Lineage Tracing for General Data Warehouse Transformations*

Yingwei Cui and Jennifer Widom
Computer Science Department, Stanford University
{cyw, widom}@db.stanford.edu

Abstract. Data warehousing systems integrate information from operational data sources into a central repository to enable analysis and mining of the integrated information. During the integration process, source data typically undergoes a series of *transformations*, which may vary from simple algebraic operations or aggregations to complex "data cleansing" procedures. In a warehousing environment, the *data lineage* problem is that of tracing warehouse data items back to the original source items from which they were derived. We formally define the lineage tracing problem in the presence of general data warehouse transformations, and we present algorithms for lineage tracing in this environment. Our tracing procedures take advantage of known structure or properties of transformations when present, but also work in the absence of such information. Our results can be used as the basis for a lineage tracing tool in a general warehousing setting, and also can guide the design of data warehouses that enable efficient lineage tracing.

1 Introduction

Data warehousing systems integrate information from operational *data sources* into a central repository to enable analysis and mining of the integrated information [CD97, LW95]. Sometimes during data analysis it is useful to look not only at the information in the warehouse, but also to investigate how certain warehouse information was derived from the sources. Tracing warehouse data items back to the source data items from which they were derived is termed the *data lineage* problem [CWW00]. Enabling lineage tracing in a data warehousing environment has several benefits and applications, including in-depth data analysis and data mining, authorization management, view update, efficient warehouse recovery, and others as outlined in, e.g., [BB99, CWW00, HQGW93, LBM98, LGMW00, RS98, RS99, WS97].

In previous work [CW00, CWW00], we studied the warehouse data lineage problem in depth, but we only considered warehouse data defined as relational materialized views over the sources, i.e., views specified using SQL or relational algebra. Related work has focused on even simpler relational views [Sto75] or on multidimensional views [DB2, Pow]. In real production data warehouses, however, data imported from the sources is generally "cleansed", integrated, and summarized through a sequence or graph of *transformations*, and many commercial warehousing systems provide tools for creating and managing such transformations as part of the *extract-transform-load* (*ETL*) process, e.g., [Inf, Mic, PPD, Sag]. The transformations may vary from simple algebraic operations or aggregations to complex procedural code.

In this paper we consider the problem of lineage tracing for data warehouses created by general transformations. Since we no longer have the luxury of a fixed set of operators or the algebraic properties offered by relational views, the problem is considerably more difficult and open-ended than previous work on lineage tracing. Furthermore, since transformation graphs in real ETL processes can often be quite complex—containing as many as 60 or more transformations—the storage requirements and runtime overhead associated with lineage tracing are very important considerations.

We develop an approach to lineage tracing for general transformations that takes advantage of known structure or properties of transformations when present, yet provides tracing facilities in the absence of such information as well. Our tracing algorithms apply to single transformations, to linear sequences of transformations, and to arbitrary acyclic transformation graphs. We present optimizations that effectively reduce the storage and runtime overhead in the case of large transformation graphs. Our results can be used as the basis for an in-depth data warehouse analysis and debugging tool, by which analysts can browse their warehouse data, then trace back to the source data that produced warehouse data items of interest. Our results also can guide the design of data warehouses that enable efficient lineage tracing.

The main contributions of our work are:

- In Sections 2 and 3 we define data transformations formally and identify a set of relevant transformation properties. We define data lineage for general warehouse transformations exhibiting these properties, but we also cover "black box" transformations with no known properties. The transformation properties we consider can be specified easily by transformation authors, and they encompass a large majority of transformations used for real data warehouses.
- In Section 3 we develop lineage tracing algorithms for single transformations. Our algorithms take advantage of transformation properties when they are present, and we also suggest how indexes can be used to further improve tracing performance.
- In Section 4 and the full version of this paper [CW01] we develop a general algorithm for lineage tracing through a sequence or graph of transformations. Our algorithm includes methods for combining transformations so that we can reduce overall tracing cost, including the number of transformations we must trace through and the number of intermediate results that must be stored or recomputed for the purpose of lineage tracing.

*This work was supported by the National Science Foundation under grants IIS-9811947 and IIS-9817799.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

- We have implemented a prototype lineage tracing system based on our algorithms, and in the full version of this paper [CW01] we present a few initial performance results.

For examples in this paper we use the relational data model, but our approach and results clearly apply to data objects in general.

This version of the paper is considerably reduced from the original full version [CW01]. Readers interested in our complete treatment of the topic are directed to read the full version instead of this one. In particular, this version omits details of several algorithms, discussion of nondeterministic transformations, indexing techniques, lineage tracing for transformations with multiple input and output sets, tracing through arbitrary transformation graphs, performance experiments, avenues of future work, some examples, and proofs for all theorems.

1.1 Related Work

There has been a significant body of work on data transformations in general, including aspects such as transforming data formats, models, and schemas, e.g., [ACM$^+$99, BDH$^+$95, CR99, HMN$^+$99, LSS96, RH00, Shu87, Squ95]. Often the focus is on data integration or warehousing, but none of these papers considers lineage tracing through transformations, or even addresses the related problem of transformation inverses.

Most previous work on data lineage focuses on *coarse-grained* (or *schema-level*) lineage tracing, and uses *annotations* to provide lineage information such as which transformations were involved in producing a given warehouse data item [BB99, LBM98], or which source attributes derive certain warehouse attributes [HQGW93, RS98]. By contrast, we consider *fine-grained* (or *instance-level*) lineage tracing: we retrieve the actual set of source data items that derived a given warehouse data item. As will be seen, in some cases we can use coarse-grained lineage information (*schema mappings*) in support of our fine-grained lineage tracing techniques. In [Cui01], we extend the work in this paper with an annotation-based technique for instance-level lineage tracing, similar in spirit to the schema-level annotation techniques in [BB99]. It is worth noting that although an annotation-based approach can improve lineage tracing performance, it is likely to slow down warehouse loading and refresh, so an annotation-based approach may only be desirable for lineage-intensive applications.

In [WS97], a general framework is proposed for computing fine-grained data lineage in a transformational setting. The paper defines and traces data lineage for each transformation based on a *weak inverse*, which must be specified by the transformation definer. Lineage tracing through a transformation graph proceeds by tracing through each path one transformation at a time. In our approach, the definition and tracing of data lineage is based on general transformation properties, and we specify an algorithm for combining transformations in a sequence or graph for improved tracing performance. In [CWW00, DB2, Sto75], algorithms are provided for generating lineage tracing procedures automatically for various classes of relational and multidimensional views, but none of these approaches can handle warehouse data created through general transformations. In [FJS97], a statistical approach is used for reconstructing base (lineage) data from summary data in the presence of certain constraints. However, the approach provides only estimated lineage information and does not ensure accuracy. Finally, [LGMW00] considers an ETL setting like ours, and defines the concept of a *contributor* in order to enable efficient resumption of interrupted warehouse loads. Although similar in overall spirit, the definition of a contributor is different from our definition of data lineage, and does not capture all aspects of lineage we consider in this paper. In addition, we consider a more general class of transformations than those considered in [LGMW00].

prod-id	prod-name	category	price	valid
111	Apple IMAC	computer	1200	10/1/1998–
222	Sony VAIO	computer	3280	9/1/1998–11/30/1998
222	Sony VAIO	computer	2250	12/1/1998–9/30/1999
222	Sony VAIO	computer	1950	10/1/1999–
333	Canon A5	electronics	400	4/2/1999–
444	Sony VAIO	computer	2750	12/1/1998–

Figure 1: Source data set for `Product`

order-id	cust-id	date	prod-list
0101	AAA	2/1/1999	333(10), 222(10)
0102	BBB	2/8/1999	111(10)
0379	CCC	4/9/1999	222(5), 333(5)
0524	DDD	6/9/1999	111(20), 333(20)
0761	EEE	8/21/1999	111(10)
0952	CCC	11/8/1999	111(5)
1028	DDD	11/24/1999	222(10)
1250	BBB	12/15/1999	222(10), 333(10)

Figure 2: Source data set for `Order`

1.2 Running Example

We present a small running example, designed to illustrate problems and techniques throughout the paper. Consider a data warehouse with retail store data derived from two source tables:

`Product`(<u>`prod-id`</u>, `prod-name`, `category`, `price`, <u>`valid`</u>)
`Order`(<u>`order-id`</u>, `cust-id`, `date`, `prod-list`)

The `Product` table is mostly self-explanatory. Attribute `valid` specifies the time period during which a price is effective.[1] The `Order` table also is mostly self-explanatory. Attribute `prod-list` specifies the list of ordered products with product ID and (parenthesized) quantity for each. Sample contents of small source tables are shown in Figures 1 and 2.

