Don't Scrap It, Wrap It!
A Wrapper Architecture for Legacy Data Sources

Mary Tork Roth
IBM Almaden Research Center
torkroth@almaden.ibm.com

Peter Schwarz
IBM Almaden Research Center
schwarz@almaden.ibm.com

Abstract

Garlic is a middleware system that provides an integrated view of a variety of legacy data sources, without changing how or where data is stored. In this paper, we describe our architecture for wrappers, key components of Garlic that encapsulate data sources and mediate between them and the middleware. Garlic wrappers model legacy data as objects, participate in query planning, and provide standard interfaces for method invocation and query execution. To date, we have built wrappers for 10 data sources. Our experience shows that Garlic wrappers can be written quickly and that our architecture is flexible enough to accommodate data sources with a variety of data models and a broad range of traditional and non-traditional query processing capabilities.

1 Introduction

Most large organizations have collected a considerable amount of data, and have invested heavily in systems and applications to manage and access that data. It is increasingly clear that powerful applications can be created by combining information stored in these historically separate data sources. For example, a medical system that integrates patient histories, EKG readings, lab results and MRI scans would greatly reduce the amount of time required for a doctor to retrieve and compare these pieces of information before making a diagnosis.

Garlic is a middleware system that provides an integrated view of heterogeneous legacy data without changing how or where the data is stored. Middleware systems leverage the storage and data management facilities provided by legacy systems, providing a unified schema and common interface for new applications without disturbing existing applications. Freed from the responsibilities of storage and data management, these systems focus on providing powerful high-level query services for heterogeneous data.

Middleware systems typically rely on *wrappers* [4] [18] [9] that encapsulate the underlying data and mediate between the data source and the middleware. The wrapper architecture and interfaces are crucial, because wrappers are the focal point for managing the diversity of data sources. Below a wrapper, each data source, or *repository*, has its own data model, schema, programming interface, and query capability. The data model may be relational, object-oriented, or specialized for a particular domain. The schema may be fixed, or vary over time. Some repositories support a query language, while others are accessed using a class library or other programmatic interface. Most critically, repositories vary widely in their support for queries. At one end of the spectrum are repositories that only support simple scans over their contents (e.g., files of records). Somewhat more sophisticated repositories may allow a record ordering to be specified, or be able to apply certain predicates to limit the amount of data retrieved. At the other end of the spectrum are repositories like relational databases that support complex operations like joins or aggregation. Repositories can also be quite idiosyncratic, allowing, for example, only certain forms of predicates on certain attributes, or joins between certain collections. The wrapper architecture of Garlic [4] addresses the challenge of diversity by standardizing how information in data sources is described and accessed, while taking an approach to query planning in which the wrapper and the middleware dynamically determine the wrapper's role in answering a query.

This paper describes the Garlic wrapper architecture, and summarizes our experience building wrappers for ten data sources with widely varying data models and degrees of support for querying. The next section gives a brief overview of Garlic, and is followed by a section that summarizes the goals of the wrapper architecture. Section 4 describes how a wrapper is built, and Section 5 discusses the current status of our system. Section 6 briefly summarizes related work, and Section 7 concludes the paper and presents some opportunities for future research.

2 An Overview of Garlic

Garlic applications see heterogeneous legacy data stored in a variety of data sources as instances of objects in a unified

This work was partially supported by DARPA Contract F33615-93-1-1339.

Proceedings of the 23rd VLDB Conference Athens, Greece, 1997.

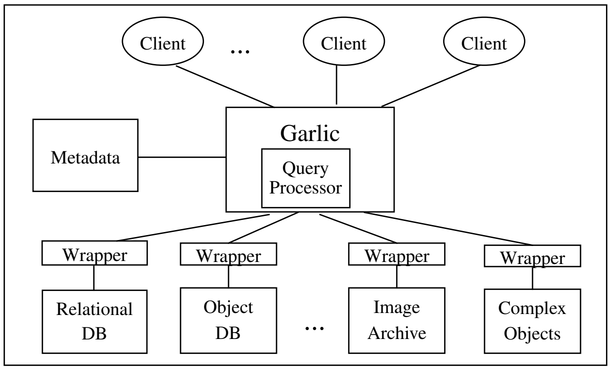

Figure 1. The Garlic Architecture.

schema. Rather than invent yet another object-oriented data model, Garlic's data model and programming interface are based closely on the Object Database Management Group (ODMG) standard [5]. Methods are of particular importance to Garlic, since they provide a convenient and natural way to model the specialized search and data manipulation facilities of non-traditional data sources. By extending SQL to allow invocations of such methods in queries, Garlic provides a single straightforward language extension that can support many different kinds of specialized search.

The overall architecture of Garlic is depicted in Figure 1. Associated with each repository is a wrapper. In addition to the repositories containing legacy data, Garlic provides its own repository for *Garlic complex objects,* which users can create to bind together existing objects from legacy repositories. Garlic also maintains global metadata that describes the unified schema. Garlic objects can be accessed both via a C++ programming interface and through Garlic's query language, an extension of SQL that adds support for path expressions, nested collections and methods. The heart of the Garlic middleware is the query processing component. The query processor develops plans to efficiently decompose queries that span multiple repositories into pieces that individual repositories can handle. The query execution engine controls the execution of such a query plan, by assembling the results from the repositories and performing any additional processing required to produce the answer to the query.

3 Goals for the Wrapper Architecture

Our experience in building wrappers for Garlic confirms that the architecture we describe in this paper achieves several goals that make it well-suited to integrate a diverse set of data sources. We summarize these goals here before describing the wrapper architecture in detail.

1. *The start-up cost to write a wrapper should be small.* We expect a typical Garlic application to combine data from several traditional sources (e.g., relational database systems from various vendors) with data from a variety of non-traditional systems such as image servers, searchable web sites, etc., and one-of-a-kind sources such as a home-grown molecular similarity search engine. Although Garlic is intended to ship with wrappers for popular data sources, we must rely on third party vendors and customer data administrators to provide wrappers for more specialized data sources. To make wrapper authoring as simple as possible, we require only a small set of key services from a wrapper, and ensure that a wrapper can be written with very little knowledge of Garlic's internal structure. In our experience, a wrapper that provides a base level of service can be written in a matter of hours. Even such a basic wrapper permits a significant amount of the repository's data and functionality to be exposed through the Garlic interface.

