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, CW01, 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 lin-

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

eage 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 this paper are summarized as follows:

- 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 Sections 4–6 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.
- We have implemented a prototype lineage tracing system based on our algorithms, and in Section 7 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.

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

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`

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 tranformation 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 in [LGMW00] is different from our definition of data lineage, and does not capture all aspects of data lineage we consider in this paper. In addition, we consider a more general class of transformations than those considered in [LGMW00].

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`(`prod-id`, `prod-name`, `category`, `price`, `valid`) // ⟨`prod-id`, `valid`⟩ is a key
`Order`(`order-id`, `cust-id`, `date`, `prod-list`) // `order-id` is a key

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

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

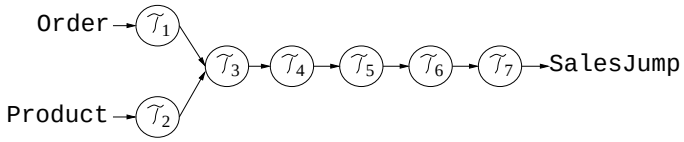


Figure 3: Transformations to derive `SalesJump`

Name	Description
$\mathcal{T}_1$	split orders
$\mathcal{T}_2$	select on product `category`
$\mathcal{T}_3$	join products and orders
$\mathcal{T}_4$	aggregate and pivot quarterly sales
$\mathcal{T}_5$	add a column `avg3`
$\mathcal{T}_6$	select on `avg3`
$\mathcal{T}_7$	remove columns

Figure 4: Transformation summary

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 3 shows how the contents of table `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 $(\texttt{q1}+\texttt{q2}+\texttt{q3})/3$.
- $\mathcal{T}_6$ 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`$\rangle$.

Figure 4 summarizes the transformations in $\mathcal{G}$. 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 5.

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

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.

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. For now, we will consider only transformations that take a single data set as input and produce a single output set. We will extend our results to transformations with multiple input sets and output sets in Section 5. 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. We will defer our discussion of when the deterministic assumption can be dropped to Section 3.5, after we have formalized data lineage and presented our lineage tracing algorithms.

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

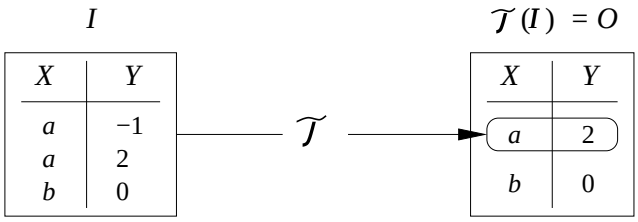


Figure 6: A transformation instance

only one), as opposed to the entire input data set. Given a transformation 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.

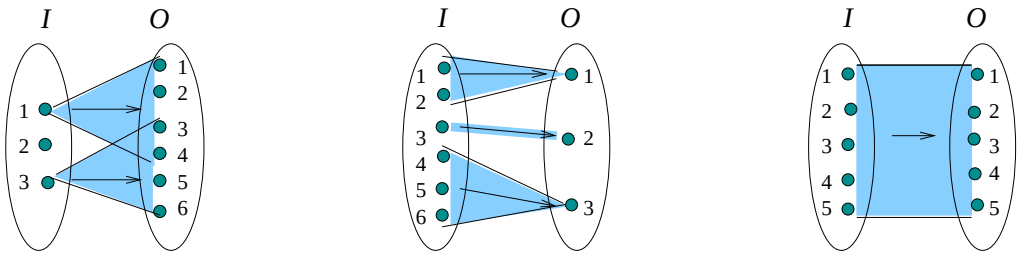
Knowing something about the workings of a transformation is important for tracing data lineage—if we know nothing, any input data item may have participated in the derivation of an output item. Let us consider an example. Given a transformation $\mathcal{T}$ and its instance $\mathcal{T}(I) = O$ in Figure 6, 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 Lineage 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 to exploit for lin-



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

Figure 7: Transformation classes

eage tracing based on a property hierarchy (Section 3.4). We also discuss nondeterministic transformations (Section 3.5), and how indexes can be used to further improve tracing performance (Section 3.6).

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 Sections 4–6), 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 7(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(`$\mathcal{T}, O^*, I$`)` in Figure 8 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 so tracing procedures can be composed when we consider transformation sequences (Section 4) and graphs (Section 6).

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 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. □

[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.

```
procedure TraceDS(T, O*, I)
  I* ← ∅;
  for each i ∈ I do
    if T({i}) ∩ O* ≠ ∅ then I* ← I* ⊎ {i};
return I*;
```

Figure 8: Tracing procedure for dispatchers

```
procedure TraceAG(T, O*, I)
  L ← all subsets of I sorted by size;
  for each I* ∈ L in increasing order do
    if T(I*) = O* then
      if T(I − I*) = O − O* then break;
      else L = all supersets of I* sorted by size;
return I*;
```

Figure 9: Tracing procedure for aggregators

`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 Section 3.6 we will discuss how indexes can be used to improve the performance of `TraceDS`. However, 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 7(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 can use the procedure `TraceAG(`$\mathcal{T}, O^*, I$`)` in Figure 9 that 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 common subclasses of aggregators, *context-free aggregators* and *key-preserving aggregators*, which allow us to apply much more efficient tracing procedures.

```
procedure TraceCF(T, O*, I)
  I* ← ∅;
  pnum ← 0;
  for each i ∈ I do
    if pnum = 0 then I_1 ← {i}; pnum ← 1; continue;
    for (k ← 1; k ≤ pnum; k++) do
      if |T(I_k ∪ {i})| = 1 then I_k ← I_k ∪ {i}; break;
    if k > pnum then pnum ← pnum + 1; I_pnum ← {i};
  for k ← 1..pnum do
    if T(I_k) ⊆ O* then I* ← I* ∪ I_k;
return I*;
```

Figure 10: Tracing proc. for context-free aggregators

```
procedure TraceKP(T, O*, I)
  I* ← ∅;
  for each i ∈ I do
    if π_key(T({i})) ⊆ π_key(O*)
    then I* ← I* ⊎ {i};
return I*;
```

Figure 11: Tracing proc. for key-preserving aggrs.

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 lineage tracing procedure `TraceCF(`$\mathcal{T}, O^*, I$`)` in Figure 10 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 All key-preserving aggregators are context-free.

Proof: See Appendix A.1. □

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.

`TraceKP(`$\mathcal{T}, O^*, I$`)` in Figure 11 traces the lineage of O^* according to a key-preserving aggregator $\mathcal{T}$. It 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 Section 3.6.

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 7(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 $\mathbf{A} = \langle A_1, \ldots, A_p\rangle$, and *input key* $A_{key} \subseteq \mathbf{A}$

output schema $\mathbf{B} = \langle B_1, \ldots, B_q\rangle$, 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* $f(A) \xrightarrow{\mathcal{T}} B$ 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* $A \xleftarrow{\mathcal{T}} g(B)$ 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 $A \xrightarrow{\mathcal{T}} B$ (or $A \xleftarrow{\mathcal{T}} B$). If $A \xrightarrow{\mathcal{T}} B$ and $A \xleftarrow{\mathcal{T}} B$ we write $A \xleftrightarrow{\mathcal{T}} B$. □

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

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 do frequently 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: $\mathbf{B} = \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{avg3}, \texttt{q4} \rangle$. $B_{key} = \langle \texttt{prod-name} \rangle$

Schema mappings: $\langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4} \rangle \overset{\mathcal{T}_5}{\leftrightarrow} \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4} \rangle$

$f(\langle \texttt{q1}, \texttt{q2}, \texttt{q3} \rangle) \overset{\mathcal{T}_5}{\rightarrow} \langle \texttt{avg3} \rangle$, where $f(\langle a, b, c \rangle) = (a + b + c)/3$ □

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 $I^* \subseteq \{i \in I \mid i.A = g(o.B)\}$.

