Schema Evolution in Data Warehousing Environments – A Schema Transformation-Based Approach

Hao Fan and Alexandra Poulovassilis

School of Computer Science & Information Systems, Birkbeck College,
University of London, Malet Street, London WC1E 7HX
{hao,ap}@dcs.bbk.ac.uk

Abstract. In heterogeneous data warehousing environments, autonomous data sources are integrated into a materialised integrated database. The schemas of the data sources and the integrated database may be expressed in different modelling languages. It is possible for either the data source schemas or the warehouse schema to evolve. This evolution may include evolution of the schema, or evolution of the modelling language in which the schema is expressed, or both. In such scenarios, it is important for the integration framework to be evolvable, so that the previous integration effort can be reused as much as possible. This paper describes how the AutoMed heterogeneous data integration toolkit can be used to handle the problem of schema evolution in heterogeneous data warehousing environments. This problem has been addressed before for specific data models, but AutoMed has the ability to cater for multiple data models, and for changes to the data model.

1 Introduction

With the increasing use of the Internet in distributed applications, data warehouses may integrate data from remote, heterogeneous, autonomous data sources. The heterogeneity of these data sources has two aspects, heterogeneous data expressed in different data models, called *model heterogeneity* [10], and heterogeneous data within different data schemas expressed in the same data model, called *schema heterogeneity* [10, 18]. The common approach to handling model heterogeneity is to use a single conceptual data model (CDM) for the data transformation/integration. Each data source has a wrapper for translating its schema and data into the CDM. The warehouse schema is derived from these CDM schemas by means of view definitions, and is expressed in the same modelling language as them. With this approach, since they are both high-level conceptual data models, semantic mismatches may occur between the CDM and a source data model, and there may be a loss of information between them. Moreover, if a data source schema changes, it is not straightforward to evolve the view definitions of the warehouse schema.

Lakshmanan *et al* [11] argue that a uniform framework for schema integration and schema evolution is both desirable and possible, and this is our view

P. Atzeni et al. (Eds.): ER 2004, LNCS 3288, pp. 639–653, 2004.

also. They define a higher-order logic language, SchemaSQL, which handles both data integration and schema evolution in relational multi-database systems. In contrast, our approach uses a simple set of schema transformation primitives, augmented with a functional query language, both of which are uniformly applicable to multiple data models. Other previous work on schema evolution, e.g. [1–4], has also presented approaches in terms of just one data model.

AutoMed is a heterogeneous data transformation and integration system which offers the capability to handle data integration across multiple data models[1]. In [7] we discussed how AutoMed metadata can be used to express the schemas and the cleansing, transformation and integration processes in heterogeneous data warehouse environments, supporting both schema heterogeneity and model heterogeneity. We discussed how this metadata can be used to populate and incrementally maintain the warehouse, and any data marts derived from it, and also to trace the lineage of data in the warehouse or the data marts. It is clearly advantageous to be able to reuse this kind of metadata if a schema evolves. In this paper we show how this can be achieved.

Earlier work [16] has shown how the AutoMed framework readily supports schema evolution in *virtual* data integration scenarios. In this paper we address the problem of schema evolution in *materialised* data integration scenarios, including both evolution of a source schema and of the warehouse schema, and also the impact on any data marts derived from the warehouse. This scenario is more complex than with virtual data integration, since both schemas and materialised data may be affected by an evolution.

The outline of the paper is as follows. Section 2 gives an overview of the AutoMed framework. Section 3 describes how AutoMed transformations can be used to express a schema evolution if either the schema changes, or the data model changes, or both. Section 4 describes the actions that are taken in order to evolve these transformations and the materialised data if the warehouse schema or a local schema evolves. Section 5 discusses the benefits of our approach and gives our concluding remarks.

2 Overview of AutoMed

AutoMed supports a low-level hypergraph-based data model (HDM). Higher-level modelling languages are defined in terms of this HDM. For example, previous work has shown how relational, ER, OO [15], XML [21], flat-file [5] and multidimensional [7] data models can be so defined. An HDM schema consists of a set of nodes, edges and constraints, and each modelling construct of a higher-level modelling language is specified as some combination of HDM nodes, edges and constraints. For any modelling language $\mathcal{M}$ specified in this way (via the API of AutoMed's Model Definitions Repository [5]), data source wrappers translate data source schemas expressed in $\mathcal{M}$ into their AutoMed representation, without loss of information. AutoMed also provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$. In particular, for

[1] See `http://www.doc.ic.ac.uk/automed/`

every construct of $\mathcal{M}$ there is an add and a delete primitive transformation which add to/delete from a schema an instance of that construct. For those constructs of $\mathcal{M}$ which have textual names, there is also a rename primitive transformation.

In AutoMed, schemas are incrementally transformed by applying to them a sequence of primitive transformations $t_1, \ldots, t_r$. Each primitive transformation adds, deletes or renames just one schema construct. Thus, intermediate schemas may contain constructs of more than one modelling language.

Each add or delete transformation is accompanied by a query specifying the extent of the new or deleted construct in terms of the rest of the constructs in the schema. This query is expressed in a functional query language IQL (see Section 2.1). Also available are contract and extend transformations which behave in the same way as add and delete except that they indicate that their accompanying query may only partially construct the extent of the new/removed schema construct. Moreover, their query may just be the constant Void, indicating that the extent of the new/removed construct cannot be derived even partially, in which case the query can be omitted.

We term a sequence of primitive transformations from one schema S_1 to another schema S_2 a transformation *pathway* from S_1 to S_2, denoted $S_1 \rightarrow S_2$. All source, intermediate, and integrated schemas, and the pathways between them, are stored in AutoMed's Schemas & Transformations Repository [5].