2. *Wrappers should be able to evolve*. Our standard methodology in building wrappers has been to start with a version that models the repository's content as objects and allows Garlic to retrieve their attributes. We then incrementally improve the wrapper to exploit more of the repository's native query processing capabilities.

3. *The architecture should be flexible and allow for graceful growth.* We require only that a data source have some form of programmatic interface, and we make no assumptions about its data model or query processing capabilities. Wrappers for new data sources can be integrated into existing Garlic databases without disturbing legacy applications, other wrappers, or existing Garlic applications.

4. *The architecture should readily lend itself to query optimization.* The author of a Garlic wrapper need not code to a standard query interface that may be too high-level or too low-level for the underlying data source. Instead, a wrapper is a full participant in query planning, and may use whatever knowledge it has about a repository's query and specialized search facilities to dynamically determine how much of a query the repository is capable of handling. This design allows us to build wrappers for simple data sources quickly, and still exploit the unique capabilities of unusual data sources such as image servers, text search engines, engines for molecular similarity search, etc.

4 Building a Garlic Wrapper

As shown in Figure 2, a wrapper provides four major services in the Garlic system. First, a wrapper models the contents of its repository as Garlic objects, and allows Garlic to retrieve references to these objects. Secondly, a wrapper allows Garlic to invoke methods on objects and retrieve their attributes. This mechanism is important, because it provides a means by which Garlic can get data out of a repository, even if the repository has almost no support for querying. Third, a wrapper participates in query planning when a Garlic query ranges over objects in its repository. The Garlic metadata does not include information about the query processing capabilities of individual repositories, so the Garlic query processor has no *a priori* knowledge about

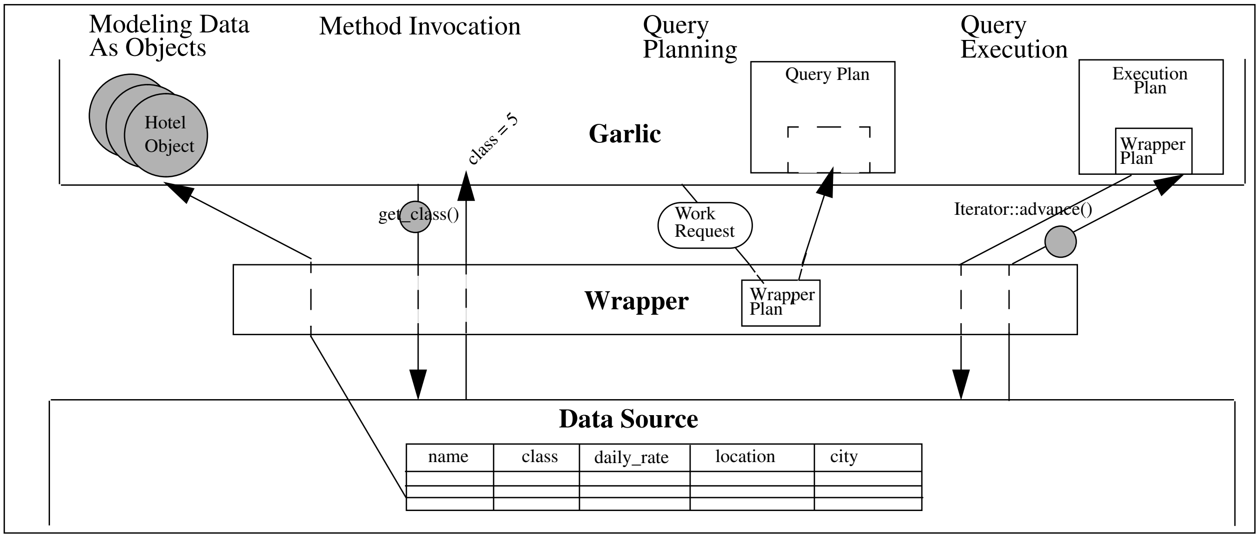

Figure 2. Services Provided by a Wrapper.

what predicates and projections can be handled by a given repository. Instead, the query processor identifies portions of a query relevant to a repository and allows the repository's wrapper to determine how much of the work it is willing to handle. The final service provided by a wrapper is query execution. During query execution, the wrapper completes the work it reported it could do in the query planning phase. A wrapper may take advantage of whatever specialized search facilities the repository provides in order to return the relevant data to Garlic.

In the sections that follow, we describe each of these services in greater detail, and provide an example of how to build wrappers for a simple travel agency application.

4.1 Modeling Data as Objects

The first service that a wrapper provides is to turn the data of the underlying repository into objects accessible by Garlic. Each Garlic object has an *interface* that abstractly describes the object's behavior, and an *implementation* that provides a concrete realization of the interface. The Garlic data model permits any number of implementations for a given interface. For example, two relational database repositories that contain information about disjoint sets of employees may each export distinct implementations of a common `Employee` interface.

During an initial registration step, wrappers provide a description of the content of their repositories using the Garlic Data Language, or GDL. GDL is a variant of the ODMG's Object Description Language (ODMG-ODL). The interfaces that describe the behavior of objects in a repository are known collectively as the *repository schema*. Repositories are registered as parts of a Garlic database and their individual repository schemas are merged into the *global schema* that is presented to Garlic users.

A wrapper also cooperates with Garlic in assigning identity to individual objects so that they can be referenced from Garlic and from Garlic applications. A Garlic object identifier (OID) has two parts. The first part, the *implementation identifier* (IID), is assigned by Garlic and identifies which implementation is responsible for the object, which in turn identifies the interface that the object supports and the repository in which it is stored. The second part of the OID, the *key*, is uninterpreted by Garlic. It is provided by the wrapper and identifies an object within a repository. Specific objects, usually collections, can be designated as *roots*. Root objects are identified by name, as well as by OID, and as such can serve as starting points for navigation or querying (e.g., root collection objects can be used in the `from` clause of a query).