Proof: See Appendix A.2. □

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 the intersection of the subsets 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}.

1. $\mathcal{T}$ 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.

Proof: See Appendix A.3. □

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.

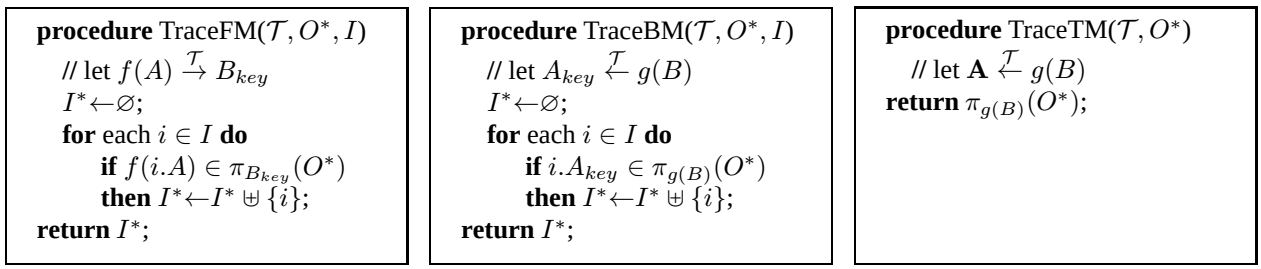


Figure 12: Tracing procedures using schema mappings

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

Proof: See Appendix A.4. □

According to Theorem 3.8, we can use the tracing procedures shown in Figure 12 for transformations with the schema mapping properties specified in Definition 3.6. For example, 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` of Figure 11. 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`$(\mathcal{T}, O^*, I)$ of Figure 8 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.

Example 3.9 Considering some examples from Section 1.2:

- $\mathcal{T}_1$ is a backward key-map with schema mapping `order-id` $\xleftarrow{\mathcal{T}_1}$ `order-id`. We can trace the lineage of an output data item o using `TraceBM`, which simply retrieves items in `Order` that have the same `order-id` as o.
- $\mathcal{T}_4$ is a forward key-map with schema mapping `prod-name` $\xrightarrow{\mathcal{T}_4}$ `prod-name`. We can trace the lineage of an output data item o using `TraceFM`, which simply retrieves the input items that have the same `prod-name` as o.
- $\mathcal{T}_5$ is a backward total-map with $\langle$`prod-name`, `q1`, `q2`, `q3`, `q4`$\rangle \xleftarrow{\mathcal{T}_5} \langle$`prod-name`, `q1`, `q2`, `q3`, `q4`$\rangle$. We can trace the lineage of an output data item o using `TraceTM`, which directly constructs $\langle o.$`prod-name`, $o.$`q1`, $o.$`q2`, $o.$`q3`, $o.$`q4`$\rangle$ as o's lineage. □

In Section 3.6 we will discuss how indexes can be used to further speed up procedures `TraceFM` and `TraceBM`.

3.3 Provided Tracing Procedure or Transformation 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 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., $\mathsf{TP}(O^*)$ 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, the inverse of $\mathcal{T}$ can be used as $\mathcal{T}$'s tracing procedure.

Definition 3.10 (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 3.11 If a transformation $\mathcal{T}$ is an aggregator with inverse $\mathcal{T}^{-1}$, then for all instances $\mathcal{T}(I) = O$ and all $o \in O$, the lineage of o according to $\mathcal{T}$ is $\mathcal{T}^{-1}(\{o\})$.

Proof: See Appendix A.5. □

According to Theorem 3.11, we can use a transformation's inverse for lineage tracing if the invertible transformation is an aggregator, as we will illustrate in Example 3.12(a). However, if the invertible transformation is a dispatcher or black-box, we cannot always use its inverse for lineage tracing, as we will illustrate in Example 3.12(b).

Example 3.12 (Lineage Tracing Using Inverses)

(a) Consider a transformation $\mathcal{T}$ that performs list merging, essentially the opposite of transformation $\mathcal{T}_1$ in Section 1.2. $\mathcal{T}$ takes a two-attribute input set and produces a two-attribute output set. It groups the input set according to the first attribute, then produces one output item for each group containing the grouping value along with a list of the second attribute values from the input. For instance, given input data set $I = \{\langle 1, a\rangle, \langle 1, c\rangle, \langle 2, b\rangle, \langle 2, g\rangle\ \langle 2, h\rangle\}$, the output is $O = \mathcal{T}(I) = \{\langle 1, \text{“}a, c\text{”}\rangle, \langle 2, \text{“}b, g, h\text{”}\rangle\}$. $\mathcal{T}$ has an inverse $\mathcal{T}^{-1}$ which splits the second attribute of its input data items to produce multiple output items, i.e., $\mathcal{T}^{-1}(O) = I$.

$\mathcal{T}$ is an aggregator, so according to Theorem 3.11 we can perform lineage tracing for $\mathcal{T}$ by applying its inverse $\mathcal{T}^{-1}$ to the traced data item(s). For example, given output item $o = \langle 2, \text{“}b, g, h\text{”}\rangle \in O$, o's lineage is $\mathcal{T}^{-1}(\{o\}) = \{\langle 2, b\rangle, \langle 2, g\rangle, \langle 2, h\rangle\}$.

(b) Now consider transformation $\mathcal{T}_1$ from Section 1.2, which is a dispatcher and has an inverse $\mathcal{T}_1^{-1}$ that assembles lists in a similar manner to transformation $\mathcal{T}$ above. If we apply inverse transformation $\mathcal{T}_1^{-1}$ to output item $\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \texttt{222}, \texttt{10}\rangle$ then we obtain $\{\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \text{“}\texttt{222(10)}\text{”}\rangle\}$, instead of the correct lineage $\{\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \text{“}\texttt{333(10)}, \texttt{222(10)}\text{”}\rangle\}$. □

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 Transformation Property Summary and Hierarchy

Figure 13 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.

Property	Tracing Procedure	# of Transformation Calls	# of Input Accesses
dispatcher	`TraceDS`	$\|I\|$	$\|I\|$
filter	**return** o	0	0
aggregator	`TraceAG`	$O(2^{\|I\|})$	$O(2^{\|I\|})$
context-free aggregator	`TraceCF`	$O(\|I\|^2)$	$O(\|I\|^2)$
key-preserving aggregator	`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 procedure requiring input	`TP`	?	?
tracing procedure not requiring input	`TP`	?	0

Figure 13: Summary of transformation properties

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 14 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, i.e., all transformations that satisfy property p_2 also satisfy property p_1. Further, according to Figure 13, 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 14: 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 15 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.