The queries present within transformations that add or delete schema constructs mean that each primitive transformation t has an automatically derivable *reverse transformation*, $\bar{t}$. In particular, each add/extend transformation is reversed by a delete/contract transformation with the same arguments, while each rename transformation is reversed by swapping its two arguments. Thus, AutoMed is a **both-as-view** (BAV) data integration system. As discussed in [17], BAV subsumes the global-as-view (GAV) and local-as-view (LAV) approaches [13], since it is possible to extract a definition of each global schema construct as a view over source schema constructs, and it is also possible to extract definitions of source schema constructs as views over the global schema. We refer the reader to [9] for details of AutoMed's GAV and LAV view generation algorithms.

Figure 1 illustrates the general integration scenario with AutoMed. Each data source is described by a local schema LS_i. Each LS_i is first conformed into a schema CS_i (which may or may not be expressed in the same modelling language as LS_i) by means of a transformation pathway T_i. Not all of the information within a local schema LS_i need be transferred into the global schema and this is asserted by means of contract transformation steps within T_i. Conversely, there may be information within the global schema which is not semantically derivable from LS_i, and this is asserted by the pathway from CS_i to a 'union-schema' US_i which consists of the necessary extend transformations[2].

All the union schemas $US_1, \ldots, US_n$ are syntactically identical and this is asserted by creating a sequence of id transformations between each pair US_i and US_{i+1}, of the form id $US_i : c$ $US_{i+1} : c$ for each schema construct c. An id transformation signifies the semantic equivalence of syntactically identical

[2] If there are none, then this pathway is empty and CS_i and US_i are the same schema

constructs in different schemas. The transformation pathways containing these id transformations are automatically generated by the AutoMed software. An arbitrary one of the US_i (US in Figure 1) can then be selected for further transformation into the global schema GS (by the pathway T_u in Figure 1). The extent of each construct c in a union schema US_i is equal to the bag-union of the extent of c in all union schemas $US_1, \ldots, US_n$. That is, id is interpreted as *bag union* by AutoMed's view generation functionality.

In a virtual data integration scenario, there is no materialised data associated with any of the schemas apart from the LS_i. In a data warehousing scenario, as illustrated in Figure 1, we assume that $CS_1, \ldots, CS_n$ are fully materialised and consist of the detailed data of the warehouse. This detailed data is further augmented with the necessary summary views by the transformations in the pathway T_u, and we assume that these summary views are materialised in the database GD. It would also be possible to partially or fully materialise more of the intermediate schemas in the network, or to not materialise $CS_1, \ldots, CS_n$ and to fully materialise GS instead. Our techniques in this paper easily generalise to these alternatives.

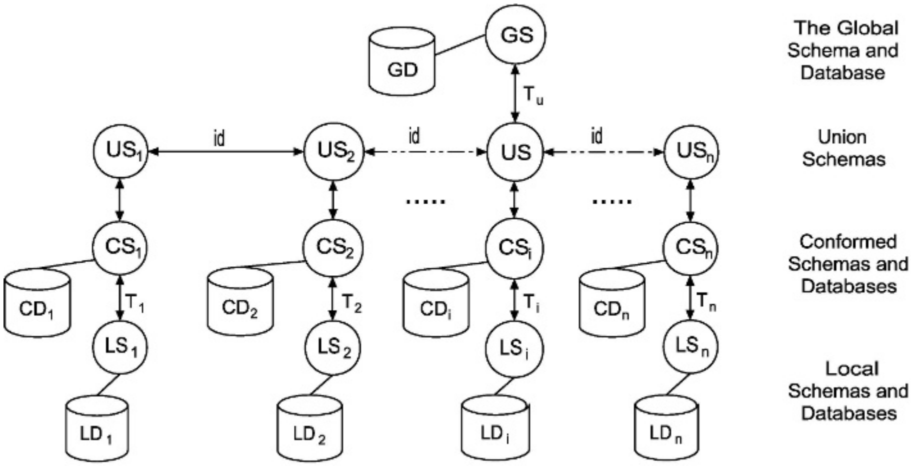


Fig. 1. Materialised Data Integration in AutoMed

For the purposes of this paper, we assume that all the LS_i and LD_i have been extracted from the original data sources and the data in the LD_i has been cleansed. The data cleansing process can also be expressed using AutoMed transformations – this is discussed in [7] and we do not consider it further here. See also that paper for some examples of how AutoMed transformations can express structural and representational changes to schemas and data.

We also assume here that there are no contract steps in the pathways T_i, i.e. that all the information in each LS_i will be transferred to CS_i and hence to US_i. This implies no loss of flexibility as each LS_i will be precisely that extract of the original data source schema whose associated data is to be transferred into the warehouse.

