Using Schema Transformation Pathways for Incremental View Maintenance

Hao Fan

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

Abstract. With the increasing amount and diversity of information available on the Internet, there has been a huge growth in information systems that need to integrate data from distributed, heterogeneous data sources. Incrementally maintaining the integrated data is one of the problems being addressed in data warehousing research. This paper presents an incremental view maintenance approach based on schema transformation pathways. Our approach is not limited to one specific data model or query language, and would be useful in any data transformation/integration framework based on sequences of primitive schema transformations.

1 Introduction

Data warehouses collect data from distributed, autonomous and heterogeneous data sources into a central repository to enable analysis and mining of the integrated information. When data sources change, the data warehouse, in particular the materialised views in the data warehouse, must be updated also. This is the problem of view maintenance in data warehouses. In contrast to operational database systems handling day-to-day operations of an organisation and dealing with small changes to the databases, data warehouses support queries by non-technical users based on long-term, statistical information integrated from a variety of data sources, and do not require the most up to date operational version of all the data. Thus, data warehouses are normally refreshed periodically and updates to the primary data sources do not have to be propagated to the data warehouse immediately.

AutoMed[1] is a heterogeneous data transformation and integration system which offers the capability to handle data integration across multiple data models. In the AutoMed approach, the integration of schemas is specified as a sequence of primitive schema transformation steps, which incrementally add, delete or rename schema constructs, thereby transforming each source schema into the target schema. We term the sequence of primitive transformations steps defined for transforming a schema S_1 into a schema S_2 a *transformation pathway* from S_1 to S_2.

In previous work (see [7]), we discussed how AutoMed metadata can be used to express the schemas and the cleansing, transformation and integration processes in heterogeneous data warehousing environments. In this paper, we will describe how AutoMed metadata can be used for maintaining the warehouse data.

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

Materialised warehouse views need to be maintained either when the data of a data source changes, or if there is an evolution of a data source schema. In previous work (see [8]), we showed that AutoMed transformation pathways can be used to handle schema evolutions in a data warehouse. In this paper, we will focus on refreshing materialised warehouse views at the data level.

Materialised views can be refreshed by recomputing from scratch or, on the other hand, by only computing the changes to the views rather than all the view data, which is termed *incremental view maintenance* (IVM). Incrementally refreshing a view can be significantly cheaper than fully recomputing the view, especially if the size of the view is large compared to the size of the change.

The problem of view maintenance at the data level has been widely discussed in the literature. Dong and Gupta $et\ al$ give good surveys of this problem [6, 10]. Colby $et\ al$, Griffin $et\ al$ and Quass present propagation formulae based on relational algebra operations for incrementally maintaining views with duplicates and aggregations [5, 9, 16]. Zhuge $et\ al$ consider the IVM problem for a single-source data warehouses and defines the ECA algorithm [20]. The IVM approaches for a multi-source data warehouse include the Strobe algorithm [21], and the SWEEP and Nested SWEEP algorithms [1]. The view maintenance approach discussed by Gupta and Quass $et\ al$ is to make views *self-maintainable*, which means that materialised views can be refreshed by only using the content of the views and the updates to the data sources, and not requiring to access the data in any underlying data source [11, 17]. Such a view maintenance approach usually needs auxiliary materialised views to store additional information.

Our IVM approach presented in this paper is based on AutoMed schema transformation pathways, which is not limited to one specific data model or query language, and would be useful in any data transformation/integration framework based on sequences of primitive schema transformations.

The outline of this paper is as follows. Section 2 gives an overview of AutoMed, as well as a data integration example. Section 3 presents our IVM formulae and algorithms using AutoMed schema transformations. Section 4 gives our concluding remarks and directions of further work.

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 [13], XML [18], flat-file [3] 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 [3], AutoMed provides a set of primitive schema transformations that can be applied to schema constructs expressed in $\mathcal{M}$. In particular, for 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, expressed in some modelling language. Thus, the intermediate (and indeed the target) 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[2]. The queries within add and delete transformations are used by AutoMed's Global Query Processor to evaluate an IQL query over a global schema in the case of a virtual data integration scenario. In the case that the global schema is materialised, AutoMed's Query Evaluator can be used directly on the materialised data.

2.1 Simple IQL

In order to illustrate our IVM algorithm, we use a subset of IQL, *Simple IQL* (SIQL), as the query language in this paper. More complex IQL queries can be encoded as a series of transformations with SIQL queries on intermediate schema constructs. We stress that although illustrated within a particular query language syntax, our IVM algorithms could also be applied to schema transformation pathways involving queries expressed in other query languages supporting operations on set, bag and list collections.