3.5 Nondeterministic Transformations

Recall from Section 2 that we have assumed all transformations to be deterministic. The reason we sometimes require determinism is that several of our tracing procedures call transformation $\mathcal{T}$, often repeatedly, as part of the lineage tracing process, specifically procedures `TraceAG`, `TraceCF`, `TraceKP`, and `TraceDS`. Those procedures that do not call transformation $\mathcal{T}$—`TraceBM`, `TraceTM`, `TraceFM`, tracing procedures

[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.

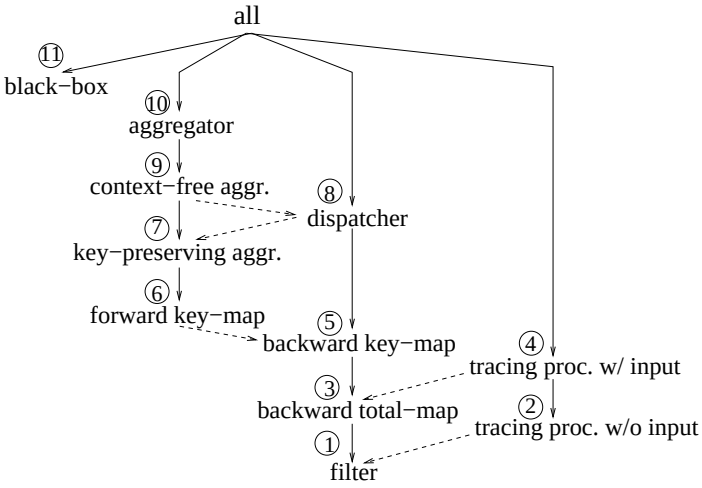


Figure 14: Transformation property hierarchy

	Best Property	Additional Properties
$\mathcal{T}_1$	backward key-map	
$\mathcal{T}_2$	filter	
$\mathcal{T}_3$	see Section 5	
$\mathcal{T}_4$	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 15: Properties of $\mathcal{T}_1$–$\mathcal{T}_9$

for filters and black-boxes, and user-provided tracing procedures—do not require determinism. Note that a transformation that selects a random sample of the input is a filter, and a transformation that attaches timestamps to tuples is a backward total-map, so neither of these common nondeterministic transformations poses a problem in our approach.

3.6 Improving Tracing Performance Using Indexes

Several of the lineage tracing algorithms presented in Sections 3.1–3.3 can be sped up if we can build indexes on the input data set. We consider two types of indexes:

- *Conventional indexes*, which allow us to quickly locate data items matching a given value. We can use a conventional index on a key for I to speed up procedure `TraceBM`, as well as the schema mapping "narrowing down" technique in Section 3.2.
- *Functional indexes* (e.g., [Ora]), which are constructed for a given function F and allow us to quickly locate data items i such that $F(i) = V$ for a value V. We can use a functional index with $F = f$, where f is the schema mapping function, to speed up procedure `TraceFM` (and again the schema mapping "narrowing down" in Section 3.2). We can also use a functional index with $F = \mathcal{T}$ to speed up procedures `TraceDS` and `TraceKP`.

Of course we could also build a complete *lineage index*, which maps the key of an output data item o to the set of input items that comprise o's lineage. This approach is similar to using *annotations* to record instance-level lineage [Cui01], which can be very expensive and tends to be worthwhile only for lineage-intensive warehouses, as discussed in Section 1.1. Some intermediate kinds of indexes—less expensive than lineage indexes but more specialized than the two index types discussed above—may also be beneficial for some of our tracing procedures, but are not explored further in this paper.

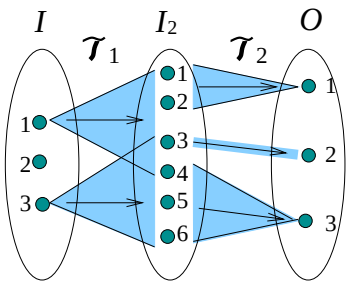


Figure 16: $\mathcal{T}_1 \circ \mathcal{T}_2$

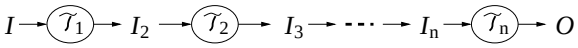


Figure 17: Transformation sequence

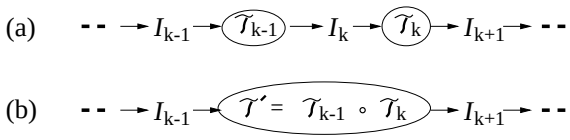


Figure 18: Combining transformations

4 Lineage Tracing through a Transformation Sequence

Now that we have specified how to perform lineage tracing for a single transformation with one input set and one output set, we will consider lineage tracing for sequences of such transformations. Multiple input and output sets are discussed in Section 5, and arbitrary acyclic transformation graphs are covered in Section 6.

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 16 composed from Figures 7(a) and 7(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 16 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 $\mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$ as illustrated in Figure 17, 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 Sections 3.1–3.3, 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.

procedure Combine($\mathcal{T}_1, \mathcal{T}_2$)
  $\mathcal{T} \leftarrow \mathcal{T}_1 \circ \mathcal{T}_2$;
  $\mathcal{T}.\mathbf{A} \leftarrow \mathcal{T}_1.\mathbf{A}$; $\mathcal{T}.A_{key} \leftarrow \mathcal{T}_1.A_{key}$;
  $\mathcal{T}.\mathbf{B} \leftarrow \mathcal{T}_2.\mathbf{B}$; $\mathcal{T}.B_{key} \leftarrow \mathcal{T}_2.B_{key}$;
  $\mathcal{T}$.*fmappings* $\leftarrow \varnothing$; $\mathcal{T}$.*bmappings* $\leftarrow \varnothing$; $\mathcal{T}$.*properties* $\leftarrow \varnothing$;
  $\mathcal{T}$.*complete* $\leftarrow \mathcal{T}_1$.*complete* **and** $\mathcal{T}_2$.*complete*;
  for each property p in {*aggregator*, *dispatcher*, *filter*, *tracing-proc w/o input*} **do**
    if $p \in \mathcal{T}_1$.*properties* **and** $p \in \mathcal{T}_2$.*properties* **then** add p to $\mathcal{T}$.*properties*;
  for each $f_1(A) \rightarrow A'$ in $\mathcal{T}_1$.*fmappings* **do**
    if $\exists\ f_2(A') \rightarrow B$ in $\mathcal{T}_2$.*fmappings* **then** add $f_1 \circ f_2(A) \rightarrow B$ to $\mathcal{T}$.*fmappings*;
  for each $A \leftarrow g_1(A')$ in $\mathcal{T}_1$.*bmappings* **do**
    if $\exists\ A' \leftarrow g_2(B)$ in $\mathcal{T}_2$.*bmappings* **then** add $A \leftarrow g_2 \circ g_1(B)$ to $\mathcal{T}$.*bmappings*;
  if $\exists\ f(A) \rightarrow \mathcal{T}.B_{key}$ in $\mathcal{T}$.*fmappings* **and** $\mathcal{T}$.*complete* **then** add *fkmap* to $\mathcal{T}$.*properties*;
  if $\exists\ \mathcal{T}.A_{key} \leftarrow g(B)$ in $\mathcal{T}$.*bmappings* **then** add *bkmap* to $\mathcal{T}$.*properties*;
  if $\exists\ \mathcal{T}.\mathbf{A} \leftarrow g(B)$ in $\mathcal{T}$.*bmappings* **then** add *btmap* to $\mathcal{T}$.*properties*;
return $\mathcal{T}$;

