# Tracing Data Lineage Using Schema Transformation Pathways

Hao Fan and 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 current problems being addressed in data warehouse research. In this paper, we propose a new approach for tracing data linage based on schema transformation pathways. We show how the individual transformation steps in a transformation pathway can be used to trace the derivation of the integrated data in a step-wise fashion. Although developed for a graph-based data model and a functional query language, our approach is not limited to these and would be useful in any data transformation/integration context.

## 1 Introduction

A *data warehouse* consists of a set of materialized views defined 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 set of source data items from which $t$ was derived is termed the *data lineage* problem [8]. Supporting lineage tracing in data warehousing environments brings several benefits and applications, including in-depth data analysis, on-line analysis mining (OLAM), scientific databases, authorization management, and materialized view schema evolution [2, 20, 6, 8, 10, 9].

**Automed** (`http://www.ic.ac.uk/Automed`) is a data 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

Common to many methods for integrating heterogeneous data sources is the requirement for logical integration [11] of the data, due to variations in the design of data models for the same universe of discourse. A common approach is to define a single integrated schema expressed using a *common data model* (CDM). In previous work within the Automed project [18, 12], a general framework has been developed to support schema transformation and integration. This framework consists of a low-level **hypergraph based data model (HDM)** and a set of primitive schema transformations defined in terms of this lower-level model.

[13] 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. The approach is extended to also encompass XML data sources in [14], and current work in Automed is also extending its scope to formatted data files and plain text files.

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. In contrast, 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 $S_1$ to a schema $S_2$ a *transformation pathway* from $S_1$ to $S_2$. That is, a transformation pathway consists of a sequence of primitive schema transformations.

[12] discusses how Automed transformation pathways are automatically reversible, thus allowing automatic translation of data and queries between schemas. In this paper we show how Automed's transformation pathways can also be used to trace the lineage of data in a data warehouse which integrates data from several sources. We assume that both the source schemas and the integrated schema are expressed in the HDM data model since, as discussed in [13], higher-level schemas and transformations between them can be automatically translated into an equivalent HDM representation. We use a functional *intermediate query language* (IQL) for expressing the semantic relationships between schema constructs in transformations.

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

[20] 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. 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.

[8] provides some fundamental definitions relating to the data lineage problem, including tuple derivation for an operator, and tuple derivation for a view. It 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. We use these ideas in our approach and define the notions of *affect-pool* and *origin-pool* in Automed.

Another fundamental concept is addressed in [4, 5], 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. The problem of why-provenance has been studied for relational databases in [8, 20, 6, 7]. Here, we introduce the notions of *affect* and *origin* provenance, give definitions for 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 [2, 9, 10]. 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 [18, 12, 13, 17].

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 just its name and for an edge it is of the form $\langle\langle edgeName, scheme_1, scheme_2, \ldots, scheme_n\rangle\rangle$, where $scheme_1, \ldots, scheme_n$ 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 $e$;
(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 being 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. not the data:

- $renameNode(fromName, toName)$ renames a node.
- $renameEdge(\langle\langle fromName, c_1, \ldots, c_n\rangle\rangle, toName)$ renames an edge.
- $addConstraint\ c$ adds a new constraint $c$.
- $delConstraint\ c$ deletes a constraint.
- $addNode(name, q)$ adds a node named name whose extent is given by the value of the query $q$ over the existing schema constructs.
- $delNode(name, q)$ 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).
- $addEdge(\langle\langle name, c_1, \ldots, c_n\rangle\rangle, q)$ adds a new edge between a sequence of existing schema constructs $c_1, \ldots, c_n$. The extent of the edge is given by the value of the query $q$ over the existing schema constructs.
- $delEdge(\langle\langle name, c_1, \ldots, c_n\rangle\rangle, q)$ 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 $n \geq 1$ primitive transformations. We term the composite transformation defined for transforming schema $S_1$ to schema $S_2$ a *transformation pathway* from $S_1$ to $S_2$.

