Data Warehouse Scenarios for Model Management

Philip A. Bernstein, Erhard Rahm[1]

Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399 U.S.A.
philbe@microsoft.com, rahm@informatik.uni-leipzig.de

Abstract. Model management is a framework for supporting meta-data related applications where models and mappings are manipulated as first class objects using operations such as Match, Merge, ApplyFunction, and Compose. To demonstrate the approach, we show how to use model management in two scenarios related to loading data warehouses. The case study illustrates the value of model management as a methodology for approaching meta-data related problems. It also helps clarify the required semantics of key operations. These detailed scenarios provide evidence that generic model management is useful and, very likely, implementable.

1 Introduction

Most meta-data-related applications involve the manipulation of models and mappings between models. Such applications include data translation, data migration, database design, schema evolution, schema integration, XML wrapper generation, message mapping for e-business, schema-driven web site design, and data scrubbing and transformation for data warehouses. By "model," we mean a complex discrete structure that represents a design artifact, such as an XML DTD, web-site schema, interface definition, relational schema, database transformation script, semantic network, or workflow definition. One way to make it easier to develop meta-data related applications is to make *model* and *mapping* first-class objects with generic high-level operations that simplify their use. We call this capability *model management* [1,2].

There are many examples of high-level algebraic operations being used for specific meta-data applications [4, 7, 10, 11, 14]. However, these operations are not defined to be generic across application domains. Our vision is to provide a truly generic and powerful model management environment to enable rapid development of meta-data related applications in different domains. To this end we need to define operations that are generic, powerful, implementable, and useful.

In this paper, we take a step toward this goal by investigating the detailed semantics of some of the operations proposed in [2]. We do this by walking through the design of two specific data warehouse scenarios. In addition to providing evidence that our model management approach can solve realistic problems, these scenarios also demonstrate a methodology benefit: Reasoning about a problem using high-level model management operations helps a designer focus on the overall strategy for

[1] On leave from University of Leipzig (Germany), Institute of Computer Science.

manipulating models and mappings — the choice of operations and their order. We believe that solution strategies similar to the ones developed in this paper can be applied in other application domains as well.

We begin in Section 2 with definitions of the model management operations. Sections 3 and 4 describe applications of these operations to two data warehouse scenarios. Section 5 summarizes what we learned from this case study.

2 Model Representation and Operations

This section summarizes the model management approach introduced in [2]. We represent models by objects in an object-oriented database. Some of the relationships in the database are distinguished as *containment relationships* (e.g., by a "containment flag" on the relationship). A *model* is identified by a root object *r* and consists of *r* plus the objects that are reachable from *r* by following containment relationships.

A mapping, *map*, is a model that relates the objects of two other models, M_1 and M_2. Each object in *map*, called a *mapping object*, has two properties, domain and range, which point to objects in M_1 and M_2 respectively. It may also have a property expr, which is an expression whose variables include objects of M_1 and M_2 referenced by its domain and range; the expression defines the semantics of that mapping object.

For example, Fig. 1 shows two Customer relations represented as models M_1 and M_2. Mapping map_1 associates the objects of the two models. Mapping object m_1 has domain {C#}, range {CustID}, and expr "Cust.C# = Customer.CustID" (not shown). Similarly for m_2. For m_3, the domain is {FirstName, LastName}, range is {Contact}, and expr is "Customer.Contact =Concatenate(Cust.FirstName, Cust.LastName)".

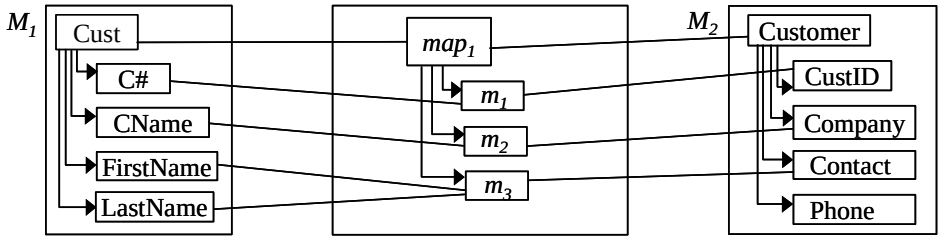


Fig. 1. A simple mapping map_1 between models M_1 and M_2

Models are manipulated by a repertoire of high-level operations including

- Match – create a mapping between two models
- ApplyFunction – apply a given function to all objects in a model
- Union, Intersection, Difference –applied to a set of objects
- Delete – delete all objects in a model
- Insert, Update – applied to individual objects in models

Unless a very controlled vocabulary is used in the models, the implementation of a generic Match operation will rely on auxiliary information such as dictionaries of synonyms, name transformations, analysis of instances, and ultimately a human arbiter. Approaches to perform automatic schema matching have been investigated in [3, 5, 7, 8, 9, 10, 12, 13].

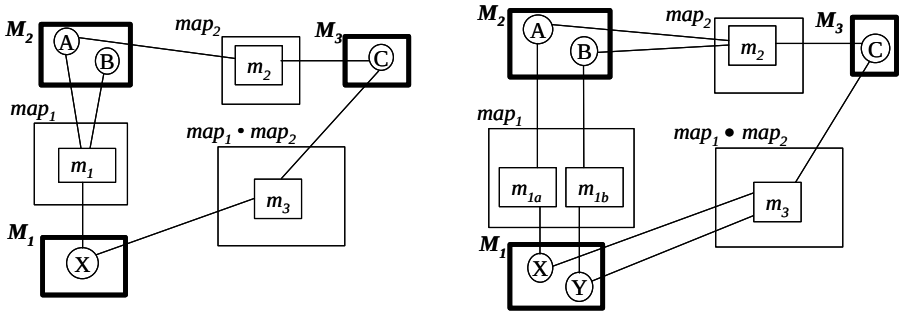
By analogy to outer join in relational databases, we use OuterMatch to ensure that all objects of an input model are represented in the match result. For instance, RightOuterMatch(M_1, M_2) creates and returns a mapping *map* that "covers" M_2. That is, every object *o* in M_2 is in the range of at least one object *m* in *map*, e.g., by matching *o* to the empty set, if *o* doesn't match anything else (i.e., range(*m*) = {*o*}, domain(*m*) = ∅). For example, in Fig. 1, to make map_1 a valid result of RightOuterMatch(M_1, M_2), we need to add a node m_4 in map_1 with range(m_4) = PhoneNo and domain(m_4) = ∅.

