# Transforming Heterogeneous Data with Database Middleware: Beyond Integration

L. M. Haas R. J. Miller B. Niswonger M. Tork Roth P. M. Schwarz E. L. Wimmers
{*laura, niswongr, torkroth, schwarz, wimmers*}*@almaden.ibm.com; miller@cs.toronto.edu*

## 1 Introduction

Many applications today need information from diverse data sources, in which related data may be represented quite differently. In one common scenario, a DBA wants to add data from a new source to an existing warehouse. The data in the new source may not match the existing warehouse schema. The new data may also be partially redundant with that in the existing warehouse, or formatted differently. Other applications may need to integrate data more dynamically, in response to user queries. Even applications using data from a single source often want to present it in a form other than that it is stored in. For example, a user may want to publish some information using a particular XML DTD, though the data is not stored in that form.

In each of these scenarios, one or more data sets must be mapped into a single target representation. Needed transformations may include schema transformations (changing the structure of the data) [BLN86, RR98] and data transformation and cleansing (changing the the format and vocabulary of the data and eliminating or at least reducing duplicates and errors) [Val, ETI, ME97, HS95]. In each area, there is a broad range of possible transformations, from simple to complex. Schema and data transformation have typically been studied separately. We believe they need to be handled together via a uniform mechanism.

Database middleware systems [PGMW95, TRV96, ACPS96, Bon95] integrate data from multiple sources. To be effective, such systems must provide one or more integrated schemas, and must be able to transform data from different sources to answer queries against these schema. The power of their query engines and their ability to connect to several information sources makes them a natural base for doing more complex transformations as well. In this paper, we look at database middleware systems as tranformation engines, and discuss when and how data is transformed to provide users with the information they need.

## 2 Architecture of a DB Middleware System

To be a successful data transformation engine for scenarios such as the above, a database middleware system must have several features. Since data these days comes from many diverse systems, it must provide access to a broad range of data sources transparently. It must have sufficient query processing power to handle complex operations, and to compensate for limitations of less sophisticated sources. Some transformation operations (especially the complex ones) require that data from different sources be interrelated in a single query.

We use Garlic [C$^+$95] to illustrate the ideas of this paper. Figure 1 shows Garlic's architecture, which is typical of many database middleware systems [PGMW95, TRV96, ACPS96]. Garlic is primarily a query processor;


**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

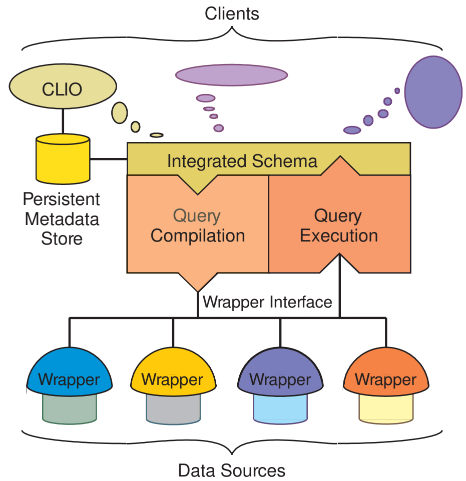


Figure 1: Garlic architecture

it optimizes and executes queries over diverse data sources posed in an object-extended SQL. Garlic's powerful query engine is capable of executing any extended SQL operation against data from any source. In both planning and executing the query, it communicates with *wrappers* for the various data sources involved in the query. Systems of this type have two opportunities to transform data: first, at the wrapper as the data is mapped from the source's model to the middleware model (Section 3), and second, by queries or views against the middleware schema (Sections 4 and 5). However, understanding how the data needs to be transformed is not always simple. The target representation is often only defined implicitly, by existing data. The data integration tool, Clio, shown here and in more detail in Figure 4 will help users understand both their source's data and the target representation and will assist them in creating a mapping between them.

# 3 Data Transformation at the Wrapper

The most basic tasks of a wrapper are a) to describe the data in its repository and b) provide the mechanisms by which users and the Garlic middleware engine may retrieve that data [RS97]. Since a data source is not likely to conform to Garlic's data model and data format, these two tasks imply that the wrapper must perform some level of schema and data transformation. To make the task of writing a wrapper as easy as possible, the Garlic wrapper architecture tries to minimize the required transformations, but wrappers can do more if desired.