Suppose an analyst wants to build a warehouse table listing computer products that had a significant sales jump in the last quarter: the last quarter sales were more than twice the average sales for the preceding three quarters. A table `SalesJump` is defined in the data warehouse for this purpose. Figure 4 shows how the contents of table

[1]We assume that `valid` is a simple string, which unfortunately is a typical ad-hoc treatment of time.

`Order`

order-id	cust-id	date	prod-list
0101	AAA	2/1/1999	333(10), 222(10)
0379	CCC	4/9/1999	222(5), 333(5)
1028	DDD	11/24/1999	222(10)
1250	BBB	12/15/1999	222(10), 333(10)

`Product`

prod-id	prod-name	category	price	valid
222	Sony VAIO	computer	2250	12/1/1998–9/30/1999
222	Sony VAIO	computer	1980	10/1/1999–

Figure 3: Lineage of $\langle$`Sony VAIO`, `11250`, `39600`$\rangle$

`SalesJump` can be specified using a *transformation graph* $\mathcal{G}$ with inputs `Order` and `Product`. $\mathcal{G}$ is a directed acyclic graph composed of the following seven transformations:

- $\mathcal{T}_1$ splits each input order according to its product list into multiple orders, each with a single ordered product and quantity. The output has schema $\langle$`order-id`, `cust-id`, `date`, `prod-id`, `quantity`$\rangle$.
- $\mathcal{T}_2$ filters out products not in the computer category.
- $\mathcal{T}_3$ effectively performs a relational join on the outputs from $\mathcal{T}_1$ and $\mathcal{T}_2$, with $\mathcal{T}_1$.`prod-id` $=$ $\mathcal{T}_2$.`prod-id` and $\mathcal{T}_1$.`date` occurring in the period of $\mathcal{T}_2$.`valid`. $\mathcal{T}_3$ also drops attributes `cust-id` and `category`, so the output has schema $\langle$`order-id`, `date`, `prod-id`, `quantity`, `prod-name`, `price`, `valid`$\rangle$.
- $\mathcal{T}_4$ computes the quarterly sales for each product. It groups the output from $\mathcal{T}_3$ by `prod-name`, computes the total sales for each product for the four previous quarters, and pivots the results to output a table with schema $\langle$`prod-name`, `q1`, `q2`, `q3`, `q4`$\rangle$, where `q1`–`q4` are the quarterly sales.
- $\mathcal{T}_5$ computes from the output of $\mathcal{T}_4$ the average sales of each product in the first three quarters. The output schema is $\langle$`prod-name`, `q1`, `q2`, `q3`, `avg3`, `q4`$\rangle$, where `avg3` is the average sales $($`q1` $+$ `q2` $+$ `q3`$)/3$.
- $\mathcal{T}_6$ selects those products whose last quarter's sales were greater than twice the average of the preceding three quarters.
- $\mathcal{T}_7$ performs a final projection to output `SalesJump` with schema $\langle$`prod-name`, `avg3`, `q4`$\rangle$.

Note that some of these transformations ($\mathcal{T}_2$, $\mathcal{T}_5$, $\mathcal{T}_6$, and $\mathcal{T}_7$) could be expressed as standard relational operations, while others ($\mathcal{T}_1$, $\mathcal{T}_3$, and $\mathcal{T}_4$) could not.

As a simple lineage example, for the data in Figures 1 and 2 the warehouse table `SalesJump` contains tuple $t = \langle$`Sony VAIO`, `11250`, `39600`$\rangle$, indicating that the sales of VAIO computers jumped from an average of 11250 in the first three quarters to 39600 in the last quarter. An analyst may want to see the relevant detailed information by tracing the lineage of tuple t, that is, by inspecting the original input data items that produced t. Using the techniques to be developed in this paper, from the source data in Figures 1 and 2 the analyst will be presented with the lineage result in Figure 3.

2 Transformations and Data Lineage

In this section, we formalize general data transformations and data lineage, then we briefly motivate why transformation properties can help us with lineage tracing.

$$\begin{tikzcd}
\texttt{Order} \arrow[r] & \mathcal{T}_1 \arrow[dr] & & & & & & \\
& & \mathcal{T}_3 \arrow[r] & \mathcal{T}_4 \arrow[r] & \mathcal{T}_5 \arrow[r] & \mathcal{T}_6 \arrow[r] & \mathcal{T}_7 \arrow[r] & \texttt{SalesJump} \\
\texttt{Product} \arrow[r] & \mathcal{T}_2 \arrow[ur] & & & & & &
\end{tikzcd}$$

Figure 4: Transformations to derive `SalesJump`

2.1 Transformations

Let a *data set* be any set of data items—tuples, values, complex objects—with no duplicates in the set. (The effect duplicates have on lineage tracing has been addressed in some detail in [CWW00].) A *transformation* $\mathcal{T}$ is any procedure that takes data sets as input and produces data sets as output. Here we will consider only transformations that take a single data set as input and produce a single output set. We extend our results to transformations with multiple input sets and output sets in [CW01]. For any input data set I, we say that the application of $\mathcal{T}$ to I resulting in an output set O, denoted $\mathcal{T}(I) = O$, is an *instance* of $\mathcal{T}$.

Given transformations $\mathcal{T}_1$ and $\mathcal{T}_2$, their *composition* $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ is the transformation that first applies $\mathcal{T}_1$ to I to obtain I', then applies $\mathcal{T}_2$ to I' to obtain O. $\mathcal{T}_1$ and $\mathcal{T}_2$ are called $\mathcal{T}$'s *component transformations*. The composition operation is associative: $(\mathcal{T}_1 \circ \mathcal{T}_2) \circ \mathcal{T}_3 = \mathcal{T}_1 \circ (\mathcal{T}_2 \circ \mathcal{T}_3)$. Thus, given transformations $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$, we represent the composition $((\mathcal{T}_1 \circ \mathcal{T}_2) \circ \ldots) \circ \mathcal{T}_n$ as a *transformation sequence* $\mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$. A transformation that is not defined as a composition of other transformations is *atomic*.

For now we will assume that all of our transformations are *stable* and *deterministic*. A transformation $\mathcal{T}$ is stable if it never produces spurious output items, i.e., $\mathcal{T}(\varnothing) = \varnothing$. A transformation is deterministic if it always produces the same output set given the same input set. All of the example transformations we have seen are stable and deterministic. An example of an unstable transformation is one that appends a fixed data item or set of items to every output set, regardless of the input. An example of a nondeterminstic transformation is one that transforms a random sample of the input set. In practice we usually require transformations to be stable but often do not require them to be deterministic. See [CW01] for a discussion of when the deterministic assumption can be dropped.

