A Wrapper Architecture for Legacy Data Sources[1]

Mary Tork Roth[2]
Peter Schwarz[2]

IBM Almaden Research Center

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. Even within a single organization, these legacy data management systems typically vary widely, ranging from simple text files with little or no support for queries to complex database management systems with sophisticated query engines. In many cases, specialized indexing technology and query engines have been developed to facilitate efficient searching of particular kinds of data, e.g., for finding compounds with a certain substructure in chemical databases, for finding overlapping regions in a geographic database, or for finding images with similar color or texture in an image archive. 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. Likewise, a pharmaceutical application that can combine chemical compound similarity search with the results of biological assays would be a powerful tool for discovering new drugs.

There are several approaches to providing such an integrated view of heterogeneous data. One is to move the data *en masse* to a new, integrated database system that is tailored to provide a unified view of data of different types. For example, object-relational systems can store image, video and text, and provide some query support for these non-traditional data types [26]. Many relational database vendors have extended their systems to do the same. The drawback of the "universal server" approach is that the data is often already adequately managed by the legacy systems, and existing applications written against those systems' interfaces must be rewritten to work with the new database. Thus, migration to an entirely new system is often not a practical solution.

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

2.torkroth@almaden.ibm.com, schwarz@almaden.ibm.com.

An alternative approach is middleware 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 the legacy systems, providing a unified schema and common interface for advanced new applications without disturbing existing applications. Freed from the responsibilities of storage and data management, middleware systems focus on providing powerful high-level query services for heterogeneous data.

Middleware systems typically rely on *wrappers* [28] [22] 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 wide variance in repository query capability means that a wrapper interface that required a standard level of query support from each repository would be impractical. If the chosen level were low, the middleware system would be unable to take full advantage of the native query power of sophisticated repositories, resulting in performance far below what it could be. If the chosen level were high, the complexity of producing a wrapper for a simple data source would be unacceptable. The wrapper architecture of Garlic [5], an object-oriented middleware system, 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 negotiate 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 in detail how a wrapper is built, and Section 5 discusses the current status of our system. Section 6 briefly summarizes related work. 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 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 [6]. Object-oriented data models are a good choice for middleware, because they are general enough to support complex nested structures, allow relationships among entities to be modeled as references, and allow methods to be associated with data. The latter is of particular importance to Garlic, since it provides a convenient and natural way to model the specialized search and data manipulation facilities of non-traditional data sources. For example, the ability of a text server to rank documents by relevance to some subject is easily modeled as a method that takes the subject as a parameter and returns a numeric score for the document. 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*

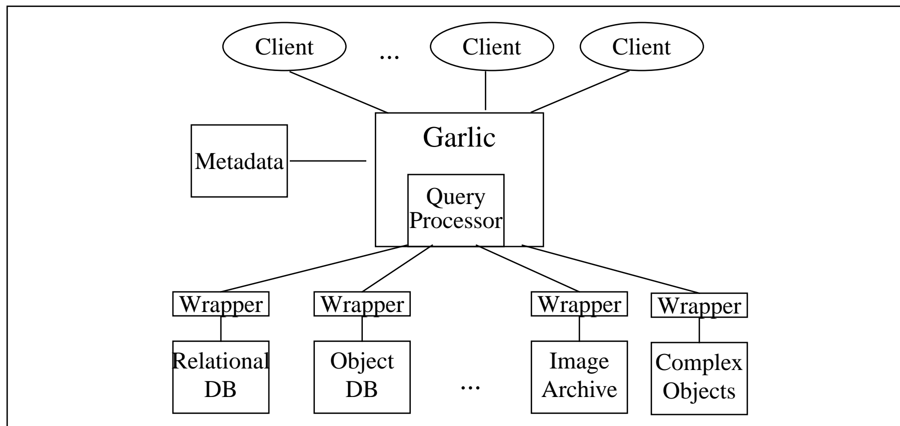

Figure 1. The Garlic Architecture.

objects, which users can create to bind together existing objects from the other 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 chemical structures database. Although Garlic is intended to ship with a set of 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 for a new repository 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 an initial 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. We have successfully built wrappers for a diverse set of data sources. These include two relational database systems

(DB2 and Oracle), a patent server stored in Lotus Notes, searchable sites on the World Wide Web (including a database of business listings and a hotel guide), and specialized search engines for collections of images, chemical structures and text. In addition, we implemented the repository for complex objects by writing a wrapper for an object-oriented database system.

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 capabilities and specialized search facilities to dynamically determine how much of a query the repository is capable of handling. This approach allows us to build wrappers for simple data sources quickly, and still exploit the unique query processing capabilities of unusual data sources such as search engines for chemical structures and images.

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 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 re-