2.1 The IQL Query Language

IQL is a comprehensions-based functional query language[3]. Such languages subsume query languages such as SQL and OQL in expressiveness [6]. IQL supports several primitive operators for manipulating lists. The list append operator, `++`, concatenates two lists together. The `distinct` operator removes duplicates from a list and the `sort` operator sorts a list. The `--` operator takes two lists and subtracts each member of the second list from the first e.g. [1,2,3,2,4]--[4,4,2,1] = [3,2]. The `fold` operator applies a given function `f` to each element of a list and then 'folds' a binary operator `op` into the resulting values. It is defined recursively as follows, where `(x:xs)` denotes a list with head `x` and tail `xs`:

```
fold f op e [] = e
fold f op e (x:xs) = (f x) op (fold f op e xs)
```

Other IQL list manipulation operators are defined using `fold` together with IQL's set of built-in operators and its support of `lambda` abstractions. For example, the IQL functions `sum` and `count` are equivalent to SQL's SUM and COUNT aggregation functions and are defined as

```
sum xs = fold (id) (+) 0 xs
count xs = fold (lambda x.1) (+) 0 xs
```

We also have

```
min xs = fold (id) lesser maxNum xs
max xs = fold (id) greater minNum xs
```

assuming constants `maxNum` and `minNum` and the following functions `lesser` and `greater`:

```
greater = lambda x.lambda y.if (x > y) then x else y
lesser  = lambda x.lambda y.if (x < y) then x else y
```

The function `flatmap` applies a list-valued function `f` to each member of a list `xs` and is defined in terms of `fold`:

```
flatmap f xs = fold f (++) [] xs
```

`flatmap` can in turn be used to define selection, projection and join operators and, more generally, comprehensions. For example, the following comprehension iterates through a list of students and returns those students who are not members of staff:

```
[x | x <- <<student>>; not (member <<staff>> x)]
```

and it translates into:

```
flatmap (lambda x.if (not (member <<staff>> x))
             then [x] else []) <<student>>
```

Grouping operators are also definable in terms of `fold`. In particular, the operator `group` takes as an argument a list of pairs `xs` and groups them on their first component, while `gc aggFun xs` groups a list of pairs `xs` on their first component and then applies the aggregation function `aggFun` to the second component.

[3] We refer the reader to [8] for details of IQL

There are several algebraic properties of IQL's operators that we can use in order to incrementally compute materialised data and to reason about IQL expressions, specifically for the purposes of this paper in a schema/data evolution context (note that the algebraic properties of `fold` below apply to all the operators defined in terms of `fold`):

(a) `e ++ [] = [] ++ e = e, e -- [] = e, [] -- e = [],`
`distinct [] = sort [] = []`
for any list-valued expression `e`. Since `Void` represents a construct for which no data is obtainable from a data source, it has the semantics of the empty list, and thus the above equivalences also hold if `Void` is substituted for `[]`.

(b) `fold f op e [] = fold f op e Void = e`, for any `f, op, e`

(c) `fold f op e (b1 ++ b2) = (fold f op e b1) op (fold f op e b2)`
for any `f, op, e, b1, b2`. Thus, we can always incrementally compute the value of fold-based functions if collections expand.

(d) `fold f op e (b1 -- b2) = (fold f op e b1) op' (fold f op e b2)`
provided there is an operator `op'` which is the inverse of `op` i.e. such that `(a op b) op' b = a` for all `a,b`. For example, if `op` = `+` then `op'` = `-`, and thus we can always incrementally compute the value of aggregation functions such as `count`, `sum` and `avg` if collections contract. Note that this is not possible for `min` and `max` since `lesser` and `greater` have no inverses. Although IQL is list-based, if the ordering of elements within lists is ignored then its operators are faithful to the expected bag semantics, and within AutoMed we generally do assume bag semantics. Under this assumption,
`(xs ++ ys) -- ys = xs`
for all `xs,ys` and thus we can incrementally compute the value of `flatmap` and all its derivative operators if collections contract[4].

2.2 An Example

We will use schemas expressed in a simple relational data model and a simple XML data model to illustrate our techniques. However, we stress that these techniques are applicable to schemas defined in *any* data modelling language that has been specified within AutoMed's Model Definitions Repository.

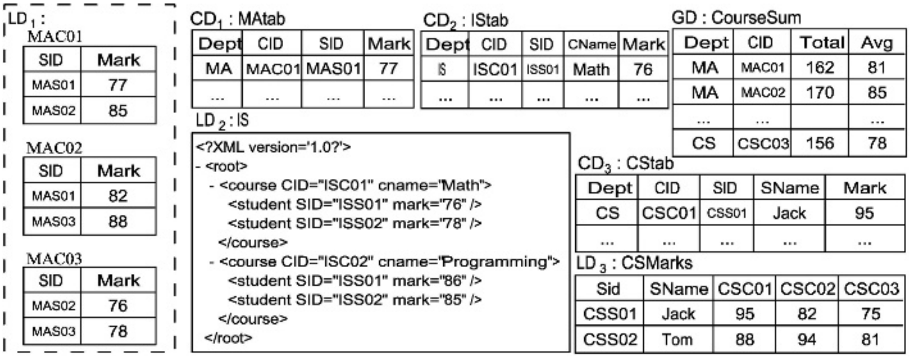
In the simple relational model, there are two kinds of schema construct: Rel and Att. The extent of a Rel construct $\langle\langle \mathsf{R} \rangle\rangle$ is the projection of the relation R onto its primary key attributes $k_1, ..., k_n$. The extent of each Att construct $\langle\langle \mathsf{R}, \mathsf{a} \rangle\rangle$ where a is an attribute (key or non-key) of R is the projection of relation R onto $k_1, ..., k_n, a$. For example, the schema of table `MAtab` in Figure 2 consists of a Rel construct $\langle\langle \mathsf{MAtab} \rangle\rangle$, and four Att constructs $\langle\langle \mathsf{MAtab}, \mathsf{Dept} \rangle\rangle$, $\langle\langle \mathsf{MAtab}, \mathsf{CID} \rangle\rangle$, $\langle\langle \mathsf{MAtab}, \mathsf{SID} \rangle\rangle$, and $\langle\langle \mathsf{MAtab}, \mathsf{Mark} \rangle\rangle$. We refer the reader to [15] for an encoding of a richer relational data model, including the modelling of constraints.

