Incremental view maintenance and data lineage tracing in heterogeneous database environments

Hao Fan
School of Computer Science & Information Systems
Birkbeck College, University of London
h.fan@dcs.bbk.ac.uk

1. Motivation for the research

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.

Automed (*Automatic Generation of Mediator Tools for Heterogeneous database Integration*) is a database transformation and integration system, which is designed to support both virtual and materialized integration of schemas expressed in a variety of modelling languages. In previous work of the Automed project [PM98, MP99a, MP99b], a general framework has been developed to support schema transformation and integration in heterogeneous database architectures. The framework consists of a low-level **hypergraph based data model (HDM)** and a set of primitive schema transformations on HDM schemas. 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 transformations.

The purpose of my research is to investigate techniques for *incremental view maintenance* and *data lineage tracing* for integrated databases which have been formed from heterogeneous source databases via Automed schema transformation pathways (data lineage tracing investigates how data in a data warehouse has been derived from the data sources). My approach is to decompose the processes of incremental view maintenance and data lineage tracing into a sequence of simple steps based on the transformation pathways. I use a functional *intermediate query language* (IQL) as the query language to implement the algorithms for incremental view maintenance and tracing data lineage.

The remainder of this paper is as follows. Section 2 outlines the related work. Section 3 gives the examples of the IQL language and Automed transformation pathways. Section 4 presents my research questions and research approach. The preliminary ideas and results achieved so far are given in Section 5. Section 6 describes contributions of my research so far and directions of future work.

2. Related work

An overview of materialized views and their maintenance can be found in [GM95], and [Dong99] gives a survey of incremental view maintenance. Many incremental view maintenance algorithms have been developed [Qua96, CGL$^+$96, GL95, GMS93, BLT86] which deal with views and source databases with duplicate elements (*bag* algebra) but do not apply in a multi-source environment because they assume views and tables are in the same source. Several algorithms have also been developed for incremental view maintenance in a multi-source scenario but are limited to relational *select-project-join* (SPJ) views without duplicates, including the ECA algorithm [ZGH$^+$95] designed for a system with a central database site, the Strobe algorithm [ZGW96] handling multiple, distributed source databases and the SWEEP algorithm [AASY97] computing view changes for multi-source updates collectively. [MS01] gives an algorithm for incrementally maintaining views with multiple independent data sources and bag algebra semantics. Most of these algorithms treat the query that defines the materialized view as a single SPJ query of the form $Q(S_1,S_2,\ldots,S_n) = \pi(\sigma(S_1 \cup S_2 \cup \ldots \cup S_n)$. In contrast, in my approach, the process creating the integrated database is decomposed into a sequence of transformation steps. Each step is a *primitive transformation* [PM98] possibly accompanied by an IQL query [Pou01a]. IQL queries can represent common database query operations, such as select-project-join (SPJ) operations and SPJ operations with aggregation (ASPJ).

Another recent topic is incremental view schema maintenance that needs to update not only the data but also the schema of the integrated views [KR02]. An extension of SQL, *SchemaSQL* [LSS01], is used for this kind of incremental view maintenance. The Automed project has also considered the problem of *source schema and global schema evolution* [MP02a, MP02b].

[QGM$^+$96] introduces the concept of self-maintainable views by storing auxiliary views in the data warehouse. This auxiliary data can also be used for improving the efficiency of view maintenance and lineage tracing [CW00]. I use similar ideas in my approach.

As to the problem of tracing data lineage, previous works have defined the notions of *fine-grained* data lineage [WS97], *derivation set* and *derivation pool* [CWW00], and the difference of *why-* and *where-provenance* [BKT01]. I have used all of these notions in my approach. I have introduced the concepts of *affect-* and *origin-pool* for data lineage, and developed algorithms of tracing data lineage which compute the derivation given the source schemas, integrated schema, and transformation pathways between them.

3. Examples of IQL queries and Automed transformation pathways

This section gives examples of IQL and Automed transformation pathways. More details of these can be found in [PM98, MP99a, MP99b, Pou01a].