2.2 Data Lineage

In the general case a transformation may inspect the entire input data set to produce each item in the output data set, but in most cases there is a much more fine-grained relationship between the input and output data items: a data item o in the output set may have been derived from a small subset of the input data items (maybe only one), as opposed to the entire input data set. Given a transfor-

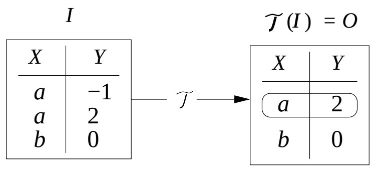


Figure 5: A transformation instance

mation instance $\mathcal{T}(I) = O$ and an output item $o \in O$, we call the actual set $I^* \subseteq I$ of input data items that contributed to o's derivation the *lineage* of o, and we denote it as $I^* = \mathcal{T}^*(o, I)$. The lineage of a set of output data items $O^* \subseteq O$ is the union of the lineage of each item in the set: $\mathcal{T}^*(O^*, I) = \bigcup_{o \in O^*} \mathcal{T}^*(o, I)$. A detailed definition of data lineage for different types of transformations will be given in Section 3.

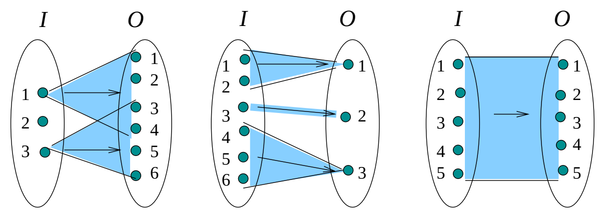
Knowing something about the workings of a transformation is important for tracing data lineage—if we know nothing, any input data item may have participated in the derivation of an output item. Let us consider an example. Given a transformation $\mathcal{T}$ and its instance $\mathcal{T}(I) = O$ in Figure 5, the lineage of the output item $\langle a, 2\rangle$ depends on $\mathcal{T}$'s definition, as we will illustrate. Suppose $\mathcal{T}$ is a transformation that filters out input items with a negative Y value (i.e., $\mathcal{T} = \sigma_{Y \geq 0}$ in relational algebra). Then the lineage of output item $o = \langle a, 2\rangle$ should include only input item $\langle a, 2\rangle$. Now, suppose instead that $\mathcal{T}$ groups the input data items based on their X values and computes the sum of their Y values multiplied by 2 (i.e., $\mathcal{T} = \alpha_{X, 2*sum(Y)\ as\ Y}$ in relational algebra, where α performs grouping and aggregation). Then the lineage of output item $o = \langle a, 2\rangle$ should include input items $\langle a, -1\rangle$ and $\langle a, 2\rangle$, because o is computed from both of them. We will refer back to these two transformations later (along with our earlier examples from Section 1.2), so let us call the first one $\mathcal{T}_8$ and the second one $\mathcal{T}_9$.

Given a transformation specified as a standard relational operator or view, we can define and retrieve the exact data lineage for any output data item using the techniques introduced in [CWW00]. On the other hand, if we know nothing at all about a transformation, then the lineage of an output item must be defined as the entire input set. In reality transformations often lie between these two extremes—they are not standard relational operators, but they have some known structure or properties that can help us identify and trace data lineage.

The transformation properties we will consider often can be specified easily by the transformation author, or they can be inferred from the transformation definition (as relational operators, for example), or possibly even "learned" from the transformation's behavior. In this paper, we do not focus on how properties are specified or discovered, but rather on how they are exploited for lineage tracing.

3 Tracing Using Transformation Properties

We consider three overall kinds of properties and provide algorithms that trace data lineage using these properties. First, each transformation is in a certain *transformation class* based on how it maps input data items to output items (Section 3.1). Second, we may have one or more *schema mappings* for a transformation, specifying how certain output attributes relate to input attributes (Section 3.2). Third, a transformation may be accompanied by a *tracing procedure* or *inverse transformation*, which is the best case for lineage tracing (Section 3.3). When a transformation exhibits many properties, we determine the best one for lineage tracing based on a property hierarchy (Section 3.4).



(a) dispatcher (b) aggregator (c) black−box

Figure 6: Transformation classes

3.1 Transformation Classes

In this section, we define three transformation classes: *dispatchers*, *aggregators*, and *black-boxes*. For each class, we give a formal definition of data lineage and specify a lineage tracing procedure. We also consider several subclasses for which we specify more efficient tracing procedures. Our informal studies have shown that about 95% of the transformations used in real data warehouses are dispatchers, aggregators, or their compositions (covered in Section 4), and a large majority fall into the more efficient subclasses.

3.1.1 Dispatchers

A transformation $\mathcal{T}$ is a *dispatcher* if each input data item produces zero or more output data items independently: $\forall I,\ \mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$. Figure 6(a) illustrates a dispatcher, in which input item 1 produces output items 1–4, input item 3 produces output items 3–6, and input item 2 produces no output items. The lineage of an output item o according to a dispatcher $\mathcal{T}$ is defined as $\mathcal{T}^*(o, I) = \{i \in I \mid o \in \mathcal{T}(\{i\})\}$.

A simple procedure `TraceDS`$(\mathcal{T}, O^*, I)$ can be used to trace the lineage of a set of output items $O^* \subseteq O$ according to a dispatcher $\mathcal{T}$. The procedure applies $\mathcal{T}$ to the input data items one at a time and returns those items that produce one or more items in O^*.[2] Note that all of our tracing procedures are specified to take a set of output items as a parameter instead of a single output item, for generality and also for tracing lineage through transformation sequences and graphs.

Example 3.1 (Lineage Tracing for Dispatchers) Transformation $\mathcal{T}_1$ in Section 1.2 is a dispatcher, because each input order produces one or more output orders via $\mathcal{T}_1$. Given an output item $o = \langle$`0101`, `AAA`, `2/1/1999`, `222`, `10`$\rangle$ based on the sample data of Figure 2, we

[2]For now we are assuming that the input set is readily available. Cases where the input set is unavailable or unnecessary will be considered later.

can trace o's lineage according to $\mathcal{T}_1$ using procedure `TraceDS(`$\mathcal{T}_1, \{o\},$ `Order)` to obtain $\mathcal{T}_1^*(o,$ `Order`$)$ = $\{\langle$`0101, AAA, 2/1/1999, "333(10), 222(10)"`$\rangle\}$. Transformations $\mathcal{T}_2$, $\mathcal{T}_5$, $\mathcal{T}_6$, and $\mathcal{T}_7$ in Section 1.2 and $\mathcal{T}_8$ in Section 2.2 all are dispatchers, and we can similarly trace data lineage for them. □

`TraceDS` requires a complete scan of the input data set, and for each input item i it calls transformation $\mathcal{T}$ over $\{i\}$ which can be very expensive if $\mathcal{T}$ has significant overhead (e.g., startup time). In [CW01] we discuss how indexes can be used to improve the performance of `TraceDS`. Next we introduce a common subclass of dispatchers, *filters*, for which lineage tracing is trivial.

Filters. A dispatcher $\mathcal{T}$ is a *filter* if each input item produces either itself or nothing: $\forall i \in I$, $\mathcal{T}(\{i\}) = \{i\}$ or $\mathcal{T}(\{i\}) = \varnothing$. Thus, the lineage of any output data item is the same item in the input set: $\forall o \in O, \mathcal{T}^*(o) = \{o\}$. The tracing procedure for a filter $\mathcal{T}$ simply returns the traced item set O^* as its own lineage. It does not need to call the transformation $\mathcal{T}$ or scan the input data set, which can be a significant advantage in many cases (see Section 4). Transformation $\mathcal{T}_8$ in Section 2.2 is a filter, and the lineage of output item $o = \langle a, 2\rangle$ is the same item $\langle a, 2\rangle$ in the input set. Other examples of filters are $\mathcal{T}_2$ and $\mathcal{T}_6$ in Section 1.2.