In the simple XML data model, there are three kinds of schema construct: Element, Attribute and NestSet. The extent of an Element construct $\langle\langle \mathsf{e} \rangle\rangle$ consists

[4] The `distinct` operator can also be used to obtain set semantics, if needed

of all the elements with tag e in the XML document; the extent of each Attribute construct $\langle\langle \mathsf{e}, \mathsf{a} \rangle\rangle$ consists of all pairs of elements and attributes x, y such that element x has tag e and has an attribute a with value y; and the extent of each NestSet construct $\langle\langle \mathsf{p}, \mathsf{c} \rangle\rangle$ consists of all pairs of elements x, y such that element x has tag p and has a child element y with tag c. We refer the reader to [21] for an encoding of a richer model for XML data sources, called XMLDSS, which also captures the ordering of children elements under parent elements and cardinality constraints. That paper gives an algorithm for generating the XMLDSS schema of an XML document. That paper also discusses a unique naming scheme for Element constructs so as to handle instances of the same element tag occurring at multiple positions in the XMLDSS tree.

Figure 2 illustrates the integration of three data sources *LD1*, *LD2*, and *LD3*, which respectively store students' marks for three departments MA, IS and CS.



LD_1 :

MAC01

SID	Mark
MAS01	77
MAS02	85

MAC02

SID	Mark
MAS01	82
MAS03	88

MAC03

SID	Mark
MAS02	76
MAS03	78

CD_1 : MAtab

Dept	CID	SID	Mark
MA	MAC01	MAS01	77
...	...	...	...

CD_2 : IStab

Dept	CID	SID	CName	Mark
IS	ISC01	ISS01	Math	76
...	...	...	...	...

GD : CourseSum

Dept	CID	Total	Avg
MA	MAC01	162	81
MA	MAC02	170	85
...	...		...
CS	CSC03	156	78

LD_2 : IS

```
<?XML version='1.0?'>
- <root>
  - <course CID="ISC01" cname='Math">
      <student SID="ISS01" mark="76" />
      <student SID="ISS02" mark="78" />
    </course>
  - <course CID="ISC02" cname='Programming">
      <student SID="ISS01" mark="86" />
      <student SID="ISS02" mark="85" />
    </course>
  </root>
```

CD_3 : CStab

Dept	CID	SID	SName	Mark
CS	CSC01	CSS01	Jack	95
...	...	...	...	...

LD_3 : CSMarks

Sid	SName	CSC01	CSC02	CSC03
CSS01	Jack	95	82	75
CSS02	Tom	88	94	81

Fig. 2. An example integration

Database *LD1* for department MA has one table of students' marks for each course, where the relation name is the course ID. Database *LD2* for department IS is an XML file containing information of course IDs, course names, student IDs and students' marks. Database *LD3* for department CS has one table containing one row per student, giving the student's ID, name, and mark for the courses CSC01, CSC02 and CSC03. CD_1, CD_2, and CD_3 are the materialised conformed databases for each data source. Finally, the global database *GD* contains one table CourseSum(<u>Dept</u>,<u>CID</u>,Total,Avg) which gives the total and average mark for each course of each department. Note that the virtual union schema *US* (not shown) combines all the information from all the conformed schemas and consists of a virtual table Details(<u>Dept</u>,<u>CID</u>,<u>SID</u>,CName,SName,Mark).

The following transformation pathways express the schema transformation and integration processes in this example. Due to space limitations, we have not given the remaining steps for deleting/contracting the constructs in the source schema of each pathway (note that this 'growing' and 'shrinking' of schemas is characteristic of AutoMed schema transformation pathways):

$T_1: \quad LS_1 \rightarrow CS_1$

```
addRel ⟨⟨MAtab⟩⟩         [{'MA','MAC01',x}|x←⟨⟨MAC01⟩⟩] ++ [{'MA','MAC02',x}|x←⟨⟨MAC02⟩⟩]
                         ++[{'MA','MAC03',x}|x←⟨⟨MAC03⟩⟩];
addAtt ⟨⟨MAtab, Dept⟩⟩   [{k1,k2,k3,k1}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt ⟨⟨MAtab, CID⟩⟩    [{k1,k2,k3,k2}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt ⟨⟨MAtab, SID⟩⟩    [{k1,k2,k3,k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩];
addAtt ⟨⟨MAtab, Mark⟩⟩   [{'MA','MAC01',k,x}|{k,x}←⟨⟨MAC01, Mark⟩⟩]
                         ++[{'MA','MAC02',k,x}|{k,x}←⟨⟨MAC02, Mark⟩⟩]
                         ++[{'MA','MAC03',k,x}|{k,x}←⟨⟨MAC03, Mark⟩⟩];
delAtt ⟨⟨MAC01, Mark⟩⟩   [{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, Mark⟩⟩; k2='MAC01'];
delAtt ⟨⟨MAC01, SID⟩⟩    [{k3,x}|{k1,k2,k3,x}←⟨⟨MAtab, SID⟩⟩; k2='MAC01'];
delRel ⟨⟨MAC01⟩⟩         [{k3}|{k1,k2,k3}←⟨⟨MAtab⟩⟩; k2='MAC01']
...
```

The removal of the other two tables in LS_1 is similar.

