

# Data Lineage: A Survey

Robert Ikeda and Jennifer Widom

Stanford University

{rmikeda,widom}@cs.stanford.edu

## 1. INTRODUCTION

*Lineage*, or *provenance*, in its most general form describes where data came from, how it was derived, and how it was updated over time. Information management systems today exploit lineage in tasks ranging from data verification in curated databases [1] to confidence computation in probabilistic databases [10, 12]. Here, we formalize and categorize lineage, discuss a set of selected papers, and then identify open problems in lineage research.

Lineage can be useful in a variety of settings. For example, molecular biology databases, which mostly store copied data, can use lineage to verify the copied data by tracking the original sources [1]. Data warehouses can use the lineage of anomalous view data to identify faulty base data [4], and probabilistic databases can exploit lineage for confidence computation [10, 12].

Although lineage can be very valuable for applications, storing and querying lineage can be expensive; for instance the Gene Ontology database has up to 10MB of lineage for single tuples [10]. Approaches have been developed to lower these costs. For example, in probabilistic databases, *approximate lineage* can compress the complete lineage by up to two orders of magnitude while allowing a selected set of queries over the lineage to be answered efficiently with low error [10]. In curated databases, storing a type of lineage called *hierarchical transactional provenance* can reduce the storage overhead by a factor of 5, relative to a more naive approach [1].

In addition to challenges related to space and time efficiency, it can be difficult even to define lineage in domains that allow arbitrary transformations. The most commonly considered transformations are relational queries [4, 9, 10, 12], but some papers have studied lineage for a broad range of transformations. For example, in data warehousing, tracing procedures have been developed for general transformations that take advantage of transformation properties specified by the transformation definer [3]. In curated databases, lineage for copying operations has been studied [1]. For a generalization of standard relational queries called *DQL* (Deterministic QL), definitions for lineage that are invariant under query rewriting have been proposed [2].

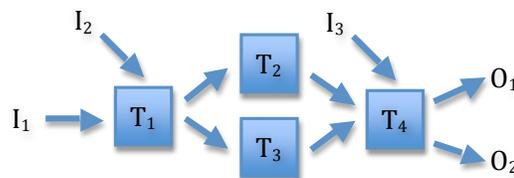


Figure 1: Example transformation graph.

## 2. PROBLEM STATEMENT AND EXAMPLES

We ground our discussion of lineage with a concrete problem statement. Input *data sets*  $I_1, \dots, I_k$  are fed into a graph of *transformations*  $T_1, \dots, T_n$  to produce output data sets  $O_1, \dots, O_m$ . An example *transformation graph* is shown in Figure 1. Typically, we assume that the transformations form a directed acyclic graph and that the input and output data sets consist of individual *items*.

Given a transformation graph, we would like to ask the following questions:

- Q1) Given some output, which inputs did the output come from?
- Q2) Given some output, how were the inputs manipulated to produce the output?

These questions delineate two *types* of lineage: *where-lineage* (Q1) and *how-lineage* (Q2). Each type of lineage has two *granularities*:

- 1) *Schema-level (coarse-grained)*
- 2) *Instance-level (fine-grained)*

Schema-level where-lineage answers questions such as which data sets were used to produce a given output data set, while schema-level how-lineage answers questions such as which transformations were used to produce a given output data set.

In contrast, instance-level lineage treats individual items within a data set separately, so we ask more fine-grained questions such as which tuples from a set of base tables are responsible for the existence of a given tuple in a derived table (where-lineage).

We now introduce two running examples to make the types and granularities of lineage in our problem statement more concrete.

**Relational Example.** Let us consider a relational database with two tables: **Sales**(autoID, date, price) and **Autos**(autoID, color). We are interested in the sales information for blue autos. To get this information, we can first perform a join  $\bowtie$  on our **Sales** and **Autos** tables, and then perform a selection  $\sigma$  on color. Figure 2 shows the transformation graph to produce our desired output. (We could also have performed the selection  $\sigma$  on **Autos** before the join.)

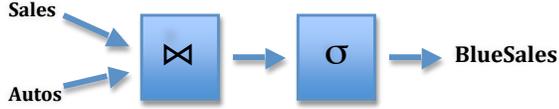


Figure 2: Auto sales relational example.