3.1.2 Aggregators

A transformation $\mathcal{T}$ is an *aggregator* if $\mathcal{T}$ is *complete* (defined momentarily), and for all I and $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$, there exists a unique disjoint partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$ for $k = 1..n$. $I_1, \ldots, I_n$ is called the *input partition*, and I_k is o_k's lineage according to $\mathcal{T}$: $\mathcal{T}^*(o_k, I) = I_k$. A transformation $\mathcal{T}$ is *complete* if each input data item always contributes to some output data item: $\forall I \neq \varnothing, \mathcal{T}(I) \neq \varnothing$. Figure 6(b) illustrates an aggregator, where the lineage of output item 1 is input items $\{1, 2\}$, the lineage of output item 2 is $\{3\}$, and the lineage of output item 3 is $\{4, 5, 6\}$.

Transformation $\mathcal{T}_9$ in Section 2 is an aggregator. The input partition is $I_1 = \{\langle a, -1\rangle, \langle a, 2\rangle\}$, $I_2 = \{\langle b, 0\rangle\}$, and the lineage of output item $o = \langle a, 2\rangle$ is I_1. Among the transformations in Section 1.2, $\mathcal{T}_4$, $\mathcal{T}_5$, and $\mathcal{T}_7$ are aggregators. Note that transformations can be both aggregators and dispatchers (e.g., $\mathcal{T}_5$ and $\mathcal{T}_7$ in Section 1.2). We will address how overlapping properties affect lineage tracing in Section 3.4.

To trace the lineage of an output subset O^* according to an aggregator $\mathcal{T}$, we use procedure `TraceAG(`$\mathcal{T}, O^*, I$`)`, which enumerates subsets of input I. It returns the unique subset I^* such that I^* produces exactly O^*, i.e., $\mathcal{T}(I^*) = O^*$, and the rest of the input set produces the rest of the output set, i.e., $\mathcal{T}(I - I^*) = O - O^*$. During the enumeration, we examine the subsets in increasing size. If we find a subset I' such that $\mathcal{T}(I') = O^*$ but $\mathcal{T}(I - I') \neq O - O^*$, we then need to examine only supersets of I', which can reduce the work significantly.

`TraceAG` may call $\mathcal{T}$ as many as $2^{|I|}$ times in the worst case, which can be prohibitive. We introduce two

```
procedure TraceCF($\mathcal{T}, O^*, I$)
  $I^* \leftarrow \varnothing$;
  $pnum \leftarrow 0$;
  for each $i \in I$ do
    if $pnum = 0$ then $I_1 \leftarrow \{i\}$; $pnum \leftarrow 1$; continue;
    for $(k \leftarrow 1; k \leq pnum; k++)$ do
      if $|\mathcal{T}(I_k \cup \{i\})| = 1$ then $I_k \leftarrow I_k \cup \{i\}$; break;
    if $k > pnum$ then $pnum \leftarrow pnum + 1$; $I_{pnum} \leftarrow \{i\}$;
  for $k \leftarrow 1..pnum$ do
    if $\mathcal{T}(I_k) \subseteq O^*$ then $I^* \leftarrow I^* \cup I_k$;
return $I^*$;
```

Figure 7: Tracing proc. for context-free aggregators

common subclasses of aggregators, *context-free aggregators* and *key-preserving aggregators*, which allow us to apply much more efficient tracing procedures.

Context-Free Aggregators. An aggregator $\mathcal{T}$ is *context-free* if any two input data items either always belong to the same input partition, or they always do not, regardless of the other items in the input set. In other words, a context-free aggregator determines the partition that an input item belongs to based on its own value, and not on the values of any other input items. All example aggregators we have seen are context-free. As an example of a non-context-free aggregator, consider a transformation $\mathcal{T}$ that clusters input data points based on their x-y coordinates and outputs some aggregate value of items in each cluster. Suppose $\mathcal{T}$ specifies that any two points within distance d from each other must belong to the same cluster. $\mathcal{T}$ is an aggregator, but it is not context-free, since whether two items belong to the same cluster or not may depend on the existence of a third item near to both.

We specify tracing procedure `TraceCF(`$\mathcal{T}, O^*, I$`)` in Figure 7 for context-free aggregators. This procedure first scans the input data set to create the partitions (which we could not do linearly if the aggregator were not context-free), then it checks each partition to find those that produce items in O^*. `TraceCF` reduces the number of transformation calls to $|I^2| + |I|$ in the worst case, which is a significant improvement.

Key-Preserving Aggregators. Suppose each input item and output item contains a unique *key* value in the relational sense, denoted i.*key* for item i. An aggregator $\mathcal{T}$ is *key-preserving* if given any input set I and its input partition $I_1, \ldots, I_n$ for output $\mathcal{T}(I) = \{o_1, \ldots, o_n\}$, all subsets of I_k produce a single output item with the same key value as o_k, for $k = 1..n$. That is, $\forall I' \subseteq I_k$: $\mathcal{T}(I') = \{o'_k\}$ and o'_k.*key* = o_k.*key*.

Theorem 3.2 Key-preserving aggregators are context-free. □

All example aggregators we have seen are key-preserving. As an example of a context-free but non-key-preserving aggregator, consider a relational groupby-aggregation that does not retain the grouping attribute.

To trace the lineage of O^* according to a key-preserving aggregator $\mathcal{T}$, we use procedure `TraceKP(`$\mathcal{T}, O^*, I$`)`, which scans the input data set once and returns all input items that produce output items with the same key as items in O^*. TraceKP reduces the number of transformation calls to $|I|$, with each call operating on a single input data item. We can further improve performance of TraceKP using an index, as discussed in [CW01].

3.1.3 Black-box Transformations

An atomic transformation is called a *black-box* transformation if it is neither a dispatcher nor an aggregator, and it does not have a *provided lineage tracing procedure* (Section 3.3). In general, any subset of the input items may have been used to produce a given output item through a black-box transformation, as illustrated in Figure 6(c), so all we can say is that the entire input data set is the lineage of each output item: $\forall o \in O$, $\mathcal{T}^*(o, I) = I$. Thus, the tracing procedure for a black-box transformation simply returns the entire input I.

As an example of a true black-box, consider a transformation $\mathcal{T}$ that sorts the input data items and attaches a serial number to each output item according to its sorted position. For instance, given input data set $I = \{\langle f, 10\rangle, \langle b, 20\rangle, \langle c, 5\rangle\}$ and sorting by the first attribute, the output is $\mathcal{T}(I) = \{\langle \mathbf{1}, b, 20\rangle, \langle \mathbf{2}, c, 5\rangle, \langle \mathbf{3}, f, 10\rangle\}$, and the lineage of each output data item is the entire input set I. Note that in this case each output item, in particular its serial number, is indeed derived from all input data items.

3.2 Schema Mappings

Schema information can be very useful in the ETL process, and many data warehousing systems require transformation programmers to provide some schema information. In this section, we discuss how we can use schema information to improve lineage tracing for dispatchers and aggregators. Sometimes schema information also can improve lineage tracing for a black-box transformation $\mathcal{T}$, specifically when $\mathcal{T}$ can be combined with another non-black-box transformation based on $\mathcal{T}$'s schema information (Section 4). A schema specification may include:

input schema $\mathbf{A} = \langle A_1..A_p\rangle$ and *input key* $A_{key} \subseteq \mathbf{A}$
output schema $\mathbf{B} = \langle B_1..B_q\rangle$ and *output key* $B_{key} \subseteq \mathbf{B}$

The specification also may include *schema mappings*, defined as follows.