Figure 19: Combining Transformation Pairs

- 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 $\mathcal{T}_k$ as shown in Figure 18(a) by replacing the two transformations with the single transformation $\mathcal{T}' = \mathcal{T}_{k-1} \circ \mathcal{T}_k$ as shown in Figure 18(b), 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. Let us associate with each transformation $\mathcal{T}_k$ all known schema information (input schema $\mathcal{T}_k.\mathbf{A}$, input key $\mathcal{T}_k.A_{key}$, output schema $\mathcal{T}_k.\mathbf{B}$, output key $\mathcal{T}_k.B_{key}$), all known schema mappings (forward mappings $\mathcal{T}_k$.*fmappings* and backward mappings $\mathcal{T}_k$.*bmappings*), whether $\mathcal{T}_k$ is complete ($\mathcal{T}_k$.*complete*), and a set $\mathcal{T}_k$.*properties* of all known properties $\mathcal{T}_k$ satisfies from the hierarchy in Figure 14. Procedure `Combine(`$\mathcal{T}_1, \mathcal{T}_2$`)` in Figure 19 sets these features for combined transformation $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ based on the features for $\mathcal{T}_1$ and $\mathcal{T}_2$. Note from the algorithm that we need all of these features in order to properly determine $\mathcal{T}$.*properties* from $\mathcal{T}_1$ and $\mathcal{T}_2$. However, only the *properties* sets will be important in our final decision of whether to combine transformations.

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

```
procedure Normalize(S = T_1 ∘ ··· ∘ T_n)
  while (k ← BestCombo(S)) ≠ 0
    replace T_k and T_{k+1} in S with T ← Combine(T_k, T_{k+1});
  if S contains black-box transformations then
    j ← lowest index of a black-box in S;
    replace transformations T_j, ..., T_n in S with T ← T_j ∘ ··· ∘ T_n;
```

```
procedure BestCombo(S = T_1 ∘ ··· ∘ T_n)
  k ← 0; N[1..5] ← [0,0,0,0,0];
  for j = 1..n do g ← group(T_j); N[g] ← N[g] + 1; // initialize vector
  for j = 1..n − 1 do
    curN ← N;
    T ← Combine(T_j, T_{j+1});
    g ← group(T);
    if g < 5 then // T is not a black-box
      g_1 ← group(T_j); g_2 ← group(T_{j+1});
      curN[g] ← curN[g] + 1; curN[g_1] ← curN[g_1] − 1; curN[g_2] ← curN[g_2] − 1;
      if curN < N then k ← j; N ← curN; // see text for defn. of curN < N
return k;
```

Figure 20: Normalizing a transformation sequence

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 20. 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 14 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 20, 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

[5]Note that the presence of non-key-map schema mappings (Section 3.2) or indexes (Section 3.6) 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.

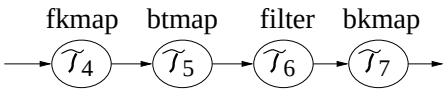


Figure 21: Before normalization

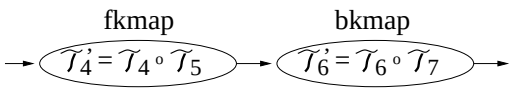


Figure 22: After normalization

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 we will see in Section 7.

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 21 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 in Figure 22. □

5 Lineage Tracing for Transformations with Multiple Input and Output Sets

So far we have addressed the lineage tracing problem for transformations with a single input set and single output set (Section 3), and for linear sequences of such transformations (Section 4). In this section, we extend our approach to handle individual transformations with multiple input and/or multiple output sets. In Section 6, we put everything together to tackle the lineage tracing problem for arbitrary acyclic transformation graphs.

5.1 Multiple-Input Single-Output Transformations

We first consider transformations with multiple input sets but only one output set, which we call *Multiple-Input Single-Output* (*MISO*) transformations. What is most relevant about a transformation $\mathcal{T}$ with multiple input sets is how exactly $\mathcal{T}$ combines its input sets to produce its output. After studying MISO transformations

in practice we determined that although it is possible to define a large number of narrow MISO transformation classes, it makes more sense to define just two broad classes—*exclusive* and *inclusive* MISO transformations. These classes still enable efficient and precise lineage tracing in almost all cases. Furthermore, as will be seen, this approach to MISO transformations exploits our entire framework for single-input transformations, simply adding to it the mechanics for handling transformations that operate on multiple inputs.

5.1.1 Exclusive MISO Transformations

Consider a MISO transformation $\mathcal{T}$ with m input sets $I_1, \ldots, I_m$ and one output set O. Informally, $\mathcal{T}$ is an *exclusive* transformation if it effectively transforms each input set independently and produces as output the union of the results. More formally, $\forall I_1, \ldots, I_m$, $\mathcal{T}(I_1, \ldots, I_m) = \bigcup_{j=1..m} \mathcal{T}(\varnothing, \ldots, I_j, \ldots, \varnothing)$. Note that since we are considering set union, it is still possible for an output item $o \in O$ to be produced by more than one input set.

By the above definition of an exclusive MISO transformation, we can "split" $\mathcal{T}$ into j independent transformations, one for each input. Specifically, for $j = 1..m$ we define a single-input *split transformation* $\mathcal{T}[j](I) = \mathcal{T}(\varnothing, \ldots, \varnothing_{j-1}, I, \varnothing_{j+1}, \ldots, \varnothing)$. Now let $O[j] = \mathcal{T}[j](I_j)$ for $j = 1..m$; that is, $O[j]$ is the portion of the output produced from the jth input. We define the lineage of an output item $o \in O$ according to $\mathcal{T}$ to be the lineage of o according to all split transformations $\mathcal{T}[j]$ where $o \in O[j]$.

Based on these definitions, let us now consider how we enable and perform lineage tracing. Since an exclusive MISO transformation effectively transforms each input independently, we assume that the transformation author can understand and specify the properties of each split transformation $\mathcal{T}[j]$, just as he would do for a single-input transformation. Any of the properties from our hierarchy in Figure 14 can be specified. Thus, to trace the lineage of an output item $o \in O$, we first determine all j's such that $o \in O[j]$ (we may want to compute and store the $O[j]$'s at load time for this purpose), then we trace o through each relevant $\mathcal{T}[j]$ to input I_j based on $\mathcal{T}[j]$'s specified properties, using the algorithms of Section 3.

The most obvious example of an exclusive MISO transformation is set union, for which each split transformation is a filter.

5.1.2 Inclusive MISO Transformations

The other class of MISO transformations—all MISO transformations that are not exclusive—is the *inclusive* MISO transformations. Informally, MISO transformations are inclusive when all of their input sets need to be combined together in some fashion to produce the output set. (A MISO transformation could treat some inputs exclusively and others inclusively, but such transformations are rare in practice and we treat them as purely inclusive.)