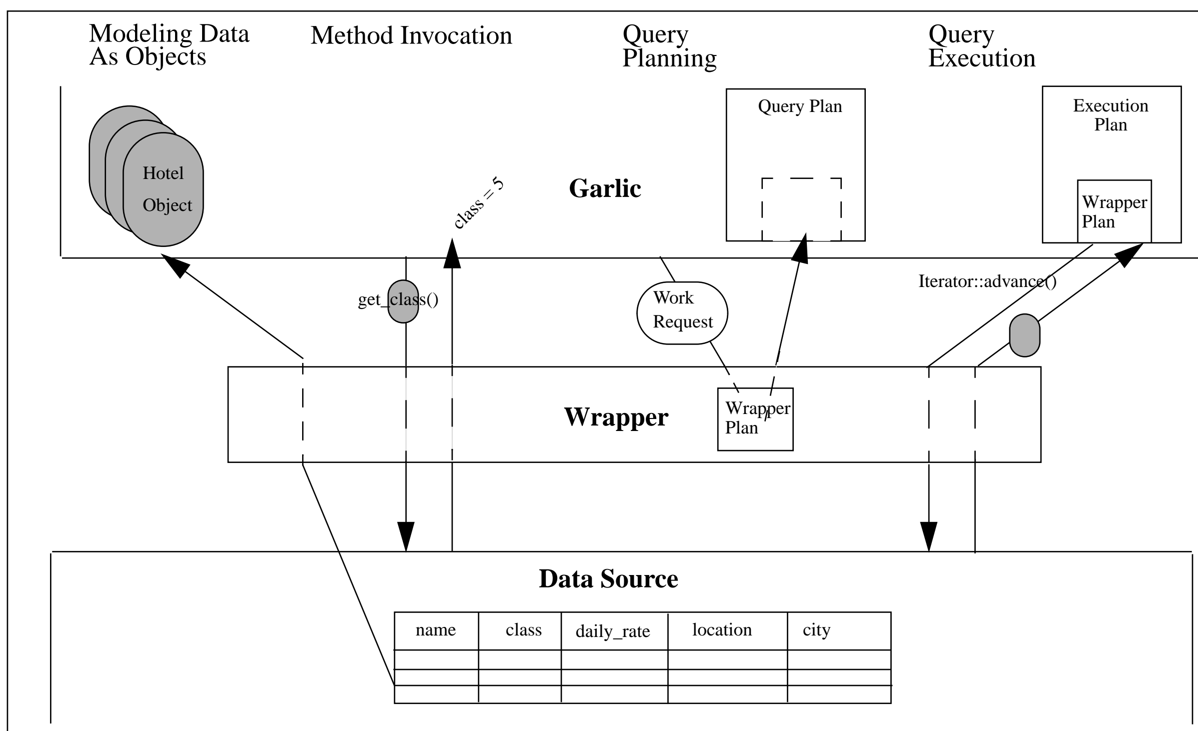

Figure 2. Services Provided by a Wrapper.

turn 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 is the *implementation identifier* (IID). It 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. It is the combined responsibility of the wrapper and repository to generate an OID when a reference to an object is needed, to ensure that no two objects with the same implementation have the same key, and to ensure that the combination of IID and key is sufficient to locate the object's data in the 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 hypothetical travel agency. 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 sources are easily described in GDL and integrated as a Garlic database. First, consider the relational database. The database contains two relations, `Countries` and `Cities`. The `Countries` relation contains general information about the countries that the travel agency serves, and its primary key is the country name. The `Cities` relation contains information about cities that can serve as travel destinations, including a column named `country` that contains the country name. The country name and the city name together form the primary key for the `Cities` relation, and the country name also serves as a foreign key to the `Countries` relation. Both relations have a `scene` column that stores the name of an image file containing a representative scene of the location.

The wrapper for the relational database exports two interfaces, one for the `Countries` relation and one for the `Cities` relation. Their description in GDL is shown in the left column of Figure 3. The attributes of each interface correspond to the columns of each relation. Note that the wrapper exposes the foreign key `country` on the `Cities` relation as a typed Garlic object reference, rather than as a string, which is its internal representation in the relational database. Similarly, the `scene` attribute on each relation is exposed as a Garlic reference to an instance of the `Image` interface, which will be described below. For each relation, the wrapper also exports an implementation of the corresponding interface. Each implementation maps

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.

the tuples of a relation into Garlic objects. The primary key value of a tuple serves as the key portion of the Garlic OID. Lastly, the wrapper exports a collection corresponding to each relation. `Cities` is registered as a root collection of `City` objects and `Countries` as a root collection of `Country` objects.

Next, consider the web site, which provides information about daily rates, class ratings, location, etc., for hotels throughout the world. Garlic's web wrapper models this source as a repository that exports a single interface, `Hotel`, and a single root collection of `Hotel` objects. The GDL for the `Hotel` interface is also shown in Figure 3[3]. Since the web site does not support updates, the attributes are marked read-only. The web site provides unique identifiers on the HTML page for the hotel listings it returns, and these identifiers serve as the key portion of `Hotel` OIDs.

Lastly, consider the image server. We have built a wrapper for an image server based on the QBIC image search engine [20]. It manages collections of images stored in files, and allows these images to be retrieved and ordered according to features such as color, shape, color position within an image, etc. As shown in Figure 3, the wrapper for the image server exports an interface definition for `Image` objects. Each object has a single read-only attribute, the name of the file in which the image is stored. The image file name also serves as the key portion of an `Image` OID. The image server's search capability is expressed in GDL as a `matches()` method, 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 a particular image matches the feature. An additional method, `display()`, models the server's ability to output an image on a specified device. The image server wrapper exports two collections of `Image` objects—one that contains scenes of countries, and one that contains scenes of cities. The `scene` attributes of the relational wrapper's `Country` and `City` interfaces are references to objects in these collections.

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). For example, consider the C++ application code fragment in Figure 4. The application uses the

3.Note that we have chosen to model the `city` attribute of `Hotel` objects as a string attribute, not as a reference to an object that supports the `City` interface. This choice seems appropriate for this example, because the web site is outside the travel agency's control and has information about cities around the world, whereas the relational database containing `City` tuples was developed by the travel agency and is constrained to cities that the travel agency serves. If necessary, Garlic complex objects or views could be used to link hotels with cities, or cities with sets of hotels. See [5] for further discussion of Garlic complex objects and views.

```
Iterator<GStruct> *result_iter;
GStruct result_tuple;
...
oql(db, result_iter,
   "select C.scene from Countries C where C.airlines_served LIKE '%American%'");
while (result_iter->next(result_tuple)) {
  Image *i = result_tuple[0].get_reference();
  i->display(myScreen);
}
```

Figure 4. A Garlic Application Fragment.