Since mappings are models, they can be manipulated by model operations, plus two operations that are specific to mappings:

- Compose – return the composition of two mappings
- Merge – merge one model into another based on a mapping.

Compose, represented by •, creates a mapping from two other mappings. If map_1 relates model M_1 to M_2, and map_2 relates M_2 to M_3, then the composition map_3 = $map_1 \bullet map_2$ is a mapping that relates M_1 to M_3. That is, given an instance *x* of M_1, $(map_1 \bullet map_2)(x) = map_2(map_1(x))$, which is an instance of M_3. There are right and left variations, depending on which mapping drives the composition and is completely represented in the result; we define RightCompose here.

The definition of composition must support mapping objects whose domains and ranges are sets. For example, the domain of a mapping object m_2 in map_2 may be covered by a proper subset of the range of a mapping object m_1 in map_1 (e.g., Fig. 2a). Or, a mapping object in map_2 whose domain has more than one member may use more than one mapping object in map_1 to cover it. For example, in Fig. 2b the domain of m_2 is covered by the union of the ranges of m_{1a} and m_{1b}.



a) domain(m_2) ⊂ range(m_1) b) domain(m_2) is covered by two objects, m_{1a} and m_{1b}.

Fig. 2. Composition with set-oriented domains and ranges (Examples)

In general, multiple mapping objects in map_1 may be able to provide a particular input to a mapping object in map_2. For example, in Fig. 2a, a second object m_1' in map_1 may have A in its range, so either m_1 or m_1' could provide input A to m_2. In the examples in this paper, each input of each M_2 object, *m*, is in the range of at most one map_1 object, so there is never a choice of which map_1 object should provide input to *m*. However, for completeness, we give a general definition of composition that handles cases where a choice of inputs is possible.

In the general case, the composition operation must identify, for each object o in the domain of each mapping object m in map_2, the mapping objects in map_1 that provide input to o. We define a function f for this purpose,

f: $\{m \in map_2\} \times \cup_{m \in map2}$ domain(m) $\rightarrow \{m' \in map_1\}$

such that if $f(m, o) = m'$ then $o \in$ range(m') (i.e., m' is able to provide input o to m). Given f, we create a copy of each mapping object m in map_2 and replace its domain by its "new-domain," which is the domain of the map_1 objects that provide m's input. More precisely, for $m \in map_2$, we define the set of objects that provide input to m:

$input(m) = \{ f(m, o) \mid o \in$ domain(m) $\}$

based on which, we define the new-domain(m) as follows:

if domain(m) $\subseteq \cup_{m' \in input(m)}$ range(m') and domain(m) $\neq \emptyset$

then new-domain(m) $= \cup_{m' \in input(m)}$ domain(m') else new-domain(m) $= \emptyset$.

So, the *right composition* of map_1 and map_2 with respect to f, represented by $map_1 \bullet_f map_2$, is defined constructively as follows:

1. Create a shallow copy map_3 of map_2 (i.e., copy the mapping objects and their relationships, but not the objects they connect to)
2. For each mapping object m'' in map_3, replace domain(m'') by newdomain(m), where m is the map_2 object of which m'' is a copy.

This definition would need to be extended to allow $f(m, o)$ to return a set of objects, that is, to allow an object in domain(m) to take its input from more than one source. We do not define f explicitly in later examples, since there is only one possible choice of f and the choice is obvious from the context.

The above definitions leave open how to construct the expression for each mapping object in the result of a composition, based on the expressions in the mapping objects being composed. Roughly speaking, in step (2) of the above definition, each reference to an object o in m''.domain should be replaced in m''.expr by the expression in the map_1 object that produces o. For example, in Fig. 2b, replace references to A and B in m_2.expr by m_{1a}.expr and m_{1b}.expr, respectively. However, this explanation is merely intuition, since the details of how to do the replacement depend very much on the expression language being used. In this paper, we use SQL.

The Merge operation copies some of the objects of one model M_2 into another M_1, guided by a mapping, *map*. We finesse the details here, as they are not critical to the examples at hand. As discussed in [2], a variety of useful semantics is possible.

3 Data Warehouse Scenario 1: Integrating a New Data Source

A data warehouse is a decision support database that is extracted from a set of data sources. A data mart is a decision support database extracted from a data warehouse. To illustrate model management operations, we consider two scenarios for extending an existing data warehouse: adding a new data source (Section 3) and a new data mart (Section 4). These are challenging scenarios that commonly occur in practice.

We assume a simple data warehouse configuration covering general order processing. It has a relational data source described by schema rdb1 (shown in Fig. 3), a relational warehouse represented by star schema dw1 (Fig. 4), and mapping map_1 between rdb1 and dw1 (Fig. 5). We note the following observations about the configuration:

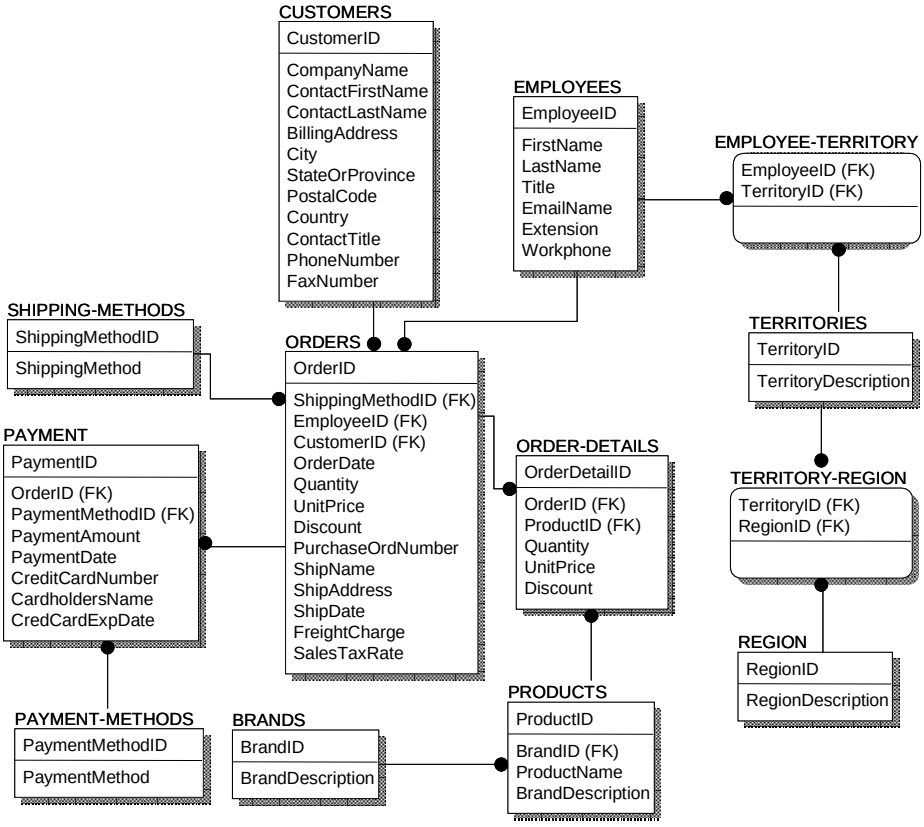


Fig. 3. Relational schema rdb1

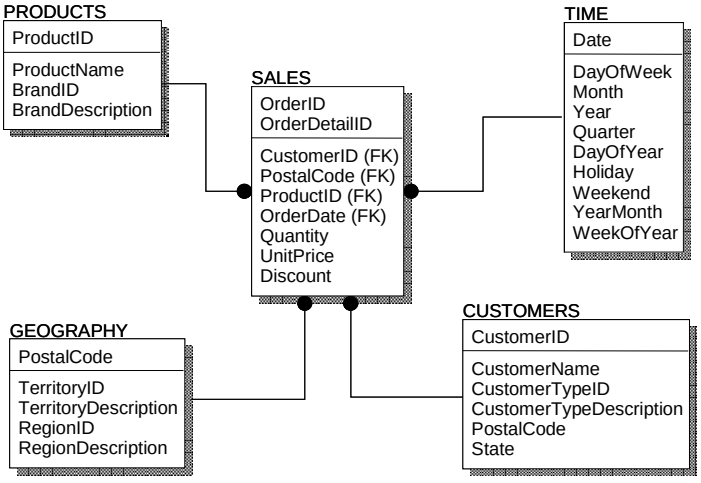
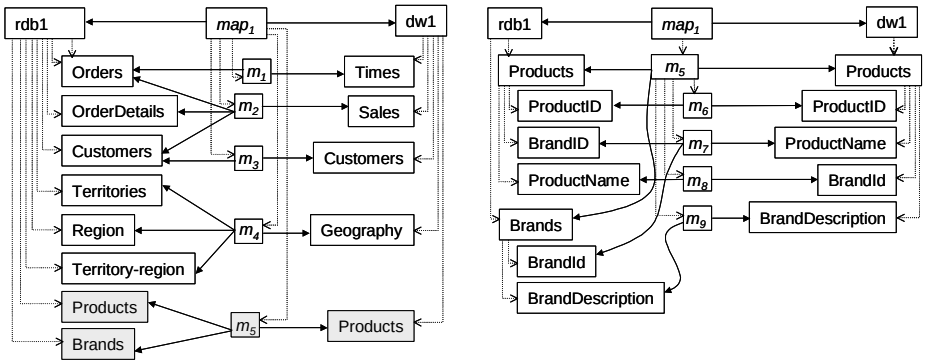


Fig. 4. Star schema of data warehouse dw1



(a) High-level structure of map_1 (b) Detailed structure of gray boxes in (a)

Fig. 5. The structure of map_1. Dotted lines are containment relationships. Solid lines are relationships to the domain and range of mapping objects.

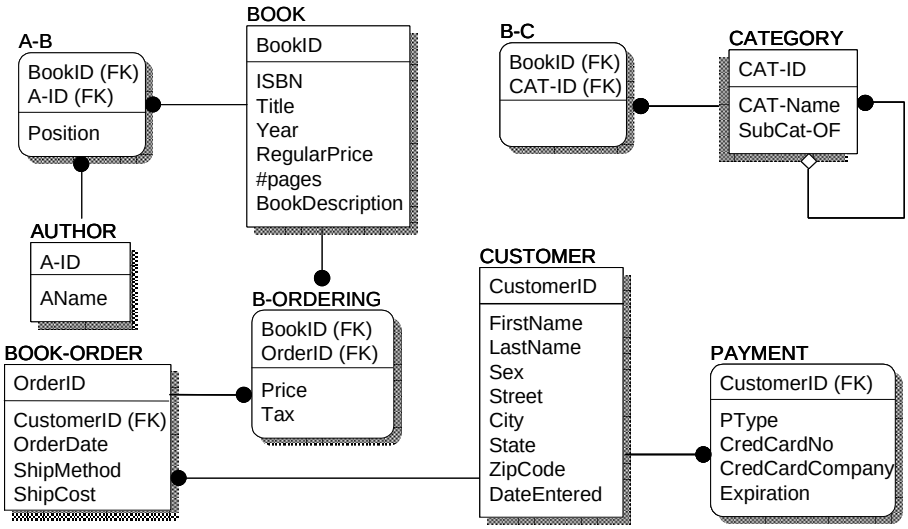


Fig. 6. Relational schema rdb2 (Book Orders)

