Using Schema Transformation Pathways for Data Lineage Tracing

Hao Fan, Alexandra Poulovassilis

School of Computer Science and Information Systems, Birkbeck College,
University of London, Malet Street, London WC1E 7HX
{hao,ap}@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. Tracing the lineage of the integrated data is one of the problems being addressed in data warehousing research. This paper presents a data lineage tracing 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

A data warehousing system collects data from distributed, autonomous and heterogeneous data sources into a central repository to enable analysis and mining of the integrated information. However, sometimes what we need is not only to analyse the data in the integrated database, but also to investigate how certain integrated information was derived from the data sources, which is the problem of *data lineage tracing* (DLT). Supporting DLT in data warehousing environments has a number of applications: in-depth data analysis, on-line analysis mining (OLAM), scientific databases, authorization management, and materialized view schema evolution [2, 18, 8, 13, 9].

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 [11] 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 focus on how AutoMed metadata can be used for tracing the lineage of data in an integrated database.

The outline of this paper is as follows. Section 2 gives a review of related work. Section 3 gives an overview of AutoMed, as well as a data integration example. Section 4 presents our DLT techniques, including the DLT formulae developed to handle virtual

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

intermediate lineage data and the DLT algorithm operating along a general schema transformation pathway. Section 5 gives our concluding remarks.

2 Related Work

The problem of data lineage tracing in data warehousing environments has been formally studied by Cui $et\ al.$ in [8, 6, 7]. In particular, the fundamental definitions regarding data lineage, including *tuple derivation for an operator* and *tuple derivation for a view*, were developed in [8], as were methods for derivation tracing with both set and bag semantics. Their work has addressed the derivation tracing problem using bag semantics and has provided the concept of *derivation set* and *derivation pool* for tracing data lineage with duplicate elements. Reference [6] also introduces a way to trace data lineage for complex views in data warehouses. However, the approach is limited to the relational data model.

Another fundamental concept of data lineage is discussed by Buneman $et\ al.$ in [4], namely the difference between "why" provenance and "where" provenance. Why-provenance refers to the source data that had some influence on the existence of the integrated data. Where-provenance refers to the actual data in the sources from which the integrated data was extracted.

In our approach, both why- and where-provenance are considered, using bag semantics. Our previous work [10] defines the notions of *affect-pool* and *origin-pool* for data lineage tracing in AutoMed — the former derives all of the source data that had some influence on the tracing data, while the latter derives the specific data in the sources from which the tracing data is extracted. In that work we develop formulae for deriving the affect-pool and origin-pool of a data item in the extent of a materialised schema construct created by a single schema transformation step. Our DLT approach is to apply these formulae on each transformation step in a transformation pathway in turn, so as to obtain the lineage data in stepwise fashion.

Cui and Widom in [7] also discuss the problem of tracing data lineage for general data warehousing transformations, that is, the considered operators and algebraic properties are no longer limited to relational views. However, without a framework for expressing general transformations in heterogeneous database environments, most of algorithms in [7] are recalling the view definition and examining each item in the data source to decide if the item is in the data lineage of the data being traced. This can be expensive if the view definition is a complex one and enumerating all items in the data source is impractical for large data sets.

Reference [18] proposes a general framework for computing *fine-grained* data lineage, $i.e.$ a specific derivation in the data source, using a limited amount of information, *weak* and *verified inversion*, about the processing steps. Based on weak and verified inversion functions, which must be specified by the transformation definer, the paper defines and traces data lineage for each transformation step in a database visualization environment. However, the system cannot obtain the exact lineage data, only a number of guarantees about the lineage is provided. Further, specifying weak and verified inversion functions for each transformation step is onerous work for the data warehouse definer. Moreover, the DLT procedures cannot straightforwardly be reused when the data warehouse evolves. Our approach considers the problem of data lineage tracing at the tuple level and computes the exact lineage data. Moreover, AutoMed's ready support for schema evolution (see [12]) means that our DLT algorithms can be reapplied if schema transformation pathways evolve.

