and APIs needed to implement it, MOCHA encodes all control information as an XML document. A DTD is provided to specify the structure of the plans that must be followed by each DAP during query processing. The developer of a DAP is free to use whatever mechanism he/she prefers to implement the query plan inside the DAP.

9. **Standard Interfaces** - In MOCHA, all the data types and operators are handled by the client, QPC and DAPs through well-known interfaces. There is no need to "hard code" any routine to manipulate a data type or operator. Instead, each class implementing one of these resources simply customizes the methods in the appropriate interface to work according to the semantics of the particular type or operator.

In summary, this framework provides the foundation for a scalable, robust, extensible and efficient middle-tier solution for the data integration and interoperability problems faced by many enterprises.

6.2 Implementation Status

We have implemented MOCHA using Sun's Java Developers Kit (JDK), version 1.2. We used Oracle's Java XML parser to manipulate all the XML documents accessed by the QPC and DAP. The current version of MOCHA includes a DAP for the Informix Universal Server, a DAP for the Oracle 8 Universal Server and one DAP for an XML repository. We have loaded the Informix Universal Server with data from the Sequoia 2000 Benchmark [Sto93]. This benchmark contains data sets with polygons, points, rectangles and AVHRR raster images, all of which were obtained from the US Geological Survey. The Oracle 8 Server was loaded with data sets describing weather forecast images for the Washington Metropolitan Area, which are stored in a Web server. Finally, the XML repository contains forecast temperatures for several of the major cities in the United States. All three of these data sources are hosted at the Department of Computer Science, University of Maryland, College Park. We have also performed extensive measurements on the performance of MOCHA, using the Informix Server and the Sequoia 2000 Benchmark [Sto93]. The results of these measurements on MOCHA can be found in [RMR99].

7 Related Work

Middleware systems have been used as the software layer that attempts to overcome the heterogeneity problem faced when data is dispersed across different data sources. The goal is to shield the applications from the differences in the data models, services and access mechanisms provided by each data source. Typically, middleware comes in two flavors: database gateways and mediator systems. Database gateways are used to import data from a particular data source into a production DBMS made by a different vendor. The gateway provides a data channel between both systems, and therefore, a different gateway is need for each of the different data sources accessed by the DBMS. Some examples of commercial database gateway products are Oracle Transparent Gateways [Cor99] and the Informix's Virtual Table API [Cor97].

The other kind of middleware system is the mediator system. Here a mediator application is used as the integration server and the data sources are accessed through wrappers. The mediator provides very sophisticated services to query multiple data sources and integrate their data sets. Typically, an object-oriented global data schema is imposed on top of the schemas used by the individual data sources. Examples of mediator systems are TSIMMIS [CGMH+94], DISCO [TRV96] and Garlic [RS97]. The work in [dFRH98] considered some of the issues and tradeoffs between the use of gateways or mediator systems.

All these middleware solutions require the administrators to manually install all the necessary functionality for query processing into every site where it is needed. In addition, these systems use ad-hoc or proprietary metadata and control exchange protocols, which make it difficult for third-party developers to create compatible and interoperable software modules for these systems.

8 Conclusions

In this paper we have proposed a novel metadata-driven framework implemented in MOCHA to automatically and seamlessly deploy all application-specific code used during query processing. We have identified the major drawbacks of existing middleware schemes, namely the cost and complexity of porting, manually installing and maintaining middleware code system-wide, and the inability to scale to a large population of clients and servers.

In contrast, MOCHA leverages Java, XML and RDF technologies to provide a robust, efficient and scalable solution in which the functionality is automatically deployed. All the code for data types and query operators is implemented in Java classes, which are stored in code repositories. For each query, MOCHA finds and retrieves all the necessary classes from the code repositories, and ships these classes to the sites that require them. Metadata is used not only to understand the behavior of each type or operator, but also to guide the entire code deployment process. The metadata and control exchange between the components in MOCHA is realized through the well-accepted XML and RDF standards. Future work includes the development of XML-based descriptions of data source capabilities and query plans for non-traditional data sources such as semi-structured databases.

References

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

[Con98] World Wide Web Consortium. Extensible Markup Language (XML) 1.0, February 1998. URL:http://www.w3.org/TR/1998/REC-xml-19980210.

[Con99a] World Wide Web Consortium. Namespaces in XML, January 1999. URL:http://www.w3.org/TR/1999/REC-xml-names-19990114/.

[Con99b] World Wide Web Consortium. Resource Description Framework (RDF) Model and Syntax, February 1999. URL:http://www.w3.org/TR/REC-rdf-syntax/.

[Cor97] Informix Corporation. Virtual Table Interface Programmer's Guide, September 1997.

[Cor99] Oracle Corporation. Oracle Transparent Gateways, 1999. URL:http://www.oracle.com/gateways/html/transparent.html.

[dFRH98] Fernando de Ferreira Rezende and Klaudia Hergula. The Heterogeneity Problem and Middleware Technology: Experiments with and Performance of Database Gateways. In *Proc. VLDB Conference*, pages 146–157, New York, NY, USA, 1998.

[For98] Internet Engineering Task Force. Uniform Resource Identifiers (URI): Generic Syntax, August 1998. RFC2396.

[RMR99] Manuel Rodríguez-Martínez and Nick Roussopoulos. MOCHA: A Self-Extensible Middleware Substrate For Distributed Data Sources. Submitted for publication, 1999.

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

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

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