`oql()` function to submit a query that retrieves the `scene` attribute for countries in the travel agency database that are served by American Airlines. The value of the `scene` attribute is a reference to an `Image` object, and for each `Image` reference retrieved, an invocation of the `display()` method is used to render the image on the user's screen.

In addition to explicitly-defined methods like `display()`, 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_file_name()` method would be implicitly defined for the read-only `file_name` attribute of the `Image` 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 method in the corresponding interface, as well as the implicitly-defined accessor methods. Unlike interfaces, which can be described by GDL, an implementation typically consists of code that maps Garlic method invocations into appropriate operations provided by the underlying repository. To accommodate the widest possible range of repositories, Garlic provides two variants of the method invocation interface. The first variant, *stub dispatch*, is a natural choice for repositories whose native programming interface is a class library. A wrapper that utilizes stub dispatch provides a stub routine for each method of an implementation. The stub routine converts method arguments from a standard form specified by Garlic into the form expected by the repository, invokes the corresponding method from the repository's class library, and performs a similar translation on the method's return value. When a wrapper utilizing stub dispatch is initialized, it exports a table that contains the entry point addresses of these stub routines.

The image server is an example of a repository for which stub dispatch is appropriate. For the `display()` method, for example, the image server wrapper provides a short routine that first extracts the file name of the target image from the key field of the OID, and unpacks the display 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, passing in the image file name and display name as arguments. (If the display function had a return value, upon return the stub routine would package the value into the standard form expected by Garlic.) Other repositories for which stub dispatch is appropriate include those that store data as persistent programming language objects, e.g., those implemented using object-oriented databases like ObjectStore [14] or O_2 [7].

The second variant of the method invocation interface, *generic dispatch*, is useful for repositories that themselves 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. The method name is given as a parameter, and its arguments are supplied in a standard form as described above. 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 well-suited for generic method dispatch. This wrapper supports only the implicitly-defined accessor methods, and each method invocation translates directly to a query over the relation that

corresponds to the implementation. For example, consider an invocation of the `get_population()` method on a `City` object. Garlic sends the name of the method and the OID of the object to the relational wrapper. The wrapper maps the method name into a column name, maps the IID portion of the object's OID into a relation name, and extracts the primary key value from the OID. It uses these values to construct the following query:

```
select population
from Cities
where name = <OID key value for name> and country = <OID key value for country>
```

Our web wrapper also uses generic dispatch. Since the attributes of its `Hotel` objects are read-only, the web wrapper supports only "get" methods. A method invocation is translated into a search URL for the web site. The wrapper extracts the hotel listing key from the object's OID and forms the URL to retrieve the HTML page. Once the page is loaded, it parses the retrieved HTML page to find the hotel listing that corresponds to the `Hotel` object, and returns the required attribute to Garlic.

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 [11] is a cost-based optimizer modeled on Lohman's grammar-like rule approach [16]. 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 generated. 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 relational 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 exact 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 SQL 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 method takes as input a *work request*, which is a 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 the `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 properties list.

A wrapper author only needs to provide implementations of the planning methods that make sense for a given repository. For example, f a repository is unable to process joins, the wrapper does not need to implement the `plan_join()` method. A default implementation supplied by Garlic will return an empty set of plans, and as a result joins over collections in that repository will be handled by the Garlic execution engine. Note, however, that if a wrapper does not provide at least a minimal `plan_access()` method, there is no way for the Garlic execution engine to iterate over the objects in the repository's collections. As a result, objects in such a repository can be accessed in a Garlic query only if references to them can be found in other repositories. The execution engine will have to use method invocation to retrieve the objects' attributes.

The data structure that describes the work request is designed to be as lightweight as possible. Wrappers can use a simple request/response protocol to quickly obtain the pieces of the work request that they care about. Predicates and projection expressions are described by simple parse trees, and a class library is provided for common operations such as moving projection list elements and predicates from the work request to the plan and the properties list. To help wrappers quickly filter out complicated expressions that they can't handle, methods are provided to help analyze expressions at a high level. For example, a wrapper may invoke the methods `refers_to_method()` and `refers_to_OID()` to determine whether or not an expression refers to a method or an object id without actually having to traverse the entire expression tree.

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 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.

It is completely up to the wrapper to decide how much of the request it can handle. The decision depends on the general query processing capabilities of the repository, on the amount of effort that has gone into writing the wrapper, and on any limitations of the data source's programmatic interface (e.g., limits on query string length or the number of predicates that can be applied, etc.). If necessary, the wrapper may consult with the repository about any restrictions specific to individual queries. Note, however, that during planning a process boundary will never be crossed unless the wrapper chooses to consult with the underlying data source in this manner, and in any event none of the data referenced by the query will be transferred during planning.

A wrapper for a simple repository that only supports iteration over collections of objects, or an initial implementation of a wrapper for a more sophisticated repository, need not apply any predicates or projections.