One limit of our earlier work described in [10] is that we assumed the transformation pathway used by our DLT algorithm is fully materialised, $i.e.$ new schema constructs created along the pathway are materialised. In practice, we need to handle the situation of virtual or partially materialised transformation pathways, in which intermediate schema constructs may or may not be materialised. In this paper, we describe an approach for tracing data lineage along a general schema transformation pathway.

3 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 [19], flat-file [3] and multidimensional [11] 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.

3.1 Simple IQL

In order to illustrate our DLT 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

[2] IQL is a comprehensions-based functional query language. Such languages subsume query languages such as SQL and OQL in expressiveness [5]. We refer the reader to [14, 17] for details of IQL and references to work on comprehension-based functional query languages.

of transformations with SIQL queries on intermediate schema constructs. We stress that although illustrated within a particular query language syntax, our DLT 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 $\mathtt{D}$, $\mathtt{D}_1$ ..., $\mathtt{D}_n$ 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 [1]. 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 [5] 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 $\pi_A(\sigma_C(D_1 \bowtie ... \bowtie D_n))$ 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:

$\mathtt{v1} = \mathtt{map}\ \ (\lambda\{\mathtt{x1,x2,x3}\}.\{\mathtt{x2,x3}\})\ \ \mathtt{D}$
$\mathtt{v}\ \ = \mathtt{gc}\ \ \mathtt{f}\ \ \mathtt{v1}$

3.2 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, 19].

In our 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 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 a non-key attribute of R is the projection of R onto $k_1, ..., k_n, a$. We refer the reader to [15] for an encoding of a richer relational data model, including the modelling of constraints.

Suppose that $\mathsf{MAtab}(\underline{\mathsf{CID}}, \underline{\mathsf{SID}}, \mathsf{Mark})$ and $\mathsf{IStab}(\underline{\mathsf{CID}}, \underline{\mathsf{SID}}, \mathsf{Mark})$ 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 $\mathsf{CourseSum}(\underline{\mathsf{Dept}}, \underline{\mathsf{CID}}, \mathsf{Total}, \mathsf{Avg})$ is in the data warehouse which gives the total and 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	⟨⟨CourseSum⟩⟩	`distinct [{k,k1}|{k,k1,k2}` ← ⟨⟨Details⟩⟩]
addAtt	⟨⟨CourseSum, Total⟩⟩	`[{x,y,z}|{{x,y},z}` ← (`gc sum`
		`[{{k,k1},x}|{k,k1,k2,x}` ← ⟨⟨Details, Mark⟩⟩])];
addAtt	⟨⟨CourseSum, 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 lineage tracing, 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

`[{x,y,z}|{{x,y},z}` ← (`gc avg [{{k,k1},x}| {k,k1,k2,x}` ← ⟨⟨Details, Mark⟩⟩])]

is decomposed into following sequence of SIQL queries:

`v1` = `map` (λ`{k,k1,k2,x}.{{k1,k2},x}`) ⟨⟨Details, Mark⟩⟩

`v2` = `gc avg v1`

`v` = `map` (λ`{{x,y},z}.{x,y,z}`) `v2`

In the rest of the paper, our discussion assumes that all queries in transformation steps are SIQL queries.

4 Data Lineage Tracing with AutoMed Schema Transformations

In heterogenous data integration environments, the data transformation and integration processes can be described using AutoMed schema transformation pathways (see [11]). Our DLT approach is to use the individual steps of these pathways to compute the lineage data of the tracing data by traversing the pathways in reverse order one step at a time. In particular, suppose a data source `LD` with schema `LS` is transformed into