The query, $q$, in each transformation is expressed in a functional *intermediate query language*, IQL [18]. 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, $++$ is bag union, $--$ is bag *monus* [1], $group$ groups a bag of pairs on their first component, $sortDistinct$ sorts a bag and removes duplicates, $aggFun$ is an aggregation function ($max$, $min$, $count$, $sum$, $avg$), and $gc$ groups a bag of pairs on their first component and applies an aggregation function to the second component:

$q = D_1 ++ D_2 ++ \ldots ++ D_r$
$q = D_1 -- D_2$
$q = group\ D$
$q = sort\ D$
$q = sortDistinct\ D$
$q = aggFun\ D$
$q = gc\ aggFun\ D$
$q = [p | p \leftarrow D_1;\ member\ D_2\ p]$
$q = [p | p \leftarrow D_1;\ not\ (member\ D_2\ p)]$
$q = [p | p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$

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

The last three constructs above are *comprehensions* [19]. 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 (the $c_i$ above are filters). 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

$$gc\ max\ [(s,t)\,|\,(s,t,d) \leftarrow StoreSales]$$

**Example: 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
$N_1 = \{mathematician, compScientist, salary\}$,
$C_1 = \{\}$,
$E_1 = \{\langle\langle \_, mathematician, salary\rangle\rangle,$
$\langle\langle \_, compScientist, salary\rangle\rangle\}$,
$N_2 = \{dept, person, salary, avgDeptSalary\}$,
$C_2 = \{\}$,
$E_2 = \{\langle\langle \_, dept, person\rangle\rangle, \langle\langle \_, person, salary\rangle\rangle,$
$\langle\langle \_, dept, avgDeptSalary\rangle\rangle\}$.

Figure 1 illustrates these two schemas, with $S_1$ on the left and $S_2$ on the right of the figure:

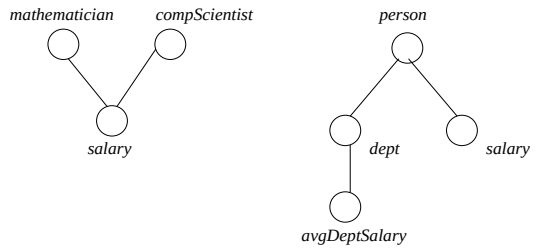


**Fig. 1.** Transforming Schema $S_1$ to Schema $S_2$

$S_1$ can be transformed to $S_2$ by the sequence of primitive schema transformations given below. The first 6 transformation steps create the constructs of $S_2$ 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 steps then delete the redundant constructs of $S_1$. The query in each of these steps shows how the extension of each deleted construct can be reconstructed from the remaining schema constructs:

*addNode* (*dept*, {*"Maths"*, *"CompSci"*});
*addNode* (*person*, $[x|x \leftarrow$ *mathematician*$] ++$
$[x|x \leftarrow$ *compScientist*$]$);
*addNode* (*avgDeptSalary*,
{*avg* $[s\,|(m,s) \leftarrow \langle\langle \_,$ *mathematician*, *salary*$\rangle\rangle]$}
$++$
{*avg* $[s|(c,s) \leftarrow \langle\langle \_,$ *compScientist*, *salary*$\rangle\rangle]$});
*addEdge* ($\langle\langle \_,$ *dept*, *person*$\rangle\rangle$,
$[($*"Maths"*$, x)|x \leftarrow$ *mathematician*$] ++$
$[($*"CompSci"*$, x)|x \leftarrow$ *compScientist*$]$);
*addEdge* ($\langle\langle \_,$ *person*, *salary*$\rangle\rangle, \langle\langle \_,$ *mathematician*, *salary*$\rangle\rangle$
$++ \langle\langle \_,$ *compScientist*, *salary*$\rangle\rangle$);
*addEdge* ($\langle\langle \_,$ *dept*, *avgDeptSalary*$\rangle\rangle$,
{*"Maths"*,
*avg* $[s|(m,s) \leftarrow \langle\langle \_,$ *mathematician*, *salary*$\rangle\rangle]$),
(*"CompSci"*,
*avg* $[s|(c,s) \leftarrow \langle\langle \_,$ *compScientist*, *salary*$\rangle\rangle]$)});
*delEdge* ($\langle\langle \_,$ *mathematician*, *salary*$\rangle\rangle$,
$[(p,s)|(d,p) \leftarrow \langle\langle \_,$ *dept*, *person*$\rangle\rangle;$
$(p',s) \leftarrow \langle\langle \_,$ *person*, *salary*$\rangle\rangle;$
$d =$ *"Maths"*$; p = p']$);
*delEdge* ($\langle\langle \_,$ *compScientist*, *salary*$\rangle\rangle$,
$[(p,s)|(d,p) \leftarrow \langle\langle \_,$ *dept*, *person*$\rangle\rangle;$
$(p',s) \leftarrow \langle\langle \_,$ *person*, *salary*$\rangle\rangle;$
$d =$ *"CompSci"*$; p = p']$);
*delNode* (*mathematician*,
$[p|(d,p) \leftarrow \langle\langle \_,$ *dept*, *person*$\rangle\rangle; d =$ *"Maths"*$]$);
*delNode* (*compScientist*,
$[p|(d,p) \leftarrow \langle\langle \_,$ *dept*, *person*$\rangle\rangle; d =$ *"CompSci"*$]$);

## 4 Tracing data lineage in Automed

The fundamental definitions regarding data lineage are given in [8], 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*[5] and what they consider is a class of views defined over base relations using the relational algebra operators: *selection* ($\sigma$), *projection*($\pi$), *join* ($\bowtie$), *aggregation* ($\alpha$), *set union* ($\bigcup$), and *set difference* ($-$). The query language used in Automed is IQL based on *bag* semantics allowing duplicate elements in a data source schema or the integrated data, 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 sources from which the resulting data is extracted.

### 4.1 Data lineage with set semantics in IQL

The definition of *tuple derivation for an operation* was given in [8] considering only the aspect of affect-provenance. We use the notions of *maximal witness*and *minimal witness* from [5] 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 in the integrated database are 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 the sequence of sets $q^A_{\langle T1,\ldots,Tm\rangle}(t) = \langle T_1^*, \ldots, T_m^*\rangle$, where $T_1^*, \ldots, T_m^*$ are **maximal** subsets of $T_1, \ldots, T_m$ such that:

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

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

In the above 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 [8]).

**Proposition 1.** The origin-set of a tuple $t$ is a subset of the affect-set of $t$.

### 4.2 Data lineage with bag semantics in IQL

As mentioned 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 the 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 $T_1, \ldots, T_m$, and let $V = q(T_1, \ldots, T_m)$ be the bag that results from applying $q$ to $T_1, \ldots, T_m$. Given a tuple $t \in V$, we define $t$'s *affect-pool in* $T_1, \ldots, T_m$ *according to* $q$ to be the sequence of bags $q^{AP}_{\langle T1,\ldots,Tm\rangle}(t) = \langle T_1^*, \ldots, T_m^*\rangle$, where $T_1^*, \ldots, T_m^*$ are **maximal** sub-bags of $T_1, \ldots, T_m$ 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* $T_i$.

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

(a) $q(T_1^*, \ldots, T_m^*) = \{x | x \leftarrow T; x = t\}$
(b) $\forall T_i^*$: $\neg\exists t^*$: $t^* \in T_i^*$, $t^* \in (T_i - T_i^*)$
(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* $T_i$.

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 $T_i^*$.