$T_2: \quad LS_2 \rightarrow CS_2$

```
addRel ⟨⟨IStab⟩⟩          [{'IS',x,y}|{c,x}←⟨⟨course, CID⟩⟩; {s,y}←⟨⟨student, SID⟩⟩];
addAtt ⟨⟨IStab, Dept⟩⟩    [{k1,k2,k3,k1}|{k1,k2,k3}←⟨⟨IStab⟩⟩];
addAtt ⟨⟨IStab, CID⟩⟩     [{k1,k2,k3,k2}|{k1,k2,k3}←⟨⟨IStab⟩⟩];
addAtt ⟨⟨IStab, SID⟩⟩     [{k1,k2,k3,k3}|{k1,k2,k3}←⟨⟨IStab⟩⟩];
addAtt ⟨⟨IStab, CName⟩⟩   [{'IS',x,y,n}|{c1,x}←⟨⟨course, CID⟩⟩; {c2,n}←⟨⟨course, cname⟩⟩; c1=c2;
                                  {c3,s1}←⟨⟨course, student⟩⟩; c3=c2; {s2,y}←⟨⟨student, SID⟩⟩; s2=s1];
addAtt ⟨⟨IStab, Mark⟩⟩    [{'IS',x,y,m}|{c1,x}←⟨⟨course, CID⟩⟩; {c2,s1}←⟨⟨course, student⟩⟩; c1=c2;
                                  {s2,y}←⟨⟨student, SID⟩⟩; s2=s1; {s3,m}←⟨⟨student, mark⟩⟩; s3=s2];
...
```

$T_3: \quad LS_3 \rightarrow CS_3$

```
addRel ⟨⟨CStab⟩⟩          [{'CS',x,y}|x←['CSC01','CSC02','CSC03']; y←⟨⟨CSMarks⟩⟩];
addAtt ⟨⟨CStab, Dept⟩⟩    [{k1,k2,k3,k1}|{k1,k2,k3}←⟨⟨CStab⟩⟩];
addAtt ⟨⟨CStab, CID⟩⟩     [{k1,k2,k3,k2}|{k1,k2,k3}←⟨⟨CStab⟩⟩];
addAtt ⟨⟨CStab, SID⟩⟩     [{k1,k2,k3,k3}|{k1,k2,k3}←⟨⟨CStab⟩⟩];
addAtt ⟨⟨CStab, SName⟩⟩   [{'CS',x,k,s}|x←['CSC01','CSC02','CSC03']; {k,s}←⟨⟨CSMarks, SName⟩⟩];
addAtt ⟨⟨CStab, Mark⟩⟩    [{'CS','CSC01',k,x}|{k,x}←⟨⟨CSMarks, CSC01⟩⟩]
                          ++[{'CS','CSC02',k,x}|{k,x}←⟨⟨CSMarks, CSC02⟩⟩]
                          ++[{'CS','CSC03',k,x}|{k,x}←⟨⟨CSMarks, CSC03⟩⟩];
...
```

$T_u: \quad US \rightarrow GS$

```
addRel ⟨⟨CourseSum⟩⟩         distinct [{k1,k3}|{k1,k2,k3}←⟨⟨Details⟩⟩];
addAtt ⟨⟨CourseSum, Dept⟩⟩   [{k1,k2,k1}|{k1,k2}←⟨⟨CourseSum⟩⟩];
addAtt ⟨⟨CourseSum, CID⟩⟩    [{k1,k2,k2}|{k1,k2}←⟨⟨CourseSum⟩⟩];
addAtt ⟨⟨CourseSum, Total⟩⟩  [{x,y,z}|{{x,y},z}←
                                   (gc sum [{{k1,k3},x}|{k1,k2,k3,x}←⟨⟨Details, Mark⟩⟩])];
addAtt ⟨⟨CourseSum, Avg⟩⟩    [{x,y,z}|{{x,y},z}←
                                   (gc avg [{{k1,k3},x}|{k1,k2,k3,x}←⟨⟨Details, Mark⟩⟩])];
...
```

3 Expressing Schema and Data Model Evolution

In a heterogeneous data warehousing environment, it is possible for either a data source schema or the integrated database schema to evolve. This schema evolution may be a change in the schema, or a change in the data model in which the schema is expressed, or both. AutoMed transformations can be used to express the schema evolution in all three cases:

(a) Consider first a schema S expressed in a modelling language $\mathcal{M}$. We can express the evolution of S to S^{new}, also expressed in $\mathcal{M}$, as a series of primitive transformations that rename, add, extend, delete or contract constructs of $\mathcal{M}$. For example, suppose that the relational schema LS_1 in the above example evolves so its three tables become a single table with an extra column for the course ID. This evolution is captured by a pathway which is identical to the pathway $LS_1 \to CS_1$ given above.

This kind of transformation that captures well-known equivalences between schemas can be defined in AutoMed by means of a parametrised transformation *template* which is schema- and data-independent. When invoked with specific schema constructs and their extents, a template generates the appropriate sequence of primitive transformations within the Schemas & Transformations Repository – see [5] for details.

(b) Consider now a schema S expressed in a modelling language $\mathcal{M}$ which evolves into an equivalent schema S^{new} expressed in a modelling language $\mathcal{M}^{new}$. We can express this translation by a series of add steps that define the constructs of S^{new} in $\mathcal{M}^{new}$ in terms of the constructs of S in $\mathcal{M}$. At this stage, we have an intermediate schema that contains the constructs of both S and S^{new}. We then specify a series of delete steps that remove the constructs of $\mathcal{M}$ (the queries within these transformations indicate that these are now redundant constructs since they can be derived from the new constructs).

For example, suppose that XML schema LS_2 in the above example evolves into an equivalent relational schema consisting of single table with one column per attribute of LS_2. This evolution is captured by a pathway which is identical to the pathway $LS_2 \to CS_2$ given above.