For inclusive transformations we also define the notion of *split transformations* to enable the definition of lineage and the tracing process, however the definition of split transformation differs from that for exclusive MISO transformations, and it must be based on each transformation instance. For an inclusive MISO transformation instance $\mathcal{T}(I_1, \ldots, I_m) = O$, the split transformation $\mathcal{T}[j]$ is defined as $\mathcal{T}[j](I) = \mathcal{T}(I_1, \ldots, I_{j-1}, I, I_j, \ldots, I_m)$. That is, $\mathcal{T}[j]$ takes as a parameter the jth input set and treats the other input sets as constants. Note that $\mathcal{T}[j](I_j) = O$ for all $j = 1..m$, so we do not need the concept of $O[j]$. Otherwise, we proceed in the same way as for the exclusive case: The transformation author specifies properties for each

split transformation. To trace the lineage of an output item $o \in O$, we trace its lineage through each split transformation $\mathcal{T}[j]$.

It is easy to see intuitively why standard relational operators such as join, intersection, and difference are inclusive: they need all of their inputs in order to produce their output. As a concrete example, consider transformation $\mathcal{T}_3$ in Section 1.2, which joins the order information in input I_1 and the product information in input I_2. $\mathcal{T}_3$'s split transformation $\mathcal{T}_3[1](I_1)$ effectively takes each order $i \in I_1$, finds (in I_2) the corresponding product information, and attaches it to i to produce an output item. $\mathcal{T}_3[1]$ has a backward schema mapping `order-id` $\overset{\mathcal{T}_3[1]}{\leftarrow}$ `order-id` to the input key, so it is a backward key-map. Similarly, $\mathcal{T}_3[2]$ is a backward key-map. Thus, to trace an output item o through $\mathcal{T}_3$, we find the corresponding order tuple in input I_1 through $\mathcal{T}_3[1]$ using `TraceBM`, and the corresponding product tuple in input I_2 through $\mathcal{T}_3[2]$ also using `TraceBM`.

5.2 Multiple-Input Multiple-Output Transformations

Consider a multiple-input multiple-output transformation $\mathcal{T}$ that takes m input sets $I_1, \ldots, I_m$ and produces n output sets $O_1, \ldots, O_n$. To trace the lineage of an output item $o \in O_k$, we consider a *restriction* $\mathcal{T}^k$ of $\mathcal{T}$ on output O_k such that $\mathcal{T}^k(I_1, \ldots, I_m) = O_k$. That is, $\mathcal{T}^k$ acts like $\mathcal{T}$ but produces only O_k, ignoring the other output sets. We define the lineage of o according to $\mathcal{T}$ as o's lineage according to $\mathcal{T}^k$. Since $\mathcal{T}^k$ is a MISO transformation, we proceed as in Section 5.1. Note that the transformation author must undertand the restricted transformations $\mathcal{T}^k$, but they are usually very straightforward and frequently symmetric.

6 Lineage Tracing Through Transformation Graphs

Finally we consider the most general case: lineage tracing for arbitrary acyclic graphs of transformations. Consider a transformation graph $\mathcal{G}$ with m initial inputs $I_1, \ldots, I_m$ and n outputs $O_1, \ldots, O_n$. Each transformation in $\mathcal{G}$ can have any number of inputs and outputs, and we know from Section 5.2 how to trace through any such transformation. Thus, to trace the lineage of an output item $o \in O_k$, we can trace o through the entire graph backwards, similar to our approach for sequences but possibly needing to follow multiple backward paths through the graph. To enable tracing in this fashion, at loading time we may need to store all intermediate results for each edge of the graph, add indexes as appropriate, and store split transformation outputs for exclusive multiple-input transformations as described in Section 5.1.1.

It is not immediately obvious how to fully generalize our idea of improving lineage tracing performance by combining transformations (Section 4) to the graph case, and doing so in depth is beyond the scope of this paper. However, because our approach to lineage tracing for multiple-input transformations is based on the notion of single-input split transformations (Sections 5.1.1 and 5.1.2), often we can improve overall performance of lineage tracing through a transformation graph by applying techniques developed earlier in this paper—we normalize and trace lineage through each path in the graph independently:

1. When $\mathcal{G}$ is defined, we create a *tracing sequence* $\mathcal{S}$ for each path in $\mathcal{G}$ from an initial input set to a final output set. If a transformation $\mathcal{T}$ on the path has multiple inputs and/or multiple outputs, then in sequence $\mathcal{S}$ we replace $\mathcal{T}$ with its restricted (Section 5.2) and split (Section 5.1) transformation $\mathcal{T}^k[j]$, where the previous transformation in $\mathcal{S}$ provides $\mathcal{T}$'s jth input in $\mathcal{G}$, and the next transformation in $\mathcal{S}$ takes $\mathcal{T}$'s kth output in $\mathcal{G}$. Given our overall approach, the transformation author will already have

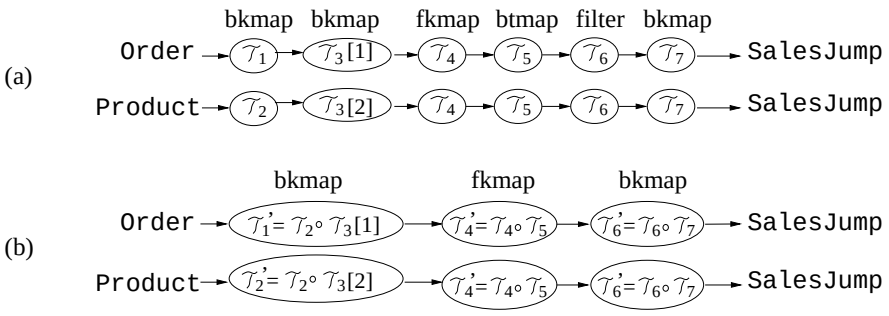


Figure 23: Tracing sequences (unnormalized and normalized) for `SalesJump`

order-id	date	prod-id	quantity	prod-name	price	valid
0101	2/1/1999	222	10	Sony VAIO	2250	12/1/1998–9/30/1999
0379	4/9/1999	222	5	Sony VAIO	2250	12/1/1998–9/30/1999
1028	11/24/1999	222	10	Sony VAIO	1980	10/1/1999–
1250	12/15/1999	222	10	Sony VAIO	1980	10/1/1999–

Figure 24: I_4^*

specified properties for all $\mathcal{T}^k[j]$'s. We then normalize each sequence $\mathcal{S}$ as described in Section 4.2. When finished, we have q normalized tracing sequences $\mathcal{S}_1, \ldots, \mathcal{S}_q$ for q paths in $\mathcal{G}$.

2. When the warehouse data is loaded, intermediate results required for tracing through each normalized tracing sequence $\mathcal{S}_1, \ldots, \mathcal{S}_q$ are saved and indexes are built as desired.

3. When tracing the lineage of output item $o \in O_k$, we simply trace o through each normalized tracing sequence $\mathcal{S}$ that ends in O_k, and combine the results.