Supposing $\mathtt{D}$, $\mathtt{D}_1 \ldots$, $\mathtt{D}_n$ denote bags of the appropriate type (base collections), SIQL supports the following queries: `group D` groups a bag of pairs `D` on their first component. `distinct D` removes duplicates from a bag. `f D` applies an aggregation function `f` (which may be `max`, `min`, `count`, `sum` or `avg`) to a bag. `gc f D` groups a bag `D` of pairs on their first component and applies an aggregation function `f` to the second component. ++ is the bag union operator and $--$ is the bag *monus* operator [2]. SIQL comprehensions are of three forms: $[\overline{\mathtt{x}}|\overline{\mathtt{x_1}} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; ...; \mathtt{C}_k]$, $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}]$, and $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{not}(\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})]$. Here, each $\overline{\mathtt{x_1}}$, ..., $\overline{\mathtt{x}_n}$ is either a single variable or a tuple of variables. $\overline{\mathtt{x}}$ is either a single variable or value, or a tuple of variables or values, and must include all of variables appearing in $\overline{\mathtt{x_1}}$, ..., $\overline{\mathtt{x}_n}$. Each $\mathtt{C}_1$, ..., $\mathtt{C}_k$ is a condition not referring to any base collection. Also, each variable appearing in $\overline{\mathtt{x}}$ and $\mathtt{C}_1$, ..., $\mathtt{C}_k$ must also appear in some $\overline{\mathtt{x}_i}$, and the variables in $\overline{\mathtt{y}}$ must appear in $\overline{\mathtt{x}}$. Finally, a query of the form $\mathtt{map}\ (\lambda\overline{\mathtt{x}}.\mathtt{e})\ \mathtt{D}$ applies to each element of a collection `D` an anonymous function defined by a lambda abstraction $\lambda\overline{\mathtt{x}}.\mathtt{e}$ and returns the resulting collection.

Comprehension syntax can express the common algebraic operations on collection types such as sets, bags and lists [4] and such operations can be readily expressed in SIQL. In particular, let us consider *selection* (σ), *projection*(π), *join* ($\bowtie$), and *aggregation* (α) (*union* ($\bigcup$) and *difference* ($-$) are directly supported in SIQL via the ++ and $--$ operators). The general form of a Select-Project-Join (SPJ) expression is

[2] IQL is a comprehensions-based functional query language. Such languages subsume query languages such as SQL and OQL in expressiveness [4]. We refer the reader to [12, 15] for details of IQL and references to work on comprehension-based functional query languages.

$\pi_A(\sigma_C(D_1 \bowtie ... \bowtie D_n))$ and this can be expressed as follows in comprehension syntax: $[\mathtt{A}|\overline{\mathtt{x_1}} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$. However, since in general the tuple of variables A may not contain all the variables appearing in $\overline{\mathtt{x_1}}, ..., \overline{\mathtt{x}_n}$ (as is required in SIQL), we can use the following two transformation steps to express a general SPJ expression in SIQL, where $\overline{\mathtt{x}}$ includes all of the variables appearing in $\overline{\mathtt{x_1}}, \ldots . \overline{\mathtt{x}_n}$:

$\mathtt{v1} = [\overline{\mathtt{x}}|\overline{\mathtt{x_1}} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$
$\mathtt{v} \ \ = \mathtt{map}\ (\lambda\overline{\mathtt{x}}.\mathtt{A})\ \mathtt{v1}$

The algebraic operator α applies an aggregation function to a collection and this functionality is captured by the gc operator in SIQL. E.g., supposing the scheme of a collection D is D(A1,A2,A3), an expression $\alpha_{\mathtt{A2,f(A3)}}(\mathtt{D})$ is expressed in SIQL as:

v1 = map (λ{x1,x2,x3}.{x2,x3}) D
v = gc f v1

2.2 An Example Data Integration

In this paper, we will use schemas expressed in a simple relational data model to illustrate our techniques. However, we stress that these techniques are applicable to schemas defined in *any* data modelling language having been specified within AutoMed's Model Definitions Repository, including modelling languages for semi-structured data [3, 18].

In our simple relational model, there are two kinds of schema construct: Rel and Att. The extent of a Rel construct ⟨⟨R⟩⟩ is the projection of relation R onto its primary key attributes $k_1, ..., k_n$. The extent of each Att construct ⟨⟨R, a⟩⟩ where a is a non-key attribute of R is the projection of R onto $k_1, ..., k_n, a$. We refer the reader to [13] for an encoding of a richer relational data model, including the modelling of constraints.

Suppose that MAtab(CID, SID, Mark) and IStab(CID, SID, Mark) are two source relations for a data warehouse respectively storing students' marks for two departments MA and IS, in which CID and SID are the course and student IDs. Suppose also that a relation Course(Dept, CID, Avg) is in the data warehouse which gives the average mark for each course of each department.