Definition 3.3 (Schema Mappings) Consider a transformation $\mathcal{T}$ with input schema $\mathbf{A}$ and output schema $\mathbf{B}$. Let $A \subseteq \mathbf{A}$ and $B \subseteq \mathbf{B}$ be lists of input and output attributes. Let $i.A$ denote the A attribute values of i, and similarly for $o.B$. Let f and g be functions from tuples of attribute values to tuples of attribute values. We say that $\mathcal{T}$ has a *forward schema mapping* $f(A) \overset{\mathcal{T}}{\rightarrow} B$ if we can partition any input set I into $I_1, \ldots, I_m$ based on equality of $f(A)$ values,[3] and partition the output set $O = \mathcal{T}(I)$ into $O_1, \ldots, O_n$ based on equality of B values, such that $m \geq n$ and:

1. for $k = 1..n$, $\mathcal{T}(I_k) = O_k$ and $I_k = \{i \in I \mid f(i.A) = o.B$ for any $o \in O_k\}$.
2. for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

Similarly, we say that $\mathcal{T}$ has a *backward schema mapping* $A \overset{\mathcal{T}}{\leftarrow} g(B)$ if we can partition any input set I into $I_1, \ldots, I_m$ based on equality of A values, and partition the output set $O = \mathcal{T}(I)$ into $O_1, \ldots, O_n$ based on equality of $g(B)$ values, such that $m \geq n$ and:

1. for $k = 1..n$, $\mathcal{T}(I_k) = O_k$ and $I_k = \{i \in I \mid i.A = g(o.B)$ for any $o \in O_k\}$.
2. for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

When f (or g) is the identity function, we simply write $A \overset{\mathcal{T}}{\rightarrow} B$ (or $A \overset{\mathcal{T}}{\leftarrow} B$). If $A \overset{\mathcal{T}}{\rightarrow} B$ and $A \overset{\mathcal{T}}{\leftarrow} B$ we write $A \overset{\mathcal{T}}{\leftrightarrow} B$. □

Although Definition 3.3 may seem cumbersome, it formally and accurately captures the intuitive notion of schema mappings (certain input attributes producing certain output attributes) that transformations often exhibit.

Example 3.4 Schema information for transformation $\mathcal{T}_5$ in Section 1.2 can be specified as:

Input schema and key:
$\mathbf{A} = \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4}\rangle$, $A_{key} = \langle \texttt{prod-name}\rangle$
Output schema and key:
$\mathbf{B} = \langle \texttt{prod-name}, \texttt{q1..q4}, \texttt{avg3}\rangle$. $B_{key} = \langle \texttt{prod-name}\rangle$
Schema mappings:
$\langle \texttt{prod-name}, \texttt{q1..q4}\rangle \overset{\mathcal{T}_5}{\leftrightarrow} \langle \texttt{prod-name}, \texttt{q1..q4}\rangle$
$f(\langle \texttt{q1..q3}\rangle) \overset{\mathcal{T}_5}{\rightarrow} \langle \texttt{avg3}\rangle$, where $f(\langle a, b, c\rangle) = \frac{(a+b+c)}{3}$ □

Theorem 3.5 Consider a transformation $\mathcal{T}$ that is a dispatcher or an aggregator, and consider any instance $\mathcal{T}(I) = O$. Given any output item $o \in O$, let I^* be o's lineage according to the lineage definition for $\mathcal{T}$'s transformation class in Section 3.1. If $\mathcal{T}$ has a forward schema mapping $f(A) \overset{\mathcal{T}}{\rightarrow} B$, then $I^* \subseteq \{i \in I \mid f(i.A) = o.B\}$. If $\mathcal{T}$ has a backward schema mapping $A \overset{\mathcal{T}}{\leftarrow} g(B)$, then $I^* \subseteq \{i \in I \mid i.A = g(o.B)\}$. □

Based on Theorem 3.5, when tracing lineage for a dispatcher or aggregator, we can narrow down the lineage of any output data item to a (possibly very small) subset of the input data set based on a schema mapping. We can then retrieve the exact lineage within that subset using the algorithms in Section 3.1. For example, consider an aggregator $\mathcal{T}$ with a backward schema mapping $A \overset{\mathcal{T}}{\leftarrow} g(B)$. When tracing the lineage of an output item $o \in O$ according to $\mathcal{T}$, we can first find the input subset $I' = \{i \in I \mid i.A = g(o.B)\}$, then enumerate subsets of I' using TraceAG($\mathcal{T}, o, I'$) to find o's lineage $I^* \subseteq I'$. If we have multiple schema mappings for $\mathcal{T}$, we can use their intersection for improved tracing efficiency.

Although the narrowing technique of the previous paragraph is effective, when schema mappings satisfy certain additional conditions, we obtain transformation properties that permit very efficient tracing procedures.

Definition 3.6 (Schema Mapping Properties) Consider a transformation $\mathcal{T}$ with input schema $\mathbf{A}$, input key A_{key}, output schema $\mathbf{B}$, and output key B_{key}.

[3] That is, two input items $i_1 \in I$ and $i_2 \in I$ are in the same partition I_k iff $f(i_1.A) = f(i_2.A)$.

2. $\mathcal{T}$ is a *forward key-map (fkmap)* if it is complete ($\forall I \neq \varnothing$, $\mathcal{T}(I) \neq \varnothing$) and it has a forward schema mapping to the output key: $f(A) \overset{\mathcal{T}}{\rightarrow} B_{key}$.
2. $\mathcal{T}$ is a *backward key-map (bkmap)* if it has a backward schema mapping to the input key: $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$.
3. $\mathcal{T}$ is a *backward total-map (btmap)* if it has a backward schema mapping to all input attributes: $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$. □

Suppose that schema information and mappings are given for all transformations in Section 1.2. Then all of the transformations except $\mathcal{T}_4$ are backward key-maps; $\mathcal{T}_2$, $\mathcal{T}_5$, and $\mathcal{T}_6$ are backward total-maps; $\mathcal{T}_4$, $\mathcal{T}_5$, and $\mathcal{T}_7$ are forward key-maps.

Theorem 3.7 (1) All filters are backward total-maps. (2) All backward total-maps are backward key-maps. (3) All backward key-maps are dispatchers. (4) All forward key-maps are key-preserving aggregators. □

Theorem 3.8 Consider a transformation instance $\mathcal{T}(I) = O$. Given an output item $o \in O$, let I^* be o's lineage based on $\mathcal{T}$'s transformation class as defined in Section 3.1.

1. If $\mathcal{T}$ is a forward key-map with schema mapping $f(A) \overset{\mathcal{T}}{\rightarrow} B_{key}$, then $I^* = \{i \in I | f(i.A) = o.B_{key}\}$.
2. If $\mathcal{T}$ is a backward key-map with schema mapping $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$, then $I^* = \{i \in I | i.A_{key} = g(o.B)\}$.
3. If $\mathcal{T}$ is a backward total-map with schema mapping $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$, then $I^* = \{g(o.B)\}$. □

According to Theorem 3.8, we can use simple tracing procedures for transformations with the schema mapping properties specified in Definition 3.6. Procedure `TraceFM(`$\mathcal{T}, O^*, I$`)` performs lineage tracing for a forward key-map $\mathcal{T}$, which by Theorem 3.7 also could be traced using procedure `TraceKP` from Section 3.1.2. Both algorithms scan each input item once, however `TraceKP` applies transformation $\mathcal{T}$ to each item, while `TraceFM` applies function f to some attributes of each item. Certainly f is very unlikely to be more expensive than $\mathcal{T}$, since $\mathcal{T}$ effectively computes f and may do other work as well; f may in fact be quite a bit cheaper. `TraceBM(`$\mathcal{T}, O^*, I$`)` uses a similar approach for a backward key-map, and is usually more efficient than `TraceDS(`$\mathcal{T}, O^*, I$`)` from Section 3.1.1 for the same reasons. `TraceTM(`$\mathcal{T}, O^*$`)` performs lineage tracing for a backward total-map, which is very efficient since it does not need to scan the input data set and makes no transformation calls. In [CW01], we discuss how indexes can be used to further speed up procedures `TraceFM` and `TraceBM`.

