Tracing Data Lineage Using Schema Transformation Pathways

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 current problems being addressed in data warehouse research. In this chapter, we propose a new approach for tracing data lineage which is 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 common data model and a functional query language, our approach is not limited to these and would be useful in any data transformation/integration framework based on sequences of primitive schema transformations.

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 [1]. 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, 3, 4, 1, 5, 6].

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 — see `http://www.ic.ac.uk/automed`

Common to many methods for integrating heterogeneous data sources is the requirement for logical integration [7] 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*. Using a high-level modelling language (e.g. ER, OO or relational) as the common data 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 previous work within the AutoMed project [8, 9, 10], a general framework has been developed to support schema transformation and integration. This framework provides a lower-level *hypergraph based data model (HDM)* as the common data model and a set of primitive schema transformations for schemas defined in this data model. One advantage of using a low-level data model such as the HDM is that semantic mismatches between modelling constructs are avoided. Another advantage is that it provides a unifying semantics for higher-level modelling constructs.

In particular, [10] shows how ER, relational and UML data models, and the set of primitive schema transformations on each of them, can be defined in terms of the lower-level HDM and its set of primitive schema transformations. That paper also discusses how inter-model transformations are possible within the AutoMed framework, thus allowing a schema expressed in one high-level modelling language to be incrementally transformed into a schema expressed in a different high-level modelling language. The approach was extended to also encompass XML data sources in [11], and ongoing work in AutoMed is also extending its scope to encompass formatted data files, plain text files, and RDF [12].

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 Each transformation step affects 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. 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. That is, a transformation pathway consists of a sequence of primitive schema transformations.

[9] discusses how AutoMed transformation pathways are *automatically reversible*, thus allowing automatic translation of data and queries between schemas. In this chapter 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. The data sources and the global schema may be expressed in the same or in different data models. AutoMed uses a functional *intermediate query language* (IQL) for defining the semantic relationships between schema constructs in each transformation step. In this chapter we show how these queries can be used to trace the lineage of data in the data warehouse.

The fundamental definitions regarding data lineage were developed in [1], including the concept of a *derivation pool* for tracing the data lineage of a tuple in a materialised view, and methods for derivation tracing with both *set* and *bag* semantics. Another fundamental concept was addressed in [13, 14], 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 [1, 3, 4, 15]. Here, we introduce the notions of *affect* and *origin* provenance in the context of AutoMed, and discuss lineage tracing algorithms for both these the two kinds of provenance. There are also other previous works related to data lineage tracing [2, 6, 5]. 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.

The remainder of this chapter is as follows. Section 2 reviews the aspects of the AutoMed framework necessary for this chapter, including the HDM data model, IQL syntax, and transformation pathways. Section 3 gives our definitions of data lineage and describes the methods of tracing data lineage that we have developed. Section 4 gives our conclusions and directions of future work.

2 The AutoMed Framework

This section gives a short review of the AutoMed framework, including the HDM data model, IQL language, and transformation pathways. More details of this material can be found in [8, 9, 10, 16].

As mentioned as above, the AutoMed framework consists of a low-level hypergraph-based data model (HDM) and a set of primitive schema transformations for schemas defined in this data model. Higher-level data models and primitive schema transformations for them are defined in terms of the lower-level HDM and its primitive schema transformations.

A *schema* in the HDM data model 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.

These definitions extend to schemas expressed in higher-level modelling languages (see for example [10] for encodings in the HDM of ER, relational and UML data models, [11] for XML and [12] for RDF). Schemas in general contain two kinds of constructs: *structural constructs* that have data extents associated with them and *constraint constructs* that map onto some set of HDM constraints. A query on a schema is defined over its structural constructs. Within AutoMed transformation pathways and queries, schema constructs are identified by a unique *scheme*, which is a tuple of labels delimited by double chevrons $\langle\langle$ and $\rangle\rangle$.

In the AutoMed integration approach, schemas are incrementally transformed by applying to them a sequence of primitive transformation steps. Each primitive transformation makes a 'delta' change to the schema, by adding, deleting or renaming just one schema construct. Each add or $delete$ step is accompanied by a query specifying the extent of the new or deleted construct in terms of the rest of constructs in the schema. Two additional kinds of primitive transformation, $extend$ and $contract$, behave in the same way as add and $delete$ except that they indicate their accompanying query may only partially construct the extent of the schema construct being added to, or deleted from, the schema. Their accompanying query may be just the distinguished constant $void$, which indicates that there is no information about how to derive the extent of the new/deleted construct from the rest of the schema constructs, even partially. We refer the reader to [17] for an extensive discussion of the AutoMed integration approach.

