MEADOW: a middleware for efficient access to multiple geographic databases through OpenGIS wrappers

SP&E

Sang K. Cha[1], Kihong Kim[1], Byung S. Lee[2,*,†], Changbin Song[1], Sangyong Hwang[1] and Yongsik Kwon[1]

[1]*Graduate School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea*
[2]*Department of Computer Science, University of Vermont, Burlington, Vermont 05405, U.S.A.*

SUMMARY

With the proliferation of various geographic databases on the Internet, we have seen increasing needs for accessing them concurrently and remotely via the Web for high-level decision making. In this paper, we present Middleware for Efficient Access to Databases through OpenGIS Wrappers (MEADOW), an object-oriented middleware system we have developed to meet these needs. Current OpenGIS standard addresses many interoperability issues involved in such a global utilization of geographic databases. However, existing Simple Feature specification for CORBA (SFCORBA) implementations of OpenGIS proved to be insufficient for MEADOW. The main problems are the complexity of system development and maintenance, and the inefficiency of accessing remote data servers for processing region queries. We resolved the complexity problem by automatically generating a major portion of the application code, specifically wrappers on database servers and client library modules called transparent access providers. A MEADOW view definition language was developed as a high-level specification language for this purpose. The efficiency problem was resolved by using a region-based group prefetching of spatial objects from a geographic region. In addition, we implemented an OID-based semijoin for efficient global query processing, and a region-level locking to enhance the level of concurrency among region queries. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: geographic databases; middleware; wrapper; caching; OpenGIS; CORBA

*Correspondence to: Byung S. Lee, Department of Computer Science, University of Vermont, Burlington, Vermont 05405, U.S.A.
†E-mail: bslee@cs.uvm.edu

INTRODUCTION

In this paper, we describe our practical experience of building Middleware for Efficient Access to Databases through OpenGIS Wrappers (MEADOW), a middleware that provides Web-based access to multiple, heterogeneous, autonomous geographic databases over the Internet. Specifically, we present the tools developed to aid software construction and maintenance, and the techniques for coping with the demands for efficient access to data sources on a large scale.

With the Internet technology facilitating information dissemination, we have seen an increasing number of geographic database servers publicly accessible over the Internet for geographic information system (GIS) applications such as Internet marketplaces [1] and spatial data mining [2]. As an example, consider a salesperson that plans to visit prospective buyers using a public transportation system. His or her decision on the best travel path depends on various factors including traffic status and desired travel time. In order to support this type of decision making, he or she needs to access all bus-line databases, subway databases, and road-and-traffic databases. It is highly likely that these databases reside on different hardware and software platforms, especially when different authorities are in charge of maintaining them. In other words, the salesperson needs to access multiple, heterogeneous geographical databases that are created and maintained by independent organizations.

To make such a global utilization of geographic databases possible, the OpenGIS standard addresses several interoperability issues. First, it defines a hierarchy of standard geometry types that represent geospatial object shapes [3]. A 'simple feature model' that enables the manipulation of geospatial objects complements this geometry model. Second, it defines standard implementation specifications of simple features in the following three computing environments: CORBA, COM, and SQL [4]. We adopted Simple Features for CORBA (SFCORBA) for the MEADOW project to leverage CORBA capabilities to resolve platform and communication heterogeneities.

Despite the apparent convenience of using a standard, OpenGIS still leaves serious engineering problems. This paper is based on two problems we have resolved—complexity of system development and maintenance, and inefficiency of information retrieval. First, the heterogeneity and autonomy of data sources complicate the development of database servers as well as the middleware linking them to client applications. It also makes subsequent maintenance difficult as the underlying databases and applications evolve. Although SFCORBA enables us to build an interface for accessing remote objects, it requires us to manually code OpenGIS wrappers as CORBA objects on a database system. In addition, it requires a client's binding to a server to be coded manually as well. We handled these problems by automatically generating wrappers on the server side and corresponding transparent access providers (TAPs) on the client side and by providing OpenGIS interfaces through both wrappers and TAPs. For this purpose, we designed high-level languages called view definition language (VDL), text definition language (TDL), and text manipulation language (TML).

The second problem stems from CORBA's inherent coarse granularity of accesses to remote objects. Although often preferred as the basis of implementing a middleware, CORBA object request broker (ORB) [5] is geared for macroscopic integration of software components. Therefore, its blind use in a data-intensive application incurs significant performance degradation [6,7]. We dealt with this problem by introducing three new techniques for efficient query processing: *region-based group fetch* for improving the caching effect on region queries (i.e. queries accessing a region of objects), *OID-based semijoin* for efficient processing of global queries, and *region-level locking* for enhancing the server concurrency level.

Based on these approaches, we designed an architecture that provides transparent, efficient accesses to multiple geographic databases, and evaluated the implemented system using two object-oriented geographic database systems. The results verified the effectiveness of our approaches.

This paper presents the system design and implementation of MEADOW, focusing on its architecture, system components, and techniques for improving performance. Following this introduction, we first present a brief overview of the relevant OpenGIS standard. Then, we describe the run-time system architecture, and follow it with the details of implementations such as VDL, automatic generation of wrappers and TAPs, and TDL/TML, as well as the techniques used to improve the query performance. We then evaluate the results of implementation and discuss related work. Finally, we provide a conclusion and outline future work.

OpenGIS FEATURE MODEL

In this section, we give an overview of OpenGIS feature interface and feature view that are necessary to understand the rest of the paper.

Feature interface

In OpenGIS, a feature denotes any real-world entity, including geographic entities such as a road segment or a lake. A feature has two associated interfaces—a `Feature` interface and a `QueryEvaluator` interface. OpenGIS extends the type system of CORBA interface definition language (IDL) by adding geometry types, such as `Point`, `LineString`, and `Polygon`, for representing spatial attributes.

The following code shows the specification of the `Feature` interface, which is used to read and modify the attributes of a feature.

```
interface Feature {
    readonly attribute FeatureType feature_type;
    any get_property(in Istring name);
    void set_property(in Istring name, in any value);
    ...
};
```

The read-only attribute `feature_type` keeps meta-information such as the names and types of attributes. Access to an attribute is done using two operations: `get_property()` and `set_property()`. For example, if a reference variable `roadSeg` is bound to a road segment object, then `roadSeg.get_property (``centerline'')` retrieves the centerline attribute of the object.

The `QueryEvaluator` interface allows expressing queries in either one of the standard query languages SQL-92 and OQL-93. In addition, it allows including geometric constraint, expressed using spatial operators such as `overlaps` and `contains`, in a query statement. For example, consider the following schema of a road database, which models roads as a network of intersections connected by road segments.

```
// Database schema in ODMG C++ ODL [26]
class Intersection : public d_Object { ... };
class RoadSegment : public d_Object {
    d_String name;
    d_Float width;
    d_Float length;
    d_String category;
    d_Float speed_limit;
    d_Float current_speed;
    LineString centerline; // spatial extension to ODMG ODL
    d_Varray<d_Ref<Intersection> > intersections; // 2 intersections
}
```

The query in Example 1 is for finding all road segments that are longer than 200 m, wider than 12 m, and overlap `queryPolygon`. The database is mapped to `QueryableContainerFeature-Collection` interface, which implements the `QueryEvaluator` interface.

```
// Assume the variable database is bound to the road database
OGIS::QueryableContainerFeatureCollectionRef database;

// Type of query language
OGIS::QueryEvaluator::QLType qlType = OGIS::QueryEvaluator::OQL_93;

char* query_string = "
    select  r
    from    RoadSegment r
    where   r.width >= 12 and r.centerline.length() >= 200 and
            r.centerline.overlaps($1)";

// A query parameter is a pair of type name and value
// Suppose that queryPolygon has been assigned by a user
OGIS::NVPairSeq param; param.length(1);
param[0].name = ``Polygon'';
param[0].value = queryPolygon;

// Query evaluation
OGIS::QueryResultSetIterator result_itr;
try {
   result_itr = database->evaluate(query_string, qlType, param);
} catch (...) {...}
```

Example 1. QueryEvaluator interface to formulate and evaluate a query.