Table 1. SQL statements defining the semantics of two mappings into dw1

map$_1$ (rdb1 → dw1)	*map*$_3$ (rdb2→dw1)
create view dw1.Sales (OrderID,OrderDetail-ID, CustomerID, PostalCode, ProductID, OrderDate, Quantity, UnitPrice, Discount) **as** **select** O.OrderID, D.OrderDetailID, O.Custo-merID, C.PostalCode, D.ProductID, O.Order-Date, D.Quantity,D.UnitPrice, D.Discount **from** rdb1.Orders O, rdb1.Order-details D, rdb1.Customers C **where** O.OrderID=D.OrderID and O.CustomerID=C.CustomerID **order by** O.OrderId, D.OrderDetailID	**create view** dw1.Sales (OrderID, OrderDetail-ID, CustomerID, PostalCode, ProductID, OrderDate, Quantity, UnitPrice, Discount) **as** **select** O.OrderID, D.BookID, O.CustomerID, C.ZipCode, D.BookID, O.OrderDate, *1*, D.Price, *0* // Default-Settings quantity=1,discount=0 **from** rdb2.Book-orders O, rdb2.B-ordering D, rdb2.Customer C **where** O.OrderID=D.OrderID and O.CustomerID=C.CustomerID **order by** O.OrderId, D.OrderDetailID
create view dw1.Customers (CustomerID, CustomerName, CustomerTypeID, Customer-TypeDescription, PostalCode, State) **as** **select** C.CustomerID, C.CompanyName, C.CustomerID%4, **case** (C.CustomerID%4) **when** 0 **then** 'Excellent' **when** 1 **then** 'Good' **when** 2 then 'Average' **when** 3 **then** 'Poor' **else** 'Average' **end**, C.PostalCode, C. StateOrProvince **from** rdb1.Customers C	**create view** dw1.Customers (CustomerID, … State) **as select** C.CustomerID, *Concatenate (C.FirstName,C.LastName),* C.CustomerID % 4, **case** (C.CustomerID % 4) **when** 0 **then** 'Excellent' **when** 1 **then** 'Good' **when** 2 **then** 'Average' **when** 3 **then** 'Poor' **else** 'Average' **end**, C. ZipCode, C. State **from** rdb2.Customer C
create view dw1.Times (Date, DayOfWeek, Month, Year, Quarter, DayOfYear, Holiday, Weekend , YearMonth, WeekOfYear) **as** **select distinct** O.OrderDate, DateName (dw, D.OrderDate), DatePart(mm ,O.OrderDate), DatePart(yy ,O.OrderDate), DatePart(qq, O.OrderDate), DatePart(dy,O.OrderDate),'N', **case** DatePart(dw,O.OrderDate) **when** (1) **then**'Y' **when** (7) **then** 'Y' **else** 'N' **end**, DateName(month, O.OrderDate) + '_' + DateName(year,O.OrderDate), DatePart(wk,O.OrderDate) **from** rdb1.Orders O	**create view** dw1.Times(Date,…,WeekOfYear) **as select distinct** O.OrderDate, DateName (dw, D.OrderDate), DatePart(mm,O.OrderDate), DatePart(yy ,O.OrderDate), DatePart(qq, O.OrderDate), DatePart(dy,O.OrderDate), 'N', **case** DatePart(dw,O.OrderDate) **when** (1) **then** 'Y' **when** (7) **then** 'Y' **else** 'N' **end**, DateName(month, O.OrderDate) + '_' + DateName(year,O.OrderDate), DatePart(wk,O.OrderDate) **from** rdb2.Book-orders O
create view dw1.Geography (PostalCode, TerritoryID, TerritoryDescription, RegionID, RegionDescription) **as** **select** T.TerritoryID, T.TerritoryID, T.TerritoryDescription, R.RegionID, R.RegionDescription **from** rdb1.Territories T, rdb1.Region R, rdb1.Territory-region TR **where** T.TerritoryID=TR.TerritoryID and TR.RegionID=R.RegionID	**create view** dw1.Geography (PostalCode, … RegionDescription) **as** **select distinct** C.ZipCode, C.ZipCode, *NULL, NULL, NULL* **from** rdb2.Customer C // Where clause dropped because required attributes not existing
create view dw1.Products (ProductID, Pro-ductName, BrandID, BrandDescription) **as** **select** P.ProductID, P.ProductName, B.BrandID, B.BrandDescription **from** rdb1.Brands B, rdb1.Products P **where** B.BrandID=P.BrandID	**create view** dw1.Products(ProductID, …) **as** **select** B.BookID, B.Title, *NULL, NULL* **from** rdb2.Book B // Where clause dropped because required attributes not existing

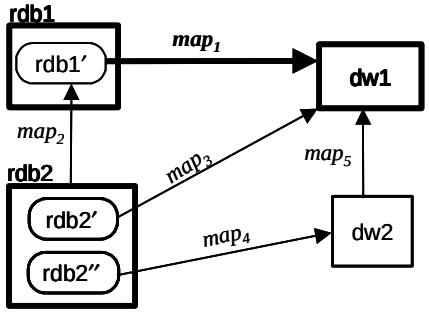
- We chose to write the expressions for map_1 as SQL view definitions, shown in column 1 of Table 1. There is one statement for each of the 5 mapping objects in Fig. 5a (one per table in dw1). To create dw1, simply materialize the views.
- Only 8 out of 13 tables in rdb1 take part in domain(map_1). In addition, only a subset of these tables' attributes are mapped to dw1, as is typical for data warehousing. This is different from other areas, such as schema integration in federated databases, where one strives for complete mappings to avoid information loss.
- Range(map_1) fully covers dw1, since map_1 is the only source of data for dw1.
- The SQL statements in map_1 perform 1:1 attribute mappings (e.g., name substitution and type conversion) and complex transformations involving joins and user-defined functions (in map_1 for date transformations and customer classification). Although all mappings in this example are invertible, this is not true in general, e.g., if aggregate values are derived and mapped to the warehouse.