To illustrate our work in this chapter, we define below a simple relational data model. However, we stress that the development in this chapter is generally applicable to all modelling languages supported by the AutoMed framework.

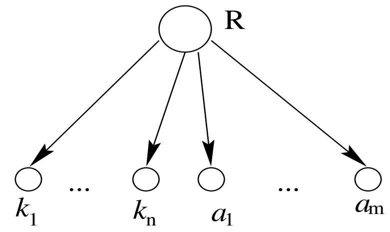

Figure 1: a simple relational data model

2.1 A simple relational data model

Schemas in our simple relational model are constructed from primary key attributes, non-primary key attributes, and the relationships between them. Figure 1 illustrates the representation of a relation R with primary key attributes $k_1, ..., k_n$ and non-primary key attributes $a_1, ..., a_m$. There is a one-one correspondence between this representation and the underlying HDM graph. In our simple relational model, there are two kinds of structural schema construct: Rel and Att (for simplicity, we ignore here the constraints present in a relational schema, e.g. key and foreign key constraints, since these are not used by our data lineage tracing methods, but see [10] for an encoding of a richer relational data model). The extent of a Rel construct $\langle\langle R\rangle\rangle$ is the projection of the relation R onto its primary key attributes $k_1, ..., k_n$. The extent of each Att construct $\langle\langle R, a\rangle\rangle$ where a is an attribute (key or non-key) is the projection of relation R onto $k_1, ..., k_n, a$.

For example, a relation $StoreSales(\underline{store\_id}, daily\_total, date)$ would be modelled by a Rel scheme $\langle\langle StoreSales\rangle\rangle$, and three Att schemes $\langle\langle StoreSales, store\_id\rangle\rangle$, $\langle\langle StoreSales, daily\_total\rangle\rangle$, $\langle\langle StoreSales, date\rangle\rangle$.

The set of primitive transformations for schemas expressed in this simple relation data model is as follows:

- addRel($\langle\langle R\rangle\rangle, q$) adds to the schema a new relation R.

  The query q specifies the set of primary key values in the extent of R in terms of the already existing schema constructs.

- addAtt($\langle\langle R, a\rangle\rangle, q$) adds to the schema an attribute a (key or non-key) for relation R.

  The query q specifies the extent of the binary relationship between the primary key attribute(s) of R and this new attribute a in terms of the already existing schema constructs.

- delRel($\langle\langle R\rangle\rangle, q$) deletes from the schema the relation R (provided all its attributes have first been deleted).

  The query q specifies how the set of primary key values in the extent of R can be restored from the remaining schema constructs.

- delAtt($\langle\langle R, a\rangle\rangle, q$) deletes from the schema attribute a of relation R.

  The query q specifies how the extent of the binary relationship between the primary key attribute(s) of R and a can be restored from the remaining schema constructs.

- renameRel($\langle\langle R\rangle\rangle, R'$) renames the relation R to R' in the schema.

- renameAtt($\langle\langle R, a\rangle\rangle, a'$) renames the attribute a of R to a'.

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, appearing in a primitive transformation is expressed in a functional *intermediate query language*, IQL [16]. For the purposes of this chapter we assume that the extents of all schema constructs are *bags*.

2.2 Simple IQL queries

In order for our data lineage tracing methods to be unambiguous, we limit the syntax of the query q that may appear within a transformation step to the set of *simple IQL* (*SIQL*) queries — we refer the reader to [14] for a discussion of ambiguity of data lineage tracing for different classes of query language. More complex IQL queries can be encoded as a series of transformations with SIQL queries on intermediate schema constructs.

The syntax of SIQL queries, q, is listed below (lines 1 to 12). D, $D_1 \ldots$, D_n denote a bag of the appropriate type (base collections). The construct in line 1 is an enumerated bag of constants. $++$ is the bag union operator and $--$ is the bag *monus* operator [18]. $group$ groups a bag of pairs on their first component. $distinct$ removes duplicates from a bag. $aggFun$ is an aggregation function (max, min, $count$, sum, avg). gc groups a bag of pairs on their first component and applies an aggregation function to the second component.

