Tracing Data Lineage Using Automed Schema Transformation Pathways

Hao Fan Alexandra Poulovassilis
School of Computer Science and Information Systems
Birkbeck College, University of London, {hao, ap}@dcs.bbk.ac.uk

1. Introduction

A *data warehouse* is a materialized view storing the tuples of the view over a number of data sources. It collects copies of data from remote, distributed, autonomous and heterogeneous data sources into a central repository to enable analysis and mining of the integrated information. Data warehousing is popularly used for on-line analytical processing (OLAP), decision support systems, on-line information publishing and retrieving, and digital libraries. However, sometimes what we need is not only to analyse the data in the warehouse, but also to investigate how certain warehouse information was derived from the data sources. Given a tuple *t* in the warehouse, finding the exact set of source data items from which *t* was derived is termed the *data lineage* problem [CWW00]. Enabling lineage tracing in data warehousing environments brings several benefits and applications, including in-depth data analysis, on-line analysis mining (OLAM) and OLAP, scientific databases, authorization management and materialized view schema evolution [BB99, WS97, Cui01, CWW00, GFS+01, FSJ97].

Automed (*Automatic Generation of Mediator Tools for Heterogeneous database Integration*) is a database transformation and integration system, supporting both virtual and materialized integration of schemas expressed in a variety of modelling languages. This system is being developed in a collaborative EPSRC-funded project between Birkbeck and Imperial Colleges, London (see http://www.ic.ac.uk/Automed).

Common to many methods for integrating heterogeneous data sources is the requirement for logical integration [Hull97] of the data, due to variations in the design of data models for the same universe of discourse. When data is to be shared or exchanged between heterogeneous databases, it is necessary to build a single integrated schema expressed using a *common data model* (CDM). In previous work of the Automed project [PM98, MP99a], a general framework has been developed to support schema transformation and integration in heterogeneous database architectures. The framework consists of a low-level **hypergraph based data model (HDM)** and a set of primitive schema transformations on HDM schemas.

[MP99b] gives the definitions of equivalent HDM representations for ER, relational and UML schemas, and discusses how inter-model transformations can be supported via this underlying common data model. Using a higher-level CDM such as an ER model or the relational model can be complicated because the original and transformed schemas may be represented in different high-level modelling languages and there may not be a simple semantic correspondence between their modelling constructs. HDM schemas contain *Nodes*, *Edges* and *Constraints* as their constructs, which can be used as the underlying representation for higher-level modelling constructs. Thus, inter-model transformations can be performed by transforming the HDM representations of higher-level modelling constructs. We term the sequence of primitive transformations defined for transforming a schema s1 to a schema s2 a *transformation pathway* from s1 to s2. That is, a transformation pathway consists of a sequence of primitive schema transformations.

In this paper we discuss how Automed's transformation pathways can be used to trace the lineage of data in a data warehouse which integrates data from several source databases. We assume that both the source database schemas and the integrated database schema are expressed in the HDM data model since, as discussed in [MP99b], higher-level schemas and the transformations between can be automatically translated into an equivalent HDM representation. We use a functional *intermediate query language* (IQL) as the query language to implement our lineage-tracing algorithm.

The remainder of this paper is as follows. Section 2 discusses related work and existing methods of tracing data lineage. Section 3 reviews the Automed framework, including the HDM data model, IQL syntax and transformation pathways. Section 4 gives our definitions of data lineage and describes the methods of tracing data lineage we have adopted in Automed. Section 5 gives our conclusions and directions of future work.

2. Related work

The data lineage problem in data warehouse environments has increasingly become a focus of database engineering.

[WS97] proposes a general framework for computing *fine-grained* data lineage using a limited amount of information about the processing steps. The notion of *weak inversion* is introduced in the paper. Based on a weak inverse function, which must be specified by the transformation definer, the paper defines and traces data lineage for each transformation step in a visualization database environment. In the Automed approach to heterogeneous database integration, transformation pathways are defined between the source and target schemas. [MP99a] discusses how both primitive and composite schema transformations are automatically reversible, thus allowing automatic translation of data and queries between schemas. In this paper, we show how the Automed transformation pathways can also be used for data lineage tracing.

[CWW00] provides some fundamental definitions relating to the data lineage problem, including tuple derivation for an operator, tuple derivation for operators and tuple derivation for a view. It also has addressed the derivation tracing problem using bag semantics and provided the concept of *derivation set* and *derivation pool* for tracing data lineage with duplicate elements. We use those ideas in our approach and define the notions of *affect-pool* and *origin-pool* in Automed.

Another fundamental concept is addressed in [BKT00, BKT01]: 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; while where-provenance refers to the actual data in the source databases from which the integrated data was extracted. The problem of why-provenance has been studied for relational databases in

[CWW00, WS97, Cui01, CW01]. We introduce the notions of *affect* and *origin* provenance, give the definitions of data lineage in Automed and discuss the lineage tracing algorithms for these the two kinds of provenance.

There are also other previous works related to data lineage tracing [BB99, FJS97, GFS+01]. Most of these consider *coarse-grained* lineage based on annotations on each data transformation step, which provides estimated lineage information not the exact tuples in the data sources. Using our approach, *fine-grained* lineage, i.e. a specific derivation in the data sources, can be computed given the source schemas, integrated schema, and transformation pathways between them. All of our algorithms are based on bag semantics using the HDM data model and the IQL query language.

3. The Automed Framework

This section gives a short review of the Automed schema transformation framework, including the HDM data model, IQL language and transformation pathways. More details of this material can be found in [PM98, MP99a, MP99b, Pou01a].