v	$DL(t)$
group D	$[\{x,y\}\|\{x,y\} \leftarrow D; x = \overline{a}]$
sort D	$D\|t$
distinct D	$D\|t$
aggFun D	D
gc aggFun D	$[\{x,y\}\|\{x,y\} \leftarrow D; x = \overline{a}]$
$\mathtt{D}_1$ ++ $\mathtt{D}_2$ ++ ... ++ $\mathtt{D}_n$	$\forall i.D_i\|t$
$\mathtt{D}_1 -- \mathtt{D}_2$	$D_1\|t, D_2$
$[\overline{\mathtt{x}}\|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$	$\forall i.[\overline{x_i}\|\overline{x_i} \leftarrow D_i; \overline{x_i} = ((\lambda\overline{x}.\overline{x_i})\ t)]$
$[\overline{\mathtt{x}}\|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ member $\mathtt{D}_2\ \overline{\mathtt{y}}]$	$D_1\|t, [\overline{y}\|\overline{y} \leftarrow D_2; \overline{y} = ((\lambda\overline{x}.\overline{y})\ t)]$
$[\overline{\mathtt{x}}\|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ not(member $\mathtt{D}_2\ \overline{\mathtt{y}}$)]	$D_1\|t, D_2$
map $(\lambda\overline{\mathtt{x}}.\mathtt{e})$ D	$[\overline{x}\|\overline{x} \leftarrow D, e = t]$

Table 1. DLT Formulae for MtMs

a global database GD with schema GS, and the transformation pathway LS $\rightarrow$ GS is $ts_1, ..., ts_n$. Given tracing data td belonging to the extent of some schema construct in GD, we firstly find the transformation step ts_i which creates that construct and obtain td's lineage, dl_i, from ts_i. We then continue by tracing the lineage of dl_i from the remaining transformation pathway $ts_1, \ldots, ts_{i-1}$. We continue in this fashion, until we obtain the final lineage data from the data source LD.

Since delete transformations do not create schema constructs, they can be ignored in the DLT process. Tracing data lineage with respect to a transformation rename(O, O') is simple — the lineage data in O is the same as the tracing data in O'. It only remains to consider add transformations. A single add transformation step can be expressed as v=q, in which v is the new schema construct created by the transformation and q is an SIQL query over the current schema constructs. We have developed a DLT formula for each type of SIQL query which, given tracing data in v, evaluates the lineage of this data from the extents of the schema constructs referenced in v=q. If these extents and the tracing data are both materialised, Table 1 gives the DLT formulae for tracing the affect-pool of a tuple t, $DL(t)$. The DLT formulae for tracing the origin-pool are similar and we refer the reader to [10] for a discussion of the difference between the affect-pool and the origin-pool.

In Table 1, $D|t$ denotes all instances of the tuple t in the bag D (i.e. the result of the query $[x|x \leftarrow D; x = t]$). Since the results of queries of the form group D and gc f D are a collection of pairs, in the DLT formulae for these two queries we assume that the tracing tuple t is of the form $\{\overline{a}, \overline{b}\}$.

The DLT formulae in Table 1 either provide a *derivation tracing query* [8] specifying the lineage data of t or, in some cases, give the lineage data directly. If a formula returns a derivation tracing query, we need to evaluate the query to obtain the lineage data. If a formula returns the lineage data directly, no such evaluation is needed.

If all schema constructs created by add transformations are materialised, a simple way to trace the lineage of data in the global database GD is to apply the above DLT formulae on each transformation step in the transformation LS $\rightarrow$ GS in reverse from GS, finally ending up with the lineage data in the original data source LD. Such a DLT method has been described in our previous work [10]. However, in general transformation pathways not all schema constructs created by add transformations will be materialised, and the above simple DLT approach is no longer applicable because it does not obtain lineage data from a virtual schema construct. In this paper, we propose a DLT approach that handles such general transformation pathways.

4.1 The Approach

One approach to solving the problem of virtual schema constructs would be to use AutoMed's Global Query Processor to evaluate the query creating the virtual construct and compute its extent, so that the above simple DLT approach could be applied. However, this approach is impractical due to the space and time overheads it incurs.