The following transformation pathway expresses the schema transformation and integration processes in this example. Due to space limitations, we have not given the steps for removing the source relation constructs (note that this 'growing' and 'shrinking' of schemas is characteristic of AutoMed schema transformation pathways). Schema constructs ⟨⟨Details⟩⟩ and ⟨⟨Details, Mark⟩⟩ are temporary ones which are created for integrating the source data and then deleted after the global relation is created.

```
addRel ⟨⟨Details⟩⟩          [{'MA',k1,k2}|{k1,k2} ← ⟨⟨MAtab⟩⟩]
                            ++[{'IS',k1,k2}|{k1,k2} ← ⟨⟨IStab⟩⟩];
addAtt ⟨⟨Details, Mark⟩⟩    [{'MA',k1,k2,x}|{k1,k2,x} ← ⟨⟨MAtab, Mark⟩⟩]
                            ++[{'IS',k1,k2,x}|{k1,k2,x} ← ⟨⟨IStab, Mark⟩⟩];
addRel ⟨⟨Course⟩⟩           distinct [{k,k1}|{k,k1,k2} ← ⟨⟨Details⟩⟩];
addAtt ⟨⟨Course, Avg⟩⟩      [{x,y,z}|{{x,y},z} ← (gc avg
                                [{{k,k1},x}|{k,k1,k2,x} ← ⟨⟨Details, Mark⟩⟩])];
delAtt ⟨⟨Details, Mark⟩⟩    [{'MA',k1,k2,x}|{k1,k2,x} ← ⟨⟨MAtab, Mark⟩⟩]
                            ++[{'IS',k1,k2,x}|{k1,k2,x} ← ⟨⟨IStab, Mark⟩⟩];
delRel ⟨⟨Details⟩⟩          [{'MA',k1,k2}|{k1,k2} ← ⟨⟨MAtab⟩⟩]
                            ++[{'IS',k1,k2}|{k1,k2} ← ⟨⟨IStab⟩⟩];
...
```

Note that some of the queries appearing in the above transformation steps are not SIQL but general IQL queries. In such cases, for the purposes of IVM, we decompose a general IQL query into a sequence of SIQL queries by means of a depth-first traversal of the IQL query tree. For example, the IQL query $[\{\texttt{x,y,z}\}|\{\{\texttt{x,y}\},\texttt{z}\} \leftarrow (\texttt{gc avg}\ [\{\{\texttt{k,k1}\},\texttt{x}\}|\ \{\texttt{k,k1,k2,x}\} \leftarrow \langle\langle \mathsf{Details}, \mathsf{Mark} \rangle\rangle])]$ is decomposed into following sequence of SIQL queries, where v1 and v2 are virtual intermediate views:

$\texttt{v1} = \texttt{map}\ (\lambda\{\texttt{k,k1,k2,x}\}.\{\{\texttt{k1,k2}\},\texttt{x}\})\ \langle\langle \mathsf{Details}, \mathsf{Mark} \rangle\rangle$
$\texttt{v2} = \texttt{gc avg v1}$
$\texttt{v} \ = \texttt{map}\ (\lambda\{\{\texttt{x,y}\},\texttt{z}\}.\{\texttt{x,y,z}\})\ \texttt{v2}$

From now on, we assume that all queries in transformation steps are SIQL queries.

3 IVM with AutoMed Schema Transformations

Our IVM algorithms use the individual steps of a transformation pathway to compute the changes to each intermediate construct in the pathway, and finally obtain the changes to the view created by the transformation pathway in a stepwise fashion. Since no construct in a global schema is contributed by delete and contract transformations, we ignore these transformations in our IVM algorithms. In addition, computing changes based on a transformation rename(O', O) is simple — the changes to O are the same as the changes to O'. Thus, we only consider add transformations here.

We can express a single add transformation step as an expression v=q(D), in which v is the schema construct created by the transformation and q is the SIQL query over the data source D. In order to incrementally maintain the global schema data, we develop a set of IVM formulae for each SIQL query, and apply these IVM formulae on each transformation step to compute the changes to the construct created by the step. By following all the steps in the transformation pathway, we compute the intermediate changes step by step, finally ending up with the final changes to the global schema data.

3.1 IVM Formulae for SIQL Queries

We use $\triangle\mathcal{C}/\triangledown\mathcal{C}$ to denote a collection of data items inserted into/deleted from a collection $\mathcal{C}$. There might be many possible expressions for $\triangle\mathcal{C}$ and $\triangledown\mathcal{C}$ but not all are equally desirable. For example, we could simply let $\triangledown\mathcal{C} = \mathcal{C}$ and $\triangle\,\mathcal{C} = \triangle\,\mathcal{C}^{new}$, but this is equivalent to recomputing the view from scratch [16]. In order to avoidsuch definitions, we use the concept of $minimality$ [9] to ensure that no unnecessary data are produced.