As an example of how data is modeled as objects in Garlic, consider a simple application for a travel agency[1]. The agency stores information about the countries and cities for which it arranges tours as tables in a relational database. It also has access to a web site that provides booking information for hotels throughout the world, and to an image server in which it stores images of different travel destinations. These images can be retrieved and ordered according to features such as color, shape, texture, etc.

These sources are easily integrated as a Garlic database. The description of the `Country` and `City` interfaces that describe the relations in the relational database are shown in the left column of Figure 3. The attributes of each interface correspond to the columns of each relation, and the primary key value of a tuple serves as the key portion of the Garlic OID. Note that the `country` attribute on the `City` interface and the `scene` attributes on the `Country` and `City` interfaces are Garlic references to other Garlic objects. The relational wrapper registers `Cities` as a root collection of `City` objects, and `Countries` as a root collection of `Country` objects.

The web wrapper exports a single root collection of `Hotel` objects. The GDL for a `Hotel` object is shown at the

1. For brevity, we have omitted many of the implementation details of this application. See [22] for a more precise description.

Relational Repository Schema

```
interface Country {
  attribute string name;
  attribute string airlines_served;
  attribute boolean visa_required;
  attribute Image scene;}

interface City {
  attribute string name;
  attribute long population;
  attribute boolean airport;
  attribute Country country;
  attribute Image scene;}
```

Web Repository Schema

```
interface Hotel {
  attribute readonly string name;
  attribute readonly short class;
  attribute readonly double daily_rate;
  attribute readonly string location;
  attribute readonly string city;}
```

Image Server Repository Schema

```
interface Image {
  attribute readonly string file_name;
  double matches(in string file_name);
  void display(in string device_name);}
```

Figure 3. Travel Agency Application Schema.

top of the right hand column in Figure 3. The web site provides unique identifiers on the HTML page for hotel listings it returns, and these identifiers serve as the key portion of `Hotel` OIDs.

The interface for the image data stored in the image server is provided at the bottom of the right hand column of Figure 3. The image server repository exports 2 methods on the `Image` interface: `matches()`, which takes as input the name of a file containing the description of an image feature and returns as output a score that indicates how well an image matches the feature, and `display()`, which models the server's ability to output an image on a specified device. Image file names provide the key for `Image` OIDs.

4.2 Method Invocation

The second service a wrapper provides is a means to invoke methods on the objects in its repository. Method invocations can be generated by Garlic's query execution engine (see Section 4.3), or by a Garlic application that has obtained a reference to an object (either as the result of a query or by looking up a root object by name).

In addition to explicitly-defined methods like `matches()`, two types of *accessor* methods are implicitly defined for retrieving and updating an object's attributes — a "get" method for each attribute in the interface, and a "set" method for attributes that are not read-only. For instance, a `get_class()` method would be implicitly defined for the read-only `class` attribute of the `Hotel` interface.

Garlic uses the IID portion of a target object's OID to route a method invocation to the object's implementation. The implementation must be able to invoke each explicitly defined method in the corresponding interface, as well as the accessor methods. An implementation consists of wrapper code that maps Garlic method invocations into appropriate operations provided by the repository. To accommodate the widest possible range of repositories, Garlic provides two variants of method invocation: *stub* and *generic* dispatch.

A wrapper that utilizes stub dispatch provides a stub routine for each method of an implementation. Stub dispatch is a natural choice for repositories whose native programming interface is a class library, such as the image server in our travel agency example. For the `display()` method, for example, the image server wrapper provides a routine that first extracts the file name of the target image from the key field of the OID, and unpacks the device name from the argument list supplied by Garlic. To display the image on the screen, the routine calls the appropriate display function from the image server's class library, giving the image file name and display name as arguments.

Generic dispatch is useful for repositories that support a generic method invocation mechanism, or for repositories that do not directly support objects and methods. A wrapper that supports generic dispatch exports a single method invocation entry point. An important advantage of generic dispatch is that it is schema-independent. A single copy of the generic dispatch code can be shared by repositories that have a common programming interface but different schemas. The relational wrapper is an example of a wrapper that uses generic method dispatch. This wrapper supports only accessor methods, and each method invocation translates directly to a query over the relation that corresponds to the target object's implementation. The wrapper maps the method name into a column name, maps the IID portion of the object's OID into a relation name, extracts the primary key value from the OID, and uses these values to construct a query to send to the database.

4.3 Query Planning

A wrapper's third obligation is to participate in query planning. The goal of query planning is to develop alternative plans for answering a query, and then to choose the most efficient one. The Garlic query optimizer [8] is a cost-based optimizer modeled on Lohman's grammar-like rule approach [12]. STARs (STrategy Alternative Rules) are used in the optimizer to describe possible execution plans for a query. The optimizer uses dynamic programming to build query plans bottom-up. First, single collection access plans are generated, followed by a phase in which 2-way join plans are considered, followed by 3-way joins, etc., until a complete plan for the query has been chosen. Garlic extends the STAR approach by introducing wrappers as full-fledged participants during plan enumeration. During each query planning phase, the Garlic optimizer identifies the largest possible query fragment that involves a particular repository, and sends it to the repository's wrapper. The wrapper returns zero or more plans that implement some or all of the work represented by the query fragment. The optimizer incorporates each wrapper plan into the set of plans it is considering to produce the results of the entire query, adding operators to perform in Garlic any portion of the query fragment that the wrapper did not agree to handle.

As noted previously, repositories vary greatly in their query processing capabilities. Furthermore, each repository has its own unique set of restrictions on the operations it will perform. These capabilities and restrictions may be difficult or impossible to express declaratively. For example, relational databases often have limits on the number of tables involved in a join, the maximum length of a query string, the maximum value of a constant in a query, etc. These limits vary for different products, and even for different versions of the same product. As another example, our web wrapper is able to handle SQL `LIKE` predicates, but is sensitive to the placement of wild card characters. A key advantage to our approach is that the optimizer does not need to track the minute details of the capabilities and restrictions of the underlying data sources. Instead, the wrapper encapsulates this knowledge and ensures that the plans it produces can actually be executed by the repository.