Example 1:(IQL query) To get the maximum daily sales total for each store in a relation *StoreSales* (*store_id*, *daily_total*, *date*), in SQL we use,

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

In IQL this query is expressed by

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

where "gc" is a "group-and-compute" operator and [(s, t)| (s, t, d) ← *StoreSales*] is a comprehension. □

Example 2:(Transforming between HDM schemas) An HDM schema consists of a set of nodes, a set of edges and a set of constraints. Consider two HDM schemas $S_1 = (N_1, E_1, C_1)$ and $S_2 = (N_2, E_2, C_2)$ where N_1 = {*mathematician*, *compScientist*, *salary*}, C_1 = {},
E_1 = {«_, *mathematician*, *salary*», «_, *compScientist*, *salary*»}; N_2 = {*dept*, *person*, *salary*, *avgDeptSalary*}, C_2 = {}, E_2 = {«_, *dept, person*», «_, *person, salary*», «_, *dept, avgDeptSalary*»}.

Figure 1 illustrates these schemas S_1 and S_2. S_1 can be transformed to S_2 by the following sequence of primitive schema transformations, where "++" is a *bag append* operator.

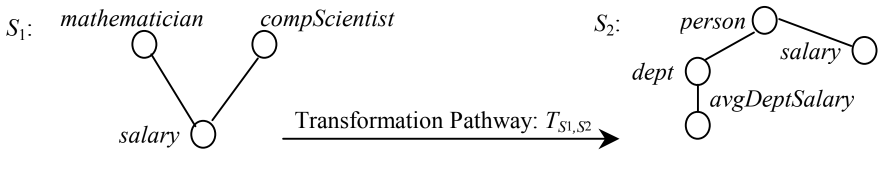

Figure 1: Transforming schema S_1 to S_2

$T_{S1,S2}$ =

```
addNode (dept, {“Maths”,“CompSci”});
addNode (person, [x| x ← mathematician] ++ [x| x ← compScientist]);
addNode (avgDeptSalary, {avg  [s| (m,s)←«_, mathematician, salary»]}  ++
                        {avg  [s| (c,s)←«_, compScientist, salary»]});
addEdge («_, dept, person», [( “Maths”, x)| x ← mathematician] ++
                        [(“CompSci”, x) | x ← compScientist]);
addEdge («_, person, salary», «_, mathematician,salary» ++ «_, compScientist, salary»);
addEdge («_, dept, avgDeptSalary», {( “Maths”, avg [s| (m,s)← «_, mathematician, salary»]),
                               (“CompSci”, avg [s| (c,s)←«_, compScientist, salary»])});
delEdge («_, mathematician, salary», [(p, s)| (d, p) ← «_, dept, person»; (p’, s) ← «_, person, salary»;
                                     d = “Maths”; p = p’]);
delEdge («_, compScientist, salary», [(p, s)| (d, p) ← «_, dept, person»; (p’, s)← «_, person, salary»;
                                     d = “CompSci”; p = p’});
delNode (mathematician, [p| (d, p) ← «_, dept, person»; d = “Maths”]);
delNode («compScientist», [p| (d, p) ← «_, dept, person»; d = “CompSci”]);
```

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

4. Research questions and approach

My research issue is to develop techniques for incrementally maintaining an integrated database and tracing the lineage of the integrated data, and I am investigating how the Automed transformation pathways can be used for both of these. The Automed transformation pathways consist of a sequence of primitive transformation steps and are automatically reversible [MP99a]. Using this feature, the process of incremental view maintenance and data lineage tracing can also be decomposed into a sequence of simple steps. One of my aims is to explore the relationship between the two processes and determine if they can be combined into an integrated approach.

To achieve these research aims, firstly, we consider simple formulae for evaluating the changes in the integrated database in response to the changes in the source databases and the simple processes for obtaining the lineage of the integrated data. All of the formulae and methods are based on *simple IQL queries* (see [FP02]), from which general IQL queries can be formed by arbitrary nesting. Then we investigate how the individual transformation steps in an Automed transformation pathway can be used to incrementally maintain materialized views and trace the derivation of the data in an integrated database in a stepwise fashion. Finally, entire procedures are given for incremental view maintenance and data lineage tracing using the Automed transformation pathways.

5. Preliminary ideas and results achieved so far

5.1 Incremental view maintenance using schema transformation pathways

Suppose we have a set of base relations D_i (i = 1, …, n), from which we derive a materialized view, V. We use ΔD_i, ∇D_i to denote the bags inserted into, deleted from a base relation, D_i, respectively. Similarly, ΔV, ∇V denote the bags inserted into, deleted from the materialized view, V, respectively. ΔD_i, ∇D_i, ΔV and ∇V are possibly empty.