Minimality Conditions Any changes ($\triangle\,\mathcal{C}/\triangledown\mathcal{C}$) to a data collection $\mathcal{C}$, including the data source and the view, must satisfy the following minimality conditions:

(i) $\triangledown\mathcal{C} \subseteq \mathcal{C}$: We only delete tuples that are in $\mathcal{C}$;
(ii) $\triangle\mathcal{C} \cap \triangledown\mathcal{C} = \emptyset$: We do not delete a tuple and then reinsert it.

We now give the IVM formulae for each SIQL query, in which v denotes the view, D denotes the data sources, $\triangle\texttt{v}/\triangledown\texttt{v}$ and $\triangle\texttt{D}/\triangledown\texttt{D}$ denote the collections inserted into/deleted from v and D, and $\texttt{D}^{new}$ denotes the source collect D after the update. We observe that these formulae guarantee that the above minimality conditions are satisfied of $\triangle\texttt{v}$ and $\triangledown\texttt{v}$ provided they are satisfied by $\triangle\texttt{D}$ and $\triangledown\texttt{D}$.

v	$\triangle$v	∇v
distinct D	distinct $[x \mid x \leftarrow \triangle\mathtt{D};$ not(member v x)]	distinct $[x \mid x \leftarrow \nabla\mathtt{D};$ not(member $\mathtt{D}^{new}$ x)]
map λe1.e2 D	map λe1.e2 $\triangle$D	map λe1.e2 ∇D
max D	let r1 = max $\triangle$D; r2 = max ∇D	
	$\begin{cases} \max \triangle\mathtt{D}, & \text{if } (\mathtt{v} < \mathtt{r1}); \\ \emptyset, & \text{if } (\mathtt{v} \geq \mathtt{r1}) \& (\mathtt{v} \neq \mathtt{r2}); \\ \max \mathtt{D}^{new}, & \text{if } (\mathtt{v} > \mathtt{r1}) \& (\mathtt{v} = \mathtt{r2}). \end{cases}$	$\begin{cases} v, \text{ if } (\mathtt{v} < \mathtt{r1}); \\ \emptyset, \text{ if } (\mathtt{v} \geq \mathtt{r1}) \& (\mathtt{v} \neq \mathtt{r2}); \\ v, \text{ if } (\mathtt{v} > \mathtt{r1}) \& (\mathtt{v} = \mathtt{r2}). \end{cases}$
min D	let r1 = min $\triangle$D; r2 = min ∇D	
	$\begin{cases} \min \triangle\mathtt{D}, & \text{if } (\mathtt{v} > \mathtt{r1}); \\ \emptyset, & \text{if } (\mathtt{v} \leq \mathtt{r1}) \& (\mathtt{v} \neq \mathtt{r2}); \\ \min \mathtt{D}^{new}, & \text{if } (\mathtt{v} < \mathtt{r1}) \& (\mathtt{v} = \mathtt{r2}). \end{cases}$	$\begin{cases} \mathtt{v}, \text{ if } (\mathtt{v} > \mathtt{r1}); \\ \emptyset, \text{ if } (\mathtt{v} \leq \mathtt{r1}) \& (\mathtt{v} \neq \mathtt{r2}); \\ \mathtt{v}, \text{ if } (\mathtt{v} < \mathtt{r1}) \& (\mathtt{v} = \mathtt{r2}). \end{cases}$
count D	v + (count $\triangle$D) − (count ∇D)	v
sum D	v + (sum $\triangle$D) − (sum ∇D)	v
avg D	avg $\mathtt{D}^{new}$	v

Table 1. IVM formulae for distinct, map, and aggregate functions

v	$\triangle$v	∇v
D1 ++ D2	($\triangle$D1 −− ∇D2) ++ ($\triangle$D2 −− ∇D1)	(∇D1 −− $\triangle$D2) ++ (∇D2 −− $\triangle$D1)
D1 −− D2	(($\triangle$D1 −− $\triangle$D2) ++ (∇D2 −− ∇D1)) −−(D2 −− D1)	((∇D1 −− ∇D2) ++ ($\triangle$D2 −− $\triangle$D1)) $\cap$ v

Table 2. IVM formulae for bag union and monus

1. *IVM formulae for* distinct, map, *and aggregate functions*;

Table 1 illustrates the IVM formulae for these functions. We can see that the IVM formulae for distinct/max/min/avg function require accessing the post-update data source and using the view data; the formulae for count/sum function need to use the view data; and the formulae for map function only use the updates to the data source.