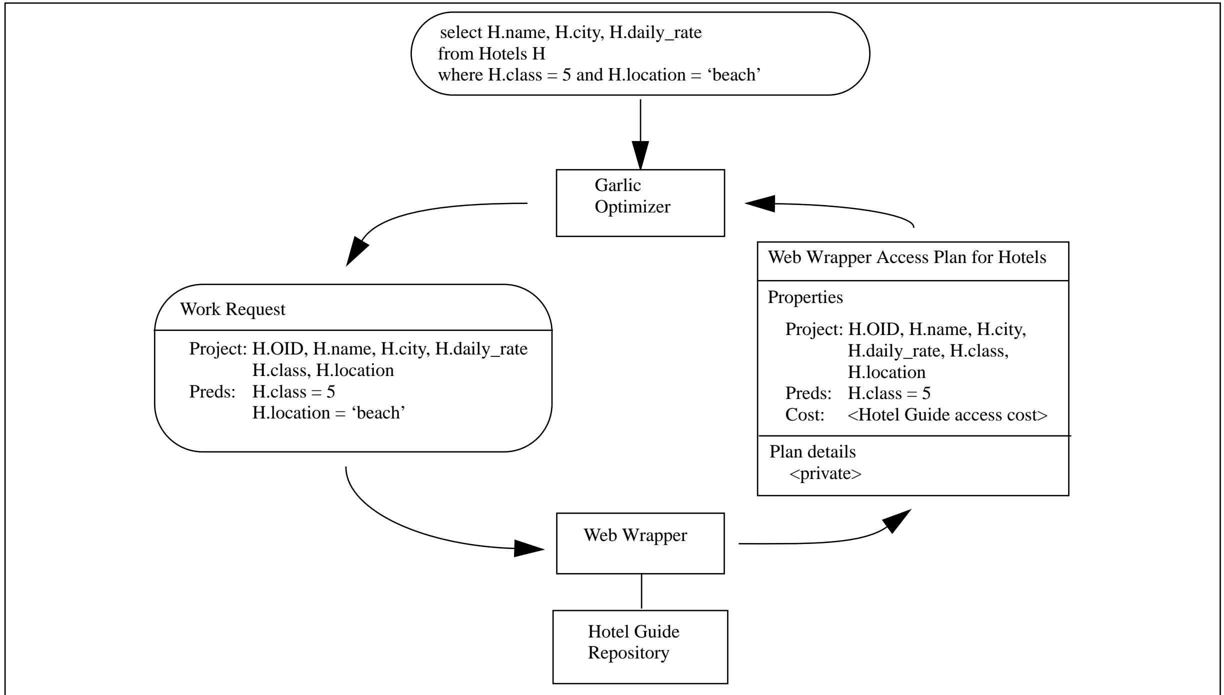

Figure 5. Construction of a Wrapper Access Plan.

If the wrapper just returns a plan for the query “`select OID from <collection>`”, the optimizer will append to the wrapper’s plan the operators necessary for the Garlic execution engine to complete the work request. The wrapper for a more sophisticated repository, such as a relational database, may perform more detailed analysis and consume as many predicates and projections as it can. However, even a relational database may refuse to handle some expressions, such as those involving method invocations.

Figure 5 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, including the OID.

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 will accept predicates of the form `<attr> <op> <const>`, where `<op>` is either = or the SQL keyword `LIKE`.[4] 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 difference 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 perform the predicate “`location LIKE '%beach%'`”, while reporting through the plan’s properties that the equality predicate will not be applied.

4.Indeed, the web wrapper insists that the work request contain at least one predicate, since the web site imposes this requirement. This restriction means that some Garlic queries that are syntactically correct are not answerable. For example, although the Garlic query “`select name, class from Hotels`” is syntactically correct, the web wrapper will not return an access plan for it because no predicate was specified. Without a wrapper plan, Garlic has no way of generating a plan to scan the collection, and therefore the query cannot be executed.

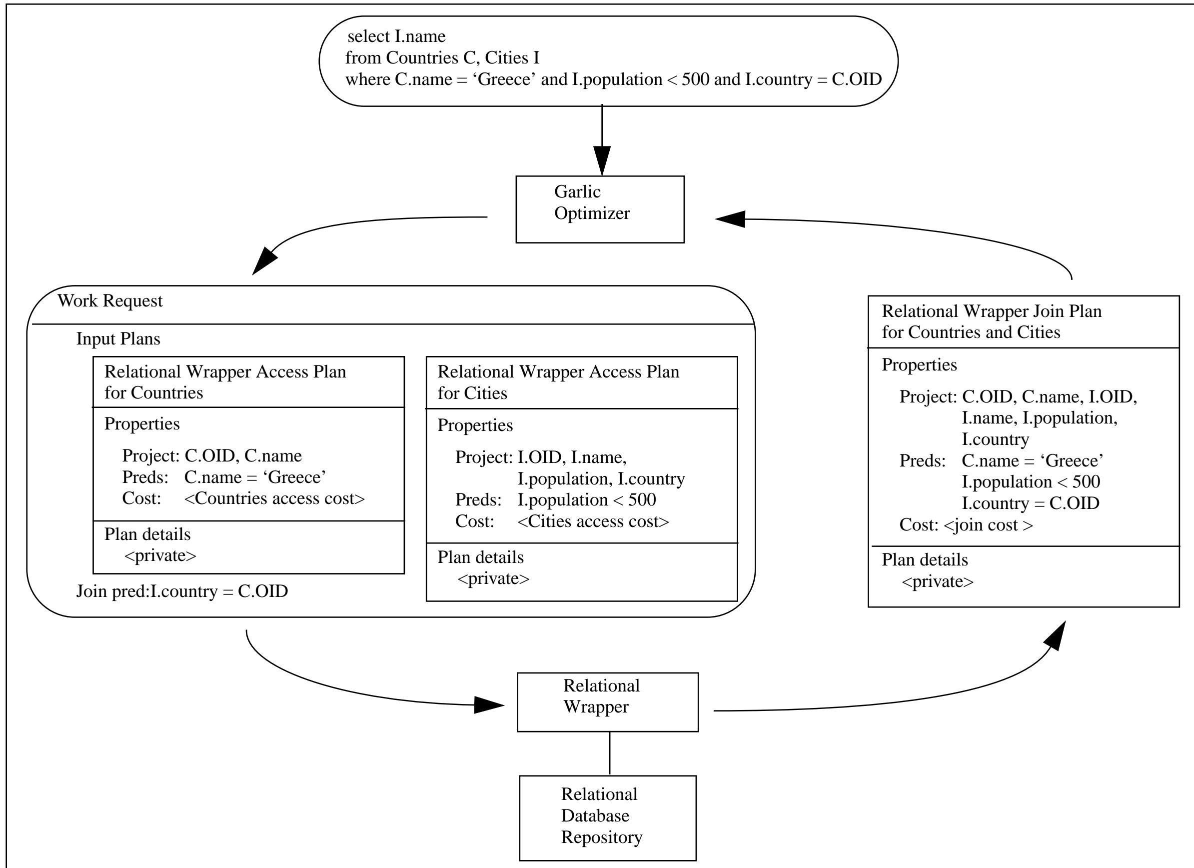

Figure 6. Construction of a Wrapper Join Plan.