The incremental maintenance of a view V is to maintain V’s data only by computing the changes to V (ΔV and ∇V) that are generated from the changes in the base relations (ΔD_i’s and ∇D_i’s). When a base relation D_i has changed, we obtain the new extent of the view as $V^{\text{new}} = (V \mathbin{++} \Delta V) \mathbin{--} \nabla V$, where "--" is the *bag monus* operator [Alb91]. Of course many such expressions for ΔV and ∇V are possible but not all are equally desirable. For example, we could simply let $\nabla V = V$ and $\Delta V = V^{\text{new}}$, but this is equivalent to recomputing the view from scratch [Qua96]. To guard against such useless definitions, it is necessary to introduce the concept of "*minimality*" [GL95] to ensure that no unnecessary tuples are produced.

Definition 1: (Minimality Condition) ΔV and ∇V should satisfy the following minimality conditions:

(1) $\nabla V \subseteq V$: we only delete tuples that are in V;
(2) $\Delta V \cap \nabla V = \emptyset$: we do not delete a tuple and then reinsert it. □

Incremental view maintenance with simple IQL queries: Let $V = q(D)$ be the bag that results from applying a simple IQL query q to a source data repository D, consisting of one or more bags. Then the formulae for computing ΔV and ∇V from ΔD and ∇D for simple IQL queries are as follows:

Simple IQL query[1]			∇V	ΔV
$D_1 \mathbin{++} D_2$			$(\nabla D_1 \mathbin{--} \Delta D_2) \mathbin{++} (\nabla D_2 \mathbin{--} \Delta D_1)$	$(\Delta D_1 \mathbin{--} \nabla D_2) \mathbin{++} (\Delta D_2 \mathbin{--} \nabla D_1)$
$D_1 \mathbin{--} D_2$			$((\nabla D_1 \mathbin{--} \nabla D_2) \mathbin{++} (\nabla D_2 \mathbin{--} \nabla D_1)) \cap V$	$((\Delta D_1 \mathbin{--} \Delta D_2) \mathbin{++} (\nabla D_2 \mathbin{--} \nabla D_1)) \mathbin{--} (D_2 \mathbin{--} D_1)$
group D			$[x \mid x \leftarrow V; y \leftarrow (\Delta D \mathbin{++} \nabla D);$ first $x =$ first $y]$	group $([x \mid x \leftarrow D; y \leftarrow (\Delta D \mathbin{++} \nabla D);$ first $x =$ first $y] \mathbin{++} \Delta D \mathbin{--} \nabla D)$
gc aggFun D	aggFun = max	ΔD	let r = $[x \mid x \leftarrow$ gc max $\Delta D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; (first x = first y) & (second $x <$ second y)]	let r = $[x \mid x \leftarrow$ gc max $\Delta D]$ in r -- $[x \mid x \leftarrow$ r; y $\leftarrow V;$ (first x = first y) & (second $x \leq$ second y)]
		∇D	let r = $[x \mid x \leftarrow$ gc max $\nabla D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; (first x = first y) & (second x = second y)]	let r = $[x \mid x \leftarrow$ gc max $\nabla D]$ in gc max $[x \mid x \leftarrow (D \mathbin{--} \nabla D);$ y $\leftarrow \nabla V;$ first x = first $y]$
	aggFun = min	ΔD	let r = $[x \mid x \leftarrow$ gc min $\Delta D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; (first x = first y) & (second $x >$ second y)]	let r = $[x \mid x \leftarrow$ gc min $\Delta D]$ in r -- $[x \mid x \leftarrow$ r; y $\leftarrow V;$ (first x = first y) & (second $x \geq$ second y)]
		∇D	let r = $[x \mid x \leftarrow$ gc min $\nabla D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; (first x = first y) & (second x = second y)]	let r = $[x \mid x \leftarrow$ gc min $\nabla D]$ in gc min $[x \mid x \leftarrow (D \mathbin{--} \nabla D);$ y $\leftarrow \nabla V;$ first x = first $y]$
	aggFun = count	ΔD	let r = $[x \mid x \leftarrow$ gc count $\Delta D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; first x = first $y]$	let r = $[x \mid x \leftarrow$ gc count $\Delta D]$ in gc sum (r ++ ∇V)
		∇D	let r = $[x \mid x \leftarrow$ gc count $\nabla D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; first x = first $y]$	let r = $[x \mid x \leftarrow$ gc count $\nabla D]$ in [(first x, (second x – second y)) $\mid x \leftarrow \nabla V;$ $y \leftarrow$ r; first x = first $y]$
	aggFun = sum	ΔD	let r = $[x \mid x \leftarrow$ gc sum $\Delta D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; first x = first $y]$	let r = $[x \mid x \leftarrow$ gc sum $\Delta D]$ in gc sum (r ++ ∇V)
		∇D	let r = $[x \mid x \leftarrow$ gc sum $\nabla D]$ in $[x \mid x \leftarrow V;$ y $\leftarrow$ r; first x = first $y]$	let r = $[x \mid x \leftarrow$ gc sum $\nabla D]$ in [(first x, (second x – second y)) $\mid x \leftarrow \nabla V;$ $y \leftarrow$ r; first x = first $y]$
	aggFun = avg	ΔD / ∇D	$[x \mid x \leftarrow V;$ y $\leftarrow (\Delta D \mathbin{++} \nabla D)$ first x = first $y]$	gc avg $[x \mid x \leftarrow (D \mathbin{++} \Delta D \mathbin{--} \nabla D);$ $y \leftarrow (\Delta D \mathbin{++} \nabla D);$ first x = first $y]$
$[x \mid x \leftarrow D_1;$ member D_2 $x]$		ΔD_1 / ∇D_1	$[x \mid x \leftarrow \nabla D_1;$ member V $x]$	$[x \mid x \leftarrow \Delta D_1;$ member D_2 $x]$
		ΔD_2 / ∇D_2	let r = $[x \mid x \leftarrow \nabla D_2;$ not (member $(D_2 \mathbin{--} \nabla D_2)$ x)] in $[x \mid x \leftarrow D_1;$ member r $x]$	let r = $[x \mid x \leftarrow \Delta D_2;$ not (member D_2 x)] in $[x \mid x \leftarrow D_1;$ member r $x]$
$[x \mid x \leftarrow D_1;$ not (member D_2 x)]		ΔD_1 / ∇D_1	$[x \mid x \leftarrow \nabla D_1;$ member V $x]$	$[x \mid x \leftarrow \Delta D_1;$ not (member D_2 x)]
		ΔD_2 / ∇D_2	$[x \mid x \leftarrow V;$ member ΔD_2 $x]$	let r = $[x \mid x \leftarrow \nabla D_2;$not (member $(D_2 \mathbin{--} \nabla D_2)$ x)] in $[x \mid x \leftarrow D_1;$ member r $x]$
$[p \mid p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$[2]		ΔD_i / ∇D_i	$[p \mid p_1 \leftarrow D_1; \ldots; p_i \leftarrow \nabla D_i; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$	$[p \mid p_1 \leftarrow D_1; \ldots; p_i \leftarrow \Delta D_i; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$

With these formulae, we can incrementally maintain a view that is created by applying a simple IQL query to the source databases. For general IQL queries, formed

[1] See [FP02] for the details of simple IQL queries. Note that the ΔV and ∇V are not applicable for the queries "aggFun D", "sort D" and "sortDistinct D", because we must recompute V entirely in these cases.
[2] In this expression, each pattern p_i is a sub-pattern of p, and $c_1; \ldots; c_k$ are conditions.

from arbitrary nesting of simple IQL queries, the procedures for incremental view maintenance can be derived from the above simple formulae. Where an integrated database is derived by an Automed transformation pathway, we have developed an algorithm for incremental view maintenance using this pathway. 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 we have an integrated schema *GS* that has been derived from this source schema *S* though an Automed transformation pathway $TP = tp_1, \ldots tp_r$. When a source database has some changes, ΔD and ∇D, we incrementally maintain the data in *GS* step by step though the transformation pathway, *TP*. We need only consider transformation steps that add or rename schema constructs (not ones that delete schema constructs). At each such step tp_i ($1 \le i \le n$), we use the current set of increments and decrements computed so far, to compute a new increment and decrement for the schema construct being added or renamed by tp_i. After considering the last step tp_n, we will have computed a set of increments/decrements for the constructs of the integrated schema *GS*.