Example 6.1 (Lineage Tracing for `SalesJump`) Recall the warehouse table `SalesJump` from Section 1.2, created by transformation graph $\mathcal{G}$ in Figure 3. From $\mathcal{G}$, we create two tracing sequences as shown in Figure 23(a), where $\mathcal{T}_3[1]$ and $\mathcal{T}_3[2]$ are split transformations of $\mathcal{T}_3$. The figure also shows the best property for each transformation in the sequences. We then use our greedy algorithm `Normalize` to combine transformations in the two tracing sequences as described in Section 4.2. Details are omitted here, but we obtain the two normalized tracing sequences shown in Figure 23(b). At loading time, to enable lineage tracing through our two normalized tracing sequences, we only need to save the initial inputs and the intermediate results from transformations $\mathcal{T}_3$ and $\mathcal{T}_5$ in the original graph $\mathcal{G}$. At tracing time, consider as an example output data item $o = \langle$`Sony VAIO`, `11250`, `39600`$\rangle$ in `SalesJump`, which was also used as a motivating example in Section 1. We first trace o's lineage in `Order` through the upper normalized tracing sequence in Figure 23(b). We trace o through transformation $\mathcal{T}_6'$ using `TraceBM` to obtain $I_6^* = \{\langle$`Sony VAIO`, `22500`, `11250`, `0`, `11250`, `39600`$\rangle\}$. We then trace I_6^* through $\mathcal{T}_4'$ using `TraceFM` to obtain I_4^* as shown in Figure 24. Finally, we trace I_4^* through $\mathcal{T}_1'$ using `TraceBM` to obtain the `Order` tuples shown in Figure 5. We also trace o's lineage in `Product` through the lower normalized tracing sequence in Figure 23(b). Details are omitted, but the final result is the `Product` tuples shown in Figure 5. □

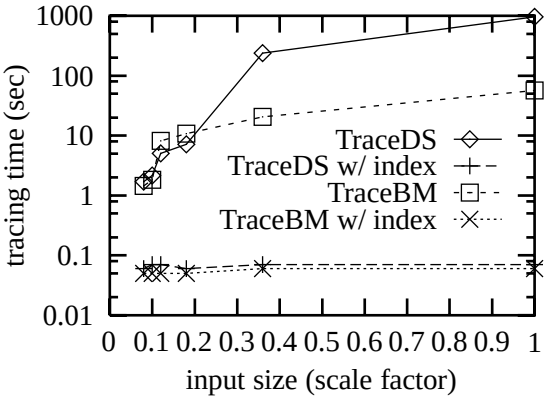


Figure 25: Tracing one transformation

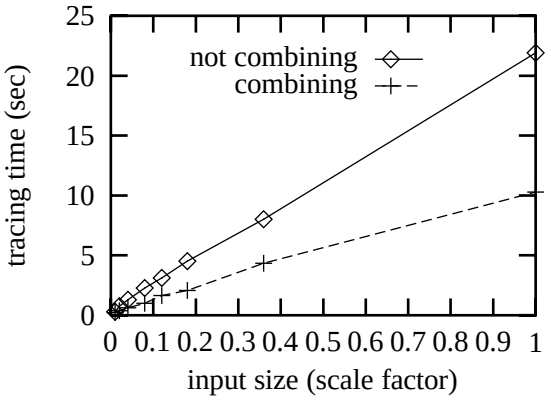


Figure 26: Combining vs. not combining

7 Performance Experiments

We have implemented all of the algorithms described in this paper in a prototype data lineage tracing system for general warehouse transformations. For convenience we decided to use a standard commercial relational DBMS for storing all data (input sets, output sets, and intermediate results), and we implemented our lineage tracing procedures as well as our test suite of data transformations as parameterized stored procedures.

In this paper we provide a few preliminary performance results. Specifically, we consider the following three questions for a few representative cases:

1. Roughly how fast are the lineage tracing procedures?
2. How much speedup can we obtain using indexes?
3. How much faster can we trace through transformation sequences when we combine transformations as in Section 4?

The data we used for our experiments is based on the TPC-D benchmark [TPC96], specifically our input tables are `LineItem`, `Order`, and `PartSupp` from the benchmark. Contents of the tables were generated by the standard `dbgen` program supplied with the benchmark. Note that a TPC-D scale factor of 1.0 means that the entire warehouse is about 1GB in size. Our experiments were conducted on a dedicated Windows NT machine with a Pentium II processor.

7.1 Tracing Performance

In the first experiment, we consider a simple relational transformation $\mathcal{T}$ which could be defined in SQL as follows:

```
SELECT PartKey, SuppKey, AvailQty*SupplyCost FROM PartSupp;
```

This transformation is a dispatcher, and also is a backward key-map with a schema mapping from the output key attributes `PartKey` and `SuppKey` to the same input attributes, so we can trace its data lineage using either `TraceDS` or `TraceBM`. We vary the input table scale factor from 0.08 to 1, and we measure lineage tracing time for a single tuple using `TraceDS` and `TraceBM` with and without the indexes described in Section 3.6. From the results shown in Figure 25, we see that when indexes are not used procedure `TraceBM` is significantly faster than `TraceDS`. (Note the log scale on the y axis.) With indexes both tracing procedures are very fast.

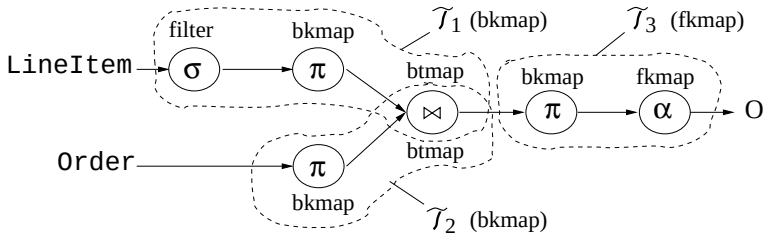


Figure 27: Transformation graph for Q12 in TPC-D

7.2 Combining versus Not Combining

Our second experiment studies the benefit of combining transformations using the techniques introduced in Section 4. We consider the transformation graph $\mathcal{G}$ in Figure 27, which is based on query Q12 from the TPC-D benchmark. The graph takes as input tables `LineItem` and `Order`, and produces as output the number of high and low priority orders on a selected set of line items for each shipment mode. (Recall that α represents grouping and aggregation.) Given this transformation graph, in one experiment we trace lineage one transformation at a time (for two paths), based on the individual transformation properties marked in Figure 27. In the other experiment, we first normalize the two sequences, which results in the combined transformations $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ shown by the dashed regions in Figure 27. We then trace lineage through the combined transformations on both paths. We vary the input scale factor from 0.01 to 1 and obtain the results shown in Figure 26. We see that as the input size grows, combining transformations provides a significant reduction in tracing time.

8 Conclusions and Future Work

We presented a complete set of techniques for data warehouse lineage tracing when the warehouse data is loaded through a graph of general transformations. Our approach relies on a variety of *transformation properties* that hold frequently in practice, and that can be specified easily by transformation authors. We presented techniques for improving lineage tracing performance, including building indexes and combining transformations for the purpose of lineage tracing. All algorithms presented in this paper have been implemented in a prototype lineage tracing system and preliminary performance results are reported.

The results in this paper can be used to develop principles for creating transformations and transformation graphs that are amenable to lineage tracing. As a first step, transformation authors can be sure to specify the most restrictive properties that a transformation satisfies based on our property hierarchy (Figure 14), to ensure that the most efficient tracing procedure is selected. Second, a transformation might be modified slightly to improve its properties, for example retaining key values so that a dispatcher becomes a backward key-map. Third, in many cases complex black-box transformations can be avoided by splitting the transformation into simpler transformations with better properties. In general, for the purposes of lineage tracing it is better to specify smaller atomic transformations rather than larger ones, since the lineage tracing system will combine transformations automatically anyway when it is beneficial to do so.