Suppose we want to integrate a second source into the warehouse. The new source covers book orders and is described by a relational schema rdb2 (see Fig. 6). The integration requires defining a mapping from rdb2 to the existing warehouse schema dw1 and possibly changing dw1 to include new information introduced by rdb2. To simplify the integration task, we want to re-use the existing mappings as much as possible. The extent to which this can be achieved depends on the degree of similarity between rdb2 and rdb1. Some of rdb2's tables and attributes are similar to rdb1 and dw1, but there are also new elements, e.g., on authors and categories. We present two solutions for the integration task.

3.1 First Solution

Figure 7 illustrates the model management steps of our first solution. The elements shown in boldface (rdb1, map_1, dw1, rdb2) are given. A Venn-diagram-like notation is used to show subsets. E.g., rdb1′ ⊆ rdb1 means every row of table rdb1′ is in rdb1.

The first solution exploits the similarities between rdb1 and rdb2 by attempting to re-use map_1 as much as possible. This requires a match between rdb2 and rdb1, to identify which elements of map_1 can be reused for rdb2. The match result is then composed with map_1, thereby reusing map_1 to create a mapping between rdb2 and dw1.



1. rdb1′ = domain(map_1)
2. map_2 = RightOuterMatch(rdb2, rdb1′)
3. map_3 = ApplyFunction(map_2, default-settings) $\bullet_f$ map_1
4. rdb2′ = domain(map_2)
5. rdb2″ = subset of (rdb2 - rdb2′) to be mapped to the warehouse
6. map_4 = user-defined-mapping(rdb2″, dw2)
7. map_5 = Match(dw1, dw2)
8. Merge (dw1, dw2, map_5)

Fig. 7. Sequence of model management operations to integrate a new data source.

Match and RightOuterMatch

For the match between rdb2 and rdb1, it is unnecessary to consider all of schema rdb1, but only the part that is actually mapped to dw1, namely rdb1′ = domain(map_1). (The latter assignment is just a macro for notational convenience. I.e., the program need not construct a physical representation of rdb1′.) This avoids identifying irrelevant rdb1-rdb2 overlaps (e.g., w.r.t. payment attributes) that are not used in the warehouse and thus need not be mapped. In our example, rdb1′ is easy to identify: it simply consists of the rdb1 tables and attributes being used in the SQL statements of map_1.

Object matching is driven by the correspondence table in Table 2, which specifies equivalence of attribute names or attribute expressions. The table consists mostly of 1:1 attribute correspondences (e.g., rdb2.Book.BookID matches rdb1.Products.ProductID, etc.). In one case, two rdb2 attributes are combined: concatenate (rdb2.Customer.FirstName, rdb2.Customer.LastName) matches rdb1.Customers.CompanyName.

Table 2. Correspondence table specifying equivalence of attributes in rdb2 and rdb1′

rdb2	rdb1′
Customer.CustomerID	Customers.CustomerID
Catenate(Customer.FirstName, Customer.LastName)	Customers.CompanyName
Customer.ZipCode	Customers.PostalCode
Customer.State	Customers.StateOrProvince
Book-Orders.OrderID	Orders.OrderID
Book-Orders. CustomerID	Orders.CustomerID
Book-Orders.OrderDate	Orders.OrderDate
B-Ordering.OrderID	Order-Details.OrderID
B-Ordering.BookID	Order-Details.ProductID
B-Ordering.Price	Order-Details.UnitPrice
B-Ordering.BookID	Order-Details.OrderDetailID
Book.BookID	Products.ProductID
Book.Title	Products.ProductName

We want to compose the result of the match operation between rdb2 and rdb1′ with map_1. However, not all rdb1′ elements have matching counterparts in rdb2, i.e., rdb1′ is a proper superset of range(Match(rdb2, rdb1′)). For instance, rdb2 has no equivalent of the Quantity and Discount attributes in the Orders table or of the Brands and Region tables, which are in rdb1′. Without this information, three of the five SQL statements in map_1 cannot be used, although only a few of the required attributes are missing.

To ensure that the match captures all of rdb1′, we use a RightOuterMatch of rdb2 and rdb1′, i.e., map_2 = RightOuterMatch(rdb2, rdb1′) in step (2) of Fig. 7. We explain in the next section what to do with objects in map_2 that have an empty domain.

An alternative strategy is to perform a RightOuterMatch of rdb2 and rdb1. This would allow the match to exploit surrounding structure not present in rdb1′, but produces a larger match result that would need to be manipulated later, an extra expense.

Composition

The next step is to compose map_2 with map_1 to achieve the desired mapping, map_3, from rdb2 to dw1. There are several issues regarding this composition. First, it needs a to work for mapping objects that have set-valued domains. For example, the last

Create View statement in map_1 represents a mapping object m_5 in Fig. 5 with multiple attributes for each of the tables in its domain. When composing map_2 with map_1, we need "enough" mapping objects in map_2 to cover domain(m), for each mapping object m in map_1. This is analogous to m_{1a} and m_{1b} covering A and B in Fig. 2b.

Second, the composition must create an expression in each mapping object that combines the expressions in the mapping objects it is composing. This requires substituting objects in the mapping expressions (i.e., SQL statements) of map_1. That is, it replaces each rdb1′ attribute and its associated table by its rdb2 counterpart defined by map_2. The right column of Table 1 shows the resulting SQL statements that make up map_3, which can automatically be generated in this way. For example, in the Sales query, since map_2 maps B-Ordering.BookID in rdb2 to Order-Details.OrderDetailID in rdb1, it substituted D.BookID for D.OrderDetailID in the Select clause.