5.2 Data lineage tracing using schema transformation pathways

As to the problem of data lineage tracing, we give the definitions of *affect-set* and *origin-set* for set semantics and *affect-pool* and *origin-pool* for bag semantics in [FP02], and refer the reader to that paper for details. What we regard as affect-provenance includes all of the source data that had some influence on the result data. Origin-provenance is simpler because here we are only interested in the specific data in the source databases from which the resulting data is extracted.

Our processes for tracing the affect-pool and origin-pool with IQL simple queries are specified below. As in [CWW00], we use *derivation tracing queries* to evaluate the lineage of a tuple *t*. That is, we apply a query to the source data repository *D* and the obtained result is the derivation of *t* in *D*. We call such a query the *tracing query for t on D*, denoted as $TQ_D(t)$.

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

Simple IQL query	$TQ^{AP}{}_D(t)$	$TQ^{OP}{}_D(t)$
D_1 ++…++ D_r	$<[x \mid x \leftarrow D_1; x = t], \ldots, [x \mid x \leftarrow D_r; x = t]>$	$<[x \mid x \leftarrow D_1; x = t], \ldots, [x \mid x \leftarrow D_r; x = t]>$
D_1 -- D_2	$<[x \mid x \leftarrow D_1; x = t], D_2>$	$<[x \mid x \leftarrow D_1; x = t], [x \mid x \leftarrow D_2; x = t]>$
Group *D*	$<[x \mid x \leftarrow D;$ first $x =$ first $t]>$	$<[x \mid x \leftarrow D;$ first $x =$ first $t]>$
sort *D*	$<[x \mid x \leftarrow D; x = t]>$	$<[x \mid x \leftarrow D; x = t]>$
sortDistinct *D*	$<[x \mid x \leftarrow D; x = t]>$	$<[x \mid x \leftarrow D; x = t]>$
aggFun *D*	$<D>$	$<[x \mid x \leftarrow D; x = t]>$ (*aggFun* = "max"| "min")
		$<D>$ (*aggFun* = "count"| "sum"| "avg")
gc aggFun *D*	$<[x \mid x \leftarrow D;$ first $x =$ first $t]>$	$<[x \mid x \leftarrow D; x = t]>$ (*aggFun* = "max"| "min")
		$<[x \mid x \leftarrow D;$ first $x =$ first $t]>$ (*aggFun* = "count"| "sum"| "avg")
$[x \mid x \leftarrow D_1;$ member $D_2\, x]$	$<[x \mid x \leftarrow D_1; x = t], [x \mid x \leftarrow D_2; x = t]>$	$<[x \mid x \leftarrow D_1; x = t], [x \mid x \leftarrow D_2; x = t]>$
$[x \mid x \leftarrow D_1;$ not (member $D_2\, x$)$]$	$<[x \mid x \leftarrow D_1; x = t], D_2>$	$<[x \mid x \leftarrow D_1; x = t]>$
$[p \mid p_1 \leftarrow D_1; \ldots; p_r \leftarrow D_r; c_1; \ldots; c_k]$ [3]	$<[p_1 \mid p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k], \ldots, [p_r \mid p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k]>$	$<[p_1 \mid p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k], \ldots, [p_r \mid p_1 \leftarrow D_1; p_1 = t_1; \ldots; p_r \leftarrow D_r; p_r = t_r; c_1; \ldots; c_k]>$

[3] Here, each pattern p_i is a sub-pattern 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*. $c_1; \ldots; c_k$ are conditions.

For more complex IQL queries, the above formulae can be recursively applied to the syntactic structure of an IQL query. An alternative (which we discuss in the Conclusions section) is to decompose a transformation step containing a complex IQL query into a sequence of transformation steps each containing a simple IQL query.

Other ongoing work within the Automed project is investigating simplification techniques for transformation pathways, such as removing matching pairs of add and delete steps for the same construct, and combining pairs of add and rename steps into a single add step [Tong02]. As a result of such simplification, we assume here that all the constructs appearing in the integrated schema *GS* must have been created from the source schema in one of three ways: (a) by an *add* transformation; (b) by a *rename* transformation; and (c) constructs existing in the source schema and remaining in the integrated schema *GS*. Thus, the problem of data lineage falls into three cases (see [FP02] for details):

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

It is simple to trace data lineage in cases (b) and (c) discussed above. If a construct *O* in *GS* was created by (b) or (c), then the lineage of a tuple *t* in *O* are all of the *t*'s copies in the source construct which is renamed or remains in *GS*. In these two cases, all of data in the construct *O* is extracted from a source database, and the affect-pool is equal to the origin-pool.