In this example, the **Sales** and **Autos** tables are the input data sets, and the  $\bowtie$  and  $\sigma$  operators are the transformations. Our output data set is the resulting table, **BlueSales**. Given the entire table **BlueSales**, we can ask for its schema-level where-lineage and how-lineage. The where-lineage of **BlueSales** consists of the **Sales** and **Autos** tables, and the how-lineage consists of the  $\bowtie$  and  $\sigma$  operators.

Given an output tuple  $o$  in **BlueSales**, we can ask for its instance-level lineage. The where-lineage of  $o$  consists of the tuple from **Sales** and the tuple from **Autos** whose join corresponds to  $o$ , and the how-lineage again consists of the  $\bowtie$  and  $\sigma$  operators. With how-lineage, we see no difference between schema-level and instance-level. However, we will see a difference in the next example.

**Deduplication Example.** Let us consider a mailing-list company that has multiple sources of mailing addresses and a lookup table for zip-plus-four codes. From the address input sets and the zip+4 lookup table, the company would like to produce an output set of deduplicated, canonicalized addresses. The desired output is generated by first feeding the input sets of addresses  $A_1$ ,  $A_2$ , and  $A_3$  into data cleaning transformations **Clean<sub>1</sub>**, **Clean<sub>2</sub>**, and **Clean<sub>3</sub>**, respectively. We assume for this example that addresses from  $A_1$  already have zip+4 codes, while addresses from  $A_2$  and  $A_3$  do not. We feed the intermediate outputs from **Clean<sub>2</sub>** and **Clean<sub>3</sub>** along with the zip+4 lookup table into the canonicalization transformation **Canon**. Then, addresses from both **Clean<sub>1</sub>** and **Canon** are fed into the deduplication transformation **Dedup** to produce the final list **Addresses**. Figure 3 shows the corresponding transformation graph.

In this example, the original address sets and the zip+4 lookup table are the input data sets. **Clean<sub>1</sub>**, **Clean<sub>2</sub>**, **Clean<sub>3</sub>**, **Canon**, and **Dedup** are the transformations. Our output data set is **Addresses**. Our “items” here are either individual addresses or entries in the zip+4 lookup table. Given the entire list **Ad-**

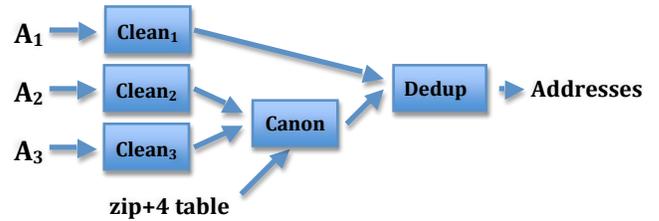


Figure 3: Address deduplication example.

**resses**, we can ask for its schema-level where-lineage and how-lineage. The where-lineage of **Addresses** consists of  $A_1$ ,  $A_2$ ,  $A_3$ , and the zip+4 table. The how-lineage of **Addresses** consists of the transformations **Clean<sub>1</sub>**, **Clean<sub>2</sub>**, **Clean<sub>3</sub>**, **Canon**, and **Dedup**.

Given an output address  $o$  in **Addresses**, we can ask for its instance-level lineage. The where-lineage of  $o$  consists of the subset of input addresses from  $A_1$ ,  $A_2$ , and/or  $A_3$  along with any entries from the zip+4 table that were used to produce an intermediate result leading to the deduplicated address  $o$ .

For how-lineage, suppose that  $o$  was produced using only addresses from **Clean<sub>1</sub>**. In other words, no addresses from **Canon** were used by **Dedup** to produce  $o$ . Then the how-lineage of  $o$  would consist only of the transformation **Clean<sub>1</sub>** followed by **Dedup**. This example illustrates a difference between schema-level and instance-level how-lineage, since the how-lineage of  $o$  consisting only of **Clean<sub>1</sub>** and **Dedup** is more specific than the how-lineage of the entire list **Addresses**.

### 3. CATEGORIES

Having formalized the lineage problem to some extent, we now identify characteristics that will help us categorize our selected papers.

**Lineage type and granularity.** As discussed in Section 2, the two main characteristics of lineage are type (where and how) and granularity (schema and instance). Figure 4 classifies the papers we will discuss based on these characteristics. Many of the papers discussed here focus on instance-level where-lineage. Note that some papers occur in more than one quadrant. For example, the paper on scientific workflow lineage [8] covers both where and how schema-level lineage. Systems that focus on schema-level lineage [8] are typically targeted for cases where the transformation graph is large and complex.