Feature view

Suppose that an application needs a simplified view of the road database as a collection of *roads* instead of a network consisting of road segments and intersections. By grouping a sequence of connected road segments into a road, we can define a view that facilitates the manipulation of data at the road level, such as displaying the name of a road on the screen. The following code shows the corresponding OpenGIS feature view definition.

```
// Desired OpenGIS Feature View in CORBA IDL
module OGIS { ... };
module RoadDB {
   interface Road : OGIS::Feature {
      readonly attribute string name;
      readonly attribute string category;
      readonly attribute float length;
      readonly attribute sequence<LineString> shape;
   };
};
```

Accessing the information through this feature view requires further information for the following mappings between the view and its component feature types.

- *Mapping extents*. For example, a `Road` object is composed of `RoadSegment` objects with the same name.
- *Mapping attribute values*. For example, the `length` attribute of a `Road` object is the sum of the `length` attributes of its component `RoadSegment` objects, and its `shape` attribute is a sequence of the `centerline` attributes of its component `RoadSegment` objects.
- *Mapping attribute accesses*. For example, suppose that a variable `road` is bound to a `Road` object. Then, an application using this feature view will retrieve the `name` attribute of the `Road` object with the code `road.get_property(``name'')`. This code is mapped to a DBMS-specific code that retrieves the `name` attribute of an arbitrary `RoadSegment` object that belongs to the `Road` object bound to the variable `road`.

In MEADOW, the first two types of information are specified in MEADOW VDL. The third type is a mapping between the DBMS interface and OpenGIS standard interface, and varies depending on the DBMS. To facilitate the description of this DBMS-dependent mapping, MEADOW provides a pair of languages, TDL and TML. We will discuss the VDL and TDL/TML further in the next section.

SYSTEM DESCRIPTION

This section details the MEADOW architecture and implementation.

Architecture

Figure 1 shows the high-level run-time architecture of MEADOW, where wrappers on the server side and TAPs on the client side interoperate via the object request broker (ORB). Since MEADOW is based
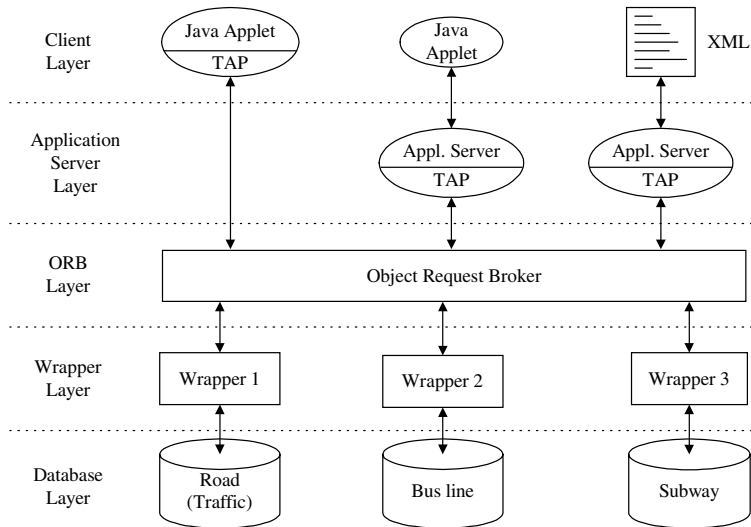
SP&E

Figure 1. Run-time MEADOW architecture.

on OpenGIS SFCORBA, the wrappers are CORBA objects that support the OpenGIS geometry data types and feature model. The TAPs are modules that hide the ORB from clients.

MEADOW generates wrapper and TAP as a pair, automatically, from a feature view definition, underlying database schema, and the rules for mapping between them. The wrapper provides a uniform OpenGIS interface for clients while hiding the specifics of individual databases. The TAPs also provide the same interface for applications, redundantly to the wrapper interfaces. In contrast, other existing SFCORBA systems implement OpenGIS interface on the wrapper side only. Although this is sufficient to allow clients to access server objects, the redundant implementation shields application programmers from dealing with CORBA-specific details. This improves the development productivity and quality substantially, and insulates applications from changes of object prefetching and caching mechanisms (to be elaborated later) used by the TAPs. Note that we still provide an alternate direct path to wrappers for clients without built-in TAPs.

TAPs automatically cache remote server objects in the clients' memory space. The TAPs may reside in either Java applets or application servers. In the former case, a Java applet caches server objects in its own memory and executes an application. This two-tier architecture is suitable for interactive and navigational applications such as map editing or browsing. In the latter case, an application server does the caching, and executes an application sharable by multiple clients. This three-tier architecture is suitable for analytical applications such as finding the shortest driving path, where multiple clients can share the graphs of paths.

MEADOW uses either Java applets or XML document browsers for data presentation on the client side. An application server generates XML documents as query results. We use vector markup language

(VML) tags for representing geometry data [8]. (VML is an early version of geography markup language (GML) [9] that became part of OpenGIS standard after the completion of MEADOW.)

To facilitate the development and maintenance of Web-based GIS applications, MEADOW maintains a global dictionary for bookkeeping the generated wrapper–TAP pairs and their dependency chain all the way down to specific databases. In order to eliminate the overhead of looking up a remote dictionary at run time, all wrappers and TAPs are compiled with the underlying database schema and a feature view. This dictionary is especially useful for maintenance purposes, namely for identifying the wrappers and TAP affected by changes in the server databases and client applications.

MEADOW VDL

In MEADOW VDL, an OpenGIS feature view is defined as a set of feature types (also known as 'view-defined types'). A feature type is defined by its population, properties, and the position in a type hierarchy:

```
interface type_name : inheritance_specification
    ( population_specification )
    { property_specification };
```

The population of a feature type can be specified in four alternative ways: one-to-one mapping, generalization, specialization, or composition.

- In *one-to-one mapping*, the population is the same as the extent of a matching database-resident type.
- In *generalization*, the population is the result of merging the extents of several database-resident types. For example, we can generalize various topologies of road segments such as intersections, branches, and ramps to `RoadNode` type. This generalization can facilitate the construction of a graph for finding the shortest path.
- In *specialization*, the population is a partition of the extent of a database-resident type. For example, road segments can be specialized into highways, national roads, and regional roads.
- In *composition*, the population is a combination of the extents of two or more database-resident types. Typical compositions include combining small objects into a larger aggregate object, creating an object-oriented view of a relational database, and restructuring a set of values into an object [10,11].

The properties may be specified in two ways.

- *Hiding attributes*. We can use `import` statement to render a set of attributes public, thereby hiding the others. If want to specify the attributes to be hidden, we use `import * except` statement instead.
- *Adding attributes*. We can add a derived attribute as a new one.

In Example 2, the `RoadLink` type specifies one-to-one mapping. It has the same extent as the `RoadDB::RoadSegment` type. It hides four attributes `current_speed`, `speed_limit`, `width`, and `category`. In addition, it adds a new attribute `driving_time` derived from two attributes `length` and `current_speed`, where the `read as` clause specifies how to calculate its value and

```
// One-to-one
interface RoadLink : OGIS::Feature
    (RoadDB::RoadSegment)
    {
        import * except current_speed, speed_limit, width, category;
        attribute float driving_time
            read as length / current_speed
            write as current_speed = length * driving_time;
    }

// Generalization
interface RoadNode : OGIS::Feature
    (RoadDB::Intersection, RoadDB::Ramp, RoadDB::Branch)
    {import *;}

// Parameterized specialization
interface RoadSeg(X) : OGIS::Feature
    (RoadDB::RoadSegment where category = X)
    {import *;}

// Composition
interface Road : OGIS::Feature
    (RoadDB::RoadSegment group by name)
    {
        attribute string name read as distinct(name);
        attribute string category read as distinct(category);
        attribute float length read as sum(length);
        attribute sequence<LineString> shape
          read as to_sequence(centerline);
    }
```

Example 2. OpenGIS feature view defined for a database `RoadDB`.

the `write as` clause specifies how to reflect the change of value in the underlying database. If the `write as` clause were omitted, `driving_time` would be for read-only.

The `RoadNode` type specifies a generalization. Its population is defined as the union of the extents of three database-resident types `Intersection`, `Ramp`, and `Branch`.