2. *IVM formulae for grouping functions such as* `group D` *and* `gc f D`;

Grouping functions group a bag of pairs D on their first component, and may apply an aggregate function f to the second component. For the IVM of a view defined by a grouping function, we firstly find the data items in D, which are in the same groups of the updates, $i.e.$ have the same first component with the updates. Then this smaller data collection can be used to compute the changes to the view, so as to save time and space overheads. For example, the IVM formulae for `v = gc f D` are as follows:

$\triangle\mathtt{v} = \mathtt{gc\ f}\ [\{x,y\} \mid \{x,y\} \leftarrow \mathtt{D}^{new}; \mathtt{member}\ [p \mid \{p,q\} \leftarrow (\triangle\mathtt{D} \mathbin{++} \nabla\mathtt{D})]\ x]$

$\nabla\mathtt{v} = [\{x,y\} \mid \{x,y\} \leftarrow \mathtt{v}; \mathtt{member}\ [p \mid \{p,q\} \leftarrow (\triangle\mathtt{D} \mathbin{++} \nabla\mathtt{D})]\ x]$

The IVM formulae for grouping functions require accessing the updated data source and using the view data.

3. *IVM formulae for bag union and monus*;

Table 2 illustrates IVM formulae for bag union and monus (see [9]), in which $\cap$ is an intersection operator with the following semantics: $\mathtt{D1} \cap \mathtt{D2} = \mathtt{D1} -- (\mathtt{D1} -- \mathtt{D2}) = \mathtt{D2} -- (\mathtt{D2} -- \mathtt{D1})$. The IVM formulae for bag union only use the changes to the data sources, while the formulae for bag monus have to use the view data and require an

```
Algorithm IVM4Comp()
Begin:
        tempView = D1^new;
        △v = △D1;
        ▽v = ▽D1;
        for i = 2 to n, do
            if (△Di or ▽Di is not empty)
                tempView = tempView ⋈ ... ⋈_{c(i-1)} D^new_(i-1);
                ▽v = (▽v ⋈_ci Di^new −− ▽v ⋈_ci △Di) ++ ▽v ⋈_ci ▽Di
                        ++(tempView ⋈_ci ▽Di−− △v ⋈_ci ▽Di);
                △v = (△v ⋈_ci Di^new−− △v ⋈_ci △Di) ++ tempView ⋈_ci △Di;
            else
                △v = △v ⋈_ci Di^new;
                ▽v = ▽v ⋈_ci Di^new;
        return △v and ▽v;
End
```

Fig. 1. The IVM4Comp Algorithm

auxiliary view $\texttt{D2} -- \texttt{D1}$. This auxiliary view is similarly incrementally maintained by using the IVM formulae for bag monus with $\texttt{D1} -- \texttt{D2}$.

4. *IVM formulae for comprehension* $[\overline{x}|\overline{x_1} \leftarrow \texttt{D1}; \ldots; \overline{x_n} \leftarrow \texttt{Dn}; C_1; C_2; ...; C_k]$;

For ease of discussion, we use the join operator $\bowtie$ to express this comprehension. In particular, $(\texttt{D1} \bowtie_c \texttt{D2}) = [\{x, y\}|x \leftarrow \texttt{D1}; y \leftarrow \texttt{D2}; c]$ where $c = C_1; ...; C_k$. More generally, $(\texttt{D1} \bowtie_{c_1,c_2} \texttt{D2} \bowtie_{c_3} \ldots \bowtie_{c_n} \texttt{Dn}) = [\overline{x}|\overline{x_1} \leftarrow \texttt{D1}; \ldots; \overline{x_n} \leftarrow \texttt{Dn}; c_1; c_2; ...; c_n]$ in which c_i is the conjunction of those predicates from $C_1, ..., C_k$ which contain variables appearing in $\overline{x_i}$ but without any variable appearing in $\overline{x_j}, j > i$.

We firstly give the IVM formulae of a view $\texttt{v} = \texttt{D1} \bowtie_c \texttt{D2}$ as follows. These IVM formulae can be derived from the propagation rules described in [9, 5].

$$\triangle\texttt{v} = (\triangle\texttt{D}_1 \bowtie_c \texttt{D}_2^{new} -- \triangle\texttt{D}_1 \bowtie_c \triangle\texttt{D}_2) \text{ ++ } \texttt{D}_1^{new} \bowtie_c \triangle\texttt{D}_2$$
$$\triangledown\texttt{v} = (\triangledown\texttt{D}_1 \bowtie_c \texttt{D}_2^{new} -- \triangledown\texttt{D}_1 \bowtie_c \triangle\texttt{D}_2) \text{ ++ } \triangledown\texttt{D}_1 \bowtie_c \triangledown\texttt{D}_2$$
$$\text{++}(\texttt{D}_1^{new} \bowtie_c \triangledown\texttt{D}_2 -- \triangle\texttt{D}_1 \bowtie_c \triangledown\texttt{D}_2)$$

Then, the IVM algorithm, IVM4Comp, for incrementally maintaining the view $\texttt{v} = (\texttt{D1} \bowtie_{c_1,c_2} \texttt{D2} \bowtie_{c_3} \ldots \bowtie_{c_n} \texttt{Dn})$ is given in Figure 1. This IVM algorithm for the comprehension needs to access all the post-update data sources.