**Transformation type and lineage queries.** Another defining characteristic of lineage is the type of transformations that it can track, ranging from copying operations [1] to SPJU queries [4, 9, 10, 12] to arbitrary “black boxes” [3]. The transformations tracked by lineage are closely related to the types of lineage queries that are typically performed. For example, since transformations sometimes leave certain inputs untouched

	Schema	Instance
Where	Workflow [8]	Why and Where [2] General Warehouse [3] Warehouse View [4] Non-Answers [9] Approximate [10] Trio [12]
How	Curated [1] Workflow [8]	Curated [1]

**Figure 4: Papers categorized by lineage type and granularity.**

in [1], the how-lineage of an output tuple  $o$  may be a subset of the sequence of transformations that resulted in the output table that contains  $o$ . Thus, in [1] instance-level how-lineage queries are typically performed. In [4] the structure of relational operators in transformations determines which tuples in the base tables are responsible for the existence of a given tuple in the derived table. Thus, in [4] instance-level where-lineage queries are typically performed. Users of systems such as [4] that track relational operators would not typically perform instance-level how-lineage queries, since for relational operators, the instance-level how-lineage of a tuple  $o$  in output table  $\mathbf{O}$  is the same as the schema-level how-lineage of the entire table  $\mathbf{O}$ .

**Eager vs. lazy.** For the common case of instance-level where-lineage, systems can be classified as either *eager* or *lazy*. Eager lineage systems store instance-level where-lineage information immediately after performing transformations. In contrast, lazy lineage systems [3, 4] instead store schema-level how-lineage immediately after transformations. The schema-level how-lineage is sufficiently detailed so that it can then be traced to produce the instance-level where-lineage when requested. There is a tradeoff between eager and lazy systems. Eager systems enable faster answers to lineage queries at the price of extra storage space and preprocessing time. In contrast, lazy systems avoid these extra costs and are thus appropriate in settings where requests for instance-level where-lineage are expensive or relatively infrequent.

## 4. SELECTED PAPERS

We now discuss the contributions of eight selected papers.

### 4.1 Efficient Lineage Tracking for Scientific Workflows [8]

The setting of this paper is *scientific workflows*. In scientific applications, workflows arrange the execution of tasks by defining the control and data flow between those tasks. The system for tracking scientific workflows described in this paper stores a variation of our transformation graph called a *workflow graph*. A work-

flow graph is a directed acyclic graph that represents the schema-level where-lineage and how-lineage of scientific workflows. In a workflow graph, nodes represent *tasks* (transformations) or *data sets*, and the edges represent *dependencies*. A directed edge points from a task to a data set if the data set is an output of the task, and from a data set to a task if the data set is an input to the task. The primary difference between a workflow graph and a transformation graph is that a workflow graph has extra nodes to represent the intermediate data sets.

- **Transformation types:** Unspecified - arbitrary “tasks”.
- **Lineage type:** Where-lineage and how-lineage.
- **Lineage granularity:** Schema-level. No instance-level information is supported in the system.
- **Lineage queries:** This system supports asking for predecessors in the workflow graph, which corresponds to asking which input or intermediate data sets (where-lineage) or tasks (how-lineage) were used to derive a given data set.

The main technical challenge is how to represent workflow graphs so that these lineage queries can be answered efficiently. The main contribution of this paper is a space- and time-efficient method of transforming the workflow graphs into trees that have the same predecessor relationships. These trees can then be encoded using previous work called *interval tree encoding*. Having used this method to preprocess the workflow graph, one can efficiently find the tasks (how-lineage) or data sets (where-lineage) that constitute the schema-level lineage of a given data set.

### 4.2 Approximate Lineage for Probabilistic Databases [10]

The setting of this paper is *probabilistic relational databases*. In a probabilistic database, each tuple within a base table has a probability of being present. Base tuple probabilities are independent; these probabilities are represented by *atoms*. Relational queries can be performed over probabilistic tables to create derived tables. Within a derived table, tuples do not have independent probabilities, so the probabilities are instead represented using Boolean formulas called *lineage formulas* over atoms. In this paper, the derived tables are such that all formulas are of the form *n-monotone DNF*, meaning a disjunction of monomials each containing at most  $n$  atoms and no negations.