The constructs in lines 9, 10, 11 are instances of *comprehension* syntax [19, 20]. Each $\overline{x_1}$, ..., $\overline{x_n}$ is either a single variable or a tuple of variables. $\overline{x}$ is either a single variable or value, or a tuple of variables or values, and must include all of variables appearing in $\overline{x_1}$, ..., $\overline{x_n}$. Each C_1, ..., C_k is a condition not referring to any base collection. In 9, by the constraints of comprehension syntax, each variable appearing in $\overline{x}$, C_1, ..., C_k must appear in some $\overline{x_i}$. In 10 and 11, the variables in $\overline{y}$ must appear in $\overline{x}$[1]. In line 12, map applies a function f to a collection D, where f is constrained not to refer to any base collections.

1. $q = [c_1, c_2, ..., c_n]$ (where $n \geq 0$ and each $c_1, ..., c_n$ is a constant)
2. $q = D_1 ++ D_2 ++ \ldots ++ D_n$ (where $n \geq 1$)
3. $q = D_1 -- D_2$
4. $q = group\ D$
5. $q = sort\ D$
6. $q = distinct\ D$
7. $q = aggFun\ D$
8. $q = gc\ aggFun\ D$
9. $q = [\overline{x} | \overline{x_1} \leftarrow D_1; \ldots ; \overline{x_n} \leftarrow D_n; C_1; ...; C_k]$ (where $n \geq 1, k \geq 0$)
10. $q = [\overline{x} | \overline{x} \leftarrow D_1;\ member\ D_2\ \overline{y}]$
11. $q = [\overline{x} | \overline{x} \leftarrow D_1;\ not(member\ D_2\ \overline{y})]$
12. $q = map\ f\ D$

IQL can represent common database queries, such as select-project-join (SPJ) queries and SPJ queries with aggregation (ASPJ). For example, to obtain the maximum daily sales total for each store in a relation $StoreSales(\underline{store\_id}, daily\_total, date)$, in SQL we would use:

[1]General comprehensions have the syntax $[e|Q_1; \ldots ; Q_n]$. 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, as are $member\ D_2\ \overline{y}$ and $not(member\ D_2\ \overline{y})$. A generator has syntax $p \leftarrow c$ where p is a pattern and c is a collection-valued expression. In SIQL, the patterns p are constrained to be single variables or tuples thereof while the collection-valued expressions c are constrained to be base collections.

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

In IQL, assuming the simple relational model defined in Section 2.1, this query would be expressed by

$$gc\ max\ \langle\langle StoreSales, daily\_total\rangle\rangle$$

2.3 An example: transforming relational schemas

Consider two relational schemas S_1 and S_2. S_1 is a source schema containing two relations $mathematician(\underline{emp\_id}, salary)$ and $compScientist(\underline{emp\_id}, salary)$. S_2 is the target chema containing two relations $person(\underline{emp\_id}, salary, dept)$ and $department\ (\underline{deptName}, avgDeptSalary)$.

By our definition of the simple relational model in Section 2.1, S_1 has a set of Rel constructs Rel_1 and a set of Att constructs Att_1, while S_2 has a set of Rel constructs Rel_2 and a set of Att constructs Att_2, where:

$$\begin{array}{lcl}
Rel_1 & = & \{\langle\langle mathematician\rangle\rangle, \langle\langle compScientist\rangle\rangle\} \\
Att_1 & = & \{\langle\langle mathematician, emp_id\rangle\rangle, \langle\langle mathematician, salary\rangle\rangle \\
& & \langle\langle compScientist, emp_id\rangle\rangle, \langle\langle compScientist, salary\rangle\rangle\} \\
Rel_2 & = & \{\langle\langle person\rangle\rangle, \langle\langle department\rangle\rangle\} \\
Att_2 & = & \{\langle\langle person, emp_id\rangle\rangle, \langle\langle person, salary\rangle\rangle, \langle\langle person, dept\rangle\rangle \\
& & \langle\langle department, deptName\rangle\rangle, \langle\langle department, avgDeptSalary\rangle\rangle\}
\end{array}$$

Schema S_1 can be transformed to S_2 by the sequence of primitive schema transformations given below. The first seven 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 extents of the existing schema constructs. The last six 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.