**Proposition 2.** The origin-pool of a tuple $t$ is a sub-bag of the affect-pool of $t$.

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 [8], we use *derivation tracing queries* to evaluate the lineage of a tuple $t$ with respect to a sequence of bags $D$. That is, we apply a query to $t$ and the 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-pool* 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 sequence of bags $D$. 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 $D$:

$$\begin{aligned} q &= D_1 + + \ldots + + D_r \quad (D = \langle D_1, \ldots, D_r\rangle) \\ TQ^{AP}_D(t) &= TQ^{OP}_D(t) = \\ &\quad \langle [x | x \leftarrow D_1; x = t], \ldots, [x | x \leftarrow D_r; x = t]\rangle \end{aligned}$$

$$\begin{aligned} q &= D_1 - - D_2 \quad (D = \langle D_1, D_2\rangle) \\ TQ^{AP}_D(t) &= \langle [x | x \leftarrow D_1; x = t], D_2\rangle \\ TQ^{OP}_D(t) &= \langle [x | x \leftarrow D_1; x = t], [x | x \leftarrow D_2; x = t]\rangle \end{aligned}$$

$$\begin{aligned} q &= group\ D \\ TQ^{AP}_D(t) &= TQ^{OP}_D(t) = \\ &\quad [x | x \leftarrow D; first\ x = first\ t] \end{aligned}$$

$$\begin{aligned} q &= sort\ D\ /\ sortDistinct\ D \\ TQ^{AP}_D(t) &= TQ^{OP}_D(t) = \\ &\quad [x | x \leftarrow D; x = t] \end{aligned}$$

$$\begin{aligned} q &= max\ D\ /\ min\ D \\ TQ^{AP}_D(t) &= D \\ TQ^{OP}_D(t) &= [x | x \leftarrow D; x = t] \end{aligned}$$

$$\begin{aligned} q &= count\ D\ /\ sum\ D\ /\ avg\ D \\ TQ^{AP}_D(t) &= TQ^{OP}_D(t) = D \end{aligned}$$

$$
\begin{aligned}
q &= gc\ max\ D\ /\ gc\ min\ D \\
TQ_D^{AP}(t) &= [x|x \leftarrow D; first\ x = first\ t] \\
TQ_D^{OP}(t) &= [x|x \leftarrow D; x = t]
\end{aligned}
$$

$$
\begin{aligned}
q &= gc\ count\ D\ /\ gc\ sum\ D\ /\ gc\ avg\ D \\
TQ_D^{AP}(t) &= TQ_D^{PP}(t)\ = \\
&\quad [x|x \leftarrow D; first\ x = first\ t]
\end{aligned}
$$

$$
\begin{aligned}
q &= [x|x \leftarrow D_1; member\ D_2\ x] \\
&\quad (D = \langle D_1, D_2 \rangle) \\
TQ_D^{AP}(t) &= TQ_D^{OP}(t)\ = \\
&\quad \langle [x|x \leftarrow D_1; x = t], [x|x \leftarrow D_2; x = t] \rangle
\end{aligned}
$$

$$
\begin{aligned}
q &= [x|x \leftarrow D_1; not\ (member\ D_2\ x)] \\
&\quad (D = \langle D_1, D_2 \rangle) \\
TQ_D^{AP}(t) &= \langle [x|x \leftarrow D_1; x = t], D_2 \rangle \\
TQ_D^{OP}(t) &= [x|x \leftarrow D_1; x = t]
\end{aligned}
$$

$$
\begin{aligned}
q &= [p|p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k] \\
&\quad (D = \langle D_1, \ldots, D_r \rangle) \\
TQ_D^{AP}(t) &= TQ_D^{OP}(t)\ = \\
&\quad \langle [p_1|p_1 \leftarrow D_1; p_1 = t_1; \ldots; \\
&\qquad p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k], \ldots, \\
&\quad [p_r|p_1 \leftarrow D_1; p_1 = t_1; \ldots; \\
&\qquad p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k] \rangle
\end{aligned}
$$

In the last query form above, each pattern $p_i$ is a subpattern of $p$ and all tuples $t \in V$ match $p$; for any $t \in V$, $t_i$ is the tuple derived by projecting the components of $p_i$from $t$.