Instead, our approach is to use a data structure, Lineage, to denote lineage data from the extent of a schema construct. If the construct is materialised, Lineage contains the actual lineage data. If the construct is virtual, Lineage contains relevant information for deriving the lineage data. This information will be used by subsequent DLT steps to evaluate the final lineage data. Each Lineage object contains five attributes: (i)data, which is a collection of materialised lineage data or, if the lineage data is virtual, the value $null$; (ii) construct, which is the name of the schema construct whose extent contains the lineage data; (iii) isVirtual, stating if the lineage data is virtual or not; (iv) elemStruct, describing the structure of the data in the extent of a virtual schema construct, $e.g.$ a 2-item tuple `{x1,x2}`, or a 3-item tuple `{x1,x2,x3}`; (v) constraint, expressing the constraint specifying the lineage data from a virtual schema construct.

For example, suppose lineage data in a schema construct `D` is derived from the query $[\{x,y\}|\{x,y\} \leftarrow \texttt{D}; x = 5]$, and lp is a Lineage object expressing the lineage data. If `D=[{1,2},{5,1},{5,2},{3,1}]` is materialised, then lp will be: lp.data = `[{5,1},{5,2}]`; lp.construct= "D"; lp.isVirtual = `false`; lp.elemStruct= `null`; and lp.constraint= `null`. On the other hand, if `D` is a virtual schema construct, then lp will be: lp.data = `null`; lp.construct= "D"; lp.isVirtual = `true`; lp.elemStruct = "`{x,y}`"; and lp.constraint= "`x=5`".

We denote by `O`|`dl` a Lineage object in which `O` is the name of the schema construct and `dl` is the data lineage. If the lineage data is materialised, `dl` will be the data itself, otherwise `dl` will be the form of (S, C), where S denotes the *elemStruct* and C the *constraint*. For example, the above two Lineage objects are denoted by `D`|`[{5,1},{5,2}]` and `D`|(`{x,y}`,`x=5`), respectively.

4.2 The DLT Formulae

It is necessary that our DLT formulae can handle the following four cases: MtMs — both the tracing data and the source data are materialised; MtVs — the tracing data is materialised and the source data is virtual; VtMs — the tracing data is virtual and the source data is materialised; and VtVs — both the tracing data and the source data are virtual. The DLT formulae for the case of MtMs were given in Table 1, and from these we have derived the DLT formulae for the other three cases:

Case MtVs. There were two kinds of DLT formulae in Table 1: tracing queries and real lineage data. Since with MtVS the source data is virtual, we cannot evaluate tracing queries and so Lineage objects are required to store the information about these

v	$DL(t)$
group D	$D \vert (\{x,y\}, x=\overline{a})$
sort D	$D \vert t$
distinct D	$D \vert t$
aggFun D	$D \vert (any, true)$
gc aggFun D	$D \vert (\{x,y\}, x=\overline{a})$
$\mathtt{D}_1$ ++ $\mathtt{D}_2$ ++ ... ++ $\mathtt{D}_n$	$\forall i.D_i \vert t$
$\mathtt{D}_1 -- \mathtt{D}_2$	$D_1 \vert t$, $D_2 \vert (any, true)$
$[\overline{\mathtt{x}} \vert \overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$	$\forall i.D_i \vert (\overline{x_i}, \overline{x_i} = ((\lambda \overline{x}.\overline{x_i})\ t))$
$[\overline{\mathtt{x}} \vert \overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ member $\mathtt{D}_2\ \overline{\mathtt{y}}]$	$D_1 \vert t$, $D_2 \vert (\overline{y}, \overline{y} = ((\lambda \overline{x}.\overline{y})\ t))$
$[\overline{\mathtt{x}} \vert \overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ not(member $\mathtt{D}_2\ \overline{\mathtt{y}})]$	$D_1 \vert t$, $D_2 \vert (any, true)$
map $(\lambda \overline{\mathtt{x}}.\mathtt{e})$ D	$D \vert (\overline{x}, e=t)$

Table 2. DLT Formulae for MtVs

queries. For example, the tracing query $[\{x,y\}|\{x,y\} \leftarrow D; x=\overline{a}]$ is expressed as $D|(\{x,y\}, x=\overline{a})$. In the case of real lineage data, the lineage data might be the tracing data, t, itself or all the items in a source collection D. If the lineage data is t, it is available no matter whether D is materialised or not. If the the lineage data is all items in a virtual collection D, it is expressed by D | (any,true). Table 2 illustrates the DLT formulae for the case of MtVs.