$$\begin{array}{ll}
(1)\ \mathsf{addRel} & (\langle\langle person\rangle\rangle, \langle\langle mathematician\rangle\rangle ++ \langle\langle compScientist\rangle\rangle) \\
(2)\ \mathsf{addAtt} & (\langle\langle person, emp_id\rangle\rangle, \\
& \quad \langle\langle mathematician, emp_id\rangle\rangle ++ \langle\langle compScientist, emp_id\rangle\rangle) \\
(3)\ \mathsf{addAtt} & (\langle\langle person, salary\rangle\rangle, \\
& \quad \langle\langle mathematician, salary\rangle\rangle ++ \langle\langle compScientist, salary\rangle\rangle) \\
(4)\ \mathsf{addAtt} & (\langle\langle person, dept\rangle\rangle, [(x, "Maths")|x \leftarrow \langle\langle mathematician\rangle\rangle] ++ \\
& \quad [(x, "CompSci")|x \leftarrow \langle\langle compScientist\rangle\rangle]) \\
(5)\ \mathsf{addRel} & (\langle\langle department\rangle\rangle, ["Maths","CompSci"]) \\
(6)\ \mathsf{addAtt} & (\langle\langle department, deptName\rangle\rangle, [("Maths","Maths"), ("CompSci","CompSci")]) \\
(7)\ \mathsf{addAtt} & (\langle\langle department, avgDeptSalary\rangle\rangle, \\
& \quad gc\ avg\ [("Maths", s)|(x, s) \leftarrow \langle\langle mathematician, salary\rangle\rangle] \\
& \quad ++\ gc\ avg\ [("Maths", s)|(x, s) \leftarrow \langle\langle mathematician, salary\rangle\rangle]) \\
(8)\ \mathsf{delAtt} & (\langle\langle mathematician, salary\rangle\rangle, [(x, s)|(x, s) \leftarrow \langle\langle person, salary\rangle\rangle; \\
& \quad (x', d) \leftarrow \langle\langle person, dept\rangle\rangle; d = "Maths"; x = x']) \\
(9)\ \mathsf{delAtt} & (\langle\langle mathematician, emp_id\rangle\rangle, [(x, id)|(x, id) \leftarrow \langle\langle person, emp_id\rangle\rangle; \\
& \quad (x', d) \leftarrow \langle\langle person, dept\rangle\rangle; d = "Maths"; x = x']) \\
(10)\ \mathsf{delRel} & (\langle\langle mathematician\rangle\rangle, [x|(x, d) \leftarrow \langle\langle person, dept\rangle\rangle; d = "Maths"]) \\
(11)\ \mathsf{delAtt} & (\langle\langle compScientist, salary\rangle\rangle, [(x, s)|(x, s) \leftarrow \langle\langle person, salary\rangle\rangle; \\
& \quad (x', d) \leftarrow \langle\langle person, dept\rangle\rangle; d = "CompSci"; x = x'])
\end{array}$$

(12) delAtt $(\langle\langle compScientist, emp\_id\rangle\rangle, [(x, id)|(x, id) \leftarrow \langle\langle person, emp\_id\rangle\rangle;$
$(x', d) \leftarrow \langle\langle person, dept\rangle\rangle; d = \text{"}CompSci\text{"}; x = x'])$
(13) delRel $(\langle\langle compScientist\rangle\rangle, [x|(x, d) \leftarrow \langle\langle person, dept\rangle\rangle; d = \text{"}CompSci\text{"}])$

Note that we actually permit the specification of compositions of SIQL queries within primitive transformations e.g. the queries in steps (4) and (7) above are not SIQL queries but are composed of nested SIQL sub-queries. Such queries are automatically broken down by our software into a sequence of add or $delete$ transformation steps with SIQL queries within them. The decomposition procedure undertakes a depth-first search of the query tree and generates the sequence of transformations from the bottom up. For example, the following decompositions would be equivalent to steps (4) and (7) above, with $(4.1) \sim (4.5)$ replacing step (4) and $(7.1) \sim (7.9)$ replacing step (7)[2]:

(4.1) addAtt $(\langle\langle person, mathsDept\rangle\rangle, [(x, \text{"}Maths\text{"})|x \leftarrow \langle\langle mathematician\rangle\rangle])$
(4.2) addAtt $(\langle\langle person, CSDept\rangle\rangle, [(x, \text{"}CompSci\text{"})|x \leftarrow \langle\langle compScientist\rangle\rangle])$
(4.3) addAtt $(\langle\langle person, dept\rangle\rangle, \langle\langle person, mathsDept\rangle\rangle ++ \langle\langle person, CSDept\rangle\rangle)$
(4.4) delAtt $(\langle\langle person, CSDept\rangle\rangle, [(x, \text{"}CompSci\text{"})|x \leftarrow \langle\langle compScientist\rangle\rangle])$
(4.5) delAtt $(\langle\langle person, mathsDept\rangle\rangle, [(x, \text{"}Maths\text{"})|x \leftarrow \langle\langle mathematician\rangle\rangle])$