Our approach allows a wrapper to model as little or as much of the repository's capabilities as makes sense. If a repository has limited query processing power, then the amount of code necessary to support the query planning interface is small. On the other hand, if a repository does have specialized search facilities and access methods that Garlic can exploit, the interface is flexible enough for a wrapper to encapsulate as much of these capabilities as possible. Even if a repository can do no more than return the OIDs of objects in a collection, Garlic can evaluate an arbitrary query by retrieving data from the repository via method invocation and processing it within Garlic.

A wrapper's participation in query planning is controlled by a set of methods that the optimizer may invoke during plan enumeration. The `plan_access()` method is used to generate single-collection access plans, and the `plan_join()` method is used to generate multi-way join plans. Joins may arise from queries expressed in standard SQL, or joins may be generated by Garlic for queries that contain path expressions, a feature of Garlic's extended SQL. The `plan_bind()` method is used to generate a specific kind of plan that can serve as the inner stream of a bind join (to be described in Section 4.3.3). Each of these methods takes as input a *work request*, which is a lightweight parse-tree description of the query fragment to be processed. The return value is a set of plans, each of which includes a list of *properties* that describe how much of the work request the plan implements, and at what cost. The plans are represented by instances of a wrapper-specific specialization of a `Wrapper_Plan` class. In addition to the property list, they encapsulate any repository-specific information a wrapper needs to actually perform the work described by the plan.

4.3.1 Single Collection Access Plans

The `plan_access()` method is the interface by which the Garlic query optimizer asks a wrapper for plans that return data from a single collection. It is invoked for each collection to which a Garlic query refers. The work request for a single-collection access includes predicates to apply, attributes to project, and methods to invoke. Since the Garlic optimizer does not know *a priori* which (if any) of the predicates a wrapper will be able to apply, the projection list in a work request contains *all* relevant attributes and methods mentioned in the query, including those that only appear in predicates. This gives the wrapper an opportunity to supply values for attributes that the Garlic execution engine will need in order to apply predicates that the wrapper chooses not to handle. As a worst-case fallback, the projection list also always includes the OID, even if the user's original query made no mention of it. The execution engine uses the OID and the method invocation interface to retrieve the values of any attributes it needs that are not directly supplied by the wrapper.

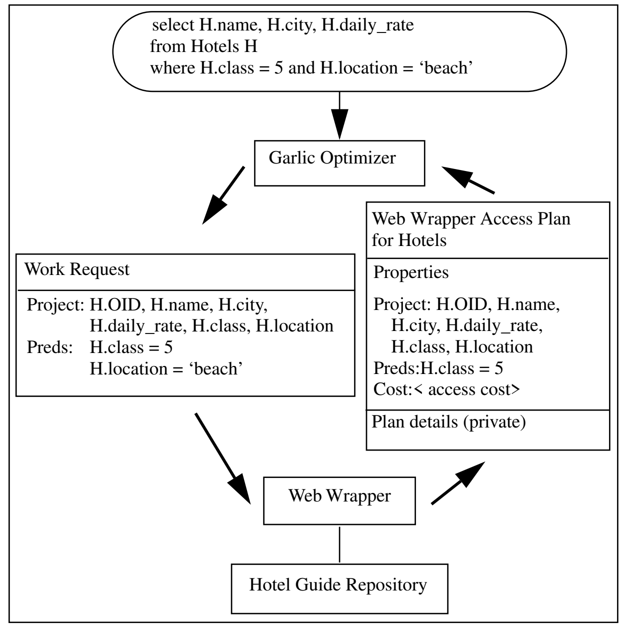

Figure 4. Construction of a Wrapper Access Plan.

Figure 4 shows the first phase of query planning for a simple single-collection query against our travel agency database. Suppose a Garlic user submits a query to find 5-star hotels with beach front property. The Garlic query optimizer analyzes the user's query and identifies the fragment that involves the `Hotels` collection. Since the `Hotels` collection is managed by the web wrapper, it invokes the web wrapper's `plan_access()` method with a description of the work to be done. This description contains the list of predicates to apply and attributes to project.

During the execution of `plan_access()`, the web wrapper looks at the work request to determine how much of the query it can handle. In general, our web wrapper can project any attribute and will accept predicates of the form `<attr> <op> <const>`, where `<op>` is either = or the SQL keyword `LIKE`. However, the web wrapper cannot handle equality predicates on strings because the web site does not adhere to SQL semantics for string equality. The web site treats the predicate "`location = 'beach'`" as "`location LIKE '%beach%'`", which provides a superset of the results of the equality predicate. This differ-

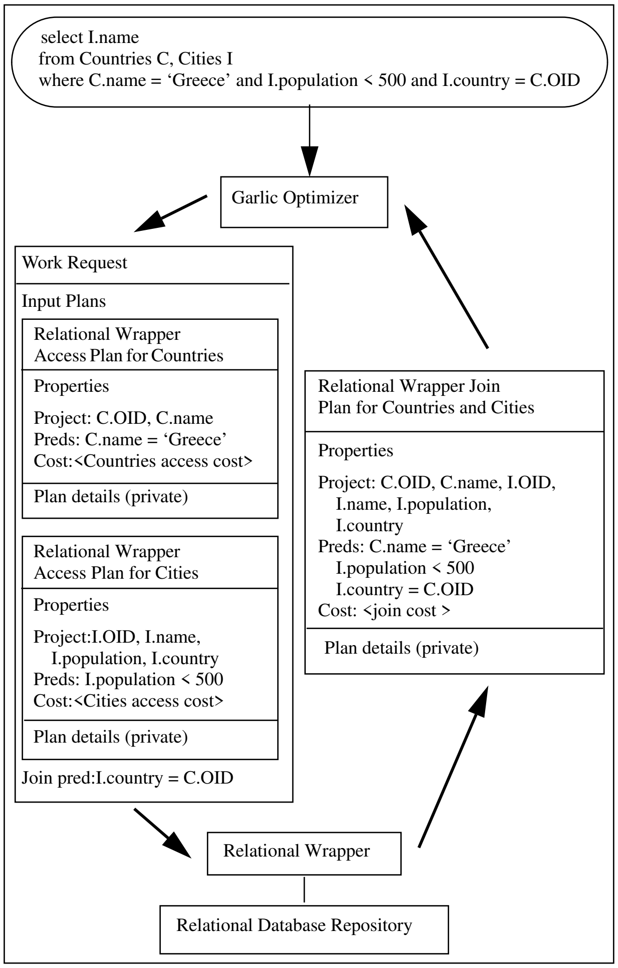

Figure 5. Construction of a Wrapper Join Plan.