- **Transformation types:** The transformations supported by this system are relational queries used to produce derived tables.
- **Lineage type:** Where-lineage only.
- **Lineage granularity:** Instance-level only.
- **Lineage queries:** The lineage queries supported are based on *explanations* and *influential atoms*. For a given derived tuple, an explanation is a minimal

conjunction of atoms whose truth implies the truth of its lineage formula. The *influence* of an atom with respect to a derived tuple’s lineage formula is the probability that flipping (changing from true to false or vice-versa) the atom would flip the lineage formula. Given a derived tuple  $t$ , the two lineage questions one can ask are: (1) find the  $k$  explanations with the highest probability, and (2) find the  $k$  most influential atoms of its lineage formula.

- **Eager vs. lazy:** Primarily eager. After a relational query is performed, each tuple in the derived table is annotated with its lineage formula. Each lineage formula represents the instance-level where-lineage of its tuple. Since the lineage formulas consist of atoms, the instance-level where-lineage always points directly to the base data as opposed to only one level down given multiple levels of derived tables.

The main technical challenge is to compress the lineage formulas into a representation that can be used to answer the above queries accurately and efficiently.

This paper explores two techniques for compressing the lineage formulas: *sufficient lineage* and *polynomial lineage*. Sufficient lineage replaces the original lineage formula by a simpler and smaller Boolean formula. Polynomial lineage replaces the original formula by a Fourier series. Both sufficient and polynomial lineage can be guaranteed to create  $\varepsilon$ -approximations (defined below), and both can be eagerly stored during the preprocessing stage, allowing efficient queries for sufficient explanations and influential atoms. Sufficient lineage tends to give more accurate explanations while polynomial lineage can provide higher compression ratios.

Finally we define an  $\varepsilon$ -approximation. Averaged over all *possible worlds*, a lineage formula has an expected value between 0 and 1; this value represents the probability that the derived tuple exists. Given a lineage formula  $\lambda$ , we say that  $\lambda'$  is an  $\varepsilon$ -approximation of  $\lambda$  if the expected value of the squared difference between  $\lambda'$  and  $\lambda$  is bounded by  $\varepsilon$ :  $\mathbf{E}[(\lambda' - \lambda)^2] \leq \varepsilon$ .

### 4.3 Provenance Management in Curated Databases [1]

The setting of this paper is *curated databases*: databases constructed by scientists who manually assimilate information from several sources. One of the characteristics of curated databases is that much of their content has been derived or copied from other sources, often other curated databases. The system described in this paper maps data sources to an XML view using *wrappers*. Lineage is tracked based on the XML view.

- **Transformation types:** Basic transformations are *operations*: insert, copy, and delete. Operations may also be grouped into *transactions* (defined below), which then become our transformations.
- **Lineage type:** How-lineage. The transformation

graph has no input data sets, so only how-lineage is supported. The graph is simply a single path of transformation nodes leading to the output node, which represents the current state of the database. Note that insert operations can be thought of as corresponding to input data.

- **Lineage granularity:** All granularities. Because the data model is XML, schema-level vs. instance-level is not relevant.
- **Lineage queries:** The queries supported are *Src* (What transformation first created the data located here?), *Hist* (What sequence of transformations copied a node to its current position?), and *Mod* (What transformations created or modified the subtree under the node?).

The main technical challenge is to minimize the overhead required to store the lineage information. Lineage queries are assumed to be relatively rare, so query performance is not a major factor. *Transactional provenance* and *hierarchical provenance* are two methods proposed for reducing the amount of storage space needed for lineage information. Transactional provenance groups multiple operations into *transactions*, which become our transformations. For each transaction, only the net changes are tracked by the stored lineage information. Hierarchical provenance exploits transformations that can be summarized. For example, if all tuples in a table are copied, hierarchical provenance will store these transformations at the schema level as opposed to at the instance level. Combining transactional and hierarchical provenance, in a method called *hierarchical transactional provenance*, can reduce the storage overhead by a factor of 5, relative to the naive approach where neither optimization is used.