As to case (a), the key point is how to trace the lineage using the IQL query, *q*. We can use the formulae for the tracing queries given earlier to obtain the lineage of the data created in this case:

We first use two procedures *affectPoolOfTuple*(*t, O*) and *originPoolOfTuple*(*t, O*) to trace the affect pool and origin pool of a tuple, where *t* is the tracing tuple in the extent of some construct *O* of the integrated schema (see [FP02] for these procedures). The result of these procedures, D^*, is a bag which contains *t*'s derivation in the source databases.

We next consider the derivations of a tuple set[4] *T* in the extent of a construct *O*. Two procedures *affectPoolOfSet*(*T, O*) and *originPoolOfSet*(*T, O*) are used to compute the derivations of a tuple set $T = \{t_1, \ldots, t_n\}$ (see [FP02] for these procedures). In these procedures, we use the procedures *affectPoolOfTuple*(*t, O*) and *originPoolOfTuple*(*t, O*) above to trace the derivations of each tuple t_i ($1 \leq i \leq n$) in turn and incrementally add each time the result to D^*.

Finally, we give our recursive derivation tracing algorithm for tracing data lineage using entire transformation pathways, *traceAffectPool*(*TL*, *OL*), in Figure 2 (the *traceOriginPool*(*TL*, *OL*) algorithm is similar, obtained by replacing "affect" by "origin" everywhere). $TL = T_1, \ldots, T_n$ is a list of tuple sets such that each T_i is contained in the extension of some integrated schema construct O_i. *OL* is the list of integrated schema constructs $O_1, \ldots, O_n$. We assume that each schema construct has an attribute *relateTP* that refers to the transformation step that created this construct (if *O* is remaining from the source schema, then *O.relateTP* = Ø). Each transformation step has attributes *transfType*, *query*, *sourceConstruct* and *resultConstruct; query* is the query used in the transformation step; *transfType* is "*add*" or "*rename*"; and for an *add* transformation, *sourceConstruct* includes all the schema constructs appearing in the *query.*

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

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

```
procedure traceAffectPool(TL, OL)
input:    a list of tuple sets $TL = T_1, \ldots, T_n$; the
          list of corresponding constructs $OL =$
          $O_1,\ldots, O_n$ in the integrated schema;
output:   $T$'s affect pool in the source schema
begin
    $D^* \leftarrow \emptyset$;
    for $i = 1$ to $n$ do {
      $temp \leftarrow affectPoolofSet(T_i, O_i)$;
      if ($T_i.relateTP.transfType$ = "add")
          $temp \leftarrow traceAffectPool(temp,$
                    $T_i.relateTP.sourceConstruct)$
                  ;
      $D^* \leftarrow D^*$ ++ [$x$| $x \leftarrow temp$;
                  not (member $D^*$ $x$)];
    }
    return ($D^*$);
end
```

Figure 2: Affect-Pool Tracing Procedure for entire transformation pathways

6. Contributions of the work so far and future work

We have presented formulae for incrementally maintaining views defined using simple IQL queries and have discussed how the Automed transformation pathways can be used for incremental view maintenance. We have also presented formulae for tracing the affect-pool and the origin-pool for a tuple derived from simple IQL queries and described how the Automed transformation pathways can be used for this problem also. The problems of incremental view maintenance and data lineage and their solutions presented here have led to a number of areas of future work, which we expect to carry out roughly in this order during the remainder of the PhD (2.5 years):