The schemas of individual repositories are merged into the global schema via a wrapper registration step. In this step, wrappers model their data as Garlic objects, and provide an *interface* definition that describes the behavior of these objects. The interface is described using the Garlic Definition Language (GDL), which is a variant of the ODMG Object Description Language [Cat96]. The interface definition provides an opportunity for a wrapper to rename objects and attributes, change types and define relationships even if the data source stores none. For example, a relational wrapper might model foreign keys as relationships. Developing interface files is typically not hard. For simple data sources, it may be best to generate them manually, as simple sources tend to have few object types, usually with fairly simple attributes and behavior. For more sophisticated sources, the process of generating an interface file can often be automated. For example, a relational wrapper can decide on a common mapping between the relational model and the Garlic data model (e.g. tuple = object, column = attribute), and provide a tool that automatically generates the interface file by probing the relational database schema. Wrappers must also provide an *implementation* of the interface which represents a concrete realization of the interface. The implementation cooperates with Garlic to assign a Garlic object id (OID) to its objects, and maps the GDL base types specified in the interface file to the native types of the underlying data source.

**Oracle Database Wrapper**

**Relational Schema**

```
CREATE TABLE COUNTRIES {
    NAME VARCHAR(30) NOT NULL,
    CLIMATE VARCHAR(256),
    HIGHESTPEAK NUMBER(4),
    PRIMARY KEY(NAME) }

CREATE TABLE CITIES {
    NAME VARCHAR(40) NOT NULL,
    COUNTRY VARCHAR(30) NOT NULL,
    POPULATION NUMBER(4),
    ELEVATION NUMBER(7,2),
    AREA NUMBER(7,2),
    PRIMARY KEY(NAME),
    FOREIGN KEY(COUNTRY)
        REFERENCES COUNTRIES }
```

**Garlic Schema**

```
interface Country_Type {
    attribute string name;
    attribute string climate;
    attribute long highest_peak;
};

interface City_Type {
    attribute string name;
    attribute ref¡Country_Type¿ country;
    attribute long population;
    attribute double elevation;
    attribute double area;
};
```

**Hotel Web Site Wrapper**

```
interface Hotel_Type {
    attribute string name;
    attribute string street;
    attribute string city;
    attribute string country;
    attribute long postal_code;
    attribute string phone;
    attribute string fax;
    attribute short number_of_rooms;
    attribute float avg_room_price;
    attribute short class;
    void display_location();
};
```

Figure 2: Example of wrapper schemas

A hypothetical travel agency application illustrates the kinds of simple schema and data transformations that wrappers can perform. The agency would like to integrate an Oracle database of information on the countries and cities for which it arranges tours with a web site that contains up-to-date booking information for hotels throughout the world. Figure 2 shows the orginal table definitions and the new interface definitions for the two relational tables, and the interface file for the web site. The relational wrapper renamed the `HIGHESTPEAK` field to `highest_peak`, and exposed the foreign key `COUNTRY` on the `CITIES` table as an explicit reference to a `Country` object in the integrated database. The wrapper must be able to map requests for this attribute from the integrated database (in OID format) into the format expected by the relational database (as a string), and vice versa. In addition, the `POPULATION`, `ELEVATION` and `AREA` columns are all stored as type `NUMBER`, yet `population` has type long in the interface file, while `elevation` and `area` are doubles.

Each hotel listing on the web site contains HTML-tagged fields describing that hotel, and a URL to map the location of a particular hotel given its key. In the interface definition file, the HTML fields are represented as attributes of the Hotel object, each with an appropriate data type, though the web site returns all data in string format. The map capability is exposed as the `display-location` method. It is the wrapper's responsibility to map names to the fields on the HTML page, and to convert data from strings into appropriate types.

# 4 Data Transformation in the Middleware

Views are an important means of reformatting data, especially for middleware, as the data resides in data sources over which the user has little control. Views provide the full power of SQL to do type and unit conversions not anticipated by the wrapper, merging or splitting of attributes, aggregations and other complex functions. In Garlic, *object views* allow further restructuring of data. It is frequently the case that the information about a particular conceptual entity is part of several objects stored in various data sources. However, end-users want to see a single object. An object view creates a new "virtual" object. This virtual object requires no storage since attributes are specified in a query rather than stored as base data. Every virtual object in Garlic is based on another object (which could itself be a virtual object). Garlic uses the OID of the base object as the basis for the virtual object's OID, and provides a function, LIFT, to map the base OID to the virtual object's OID.