ence in semantics means that the web wrapper cannot report to the optimizer that it can apply a string equality predicate. Nevertheless, when string equality is requested, it is still beneficial for the wrapper to apply the less restrictive `LIKE` predicate in order to reduce the amount of data returned to Garlic. The wrapper therefore creates a plan that will handle the entire projection list, perform the predicate on `class`, and the predicate "`location LIKE '%beach%'`", while reporting through the plan's properties that the `location` predicate will not be applied.The wrapper assigns the plan an estimated cost and returns it to the optimizer. If this access plan is chosen to be part of the global plan for the user's query, the optimizer will need to add the necessary operator to apply the predicate on `location`, although it would be applied to a far smaller set of objects than if the wrapper had not (covertly) applied the `LIKE` predicate.

4.3.2 Join Plans

The Garlic query optimizer uses the access plans generated in the first phase of optimization as a starting point for join enumeration. If the optimizer recognizes that two collections reside in the same repository, it invokes the wrapper's `plan_join()` method (if one is implemented) to try to push the join down to that repository. The work request includes the join predicates as well as the single-collection access plans that the wrapper had previously generated for the collections being joined. In the `plan_join()` method, the wrapper can re-examine these plans, and consider the effect of adding join predicates.

Let's return to our travel agency. Figure 5 shows how the relational wrapper provides a plan for a join between `Countries` and `Cities`. In the first phase of optimization (omitted from the picture), the optimizer requested and received access plans for `Cities` and `Countries` from the relational wrapper. During join enumeration, the optimizer invokes the relational wrapper's `plan_join()` method and passes in the join predicate as well as the two access plans previously created. The wrapper agrees to perform all of the work from its original access plans and to accept the join predicate, and creates a new plan for the join.The new plan's properties are made up of the properties from the input plans and the new join predicate.

During the next phase of join enumeration, the optimizer will follow a similar procedure for 3-ways joins of collections that reside in the same repository, and so on.

4.3.3 Bind Plans

During the join enumeration phase, the Garlic optimizer also considers a particular kind of join called a *bind join,* similar to the fetch-matches join methods of [14] and [13]. In a bind join, values produced by the outer node of the join are passed by Garlic to the inner node, and the inner node uses these values to evaluate some subset of the join predicates. A wrapper is well suited to serve as the inner node of a bind join if the programming interface of its repository provides some mechanism for posing parameterized queries. As an example, ODBC and the call level interfaces of most relational database systems contain such support.

Suppose our travel agency user is really interested in finding 5-star hotels on beaches in small towns in Greece. This query involves the `Countries` and `Cities` collections managed by the relational wrapper, and the `Hotels` collection managed by the web wrapper. The web wrapper does not support the `plan_bind()` method, but the relational wrapper does. Figure 6 shows how a bind plan for this query is created. During the first phase of optimization, the optimizer would have requested and received an access plan from the web wrapper for the `Hotels` collection as described in Section 4.3.1. It would also have requested and received access plans for the `Countries` and `Cities` collections from the relational wrapper. While considering 2-way joins, the optimizer would have received a join plan for `Countries` and `Cities` from the relational wrapper, as described in the previous section.

Next, the optimizer develops a plan to join all three collections. The optimizer recognizes that a bind join is possible, with the web wrapper's access plan as the outer stream

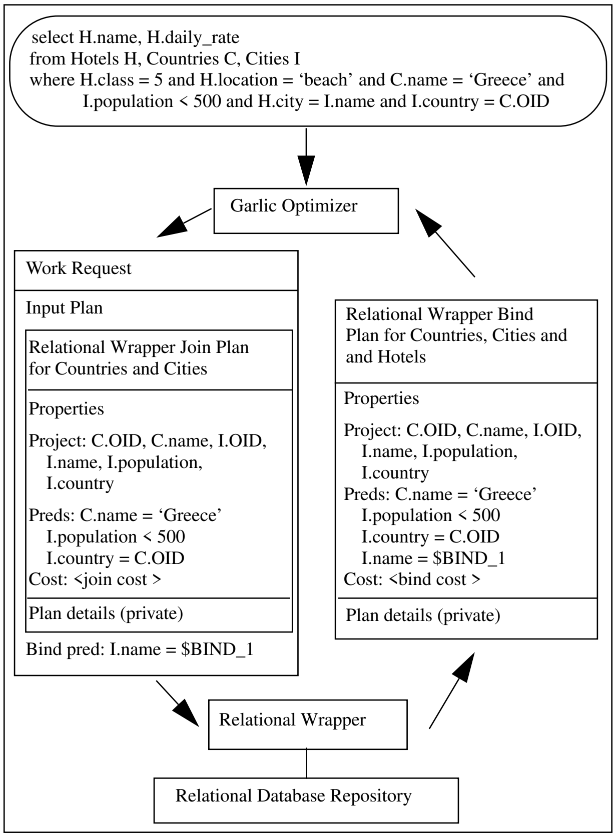

Figure 6. Construction of a Wrapper Bind Plan.

and the join plan provided by the relational wrapper as the inner stream. The optimizer invokes the relational wrapper's `plan_bind()` method, passing in a work request that consists of the join plan for `Countries` and `Cities` that the wrapper previously provided and the description of the bind join predicate between `Cities` and `Hotels`. The relational wrapper creates a new plan that handles the work of the original join plan plus the bind predicate. It uses the input plan's properties to fill in the new bind plan properties, and adds in the bind predicate.