A **schema** in the **Hypergraph Data Model (HDM)** is a triple <*Nodes, Edges, Constraints*> containing a set of nodes, a set of edges, and a set of constraints. A **query** q over a schema S is an expression whose variables are members of *Nodes* and *Edges*. *Nodes* and *Edges* define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. *Constraints* is a set of boolean-valued queries over S. The nodes and edges of a schema are identified by their *scheme*. For a node this is the form *nodeName* and for an edge it is of the form «*edgeName*, $scheme_1$, $scheme_2$,..., $scheme_n$», where $scheme_1$, ..., $scheme_n$ are the schemes of the constructs connected by the edge. Edge names are optional and the absence of a name is denoted by "_".

An **instance** I of a schema S = <*Nodes, Edges, Constraints*> is a set of sets satisfying the following:

(i) each construct $c \in Nodes \cup Edges$ has an extent, denoted by $Ext_{S,I}(c)$, that can be derived from I ;
(ii) conversely, each set in I can be derived from the set of extents $\{Ext_{S,I}(c) | c \in Nodes \cup Edges\}$
(iii) for each $e \in Edges$, $Ext_{S,I}(e)$ contains only values that appear within the extents of the constructs linked by e (domain integrity);
(iv) the value of every constraint $c \in Constraints$ is true, the **value** of a query q being given by $q[c_1/Ext_{S,I}(c_1), \ldots, c_n/Ext_{S,I}(c_n)]$ where $c_1, \ldots, c_n$ are the constructs in $Nodes \cup Edges$.

The function $Ext_{S,I}$ is called an **extension mapping**. A HDM **model** is a triple <S, I, $Ext_{S,I}$>. The primitive transformations on HDM models are as follows. Each transformation is a function that when applied to a model returns a new model (note that only the schema and extension mapping are affected by these transformations, not the instance i.e. the data):

1. *renameNode*(*fromName,toName*) renames a node.
2. *renameEdge*(«*fromName,*c_1,…,c_n», *toName*) renames an edge.
3. *addConstraint c* adds a new constraint *c*.
4. *delConstraint c* deletes a constraint.
5. *addNode*(*name, q*) adds a node named *name* whose extent is given by the value of the query q over the existing schema constructs.
6. *delNode*(*name, q*) deletes a node. Here, q is a query that states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs (thus, not violating property (ii) of an instance).
7. *addEdge*(«*name,* c_1, … , c_n», *q*) adds a new edge between a sequence of existing schema constructs c_1, ..., c_n. The extent of the edge is given by the value of the query q over the existing schema constructs.
8. *delEdge*(«*name,* c_1, … , c_n», *q*) deletes an edge. q states how the extent of the deleted edge could be recovered from the extents of the remaining schema constructs.

A **composite transformation** is a sequence of n ≥ 1 primitive transformations. We term the composite transformation defined for transforming schema s1 to schema s2 a *transformation pathway* from s1 to s2.

The query, q, in each transformation is expressed in a functional *intermediate query language,* IQL [Pou01a]. This supports a number of primitive types, such as booleans, strings and numbers, as well as product, function and bag types. The set of *simple* IQL queries are as follows, where D, D_1 ..., D_r denote a bag of the appropriate type:

1. $q = D_1 ++ D_2 ++ \ldots ++ D_r$ /* bag union*/
2. $q = D_1 -- D_2$ /* bag *monus* [Alb91, GL99] */
3. q = group D /* group a bag of pairs on their first component*/
4. q = sort D
5. q = sortDistinct D /*sort and remove duplicates*/
6. q = aggFun D (aggFun = "max" | "min" | "count" | "sum" | "avg") /*apply an aggregation function*/
7. q = gc aggFun D (aggFun = "max" | "min" | "count" | "sum" | "avg") /*group a bag of pairs on their first component and apply an aggregation function to the second component*/
8. $q = [p | p \leftarrow D_1;$ member D_2 $p]$ /*members of D_1 that are members of D_2*/
9. $q = [p | p \leftarrow D_1;$ not (member D_2 $p)]$ /*members of D_1 that are not members of D_2*/
10. $q = [p \mid p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$ /* the c_i are filters */

General IQL queries are formed by arbitrary nesting of the above simple query constructs.

The constructs in 8,9,10 above are *comprehensions* [Tri91]. These have the general syntax $[e|Q_1; \ldots; Q_n]$, where Q_1 to Q_n are qualifiers, each qualifier being either a filter or a generator. A filter is a boolean-valued expression. A generator has syntax "$p \leftarrow q$" where p is a pattern and q is a collection-valued expression. A pattern is either a variable or a tuple of patterns. In IQL, the head expression e of a comprehension is also constrained to be a pattern.

IQL can represent common database query operations, such as select-project-join (SPJ) operations and SPJ operations with aggregation (ASPJ). For example, to get the maximum daily sales total for each store in the relation StoreSales (store_id, daily_total, date), in SQL we use:

SELECT store_id, max(daily_total)
FROM StoreSales
GROUP BY store_id

In IQL this query is expressed by

V = gc max [(s, t)| (s, t, d) ← StoreSales]

Example 1: Transforming between HDM schemas

Consider two HDM schemas $S_1 = (N_1, E_1, C_1)$ and $S_2 = (N_2, E_2, C_2)$ where

N_1 = {*mathematician, compScientist, salary*},
C_1 = {},
E_1 = {«_, *mathematician, salary*»,
«_, *compScientist, salary*»};
N_2 = {*dept, person, salary, avgDeptSalary*},
C_2 = {},
E_2 = {«_, *dept, person*», «_, *person, salary*»,
«_, *dept, avgDeptSalary*»}.