The `RoadSeg(X)` type specifies a specialization. Its population is defined as a partition of the `RoadSegment` extent. In particular, it shows a parameterized specialization based on the value of the attribute `category`. In effect, this single type is equivalent to several types such as `RoadSeg_Highway`, `RoadSeg_NationalRoad`, and `RoadSeg_RegionalRoad`.

The `Road` type specifies a composition. Its population is defined as a collection of groups of `RoadSegment` objects with the same value of the attribute `name`. In MEADOW, the result of a `group by` operation has the type 'a set of bag of objects'. Therefore, the attributes of a `Road` object need to be derived from those of a bag of `RoadSegment` objects. Example 2 shows three aggregation functions used for this purpose: `distinct()`, `sum()`, and `to_sequence()`, each of which is mapped to a method of RoadWrapper class. For instance, sum(length) is mapped to a method length() as:

```
float length()
    {
        d_Iterator<d_Ref<RoadSegment> > itr = dbObjs.create_iterator();
        d_Ref<RoadSegment> obj;
        float length_sum = 0;
        while ( itr.next(obj) )
          length_sum += obj->length;
        return length_sum;
    }
```

Automatic generation of wrappers and TAPs

Overview

Figure 2 shows the overall process of generating a wrapper and a matching TAP, assuming that the underlying database schema already exists and DBMS-specific mapping rules need to be defined only when using a new DBMS.

1. Define a desired feature view of the database, which consists of a set of feature types, in MEADOW VDL.
2. Generate a CORBA IDL interface declaration for each feature type in the feature view using a MEADOW preprocessor.
3. Compile the generated IDL interfaces into classes in a target programming language (e.g. Java, C++) using an appropriate IDL compiler. Client stubs and the server skeletons are generated as a result.
4. Generate a schema-dependent wrapper code on the server side using the MEADOW preprocessor.
5. Generate a wrapper by compiling the schema-dependent wrapper code and linking the output code with a schema-neutral wrapper library and a DBMS library.
6. Generate a schema-dependent TAP code on the client side using the MEADOW preprocessor.
7. Generate a TAP library by compiling the schema-dependent TAP code and linking the output code with a schema-neutral TAP library. The TAP library is to be linked with an application program.

Automatically generated classes

MEADOW generates one type of classes called *feature proxies* in the schema-dependent TAP code, and two types of classes called *feature wrappers* and *feature shippers* in the schema-dependent wrapper
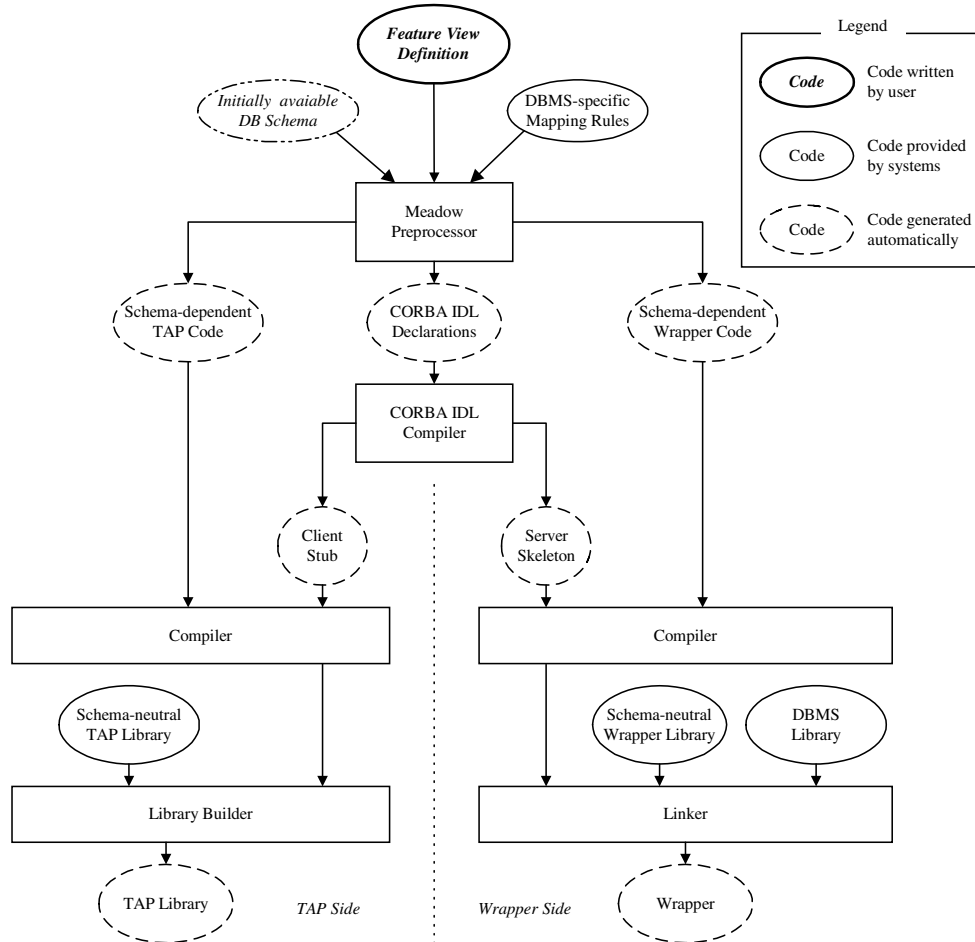
Figure 2. Process of generating a wrapper and a TAP.

code. A feature proxy class implements an OpenGIS simple feature interface specified in a TAP. MEADOW creates one feature proxy object for every remote feature object fetched into the client's memory, and has the proxy object encapsulate the fetched object. A feature wrapper class implements an OpenGIS simple interface specified in a wrapper. Each feature wrapper object encapsulates one remote feature object, and MEADOW creates one when the encapsulated object is accessed directly without the TAP. Note that, if the encapsulated object is accessed through a TAP, no feature wrapper object is created as long as no server method is invoked. A feature shipper class inherits from a feature wrapper class. Only one feature shipper object is created for each feature *type*, and it interacts with the prefetch manager of a TAP to ship objects to the client.
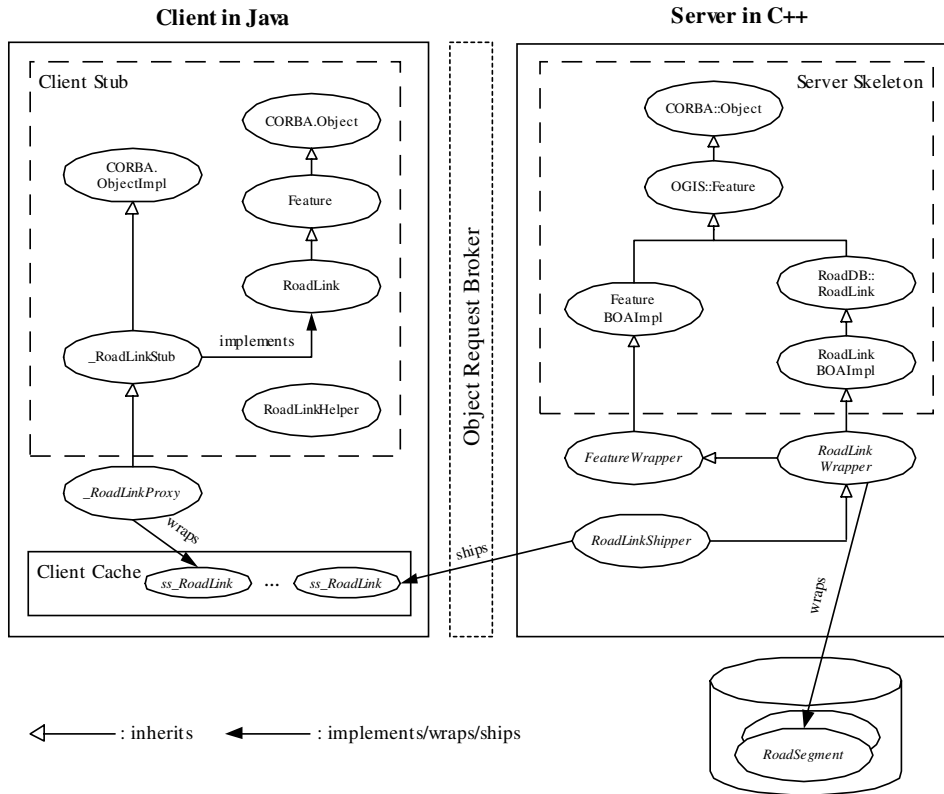
Figure 3. Relationship among automatically generated classes.