In the example of Figure 5, the wrapper analyzes the work request and creates a `Wrapper_Plan` containing the information it will need in order to execute the plan (including application of a `LIKE` predicate on `location`). The plan's property list indicates that the plan will apply the predicate on `class`, but omits the predicate on `location`. The HTML pages returned by the web site contain all of the information for a hotel listing, so the wrapper agrees to handle the entire projection list: `OID`, `name`, `city`, `daily_rate`, `class`, and `location`. Lastly, the wrapper assigns the plan an estimated cost. Note that since the wrapper agreed to apply the predicate on `class`, it really does not need to return the value of `class` to Garlic. However, the Garlic query optimizer could not predict ahead of time that the wrapper would handle this predicate. It conservatively requested that the wrapper return the `class` attribute so the engine would not have to get it via a `get_class()` method invocation in order to apply the predicate.

This access plan is returned to the Garlic optimizer. If it 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. As in the

plan_access() case, the wrapper can produce zero or more plans and agree to handle any subset of the original projections and predicates, and/or the join predicates, as long as the property lists for the new plans are properly filled out. Garlic will add the new wrapper plans to the set it has under consideration. 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.

Let's return to our travel agency. Suppose that a user submits a query to find the names of cities in Greece with a small population. This query involves two collections, Countries and Cities, both managed by the relational wrapper. Since relational databases handle joins, our relational wrapper implements the plan_join() method. The wrapper will accept all predicates and projection requests that can be directly translated into SQL for the relational database. Join predicates may be on any attribute described in the interface, or an equality predicate on OID. The wrapper maps a Garlic work request to an SQL query against the underlying database, and stores the elements of the select, from and where clauses of the query in the private section of its plan.

Figure 6 shows how the relational wrapper provides a plan for the join between Countries and Cities. In the first phase of optimization (omitted from the picture), the optimizer requests access plans for Cities and Countries. The relational wrapper returns a plan for each and agrees to handle the simple predicates and projections on those collections. 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.

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 [18] and [17]. 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 bind join is attractive if the amount of data transferred from the outer node to the inner node is small, and the bind join predicates are selective enough to significantly reduce the amount of data returned by the inner node to Garlic. Not all wrappers can or need to serve as the inner node of a bind join. A wrapper is well suited to serve as the inner node of a bind join if the programming interface of its underlying data source 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 7 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 two-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 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. In the bind join predicate, $BIND_1 represents the name of a city where a hotel is located, as returned for each hotel

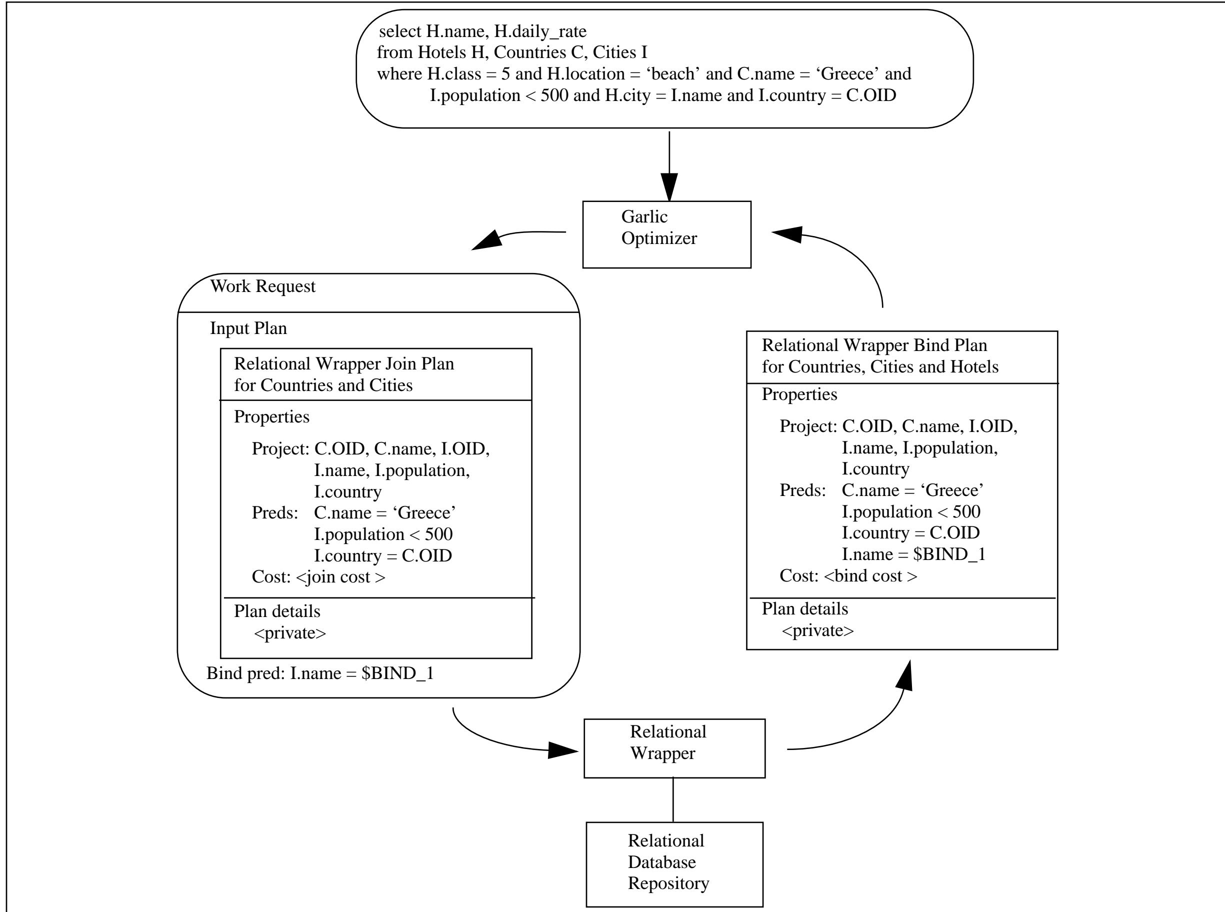

Figure 7. Construction of a Wrapper Bind Plan.