Figure 1 illustrates these schemas S_1 and S_2.

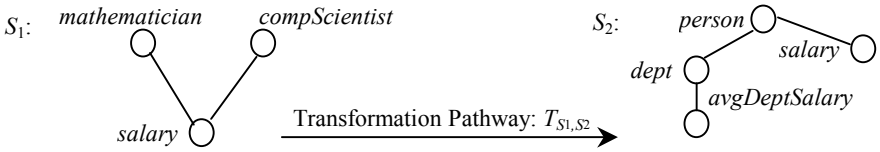

Figure 1: Transforming schema S_1 to S_2

S_1 can be transformed to S_2 by the following sequence of primitive schema transformations:

$T_{S1,S2}$ =
addNode (*dept*,{"*Maths*","*CompSci*"});
addNode (*person,* [*x*| *x* ← *mathematician*] ++
[*x*| *x* ← *compScientist*]);
addNode (*avgDeptSalary,*
{avg [*s*| (*m,s*)←«_, *mathematician, salary*»]} ++
{avg [*s*| (*c,s*)←«_, *compScientist, salary*»]});
addEdge («_, *dept, person*»,
[("*Maths*", *x*)| *x* ← *mathematician*] ++
[("*CompSci*", *x*) | *x* ← *compScientist*]);
addEdge («_, *person, salary*»,
«_, *mathematician,salary*» ++
«_, *compScientist*, *salary*»);
addEdge («_, *dept, avgDeptSalary*»,
{("*Maths"*,
avg [*s*| (*m,s)*← «_, *mathematician, salary*»]),
("*CompSci*",
avg [*s*| (*c,s)*←«_, *compScientist, salary*»])});
delEdge («_, *mathematician, salary*»,
[(*p, s*)| (*d, p*) ← «_, *dept, person*»;
(*p', s*) ← «_, *person, salary*»;
d = "*Maths*"; *p* = *p'*]);
delEdge («_, *compScientist, salary*»,
[(*p, s*)| (*d, p*) ← «_, *dept, person*»;
(*p', s*)← «_, *person, salary*»;
d = "*CompSci*"; *p* = *p'*});
delNode (*mathematician,* [*p*| (*d, p*)←
«_, *dept, person*»; *d* = "*Maths*"]);
delNode («*compScientist*», [*p*| (*d, p*) ←
«_, *dept, person*»; *d* = "*CompSci*"]); □

The first 6 transformation steps in $T_{S1,S2}$, create the constructs in S_2 which do not exist in S_1. The query in each step gives the extension of the new schema construct in terms of the existing schema constructs. The last 4 transformation steps then delete the redundant constructs of S_1. The extension of each deleted construct can be reconstructed by the query in the transformation step.

4. Tracing data lineage in Automed

What we investigate in this paper is how the lineage of data items in an integrated database can be computed given the source databases and the transformation pathways between the source schemas and the integrated schema. In this section we present our definitions of data lineage and describe our lineage tracing methods.

4.1 Data lineage in Automed

Regarding the definitions of data lineage, the fundamental ones are given in [CWW00], including tuple derivation for an operator, tuple derivation for a view, and methods of derivation tracing with both *set* and *bag* semantics. However, these definitions and methods are limited to *why-provenance* [BKT01] and what they consider is a class of views defined over base relations using the relational algebra operators: *selection* (σ), *projection* (π), *join* (⋈), *aggregation* (α), *set union* (∪), and *set difference* (–). The query language used in Automed is IQL based on *bag* semantics allowing duplicate elements in a source schema or the integrated schema, and also within the collections that are derived during lineage tracing. Also, we consider both *affect-provenance* and *origin-provenance* in our treatment of the data lineage problem.

What we regard as affect-provenance includes all of the source data that had some influence on the result data. Origin-provenance is simpler because here we are only interested in the specific data in the source databases from which the resulting data is extracted.

4.1.1 Data lineage with set semantics in IQL

The definition of *tuple derivation for an operation* was given in [CWW00] considering only the aspect of affect-provenance. We use the notions of *maximal witness* and *minimal witness* from [BKT01] to classify data lineage into two aspects: *affect-set* and *origin-set*. For set semantics and simple IQL queries, the definitions of affect-set and origin-set for a tuple and a tuple set[1] in the integrated database are given as follows. The q in these definitions is any IQL simple query.

Definition 1 (Affect-set for a simple query in IQL) Let q be any simple query over sets $T_1, \ldots, T_m$, and let $V = q(T_1, \ldots, T_m)$ be the set that results from applying q to $T_1, \ldots, T_m$. Given a tuple $t \in V$, we define t's *affect-set in* $T_1, \ldots, T_m$ *according to* q to be $q^{A}_{<T1, \ldots, Tm>}(t) = <T_1^*, \ldots, T_m^*>$, where $T_1^*, \ldots, T_m^*$ are **maximal** subsets of $T_1, \ldots, T_m$ such that:

(a) $q(T_1^*, \ldots, T_m^*) = \{t\}$

[1] By *tuple set* we mean a set of tuples, and by *tuple bag* we mean a bag of tuples.