### 4.4 Trio: A System for Data, Uncertainty, and Lineage [12]

The setting of this paper is *Uncertainty-Lineage Databases (ULDBs)*. ULDBs extend the standard relational model to capture various types of uncertainty along with data lineage, and the *TriQL* query language extends SQL with a new semantics for uncertain data and new constructs for querying uncertainty and lineage. In Trio, query results depend on lineage to their input tables, and results are often stored as derived tables. Trio is implemented using a translation-based layer on top of a conventional relational DBMS.

- **Transformation types:** The transformations supported by this system are the relational queries used to produce derived results.
- **Lineage type:** Where-lineage.
- **Lineage granularity:** Instance-level.
- **Lineage queries:** For querying lineage, TriQL includes a built-in predicate  $\text{Lineage}(T_1, T_2)$  designed to be used as a join condition in a TriQL query. Us-

ing this predicate, we can find pairs of tuples  $t_1$  and  $t_2$  from ULDB tables  $T_1$  and  $T_2$  such that  $t_1$ 's lineage directly or transitively includes  $t_2$ .

- **Eager vs. lazy:** Eager. After a relational query is performed, each tuple in the derived result is annotated with its lineage formula, representing its instance-level where-lineage. In contrast to [10], the instance-level where-lineage points one level down, even when there are multiple levels of derived tables.

Lineage is critical in the Trio system in many ways. Without lineage, the ULDB model is not expressive enough to represent the correct results of all queries: lineage is necessary to constrain the set of possible worlds. Lineage is also useful for *confidence* computation. In the ULDB model, base data can have confidences (probabilities). Computing confidences on query results is a hard problem. In contrast to previous work, Trio by default does not compute confidence values during query evaluation. Instead, Trio performs on-demand confidence computation, which can be more efficient and is enabled by lineage. Related to confidence computation, lineage allows *extraneous data* (data in the query result that does not exist in any possible world) to be detected for removal. Because lineage in Trio points only one level down, the system needs to recursively “unfold” the lineage to base data for confidence computation or extraneous data removal. Since this unfolding can be expensive, some optimizations and shortcuts have been developed.

## 4.5 Lineage Tracing for General Warehouse Transformations [3]

The setting of this paper is *data warehousing systems*: systems that integrate information from operational data sources into a central repository to enable analysis and mining of the integrated information. Sometimes during data analysis it is useful to look not only at the information in the warehouse, but also to investigate how certain warehouse items were derived from the sources. Generally, data imported from the sources is cleansed, integrated, and summarized through a transformation graph. Since the transformations may vary from simple algebraic operations or aggregations to complex procedural code, this paper considers data warehouses created by general transformations.

- **Transformation types:** General transformations.
- **Lineage type:** Where-lineage.
- **Lineage granularity:** Instance-level.
- **Lineage queries:** Given a transformation graph and an output item  $o$ , find  $o$ 's instance-level where-lineage.
- **Eager vs. lazy:** Lazy. This system stores the transformation details (schema-level how-lineage) and the transformation graph. The transformation graph is then traced to produce the instance-level where-

lineage of a given output item  $o$  when requested.

The main technical challenge is to provide as specific lineage as possible in a setting of general transformations. Whenever possible, transformation *properties* are specified in advance so that the system can exploit them for lineage tracing.

A first property is the *transformation class*. There are three main transformation classes: *dispatchers*, *aggregators*, and *black-boxes*. Depending on the transformation class, an appropriate tracing procedure can be selected. A transformation  $T$  is a dispatcher if each input item produces zero or more output items independently. If  $T$  is an aggregator, there is a unique disjoint partitioning of the input set such that each partition produces one of the output items. There are also special cases of dispatchers and aggregators for which even more specific tracing procedures can be selected. Black-boxes are neither dispatchers nor aggregators.

Another property is whether a transformation  $T$  has a schema mapping. When  $T$  has a schema mapping, its mapping can be used to improve the tracing procedure for certain types of transformations. Sometimes a transformation may come with a tracing procedure or inverse transformation. These tracing procedures should normally be used if given.

Given a transformation graph, the default method of tracing through multiple transformations is a backwards, step-by-step algorithm that considers one transformation at a time. However, it may be beneficial to combine transformations to improve tracing performance. A greedy algorithm is presented to decide when to combine transformations.