Case VtMs. Virtual tracing data can be created by virtual source data. In particular, there are three kinds of virtual lineage data created in Table 2: (any,true), ({x,y},x=$\overline{a}$), and ($\overline{\mathtt{x}}$,e=t)[3]. The DLT formulae for VtMs can be derived by applying these three kinds of virtual tracing data to the formulae given in Table 1. In this case, all source data is materialised, there is no virtual intermediate lineage data created.

For example, suppose the query is v=group D. If the virtual tracing tuple t is (any,true), the lineage data $DL(t)$ is all data in D, $i.e.$ $DL(t) = \mathtt{D}$. If t is ({x,y}, x=$\overline{a}$), $DL(t)$ is all tuples in D with first component equal to $\overline{a}$, which is the result of the query $[\{x,y\}|\{x,y\} \leftarrow \mathtt{D}; x=\overline{a}]$. If t is ($\overline{\mathtt{x}}$,e=t), $DL(t)$ is all tuples in D with first component equal to the first component of the tracing data t, which is the result of the query $[\{x,y\}|\{x,y\} \leftarrow \mathtt{D}; member\ [first\ \overline{x}|\overline{x} \leftarrow \mathtt{v}; e=t]]$. We can see that the virtual view, v, is used in this query. Since the source data is materialised, we can easily recover v and evaluate the tracing query.

Table 3 gives the whole list of formulae for the case of VtMs with virtual tracing data of the form ($\overline{\mathtt{x}}$,e=t). The formulae for the other two kinds of virtual tracing data can easily be derived.

Case VtVs. The DLT formulae for VtVs are similar to the formulae for VtMs but in this case the source data are unavailable. Thus, we use Lineage objects to store the virtual intermediate lineage data.

[3] Note that in Table 2 the lineage data $(\overline{x_i}, \overline{x_i} = ((\lambda \overline{x}.\overline{x_i})\ t))$ and $(\overline{y}, \overline{y} = ((\lambda \overline{x}.\overline{y})\ t))$ in the 8th and 9th lines are not virtual. Since t is real data and variable tuple $\overline{x}$ contains all variables appearing in $\overline{x_i}$, the expression $(\lambda \overline{x}.\overline{x_i})\ t$ returns real data too. For example, supposing $\overline{x} = \{x_1, x_2, x_3\}$, $\overline{x_i} = \{x_1, x_3\}$, and $t = \{1,2,3\}$, then $(\lambda \overline{x}.\overline{x_i})\ t = (\lambda\{x_1,x_2,x_3\}.\{x_1,x_3\})\ \{1,2,3\} = \{1,3\}$.