by the access plan for the `Hotels` collection. The relational wrapper is able to handle the bind predicate, and 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 8 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 for the `Hotels` collection (from Section 4.3.1), and the inner node is the relational wrapper's bind plan for `Countries` and `Cities` (from Section Section 4.3.3). Since the web wrapper reported that it only handled the predicate on `class`, the Garlic optimizer added a FILTER operator to handle the predicate on `location`. Finally, since the select clause of the user's query contained `name` and `daily_rate`, the optimizer added a PROJECT operator to project those fields out.

4.4.1 Translation

Once the optimizer has chosen a plan for a query, the plan must be translated into a form suitable for execution. As is common in demand-driven runtime systems [10], operators of the optimized plan are mapped into iterators, and each wrapper provides a specialized `Iterator` subclass that controls execution of the work described by one of its wrapper plans. The wrapper must also supply an implementation of

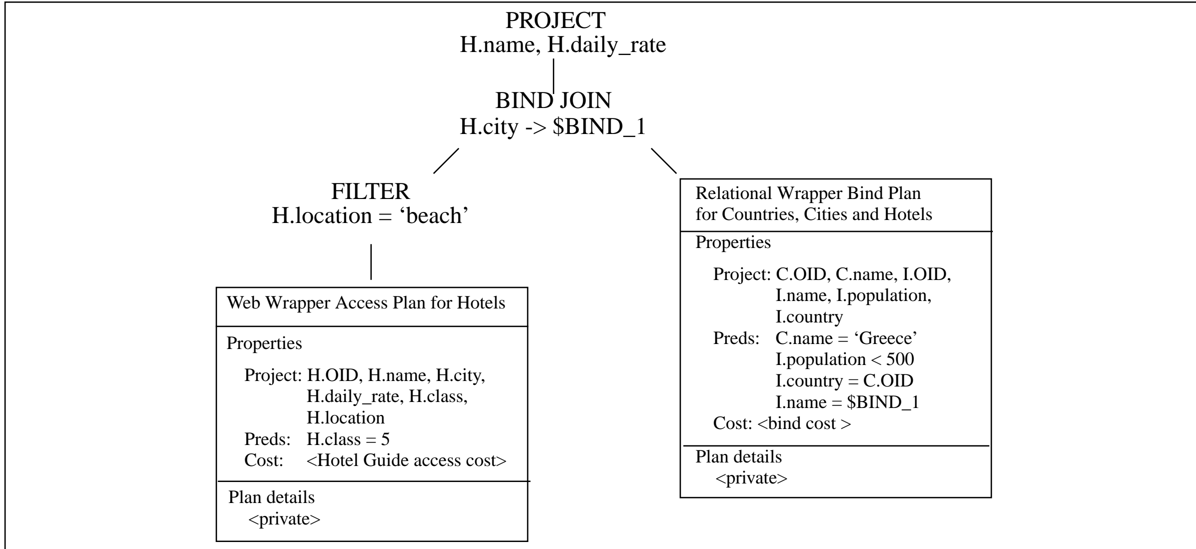

Figure 8. A Plan for a Garlic Query.

`Wrapper_Plan::translate()`, a method that takes no arguments and returns an instance of the wrapper's `Iterator` subclass, as described in the next section. At plan translation time, the Garlic query processor walks the optimized query plan and creates a tree of iterators. Whenever a wrapper's plan is encountered, it invokes the wrapper's `translate()` method.

For most wrappers, 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 in the private section of its plan. At plan translation time, the wrapper extracts these elements, constructs the query string to be sent to the relational database, and stores it in an instance of its `Iterator` subclass. OIDs mentioned in the plan are converted into accesses to the appropriate relation's primary key fields. If the plan is a bind plan, question marks appear in the query string to represent the unbound values. For example, the query string that would be generated for the relational wrapper plan in Figure 8 is the following (recall that `name` is the primary key for the `Countries` relation, and `name`, `country` is the primary key for the `Cities` relation):

```
select C.name, I.name, I.country, I.population
from Countries C, Cities I
where C.name = ‘Greece’ and I.population < 500 and
      I.country = C.name and I.name = ?
```

Similarly, 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 in Figure 8. At plan translation time, the predicates are used to form a query URL that the web site will accept. In order to form a correct URL, the web wrapper will have to fill in default values for attributes that do not have predicates specified in the plan.

4.4.2 Execution

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. Only wrappers that provide bind plans need to implement this method.

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 (represented by question marks) in the query string. The `advance()` method fetches the next set of tuples from the database. For efficiency, the wrapper manages a small cache and fetches blocks of tuples at a time. The wrapper repackages the retrieved data before returning it to Garlic. For example, after executing the relational query described in Section 4.4.1, the wrapper repackages the primary key and foreign key values retrieved from the relational database into the OIDs that the query processor is expecting.