(7.1) addRel $(\langle\langle mathsSalary\rangle\rangle, map\ (\lambda(x, s).(\text{"}Maths\text{"}, s))\ \langle\langle mathematician, salary\rangle\rangle)$
(7.2) addRel $(\langle\langle avgMathsSalary\rangle\rangle, gc\ avg\ \langle\langle mathsSalary\rangle\rangle)$
(7.3) addRel $(\langle\langle compSciSalary\rangle\rangle, map\ (\lambda(x, s).(\text{"}CompSci\text{"}, s))\ \langle\langle compScientist, salary\rangle\rangle)$
(7.4) addRel $(\langle\langle avgCompSciSalary\rangle\rangle, gc\ avg\ \langle\langle compSciSalary\rangle\rangle)$
(7.5) addAtt $(\langle\langle department, avgDeptSalary\rangle\rangle,$
$\langle\langle avgMathsSalary\rangle\rangle ++ \langle\langle avgCompSciSalary\rangle\rangle)$
(7.6) delRel $(\langle\langle avgCompSciSalary\rangle\rangle, gc\ avg\ \langle\langle compSciSalary\rangle\rangle)$
(7.7) delRel $(\langle\langle compSciSalary\rangle\rangle, map\ (\lambda(x, s).(\text{"}CompSci\text{"}, s))\ \langle\langle compScientist, salary\rangle\rangle)$
(7.8) delRel $(\langle\langle avgMathsSalary\rangle\rangle, gc\ avg\ \langle\langle mathsSalary\rangle\rangle)$
(7.9) delRel $(\langle\langle mathsSalary\rangle\rangle, map\ (\lambda(x, s).(\text{"}Maths\text{"}, s))\ \langle\langle mathematician, salary\rangle\rangle)$

3 Tracing data lineage in AutoMed

3.1 Data lineage with bag semantics in SIQL

The fundamental definitions regarding data lineage were developed in [1], as were methods for derivation tracing with both *set* and *bag* semantics. However, these definitions and methods are limited to *why-provenance*[14], and also to the class of views defined over base relations using the relational algebra operators *selection* (σ), *projection*(π), *join* $(\bowtie)$, *aggregation* (α), *set union* $(\bigcup)$, and *set difference* $(-)$. The query language used in AutoMed is based on *bag* semantics, allowing duplicate elements within a source construct or integrated construct, 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. In particular, we use the notions of *maximal witness* and *minimal witness* from [14] in order to define the

[2]We have left the IQL queries in steps $(8) \sim (13)$ as is, since all $delete$ transformations are ignored in our data lineage tracing procedures (see Section 3.2 below).

notions of *affect-pool* and *origin-pool* in Definitions 1 and 2 below.

In both these definitions, condition (a) states that the result of applying query q to the lineage must be the bag of all tuples t in V.

Condition (b) is used to enforce the maximizing and minimizing properties, respectively. Thus, the affect-pool includes all elements in data sources which could generate t by applying q to them, while if any element and all of its copies in the origin-pool was deleted, then t or all of t's copies in V could not be generated by applying the query q to the lineage.

Condition (c) in both definitions removes the redundant elements in the computed derivation of tuple t (see [1]). Condition (d) in Definition 2 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^*.

Definition 1 (Affect-pool for a SIQL query) Let q be any SIQL 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) $q(T_1^*, \ldots, T_m^*) = \{x|x \leftarrow V; x = t\}$

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

(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 2 (Origin-pool for a SIQL query) 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) $q(T_1^*, \ldots, T_m^*) = \{x|x \leftarrow V; x = t\}$

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

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

(d) $\forall T_i^*$: $\neg\exists t^*$: $t^* \in T_i^*, t^* \in (T_i - T_i^*)$

Proposition 1. The origin-pool of a tuple t is a sub-bag of the affect-pool of t.