1) *Implementing our lineage tracing and view maintenance algorithms*. As a part of the Automed project, we will implement our algorithms in Java over the Automed repository and API [BT01, Auto].
2) *Handling more complex IQL queries appearing in transformation pathways*. We will derive techniques for decomposing complex IQL queries appearing in single a transformation step into a sequence of transformation steps each accompanied by a single simple query, so that our techniques for simple queries can be applied.
3) *Extending the lineage tracing and view maintenance algorithms to a more expressive transformation language*. [Pou01b] extends the Automed transformation language with parametrised procedures and iteration and conditional constructs, and we plan to extend our algorithms to this more expressive transformation language.
4) *Combining our approach for tracing data lineage with the problem of incremental view maintenance.* We plan to explore the relationship between our lineage tracing and view maintenance algorithms, to determine if an integrated approach can be adopted for both.
5) Apply and evaluate our techniques for incremental view maintenance and data lineage tracing in the area of genomic data warehouses, in collaboration with an ongoing Bioinformatics project at Birkbeck, UCL and EBI ("Structural and Functional Annotation of Genomes through Synchronised Data Warehouses").

References

[AASY97] D. Agrawal, A. Abbadi, A. Singh and T. Yurek. Efficient view maintenance at data warehouses. In *ACM'97*, pages 417-427, May, 1997.

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

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

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

[BLT86] J. A. Blakeley, P. Larson and F. Tompa. Efficiently updating maintenance views. In *SIGMOD'86*, pages 61-71, June, 1986.

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

[CGL$^+$96] L. Colby, T. Griffin, L. Libkin, I. Mumick and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD'96*, pages 469-480, June, 1996.

[CW00] Y. Cui and J. Widom. Storing auxiliary data for efficient maintenance and lineage tracing of complex views. In *DMDW'00*, Stockholm, Sweden, June 2000.

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

[Dong99] G. Dong. Incremental maintenance of recursive views: a surevy. In A. Gupta and I. S. Mumick, editors, *Materialized Views Techniques, Implementations, and Applications*, The MIP Press, pages 159-162, 1999.

[FP02] H. Fan and A. Poulovassilis. Tracing data lineage using Automed schema transformation pathways. Automed Technical Report, April 2002. http://www.dcs.bbk.ac.uk/~hao/publications/DLSTP.pdf

[GL95] T. Griffin and L. Libkin, Incremental maintenance of views with duplicates. In *SIGMOD'95*, pages 328-339, May 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, Techniques, and Applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2): 3-18, June, 1995

[GMS93] A. Gupta, I. S. Mumick and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD'93,* Washington, DC, May 26-28 1993.

[KR02] A. Koeller and E. A. Rundensteiner. Incremental maintenance of schema-restructuring views. In *EDBT'02, LNCS 2287,* pages 354-367, Spring-Verlag Berlin Heidelberg 2002.

[LSS01] L. Lakshmanan, F. Sadri and S. Subramanian. SchemaSQL – an extension to SQL for multi-database interoperability. In *ACM TODS*, Vol. 26, No. 4 pages 476-519, December, 2001.

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

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

[MP02a] P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *CaiSE'02,* volume TBC, Springer-Verlag *LNCS*, 2002.

[MP02b] P.McBrien and A.Poulovassilis. Data Integration by Bi-Directional Schema Transformation Rules, Automed Technical Report, February 2002. http://www.dcs.bbk.ac.uk/~ap/pubs/BAVTechRep.ps

[MS01] G. Moro and C. Sartori. Incremental maintenance of multi-source views. In *Proceedings of the 12th Australasian Database Cinference* (*ACD'01*), 2001.

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

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

[Pou01b] A. Poulovassilis. An enhanced transformation language for the HDM. Automed Working Document 4. Technical Report, Birkbeck College, University of London, July 2001. http://www.doc.ic.ac.uk/automed/techreports/enhanced_transformation_language.ps

[QGM$^+$96] D. Quass, A. Gupta, I. Mumick and J. Widom. Making views self-maintainable for data warehousing. In *Proceedings of the 4th International Conference on Parallel and Distributed Information Systems (PDIS),* pages 158-169, December 1996.

[Qua96] D. Quass, Maintenance Expressions for Views with Aggregation. In *VIEWS'96*. pages 110-118, June 1996.

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

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

[ZGH$^+$95] Y. Zhuge, H. Garcia-Molina, J. Hammer and J. Widom. View maintenance in a warehousing environment. In *SIGMOD'95*, pages 316-327, May 1995.

[ZGW96] Y. Zhuge, H. Garcia-Molina and J. Wiener. The strobe algorithms for multi-source warehouse consistency. In *PDIS'96*, IEEE Computer Society, December 1996.