(b) $\forall T_i'$: $q(T_1^*, \ldots, T_i', \ldots, T_m^*) = \{t\} \Rightarrow T_i' \subseteq T_i^*$

(c) $\forall T_i^*$: $\forall t^* \in T_i^*$: $q(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \emptyset$

Also, we say that $q^A_{Ti}(t) = T_i^*$ is *t's affect-set in* T_i. The affect-set of a tuple set $T \subseteq V$ contains all tuples in the affect-set of any tuple in T, denoted as $q^A_{<T1, \ldots, Tm>}(T)$. □

Definition 2 (Origin-set for a simple query in IQL) Let q, T_1, …, T_m, V and t be as above. We define t's *origin-set in* T_1, …, T_m *according to* q to be $q^O_{<T1, \ldots, Tm>}(t) = <T_1^*, \ldots, T_m^*>$, where T_1^*, …, T_m^* are **minimal** subsets of T_1, …, T_m such that:

(a) $q(T_1^*, \ldots, T_m^*) = \{t\}$

(b) $\forall T_i'$: $T_i' \subset T_i^*$: $q(T_1^*, \ldots, T_i', \ldots, T_m^*) \neq \{t\}$

(c) $\forall T_i^*$: $\forall t^* \in T_i^*$: $q(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \emptyset$

Also, we say that $q^O_{Ti}(t) = T_i^*$ is *t's origin-set in* T_i, and $q^O_{<T1, \ldots, Tm>}(T)$ is the origin-set of a tuple set $T \subseteq V$. □

In those two definitions, condition (a) states that the result of applying query q to the lineage must be the tracing tuple t; condition (b) is used to enforce the maximizing and minimizing properties respectively; and condition (c) removes the redundant elements in the computed derivation of tuple t (see [CWW00]).

Proposition 1: The origin-set of a tuple set T is a subset of the affect-set of T. □

4.1.2 Data lineage with bag semantics in IQL

As mentioned as above, our approach for tracing data lineage is based on bag semantics which allow duplicate elements to exist in the source schemas, the integrated schema and computed lineage collections. We use the notions of *affect-pool* and *origin-pool* to describe the data lineage problem with bag semantics:

Definition 3 (Affect-pool for a simple query in IQL) Let q be any simple query over bags T_1, …, T_m, and let $V = q(T_1, \ldots, T_m)$ be the bag that results from applying q to T_1, …, T_m. Given a tuple $t \in V$, we define t's *affect-pool in* T_1, …, T_m *according to* q to be $q^{AP}_{<T1, \ldots, Tm>}(t) = <T_1^*, \ldots, T_m^*>$, where T_1^*, …, T_m^* are **maximal** sub-bags of T_1, …, T_m such that:

(a) $q(T_1^*, \ldots, T_m^*) = \{x | x \leftarrow T; x = t\}$

(b) $\forall T_i'$: $q(T_1^*, \ldots, T_i', \ldots, T_m^*) = \{x | x \leftarrow T; x = t\} \Rightarrow T_i' \subseteq T_i^*$

(c) $\forall T_i^*$: $\forall t^* \in T_i^*$: $q(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \emptyset$

Also, we say that $q^{AP}_{Ti}(t) = T_i^*$ is *t's affect-pool in* T_i. The affect-pool of a tuple bag $T \subseteq V$ contains all tuples in the affect-pool of any tuple in T, denoted as $q^{AP}_{<T1, \ldots, Tm>}(T)$. □

Definition 4 (Origin-pool for a simple query in IQL) Let q, T_1, …, T_m, V and q be as above. We define t's *origin-pool in* T_1, …, T_m *according to* q to be $q^{OP}_{<T1, \ldots, Tm>}(t) = <T_1^*, \ldots, T_m^*>$, where T_1^*, …, T_m^* are **minimal** sub-bags of T_1, …, T_m such that:

(a) $q(T_1^*, \ldots, T_m^*) = \{x | x \leftarrow T; x = t\}$

(b) $\forall T_i^*$: $\neg\exists t^*$: $t^* \in T_i^*$, $t^* \in (T_i \text{ -- } T_i^*)$

(c) $\forall T_i^*$: $\forall t^* \in T_i^*$: $q(T_1^*, \ldots, \{x | x \leftarrow T_i^*; x \neq t^*\}, \ldots, T_m^*) \neq \{x | x \leftarrow T; x = t\}$

(d) $\forall T_i^*$: $\forall t^* \in T_i^*$: $q(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \emptyset$

Also, we say that $q^{OP}_{Ti}(t) = T_i^*$ is *t's origin-pool in* T_i, and $q^{OP}_{<T1, \ldots, Tm>}(T)$ is the origin-pool of a tuple bag $T \subseteq V$. □

Note that the condition (b) in Definition 4 ensures that if the origin-pool of a tuple t is T_i^* in the source bag T_i, then for any tuple in T_i, either all of the copies of the tuple are in T_i^* or none of them are in T_i^*.

From above definitions and the definition of simple IQL queries in Section 3, we now specify the affect-pool and origin-pool for IQL simple queries. As in [CWW00], we use *derivation tracing queries* to evaluate the lineage of a tuple t. That is, we apply a query to the source data repository D and the obtained result is the derivation of t in D. We call such a query the *tracing query for t on D*, denoted as $TQ_D(t)$.