Following on from the above definitions and the definition of SIQL queries in Section 2.2, we now specify the affect-pool and origin-pool for SIQL queries. As in [1], 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 SIQL queries). Let $V = q(D)$ be the bag that results from applying a SIQL query q to a sequence of bags D. Then, for any tuple $t \in V$, the tracing queries$TQ_D^{AP}(t)$ below give the affect-pool of t in D, and the tracing queries $TQ_D^{OP}(t)$ give the origin-pool of t in D:

$$
\begin{array}{lrcl}
1. & q & = & [c_1, ..., c_n] \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = [x|x \leftarrow [c_1, ..., c_n]; x = t] \\
 & & & \text{/*An enumerated bag can be regarded as a system-defined base collection*/} \\
 & & & \text{/*for the purposes of data lineage tracing. */} \\
2. & q & = & D_1 ++ \ldots ++ D_n \quad (D = \langle D_1, \ldots, D_n \rangle) \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = \langle [x|x \leftarrow D_1; x = t], \ldots, [x|x \leftarrow D_n; x = t] \rangle \\
3. & q & = & D_1 -- D_2 \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) & = & \langle [x|x \leftarrow D_1; x = t], [x|x \leftarrow D_2; x = t] \rangle \\
4. & q & = & group\ D \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = [x|x \leftarrow D; first\ x = first\ t] \\
5. & q & = & sort\ D /\ distinct\ D \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = [x|x \leftarrow D; x = t] \\
6. & q & = & max\ D\ /\ min\ D \\
 & TQ_D^{AP}(t) & = & D \\
 & TQ_D^{OP}(t) & = & [x|x \leftarrow D; x = t] \\
7. & q & = & sum\ D \\
 & TQ_D^{AP}(t) & = & D \\
 & TQ_D^{OP}(t) & = & [x|x \leftarrow D; x \neq 0] \\
8. & q & = & count\ D\ /\ avg\ D \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = \ D \\
9. & 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] \\
10. & q & = & gc\ sum\ D \\
 & TQ_D^{AP}(t) & = & [x|x \leftarrow D; first\ x = first\ t] \\
 & TQ_D^{OP}(t) & = & [x|x \leftarrow D; first\ x = first\ t; second\ x \neq 0] \\
11. & q & = & gc\ count\ D\ /\ gc\ avg\ D \\
 & TQ_D^{AP}(t) & = & TQ_D^{PP}(t) \ = [x|x \leftarrow D; first\ x = first\ t] \\
12. & q & = & [\overline{x}|\overline{x_1} \leftarrow D_1; \ldots; \overline{x_n} \leftarrow D_n; C_1; ...; C_k] \qquad (D = \langle D_1, \ldots, D_n \rangle) \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = \\
 & & & \langle [x_1|x_1 \leftarrow D_1; x_1 = (\lambda \overline{x}.\overline{x_1})\ t], \ldots, [x_n|x_n \leftarrow D_n; x_n = (\lambda \overline{x}.\overline{x_n})\ t] \rangle \\
13. & q & = & [\overline{x}|\overline{x} \leftarrow D_1; member\ D_2\ \overline{y}] \qquad (D = \langle D_1, D_2 \rangle) \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) \ = \langle [\overline{x}|\overline{x} \leftarrow D_1; \overline{x} = t], [y|y \leftarrow D_2; y = (\lambda \overline{x}.\overline{y})\ t] \rangle \\
14. & q & = & [\overline{x}|\overline{x} \leftarrow D_1; not(member\ D_2\ \overline{y})] \qquad (D = \langle D_1, D_2 \rangle) \\
 & TQ_D^{AP}(t) & = & \langle [\overline{x}|\overline{x} \leftarrow D_1; \overline{x} = t], [y|y \leftarrow D_2; y = (\lambda \overline{x}.\overline{y})\ t] \rangle \\
 & TQ_D^{OP}(t) & = & \langle [\overline{x}|\overline{x} \leftarrow D_1; \overline{x} = t], \emptyset \rangle \\
15. & q & = & map\ f\ D \\
 & TQ_D^{AP}(t) & = & TQ_D^{OP}(t) = [\overline{x}|\overline{x} \leftarrow D; (f\ \overline{x}) = t] \\
 & & & \text{If } f \text{ is invertible, we can obtain } t\text{'s data lineage directly using } f\text{'s} \\
 & & & \text{inverse function } f^{-1} \text{ i.e. } TQ_D^{AP}(t) = TQ_D^{OP}(t) = f^{-1}(t).
\end{array}
$$

It is straightforward to show that the results of queries $TQ_D^{AP}(t)$ and $TQ_D^{OP}(t)$ satisfy Definition 1 and 2 respectively.

3.2 Tracing data lineage through transformation pathways

For simplicity of exposition, 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 3 (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 4 (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 3 and 4 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 consists of a sequence of primitive transformations which generate the integrated schema from the given source schemas. The schema constructs are generally different for different modelling languages. For example, HDM schemas have Node, Edge, and Constraint constructs, ER schemas have Entity, Attribute, Relationship and Generalisation constructs [10], while the simple relational schemas of Section 2.1 have Rel and Att constructs.