Example

Figure 3 illustrates the relationship among the client and server classes generated from the `RoadLink` feature shown in Example 2. An IDL compiler generates the classes inside the dashed boxes, whereas those outside the dashed boxes are generated by the MEADOW preprocessor. The server uses an IDL-to-C++ compiler, and the client uses an IDL-to-Java compiler. We omit name scopes such as `CORBA::` and `OGIS::` whenever unambiguous.

The classes inside the dashed boxes form the stubs on the client and the skeletons on the server, and provide the basic code for client–server communication. The `RoadDB::RoadLinkBOAImpl` class, called the basic object adapter (BOA) implementation of the `RoadLink` interface, implements CORBA-defined methods like `_bind()` that are used to communicate with a client. The `RoadLinkHelper` class in the client implements the matching CORBA methods.

The classes outside the dashed boxes are the proxy, wrapper, and shipper classes described above. Each class has a pair of methods generated for each attribute, one for reading and one for writing.

Examples are `driving_time(void)` method for reading the attribute `driving_time` of the `RoadLink` interface, and `driving_time(float)` for writing it. The `_RoadLinkStub` class in the client, generated by the IDL compiler, provides default implementation of these methods, which in turn invoke the corresponding server methods remotely. However, the `_RoadLinkProxy` class overrides this default implementation, and provides access to the cached objects of the `ss_RoadLink` type.

The `_RoadLinkProxy` class is similar to the `RoadLinkWrapper` class in terms of its method implementation. The difference is that the former wraps the cached objects while the latter wraps the database-resident objects.

Example 3 shows the `RoadLinkWrapper` class generated by the MEADOW pre-processor. This feature wrapper implements two categories of methods: member-access methods like `driving_time()`, and OpenGIS feature methods like `get_property()` and `set_property()`. While the first category can be implemented easily with a simple translation, the second one necessitates a table that maps an attribute name to an associated code. For example, `get_property(''driving_time'')` needs to be mapped to the code that retrieves the attribute `driving_time`. The mapping table is looked up using `lookupGetFunc()` and `lookupSetFunc()` functions.

Example 4 shows the `ss_RoadLink` struct and the `RoadLinkShipper` class generated by the MEADOW preprocessor. The `ss_RoadLink` struct is used for shipping the OID and the data part of a `RoadLink` object from a server and to a client. This struct has the same attributes as the `RoadLink` interface defined in Example 2. The difference is that a struct is instantiated as a data stream, which a client can fetch via the ORB, whereas the interface is instantiated as a CORBA object, which a client can bind to but cannot fetch. The `RoadLinkShipper` class implements object import and export methods that are invoked by the prefetch manager of a TAP. The import method flushes all modified objects from the client cache to the server at transaction commit time to update a shared database. The export method returns a sequence of `ss_RoadLink` structures corresponding to the input sequence of OIDs.

There are three parts in a global object ID: a wrapper ID, a feature type ID, and a database-resident OID. First, a unique number is assigned to a wrapper as its ID when a TAP binds to it. Second, each feature type is given a sequential number as its ID when its view definition is processed. Third, each database-resident object is assigned an OID when created. Within each wrapper, the combination of a feature type ID and a database-resident OID suffices to identify an object uniquely, provided that every database-resident object appears only once in the extent of the feature type. The MEADOW VDL satisfies this condition. For example, the combined ID is used as an OID in the feature shipper interface of Example 4 because the identity of a wrapper can be determined implicitly. A full combination of the three IDs is used only within a TAP.

Text definition language and text manipulation language

As mentioned, MEADOW needs DBMS-specific rules in order to generate the wrapper and TAP codes automatically. These mapping rules are specified in TDL and TML. TDL is a parsing tool used to describe a server-database schema and to resolve DBMS-specific syntactic inconsistencies encountered while parsing the database schema and a feature view definition. Given a description in TDL, the MEADOW preprocessor generates, for each type, meta-information such as the pairs of a feature type

```
class RoadLinkWrapper :
   public RoadLinkBOAImpl, public FeatureWrapper
{
 public:
   // Methods defined in RoadLink interface
   float driving_time()
      { return dbObj->length / dbObj->current_speed; }
   void driving_time(float t)
      { dbObj->current_speed = dbObj->length / t; }
   ...
   // Methods derived from Feature interface
   CORBA::any* get_property(const char* name)
      { return lookupGetFunc(name)(); }
   void set_property(const char* name, CORBA::any* value)
      { lookupSetFunc(name)(value); }
   ...

 private:
   // helping functions for set_/get_property()
   static GetFuncPtr lookupGetFunc(const char* name);
   static SetFuncPtr lookupSetFunc(const char* name);
   ...
   CORBA::any* _get_driving_time()
      { CORBA::any* a = new CORBA::any;
        *a <<= driving_time(); return a; }
   void _set_driving_time(CORBA::any* a)
      { char* tmp; *a >>= tmp; driving_time(tmp); }
   ...
   d_Ref<RoadSegment> dbObj;
};
```

Example 3. Generated feature wrapper.

name and a matching database-resident type name, the names of supertypes, and the specification of attributes.

TML is used to specify how to implement the mapping between a schema and a view in a manner similar to Java server pages (JSPs). Example 5 shows a TML template code for an ODMG-compliant DBMS. Two main features of TML are iteration and substitution. For example, `%foreach(type, one_to_one_types)` specifies the iteration over a set `one_to_one_types` using an iteration variable `type`, and `<%type.name.view%>` specifies the substitution for a variable `type.name.view`. Here we assume that `one_to_one_types` contains the meta-information of all feature types whose populations are specified with one-to-one mapping.

```
// Definitions in IDL
// typedef sequence<octet> OID;
// typedef sequence<OID> OIDSeq;
// struct ss_RoadLink { ... };
// typedef sequence<ss_RoadLink> RoadLinkSeq;
// interface RoadLinkShipper {
//     sequence<ss_RoadLink> object_export (in sequence<OID> ids);
//     void object_import (in sequence<ss_RoadLink> data);
// };

// Generated C++ Definitions
struct ss_RoadLink {
   OID id;
   CORBA::String_mgr name;
   CORBA::Float length;
   CORBA::Float driving_time;
   ss_LineString centerline;
   OIDSeq intersections;
   ...
};

class RoadLinkShipper : public RoadLinkWrapper {
   RoadLinkSeq* object_export(const OIDSeq& ids) {
      RoadLinkSeq* roadlinks = new RoadLinkSeq(ids.length());
      for (int i = 0; i < ids.length(); i++) {
        // operator=(), name, length() are defined in RoadLinkWrapper
        *this = ids[i]; // set this to indicate the feature of ids[i]
        (*roadlinks)[i].id = ids[i];
        (*roadlinks)[i].name = name();
        (*roadlinks)[i].length = length();
        ...
      }
      return roadlinks;
   }
   void object_import(const RoadLinkSeq& data){ ... }
};
```

Example 4. Generated feature shipper.

```
%foreach(type, one_to_one_types)
class <%type.name.view%>Wrapper:
   public <%type.name.view%>BOAImpl
   %foreach(super, type.super_types)
   , public <%super.name%>Wrapper
   %end
{
 public:
   ...
   %foreach(member, type.members)
   <%member.type%> <%member.name%>()
      { <%member.read_as%> }
   void <%member.name%>(<%member.type%> <%member.varname%>)
      { <%member.write_as%> }
   %end
 private
   ...
   %foreach(member, type.members)
   CORBA::any* _get_<%member.name%>()
      { CORBA::any* a = new CORBA::any;
        *a <<= <%member.name%>(); return a; }
   ...
   %end
   d_Ref<<%type.name.db%>> <%type.varname%>;
}
%end
```

Example 5. TML template code for wrapper generation.

In addition, there are DBMS-specific mapping rules specified in TDL and TML.