It is simple to show that the results of queries $TQ_D^{AP}(t)$ and $TQ_D^{OP}(t)$ satisfy Definition 3 and 4 respectively. For more complex IQL queries, the above formulae can be successively applied to the syntactic structure of an IQL query. An alternative approach would be to decompose a transformation step containing a complex IQL query into a sequence of transformation steps each containing a simple IQL query.

### 4.3 Tracing data lineage through transformation pathways

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 a transformation pathway $TP = tp_1, \ldots, tp_r$. Treating each transformation step as a function applied to *S*, *GS* can be obtained as *GS* = $tp_1 \circ tp_2 \circ \ldots \circ tp_r(S) = tp_r(\ldots(tp_2(tp_1(S)))\ldots)$. 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 a sequence of bags $D$, and let $V = Q(D) = q_1 \circ q_2 \circ \ldots \circ q_r(D)$ be the bag that results from applying $Q$ to $D$. Given a tuple $t \in V$, we define $t$'s *affect-pool in D according to Q* to be $Q_D^{AP}(t) = D^*$, where $D_i^* = q_i^{AP}(D_{i+1}^*)$ $(1 \leq i \leq r)$, $D_{i+1}^* = \{t\}$ and $D^* = D_1^*$.

**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 $Q_D^{OP}(t) = D^*$, where $D_i^* = q_i^{OP}(D_{i+1}^*)$ $(1 \leq i \leq r)$, $D_{i+1}^* = \{t\}$ and $D^* = D_1^*$.

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

An Automed transformation pathway is a composite transformation consisting 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 extent, 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 consider the primitive transformations *addNode* and *addEdge* as a single *addConstruct* transformation, *delNode* and *delEdge* as *delConstruct*, *renameNode* and *renameEdge* as *renameConstruct*, and we ignore *addConstraint* and *delConstraint* transformations.

We thus summarize the problem of data lineage for each kind transformation step as follows:

(a) For an *addConstruct*$(O, q)$ transformation, the lineage of data in the extent of schema construct $O$ is located in the extents of the schema constructs appearing in $q$.
(b) For a *renameConstruct*$(O', O)$ transformation, the lineage of data in the extent of schema construct $O$ is located in the extent of schema construct $O'$.
(c) All *delConstruct*$(O, q)$ transformations can be ignored since they create no schema constructs.

### 4.4 Algorithms for tracing data lineage

In our algorithms below, we assume that each schema construct, $O$, has two attributes: *relateTP* is the transformation step that created $O$, and *extent* is the current extent of $O$. If a schema construct remains in the global schema directly from one of the source schemas, its *relateTP* value is Ø.

In our algorithms, each transformation step *tp* has four attributes:

- *transfType*, which is "*add*", "*ren*" or "*del*";
- *query*, which is the query used in this transformation step (if any);
- *source*, which for a *renameConstruct*$(O', O)$ returns just $O'$, and for an *addConstruct*$(O, q)$ returns a sequence of all the schema constructs appearing in $q$; and
- *result* which is $O$ for both *renameConstruct*$(O', O)$ and *addConstruct*$(O, q)$.

It is simple to trace data lineage in case (b) discussed above. If *B* is a tuple bag (i.e. bag of tuples) contained in the extent of $O$, *B*'s data lineage in $O'$ is just *B* itself, and we define this to be both the affect-pool and the origin-pool of *B* in $O'$.

In case (a), where the construct $O$ was created by a transformation step *addConstruct*$(O, q)$, the key point is how to trace the lineage using the query $q$. We can use the formulae given in Theorem 1 to obtain the lineage of data created in

this case. The procedures *affectPoolOfTuple*$(t, O)$ and *originPoolOfTuple*$(t, O)$ below can be applied to trace the affect pool and origin pool of a tuple $t$ in the extent of schema construct $O$. The result of these procedures, $DL$, is a sequence of pairs

$$\langle (D_1, O_1), \ldots, (D_n, O_n) \rangle$$

in which each $D_i$ is a bag which contains $t$'s derivation within the extent of construct $O_i$. Note that in these procedures, the sequence returned by the tracing queries $TQ^{AP}$ and $TQ^{OP}$ may consist of bags from different schema constructs. For any such bag, $B$, $B.construct$ denotes the schema construct from whose extent $B$ originates.