The IVM4Comp algorithm is similar to the IVM algorithms discussed in [21] and [1], $i.e.$ the Strobe and SWEEP algorithms, in the context of maintaining a multi-source data warehouse. Both the Strobe and the SWEEP algorithm perform an IVM procedure for each update to a data source so as to ensure the data warehouse is consistent with the updated data source. For both algorithms, the cost of the messaging between the data warehouse and the data sources for each update is $O(n)$ where n is the number of data sources. However, in practice, warehouse data are normally long-term and just refreshed periodically. Our IVM4Comp algorithm is able to handle a batch of updates and is specifically designed for a periodic view maintenance policy. The message cost of our algorithm for a batch of updates to any of the data sources is $O(n)$.

5. *IVM formulae for* `member` *and* `not member` *functions*;

For ease of discussion, we use $\wedge$ and $\bar{\wedge}$ to denote expressions with `member` and `not member` functions, *i.e.* $\mathtt{D1} \wedge \mathtt{D2} = [x|x \leftarrow \mathtt{D1}; \mathtt{member\ D2}\ x]$ and $\mathtt{D1} \bar{\wedge} \mathtt{D2} = [x|x \leftarrow \mathtt{D1}; \mathtt{not}\ (\mathtt{member\ D2}\ x)]$. The IVM formulae for these two functions are given below, in which the function `countNum a D` returns the number of occurrences of the data item `a` in `D`, *i.e.* `countNum a D` = `count [x | x ← D; x=a]`. We can see that all post-update data sources are required in the IVM formulae.

$$
\begin{array}{l}
\mathtt{v} = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D1}; \mathtt{member\ D2\ x}] \\
\ \mathtt{let\ r1} = [x|x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{new})] \\
\qquad \mathtt{r2} = \triangledown\mathtt{D2}\ \bar{\wedge}\ \mathtt{D2}^{new} \\
\qquad \triangle\mathtt{v} = (\triangle\mathtt{D1} \wedge \mathtt{D2}^{new} -\!- \triangle\mathtt{D1} \wedge \mathtt{r1}]) \mathbin{++} \mathtt{D1}^{new} \wedge \mathtt{r1} \\
\qquad \triangledown\mathtt{v} = (\triangledown\mathtt{D1} \wedge \mathtt{D2}^{new} -\!- \triangledown\mathtt{D_1} \wedge \mathtt{r1}) \mathbin{++} (\mathtt{D1}^{new} \wedge \mathtt{r2} -\!- \triangle\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} \triangledown\mathtt{D_1} \wedge \mathtt{r_2} \\
\mathtt{v} = [\mathtt{x}|\mathtt{x} \leftarrow \mathtt{D1}; \mathtt{not(member\ D2\ x)}] \\
\ \mathtt{let\ r1} = [x|x \leftarrow \triangle\mathtt{D2}; (\mathtt{countNum}\ x\ \ \triangle\mathtt{D2}) = (\mathtt{countNum}\ x\ \mathtt{D2}^{new})] \\
\qquad \mathtt{r2} = \triangledown\mathtt{D2}\ \bar{\wedge}\ \mathtt{D2}^{new} \\
\qquad \triangle\mathtt{v} = (\triangle\mathtt{D1}\ \bar{\wedge}\ \mathtt{D2}^{new} -\!- \triangle\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} \mathtt{D1}^{new} \wedge \mathtt{r2} \\
\qquad \triangledown\mathtt{v} = (\triangledown\mathtt{D1}\ \bar{\wedge}\ \mathtt{D2}^{new} -\!- \triangledown\mathtt{D1} \wedge \mathtt{r2}) \mathbin{++} (\mathtt{D1}^{new} \wedge \mathtt{r1} -\!- \triangle\mathtt{D1} \wedge \mathtt{r1}) \mathbin{++} \triangledown\mathtt{D1} \wedge \mathtt{r1}
\end{array}
$$

3.2 IVM for Schema Transformation Pathways

Having defined the IVM formulae for each SIQL query, the update to a construct created by a single add transformation step is obtained by applying the appropriate formulae to the step's query. Our IVM procedure for a single transformation step is IVM4AStep(cd, ts) and its output is the change to the construct created by step ts based on the changes cd to ts's data sources. After obtaining the change to all the constructs created by a transformation pathway, the view created by the transformation pathway is incrementally maintained.