Theorem 1 (*Affect-* and *Origin-pool* for a tuple with IQL simple queries):

Let $V = q(D)$ be the bag that results from applying a simple IQL query q to a source data repository D, consisting of one or more bags. Then, for any tuple $t \in V$, the tracing queries $TQ^{AP}{}_D(t)$ below give the affect-pool of t in D, and the tracing queries $TQ^{OP}{}_D(t)$ give the origin-pool of t in D:

1. $q = D_1 \text{ ++ } \ldots \text{ ++ } D_r$ $\quad (D = <D_1, \ldots, D_r>)$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D_1; x = t], \ldots, [x | x \leftarrow D_r; x = t]>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D_1; x = t], \ldots, [x | x \leftarrow D_r; x = t]>$

2. $q = D_1 \text{ -- } D_2$ $\quad (D = <D_1, D_2>)$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D_1; x = t], D_2>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D_1; x = t], [x | x \leftarrow D_2; x = t]>$

3. $q = \text{group } D$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D;\ \text{first } x = \text{first } t]>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D;\ \text{first } x = \text{first } t]>$

4. $q = \text{sort } D$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D; x = t]>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D; x = t]>$

5. $q = \text{sortDistinct } D$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D; x = t]>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D; x = t]>$

6. $q = \text{aggFun } D$ $\quad$ (aggFun = "max" | "min" | "count" | "sum" | "avg")

$TQ^{AP}{}_D(t) = <D>$

$$TQ^{OP}{}_D(t) = \begin{cases} <[x | x \leftarrow D; x = t]> & (aggFun = \text{"max"} | \text{"min"}) \\ <D> & (aggFun = \text{"count"} | \text{"sum"} | \text{"avg"}) \end{cases}$$

7. $q = \text{gc aggFun } D$ $\quad$ (aggFun = "max" | "min" | "count" | "sum" | "avg")

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D;\ \text{first } x = \text{first } t]>$

$$TQ^{OP}{}_D(t) = \begin{cases} <[x | x \leftarrow D; x = t]> & (aggFun = \text{"max"} | \text{"min"}) \\ <[x | x \leftarrow D;\ \text{first } x = \text{first } t]> & (aggFun = \text{"count"} | \text{"sum"} | \text{"avg"}) \end{cases}$$

8. $q = [x | x \leftarrow D_1; \text{member } D_2\ x]$ $\quad (D = <D_1, D_2>)$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D_1; x = t], [x | x \leftarrow D_2; x = t]>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D_1; x = t], [x | x \leftarrow D_2; x = t]>$

9. $q = [x | x \leftarrow D_1; \text{not (member } D_2\ x)]$ $\quad (D = <D_1, D_2>)$

$TQ^{AP}{}_D(t) = <[x | x \leftarrow D_1; x = t], D_2>$

$TQ^{OP}{}_D(t) = <[x | x \leftarrow D_1; x = t]>$

10. $q = [p | p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$ $\quad (D = <D_1, \ldots, D_r>)$

In the above expression, each pattern p_i is a sub-pattern of p and all tuples $t \in V$ match p. For any $t \in V$, let t_i be the tuple derived by projecting the components of p_i from t. Then:

$TQ^{AP}{}_{D}(t) = <[p_1 | p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k], \ldots, [p_r | p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k]>$

$TQ^{OP}{}_{D}(t) = <[p_1 | p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k], \ldots, [p_r | p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k]>$ □

It is simple to show that the results of queries $TQ^{AP}{}_{D}(t)$ and $TQ^{OP}{}_{D}(t)$ satisfy Definition 3 and 4 respectively. For more complex IQL queries, the above formulae can be recursively applied to the syntactic structure of an IQL query. An alternative (which we discuss in the Conclusions section) is to decompose a transformation step containing a complex IQL query into a sequence of transformation steps each containing a simple IQL query.

4.1.3 Data lineage through Automed transformation pathways

In the Automed framework, given an integrated schema *GS,* an instance of it *I*, and a construct *O* of *GS*, a tuple $t \in Ext_{GS,I}(O)$ may have multiple derivations in the source databases. Some derivations are the "actual" source data that *t* was extracted from i.e. the origin-pool, while some derivations just had an influence on the existence of *t* i.e. the affect-pool.

For simplicity of exposition, henceforth we assume that all of the source schemas have first been integrated into a single schema *S* consisting of the union of the constructs of the individual source schemas (with appropriate renaming of schema constructs to avoid duplicate names).

Suppose an integrated schema *GS* has been derived from this source schema *S* though an Automed transformation pathway $TP = tp_1, \ldots, tp_r$. Treating each transformation step as a function applied to *S*, *GS* can be obtained as $GS = tp_1 \circ tp_2 \circ \ldots \circ tp_r(S) = tp_r(tp_{r-1}(\ldots(tp_1(S))\ldots))$. Thus, tracing the lineage of data in *GS* requires tracing data lineage via a query-sequence, defined as follows:

Definition 5 (Affect-pool for a query-sequence) Let $Q = q_1, q_2, \ldots, q_r$ be a query sequence over bags *D*, and let $V = Q(D) = q_1 \circ q_2 \circ \ldots \circ q_r(D)$ be the set of bags that results from applying *Q* to *D*. Given a tuple *t* contained in some bag $B \in V$, we define *t*'s *affect-pool in D according to Q* to be $Q^{AP}{}_{D}(t) = D^*$, where $D_i^* = q_i^{AP}(D_{i+1}^*)$ $(1 \le i \le r)$, $D_{i+1}^* = \{t\}$ and $D^* = D_1^*$. The affect-pool of a tuple bag $T \subseteq V$ according to *Q* contains all tuples in the affect-pool according to *Q* of any tuple in *T*, denoted as $Q^{AP}{}_{D}(T)$. □

Definition 6 (Origin-pool for query-sequence) Let *Q*, *D*, *V* and *t* be as above. We define *t*'s *origin-pool in D according to Q* to be $Q^{OP}{}_{D}(t) = D^*$, where $D_i^* = q_i^{OP}(D_{i+1}^*)$ $(1 \le i \le r)$, $D_{i+1}^* = \{t\}$ and $D^* = D_1^*$. The origin-pool of a tuple bag $T \subseteq V$ according to *Q* contains all tuples in the origin-pool according to *Q* of any tuple in *T*, denoted as $Q^{OP}{}_{D}(T)$. □

Definitions 5 and 6 show that the derivations of data in an integrated schema can be derived though the reverse transformation pathways, step by step.