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 retrieve the next set of results. To manufacture OIDs for the plan in Figure 8, the wrapper uses the internal keys for hotel listings stored on the page. Several results may appear on a page, so like the relational wrapper, the web wrapper maintains a small cache. Thus, in subsequent calls to `advance()`, the wrapper retrieves results stored in its cache. Each HTML page provides a link to the next page of results, so after all of the results in the cache are returned to Garlic, the wrapper loads the next page and caches the results stored on that page.

4.5 Wrapper Packaging

In previous sections, we have described the services that a wrapper provides to the Garlic middleware. Table 1 summarizes the tasks a wrapper author performs in order to provide these services. The wrapper author's final task is to package these pieces as a complete wrapper. A wrapper may include three kinds of components:

1. *Interface files* that contain one or more interface definitions written in GDL.
2. *Libraries* that contain dynamically loadable code to implement schema registration, method invocation, and the query interfaces. These are further subdivided as follows:
   a. *Core* libraries that contain common code shared among several similar repositories.
   b. *Implementation* libraries that contain repository-specific implementations of one or more interfaces.
3. *Environment files* that contain name/value pairs to encode repository-specific information for use by the wrapper. Garlic parses environment files and makes their contents available to the wrapper, but does not interpret them itself.

Packaging the code portion of wrappers as dynamically loadable libraries that reside in the same address space as the Garlic query processor 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 op-

Service	Tasks
Modeling Data as Objects	Describe interface of objects using GDL, provide key portion for OIDs of objects in repository, and identify selected objects as roots.
Method Invocation	Implement accessor methods and methods explicitly defined in interfaces.
Query Planning	Provide `Wrapper_Plan` subclass and implement query planning methods that model as much of the repository's query processing and search facilities as desired.
Query Execution	Provide `Iterator` subclass to control execution of a query plan, and implement `Wrapper_Plan::translate()` to create an iterator instance from a plan.

TABLE 1. Tasks Performed by a Wrapper Author.

timization of a query, and non-trivial data structures (work requests and plans) are exchanged at each interaction. Furthermore, wrapper authors are not constrained to use a particular inter-process communication or remote procedure call protocol to communicate with Garlic. Very simple repositories can be accessed without crossing address space boundaries at all, 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, different instances of our image server share the same schema and have the same query capabilities, but are contacted via different network addresses and export differently-named collections of images. Therefore, this wrapper is packaged as an interface file and core library that are shared among all image servers, plus environment files for each individual server. Our wrapper for relational database systems 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, and an environment file encodes the name of the database to which the wrapper must connect, the names of the roots 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, additional implementation libraries can be registered with stubs for the new implementations.

5 Current Status

To date, we have implemented wrappers for 10 data sources. Table 2 describes some of the features of these wrappers.Wrappers such as the relational wrapper and the web wrapper have been fine tuned and are fairly mature. Others are still in a state of evolution. For example, we have been investigating different database products to serve as Garlic's complex object repository, including object-oriented databases and object-relational systems. As a result, we temporarily modeled the wrapper as a simple repository so that we could quickly change the underlying data source. When we settle on the actual implementation of the complex object repository, we will enhance its wrapper appropriately.

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 of the 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 two-day visit to our lab.

The relational wrapper is by far the most complex. We spent a considerable amount of effort on it because a substantial amount of legacy data resides in relational databases, and we intend to package Garlic with a relational wrapper. The wrapper can be quickly adapted to any relational database system with a call level interface. A relational database can be integrated into a Garlic database in a matter of minutes, and we provide tools to automate the process. A DBA runs a tool that generates the GDL and environment files for the database by scanning the relational catalogs, and then uses a utility to register the wrapper as part of a Garlic database. After the registration step, the database can immediately be accessed through the Garlic interface.

Data Source	Schema description	Method invocation	Query operations handled by wrapper
DB2, Oracle	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	accessor methods only; generic dispatch	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, a directory of U.S. business listings	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	accessor methods only; generic dispatch	attribute projection, equality predicates on attributes, LIKE predicates of the form '%<value>%
Proprietary database for molecular similarity search	A single collection of molecule objects; interface has contains_substructure() and simliarity_to() methods to model search capability of engine; molecule l-number is key for OID	stub dispatch	attribute and method projection, predicates of the form <attr> <op> <const> and <method> <op> <const>, where <op> is a comparison operator
QBIC image server that orders images according to color, texture and shape features	Collections of image objects; interface contains matches() method to model ordering capability; image file name is key for OID	stub dispatch	ordering of image objects by image feature
Glimpse[19] text search engine that searches for specific patterns in text files	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	stub dispatch	projection of attributes and methods
Lotus Notes databases: Phone Directory database, Patent Server database	Notes database becomes a collection of note objects; interface defined by database Form; note NOTEID is key for Garlic OID	accessor methods only; generic dispatch	attribute projection, predicates containing logical, comparison and arithmetic operators, LIKE predicates, tests for NULL.
Complex Object Wrapper	Collections of objects; interface corresponds to interface of objects in database; database OID is key for Garlic OID	stub dispatch	attribute projection

TABLE 2. A Description of Existing Wrappers.

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 [25]. A common thread in most of this work has been an assumption that the underlying information sources are themselves database systems of one kind or another, and consequently provide relatively complete query processing services. As a result, much of the research has been focused on topics such as the formalization of mappings between different data models and their associated query languages, strategies for concurrency control and recovery among databases with different transaction models, and the reconciliation of data semantics and representation when several databases contain information about the same real-world entities. Surveys of much of this work can be found in [3] [8] [13] [24]. The architectures of most earlier systems are built around a *lingua franca* for communicating with the underlying information sources. A query that requires access to multiple data sources goes first to a central query decomposition facility, where it is broken into fragments that are expressed in a common language or by means of a common data structure. Each source provides a "local query

translator” that maps fragments represented in this form into appropriate operations on the data source. In Pegasus [1], for example, fragments are represented in the HOSQL query language. MAKBIS [9] uses MAKBIS-QL and EDA/SQL [27] uses iSQL. All of these systems assume that any data source, assisted by the translator, can readily execute any query fragment.