However, as discussed above, the post-update data sources and the view itself are required by some IVM formulae. In a general transformation pathway, some intermediate constructs might be virtual. If a required data collection is unavailable, *i.e.* not materialised, the IVM4AStep procedure cannot be applied.

Thus, in order to apply the IVM4AStep procedure along a transformation pathway, we have to precheck each add transformation in the pathway. If a virtual data collection is required by the IVM formula for a transforation step, we must firstly recover this data collection and store it in the data warehouse. This precheck only needs to be performed once for each transformation pathway in a data warehouse, unless the transformation pathway evolves due to the evolution of a data source schema. This materialisation increases the storage overhead of the data warehouse, but does not increase the message cost of the IVM process since these materialised constructs are also maintainable by using the IVM process along the transformation pathway.

Alternatively, we can use AutoMed's Global Query Processor (GQP) to evaluate the extent of a virtual construct during the IVM process so as to avoid increasing persistent storage overheads. However, since it is based on post-update data sources, AutoMed's GQP can only recover a post-update view. If a view is used in an IVM formula, this means the view is *before* the update, which cannot be recovered by AutoMed's GQP.

We now give an example of prechecking a transformation pathway. In Section 2.2, the transformation pathway generating the construct ⟨⟨Course, Avg⟩⟩ in the global schema can be expressed as the following sequence of view definitions, where the intermediate constructs $\texttt{v1}, \ldots, \texttt{v4}$ and $\langle\langle\mathsf{Details}, \mathsf{Mark}\rangle\rangle$ are virtual:

$$\begin{aligned}
&\texttt{v1} && = [\{\texttt{'IS'},\texttt{k1},\texttt{k2},\texttt{x}\} \mid \{\texttt{k1},\texttt{k2},\texttt{x}\} \leftarrow \langle\langle\mathsf{IStab}, \mathsf{Mark}\rangle\rangle]\\
&\texttt{v2} && = [\{\texttt{'MA'},\texttt{k1},\texttt{k2},\texttt{x}\} \mid \{\texttt{k1},\texttt{k2},\texttt{x}\} \leftarrow \langle\langle\mathsf{MAtab}, \mathsf{Mark}\rangle\rangle]\\
&\langle\langle\mathsf{Details}, \mathsf{Mark}\rangle\rangle && = \texttt{v1} \mathbin{++} \texttt{v2}\\
&\texttt{v3} && = \texttt{map}\ (\lambda\{\texttt{k},\texttt{k1},\texttt{k2},\texttt{x}\}.\{\{\texttt{k},\texttt{k1}\},\texttt{x}\})\ \langle\langle\mathsf{Details}, \mathsf{Mark}\rangle\rangle\\
&\texttt{v4} && = \texttt{gc avg v3}\\
&\langle\langle\mathsf{Course}, \mathsf{Avg}\rangle\rangle && = \texttt{map}\ (\lambda\{\{\texttt{x},\texttt{y}\},\texttt{z}\}.\{\texttt{x},\texttt{y},\texttt{z}\})\ \texttt{v4}
\end{aligned}$$

In order to incrementally maintain $\langle\langle\mathsf{Course}, \mathsf{Avg}\rangle\rangle$, the intermediate views $\texttt{v3}$ and $\texttt{v4}$ must be materialised (based on the IVM formulae for grouping functions). For example, supposes that an update to the data sources is a tuple inserted into $\langle\langle\mathsf{IStab}, \mathsf{Mark}\rangle\rangle$, $\triangle\langle\langle\mathsf{IStab}, \mathsf{Mark}\rangle\rangle = \{\texttt{'ISC01'},\texttt{'ISS05'},\texttt{80}\}$. Following on the transformation pathway, we obtain the changes to the intermediate views as follows:

$$\begin{aligned}
&\triangle\texttt{v1} && = \{\texttt{'IS'},\texttt{'ISC01'},\texttt{'ISS05'},\texttt{80}\}\\
&\triangle\langle\langle\mathsf{Details}, \mathsf{Mark}\rangle\rangle && = \{\texttt{'IS'},\texttt{'ISC01'},\texttt{'ISS05'},\texttt{80}\}\\
&\triangle\texttt{v3} && = \{\texttt{'IS'},\texttt{'ISC01'},\texttt{80}\}
\end{aligned}$$

Since the extents of $\texttt{v3}$ and $\texttt{v4}$ are recovered, changes to $\texttt{v4}$ can be obtained by using the IVM formulae for grouping functions, and then be used to compute changes to $\langle\langle\mathsf{Course}, \mathsf{Avg}\rangle\rangle$ by using the IVM formulae for map.