An Automed transformation pathway is a composite transformation that consists of a sequence of primitive transformations, which generate the integrated schema from the given source schemas. The constructs of an HDM schema are *Nodes*, *Edges*, and *Constraints.* When considering data lineage tracing, we treat *Nodes* and *Edges* similarly since both of these kinds of constructs have an extension i.e. contain data. We ignore the *Constraints* part of a schema because a constraint is just a query over the nodes and edges of a schema and does not contain any data.

Thus, for data lineage tracing, we integrate the primitive transformations *addNode* and *addEdge* into a single *addConstruct* tranformation, we integrate *delNode* and *delEdge* into *delConstruct*, we integrate *renameNode* and *renameEdge* into *renameConstruct*, and we ignore *addConstraint* and *delConstraint* transformations in a transformation pathway.

Other ongoing work within the Automed project is investigating simplification techniques for transformation pathways, such as removing matching pairs of add and delete steps for the same construct, and combining pairs of add and rename steps into a single add step [Tong02]. As a result of such simplifications, we assume here that the following pre- and post-conditions hold for each step in an Automed transformation pathway:

(**Pre- and Post-conditions for transformation pathways)** Suppose source schema *S* was transformed to integrated schema *GS* via a transformation pathway, $TP = tp_1, \ldots, tp_r$. Then,

1. The pre- and post-conditions for $tp_i = addConstruct(O, q)$ $(1 \le i \le r)$ are as follows:
   (i) *O* must not exist in *S* and not be created in the transformation pathway $TP' = tp_1, \ldots, tp_{i-1}$;
   (ii) The constructs appearing in *q* must already exist in *S* or have been created by the transformation pathway $TP' = tp_1, \ldots, tp_{i-1}$;
   (iii) *O* must exist in *GS* after the transformation pathway has been applied.
2. The pre- and post-conditions for $tp_i = delConstruct(O, q)$ $(1 \le i \le r)$ are as follows:
   (i) *O* must already exist in *S*;
   (ii) The constructs appearing in *q* must already exist in *S* or have been created by the transformation pathway $TP' = tp_1, \ldots, tp_{i-1}$;.
   (iii) *O* must not exist in *GS* after the transformation pathway has been applied.
3. The pre- and post-conditions for $tp_i = renameConstruct(P, O)$ $(1 \le i \le r)$ are as follows:
   (i) *P* must exist in *S*;
   (ii) *P* must not exist in *GS* after the transformation pathway has been applied;
   (iii) *O* must not exist in *S* and not be created in the transformation pathway $TP' = tp_1, \ldots, tp_{i-1}$;
   (iv) *O* must exist in *GS* after the transformation pathway has been applied.

With these pre- and post-conditions, all the constructs appearing in *GS* must have been created in one of three ways: (a) created by an *addConstruct* transformation; (b) created by a *renameConstruct* transformation; and (c) constructs existing in the source schema *S* and remaining in the integrated schema *GS*. Thus, the problem of data lineage, falls into three cases:

(a) If a construct O was created by an *addConstruct*(O, q) transformation, then the lineage of data in O is located in the constructs that appear in q.
(b) If a construct O was created by a *renameConstruct*(P, O) transformation, then the lineage of data in O is located in the source construct P.
(c) If a construct O exists in the source schema and remains in the integrated schema, the lineage of data in the integrated construct O is located in the source construct O.

4.2 Algorithm for tracing derivations through Automed transformation pathways

It is simple to trace data lineage in cases (b) and (c) discussed above. Procedure *traceRename*(t, O) shown in Figure 2 can be used to trace the lineage of a tuple t in the schema construct O created by *renameConstruct*(P, O) transformation (case (b) above). Procedure *traceRemaining*(t, O) shown in Figure 3 can be used for

```
procedure traceRename(t, O)
// O is the construct containing tuple t;
  D ← Ext_S,I(O.relateTP.sourceConstruct);
  D* ← [x| x ← D; x = t];
return (D*);
```

Figure 2: Tracing for *renameConstruct*

the remaining schema constructs (case (c)). In these two cases, all of data in the construct O is extracted from the source schema, so the affect-pool is equal to the origin-pool.

We assume that each schema construct, O, has an

```
procedure traceRemaining(t, O)
// O is the construct containing tuple t;
// O.relateTP = Ø;
  D* ← [x| x ← Ext_S,I(O); x = t];
return (D*);
```

Figure 3: Tracing for remaining constructs

attribute, *relateTP,* that refers to the transformation step that created O. If O is remaining from the source schema, then $O.relateTP = \emptyset$. Furthermore, each transformation step tp has four attributes:*transfType* which is "*add"* or "*rename"* (we ignore the "*delConstruct*" operator because no construct in the integrated schema can be created by this operator); *query* which is the query used in this transformation step; *sourceConstruct* which includes all constructs appearing in the *query*; and *resultConstruct* which is the construct created by this

```
procedure affectPoolOfTuple(t, O)
input:    a tracing tuple t; the construct O which
          contains tuple t.
output:   t's affect pool
begin
    case (O.relateTP = Ø) do
        D* ← traceRemaining(t,O);
    case (O.transfType = "rename") do
        D* ← traceRename(t, O);
    case (O.transfType = "add") do {
        D ← {Ext_S,I(o)| o ←
                O.relateTP.sourceConstructt)};
        D* ← TQ^AP_D(t); }   // from Theorem 1
    return (D*);
end
```