4.4 Query Execution

A wrapper's final service is to participate in plan translation and query execution. A Garlic query plan is represented as a tree of operators, such as FILTER, PROJECT, JOIN, etc. Wrapper plans show up as the operators at the leaves of the plan tree. Figure 7 shows an example of a complete Garlic plan based on the bind join plan for the query discussed in Section 4.3.3. The outer node of the bind join is the web wrapper's access plan from Section 4.3.1, and the inner node is the relational wrapper's bind plan described in Section 4.3.3. The Garlic optimizer added a FILTER operator to handle the predicate on `location` and a PROJECT operator to project `name` and `daily_rate`.

The optimized plan must be translated into a form suitable for execution. As is common in demand-driven runtime systems [7], operators are mapped into iterators, and each wrapper provides a specialized `Iterator` subclass

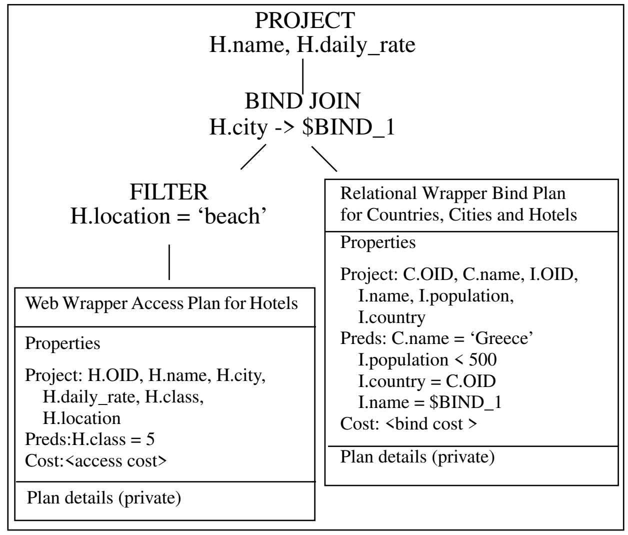

Figure 7. A Plan for a Garlic Query.

that controls execution of the work described by one of its plans. The wrapper must also supply an implementation of `Wrapper_Plan::translate()`, to translate a wrapper's plan into an instance of the wrapper's `Iterator` subclass. Translation involves converting the information stored in the plan into a form that can be sent to the repository. For example, our relational wrapper stores the elements of the `select`, `from` and `where` clauses of the query to be sent to the relational database in the private section of its plan. At plan translation time, the wrapper extracts these elements, constructs the query string, and stores it in an instance of its `Iterator` subclass. As another example, our web wrapper stores the list of attributes to project and the set of predicates to apply in the private data section of its plan. At plan translation time, the predicates are used to form a query URL that the web site will accept.

The Garlic execution engine is pipelined, and employs a fixed set of methods on iterators at runtime to control the execution of a query. Default implementations for most of the methods exist, but for each operator, two methods in particular define the unique behavior of its iterator: `advance()` and `reset()`. The `advance()` method completes the work necessary to produce the next output value, and the `reset()` method resets an iterator so that it may be executed again. An additional `bind()` method is unique to wrapper iterators, and provides the mechanism by which Garlic can transfer the next set of bindings to the inner node of a bind join.

Our relational wrapper uses standard ODBC calls to implement `reset()`, `advance()` and `bind()`. `reset()` prepares a query at the underlying database, and `bind()` binds the parameters sent by Garlic to the unbound values in the query string. The `advance()` method fetches the next set of tuples from the database.

The web wrapper's `Iterator` subclass is very simple. The `reset()` method loads the HTML page that corresponds to the query URL generated at plan translation time. In the `advance()` method, the wrapper parses the HTML page to extract the query results. Each HTML page pro-

TABLE 1. A Description of Existing Wrappers.

DB2, Oracle	Schema description: Columns of a relation map to attributes of an interface; relations become collections of objects; primary key value of a tuple is key for OID. Method invocation: accessor methods only, generic dispatch. Query operations: general expression projections, all basic predicates, joins, bind joins, joins based on OID.
Searchable web sites: http://www.hotelguide.ch, a hotel guide, and http://www.bigbook.com, U.S. business listings	Schema description: Each web site exports a single collection of listing objects; HTML page data fields map to attributes of an interface; unique key for a listing provided by web site is key for OID. Method invocation: accessor methods only; generic dispatch. Query operations: attribute projection, equality predicates on attributes, LIKE predicates of the form '%<value>%.
Proprietary database for molecular similarity search	Schema description: A single collection of molecule objects; interface has contains_substructure() and similarity_to() methods to model search capability of engine; molecule l-number is key for OID. Method invocation: stub dispatch. Query operations: attribute and method projection; predicates of the form <attr> <op> <const> and <method> <op> <const>, if <op> is a comparison operator; bind plans if similarity_to() is in bind predicate.
QBIC [16] image server that orders images according to color, texture and shape features	Schema description: Collections of image objects; interface has matches() method to model ordering capability; image file name is key for OID. Method invocation: stub dispatch. Query operations: ordering of image objects by image feature.
Glimpse [15] text search engine that searches for specific patterns in text files	Schema description: Collections of files; interface contains several methods to model text search capability and retrieve relevant text of a file; file name is key for OID. Method invocation: stub dispatch. Query operations: projection of attributes and methods.
Lotus Notes databases: Phone Directory database, Patent Server database	Schema description: Notes database becomes a collection of note objects; interface defined by database Form; note NOTEID is key for Garlic OID. Method invocation: accessor methods only, generic dispatch. Query operations: attribute projection; predicates with logical, comparison and arithmetic operations; LIKE predicates.
Complex Object Wrapper	Schema description: Collections of objects; interface corresponds to interface of objects in database; database OID is key for Garlic OID. Method invocation: stub dispatch. Query operations: attribute projection.

vides a link to the next page of results, so after all of the results on one page are returned to Garlic, the wrapper follows the link and retrieves the next page.