- *Query language*. Different DBMS products may support different query features and syntaxes. For example, an overlap relationship between two polygons *A* and *B* may be expressed in three different ways depending on the DBMS, e.g. A.overlaps(B), A overlaps B, or overlap(A, B).
- *Data type*. Type names and data representations may vary depending on the DBMS. For example, Polyline or Line has been widely used by DBMS products as the type name of piecewise linear representation of curves, instead of the OpenGIS standard name LineString.
- *Operation*. Different operations may be used to define the same derived attribute. Besides, the same or equivalent operations may be named differently on different DBMSs. It is also possible that an operation is defined as a composition of several other operations.

EFFICIENT RETRIEVAL OF INFORMATION

In this section, we present three ideas we implemented to improve the efficiency of clients' retrieval of data from remote servers: region-based group fetch, OID-based semijoin, and region-level locking.

Region-based group fetch

CORBA uses a *method-shipping* (MS) mechanism for fetching objects from a server to a client via ORB. This mechanism allows a client to retrieve only references to objects and therefore incurs the overhead of accessing a remote server every time an object is to be fetched. This overhead is prohibitive in MEADOW, whose client applications typically require the access of a large number of objects on the server.

Therefore, prefetching and caching is crucial to the efficiency of accessing remote server objects. Recent CORBA revision addresses this issue by introducing an object-by-value (OBV) scheme [2], with which a client retrieves objects themselves instead of references to them. However, this scheme ships only a single object at a time, therefore still lacking the feature of a group prefetching that is widely used in the database field.

In MEADOW, we investigated three alternative prefetching mechanisms: *single object fetch* (SOF), *blind group fetch* (BGF), and *region-based group fetch* (RGF). We will describe these mechanisms in the rest of this subsection, after outlining the general steps taken by a TAP for retrieving each object in the result of a query.

Algorithm 1. Object fetch.

1. A feature proxy object is created and initialized with the OID of an object.
2. The proxy checks whether the object is already in the client cache by looking up an OID table of cached objects.
3. If the object is not in the cache,

    (i) the TAP requests its prefetch manager to fetch the object from a wrapper to its cache space, and register the object in its OID table, and
    (ii) the prefetch manager examines the content of the cached object, and adds references to other objects to be fetched to an object-to-fetch list, which is used as a hint for subsequent prefetches.

4. The proxy binds itself to the cached object.

When a TAP retrieves objects, the three alternative prefetching mechanisms differ in their object-to-fetch lists. In the SOF mechanism, it fetches only one object on demand, and does not prefetch any other object, thus not need an object-to-fetch list. In the BGF mechanism, *all* references in the cached object are added to the object-to-fetch list, under the blind assumption that they may also be fetched soon. In the RGF mechanism, the object-to-fetch list includes only the references to objects spatially adjacent to the cached object. Each object has an associated minimum bounding rectangle (MBR), and is considered spatially adjacent to other objects in the same MBR. We maintain such adjacency information using an index that returns an object-to-fetch list given an MBR. This mechanism is particularly useful for GIS applications, where a region-oriented data access is very common.

In general, a sophisticated prefetching mechanism incurs some performance overhead for individual object fetches. However, it is easily amortized by significant performance gain obtained while accessing a large number of objects in several GIS applications. Later, in the Evaluation section, we will compare the performance of the three prefetching mechanisms.

OID-based semijoin

MEADOW supports the processing of global queries that contain join predicates among tables or collections in different databases. For example, suppose there are two databases—a bus line database (`BusLineDB`) and a road database (`RoadDB`)—maintained at different sites, and we want to execute a query for finding the bus lines that pass through any congested area, under the following two assumptions: (1) a road link is regarded congested when the average driving speed is lower than 8.3 m s^{-1} ($= 30$ km h^{-1}); (2) the geometry of a bus line is extended by five meters in all directions (using a `buffer()` function) to tolerate marginal mapping errors between the geometric representations of a bus line and a road link. Then, the global query, involving the two databases, can be written as follows:

```
select  bl
from    BusLineDB::BusLine bl, RoadDB::RoadLink rl
where   rl.length / rl.driving_time < 8.3 and
        rl.centerline.overlaps(bl.geometry.buffer(5))
```

The join between `RoadLink` objects and `BusLine` objects is a 'buffered spatial join' conditioned on the relationship `overlaps`. MEADOW processes a global join query of this kind using a modified semijoin technique utilizing OIDs (we call it an 'OID-based semijoin'). In contrast with a conventional semijoin technique that requires two join operations, once at each of the two participating sites, the OID-based semijoin requires only one join as long as at least one of the two sites allows accessing its objects by OID. The following algorithm outlines the steps of a global join using the OID-based semijoin.

Algorithm 2. Global join using OID-based semijoin.
Let R and S be the collection of objects to be joined, residing at different sites A and B, respectively. Then, a global join $R\ join(X)\ S$, where X is a join attribute, is processed in the following steps.

1. Choose the collection with the smaller number of objects. Without loss of generality, assume that R (at site A) is smaller.
2. At site A, project the join attribute X of R, and augment the resulting one-column table with the OID field. Then, send the resultant table of two columns (i.e. OID and X) to site B.
3. At site B, process the join on X, and generate a result table of two OID columns—one OID of an object in R, the other of an object in S.

Figure 4 illustrates processing a global query using the OID-base semijoin, coordinated by a TAP. First, after parsing and decomposing the query into two subqueries Q1 and Q2, the TAP sends the subquery string Q1 to the wrapper for `RoadDB`. Then, the wrapper processes Q1 and returns the table `tbl` of two columns: the OID and the attribute `centerline` of `RoadLink` objects. Second, the TAP sends the intermediate table `tbl` and the subquery string Q2 to the wrapper for `BusLineDB`.
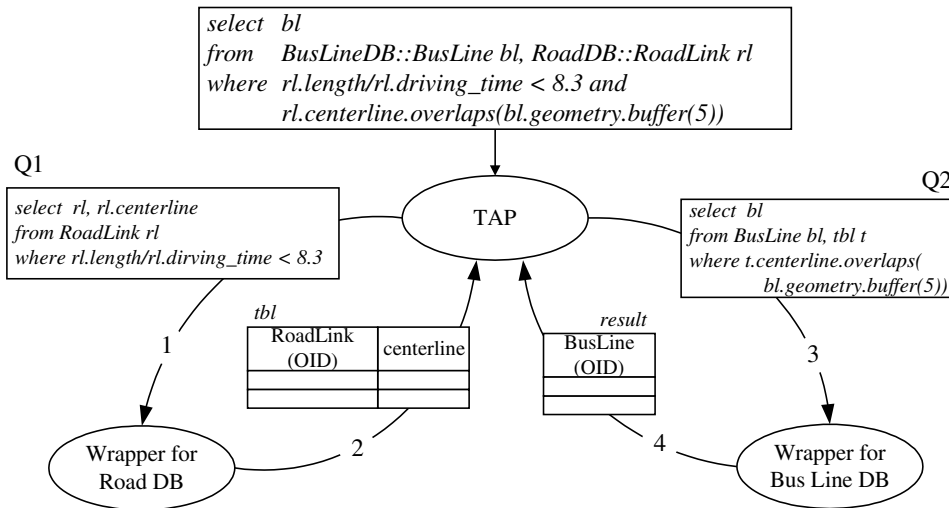
Figure 4. Global query processing.

The wrapper processes Q2 and returns the table `result` of one column, the OID of `BusLine` objects. Finally, upon receiving the table `result`, the TAP fetches the BusLine objects using the OIDs in the table.

Region-level locking

MEADOW's concurrency control isolation level is determined using a global lock manager and a global transaction manager. Region-oriented data access, commonly executed in GIS applications, usually requires long transactions, which are not well supported by strict concurrency control schemes used by traditional DBMSs. Therefore, MEADOW loosens the control by allowing transactions to perform dirty reads, and uses a proprietary mechanism for notifying updates on a region. That is, a transaction is allowed to read a dirty data item, and is notified of subsequent updates at times determined by a notification mechanism that will be explained below.

To support such a loose concurrency control, a wrapper uses two region-based notification lock modes, *read with update-notification request* (RNR) and *write with update-notification request* (WNR), in addition to the traditional read and write lock modes. A transaction holding an RNR lock on a region is allowed to read dirty data, and is notified of subsequent updates on objects in the region. A transaction holding a WNR lock on a region allows its data to be overwritten by other transactions holding the same WNR lock, and is notified of the updates on objects in the region.