**proc** $affectPoolOfTuple(t, O)$
*input* : a tracing tuple $t$ in the extent of construct O
*output* : $t$'s affect-pool, $DL$
*begin*
$D = [(O'.extent, O') \mid O' \leftarrow O.relateTP.source]$
$D^* = TQ_D^{AP}(t)$;
$DL = [(B, B.construct) \mid B \leftarrow D^*]$
$return(DL)$;
*end*

**proc** $originPoolOfTuple(t, O)$
*input* : a tracing tuple $t$ in the extent of construct O
*output* : $t$'s origin-pool, $DL$
*begin*
$D = [(O'.extent, O') \mid O' \leftarrow O.relateTP.source]$
$D^* = TQ_D^{OP}(t)$;
$DL = [(B, B.construct) \mid B \leftarrow D^*]$
$return(DL)$;
*end*

Two procedures *affectPoolOfSet*$(T, O)$ and *originPoolOfSet*$(T, O)$ can then be used to compute the derivations of a tuple set (i.e. set of tuples), $T$. (Because duplicate tuples have an identical derivation, we eliminate any duplicate items and convert the tracing bag to a tracing set first.) We give *affectPoolOfSet* below. *originPoolOfSet*$(T, O)$ is identical, with *originPoolOfTuple* replacing *affectPoolOfTuple*. In these two procedures, we trace the data lineage of each tuple $t \in T$ in turn and incrementally merge each time the result into $DL$:

**proc** $affectPoolOfSet(T, O)$
*input* : a tracing tuple set $T$ contained in construct $O$
*output* : $T$'s affect-pool, $DL$
*begin*
$DL = \langle\rangle$; /* *the empty sequence* */
*for each* $t \in T$ *do*
$DL = merge(DL, affectPoolOfTuple(t, O))$;
$return(DL)$;
*end*

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 a data source have already been added to $DL$ as the lineage of $t_i$, we do not add them again into $DL$ as the lineage of $t_j$. This is accomplished by the procedure $merge$ given below, where the operator $-$ removes a element from a sequence and the operator $+$ appends an element to a sequence:

**proc** $merge(DL, DL^{new})$
*input* : data lineage sequence $DL = \langle (D_1, O_1), \ldots, (D_n, O_n) \rangle$;
new data lineage sequence $DL^{new}$
*output* : merged data lineage sequence $DL$
*begin*
*for each* $(D^{new}, O^{new}) \in DL^{new}$ *do*
*if* $O^{new} = O_i$ *for some* $O_i$ *in* $DL$ *then* {
$oldData = D_i$;
$newData = oldData ++ [x \mid x \leftarrow D^{new}; \textit{not } (member\ oldData\ x)]$;
$DL = (DL - (oldData, O_i)) + (newData, O_i)$;
}
*else*
$DL = DL + (D^{new}, O^{new})$;
$return(DL)$;
*end*

Finally, we give below our algorithm *traceAffectPool*(*B*, *O*) for tracing affect lineage using entire transformation pathways given a integrated schema *GS*, the source schema $S$, and a transformation pathway $tp_1, \ldots, tp_r$ from $S$ to *GS*. Here, *B* is a tuple bag contained in the extent of schema construct $O \in GS$. We recall that each schema construct has attributes *relateTP* and *extent*, and that each transformation step has attributes *transfType*, *query*, *source* and *result*.