v	$DL(t)$
group D	$[\{x,y\} \mid \{x,y\} \leftarrow D; member\ [first\ \overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t]\ x]$
sort D	$[\overline{x} \mid \overline{x} \leftarrow D; e=t]$
distinct D	$[\overline{x} \mid \overline{x} \leftarrow D; e=t]$
aggFun D	D
gc aggFun D	$[\{x,y\} \mid \{x,y\} \leftarrow D; member\ [first\ \overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t]\ x]$
D_1 ++ D_2 ++ ... ++ D_n	$\forall i.[\overline{x} \mid \overline{x} \leftarrow D_i; e=t]$
D_1 −− D_2	$D_1 \mid [\overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t], D_2$
$[\overline{\mathrm{x}} \mid \overline{\mathrm{x}_1} \leftarrow \mathrm{D}_1; \ldots; \overline{\mathrm{x}_n} \leftarrow \mathrm{D}_n; \mathrm{C}]$	$\forall i.[\overline{x_i} \mid \overline{x_i} \leftarrow D_i;$ $member\ (map\ (\lambda \overline{x}.\overline{x_i})\ [\overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t])\ \overline{x_i}]$
$[\overline{\mathrm{x}} \mid \overline{\mathrm{x}} \leftarrow \mathrm{D}_1; \mathrm{member}\ \mathrm{D}_2\ \overline{\mathrm{y}}]$	$[\overline{x} \mid \overline{x} \leftarrow D_1; member\ D_2\ \overline{y}; e=t]$, $[\overline{y} \mid \overline{y} \leftarrow D_2; member\ (map\ (\lambda \overline{x}.y)\ [\overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t])\ \overline{y}]$
$[\overline{\mathrm{x}} \mid \overline{\mathrm{x}} \leftarrow \mathrm{D}_1; \mathrm{not}(\mathrm{member}\ \mathrm{D}_2\ \overline{\mathrm{y}})]$	$D_1 \mid [\overline{x} \mid \overline{x} \leftarrow \mathrm{v}; e=t], D_2$
map $(\lambda \overline{\mathrm{x}_1}.\mathrm{e}_1)$ D	$[\overline{x_1} \mid \overline{x_1} \leftarrow D; e=t]$

Table 3. DLT Formulae for VtMs with tracing data $(\overline{x}, e=t)$

For example, suppose the query is v=group D. If the virtual tracing tuple t is (any,true), the virtual lineage data $DL(t)$ is D|(any,true). If t is ({x,y},x=$\overline{a}$), the virtual $DL(t)$ is D|({x,y},x=$\overline{a}$). If t is ($\overline{\mathrm{x}}$, e=t), the virtual $DL(t)$ is D|({x,y}, member[first $\overline{\mathrm{x}} \mid \overline{\mathrm{x}} \leftarrow$ v; e=t] x). Note that, the virtual view v is used in this virtual lineage data expression. However, since the source data D is virtual, we cannot recover v by just evaluating the query v=group D. In this case, AutoMed's Global Query Processor can be used to materialise v. Once v is materialised, the virtual tracing data t can also be recovered and this situation reverts to the case of MtVs which we discussed earlier. Alternatively, the view definition of v can be propagated through the remaining DLT steps until the end of the process. So far we have only implemented the first approach and it remains to implement the second approach and investigate their trade-offs.

4.3 DLT for General Transformation Pathways

Having obtained the DLT formulae for above four cases, lineage data based on a single transformation step is obtained by applying the appropriate formula to the step's query. Our DLT procedure for a single transformation step is DLT4AStep(td, ts) and its output is the lineage of td in ts's data sources i.e. a list of Lineage objects which might contain either materialised or virtual lineage data. In our DLT algorithms for a general transformation pathway, there are two further procedures: tracing the lineage of a single tuple along a transformation pathway and tracing the lineage of a set of tuples along a transformation pathway. This is because the lineage of one Lineage object based on a single transformation step might be a list of Lineage objects, if the transformation step has multiple data sources. Figure 1 gives the two procedures: oneDLT4APath$(td, [ts_1, ..., t_n])$ traces the lineage of a single tracing tuple td along a transformation pathway $[ts_1, ..., t_n]$, and listDLT4APath$([td_1, ..., td_m], [ts_1, ..., ts_n])$ traces the lineage of a list of tracing tuples along a transformation pathway.

```
Proc oneDLT4APath(td, [ts_1, ..., ts_n])
{  lpList = ø;
   for i = n downto 1, do
     if (td.construct is created by ts_i)
       Num = i;
       lpList = DLT4AStep(td, ts_i);
       continue; //* End the for loop
   restTP = [ts_1, ..., ts_Num];
   return listDLT4APath(lpList, restTP);
}

Proc listDLT4APath([td_1, ..., td_m], [ts_1, ..., ts_n])
{  lpList = ø;
   for i = 1 to m, do
     lpList = merge(lpList, oneDLT4APath(td_i, [ts_1, ..., ts_n]));
   return lpList;
}
```