Figure 4: Affect Pool Tracing Procedure for a tuple

```
procedure originPoolOfTuple(t, O)
input:    a tracing tuple t; the construct O which
          contains tuple t.
output:   t's origin pool
begin
    case (O.relateTP = Ø) do
        D* ← traceRemaining(t,O);
    case (O.transfType = "rename") do
        D* ← traceRename(t, O);
    case (O.transfType = "add") do {
        D ← [Ext_S,I(o)| o ←
                O.relateTP.sourceConstruct)];
        D* ← TQ^OP_D(t); }  // from Theorem 1
    return (D*);
end
```

Figure 5: Origin Pool Tracing Procedure for a tuple transformation step.

As to case (a), in which the construct O was created by a transformation step *addConstruct*(O, q), the key point is how to trace the lineage using the IQL query, q. We can use the formulae given in Theorem 1 to obtain the lineage of the data created in this case. The procedures *affectPoolOfTuple*(t, O) and *originPoolOfTuple*(t, O) shown in Figures 4 and 5 can be applied to trace the affect pool and origin pool of a tuple in this case, where t is the tracing tuple in the schema construct O. The result of these procedures, D*, is a bag which contains t's derivation in the source schema. Note that for any tuple in the source database, either all of the copies of the tuple are in D* or none of them are.

The procedures *affectPoolOfSet*(T, O) and *originPoolOfSet*(T, O) in Figure 6 can then be used to compute the derivations of a tuple set, T. (Because duplicate tuples have an identical derivation, we eliminate duplicate items and convert T into a set first.) In these two procedures, we trace the data lineage of each

```
procedure affectPoolOfSet(T, O)
input:    a tracing tuple set T = {t_1, …, t_n}, the
          construct O which contains tuple set T.
output:   T's affect pool
begin
          D* ← Ø;
          for i ← 1 to n do
            D* ← D* ++
                  [x|x ← affectPoolOfTuple(t_i, O);
                              not (member D* x)];
          return (D*);
end
```

```
procedure originPoolOfSet(T, O)
input:    a tracing tuple set T = {t_1, …, t_n}, the
          construct O which contains tuple set T.
output:   T's origin pool
begin
          D* ← Ø;
          for i ← 1 to n do
            D* ← D* ++
                  [x|x ← originPoolOfTuple(t_i, O);
                              not (member D* x)];
          return (D*);
end
```

Figure 6: Derivation Tracing Procedures for a set of tuples

tuple $t_i \in T$ in turn and incrementally add each time the result into D^*. Because a tuple t^* can be the lineage of both t_i and t_j ($i \neq j$), if t^* and all of its copies in the source database have already been added to D^* as the lineage of t_i, we then do not add them again into D^* as the lineage of t_j (we use the test, not (member D^* x), to avoid such repetitions).

Finally, Figure 7 gives our recursive derivation tracing

```
procedure traceAffectPool(TL, OL)
input:   a list of tuple sets TL = T1, …, Tn; the list of
         corresponding constructs OL = O1,…, On in
         the integrated schema;
output:  T's affect pool in the source schema
begin
    D* ← Ø;
    for i = 1 to n do {
      temp ← affectPoolofSet(Ti, Oi);
      if (Ti.relateTP.transfType = "add")
        temp ← traceAffectPool(temp,
                  Ti.relateTP.sourceConstruct);
      D* ← D* ++
            [x| x ← temp; not (member D* x)];
    }
    return (D*);
end
```

```
procedure traceOriginPool(TL, OL)
input:   a list of tuple sets TL = T1, …, Tn; the list of
         corresponding constructs  OL = O1,…, On in
         the integrated schema;
output:  T's origin pool in the source schema
begin
    D* ← Ø;
    for i = 1 to n do {
      temp ← originPoolofSet(Ti, Oi);
      if (Ti.relateTP.transfType = "add")
        temp ← traceOriginPool(temp,
                  Ti.relateTP.sourceConstruct);
      D* ← D* ++
            [e| e ← temp; not (member D* e)];
    }
    return (D*);
end
```

Figure 7: Derivation Tracing Procedures for entire transformation pathways

algorithms, *traceAffectPool*(*TL*, *OL*) and *traceOriginPool*(*TL*, *OL*), for tracing data lineage using entire transformation pathways. Given a integrated schema *GS*, the source schema *S*, and a transformation pathway $TP = tp_1, \ldots, tp_r$ from *S* to *GS*. $TL = T_1,\ldots, T_n$ is a list of tuple sets such that each T_i is contained in the extension of some integrated schema construct O_i. *OL* is the list of integrated schema constructs $O_1,\ldots, O_n$. We recall that each schema construct has an attribute *relateTP*, and each transformation step has attributes *operatorType*, *query*, *sourceConstruct* and *resultConstruct*.

In procedure *traceAffectPool*(*TL, OL*) (and similarly in *traceOriginPool*(*TL, OL*)), we compute derivations for each tuple set T_i in *TL* one by one using the procedure *affectPoolofSet*(T_i, O_i). If the construct O_i which contains tuple set T_i is created by a *renameConstruct* transformation or remains from the source schema (i.e. *relateTP* is Ø), then the computed data can be directly extracted from the source schema (as a result of the pre- and post-conditions of Section 4.1.3). If O_i is created by an *addConstruct*(O_i, q) transformation, the constructs in query q may have been created by the earlier part of the transformation pathway, and the computed data needs to be extracted from these constructs. Therefore, we call procedure *traceAffectPool* recursively while the *relateTP* of the construct is "*addConstruct*".