Thus, each modelling language also generally has a different set of primitive transformations. For example, for the HDM these are addNode, delNode, addEdge, delEdge etc. while for our simple relational data model of Section 2.1 they are addRel, delRel, addAtt, delAtt etc.

When considering data lineage tracing, we are only concerned with structural constructs associated with a data extent e.g. Node and Edge constructs in the HDM, and Rel and Att constructs in the simple relational data model. Thus, for data lineage tracing, we ignore primitive schema transformation steps which are adding, deleting or renaming only constraints. Moreover, we treat any primitive transformation which is adding a construct to a schema as a generic *addConstruct* transformation, any primitive transformation which is deleting a construct from a schema as a generic *delConstruct* transformation, and any primitive transformation which is renaming a schema construct as a generic *renameConstruct* transformation. We can summarize the problem of data lineage for each of these transformations 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 the query 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.

3.3 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. a 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 schema 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.

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

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

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. The procedure *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:

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

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 an element from a sequence and the operator $+$ appends an element to a sequence:

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

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:

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

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

We illustrate the use of the $traceAffectPool$ algorithm above by means of a simple example. Referring back to the example schema transformation in Section 2.3, suppose we have a tracing tuple $t = (\,''Maths'', 2500)$ in the extent of $\langle\langle department, avgDeptSalary\rangle\rangle$ in S_2. The affect-pool, DL, of this tuple is traced as follows.

Initially, $DL \ = \ \langle((\,''Maths'', 2500), \langle\langle department, avgDeptSalary\rangle\rangle)\rangle$. $traceAffectPool$ ignores all the del steps, and finds the add transformation step whose $result$ is $''\langle\langle department, avgDeptSalary\rangle\rangle''$. This is step (7.5), $tp_{(7.5)}$, and:

$tp_{(7.5)}.query = \langle\langle avgMathsSalary\rangle\rangle + + \langle\langle avgCompSciSalary\rangle\rangle$ and

$tp_{(7.5)}.source = [\langle\langle avgMathsSalary\rangle\rangle,\ \langle\langle avgCompSciSalary\rangle\rangle]$

Using algorithm *affectPoolOfSet*, t's lineage at $tp_{(7.5)}$ is as follows:

$$
\begin{aligned}
DL_{(7.5)} &= \langle([x|x \leftarrow \langle\langle avgMathsSalary\rangle\rangle; x = (''Maths'', 2500)], \langle\langle avgMathsSalary\rangle\rangle), \\
&\quad ([x|x \leftarrow \langle\langle avgCompSciSalary\rangle\rangle; x = (''Maths'', 2500)], \langle\langle avgCompSciSalary\rangle\rangle)\rangle \\
&= \langle((\,''Maths'', 2500), \langle\langle avgMathsSalary\rangle\rangle), (\emptyset, \langle\langle avgCompSciSalary\rangle\rangle)\rangle \\
&= \langle((\,''Maths'', 2500), \langle\langle avgMathsSalary\rangle\rangle)\rangle
\end{aligned}
$$

After removing the original tuple, $((\,''Maths'', 2500), \langle\langle department, avgDeptSalary\rangle\rangle)$, and merging its lineage $DL_{(7.5)}$, the updated DL is $\langle((\,''Maths'', 2500), \langle\langle avgMathsSalary\rangle\rangle)\rangle$.

Similarly, we obtain the data lineage relating to above DL. Thus, $DL_{(7.2)}$ is all of the tuples in $\langle\langle mathsSalary\rangle\rangle$ and $DL_{(7.1)}$ is all of the tuples in $\langle\langle mathematician, salary\rangle\rangle$, where $\langle\langle mathematician, salary\rangle\rangle$ is a base collection in S_1.

We conclude that the affect-pool of tuple $(\,''Maths'', 2500)$ in the extent of $\langle\langle department, avgDeptSalary\rangle\rangle$ in S_2 consists of all of the tuples in the extent of $\langle\langle mathematician, salary\rangle\rangle$ in S_1.

4 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, OO or relational model, the AutoMed integration approach is based on a lower-level common data model – the HDM data model. High-level modelling languages, and the set of primitive schema transformations on each of them, are defined in terms of the HDM and its set of primitive schema transformations. 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. Each transformation step affects one schema construct, expressed in some modelling language, and the intermediate and target schemas may contain constructs of more than one modelling language.