Fig. 1. DLT Algorithms for a general transformation pathway

oneDLT4APath firstly finds the transformation step, ts_i, which creates the schema construct containing td and then calls the procedure DLT4AStep to obtain the lineage of td based on this transformation step. DLT4AStep returns a list of Lineage objects. After that, the procedure oneDLT4APath calls the procedure listDLT4APath to further trace the lineage of this list of Lineage objects along the rest of the transformation pathway (i.e. the steps prior to ts_i). oneDLT4APath also returns a list of Lineage objects. listDLT4APath itself calls oneDLT4APath for each item td_i in the tracing data list to find the entire lineage of the whole list based on the transformation pathway. The merge function is used to avoid duplication of lineage data: A tuple, dl, might be in the lineage of two different tracing tuples, td_i and td_j ($i \neq j$). If dl and all its copies in a source collection have already been added to $lpList$ as the lineage of td_i, we do not add them again into $lpList$ as the lineage of td_j.

The complexity of the overall DLT process is $O(n \times m)$ where n is the number of add transformations in the transformation pathway and m is the number of different schema constructs referenced in the pathway.

4.4 Example

We use the example described in Section 3.2 to illustrate our DLT approach. Recall that some queries appearing in the example are not SIQL queries but general IQL queries. In such situations, we firstly decompose these IQL queries into sequences of SIQL queries.

Supposing $td = \{$`'MA','MAC01',81`$\}$ is a tuple in the extent of the construct ⟨⟨CourseSum, Avg⟩⟩ in the global database GD, the transformation pathway generating ⟨⟨CourseSum, Avg⟩⟩ construct can be expressed as following sequence of view definitions, where the intermediate constructs `v1`, . . ., `v4` and ⟨⟨Details, Mark⟩⟩ are virtual:

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

Traversing this transformation pathway in reverse, we obtain td's lineage data, dl, with respect to each view as follows:

$$
\begin{aligned}
&\quad td && = \langle\langle \mathsf{CourseSum}, \mathsf{Avg}\rangle\rangle|\{\texttt{'MA'},\texttt{'MAC01'},\texttt{81}\}\\
&\overset{\mathsf{MtVs}}{\Longrightarrow} \texttt{v4}\,|\,\texttt{dl} && = \texttt{v4}|\{\{\texttt{'MA'},\texttt{'MAC01'}\},\texttt{81}\}\\
&\overset{\mathsf{MtVs}}{\Longrightarrow} \texttt{v3}\,|\,\texttt{dl} && = \texttt{v3}|(\{\texttt{x},\texttt{y}\},\texttt{x=}\{\texttt{'MA'},\texttt{'MAC01'}\})\\
&\overset{\mathsf{VtVs}}{\Longrightarrow} \langle\langle \mathsf{Details}, \mathsf{Mark}\rangle\rangle|\texttt{dl} && = \langle\langle \mathsf{Details}, \mathsf{Mark}\rangle\rangle|(\{\texttt{k},\texttt{k1},\texttt{k2},\texttt{x}\},\{\texttt{k='MA';k1='MAC01'}\})\\
&\overset{\mathsf{VtVs}}{\Longrightarrow} \texttt{v2}\,|\,\texttt{dl} && = \texttt{v2}|(\{\texttt{k},\texttt{k1},\texttt{k2},\texttt{x}\},\{\texttt{k='MA';k1='MAC01'}\}),\\
&\quad \texttt{v1}\,|\,\texttt{dl} && = \texttt{v1}|(\{\texttt{k},\texttt{k1},\texttt{k2},\texttt{x}\},\{\texttt{k='MA';k1='MAC01'}\})\\
&\overset{\mathsf{VtMs}}{\Longrightarrow} \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle|\texttt{dl} && = \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle|(\{\texttt{k1},\texttt{k2},\texttt{x}\},\{\texttt{'MA'='MA';k1='MAC01'}\})\\
&\quad \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle|\texttt{dl} && = \langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle|(\{\texttt{k1},\texttt{k2},\texttt{x}\},\{\texttt{'IS'='MA';k1='MAC01'}\})
\end{aligned}
$$