Third, since map_2 is the result of a RightOuterMatch, we need to deal with each object m_2 in map_2 where domain(m_2) is empty. The desired outcome is to modify the SQL expression from map_1 to substitute either NULL or a user-defined value for the item in range(m_2). One way to accomplish this is to extend map_2 by adding dummy objects with all the desired default values (e.g., "NULL") to rdb2, and adding a dummy object to domain(m_2) for each m_2 in map_2 where domain(m_2) is empty. The latter can be done by using the model management ApplyFunction operation to apply the function "set domain(m_2) = {dummy-object} where domain(m_2) = ∅" to map_2. This makes the substitution of default values for range(m_2) automatic (step (3) of Fig. 7).

As shown in the first Create View of Table 1, we use default values 1 and 0 for attributes Quantity and Discount (resp.), which were not represented in rdb2. All other unmatched attributes from rdb1′ are replaced by NULL. Note that this allows two queries to be simplified (eliminating joins in the Geography and Products queries).

While the query substitutions implementing the composition are straightforward in our example, problems arise if more complex match transformations have to be incorporated, such as aggregates. This is because directly replacing attributes with the equivalent aggregate expression can lead to invalid SQL statements, e.g., by using an aggregate expression within a Where clause. Substitution is still possible, but requires more complex rules than simple variable substitution.

Re-using existing transformations may not always be desirable, as these transformations may only be meaningful for a specific source. For instance, the customer mapping entails specific expressions for customer classification (second SQL statement in Table 1), which may not be useful for a different set of customers. Such situations could be handled by allowing the user to define new transformations.

Final Steps

The final integration steps check whether any parts of rdb2 not covered by the previous steps should be incorporated into the warehouse. In our example one might want to add authors as a new dimension to the data warehouse. Determining the parts to be integrated obviously cannot be done automatically. Hence, we require a user-defined specification of the additional mapping (step (6) of Fig. 7). Merging the resulting warehouse elements with the existing schema dw1 may require combining tables in a preliminary Match (step (7)) followed by the actual Merge (step (8)).

Observations

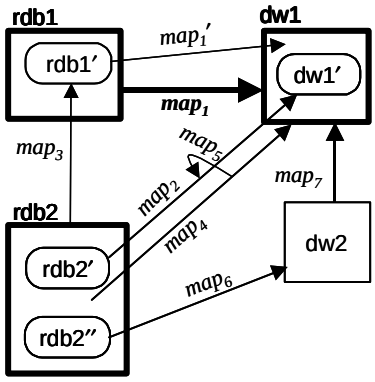
Obviously, Match and Compose are the key operations in the proposed solution to achieve a re-use of an existing mapping. The use of SQL as the expression language requires that these operations support mapping objects with set-valued domains and ranges. The use of RightOuterMatch in combination with ApplyFunction to provide default values allowed us to completely re-use the existing mapping

The power and abstraction level of the model management operations resulted in a short solution program, a huge productivity gain over the specification and programming work involved with current warehouse tools. This is especially remarkable given the use of generic operations, not tailored to data warehousing. The main remaining manual work is in supporting the Match operations (although its implementation can at least partially be automated) and in specifying new mapping requirements that cannot be derived from the existing schemas and mappings. Of course, more effort may be needed at the data instance level for data cleaning, etc.

3.2 Alternative Solution

An alternative solution to integrate rdb2 is illustrated in Fig. 8. In contrast to the previous solution, it first identifies which parts of rdb2 can be directly matched with the warehouse schema. It tries to re-use the existing mapping map_1 only for the remaining parts of rdb2.

In step (1), we thus start by matching rdb2 with the warehouse schema dw1, resulting in a mapping map_2 that identifies common tables and attributes of rdb2 and dw1. This gives a direct way to populate range(map_2), called dw1′, by copying data from domain(map_2). Note that we do not have a RightOuterMatch since we can expect that only some parts of dw1 can be derived from rdb2.



1. map_2 = Match (rdb2, dw1)
2. rdb2′ = domain (map_2)
3. dw1′ = range (map_2)
4. map_1' = Copy(map_1)
5. ApplyFunction(map_1', "for each x in range(m_1), if x is in dw1′ … ") // see text for details
6. map_3 = RightOuterMatch (rdb2, domain (map_1'))
7. map_4 = ApplyFunction(map_3, defaults) $\bullet_f map_1'$
8. map_5 = Match(map_4, map_2)
9. Merge (map_2, map_4, map_5)
10. rdb2″ = subset of (rdb2 - rdb2′ - domain(map_4)) to be mapped to warehouse
11. map_6 = user-defined mapping (rdb2″, dw2)
12. map_7 = Match(dw1, dw2)
13. Merge (dw1, dw2, map_7)

Fig. 8. Alternative sequence of model management operations to integrate a new source

For those parts of the warehouse schema that cannot be matched directly with rdb2 (i.e., dw1 – dw1′), we try to re-use the existing mapping map_1. We therefore create a copy map_1' of map_1 and, in step(5), use ApplyFunction to remove objects from the range of map_1' that are in dw1′. That is, for each object m_1 in map_1', "for each x in range(m_1), if x is in dw1′ and not part of a primary key, then remove x from range(m_1) and from the SQL statement associated with m_1." We avoid deleting primary key

attributes to ensure the mapping produced in steps (6)-(7) can be merged with existing tables in steps (8)-(9). Deleting *x* from the SQL statement involves deleting *x* from the Create View and deleting the corresponding terms of rdb1 from the Select clause, but not, if present, from the Where clause, since its use there indicates that *x* is needed to define a relevant restriction or join condition. After all such *x* are deleted from the statement, additional equivalence-preserving simplifications of the statement may be possible. In particular, if a dw1 table *T* is completely in dw1′, then the map_1 SQL statement for *T* will be eliminated from the result mapping map_1'. The model management algebra needs to be structured in a way that allows the SQL inferencing plug-in to make such modifications to the SQL statement.