3.3 Provided Tracing Procedure or Inverse

If we are very lucky, a lineage *tracing procedure* may be provided along with the specification of a transformation $\mathcal{T}$. The tracing procedure `TP` may require access to the

Property	Tracing Procedure	# Transformation Calls	# Input Accesses
dispatcher	`TraceDS`	$\|I\|$	$\|I\|$
filter	**return** o	0	0
aggregator	`TraceAG`	$O(2^{\|I\|})$	$O(2^{\|I\|})$
context-free aggr.	`TraceCF`	$O(\|I\|^2)$	$O(\|I\|^2)$
key-preserving aggr.	`TraceKP`	$\|I\|$	$\|I\|$
black-box	**return** I	0	0
forward key-map	`TraceFM`	0	$\|I\|$
backward key-map	`TraceBM`	0	$\|I\|$
backward total-map	`TraceTM`	0	0
tracing-proc. w/ input	`TP`	?	?
tracing-proc. w/o input	`TP`	?	0

Figure 8: Summary of transformation properties

input data set, i.e., `TP`(O^*, I) returns O^*'s lineage according to $\mathcal{T}$, or the tracing procedure may not require access to the input, i.e., `TP`(O^*) returns O^*'s lineage. A related but not identical situation is when we are provided with the *inverse* for a transformation $\mathcal{T}$. Sometimes, but not always, $\mathcal{T}$'s inverse can be used as $\mathcal{T}$'s tracing procedure.

Definition 3.9 (Inverse Transformation) A transformation $\mathcal{T}$ is *invertible* if there exists a transformation $\mathcal{T}^{-1}$ such that $\forall I$, $\mathcal{T}^{-1}(\mathcal{T}(I)) = I$, and $\forall O$, $\mathcal{T}(\mathcal{T}^{-1}(O)) = O$. $\mathcal{T}^{-1}$ is called $\mathcal{T}$'s *inverse.* □

Theorem 3.10 If a transformation $\mathcal{T}$ is an aggregator with inverse $\mathcal{T}^{-1}$, then for all instances $\mathcal{T}(I) = O$ and all $o \in O$, o's lineage according to $\mathcal{T}$ is $\mathcal{T}^{-1}(\{o\})$. □

According to Theorem 3.10, we can use a transformation's inverse for lineage tracing if the invertible transformation is an aggregator. However, if the invertible transformation is a dispatcher or black-box, we cannot always use its inverse for lineage tracing, as illustrated by an example in [CW01].

Although we can guarantee very little about the accuracy or efficiency of provided tracing procedures or transformation inverses in the general case, it is our experience that, when provided, they are usually the most effective way to perform lineage tracing. We will make this assumption in the remainder of the paper.

3.4 Property Summary and Hierarchy

Figure 8 summarizes the transformation properties covered in the previous three sections. The table specifies which tracing procedure is applicable for each property, along with the number of transformation calls and number of input data item accesses for each procedure. We omit transformation inverses from the table, since when applicable they are equivalent to a provided tracing procedure not requiring input.

As discussed earlier, a transformation may satisfy more than one property. Some properties are better than others: tracing procedures may be more efficient, they may return a more accurate lineage result, or they may not require access to input data. Figure 9 specifies a hierarchy for determining which property is best to use for a specific transformation. In the hierarchy, a solid arrow from property p_1 to p_2 means that p_2 is more restrictive than p_1,

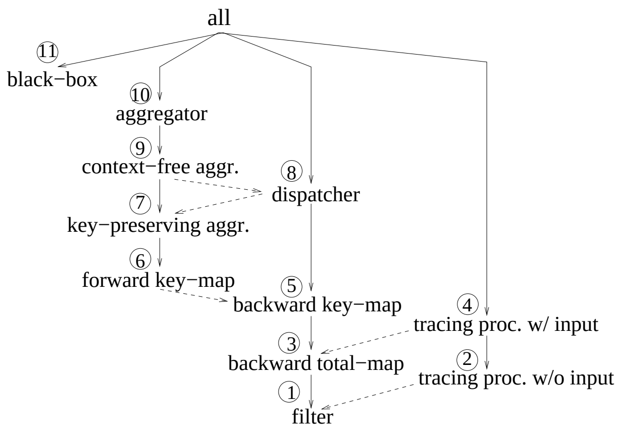


Figure 9: Transformation property hierarchy

	Best Property	Additional Properties
$\mathcal{T}_1$	backward key-map	
$\mathcal{T}_2$	filter	
$\mathcal{T}_3$	see [CW01]	
$\mathcal{T}_4$	forward key-map	
$\mathcal{T}_5$	backward total-map	forward key-map
$\mathcal{T}_6$	filter	
$\mathcal{T}_7$	backward key-map	forward key-map
$\mathcal{T}_8$	filter	
$\mathcal{T}_9$	forward key-map	

Figure 10: Properties of $\mathcal{T}_1$–$\mathcal{T}_9$

i.e., all transformations that satisfy property p_2 also satisfy property p_1. Further, according to Sections 3.1–3.3, whenever p_2 is more restrictive than p_1, the tracing procedure for p_2 is no less efficient (and usually more efficient) by any measure: number of transformation calls, number of input accesses, and whether the input data is required at all. (Black-box transformations, which are the only type with less accurate lineage results, are placed in a separate branch of the hierarchy.) A dashed arrow from property p_1 to p_2 in the hierarchy means that even though p_2 is not strictly more restrictive than p_1, p_2 does yield a tracing procedure that again is no less efficient (and usually more efficient) by any measure.[4]

Let us make the reasonable assumption that a provided tracing procedure requiring input is more efficient than `TraceBM`, and that a tracing procedure not requiring input is more efficient than `TraceTM`. Then we can derive a total order of the properties as shown by the numbers in Figure 9: the lower the number, the better the property is for lineage tracing. Given a set of properties for a transformation $\mathcal{T}$, we always use the best one, i.e., the one with the lowest number, to trace data lineage for $\mathcal{T}$. Figure 10 lists the best property for example transformations $\mathcal{T}_1$–$\mathcal{T}_7$ from Section 1.2 and $\mathcal{T}_8$–$\mathcal{T}_9$ from Section 2.2, along with other properties satisfied by these transformations. Note that we list only the most restrictive property on each branch of the hierarchy.

[4]In some cases the tracing efficiency difference represented by a solid or dashed arrow is significant, while in other cases it is less so. This issue is discussed further in Section 4.

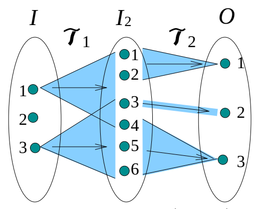


Figure 11: $\mathcal{T}_1 \circ \mathcal{T}_2$

4 Lineage Tracing through a Sequence

Having specified how to trace lineage for a single transformation with one input set and one output set, we now consider lineage tracing for sequences of such transformations. Multiple input and output sets and arbitrary acyclic transformation graphs are covered in [CW01].