Again, such generic inter-model translations between one data model and another can be defined in AutoMed by means of transformation templates.

(c) Considering finally to an evolution which is both a change in the schema and in the data model, this can be expressed by a combination of (a) and (b) above: either (a) followed by (b), or (b) followed by (a), or indeed by interleaving the two processes.

4 Handling Schema Evolution

In this section we consider how the general integration network illustrated in Figure 1 is evolvable in the face of evolution of a local schema or the warehouse schema. We have seen in the previous section how AutoMed transformations can be used to express the schema evolution if either the schema or the data model changes, or both. We can therefore treat schema and data model change in a uniform way for the purposes of handling schema evolution: both are expressed as a sequence of AutoMed primitive transformations, in the first case staying within the original data model, and in the second case transforming the original schema in the original data model into a new schema in a new data model.

In this section we describe the actions that are taken in order to evolve the integration network of Figure 1 if the global schema GS evolves (Section 4.1) or if a local schema LS_i evolves (Section 4.2). Given an evolution pathway from a schema S to a schema S^{new}, in both cases each successive primitive transformation within the pathway $S \to S^{new}$ is treated one at a time. Thus, we describe in sections 4.1 and 4.2 the actions that are taken if $S \to S^{new}$ consists of just one primitive transformation. If $S \rightarrow S^{new}$ is a composite transformation, then it is handled as a sequence of primitive transformations. Our discussion below assumes that the primitive transformation being handled is adding, removing or renaming a construct of S that has an underlying data extent. We do not discuss the addition or removal of constraints here as these do not impact on the materialised data, and we make the assumption that any constraints in the pathway $S \rightarrow S^{new}$ have been verified as being valid.

4.1 Evolution of the Global Schema

Suppose the global schema GS evolves by means of a primitive transformation t into GS^{new}. This is expressed by the step t being appended to the pathway T_u of Figure 1. The new global schema is GS^{new} and its associated extension is GD^{new}. GS is now an intermediate schema in the extended pathway $T_u; t$ and it no longer has an extension associated with it. t may be a rename, add, extend, delete or contract transformation. The following actions are taken in each case:

1. If t is rename c c', then there is nothing further to do. GS is semantically equivalent to GS^{new} and GD^{new} is identical to GD except that the extent of c in GD is now the extent of c' in GD^{new}.

2. If t is add c q, then there is nothing further to do at the schema level. GS is semantically equivalent to GS^{new}. However, the new construct c in GD^{new} must now be populated, and this is achieved by evaluating the query q over GD.

3. If t is extend c, then the new construct c in GD^{new} is populated by an empty extent. This new construct may subsequently be populated by an expansion in a data source (see Section 4.2).

4. If t is delete c q or contract c, then the extent of c must be removed from GD in order to create GD^{new} (it is assumed that this a legal deletion/contraction, e.g if we wanted to delete/contract a table from a relational schema, then first the constraints and then the columns would be deleted/contracted and lastly the table itself; such syntactic correctness of transformation pathways is automatically verified by AutoMed). It may now be possible to simplify the transformation network, in that if T_u contains a matching transformation add c q or extend c, then both this and the new transformation t can be removed from the pathway $US \rightarrow GS^{new}$. This is purely an optimization – it does not change the meaning of a pathway, nor its effect on view generation and query/data translation. We refer the reader to [19] for details of the algorithms that simplify AutoMed transformation pathways.

In cases *2* and *3* above, the new construct c will automatically be propagated into the schema DMS of any data mart derived from GS. To prevent this, a transformation contract c can be prefixed to the pathway $GS \rightarrow DMS$. Alternatively, the new construct c can be propagated to DMS if so desired, and materialised there. In cases *1* and *4* above, the change in GS and GD may impact on the data marts derived from GS, and we discuss this in Section 4.3.

4.2 Evolution of a Local Schema

Suppose a local schema LS_i evolves by means of a primitive transformation t into LS_i^{new}. As discussed in Section 2, there is automatically available a reverse transformation $\overline{t}$ from LS_i^{new} to LS_i and hence a pathway $\overline{t}; T_i$ from LS_i^{new} to CS_i. The new local schema is LS_i^{new} and its associated extension is LD_i^{new}. LS_i is now just an intermediate schema in the extended pathway $\overline{t}; T_i$ and it no longer has an associated extension.

t may be a rename, add, delete, extend or contract transformation. In 1–5 below we see what further actions are taken in each case for evolving the integration network and the downstream materialised data as necessary.

We first introduce some necessary terminology: If p is a pathway $S \to S'$ and c is a construct in S, we denote by $descendants(c, p)$ the constructs of S' which are directly or indirectly dependent on c, either because c itself appears in S' or because a construct c' of S' is created by a transformation add c' q within p where the query q directly or indirectly references c. The set $descendants(c, p)$ can be straight-forwardly computed by traversing p and inspecting the query associated with each add transformation within in.

1. If t is rename c c', then schema LS_i^{new} is semantically equivalent to LS_i. The new transformation pathway $T_i^{new} : LS_i^{new} \to CS_i$ is $\overline{t}; T_i$ = rename c' c; T_i. The new local database LD_i^{new} is identical to LD_i except that the extent of c in LD_i is now the extent of c' in LD_i^{new}.

2. If t is add c q, then LS_i has evolved to contain a new construct c whose extent is equivalent to the expression q over the other constructs of LS_i. The new transformation pathway $T_i^{new} : LS_i^{new} \to CS_i$ is $\overline{t}; T_i$ = delete c q; T_i.