There are several avenues for future work:

- As discussed in Section 4.2, we use a simple cost metric and greedy algorithm for normalizing trans-

formation sequences. Although we already obtain good performance improvements (Section 7), clearly it would be interesting to develop a more detailed and accurate cost model and more sophisticated algorithms for exploring the space of transformation combinations. More generally, in the case of transformation graphs, we might benefit from normalizing the graph globally, rather than normalizing each path independently. Even more generally, in the presence of commutative transformations (which are not common but do occur), we might consider reorganizing a transformation sequence or graph to obtain a better normalized result.

- In this paper we have assumed that properties of a transformation are provided to the lineage tracing system, either by the transformation author or because the transformation is a prepackaged component with known properties. A separate line of research is that of inferring a transformation's properties, either by examining the specification (e.g., using program analysis techniques over the code), or by running sample data through the transformation and examining the results.
- In this paper we have assumed that most of the work for lineage tracing should be done at tracing time. That is, we don't want to expend considerable extra computation or storage cost during the loading process just for lineage tracing. The other extreme is the *annotation* approach, discussed in Sections 1.1 and 3.6, where considerable additional information is computed and stored at loading time to speed up lineage tracing [Cui01]. The decision of which extreme to take is dependent on the expected tracing workload, and on any performance requirements for loading or lineage tracing. It might be interesting to explore middle-ground approaches, which compute and store some amount of additional information for lineage tracing, but without incurring undue performance degradation at loading time.

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, Special Issue on Data Transformations*, 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. Run-time translation of view tuple deletions using data lineage. Technical report, Stanford University Database Group, June 2001.

[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 Corporation: 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, Bombay, 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 7.0, Data Transformation Services. http://msdn.microsoft.com/library/psdk/sql/dts_ovrw.htm.

[Ora] Oracle 8i. http://technet.oracle.com/products/oracle8i/.

[Pow] Cognos: PowerPlay OLAP Analysis 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, San Jose, 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, San Jose, California, May 1975.

[TPC96] Transaction Processing Performance Council. *TPC-D Benchmark Specification, Version 1.2*, 1996. http://www.tpc.org/.

[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.

A Proofs

A.1 Proof for Theorem 3.2

Given a key-preserving aggregator $\mathcal{T}$, we want to prove that $\mathcal{T}$ is context-free, i.e, $\forall i$ and i', they either always belong to the same input partition or they always do not. Suppose for the sake of a contradiction that $\mathcal{T}$ is not context-free. Then there exist input sets I and I' and items i and i' in both I and I' such that

1. i and i' belong to the same input partition I_j in transformation instance $\mathcal{T}(I) = O$. Let $\mathcal{T}(I_j) = o_j$. According to the definition of key-preserving aggregator, $\mathcal{T}(\{i\})$ and $\mathcal{T}(\{i'\})$ produce output items with the same key value as $o_j.key$.

2. i and i' belong to different input partitions I_j and I_k respectively in $\mathcal{T}(I') = O'$. Let $\mathcal{T}(I_j) = o_j$ and $\mathcal{T}(I_k) = o_k$. According to the definition of key-preserving aggregator, $\mathcal{T}(\{i\})$ produces an item with key value $o_k.key$, while $\mathcal{T}(\{i'\})$ produces an item with key value $o_j.key$.

Since $o_j.key \neq o_k.key$ by the definition of a key, we obtain our contradiction. □

A.2 Proof for Theorem 3.5

Consider a dispatcher or aggregator $\mathcal{T}$ and an instance $\mathcal{T}(I) = O$. We prove that if $\mathcal{T}$ has a forward schema mapping $f(A) \xrightarrow{\mathcal{T}} B$, the lineage of any output item $o \in O$ is a subset of $\{i \in I \mid f(i.A) = o.B\}$. The case for $\mathcal{T}$ with backward schema mapping $A \xleftarrow{\mathcal{T}} g(B)$ can be proved in the same manner.

If $\mathcal{T}$ is a dispatcher, o's lineage is $\{i\}$ such that $o \in \mathcal{T}(\{i\})$, according to the lineage definition for dispatchers in Section 3.1. Thus, we just need to prove $f(i.A) = o.B$. Consider transformation instance $\mathcal{T}(\{i\}) = O'$. By Definition 3.3 of schema mappings, $m = 1$ because $\{i\}$ contains a single item. Furthermore, $n \geq 1$ because O' is nonempty. Since $m \geq n$, we have $m = n = 1$. Thus, according to Definition 3.3, $f(i.A) = o.B$.

Now suppose $\mathcal{T}$ is an aggregator. From Definition 3.3 we can partition instance $\mathcal{T}(I) = O$ according to the schema mapping to obtain input and output partitions $I_1, \ldots, I_m$ and $O_1, \ldots, O_n$, and since aggregators are complete we know $m = n$. For $k = 1..n$, the pair of partitions I_k and O_k generate a transformation instance $\mathcal{T}(I_k) = O_k = \{o_k^1, \ldots, o_k^{l_k}\}$. From the definition of aggregator, there exists a unique partition $I_k^1 \ldots, I_k^{l_k}$ of I_k such that $\mathcal{T}(I_k^j) = \{o_k^j\}$ for $j = 1..l_k$, and $f(i.A) = o_k^j.B$ for all $i \in I_k^j$. Considering the entire input set I and $\mathcal{T}(I) = O = \{o_1^1, \ldots, o_n^{l_n}\}$, $I_1^1, \ldots, I_n^{l_n}$ is a partition of I such that $\mathcal{T}(I_k^j) = \{o_k^j\}$, so it is the unique such partition. According to the lineage definition for aggregators in Section 3.1.2, given an output data item $o = o_k^j \in O$, o_k^j's lineage is I_k^j. Since $f(i.A) = o_k^j.B$ for all $i \in I_k^j$, and $o = o_j^k$, we conclude $I_k^j \subseteq \{i \in I \mid f(i.A) = o.B\}$. □

A.3 Proof for Theorem 3.7

(1) Given a filter $\mathcal{T}$, each input item either produces itself or nothing, i.e., $\forall i \in I$: $\mathcal{T}(\{i\}) = \{i\}$ or $\varnothing$. Thus, I and O always have the same schema as each other, call it $\mathbf{A}$, and we can partition any input I and output $O = \mathcal{T}(I)$ into singleton sets $I_1, \ldots, I_m$ and $O_1, \ldots, O_n$ such that $m \geq n$ and:

   (a) for $k = 1..n$, $\mathcal{T}(I_k) = O_k$, and I_k and O_k both contain the same single item, so $I_k = \{i \in I \mid i = o$ for any $o \in O_k\}$.

(b) for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

Therefore, we have a backward schema mapping $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} \mathbf{A}$ to all input attributes, so $\mathcal{T}$ is a backward total-map.

(2) Consider a backward total-map $\mathcal{T}$ where the input has schema $\mathbf{A}$, so we have $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$. Since $\mathbf{A}$ is an input key (we are assuming set semantics), $\mathcal{T}$ is a backward key-map.