## 4.6 Tracing the Lineage of View Data in a Warehousing Environment [4]

This paper considers a similar data warehousing problem to the general warehouse transformation paper [3]. However, here the transformations are assumed to produce *relational materialized views*, which are defined, computed, and stored in the warehouse to answer queries about the source data in an integrated and efficient way. Tracing algorithms are presented that exploit the fixed set of operators and the algebraic properties of relational views.

- **Transformation types:** Relational views.
- **Lineage type:** Where-lineage.
- **Lineage granularity:** Instance-level.
- **Lineage queries:** Given a transformation graph and an output item  $o$ , find  $o$ 's instance-level where-lineage.
- **Eager vs. lazy:** Lazy. This system stores the details of the relational views (schema-level how-lineage) along with the transformation graph. The transformation graph is then used to produce the instance-level where-lineage of a given output item  $o$  when requested.

The main technical challenge is to trace the transformation graph of relational views correctly. The main contribution of this paper is a set of lineage tracing algorithms for relational views along with proofs of correctness. Given a *Select-Project-Join view*, the basic tracing algorithm first transforms the view into a canonical form. A single tracing query can then take the canonical form of the view to systematically compute the instance-level where-lineage of a given output item  $o$ . Optimizations such as pushing selection operators below joins can improve performance. Given a transformation graph of relational views and an output item  $o$ , tracing queries are called recursively to compute  $o$ 's instance-level where-lineage backwards through the transformation graph.

#### 4.7 On the Provenance of Non-Answers to Queries over Extracted Data [9]

The setting of this paper is *information extraction*. In information extraction, it is useful to provide users querying extracted data with explanations for the answers they receive. However, in some cases explaining why an expected answer is not in the result may be just as helpful. Although this paper was motivated by information extraction, it is relevant to any setting in which relational queries are issued and explanations for *non-answers* are desired. A non-answer to a query is a tuple that fits the schema of the query result but is not present in the result. The *lineage of a non-answer* consists of those tuples whose presence in the base tables through insertions or updates would have made the non-answer an answer.

- **Transformation types:** Relational queries.
- **Lineage type:** Where-lineage.
- **Lineage granularity:** Instance-level.
- **Lineage queries:** Given a relational query and a non-answer  $o$ , find  $o$ 's non-answer lineage.
- **Eager vs. lazy:** Lazy. The lineage of a given non-answer is not generated until requested.

Given a relational query and a non-answer  $o$ , often a large or infinite number of insertions and updates can make  $o$  an answer. The main technical challenge is to restrict the size of non-answer lineage. (Since this paper only considers Select-Project-Join expressions with conjunctive predicates, only insertions and updates, not deletions, to the base data can lead to non-answers turning into answers.) To avoid blowup, the definition of non-answer lineage is restricted to include only those base tuples that are “likely to exist”. Likelihood is related to *trust*; some tables and attributes are assumed to be correct, while others could possibly have errors.

#### 4.8 Why and Where: A Characterization of Data Provenance [2]

The setting of this paper is a generalization of relational queries called *DQL (Deterministic QL)*. In the

DQL data model, the *location* of any piece of data can be uniquely described by a *path*. This model uses a variation of existing edge-labeled tree models for semistructured data. The focus of this paper is defining different types of provenance over the DQL data model.

- **Transformation types:** DQL queries.
- **Lineage type:** Where-lineage. (Further explanation is given below.)
- **Lineage granularity:** Instance-level.
- **Lineage queries:** Given a DQL query and an output item  $o$ , find  $o$ 's *why-provenance* and *where-provenance* (informally defined below).

The main technical challenge is to give formal definitions of two different types of lineage, both of which are invariant under query rewriting. The two types of lineage discussed are *why-provenance* and *where-provenance*. Informally, given an output item  $o$ , why-provenance captures the input items that contributed to  $o$ 's derivation. Thus, why-provenance is what we have called instance-level where-lineage. Where-provenance is even more fine-grained, capturing which pieces of input data actually appear in the output. For example, if we perform a projection on a input relational table, given an output tuple  $o$ , those attribute values of the input tuple that contributed to  $o$  are considered to be part of  $o$ 's where-provenance.

### 5. OPEN PROBLEMS

We now outline several areas for future work.