3. If t is delete c q, this means that LS_i has evolved to not include a construct c whose extent is derivable from the expression q over the other constructs of LS_i, and the new local database LD_i^{new} no longer contains an extent for c. The new transformation pathway $T_i^{new} : LS_i^{new} \to CS_i$ is $\overline{t}; T_i$ = add c q; T_i.

In the above three cases, schema LS_i^{new} is semantically equivalent to LS_i, and nothing further needs to be done to any of the transformation pathways, schemas or databases $CD_1, \ldots, CD_n$ and GD. This may not be the case if t is a contract or extend transformation, which we consider next.

4. If t is extend c, then there will be a new construct available from LS_i^{new} that was not available before. That is, LS_i has evolved to contain the new construct c whose extent is not derivable from the other constructs of LS_i. If we left the transformation pathway T_i as it is, this would result in a pathway T_i^{new} = contract c; T_i from LS_i^{new} to CS_i, which would immediately drop the new construct c from the integration network. That is, T_i^{new} is consistent but it does not utilize the new data.

However, recall that we said earlier that we assume no contract steps in the pathways from local schemas to their union schemas, and that all the data in LS_i should be available to the integration network. In order to achieve this, there are four cases to consider:

(a) c appears in US_i and has the same semantics as the newly added c in LS_i^{new}. Since c cannot be derived from the original LS_i, there must be a transformation extend c, in $CS_i \rightarrow US_i$.
We remove from T_i^{new} the new contract c step and this matching extend c step. This propagates c into CS_i, and we populate its extent in the materialised database CD_i by replicating its extent from LD_i^{new}.
(b) c does not appear in US_i but it can be derived from US_i by means of some transformation T.
In this case, we remove from T_i^{new} the first contract c step, so that c is now present in CS_i and in US_i. We populate the extent of c in CD_i by replicating its extent from LD_i^{new}.
To repair the other pathways $T_j : LS_j \rightarrow CS_j$ and schemas US_j for $j \neq i$, we append T to the end of each T_j. As a result, the new construct c now appears in all the union schemas. To add the extent of this new construct to each materialised database CD_j for $j \neq i$, we compute it from the extents of the other constructs in CS_j using the queries within successive add steps in T.
We finally append the necessary new id steps between pairs of union schemas to assert the semantic equivalence of the construct c within them.
(c) c does not appear in US_i and cannot be derived from US_i.
In this case, we again remove from T_i^{new} the first contract c step so that c is now present in schema CS_i.
To repair the other pathways $T_j : LS_j \rightarrow CS_j$ and schemas US_j for $j \neq i$, we append an extend c step to the end of each T_j. As a result, the new construct c now appears in all the conformed schemas $CS_1, \ldots, CS_n$.
The construct c may need further translation into the data model of the union schemas and this is done by appending the necessary sequence, T, of add/delete/rename steps to all the pathways $LS_1 \rightarrow CS_1, \ldots, LS_n \rightarrow CS_n$.
We compute the extent of c within the database CD_i from its extent within LD_i^{new} using the queries within successive add steps in T.
We finally append the necessary new id steps between pairs of union schemas to assert the semantic equivalence of the new construct(s) within them.
(d) c appears in US_i but has different semantics to the newly added c in LS_i^{new}. In this case, we rename c in LS_i^{new} to a new construct c'. The situation reverts to adding a new construct c' to LS_i^{new}, and one of (a)-(c) above applies.

We note that determining whether c can or cannot be derived from the existing constructs of the union schemas in (a)–(d) above requires domain or expert human knowledge. Thereafter, the remaining actions are fully automatic.

In cases (a) and (b), there is new data added to one or more of the conformed databases which needs to be propagated to GD. This is done by computing $descendants(c, T_u)$ and using the algebraic equivalences of Section 2.1 to propagate changes in the extent of c to each of its descendant constructs gc in GS. Using these equivalences, we can in most cases incrementally recompute the extent of gc. If at any stage in T_u there is a transformation add c' q where no equivalence can be applied, then we have to recompute the whole extent of c'.

In cases (b) and (c), there is a new schema construct c appearing in the US_i. This construct will automatically appear in the schema GS. If this is not desired, a transformation contract c can be prefixed to T_u.

5. If t is contract c, then the construct c in LS_i will no longer be available from LS_i^{new}. That is, LS_i has evolved so as to not include a construct c whose extent is not derivable from the other constructs of LS_i. The new local database LD_i^{new} no longer contains an extent for c.
The new transformation pathway $T_i^{new} : LS_i^{new} \rightarrow CS_i$ is $\overline{t}; T_i = \mathsf{extend}\ c; T_i$. Since the extent of c is now Void, the materialised data in CD_i and GD must be modified so as to remove any data derived from the old extent of c.
In order to repair CD_i, we compute $descendants(c, LS_i \rightarrow CS_i)$. For each construct uc in $descendants(c, LS_i \rightarrow CS_i)$, we compute its new extent and replace its old extent in CD_i by the new extent. Again, the algebraic properties of IQL queries discussed in Section 2.1 can be used to propagate the new Void extent of construct c in LS_i^{new} to each of its descendant constructs uc in CS_i. Using these equivalences, we can in most cases incrementally recompute the extent of uc as we traverse the pathway T_i.
In order to repair GD, we similarly propagate changes in the extent of each uc along the pathway T_u.
Finally, it may also be necessary to amend the transformation pathways if there are one or more constructs in GD which now will always have an empty extent as a result of this contraction of LS_i. For any construct uc in US whose extent has become empty, we examine all pathways $T_1, \ldots, T_n$. If all these pathways contain an extend uc transformation, or if using the equivalences of Section 2.1 we can deduce from them that the extent of uc will always be empty, then we can suffix a contract gc step to T_u for every gc in $descendants(uc, T_u)$, and then handle this case as paragraph 4 in Section 4.1.