(3) Let $\mathcal{T}$ be a backward key-map with $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$. We want to prove that $\mathcal{T}$ is a dispatcher, i.e., $\forall I$: $\mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$. Given any instance $\mathcal{T}(I) = O$, according to Definition 3.3 of schema mapping, we can partition I into $I_1, \ldots, I_m$ based on A_{key} equality and partition O into $O_1, \ldots, O_n$ based on $g(B)$ equality such that $\mathcal{T}(I_k) = O_k$ for $k = 1..n$ and $\mathcal{T}(I_k) = \varnothing$ for $k = (n+1)..m$. Thus, $\mathcal{T}(I) = \bigcup_{k=1..n} O_k = \bigcup_{k=1..m} \mathcal{T}(I_k)$. Since each I_k contains items with the same A_{key} value, it contains a single item by the definition of key. Therefore, $\mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$, and $\mathcal{T}$ is a dispatcher.

(4) Let $\mathcal{T}$ be a forward key-map with schema mapping $f(A) \overset{\mathcal{T}}{\rightarrow} B_{key}$. We want to prove that $\mathcal{T}$ is a key-preserving aggregator. We first prove that $\mathcal{T}$ is an aggregator, i.e., $\mathcal{T}$ is complete and given any instance $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$, there exists a unique partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$. By Definition 3.6 of forward key-map, $\mathcal{T}$ is complete. Consider any $\mathcal{T}(I) = O$. From Definition 3.3 of schema mapping we obtain input and output partitions $I_1, \ldots, I_m$ and $O_1, \ldots, O_n$, where $m = n$ since $\mathcal{T}$ is complete. We will prove that $I_1, \ldots, I_n$ is the unique input partition for $\mathcal{T}(I) = O$ as specified in the aggregator definition (Section 3.1.2). Since each O_k, $k = 1..n$, contains items with the same B_{key} value, it contains a single item; let it be o_k. Then, $I_1, \ldots, I_n$ is an input partition such that $\mathcal{T}(I_k) = \{o_k\}$, and by Definition 3.3 $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$. We now prove that this input partition is unique. Suppose there exists another input partition $I'_1, \ldots, I'_n$ such that $\mathcal{T}(I'_k) = \{o_k\}$ for $k = 1..n$. Consider an instance $\mathcal{T}(I'_k) = \{o_k\}$. Since the output contains a single item, and since $\mathcal{T}$ is complete, when we partition this instance based on the schema mapping we obtain a single input partition and a single output partition. From Definition 3.3, we know that $\forall i \in I'_k$: $f(i.A) = o_k.B_{key}$. Thus, $I'_k \subseteq I_k$, for $k = 1..n$. Since $I_1 \cup \cdots \cup I_n = I'_1 \cup \cdots \cup I'_n = I$ and the partitions are disjoint, we know that $I_k = I'_k$ for $k = 1..n$. Therefore, $I_1, \ldots, I_n$ is unique, and $\mathcal{T}$ is an aggregator.

We now prove that $\mathcal{T}$ is key-preserving. Consider instance $\mathcal{T}(I) = O$ with aggregator partition $I_1, \ldots, I_n$ for output $\{o_1, \ldots, o_n\}$. $\forall I'_k \subseteq I_k$: $\mathcal{T}(I'_k) = \{o'_k\}$ where $o'_k.B_{key} = o_k.B_{key}$, $k = 1..n$. Consider any $k = 1..n$ and any $I'_k \subseteq I_k$ and let $O'_k = \mathcal{T}(I'_k)$. Since all items in I'_k have the same $f(i.A)$, according to Definition 3.3, we can partition this instance based on the schema mapping to obtain a single input partition and a single output partition. Thus, O'_k contains items with the same $o.B_{key}$, which means O'_k contains a single item; let it be o'_k. Then, $\forall i \in I'_k$: $f(i.A) = o'_k.B_{key}$. Since $I'_k \subseteq I_k$, $\forall i \in I'_k$: $i \in I_k$. Further because $\forall i \in I_k$: $f(i.A) = o_k.B_{key}$, we know that $o'_k.B_{key} = o_k.B_{key}$. Therefore, $\mathcal{T}$ is key-preserving. □

A.4 Proof for Theorem 3.8

1. Let $\mathcal{T}$ be a forward key-map with schema mapping $f(A) \overset{\mathcal{T}}{\rightarrow} B_{key}$. Consider an instance $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$. We want to prove that $\forall o_k$, o_k's lineage is $\{i \in I \mid f(i.A) = o_k.B_{key}\}$. According to

Theorem 3.7, $\mathcal{T}$ is an aggregator, and from part (4) of our proof in Appendix A.3, we know that partition $I_1, \ldots, I_n$ where $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$ is the unique input partition such that $\mathcal{T}(I_k) = \{o_k\}$. According to the lineage definition for aggregators, o_k's lineage is $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$.

2. Let $\mathcal{T}$ be a backward key-map with schema mapping $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$. Consider an instance $\mathcal{T}(I) = O$. We want to prove that $\forall o \in O$, o's lineage is $\{i \in I \mid i.A_{key} = g(o.B)\}$. According to Theorem 3.7, $\mathcal{T}$ is a dispatcher. Thus, according to the lineage definition for dispatchers, o's lineage is $I^* = \{i \in I \mid o \in \mathcal{T}(\{i\})\}$. According to Theorem 3.5, I^* is a subset of $I' = \{i \in I \mid i.A_{key} = g(o.B)\}$. Since I' contains a single item by the definition of key, I^* is either empty or contains a single item. We will prove that I^* is nonempty by contradiction, from which we can conclude that $I^* = I' = \{i \in I \mid i.A_{key} = g(o.B)\}$. Suppose that I^* is empty. Then, $\forall i \in I$, $o \notin \mathcal{T}(\{i\})$. Therefore, since $\mathcal{T}$ is a dispatcher according to Theorem 3.7, $o \notin \bigcup_{i \in I} \mathcal{T}(\{i\}) = \mathcal{T}(I)$, which contradicts $o \in O$.

3. Let $\mathcal{T}$ be a backward total-map with schema mapping $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$ to the entire input schema $\mathbf{A}$. We want to prove that $\forall I$ and $\forall o \in O = \mathcal{T}(I)$, o's lineage is $\{g(o.B)\}$. Since a backward total-map is a backward key-map, by part(2) of this proof we know that the lineage of o is $I^* = \{i \in I \mid i.A = g(o.B)\}$. Since $i.A = i$ for all $i \in I$, $I^* = \{g(o.B)\}$. □

A.5 Proof for Theorem 3.11

Consider an aggregator $\mathcal{T}$ with inverse $\mathcal{T}^{-1}$. We want to prove that given any instance $\mathcal{T}(I) = O$ and $o \in O$, o's lineage is $\mathcal{T}^{-1}(\{o\})$. Since $\mathcal{T}$ is an aggregator, according to the definition of aggregator in Section 3.1.2, for $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$ there exists a unique partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$ for $k = 1..n$, and I_k is o_k's lineage. According to Definition 3.10 $\mathcal{T}^{-1}(\mathcal{T}(I)) = I$. Therefore, $\forall o_k \in O$, $\mathcal{T}^{-1}(\{o_k\}) = \mathcal{T}^{-1}(\mathcal{T}(I_k)) = I_k$, which is o_k's lineage. □