However, the post-update extent of $\texttt{v3}$ can be recovered by AutoMed's GQP, and using the inverse query of $\texttt{map}\ (\lambda\{\{\texttt{x},\texttt{y}\},\texttt{z}\}.\{\texttt{x},\texttt{y},\texttt{z}\})\ \texttt{v4}$, the pre-update extent of $\texttt{v4}$ can also be recovered as $\texttt{v4} = \texttt{map}\ (\lambda\{\texttt{x},\texttt{y},\texttt{z}\}.\{\{\texttt{x},\texttt{y}\},\texttt{z}\})\ \langle\langle\mathsf{Course}, \mathsf{Avg}\rangle\rangle$. Thus, in practice, no intermediate view needs to be materialised for incrementally maintaining $\langle\langle\mathsf{Course}, \mathsf{Avg}\rangle\rangle$ along the pathway. In the future, we will investigate these avoidable materializations more generally, so as to apply them in our IVM algorithms.

4 Concluding Remarks

AutoMed schema transformation pathways can be used to express data transformation and integration processes in heterogeneous data warehousing environments. This paper has discussed techniques for incremental view maintenance along such pathways and thus addresses the general IVM problem for heterogeneous data warehouses. We have developed a set of IVM formulae. Based on these formulae, our algorithms perform an IVM process along a schema transformation pathway. We are currently implementing the algorithms as part of a broader bioinformatics data warehousing project (BIOMAP).

One of the advantages of AutoMed is that its schema transformation pathways can be readily evolved as the data warehouse evolves [8]. In this paper we have shown how to perform IVM along such evolvable pathways.

Although this paper has used IQL as the query language in which transformations are specified, our algorithms are not limited to one specific data model or query language, and could be applied to other query languages involving common algebraic operations such as selection, projection, join, aggregation, union and difference.

Finally, since our algorithms consider in turn each transformation step in a transformation pathway in order to compute data changes in a stepwise fashion, they are useful not only in data warehousing environments, but also in any data transformation and integration framework based on sequences of primitive schema transformations. For example, Zamboulis and Poulovassilis present an approach for integrating heterogeneous XML documents using the AutoMed toolkit [18, 19]. A schema is automatically extracted for each XML document and transformation pathways are applied to these schemas. McBrien and Poulovassilis also discusses how AutoMed can be applied in peer-to-peer data integration settings [14]. Thus, the IVM approach we have discussed in this paper is readily applicable in peer-to-peer and semi-structured data integration environments.

References

1. D. Agrawal, A. E. Abbadi, A. K. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proc. ACM SIGMOD'97*, pages 417–427. ACM Press, 1997.
2. J. Albert. Algebraic properties of bag data types. In *Proc. VLDB'91*, pages 211–219, 1991.
3. M. Boyd, S. Kittivoravitkul, and C. Lazanitis. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE'04*, LNCS 3084, 2004.
4. P. Buneman *et al*. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
5. L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proc. ACM SIGMOD'96*, pages 469–480, 1996.
6. G. Dong. Incremental maintenance of recursive views: A survey. In *Materialized Views: Techniques, Implementations, and Applications*, pages 159–162. The MIT Press, 1999.
7. H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. DOLAP'03*, pages 86–93. ACM Press, 2003.
8. H. Fan and A. Poulovassilis. Schema evolution in data warehousing environments — a schema transformation-based approach. In *Proc. ER'04*, LNCS, pages 639–653, 2004.
9. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. ACM SIGMOD'95*, pages 328–339. ACM Press, 1995.
10. A. Gupta and I. S. Mumick. Maintenance polices. In *Materialized Views: Techniques, Implementations, and Applications*, pages 9–11. The MIT Press, 1999.
11. Ashish Gupta, H. V. Jagadish, and Inderpal Singh Mumick. Data integration using self-maintainable views. In *Extending Database Technology*, pages 140–144, 1996.
12. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report 20, Automed Project, 2003.
13. P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer, 1999.
14. P. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P, Berlin, Germany, September 7-8*, LNCS. Springer, 2003.
15. A. Poulovassilis. A Tutorial on the IQL Query Language. Technical Report 28, Automed Project, 2004.
16. D. Quass. Maintenance expressions for views with aggregation. In *Proc VIEW'96*, pages 110–118, 1996.
17. D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. PDIS'96*, pages 158–169, 1996.
18. L. Zamboulis. XML data integration by graph restrucring. In *Proc. BNCOD'04*, volume 3112 of *LNCS*, pages 57–71. Springer-Verlag, 2004.
19. L. Zamboulis and A. Poulovassilis. Using AutoMed for XML data transformation and integration. In *Proc. DIWeb'04*, pages 58–69, 2004.
20. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD'95*, pages 316–327, 1995.
21. Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.