Recently, the proliferation both in number and kind of information sources on the World Wide Web and elsewhere has stimulated interest in systems, like Garlic, that attempt to integrate a wider variety of data sources. The assumption that all data sources provide general-purpose query processing capabilities is much harder to justify in this context, hence, for these systems, accommodating a range of query processing power has become an important area of research.

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 is unable to handle to be performed by the middleware.

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. The Information Manifold [15] requires each data source to supply a set of *capability records* that describe its ability to retrieve and filter data. The capability record for each relation (or class of objects) specifies three sets of attributes. The first set lists input attributes. In a supported query, some minimum number of these (also specified in the capability record) must be bound to constant values. The second set lists output attributes. A supported query can request the values of up to some maximum number of these as output. The third set lists selection attributes. A supported query may contain range predicates that compare any or all of these attributes to a constant.

DISCO [12] defines a query processing model based on an algebra of logical operators such as `select`, `project`, and `scan`, and imposes this model on the wrappers. At wrapper registration time, the wrapper describes the subset of the logical operators that are supported by the data source. The wrapper interface language contains some support for describing restrictions on supported operators, such as which operators can be applied to which collections, or which predicates can be applied to which attributes. The language also provides some support for restricting the combination of operators that a data source supports.

In the TSIMMIS system [22], specifications of query power are expressed in the Query Description and Translation Language (QDTL) [21]. A QDTL specification for a data source is a context-free grammar for generating supported queries. In its simplest form, a QDTL specification is a query template containing one or more *placeholders*, variables that will be bound to constant values in actual queries. A specification may also contain *metapredicates*, built-in or source-specific C functions that test the value bound to a placeholder. More complex QDTL specifications use nonterminal symbols and recursion to allow large numbers of query templates to be expressed compactly. For example, one can describe a class of predicates and the attributes to which they can be applied without enumerating all possible combinations of predicates and attributes. A yacc-like wrapper generation tool is provided to facilitate the translation of query fragments expressed in MSL, the system’s query language. The tool allows a wrapper author to augment QDTL specifications with semantic actions that generate corresponding primitives understood by the data source. A more recent paper [23] discusses the Relational Query Description Language (RQDL), an extension of QDTL that supports schema-independent specifications. For example, in RQDL, one can compactly specify

a template that describes the ability to apply a certain kind of predicate to any attribute of any relation.

The idea of compact declarative specifications of query power is inherently 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 such cases, a good execution strategy is to send the subsuming query to the data source for evaluation, and subsequently apply a filter to obtain the answer to 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 underlying data source [21].

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 a wrapper dynamically examines a query fragment and presents the Garlic optimizer with one or more supported subsuming queries. 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, taking advantage of repository-specific semantic knowledge as they see fit. Since our approach is not limited by the expressive power of a query specification language, it 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, including standard data sources such as relational databases, searchable web sites, image servers, and text search engines, as well as specialized data sources such as a chemical compound search engine. 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. Rather than requiring a wrapper to raise a repository's query processing capabilities to a fixed level, or "dumbing down" the query processing interface to the lowest common denominator, our architecture allows each wrapper to dynamically determine 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/RQDL-style tem-

plates 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. For example, a wrapper could use a template to indicate that it only supported comparisons of the form "`<attr> = <const>`". Given this information, the query processor would never include predicates like "`<attr> > <const>`" in a work request for that wrapper. Such template would be a step toward a hybrid system, combining Garlic's dynamic approach to query planning with the declarative approach of TSIMMIS and the Information Manifold; striking an appropriate balance between the two 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., "PESTO: An Integrated Query/Browser for Object Databases", *Proc. of the 22nd International Conference on Very Large Data Bases,* Bombay, India, September 1996.

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

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

[7] O. Deux et al., "The Story of O_2", *IEEE Transactions on Knowledge and Data Engineering* 2(1), March 1990.

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

[9] N. Elshiewy, "MAKBIS: Coordinated Access to Heterogeneous and Autonomous Information System", *Proc. IEEE RIDE-DOM*, Taipei, Taiwan, March 1995.

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

[11] L. Haas, D. Kossmann, E. Wimmers, and J. Yang, "Optimizing Queries Across Diverse Data Sources", *Proc of the 23rd International Conference on Very Large Data Bases,* Athens, Greece, August 1997.

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

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

[14] A. Lamb, et. al., “The ObjectStore Database System”, *Communications of the ACM,*34(10), October 1991.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] U. Manber, S. Wu, and B. Gopal, see documentation at: http://glimpse.cs.arizona.edu/.

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

[21] Y. Papakonstantiou, A. Gupta, H. Garcia-Molina, and J. Ullman, “A Query Translation Scheme for Rapid Implementation of Wrappers”, *Proc. of the Conference on Deductive and Object-Oriented Databases (DOOD)*, 1995.

[22] Y. Papakonstantiou, H. Garcia-Molina, and J. Widom, “Object Exchange Across Heterogeneous Information Sources”, *Data Engineering Conf.,* March 1995.

[23] Y. Papakonstantiou, A. Gupta, and L. Haas, “Capabilities-based Query Rewriting in Mediator Systems”, *Proc. of the International IEEE Conference on Parallel and Distributed Information Systems*, Miami, FL, USA, December 1996.

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

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

[26] M. Stonebraker, “Object-Relational DBMSs: The Next Great Wave”, Morgan Kaufmann Publishers, San Francisco, 1996.

[27] 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.

[28] M. Tork Roth, et. al., “The Garlic Project”, *Proc. of the ACM SIGMOD Conference on Management of Data, Montreal, PQ, Canada,* June 1996.