4.1 Data Lineage for a Transformation Sequence

Consider a simple sequence of two transformations, such as $\mathcal{T}_1 \circ \mathcal{T}_2$ in Figure 11 composed from Figures 6(a) and 6(b). For an input data set I, let $I_2 = \mathcal{T}_1(I)$ and $O = \mathcal{T}_2(I_2)$. Given an output data item $o \in O$, if $I_2^* \subseteq I_2$ is the lineage of o according to $\mathcal{T}_2$, and $I^* \subseteq I$ is the lineage of I_2^* according to $\mathcal{T}_1$, then I^* is the lineage of o according to $\mathcal{T}_1 \circ \mathcal{T}_2$. For example, in Figure 11 if $o \in O$ is item 3, then I_2^* is items $\{4, 5, 6\}$ in I_2, and I^* is items $\{1, 3\}$ in I. This lineage definition generalizes to arbitrarily long transformation sequences using the associativity of composition.

Given a transformation sequence $\mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$, where each I_k is the intermediate result output from $\mathcal{T}_{k-1}$ and input to $\mathcal{T}_k$, a correct but brute-force approach is to store all intermediate results $I_2, \ldots, I_n$ (in addition to initial input I) at loading time, then trace lineage backward through one transformation at a time. This approach is inefficient both due to the large number of tracing procedure calls when iterating through all transformations in the sequence, and due to the high storage cost for all intermediate results. The longer the sequence, the less efficient the overall tracing process, and for realistic transformation sequences (in practice sometimes as many as 60 transformations) the cost can be prohibitive. Furthermore, if any transformation $\mathcal{T}$ in the sequence is a black-box, we will end up tracing the lineage of the entire input to $\mathcal{T}$ regardless of what transformations follow $\mathcal{T}$ in the sequence. Fortunately, it is often possible to relieve these problems by *combining* adjacent transformations in a sequence for the purpose of lineage tracing. Also since we do not always need input sets for lineage tracing as discussed in Section 3, some intermediate results can be discarded.

We will use the following overall strategy.

- When a transformation sequence $\mathcal{S} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$ is defined, we first *normalize* the sequence, to be specified in Section 4.2, by combining transformations in $\mathcal{S}$ when it is beneficial to do so. We then determine which intermediate results need to be saved for lineage tracing, based on the best properties for the remaining transformations.
- When data is loaded through the transformation sequence, the necessary intermediate results are saved.

- We can then trace the lineage of any output data item o in the warehouse through the normalized transformation sequence using the iterative tracing procedure described at the beginning of this section.

4.2 Transformation Sequence Normalization

As discussed in Section 4.1, we want to combine transformations in a sequence for the purpose of lineage tracing when it is beneficial to do so. Specifically, we can *combine* transformations $\mathcal{T}_{k-1}$ and $\mathcal{T}_k$ by replacing the two transformations with the single transformation $\mathcal{T}' = \mathcal{T}_{k-1} \circ \mathcal{T}_k$, eliminating the intermediate result I_k and tracing through the combined transformation in one step.

To decide whether combining a pair of transformations is beneficial, and to use combined transformations for lineage tracing, as a first step we need to determine the properties of a combined transformation based on the properties of its component transformations. Function `Combine(`$\mathcal{T}_1, \mathcal{T}_2$`)`, specified in [CW01], returns combined transformation $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ and sets $\mathcal{T}$'s properties based on those of $\mathcal{T}_1$ and $\mathcal{T}_2$.

Theoretically we can combine any adjacent transformations in a sequence, in fact we can collapse the entire sequence into one large transformation, but combined transformations may have less desirable properties than their component transformations, leading to less efficient or less accurate lineage tracing. Thus, we want to combine transformations only if it is beneficial to do so. Given a transformation sequence, determining the best way to combine transformations in the sequence is a difficult combinatorial problem—solving it accurately, or even just determining accurately when it is beneficial to combine two transformations, would require a detailed cost model that takes into account transformation properties, the cost of applying a transformation, the cost of storing intermediate results, and an estimated workload (including, e.g., data size and tracing frequency). Developing such a cost model is beyond the scope of this paper.

Instead, we suggest a greedy algorithm `Normalize` shown in Figure 12. The algorithm repeatedly finds beneficial combinations of transformation pairs in the sequence, combines the "best" pair, and continues until no more beneficial combinations are found. In general, a combination should be considered beneficial only if it reduces the overall tracing cost while improving or retaining tracing accuracy. We determine whether it is beneficial to combine two transformations based solely on their properties using the following two heuristics. First, we do not combine transformations into black-boxes, unless we are certain that the combination will not degrade the accuracy of the lineage result, which can only be determined as a last step of the `Normalize` procedure. Second, we do not combine transformations if their composition is significantly worse for lineage tracing, i.e., it has much higher tracing cost or leads to a less accurate result. We divide the properties in Figure 9 into five groups: group 1 contains properties 1–3, group 2 contains properties 4–8, group 3 contains property 9, group 4 contains property 10, and group 5 contains property 11. Within each group, the efficiency and accuracy of the tracing procedures are fairly similar, while they differ significantly across groups. The group of a transformation $\mathcal{T}$, denoted $group(\mathcal{T})$, is the group that $\mathcal{T}$'s best property belongs to. The lower the group number, the better $\mathcal{T}$ is for lineage tracing, and we consider it beneficial to combine two transformations $\mathcal{T}_1$ and $\mathcal{T}_2$ only if $group(\mathcal{T}_1 \circ \mathcal{T}_2) \leq max(group(\mathcal{T}_1), group(\mathcal{T}_2))$.[5]

Based on the above approach, procedure `BestCombo` in Figure 12, called by `Normalize`, finds the best pair of adjacent transformations to combine in sequence $\mathcal{S}$, and returns its index. The procedure returns 0 if no combination is beneficial. We consider a beneficial combination to be the best if the combination leaves the fewest "bad" transformations in the sequence, compared with other candidates. Formally, we associate with $\mathcal{S}$ a vector $N[1..5]$, where $N[j]$ is the number of transformations in $\mathcal{S}$ that belong to group j. (So $\sum_{j=1..5} N[j]$ equals the length of $\mathcal{S}$.) Given two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ with vectors N_1 and N_2 respectively, let k be the highest index in which $N_1[k]$ differs from $N_2[k]$. We say $N_1 < N_2$ if $N_1[k] < N_2[k]$, which implies that $\mathcal{S}_1$ has fewer "bad" transformations than $\mathcal{S}_2$. Then we say that the best combination is the one that leads to the lowest vector N for the resulting sequence.

After we finish combining transformations as described above, suppose the sequence still contains one or more black-box transformations. During lineage tracing, we will end up tracing the lineage of the entire input to the earliest (left-most) black-box $\mathcal{T}$ in $\mathcal{S}$, regardless of what transformations follow $\mathcal{T}$. Therefore, as a final step we combine $\mathcal{T}$ with all transformations that follow $\mathcal{T}$ to eliminate unnecessary tracing and storage costs.

Our `Normalize` procedure has complexity $O(n^2)$ for a transformation sequence of length n. Although we use a greedy algorithm and heuristics for estimating the benefit of combining transformations, our approach is quite effective in improving tracing performance for sequences, as shown in [CW01].