One important reason to have virtual objects is to allow new behavior, i.e., new methods, to be defined for these objects. Methods on views can also be used to "lift" methods on the base objects so that virtual objects can retain the base object's functionality. Each method on a virtual object is defined by an SQL query. This query has access to the OID of the virtual object upon which the method is invoked via the keyword *self*, and can find the OID of the base object, if needed. Methods return at most one item; otherwise a run-time error results.

```
interface city_listing_Type {
    attribute string name;
    attribute string country;
    attribute float population_in_millions;
    attribute float elevation_in_meters;
    attribute set¡ref¡Hotel_Type¿¿ hotels;
    string find_best_hotel(IN long budget);
};
```

```
create view city_listing (name, country, population_in_millions, elevation_in_meters, hotels, self)
    as select C.name, C.country, C.population/1000000, C.elevation*0.3048,
        MAKESET(H.OID), LIFT('city_listing', C.OID)
    from Cities C, Hotels H
    where C.name=H.city and UCASE(C.country-¿name)=H.country
    group by C.name, C.country, C.population, C.elevation, C.OID

create method find_best_hotel(long budget)
    return
    select h1.name from unnest self.hotels h1
    where h1.class > all (select h2.rating from unnest self.hotels h2
                          where h2.name ≠ h1.name and h2.avg_room_price ≤ budget)
        and h1.avg_room_price ≤ budget
```

Figure 3: A sample view definition, with method

To register an object view in Garlic, the user must provide both an interface and an implementation (definitions of the view and any methods), as illustrated in Figure 3. This view, based on the `City` objects defined in Section 3, creates `City_Listing` objects that have most of the attributes of a `City` (but omit, for example, area), and an associated set of hotels. The view definition shows how these objects would be created. It uses some of Garlic's object extensions to SQL, including a path expression to get the name of the `City`'s country, and a new aggregate function, `MAKESET`, that creates a set. Note that the LIFT function is used to compute the OID of the new virtual object. All the attributes of the virtual object must be included in the select list. The view definition does some simple unit conversions using arithmetic functions, and uses the uppercase function to map country names from the Oracle database to the names of countries in the Web source. More complex mappings (using translation tables or user-defined functions, for example) would also be possible. The method finds the best hotel within a certain budget in the city on which it is invoked. The budget is an argument of the method. Note the use of *self* to identify the correct set of hotels.

# 5 Data Mapping and Integration

We have described two components of Garlic that provide important data transformation facilities. The wrappers provide transformations required by the individual data sources, including data model translation and simple schema and data transformations. Object views enhance the wrapper transformations with a general view mechanism for integrating schemas. Object views support integrated cooperative use of different legacy databases, through query language based transformations, such as horizontal or vertical decomposition (or composition) of classes. Such transformations are required to integrate overlapping portions of heterogeneous databases.

In addition, we are working to provide a more powerful suite of schema and data transformations to permit integration of schemas that exhibit schematic discrepancies, and matching of data objects that represent the same real-world entity. Across heterogeneous data sources, different assumptions may have been made as to what data is time invariant and therefore appropriate to include as metadata rather than data. Data under one schema may be represented as metadata (for example, as attribute or class names) in another. Such heterogeneity has been referred to as *schematic heterogeneity*. Traditional query languages are not powerful enough to restructure both data and metadata [Mil98, LSS96]. Likewise, in heterogeneous systems different representations of the same entity in different sources are common. The same object may be identified by a different name in two sources, or even by a different key. Further, different entities may bear similar names in different sources. Identifying equivalent objects and merging them also requires new, powerful transformations [ME97, HS95, Coh98].

In many tasks requiring data translation, the form of the translated data (its schema) is fixed or at least constrained. In a data warehouse, the warehouse schema may be determined by the data analysis and business support tasks the warehouse must support. As new data sources are added to the warehouse, their schemas must be mapped into the warehouse schema, and equivalent objects must be found and converged. Hence, the integrated

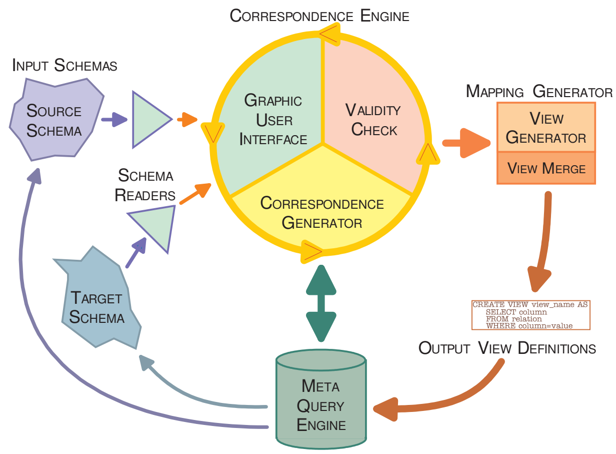


Figure 4: Tool architecture

view is no longer synthesized via a set of transformations local to individual source databases. Rather, it must be possible to discover how a source's schema components and data correspond to a fixed target schema and any existing data. Unlike traditional schema and data integration, this mapping process may require non-local transformations of source schemas and data objects.