In conclusion, we can see that the lineage from $\langle\langle \mathsf{IStab}, \mathsf{Mark}\rangle\rangle$ is empty and the lineage form $\langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle$ is obtained by evaluating the final tracing query $[\{\texttt{k1},\texttt{k2},\texttt{x}\} \mid \{\texttt{k1},\texttt{k2},\texttt{x}\} \leftarrow \langle\langle \mathsf{MAtab}, \mathsf{Mark}\rangle\rangle; \texttt{'MA'='MA'}; \texttt{k1='MAC01'}]$.

5 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 tracing data lineage along such pathways and thus addresses the general DLT problem for heterogeneous data warehouses.

We have developed a set of DLT formulae using virtual arguments to handle virtual intermediate schema constructs and virtual lineage data. Based on these formulae, our algorithms perform data lineage tracing along a general schema transformation pathway, in which each add transformation step may create either a virtual or a materialised schema construct. The algorithms described in this paper have been implemented and tested over simple relational data source and integrated schemas. We are currently deploying them 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 [12]. In this paper we have shown how to perform data lineage tracing 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 on collections 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 evaluate lineage data 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, [19, 20] present an approach for integrating heterogeneous XML documents using the AutoMed toolkit. A schema is automatically extracted for each XML document and transformation pathways are applied to these schemas. Reference [16] also discusses how AutoMed can be applied in peer-to-peer data integration settings. Thus, the DLT approach we have discussed in this paper is readily applicable in peer-to-peer and semi-structured data integration environments.

References

1. J. Albert. Algebraic properties of bag data types. In *Proc. VLDB'91*, pages 211–219. Morgan Kaufmann, 1991.
2. P. A. Bernstein and T. Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, 1999.
3. M. Boyd, S. Kittivoravitkul, and C. Lazanitis. AutoMed: A BAV data integration system for heterogeneous data sources. In *Proc. CAiSE'04*, LNCS. Springer-Verlag, 2004.
4. P. Buneman, S. Khanna, and W.C. Tan. Why and Where: A characterization of data provenance. In *Proc. ICDT'01*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001.
5. P. Buneman *et al*. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
6. Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. ICDE'00*, pages 367–378. IEEE Computer Society, 2000.
7. Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proc. VLDB'01*, pages 471–480. Morgan Kaufmann, 2001.
8. Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.
9. C. Faloutsos, H.V. Jagadish, and N.D. Sidiropoulos. Recovering information from summary data. In *Proc. VLDB'97*, pages 36–45. Morgan Kaufmann, 1997.
10. H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. In *Knowledge Transformation for the Semantic Web*, volume 95 of *Frontiers in Artificial Intelligence and Applications*, pages 64–79. IOS Press, 2003.
11. H. Fan and A. Poulovassilis. Using AutoMed metadata in data warehousing environments. In *Proc. DOLAP'03*, pages 86–93. ACM Press, 2003.
12. 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.
13. H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.A. Saita. Improving data cleaning quality using a data lineage facility. In *Proc. DMDW'01*, page 3, 2001.
14. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report 20, Automed Project, 2003.
15. 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.
16. 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.
17. A. Poulovassilis. A Tutorial on the IQL Query Language. Technical Report 28, Automed Project, 2004.
18. A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. ACDE'97*, pages 91–102. IEEE Computer Society, 1997.
19. L. Zamboulis. XML data integration by graph restrucring. In *Proc. BNCOD'04*, volume 3112 of *LNCS*, pages 57–71. Springer-Verlag, 2004.
20. L. Zamboulis and A. Poulovassilis. Using automed for xml data transformation and integration. In *DIWeb*, volume 3084 of *LNCS*, pages 58–69. Springer-Verlag, 2004.