4.5 Wrapper Packaging

In the previous sections, we have described the services that a wrapper provides to the Garlic middleware. The wrapper author's final task is to package these pieces as a complete wrapper. A wrapper may include three kinds of components: *interface files* that contain one or more interface definitions written in GDL, *environment files* that contain name/value pairs to encode repository-specific information for use by the wrapper, and *libraries* that contain dynamically loadable code to implement schema registration, method invocation, and the query interfaces. Libraries are further subdivided as follows: *core* libraries that contain common code shared among several similar repositories, and *implementation* libraries that contain repository-specific implementations of one or more interfaces.

Packaging wrapper code as dynamically loadable libraries that reside in the same address space as Garlic keeps the cost of communicating with a wrapper as low as possible. This is important during query processing, since a given wrapper may be consulted several times during the optimization of a query, and non-trivial data structures are exchanged at each interaction. Very simple repositories can be accessed without crossing address space boundaries, and repositories that are divided into client and server components are easily accommodated by linking their wrapper with the repository's client-side library. This approach encapsulates the choice of a particular client-server protocol (e.g., CORBA-IIOP, ActiveX/DCOM, or ODBC) within the wrapper, allowing Garlic to integrate repositories regardless of the particular protocol(s) they support.

Decomposing wrappers into interface files, libraries, and environment files gives the designer of a wrapper for a particular repository or family of repositories considerable flexibility. For example, our relational wrapper packages generic method dispatch, query planning and query execution code as a sharable core library. For each repository, an interface file describes the objects in the corresponding database. An environment file encodes the name of the database to which the wrapper must connect, the names of the collections exported by the repository and the tables to which they are bound, the correspondence between attributes in interfaces and columns in tables, etc.

Implementation libraries are useful when a wrapper that employs stub dispatch is built for a data source whose schema can evolve over time. As new kinds of objects are added to the repository schema, implementation libraries can be registered with stubs for the new implementations.

5 Current Status

To test the flexibility of our architecture, we have implemented wrappers for a diverse set of 10 data sources. Table 1 describes some of the features of these wrappers. The data models for these sources vary widely, including relational, object-oriented, a simple file system, and a specialized molecular search data model. Likewise, the data sources provide query processing power that ranges from simple scanning to basic predicate application to complex join processing. Wrappers such as the relational wrapper have been fine tuned and are fairly mature. Others, such as the molecular wrapper, are still in a state of evolution.

Based on our experience writing these wrappers, we have identified 3 general categories of wrappers, and provide a base class for each category. We also provide wrapper writers with a library of schema registration tools, query plan construction routines, and other useful routines in order to automate the task of writing a wrapper as much as possible. To test our assertion that wrappers are easy to write, we asked developers outside of the project to write several wrappers listed in the table. For example, a summer student wrote the text and image server wrappers over a period of a few weeks, and a chemist was able to write the molecular database wrapper during a 2-day visit to our lab.

6 Related Work

Presenting a uniform interface to a diverse set of information sources has been the goal of a great deal of previous research, dating back to projects like CCA's Multibase [20]. Surveys of much of this work can be found in [3] [6] [10] [19], and [1] [21] describe actual implementations. In terms of query processing, the architectures of these earlier systems are built around a *lingua franca* for communicating with the underlying sources. These systems assume that any data source, assisted by the translator, can readily execute any query fragment.

OLE DB [2] takes an important step towards integrating heterogeneous data sources by defining a standardized construct, the *rowset*, to represent streams of values obtained from a data source. A simple tabular data source with no querying capability can easily expose its data as a rowset. More powerful data sources can accept commands (either as text or as a data structure) that specify query processing operations peculiar to that data source, and produce rowsets as a result. Thus, although OLE DB does not include a middleware query processing component like Garlic, it does define a protocol by which a middleware component and data sources can interact. This protocol differs from the Garlic wrapper interface in several ways. First, the format of an OLE DB command is defined entirely by the data source which accepts it, whereas Garlic query fragments are expressed in a standard form based on object-extended SQL. Secondly, an OLE DB data source must either accept or reject a command in its entirety, whereas a Garlic wrapper can agree to perform part of a work request and leave any parts it cannot handle to be performed by Garlic.

A different set of techniques for integrating data sources with various levels of query support relies upon an *a priori* declarative specification of query capability for each data source. In the TSIMMIS system [18], specifications of query power are expressed in the Query Description and Translation Language (QDTL) [17]. A QDTL specification for a data source is a context-free grammar for generating supported queries. DISCO [9] builds on the notion of capability records described in the Information Manifold [11] and requires a wrapper writer to describe a data source's capabilities by means of a language based on a set of (relational) logical operators such as `select`, `project`, and `scan`.

The idea of compact declarative specifications of query power is attractive, but there are some practical problems with this approach. First, it is often the case that a data source cannot process a particular query, but can process a *subsuming* query whose answer set includes the answer set of the original query. In general, finding maximal subsuming queries is computationally costly, and choosing the optimal subsuming query may require detailed knowledge of the contents, semantics, and statistics of the repository.

Secondly, in defining a common language to describe all possible repository capabilities, it is difficult to capture the unique restrictions associated with any individual repository. For example, as we noted earlier, relational database systems often place limits on the query string length, the maximum constant value that can appear in a query, etc. Likewise, our web wrapper can handle LIKE predicates, but only if the pattern is of a specific form. The molecular wrapper is sensitive to which attributes and methods appear together in the projection and predicate lists. A language to express these and other repository-specific restrictions would quickly become very cumbersome. Furthermore, in a strictly declarative approach such as DISCO, as new sources are integrated, the language would need to be extended to handle any unanticipated restrictions or capabilities introduced by the new sources.