Next, we match rdb2 with the domain of map_1', called rdb1′ (step (6) of Fig. 8). It is not sufficient to perform the match for rdb2 – rdb2′, even though rdb2′ has already been mapped to dw1. This is because for some objects m_1 in map_1', there may be an object *x* in domain(m_1) that maps to an object in rdb2′ but not to one in rdb2 – rdb2′. There is no problem using *x* as input to m_1 as well as mapping *x* directly to dw1 using map_2. As in the first solution, we use RightOuterMatch to ensure the resulting map includes all elements of domain(map_1').

As in the previous solution, we use ApplyFunction to add default mappings for elements of domain(map_1') that do not correspond to an element of rdb2 via map_3. And then we compose map_3 and map_1', resulting in map_4 (step (7)).

The mapping between rdb2 and dw1 computed so far consists of map_2 and map_4, which we match and merge in steps (8) and (9). If map_2 and map_4 populate different tables of dw1 then Merge is a simple union. However, if there is a table that they both populate, more work is needed; hence the need for the preliminary Match forming map_5. For tables common to both maps, the two Create View statements need to be combined. This may involve non-trivial manipulation of SQL w.r.t. key columns.

As in steps (5)-(8) of the first solution, there may be a user-defined mapping for other rdb2 elements to add to the warehouse (steps (10)-(13) in Fig. 8). If there is any overlap with previous maps, then these mappings too must be merged with other Create View statements.

In our example, in step (1) we can directly match the dw1 tables Products, Customers and Geography with rdb2 tables Book and Customer as only 1:1 attribute relationships are involved. Among other things, this avoids the unwanted re-use of CustomerTypeDescription, applied for rdb1. For the two other warehouse tables, Time and Sales, we match rdb2 with rdb1 in step (6) to re-use the corresponding mapping expressions in map_1, particularly the time transformations and join query. We thus have two mappings referring to different tables; their union in step (9) provides the complete mapping from rdb2 to dw1.

Alternatively, instead of deriving the Sales table in steps (6)-(7), we could match three of its attributes, OrderID, CustomerID, and OrderDate, with table Book-Orders when creating map_2 in step (1), and using map_1 for the remaining attributes in steps (6)-(7). We thus would use ApplyFunction in step (5) to eliminate the three attributes from the Create View and Select clauses of the Sales statement in map_1 and keep the reduced query in map_1' (together with the Time query). We would leave the OrderID and CustomerID attributes in the Where clause of the modified Sales query in step (5) to perform the required joins. We thus obtain these two mapping statements for Sales:

Map$_2$:

```
create view dw1.Sales (OrderID, CustomerID, OrderDate) as
select      B.OrderID, B.CustomerID, B.OrderDate
from        rdb2.Book-Orders B
```

Map$_4$:

```
create view dw1.Sales1 (OrderID, OrderDetailID, PostalCode, ProductID, Quantity,
                        UnitPrice, Discount) as
select    D.OrderID, D.BookID, C.ZipCode, D.BookID, 1, D.Price, 0
from      rdb2.Book-Orders O, rdb2.B-ordering D, rdb2.Customer C
where     O.OrderID = D.OrderID and O.CustomerID = C.CustomerID
order by O.OrderId, D.BookID
```

Notice that we retain OrderID in Sales1, so we can match *map$_2$* and *map$_4$* in step (8) to drive a Merge in step (9). The result corresponds to the SQL statement in the right column of row 1 in Table1.

Observations

This approach applied similar steps to the first solution, in particular for RightOuter-Match, RightCompose and ApplyFunction. Its distinguishing feature is the partial re-use of an existing mapping, which is likely to be more often applicable than a complete re-use. The new source was matched against both the warehouse and the first source, leading to the need to merge mappings. The solution can be generalized for more than one preexisting data source. In this case, multiple mappings to the warehouse schema may be partially re-used for the integration of a new data source.

4 Data Warehouse Scenario 2: Adding a New Data Mart

The usage of model management operations described in Section 3 seems to be typical, at least for data warehouse scenarios. To illustrate these recurring patterns, we briefly consider a second scenario. We assume a given star schema dw, an existing data mart dm1 and a mapping *map$_1$* from dw to dm1, where range(*map$_1$*) = dm1. We want to add a second data mart dm2. The task is to determine the mapping from dw to dm2. Obviously this mapping must be complete with respect to dm2.

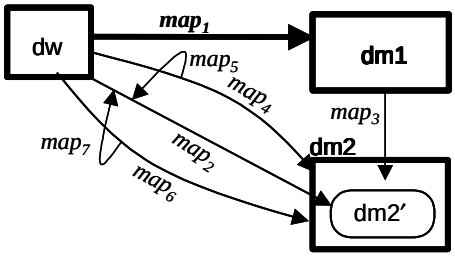
To solve the problem we can use solution patterns similar to Section 3, allowing us to give a compact description. Three possibilities are illustrated in Fig. 9. Solution 1 is the simplest approach; just apply RightOuterMatch to dw and dm2. This is possible if the two schemas differ little in structure, e.g., if dm2 is just a subset of dw.

Solution 2 is useful if some but not all of dm2 can be matched with dw. We first match dw with dm2 and then match the unmatched parts of dm2 with dm1 to re-use the associated parts of *map$_1$*. Remaining parts of dm2 are derived by a user-specified mapping *map$_6$* and then merged in.

Solution 3 tries to maximally re-use the existing mapping *map$_1$* as in Section 3.1. This is appropriate if the data marts are similarly structured and *map$_1$* contains complex transformations that are worth re-using. We first compose *map$_1$* with the match of dm1 and dm2. The rest of dm2 not covered by this mapping is matched with dw. Any remaining dm2 elements are derived by a user-specified mapping *map$_6$*.