MEADOW uses three alternative notification types—*immediate*, *threshold*, and *no*—to control the timing of notification. The immediate notification type requires every update to be notified immediately, whereas the no notification type defers it until the transaction commits. The threshold

Table I. Lock compatibility matrix.

	Requested mode			
Current mode	R	W	RNR	WNR
R	Y	N	Y	N
W	N	N	Y	N
RNR	Y	Y	Y	Y
WNR	N	N	Y	Y

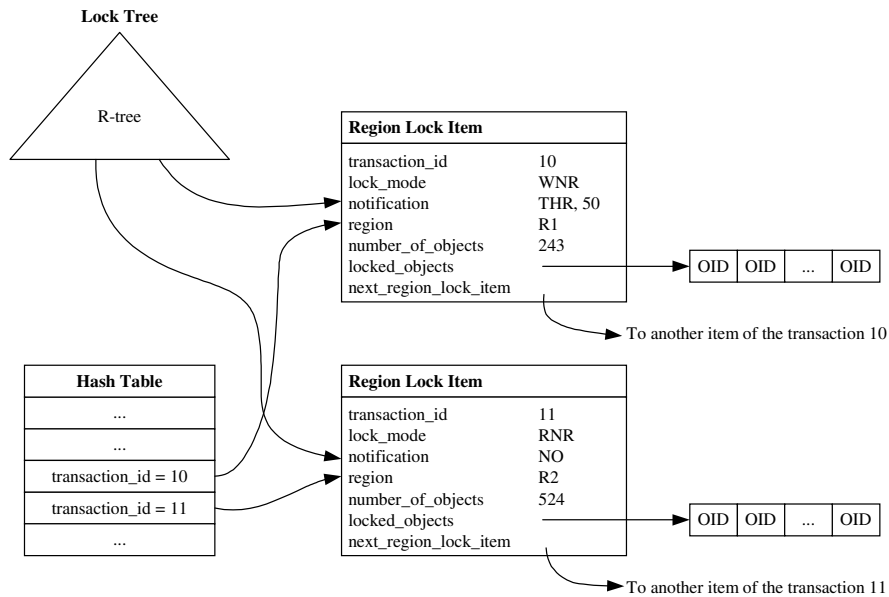R: read lock; W: write lock; RNR: read with update notification request; WNR: write with update notification request.

Figure 5. Internal logic of region lock manager.

notification is a compromise between the two, that is, notification is made when the number of objects updated in a region exceeds a given threshold value.

Table I shows the lock compatibility matrix. Read-only applications such as browsing or monitoring a region can benefit from using the RNR lock. The RNR lock allows other transactions to modify the data, thus potentially causing non-repeatable reads. However, this is rather desirable in a long-running GIS transaction because the transaction can always read data up to date. Similarly, an update transaction holding a WNR lock does not conflict with another transaction requesting the same WNR

lock, thus keeping the data current for all transactions. This is especially useful when the updated data is stateless, such as shared traffic data that resides outside the database and is therefore independent of the internal database state. The WNR lock is also useful for design transactions in the geospatial domain, where a designer edits a region of a map and merges the edited result with the results edited by other designers.

Figure 5 illustrates the internal logic of our region lock manager. The hash table represents a region lock management table. It shows that a transaction with ID 10 (T10) holds a WNR lock on region R1, and another transaction with ID 11 (T11) holds an RNR lock on region R2. An R-tree [7,12] is used to keep track of the transactions holding locks on each region. The lock management table and the R tree are used in the following manner. Suppose a notification of T10's updates is to be sent to the other transactions. The region lock manager searches the lock management table to find the first region lock held by T10, and chases the next-region-lock-item field to find all the other region locks held by T10. Then, given the 'region' field in each Region Lock Item record, it searches the R-tree to find the transactions to be notified of the updates by T10.

EVALUATION

In this section, we present the status of MEADOW implementation, and discuss how well the implementation accomplished our two goals: (1) facilitating system development and maintenance with automatic code generation, and (2) expediting information retrieval from multiple remote data servers, for which we concentrate on evaluating the prefetching mechanism. In addition, we discuss the limitations of the current MEADOW implementations.

Implementation status

A prototype MEADOW middleware was written in about 70 000 lines of C++ and Java code implementing the MEADOW preprocessor and the schema-independent wrapper libraries and TAP libraries. When applied to a transportation information system, MEADOW automatically generated about two thirds of the 36 000 lines of code needed for running a Web-based application. The remaining one third was almost equally divided between the application logic and GUI, and was coded manually.

The middleware was tested on Solaris and Windows NT with OrbixMT [13] and OrbixWeb [14] for CORBA. The data servers utilized two object-oriented DBMSs, Objectivity/DB [15] and OMEGA [16]. Objectivity/DB had no direct support for spatial data and therefore we added a thin spatial extension layer to support the OpenGIS geometry model [17]. OMEGA is a research prototype supporting a spatial data model and a spatial query language on top of the SHORE storage system [18].

Ease of development and maintenance

The generation of wrappers and TAPs are fully automatic in MEADOW, and manual efforts are limited to developing the application logic and GUI. To demonstrate the effectiveness of our approach, we constructed a geographic database from a 1:5000 digital map of the Seoul metropolitan area and implemented several applications. The database contained data about road segments, bus lines, subway lines, contours, and so on. Its schema consisted of 14 classes and about 100 attributes. We partitioned the database into two servers running Objectivity/DB and OMEGA.
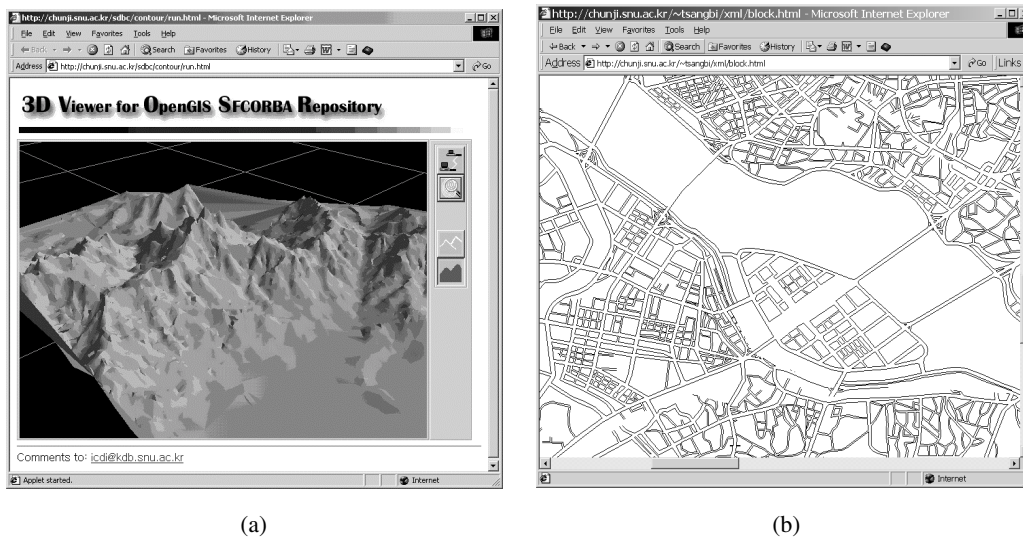
Figure 6. Implemented Web clients: (a) 3D view of contour data in Java; (b) map in XML.

We developed several applications including three map browsers, one map editor, and one 3D contour browser. For this development, we wrote the definitions of feature views in just 235 lines of MEADOW VDL code, from which MEADOW generated TAP and wrapper libraries in 23 400 lines of C++ and Java codes. Since the generated TAPs handle client-side caching and prefetching, application developers could focus on implementing the application logic. As a result, a complicated Window-based map editing/browsing program was implemented in only 6700 lines of application logic code and 6100 lines of GUI code.