The contribution of the work described in this chapter 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 schema transformations carry a query which defines the new schema construct in terms of the other schema constructs. Although developed for a graph-based common data model and a functional query language, our lineage tracing approach is not limited to these and could be applied in any data transformation/integration framework which based on sequences of primitive schema transformations.

We are currently implementing the algorithms presented here, and also algorithms for incremental view maintenance. The data lineage problem and the solutions presented in this chapter have led to a number of areas of further work:

- Making use of the information imparted by queries in $delete$ transformation steps, and also the partial information imparted by queries in $extend$ and $contract$ transformation steps.

- Combining data lineage tracing 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.

- Extending the lineage tracing and view maintenance algorithms to a more expressive transformation language: [21] 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] Cui, Y., Widom, J., Wiener, J.: Tracing the lineage of view data in a warehousing environment. ACM Transactions on Database Systems (TODS) **25** (2000) 179–227

[2] Bernstein, P.A., Bergstraesser, T.: Meta-data support for data transformations using microsoft repository. IEEE Data Engineering Bulletin **22** (1999) 9–14

[3] Woodruff, A., Stonebraker, M.: Supporting fine-grained data lineage in a database visualization environment. In: Proceedings of the Thirteenth International Conference on Data Engineering (ACDE'97), Birmingham, U.K, IEEE Computer Society (1997) 91–102

[4] Cui, Y.: Lineage tracing in data warehouses. PhD thesis, Computer Science Department, Stanford University (2001)

[5] Galhardas, H., Florescu, D., Shasha, D., Simon, E., Saita, C.: Improving data cleaning quality using a data lineage facility. In: Proc. Design and Management of Data Warehouses (DMDW), Interlaken, Switzerland. (2001) 3

[6] Faloutsos, C., Jagadish, H., Sidiropoulos, N.: Recovering information from summary data. In: VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, Athens, Greece, Morgan Kaufmann (1997) 36–45

[7] Hull, R.: Managing semantic heterogeneity in databases: A theoretical perspective. In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona, ACM Press (1997) 51–61

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

[9] McBrien, P., Poulovassilis, A.: Automatic migration and wrapping of database applications - a schema transformation approach. In: Conceptual Modeling - ER '99, 18th International Conference on Conceptual Modeling, Paris, France. Volume 1728 of Lecture Notes in Computer Science., Springer (1999) 96–113

[10] McBrien, P., Poulovassilis, A.: A uniform approach to inter-model transformations. In: Advanced Information Systems Engineering, 11th International Conference CAiSE'99, Heidelberg, Germany. Volume 1626 of Lecture Notes in Computer Science., Springer (1999) 333–348

[11] McBrien, P., Poulovassilis, A.: A semantic approch to integrating XML and structured data sources. In: Advanced Information Systems Engineering, 13th International Conference, CAiSE 2001, Interlaken, Switzerland. Volume 2068 of Lecture Notes in Computer Science., Springer (2001) 330–345

[12] Williams, D.: Representing RDF and RDF Schema in the HDM. Technical report, Automed Project (2002) BBKCS-02-11.

[13] Buneman, P., Khanna, S., Tan, W.: Data provenance: some basic issues. In: Foundations of Software Technology and Theoretical Computer Science, 20th Conference, (FST TCS) New Delhi, India. Volume 1974 of Lecture Notes in Computer Science., Springer (2000) 87–93

[14] Buneman, P., Khanna, S., Tan, W.: Why and where: A characterization of data provenance. In: Database Theory - ICDT 2001, 8th International Conference, London, UK. Volume 1973 of Lecture Notes in Computer Science., Springer (2001) 316–330

[15] Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. In: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy, Morgan Kaufmann (2001) 471–480

[16] Poulovassilis, A.: The Automed Intermediate Query Language. Technical report, Automed Project (2001)

[17] McBrien, P., Poulovassilis, A.: Data integration by bi-directional schema transformation rules. In: 19th International Conference on Data Engineering, ICDE'03(to appear), March 5 - March 8, 2003 - Bangalore, India. (2003)

[18] Albert, J.: Algebraic properties of bag data types. In: 17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings, Morgan Kaufmann (1991) 211–219

[19] Trinder, P.W.: Comprehensions, a query notation for dbpls. In: Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, Nafplion, Greece, Proceedings, Morgan Kaufmann (1991) 55–68

[20] Buneman, P., Libkin, L., Suciu, D., Tanen, V., Wong, L.: Comprehension syntax. SIGMOD Record **23** (1994) 87–96

[21] Poulovassilis, A.: An enhanced transformation language for the HDM. Technical report, Automed Project (2001)