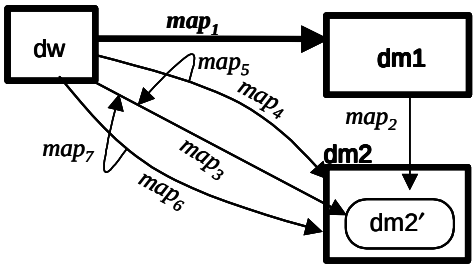
Solution 1: map_2 = RightOuterMatch (dw, dm2)

Solution 2: (no OuterMatch, but partial re-use of map_1)



1. map_2 = Match(dw, dm2)
2. dm2′ = range(map_2)
3. map_3 = Match(dm1, dm2 - dm2′)
4. $map_4 = map_1 \bullet_f map_3$
5. map_5 = Match(map_4 , map_2)
6. Merge(map_2, map_4 , map_5)
7. map_6 = user-defined-mapping(dw, dm2 - dm2′ - range(map_3))
8. map_7 = Match(map_2 , map_6)
9. Merge (map_2, map_6 , map_7)

Solution 3: (maximal re-use of map_1)



1. map_2 = Match(dm1, dm2)
2. $map_3 = map_1 \bullet_f map_2$
3. dm2′ = range(map_2)
4. map_4 = Match(dw, dm2 - dm2′)
5. map_5 = Match(map_4 , map_3)
6. Merge(map_3, map_4 , map_5)
7. map_6 = user-defined-mapping(dw, dm2 - dm2′- range(map_3))
8. map_7 = Match(map_6 , map_3)
9. Merge (map_3, map_6 , map_7)

Fig. 9. Three alternatives to add a new data mart

5. Conclusions

We evaluated the application of a generic model management approach for two data warehouse scenarios which used relational sources, star schemas, and SQL as an expression language for mappings. We devised several alternatives for solving typical mapping problems in a generic way: integrating a new data source and adding a new data mart. The solutions re-use existing mappings to a large extent and combine model operators in different ways. User interaction may be required to provide semantic equivalence information for match operations and to specify new mapping requirements that cannot be derived from existing models (mappings, schemata).

The study has deepened our understanding of two key operators: Match and Compose. In particular, we introduced the notion of OuterMatch. We showed the need for composition semantics to cover mapping objects with set-valued domains and ranges. We also proposed a general way to provide default values by employing the ApplyFunction operation. We expect this idiom will be commonly used when composing mappings.

We would like the expression manipulation associated with Compose to be managed by a module that can plug into the algebraic framework. One such module would handle SQL. The examples in this paper show what such a module must be able to do.

We found the model management notation to be a useful level of abstraction at which to consider design alternatives. By focusing on mappings as abstract objects,

the designer is encouraged to think about whether a mapping is total, is onto, has a set-valued domain, can be composed with another mapping, and has a range entirely contained within the set of interest. In this paper's examples, at least, these were the main technical factors in deriving the solution. Moreover, we introduced a Venn-diagram-like notation, which enables quick comparisons between design choices, such as Figures 7 and 8 and Solutions 2 and 3 of Fig. 9. These examples show that the notation is a compact representation of each solution's approach, highlighting how the approaches differ.

Altogether, the study has provided evidence of the usefulness of a general model management approach to manage models and mappings in a generic way. Furthermore, the considered level of detail suggests that model management is more than a vision but likely to be implementable in an effective way.

Acknowledgments

We thank Alon Levy, Jayant Madhavan, Sergey Melnik, and Rachel Pottinger for many suggested improvements to the paper. We are also grateful to the Microsoft SQL Server group for providing the schemas used in the examples.

References

1. Bernstein, P.A.: Panel: Is Generic Metadata Management Feasible? VLDB 2000
2. Bernstein, P.A., Levy, A., Pottinger, R.: A Vision for Management of Complex Models. MSR-TR-2000-53, http://www.research.microsoft.com/pubs/, June 2000
3. Doan, AH., Domingos, P., Levy, A.: Learning Source Descriptions for Data Integration. Proc. WebDB 2000, pp. 81-92
4. Jannink, J., Mitra, P., Neuhold, E., Pichai, S., Studer, R., Wiederhold, G.: An Algebra for Semantic Interoperation of Semistructured Data. Proc. 1999 IEEE Knowledge and Data Engineering Exchange Workshop (KDEX'99), Nov. 1999.
5. Li, W., Clifton, C.: Semantic Integration in Heterogeneous Databases using Neural Networks. Proc. VLDB94
6. Li, W., Clifton, C.: SEMINT: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. Data and Knowledge Engineering, 33 (1), 2000
7. Miller, R., Ioannidis, Y.E., Ramakrishnan, R.: Schema Equivalence in Hetereogeneous Systems: Bridging Theory and Practice. Information Systems 19(1), 3-31, 1994
8. Milo, T., Zohar, S.: Using Schema Matching to Simplify Heterogeneous Data Translation. Proc. VLDB98
9. Mitra, P., Wiederhold , G., Jannink, J.: Semi-automatic Integration of Knowledge Sources. Proc. of Fusion '99, Sunnyvale, USA, July 1999
10. Mitra, P., Wiederhold, G., Kersten, M.: A Graph-Oriented Model for Articulation of Ontology Interdependencies; Proc. Extending DataBase Technologies, EDBT 2000, LNCS Springer Verlag.
11. Mylopoulos, J., Motschnig-Pitrik, R.: Partitioning Information Bases with Contexts. Proc. 3rd CoopIS, Vienna, pp. 44-54, May 1995.
12. Palopoli, L., Sacca, D., Ursino, D.: Semi-automatic, semantic discovery of properties from database schemas. Proc. IDEAS, 1998.
13. Palopoli, L., Sacca, D., Ursino, D.: An automatic technique for detecting type conflicts in database schemas. Proc. CIKM, 1998
14. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., Lum, V.Y.: EXPRESS: A Data EXtraction, Processing and REStructuring System. ACM TODS 2,2: 134-174, 1977.