Figure 6 shows two examples of Web-based applications we implemented. The first example is the picture of a mountain area in Seoul, displayed with a Java applet. This applet retrieves the contour data in a given rectangular area from a database server, generates 3D Triangular Irregular Network (TIN) data from the contour LineString data, and renders them. This applet allows a user to rotate and zoom the view using a mouse or a keyboard. The second example shows a map displayed with an XML document generated by an application server. The following code shows a part of the XML document.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="block.xsl"?>
<v:polyline points = ''1344 2054 1348 2048 ... 1292 2043 ''
     href="..." fillcolor="#FFFFFF"
     strokecolor="#000000" strokeweight="1px">
   </v:polyline>
<v:polyline ...> </v:polyline>
...
```

In this code, the `v:polyline` tag is defined in the MEADOW VML. Each `polyline` element can have a predefined set of attributes. The `points` attribute stores a list of point pairs that defines a sequence of line segments. The `href` attribute stores a URL link to another document.

MEADOW is as scalable and extensible as the underlying CORBA (we do not have control over the CORBA implementation). In addition, MEADOW can readily incorporate new services available from other data sources through wrappers. For example, current object-relational DBMSs facilitate providing GIS-related services through their extension modules (e.g. Oracle Data Cartridges, DB2 Extenders). With the help of automatic wrapper generator, MEADOW absorbs these services into its suite of wrapped services. Last but not the least, CORBA is interoperable with other middleware technologies such as Java Enterprise Beans and COM+ Components, thus furthering the horizon of MEADOW.

Retrieval efficiency with prefetching and caching

To demonstrate the efficiency of the RGF, we performed an experiment for comparing the RGF with three alternative object fetch methods (MS, SOF, and BGF) introduced previously. The test query was a 'refining query', which incrementally asks for detailed information about a cluster of objects. A possible scenario is that a user selects a geographic region on screen and then requests details about the menus and prices at restaurants in the region. In our experiment, region queries retrieved the MBRs and OIDs of all spatial objects intersecting a given query window, and then refining queries retrieved objects in ten random sub-regions, each of which is bounded by one tenth the first query window size.

The test data set was from the database of road networks in Seoul metropolitan area. It contained 106 533 road segment objects, 71 213 intersection objects, and thousands of landmark objects. We performed the experiment on two SUN UltraSPARC stations and one Pentium II PC connected through 10 Mbps LAN. We used the SUN workstations as database servers, and the PC for running client applications written in Java with OrbixWeb.

Figure 7(a) shows the elapsed transfer time for shipping objects from a server to a client. The speed-up from MS to SOF is about 6 times, from SOS to BGF is 46 times, and from BGF to RGF is 1.4 times. Figure 7(b) shows the number of interactions that occurred between the client and the server during the query processing. The number was reduced by a factor of six from MS to SOS, 387 from SOS to BGF, and 1.7 from BGF to RGF. As we expected, RGF outperformed all the others for a refining query that visits a cluster of objects in the same region.

Limitations

MEADOW has a number of limiting factors. First, it provides no implementation of OpenMap standards, which is just out of the project scope although easy to implement. Second, it lacks a versatile global schema design language like SchemaSQL [19]. The project focus is not on designing or using such a language. Third, it compromises some of the idiosyncratic features of local data sources in order to make a global access possible for users, which happens commonly to federated databases.

RELATED WORK

At the core of MEADOW is building an interoperable GIS framework that integrates heterogeneous data sources into a virtual database. This defines the following two areas related to MEADOW.
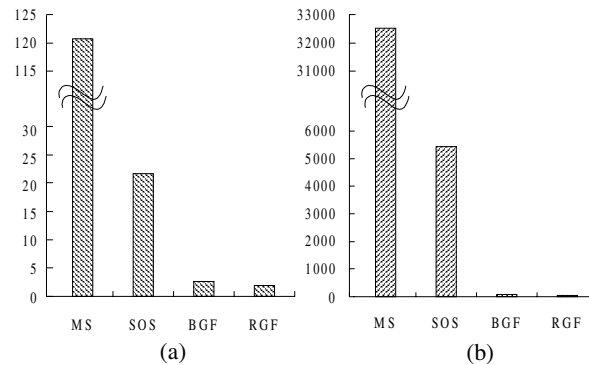
SP&E

Figure 7. Comparison of prefetching schemes: (a) object transfer time; (b) number of interactions.

Implementing interoperable GIS

Until recently, most efforts for building an interoperable GIS considered OpenMap only, primarily due to the lag of OpenGIS Simple Features specifications. (OpenMap specifies an interface between user interfaces and applications, whereas OpenGIS Simple Features specifies an interface between applications and data sources.) Let us briefly describe two cases that use OpenMap only and two cases that use both OpenMap and OpenGIS Simple Features, and state where MEADOW stands in that context.

Nebert and Doyle's implementation [20], sponsored by US Federal Geographic Data Committee (FGDC), was based on OpenMap alone because no Simple-Features-compliant product was available at that time. The key architectural component enabling interoperability in OpenMap is a 'specialist' object, which implements methods inherited from the Specialist class so that an OpenMap client (e.g. browser) can communicate with it. Cranston *et al.* built a 'SAND specialist' [21] that connects an OpenMap client to their SAND spatial database system via CORBA middleware. Because their goal was to demonstrate an OpenMap browser showing multiple maps from multiple sources overlaid on screen, no effort was made to implement OpenGIS Simple Features. An OpenMap specialist is similar to a MEADOW wrapper in its functionality.

Doyle *et al.* has made another demonstration [22] at GEObit trade show, this time including OpenGIS Simple Features. The GEObit demo extended the FGDC demo with more data sources and associated specialists for OpenMap clients, and used OpenGIS Simple Features for SQL instead of SFCORBA. The EXC3ITE project [23] at Defence Science and Technology Organization is implementing SFCORBA. With the goal of providing large-scale access to diverse geospatial data sources, this project implements not only OpenGIS Simple Features but also simple Java clients for plugging into an OpenMap product.

MEADOW also implements OpenGIS Simple Features specifications. However, it is not particularly concerned with OpenMap, although its Java applets or XML document browsers can be readily incorporated into OpenMap clients. Besides, MEADOW OpenGIS Simple Features is not for SQL but for CORBA because the data sources do not necessarily understand SQL. (It is not for COM,

either, because the data servers are not Microsoft platforms.) In particular, MEADOW is characterized by it use of TAPs which, redundantly to wrappers, present encapsulated services of data sources to client applications. To our knowledge, MEADOW is the only system that offers such a configuration.

Integrating heterogeneous data sources

When viewed as a data integration system, MEADOW relates to TSIMMIS [24] and Garlic [25], the two recent developments in this area. TSIMMIS aims at providing an integrated access to heterogeneous data sources via middleware. It uses a self-describing object model—called Object Exchange Model (OEM)—as the global model to facilitate integrating unstructured as well as structured data. The architectural focus is on mediators and translators (equivalent to wrappers). Mediators invoke one or more wrappers and integrate returned results on behalf of the application, and wrappers translate queries into source-specific commands and the results into application-specific formats. Wrappers are generated based on templates prepared for commonly used queries. Then, a wrapper accepts queries matching a template, performs actions specified in the template, and returns the resulting objects. In addition, a wrapper can post-process intermediate results from a data source with limited query processing capability, thus extending the capability of the source. Wrapper generation is semi-automatic, facilitated by a library of commonly used functions.

Garlic aims at providing an integrated view of heterogeneous data sources via middleware. It uses a global data model and programming interface based on the Object Data Management Group standard [26] to facilitate integrating diverse legacy data sources. One of Garlic's foci is on flexible query processing against data sources with different requirements of processing time and complexity. For this purpose, wrappers translate queries and results between the global model and the local models of individual data sources. Each wrapper is capable of generating its own local query execution plan for incorporation into a global plan as well as executing the local plan as part of the global execution. Wrapper generation is manual, or its automation is not a concern to Garlic.