5. Conclusions and future work

We have presented definitions for data lineage in Automed based on both why-provenance and where-provenance, which we have termed *affect-pool* and *origin-pool*, respectively. We have given formulae for tracing the affect-pool and the origin-pool for tuples and tuple sets derived from sequences of simple IQL queries. Rather than relying on a high-level common data model such as an ER or relational model, the Automed integration approach is based on a lower-level CDM – the HDM data model. Heterogeneous source schemas can be automatically translated into the equivalent HDM representation, and transformations between them expressed as transformations on their HDM representations. The contribution of the work we have discussed in this paper is that we have shown how the individual transformation steps in an Automed transformation pathway can be used to trace the derivation of data in the integrated database in a step-wise fashion, thus simplifying the lineage tracing process. The data lineage problem and the solutions presented in this paper have led to a number of areas of further work:

- *Handling more complex IQL queries appearing in transformation pathways*. We are investigating techniques for decomposing complex IQL queries appearing in single a transformation step into a sequence of transformation steps each accompanied by a single simple query, so that the formulae in Theorem 1 can be applied directly.
- *Combining our approach for tracing data lineage with the problem of incremental view maintenance.* Automed transformation pathways are automatically reversible and this feature can be exploited for both these issues. We have already done some preliminary work on using the Automed transformation pathways for incremental view maintenance. We now plan to explore the relationship between our lineage tracing and view maintenance algorithms, to determine if an integrated approach can be adopted for both.
- *Implementing our lineage tracing and view maintenance algorithms*. As a part of the Automed project, we will implement our algorithms in Java over the Automed repository and API [BT01, Auto].
- *Extending the lineage tracing and view maintenance algorithms to a more expressive transformation language*. [Pou01b] extends the Automed transformation language with parametrised procedures and iteration and conditional constructs, and we plan to extend our algorithms to this more expressive transformation language.

References

[Alb91] J. Albert. Algebraic properties of bag data types. In *VLDB'91*, pages 211-219, 1991.

[Auto] http://www.doc.ic.ac.uk/automed/resources/apidocs/index.html

[BB99] P. Bernstein and T. Bergstraesser. Meta-data support for data transformations using Microsoft repository. *IEEE Data Engineering Bulletin,* 22(1): 9-14, March 1999.

[BKT00] P. Buneman, S. Khanna and W. Tan. Data Provenance: some basic issues. In *Foundations of Software Technology and Theoretical Computer Science,* 2000.

[BKT01] P. Buneman, S. Khanna and W. Tan. Why and Where: a characterization of data provenance. In *ICDT'01*, LNCS 1973, pp. 316-330, Springer-Verlag, Berlin Heidelberg, 2001.

[BT01] M. Boyd and N. Tong. The Automed repositories and API. Technical Report. Imperial College, University of London, August 2001. http://www.doc.ic.ac.uk/automed/techreports/automed_repository.ps

[Cui01] Y. Cui. Lineage tracing in data warehouses. Ph.D. Thesis, Computer Science Department, Stanford University, 2001.

[CW01] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB'01,* Rome, Italy. September 2001.

[CWW00] Y. Cui, J. Widom and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems,* June 2000.

[FJS97] C. Faloutsos, H.V. Jagadish and N.D. Sidiropoulos. Recovering information from summary data. In *VLDB'97*. Pages 36-45, Athens, Greece, August 1997.

[GFS+01] H. Galhardas, D. Florescu, D. Shasha, E. Simon and C.A. Saita. Improving data cleaning quality using a data lineage facility. In *Proceedings of the International Workshop on Design and Management of Data Warehouses* (*DMDW'01*), Interlaken, Switzerland, June 2001.

[GL99] T. Griffin and L. Libkin, Incremental Maintenance of Views with Duplicates. In A. Gupta and I. S. Mumick, editors, *Materialized Views Techniques, Implementations, and Applications*, The MIP Press, 1999.

[Hull97] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *PODS'97,* 1997

[Jas01] E. Jasper. Query translation in heterogeneous database environment. MSc thesis, Birkbeck College, University of London, September 2001.

[MP99a] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications – a schema transformation approach. In *ER'99,* Volume 1728 of *LNCS,* pages 96 – 113. Springer-Verlag, 1999.

[MP99b] P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *CAiSE'99,* volume 1626 of *LNCS,* pages 333-348. Springer-Verlag, 1999.

[MP02] P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *CaiSE'02,* volume TBC, Springer-Verlag *LNCS*, 2002.

[PM98] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering,* 28(1): 47-71, 1998.

[Pou01a] A. Poulovassilis. The Automed Intermediate Query Language. Automed Working Document 2. June 2001. http://www.doc.ic.ac.uk/automed/techreports/query_language.ps

[Pou01b] A. Poulovassilis. An enhanced transformation language for the HDM. Automed Working Document 4. July 2001. http://www.doc.ic.ac.uk/automed/techreports/enhanced_transformation_language.ps

[SL90] A. Sheth and J. Larson. Federated database system for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys,* 22(3): 183-236, 1990

[Tong02] N. Tong. Database schema transformation optimisation techniques for the Automed system. Technical Report, Imperial College, University of London, March 2002.

[Tri91] P. Trinder. Comprehensions, a query notation for DBPLs. In *DBPL'91,* pages 55-68, 1991.

[WS97] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE'97*, pages 91-102, 1997.