4.3 Evolution of Downstream Data Marts

We have discussed how evolutions to the global schema or to a source schema are handled. One remaining question is how to handle the impact of a change to the data warehouse schema, and possibly its data, on any data marts that have been derived from it.

In [7] we discuss how it is possible to express the derivation of a data marts from a data warehouse by means of an AutoMed transformation pathway. Such a pathway $GS \rightarrow DMS$ expresses the relationship of a data mart schema DMS to the warehouse schema GS. As such, this scenario can be regarded as a special case of the general integration scenario of Figure 1, where GS now plays the role of the single source schema, databases $CD_1, \ldots, CD_n$ and GD collectively play the role of the data associated with this source schema and DMS plays the role of the global schema. Therefore, the same techniques as discussed in sections 4.1 and 4.2 can be applied.

5 Concluding Remarks

In this paper we have described how the AutoMed heterogeneous data integration toolkit can be used to handle the problem of schema evolution in heterogeneous data warehousing environments so that the previous transformation, integration and data materialisation effort can be reused. Our algorithms are mainly automatic, except for the aspects that require domain or expert human knowledge regarding the semantics of new schema constructs.

We have shown how AutoMed transformations can be used to express schema evolution within the same data model, or a change in the data model, or both, whereas other schema evolution literature has focussed on just one data model. Schema evolution within the relational data model has been discussed in previous work such as [11, 12, 18]. The approach in [18] uses a first-order schema in which all values in a schema of interest to a user are modelled as data, and other schemas can be expressed as a query over this first-order schema. The approach in [12] uses the notation of a *flat scheme*, and gives four operators UNITE, FOLD, UNFOLD and SPLIT to perform relational schema evolution using the SchemaSQL language. In contrast, with AutoMed the process of schema evolution is expressed using a simple set of primitive schema transformations augmented with a functional query language, both of which are applicable to multiple data models.

Our approach is complementary to work on mapping composition, e.g. [20, 14], in that in our case the new mappings are a composition of the original transformation pathway and the transformation pathway which expresses the schema evolution. Thus, the new mappings are, by definition, correct. There are two aspects to our approach: (i) handling the transformation pathways and (ii) handling the queries within them. In this paper we have in particular assumed that the queries are expressed in IQL. However, the AutoMed toolkit allows any query language syntax to be used within primitive transformations, and therefore this aspect of our approach could be extended to other query languages.

Materialised data warehouse views need to be maintained when the data sources change, and much previous work has addressed this problem at the data level. However, as we have discussed in this paper, materialised data warehouse views may also need to be modified if there is an evolution of a data source schema. Incremental maintenance of schema-restructuring views within the relational data model is discussed in [10], whereas our approach can handle this problem in a heterogeneous data warehousing environment with multiple data models and changes in data models. Our previous work [7] has discussed how AutoMed transformation pathways can also be used for incrementally maintaining materialised views at the data level. For future work, we are implementing our approach and evaluating it in the context of biological data warehousing.

References

1. J. Andany, M. Léonard, and C. Palisser. Management of schema evolution in databases. In *Proc. VLDB'91*, pages 161–170. Morgan Kaufmann, 1991.
2. Z. Bellahsene. View mechanism for schema evolution in object-oriented DBMS. In *Proc. BNCOD'96, LNCS 1094*. Springer, 1996.
3. B. Benatallah. A unified framework for supporting dynamic schema evolution in object databases. In *Proc. ER'99, LNCS 1728*. Springer, 1999.
4. M. Blaschka, C. Sapia, and G. Höfling. On schema evolution in multidimensional databases. In *Proc. DaWaK'99, LNCS 1767*. Springer, 1999.
5. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien, and N. Rizopoulos. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE'04*, 2004.
6. P. Buneman *et al.* Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
7. H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. DOLAP'03*, pages 86–93. ACM Press, 2003.
8. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report 20, Automed Project, 2003.
9. E. Jasper, N. Tong, P. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. In *Proc. 6th Baltic Conference on Databases and Information Systems*, 2004.
10. A. Koeller and E. A. Rundensteiner. Incremental maintenance of schema-restructuring views. In *Proc. EDBT'02, LNCS 2287*. Springer, 2002.
11. L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In *Proc. DOOD'93, LNCS 760*. Springer, 1993.
12. L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On efficiently implementing SchemaSQL on an SQL database system. In *Proc. VLDB'99*, pages 471–482. Morgan Kaufmann, 1999.
13. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, 2002.
14. Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *Proc. VLDB'03*. Morgan Kaufmann, 2003.
15. P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99, LNCS 1626*, pages 333–348. Springer, 1999.
16. P. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02, LNCS 2348*, pages 484–499. Springer, 2002.
17. P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, pages 227–238, 2003.
18. Renée J. Miller. Using schematically heterogeneous structures. In *Proc. ACM SIGMOD'98*, pages 189–200. ACM Press, 1998.
19. N. Tong. Database schema transformation optimisation techniques for the AutoMed system. In *Proc. BNCOD'03, LNCS 2712*. Springer, 2003.
20. Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Mapping adaptation under evolving schemas. In *Proc. VLDB'03*. Morgan Kaufmann, 2003.
21. L. Zamboulis. XML data integration by graph restrucring. In *Proc. BNCOD'04, LNCS 3112*. Springer, 2004.