While MEADOW is similar to TSIMMIS and Garlic in providing an integrated access to heterogeneous data sources via a middleware and using wrappers to encapsulate heterogeneous data sources, it is different in its specific objective, data model, and wrapper generation. First, MEADOW aims at providing a computing environment for facilitating the development and maintenance of systems implementing SFCORBA, not just providing integrated access or view to data sources. Second, MEADOW uses an object model defined by OpenGIS Simple Features specifications as its global data model and uses OpenGIS feature view as the basis of automatically generating wrappers. Third, when MEADOW generates a wrapper, it also generates the associated TAP and the mapping rules between them. Furthermore, it uses the TAP to prefetch and cache data from the wrapper to answer queries quickly.

CONCLUSION AND FUTURE WORK

In this paper we presented our experience of building an OpenGIS SFCORBA-based middleware system for efficient Web-based accesses to multiple, heterogeneous geographic databases, of which the primary focus was to provide additional facilities to solve problems occurring when using CORBA as is to build a GIS platform. Named MEADOW, it provides a VDL for automatically generating OpenGIS server wrappers on existing databases, and corresponding client TAPs. Desired OpenGIS feature views

(written in VDL), database schema, and DBMS-specific mapping rules guide the automatic generation of wrappers and TAPs. The significant benefit of such an automatic code generation was ascertained through application to a government transportation information system.

A TAP, in cooperation with a wrapper, processes global queries on multiple databases, and makes it efficient by using region-based prefetching and caching. Our experiment showed an order of magnitude speed-up of interactive Web-based applications compared with naïve implementations of SFCORBA. We also described two additional methods of improving the performance—OID-based semijoin and region-level locking.

Our immediate future work includes utilizing high-level, domain-specific semantics for improving the efficiency even further. For instance, we can extend the current region-based prefetching scheme to make use of the relationships among geographic objects. Another on-going direction is to include other kinds of DBMSs in the MEADOW system environment. We are currently working on incorporating commercial object-relational DBMSs (Informix and UniSQL) and our own highly parallel main-memory DBMS [27–29].

Global utilization of multiple geographic databases requires the resolution of several semantic heterogeneity issues such as different levels of abstraction and different criteria to application completeness and correctness [30]. For example, the geometry of a bus line stored in one database may not exactly coincide with that of matching road segments in another database. Wiederhold [31] suggested mediator architecture for dealing with such semantic heterogeneity issues. We are investigating the feasibility of using MEADOW as a tool for building mediators.

REFERENCES

1. Abel DJ. Spatial Internet marketplaces. *Proceedings of the International Symposium on Spatial Databases*, Berlin, Germany, July 1997; 3–8.
2. Koperski K, Adhikary J, Han J. Spatial data mining: Progress and challenges. *Proceedings of the Workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, QC, May 1996.
3. Buehler K *et al.* The OpenGIS guide: Introduction to interoperable geoprocessing. *Technical Report*, Open GIS Consortium, Inc., 1996.
4. Open GIS Consortium, Inc. *OpenGIS Simple Features Specification for CORBA*, revision 1.0. 1998.
5. Siegel J. *CORBA: Fundamentals and Programming*. Wiley: New York, 1996.
6. Dogac A, Dengi C, Öszu MT. Distributed object computing platforms. *Communications of the ACM*. ACM Press, 1998; **41**(9):95–103.
7. Guttman A. R-trees: A dynamic index structure for spatial searching. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, MA, June 1984; 47–57.
8. Mathews B, Lee D, Dister B, Bowler J, Cooperstein H, Jindal A, Nguyen T, Wu P, Sandal T. Vector markup language. *World Wide Web Consortium Note*. http://www.w3.org/TR/NOTE-VML [1998].
9. Geography Markup Language (GML) 2.0. http://www.opengis.net/gml/01-029/GML2.html [February 2001].

10. Rundensteiner EA. Multiview: A methodology for supporting multiple views in object-oriented databases. *Proceedings of the International Conference on Very Large Data Base (VLDB)*, Vancouver, BC, August 1992; 187–198.

11. Abiteboul S, Bonner A. Objects and views. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991; 238–247.

12. Beckmann N, Kriegel H-P, Schneider R, Seeger B. The R*-tree: An efficient and robust access method for points and rectangles. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, June 1990; 322–331.

13. IONA Technologies Ltd. *Orbix Programming Guide*. 1997.

14. IONA Technologies Ltd. *OrbixWeb Programmers's Guide*. 1997.

15. Objectivity, Inc. *Using Objectivity/C++*, version 4. 1996.

16. Park H-H, Lee Y-J, Chung C-W. Spatial query optimization utilizing early separated filter and refinement strategy. *Information Systems* 2000; **25**(1):1–22.

17. Cha SK, Kim K, Song C, Kim J, Kwon Y. A middleware architecture for transparent access to multiple spatial object databases. *Interoperating Geographic Information Systems*, ch. 22, Goodchild MF, Egenhofer MJ, Fegeas R, Kottman CA (eds.). Kluwer: Dordrecht, 1999; 267–282.

18. Carey MJ *et al.* Shoring up persistent applications. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MA, June 1994; 383–394.

19. Lakshmanan LVS, Sadri F, Subramanian IN. SchemaSQL—A language for interoperability in relational multi-database systems. *Proceedings of the International Conference on Very Large Data Base (VLDB)*, Mumbai, India, September 1996; 239–250.

20. Nebert D, Doyle A. Discovery and viewing of distributed spatial data: The OpenMap testbed. *Proceedings of the Earth Observation and Geo-spatial Web and Internet Workshop*, 1998.

21. Cranston CB, Brabec F, Hjaltason CR, Nebert D, Samet H. Adding an interoperable server interface to a spatial database: Implementation experience with OpenMap. *Proceedings of the International Conference on Interoperating Graphic Information Systems (INTEROP)*, Berlin, Germany, March 1999; 115–128.

22. Doyle A, Dietrick D, Ebbinghaus J, Ladstatter P. Building a prototype OpenGIS demonstration from interoperable GIS components. *Proceedings of the International Conference on Interoperating Graphic Information Systems (INTEROP)*, Berlin, Germany, March 1999; 139–149.

23. Defence Science and Technology Organisation. OpenGIS Simple Features for CORBA work in DSTO. http://www.opengis.org/techno/interop/dsto/.

24. Chawathe S, Garcia-Molina H, Hammer J, Ireland K, Papakonstantinou Y, Ullman J, Widom J. The TSIMMIS Project: Integration of heterogeneous information sources. *Proceedings of the Information Processing Society of Japan (IPSJ) Conference*, Tokyo, Japan, October 1994; 7–18.

25. Carey M, Haas LM, Schwarz PM, Arya M, Cody WF, Fagin R, Flickner M, Luniewski AW, Niblack W, Petkovic D, Thomas J, Williams JH, Wimmers EL. Towards heterogeneous multimedia information systems: The Garlic approach. *Proceedings of the International Workshop on Research Issues in Data Engineering-Distributed Object Management (RIDE-DOM)*, Taipei, Taiwan, March 1995.

26. Cattell RGG, Barry DK (eds.). *The Object Database Standard: ODMG-2.0*. Morgan Kaufmann, 1997.

27. Park J, Kwon Y, Kim K, Lee S, Park B, Cha S. Xmas: An extensible main memory storage system for high-performance applications. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998; 578–580.

28. Lee J, Kim K, Cha SK. Differential logging: A cumulative and associative logging scheme for highly parallel main memory database. *Proceedings IEEE International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001; 173–182.

29. Cha SK, Hwang S, Kim K, Kwon K. Cache-conscious concurrency control of main memory indexes on shared-memory multiprocessor systems. *Proceedings of the International Conference on Very Large Data Base (VLDB)*, Rome, Italy, September 2001; 181–190.

30. Devogele T, Parent C, Spaccapietra S. On spatial database integration. *International Journal of Geographical Information Science* 1998; **12**(4):335–352.

31. Wiederhold G. Mediators in the architecture of future information systems. *IEEE Computer* 1992; **25**(3):38–49.

32. Včkovski A, Brassel KE, Schek H-J (eds.). *Proceedings of the Second International Conference on Interoperating Geographic Information Systems (INTEROP)*, March. Springer: Berlin, 1999.

33. Sellentin J, Mitschang B. Data-intensive intra- and Internet applications: Experiences using Java and CORBA in the World Wide Web. *Proceedings IEEE International Conference on Data Engineering (ICDE)*, Orlando, FL, February. 1998; 302–311.

34. Seetharaman K. The CORBA connection. *Communications of the ACM*. ACM Press, 1998; **41**(10):34–36.

35. Egenhofer MJ. Reasoning about binary topological relations. *Proceedings of the 2nd International Symposium on Spatial Databases*, Zurich, Switzerland, August 1991; 143–160.