As an example, consider the `Hotel_Type` defined in Figure 2. Now imagine that a new source of hotel data has become available, in which some of the data is redundant with the original web source, but most is not. This new source of data has a collection for each country: one for France, one for China, etc. We wish to incorporate this new source in such a way that the user sees only one collection of hotels. However, there are several obstacles to this goal. First, there is schematic heterogenity: in the new source the country name is metadata rather than an attribute value. Second, the system needs to be able to identify when hotels from the two sources are the same. The same hotel may be identified by a different name in the two sources (e.g., "Hyatt St. Claire" vs. "St. Claire Hyatt"), and two different hotels may have the same name (e.g., "Holiday Inn Downtown" exists in many cities). Metadata transformations that use higher order query language operators [Ros92] are needed for the former, dynamic cleansing operations such as joins based on *similarity* [Coh98] for the latter.

## 6 Building the Integrated Schema

Converting from one data representation to another is time-consuming and labor-intensive, with few tools available to ease the task. We are building a tool, Clio[1], that will create mappings between two data representations semi-automatically (i.e., with user input). Clio moves beyond the state of the art in several ways. First, while most tools deal with either schema integration or data transformation, it tackles both in an integrated fashion. Second, it employs a full database middleware engine, giving it significantly more leverage than the *ad hoc* collections of tools available today, or the lightweight "agents" proposed by others. Third, it will exploit the notion of a target schema, and, where it exists, target data, to make the integration problem more tractable. Fourth, because the middleware engine is being enhanced with more powerful transformation capabilities (Section 5) than most, it will allow more complex transformations of both schema and data. Finally, it will use data mining techniques to help discover and characterize the relationships between source and target schema and data.

The tool has three major components (Figure 4): a set of Schema Readers, which read a schema and translate it into an internal representation (possibly XML); a Correspondence Engine (CE), which finds matching parts of the schemas or databases; and a Mapping Generator, which will generate view definitions to map data in the

[1] Named for the muse of history, so that it will deal well with legacy data!

source schema into data in the target schema. The CE has three major subcomponents: a GUI for graphical display of the schemas and relevant data items, a correspondence generator, and a component to test correspondences for validity. Initially, the CE will expect the user to identify possible correspondences, via the graphical interface, and will provide appropriate data from source and target (using the meta query engine) for verifying the correspondences and identifying the nature of the relationship (again, initially relying on the user). This will be an iterative process. Over time, we anticipate increasing the "intelligence" of the tool using mining techniques so that it can propose correspondences, and eventually, verify them.

Clio will be general, flexible, and extensible. We expect to have a library of code modules (e.g. Java Beans) for transformation operators, which the middleware engine will be able to apply internally. Open research issues include what set of transformations are useful, and whether all transformations (particularly data cleansing) can be done efficiently on the fly. We believe Clio's modular design provides the flexibility required to experiment with a wide range of transformation operators, allowing it to serve as a test bed for further research in this area.

# References


[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. ACM SIGMOD*, 25(2):137–148, Montreal, Canada, June 1996.

[BLN86] C Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies of database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[Bon95] C. Bontempo. *DataJoiner for AIX*. IBM Corporation, 1995.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems. In *Proc. of the Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[Cat96] R. G. G. Cattell. *The Object Database Standard – ODMG-93*. Morgan-Kaufmann, San Mateo, CA, 1996.

[Coh98] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. ACM SIGMOD*, 27(2):201–212, Seattle, WA, June 1998.

[ETI] ETI - Evolutionary Technologies International. http://www.evtech.com/.

[HS95] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proc. ACM SIGMOD*, 24(2):127–138, San Jose, CA, May 1995.

[LSS96] L. Lakshmanam, F. Sadri, and I. N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[ME97] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. *Proc. of SIGMOD 1997 Workshop on Data Mining and Knowledge Discovery*, May 1997.

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. *Proc. ACM SIGMOD*, 27(2):189–200, Seattle, WA, June 1998.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. IEEE Conf. on Data Engineering*, pages 251–260, Taipeh, Taiwan, 1995.

[Ros92] K. A. Ross. Relations with Relation Names as Arguments: Algebra and Calculus. *Proc. ACM PODS*, pages 346–353, San Diego, CA, June 1992.

[RR98] S. Ram and V. Ramesh. Schema integration: Past, present, and future. In A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors, *Management of Heterogeneous and Autonomous Database Systems*. Morgan-Kaufmann, San Mateo, CA, 1998.

[RS97] M. Tork Roth and P. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 266–275, Athens, Greece, August 1997.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. In *Proc. ICDCS*, 1996.

[Val] Vality Technology Incorporated. http://www.vality.com/.