We examine each transformation step from $tp_r$ down to $tp_1$. If it is a delete step, we ignore it. Otherwise we determine if the $result$ of this step is contained in the current $DL$. If so, we then trace the data lineage of the current data of $O$ in $DL$, merge the result into $DL$, and delete $O$ from $DL$. At the end of this processing the resulting $DL$ is the lineage of $T$ in the data sources:

**proc** $traceAffectPool(B, O)$
*input* : tracing tuple bag $B$ contained in construct $O$
transformation pathway $tp_1, \ldots, tp_r$
*output* : $B$'s affect-pool, $DL$
*begin*
$DL = \langle (B, O) \rangle$;
*for* $j = r$ *downto* 1 *do*
*case* $tp_j.transfType =$ *"del"*
*continue*;
*case* $tp_j.transfType =$ *"ren"*
*if* $tp_j.result = O_i$ *for some* $O_i$ *in* $DL$ *then*
$DL = (DL - (D_i, O_i)) + (D_i, tp_j.source)$;
*case* $t_j.transfType =$ *"add"*
*if* $tp_j.result = O_i$ *for some* $O_i$ *in* $DL$ *then* {
$DL = DL - (D_i, O_i)$;
$D_i = sortDistinct\ D_i$;
$DL = merge(DL, affectPoolOfSet(D_i, O_i))$;
}
*end*
$return(DL)$;
*end*

Procedure $traceOriginPool$ is identical, obtained by replacing *affectPoolOfSet* by *originPoolOfSet*.

## 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. 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 described in this paper is that we have shown how the individual steps of Automed schema transformation pathways can be used to trace the affect-pool and origin-pool of items of integrated data in a step-wise fashion.

Fundamental to our lineage tracing method is the fact that $add$ and $del$ schema transformations carry a *query* which defines the new or deleted schema construct in terms of the other schema constructs. Thus, our general approach is not limited to the HDM data model and IQL query language and can be applied to schema transformations defined on other data models using different query languages, or indeed to inter-model schema transformation pathways (as used in [14, 15] for example).

The data lineage problem and the solutions presented in this paper have led to a number of areas of further work:

- *Combining our approach for tracing data lineage with the problem of incremental view maintenance.* 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 are implementing these algorithms over the Automed repository and API [3].
- *Extending the lineage tracing and view maintenance algorithms to a more expressive transformation language*. [16] 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


1. J. Albert. Algebraic properties of bag data types. In *Proc. VLDB'91*, pages 211–219, 1991.
2. P. 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 and N. Tong. The Automed repositories and API. Technical report, Automed Project, 2001.
4. P. Buneman, S. Khanna, and W.C. Tan. Data provenance: some basic issues. In *Proc. FSTTCS 2000*, pages 87–93, 2000.
5. P. Buneman, S. Khanna, and W.C. Tan. Why and Where: a characterization of data provenance. In *Proc. ICDT 2001*, pages 316–33, 2001.
6. Y. Cui. Lineage tracing in data warehouses. phd thesis. Technical report, Computer Science Department, Stanford University, 2001.
7. Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proc. VLDB'01*, pages 471–480, 2001.
8. Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM 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, 1997.
10. 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.
11. R. Hull. Managing semantic heterogeneity in databases. In *Proc. PODS'97*, pages 51–61, 1997.
12. P. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications - a schema transformation approach. In *Proc. ER'99*, pages 96–133, 1999.
13. P. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. CAiSE'99*, pages 333–348, 1999.
14. P. McBrien and A. Poulovassilis. A semantic approch to integrating XML and structured data sources. In *Proc. CAiSE'01*, pages 330–345, 2001.
15. P. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. CAiSE'02*, 2002.
16. A. Poulovassilis. An enhanced transformation language for the HDM. Technical report, Automed Project, 2001.
17. A. Poulovassilis. The Automed Intermediate Query Language. Technical report, Automed Project, 2001.
18. A. Poulovassilis and P. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.
19. P. Trinder. Comprehensions, a query notation for DBPLs. In *Proc DBPL'91*, pages 55–68, 1991.
20. A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc ICDE'97*, pages 91–102, 1997.