Example 4.1 Consider the sequence of transformations $\mathcal{S} = \mathcal{T}_4 \circ \mathcal{T}_5 \circ \mathcal{T}_6 \circ \mathcal{T}_7$ from Section 1.2. Figure 13 shows the sequence and the best property of each transformation. The initial vector of $\mathcal{S}$ is $N = [2, 2, 0, 0, 0]$. Using our greedy normalization algorithm, we first consider combining $\mathcal{T}_4 \circ \mathcal{T}_5$ into $\mathcal{T}_4'$ with best property *fkmap*, combining $\mathcal{T}_5 \circ \mathcal{T}_6$ into $\mathcal{T}_5'$ with best property *btmap*, or combining $\mathcal{T}_6 \circ \mathcal{T}_7$ into $\mathcal{T}_6'$ with best property *bkmap*. It turns out that all these combinations reduce $\mathcal{S}$'s vector N to $[1, 2, 0, 0, 0]$. So let us combine $\mathcal{T}_4 \circ \mathcal{T}_5$ obtaining $\mathcal{T}_4'$, $\mathcal{T}_6$, and $\mathcal{T}_7$. In the new sequence, combining $\mathcal{T}_4' \circ \mathcal{T}_6$ results in a black-box, which is disallowed, while combining $\mathcal{T}_6 \circ \mathcal{T}_7$ results in a transformation $\mathcal{T}_6'$ with best property *bkmap*, which reduces N to $[0, 2, 0, 0, 0]$. Therefore, we choose to combine $\mathcal{T}_6 \circ \mathcal{T}_7$ obtaining $\mathcal{T}_4'$ and $\mathcal{T}_6'$. Combining these two transformations would result in a black-box, so we stop at this point. The final normalized sequence is shown

[5]Note that the presence of non-key-map schema mappings (Section 3.2) or indexes [CW01] for a transformation $\mathcal{T}$ does not improve $\mathcal{T}$'s tracing efficiency to the point of moving it to a different group. Thus, we do not take these factors into account in our decision process.

```
procedure Normalize($\mathcal{S} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$)
  while ($k \leftarrow$ BestCombo($\mathcal{S}$)) $\neq 0$
      replace $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$ with $\mathcal{T} \leftarrow$ Combine($\mathcal{T}_k, \mathcal{T}_{k+1}$);
  if $\mathcal{S}$ contains black-box transformations then
      $j \leftarrow$ lowest index of a black-box in $\mathcal{S}$;
      replace $\mathcal{T}_j, \ldots, \mathcal{T}_n$ with $\mathcal{T} \leftarrow \mathcal{T}_j \circ \cdots \circ \mathcal{T}_n$;
```

```
procedure BestCombo($\mathcal{S} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$)
  $k \leftarrow 0$;  $N[1..5] \leftarrow [0,0,0,0,0]$;
  for $j = 1..n$ do $g \leftarrow group(\mathcal{T}_j)$;  $N[g] \leftarrow N[g] + 1$;
  for $j = 1..n-1$ do
      $curN \leftarrow N$;
      $\mathcal{T} \leftarrow$ Combine($\mathcal{T}_j, \mathcal{T}_{j+1}$);  $g \leftarrow group(\mathcal{T})$;
      if $g < 5$ then  // $\mathcal{T}$ is not a black-box
          $g_1 \leftarrow group(\mathcal{T}_j)$;  $g_2 \leftarrow group(\mathcal{T}_{j+1})$;
          $curN[g] \leftarrow curN[g] + 1$;
          $curN[g_1] \leftarrow curN[g_1] - 1$;
          $curN[g_2] \leftarrow curN[g_2] - 1$;
          if $curN < N$  then $k \leftarrow j$;  $N \leftarrow curN$;
return $k$;
```

Figure 12: Normalizing a transformation sequence

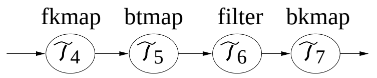


Figure 13: Before normalization

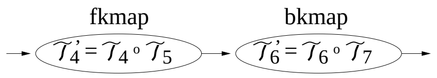

fkmap
bkmap
$\mathcal{T}_4' = \mathcal{T}_4 \circ \mathcal{T}_5$
$\mathcal{T}_6' = \mathcal{T}_6 \circ \mathcal{T}_7$

Figure 14: After normalization

in Figure 14. □

5 Conclusions and Reminder

We have developed a complete set of techniques for data warehouse lineage tracing when the warehouse data is loaded through a graph of general transformations. Some of our techniques are presented in this version of the paper, but interested readers are strongly encouraged to investigate the full version [CW01] for a complete treatment of the topic: `http://dbpubs.stanford.edu/pub/2001-5`.

References

[ACM+99] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Simeon, and S. Zohar. Tools for data translation and integration. *IEEE Data Engineering Bulletin*, 22(1):3–8, March 1999.

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

[BDH+95] P. Buneman, S.B. Davidson, K. Hart, G.C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. of the Twenty-first International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, September 1995.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[CR99] K.T. Claypool and E.A. Rundensteiner. Flexible database transformations: The SERF approach. *IEEE Data Engineering Bulletin*, 22(1):19–24, March 1999.

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

[CW00] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. of the Sixteenth International Conference on Data Engineering*, pages 367–378, San Diego, California, February 2000.

[CW01] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. Technical report, Stanford University Database Group, January 2001. http://dbpubs.stanford.edu/pub/2001-5.

[CWW00] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, June 2000.

[DB2] IBM: DB2 OLAP Server. http://www.software.ibm.com/data/db2/.

[FJS97] C. Faloutsos, H.V. Jagadish, and N.D. Sidiropoulos. Recovering information from summary data. In *Proc. of the Twenty-Third International Conference on Very Large Data Bases*, pages 36–45, Athens, Greece, August 1997.

[HMN+99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P.M. Schwarz, and E.L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, March 1999.

[HQGW93] N. I. Hachem, K. Qiu, M. Gennert, and M. Ward. Managing derived data in the Gaea scientific DBMS. In *Proc. of the Ninteenth International Conference on Very Large Data Bases*, pages 1–12, Dublin, Ireland, August 1993.

[Inf] Informix Formation Data Transformation Tool. http://www.informix.com/informix/products/integration/formation/formation.html.

[LBM98] T. Lee, S. Bressan, and S. Madnick. Source attibution for querying against semi-structured documents. In *Proc. of the Workshop on Web Information and Data Management*, pages 33–39, Washington, DC, November 1998.

[LGMW00] W.J. Labio, H. Garcia-Molina, and J.L. Weiner. Efficient resumption of interrupted warehouse loads. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 46–57, Dallas, Texas, May 2000.

[LSS96] L. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL – a language for interoperability in relational multi-database systems. In *Proc. of the Twenty-Second International Conference on Very Large Data Bases*, pages 239–250, India, September 1996.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing*, IEEE Data Engineering Bulletin 18(2), June 1995.

[Mic] Microsoft SQL Server, Data Transformation Services. http://msdn.microsoft.com/library/psdk/sql/dts_ovrw.htm.

[Pow] Cognos: PowerPlay OLAP Data Analysis and Reporting Tool. http://www.cognos.com/powerplay/.

[PPD] PPD Informatics: TableTrans Data Transformation Software. http://www.belmont.com/tt.html.

[RH00] V. Raman and J. Hellerstein. Potters Wheel: An interactive framework for data cleaning. Technical report, U.C. Berkeley, 2000. http://control.cs.berkeley.edu/abc.

[RS98] A. Rosenthal and E. Sciore. Propagating integrity information among interrelated databases. In *Proc. of the Second Working Conference on Integrity and Internal Control in Information Systems*, pages 5–18, Warrenton, Virginia, November 1998.

[RS99] A. Rosenthal and E. Sciore. First class views: A key to user-centered computing. *SIGMOD Record*, 28(3):29–36, March 1999.

[Sag] Sagent Technology. http://www.sagent.com/.

[Shu87] N.C. Shu. Automatic data transformation and restructuring. In *Proc. of the Third International Conference on Data Engineering*, pages 173–180, Los Angeles, California, February 1987.

[Squ95] C. Squire. Data extraction and transformation for the data warehouse. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 446–447, California, May 1995.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 65–78, California, May 1975.

[WS97] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of the Thirteenth International Conference on Data Engineering*, pages 91–102, Birmingham, UK, April 1997.