As we saw in Section 4.3, Garlic forgoes the declarative approach for one in which the knowledge about what a specific repository can and cannot do is encapsulated by the wrapper. Rather than solve the query subsumption problem in general at the Garlic level, we ask wrapper authors to solve the simpler special-case problem for their own repositories. Decisions about how much of a query can be handled by a repository are made by the wrapper at query planning time, taking advantage of repository-specific semantic knowledge. Since our approach is not limited by the expressive power of a query specification language, we can accommodate the idiosyncrasies of almost any data source.

7 Conclusions

In this paper, we have described the wrapper architecture for Garlic, a middleware system designed to provide a unified view of heterogeneous, legacy data sources. Our architecture is flexible enough to accommodate almost any kind of data source. We have developed wrappers for sources that represent a broad spectrum of data models and query capabilities. For sources with specialized query processing capabilities, representing those capabilities as methods has proven to be viable and convenient.

The Garlic wrapper architecture makes the wrapper writer's job relatively simple, and as a result, we have been able to produce wrappers for new data sources in a matter of days or hours instead of weeks or months. Wrapper authoring is especially simple for repositories with limited query power, but even for more powerful repositories, a basic wrapper can be written very quickly. This allows applications to access data from new sources as soon as possible, while subsequent enhancements to the wrapper can transparently improve performance by taking greater advantage of the repository's query capabilities.

Our design also allows the Garlic query optimizer to develop efficient query execution strategies. Our approach does not require a complex language to describe the minute details of the capabilities and restrictions of the underlying data sources. Furthermore, we do not require a wrapper to raise a repository's query processing capabilities to a fixed level, or "dumb down" the query processing interface to the lowest common denominator. Instead, our architecture allows each wrapper to determine on a case-by-case basis how much of a query its repository is capable of handling.

In the future, we will continue to refine the wrapper interfaces. An open research question is to develop a truly satisfactory cost model for a diverse set of data sources. We intend to focus on making the wrapper's job of providing a cost model easier, by providing a basic framework that a wrapper writer can customize for a specific repository. We will also investigate the possibility of introducing QDTL-style templates to allow a wrapper to declare up-front a specification of the expressions it will support. With such information, the Garlic query processor could filter out expressions that a wrapper is unable to handle before the work request is generated. Such a template would be a step toward a hybrid system, combining Garlic's dynamic approach to query planning with the declarative approach of TSIMMIS and DISCO; striking an appropriate balance between the techniques is an interesting research opportunity.

Acknowledgements

We would like to thank the Garlic team members, both past and present, whose hard work and technical contributions made the Garlic project possible. In particular, Laura Haas spent many hours with us working out the details of the query processing interface. We'd also like to thank Mike Carey and Laura Haas for reviewing an initial draft of this paper and providing us with excellent suggestions that improved its presentation and readability.

References

[1] R. Ahmed, et. al., "The Pegasus Heterogeneous Multidatabase System", *IEEE Computer,* 24(12) pp. 19-27, December 1991.

[2] J. Blakely, "Data Access for the Masses Through OLE DB", *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, PQ, Canada, June 1996.

[3] O. Bukhres, and A. Elmagarmid, eds., Object-Oriented Multidatabase Systems, Prentice Hall, publishers, New Jersey, 1996.

[4] M. Carey, et al., "Towards Heterogeneous Multimedia Information Systems: The Garlic Approach", *Proc. IEEE RIDE-DOM*, Taipei, Taiwan, March 1995.

[5] R. Cattell, ed., Object Database Standard: ODMG-93 (Release 1.2), Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[6] A. Elmagarmid and C. Pu, eds., "Special Issues on Heterogeneous Databases", *ACM Comp. Surveys 22(3),* September 1990.

[7] G. Graefe, "Query Evaluation Techniques for Large Data Bases", *ACM Computing Surveys 25(2)*, June 1993.

[8] L. Haas, et. al., "Optimizing Queries Across Diverse Data Sources", *Proc of the 23rd International Conference on Very Large Data Bases,* Athens, Greece, August 1997.

[9] Kapitskaia, O., et. al., "Dealing with Discrepancies in Wrapper Functionality", *INRIA Technical Report RR-3138*, 1997.

[10] W.Kim, Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press, Addison-Wesley Publishers, 1995.

[11] A. Levy, et. al., "Querying Heterogeneous Information Sources Using Source Descriptions", *Proc of the 22nd International Conference on Very Large Data Bases,* Bombay, India, September 1996.

[12] G. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives", *Proc. of the ACM SIGMOD Conference on Management of Data,* Chicago, IL, USA, May 1988.

[13] H. Lu and M. Carey, "Some Experimental Results on Distributed Join Algorithms in a Local Network", *Proc. 11th International Conference on Very Large Data Bases*, Stockholm, Sweden, August 1985.

[14] L. Mackert and G. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries", in Readings in Database Systems, M. Stonebraker, ed., Morgan-Kaufmann Publishers, San Mateo, CA, 1988.

[15] U. Manber, et. al., http://glimpse.cs.arizona.edu/

[16] W. Niblack, et al., "The QBIC Project: Querying Images By Content Using Color, Texture and Shape", *Proc. SPIE*, San Jose, CA, February1993.

[17] Y. Papakonstantinou, et. al., "A Query Translation Scheme for Rapid Implementation of Wrappers", *Proc. of the Conference on Deductive and Object-Oriented Databases (DOOD)*, 1995.

[18] Y. Papakonstantinou, et. al., "Object Exchange Across Heterogeneous Information Sources", *Data Engineering Conf.,* March 1995.

[19] S. Ram, guest ed., *IEEE Computer Special Issue on Heterogeneous Distributed Database Systems*, 24(12), December 1991.

[20] R. Rosenberg and T. Landers, "An Overview of MULTIBASE", in Distributed Databases, H. Schneider, ed. North-Holland Publishers, New York, NY, 1982.

[21] R. Stout, "EDA/SQL", in Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, ed., pp 649-663, ACM Press, Addison-Wesley Publishers, 1995.

[22] M. Tork Roth, and P. Schwarz, "A Wrapper Architecture for Legacy Data Sources", *IBM Technical Report RJ10077*, 1997, http://www.almaden.ibm.com/cs/garlic/.