#### 5.1 Lineage-Supported Detection and Correction of Data Errors

It is an open problem to build a general “workbench” for data analysis that uses lineage to detect and correct data errors. To illustrate the functionality we would like our workbench to have, let us extend the auto sales relational database example from Figure 2. Recall that to get the sales information for blue autos, we first perform a join  $\bowtie$  on our **Sales**(autoID, date, price) and **Autos**(autoID, color) tables, and then perform a selection  $\sigma$  on color to produce table **BlueSales**. Let us then perform an aggregation  $\alpha$  to sum the sales grouped by date, producing output table **SumBlue**. The extended transformation graph is shown in Figure 5.



Figure 5: Sum of sales example.

Suppose we see that output  $o \in \mathbf{SumBlue}$  for January 1 is \$100 even though we know we sold a blue Sky-Bird for \$500 on this date. We would like to discover why  $o$  is wrong.

There are two possible sources of errors, both of which can be detected using  $o$ 's lineage:

1. *Incorrect input data.* We may determine that one of the base tuples in  $o$ 's instance-level where-lineage has incorrect data. For example, tuple  $s \in \mathbf{Sales}$  may have the price of the January 1 SkyBird sale recorded as \$50 instead of \$500. If  $s$  is the source of  $o$ 's error, we can correct  $s \in \mathbf{Sales}$  and then recalculate  $o$  to see that the total sales for January 1 was actually \$550 as opposed to \$100.
2. *Wrong transformations.* Tuple  $o$  can still be wrong even if all base tuples in  $o$ 's instance-level where-lineage are correct. Suppose the color of the SkyBird is recorded in  $\mathbf{Autos}$  as 'dark blue' instead of 'blue'. Then if  $\sigma$  selects on "color='blue'" instead of on "color like '%blue%'", it will incorrectly filter out the SkyBird sale from  $\mathbf{BlueSales}$ , and hence, from contributing to  $o \in \mathbf{SumBlue}$ . To correct  $o$  in this situation, we would first replace  $\sigma$  with  $\sigma'$  ( $\sigma'$  having correct filter "color like '%blue%'"), and then recalculate  $\mathbf{BlueSales}$  and  $\mathbf{SumBlue}$ .

**Challenges.** We now identify challenges associated with creating a general workbench.

- *Finding sources of errors.* As we saw in the example from Figure 5, where-lineage is useful for finding incorrect input data while how-lineage is useful for finding wrong transformations. A workbench that enables the user to query over both where- and how-lineage would facilitate finding the sources of errors.
- *Efficiently propagating corrections forwards.* After correcting either incorrect input data or wrong transformations, we would like to efficiently propagate the corrections forwards through the transformation graph. Following the correction of input data, we would like to rerun the transformations incrementally to avoid recalculating old results that were unaffected by the corrections. Following the correction of a transformation, we would like to run the corrected transformation, but only on the input data that could potentially produce different results. There has been a large amount of work on *incremental view maintenance*: the efficient propagation of modifications of base data in a relational setting [7]. However, even in a relational setting, we are unaware of work that studies how to rerun corrected transformations incrementally. For general transformations, rerunning transformations incrementally needs to be studied for both corrected base data and corrected transformations.
- *Propagating corrections backwards.* Besides correcting the sources of errors directly, another approach to handling incorrect output data is to give the expected output values and then propagate these corrections backwards to the base data. This approach is related to both the relational view update problem [6] and the lineage paper on non-answers [9].

Propagating corrections backwards is challenging, especially with general transformations, because given a correction to an output item, there could be many ways of altering the base data or transformations to generate the correct output. To select among the many possible ways of propagating the corrections backwards, user intervention will probably be needed.

- *Fixing intermediate errors.* In our transformation graph, we may find that one of our intermediate data sets contains an error. Given this error, we would like to correct the error and have these corrections propagate both forwards and backwards through our transformation graph.
- *Semi-automatically detecting data errors.* We anticipate that data errors will typically be detected by users as they are examining the output data. Given a few manual corrections, it may be possible to use semi-automatic methods to further clean the data. One such method might inspect the lineage of the manually corrected data items to find patterns in the lineage that are associated with errors. Having found these lineage patterns, data items with similar lineage can be flagged as questionable.

As an example of semi-automatic error detection, revisiting  $\mathbf{SumBlue}$ , suppose we see that the tuples for February 3, February 11, and February 14 all indicate abnormally high total sales. The correction system may detect that all three of these tuples in  $\mathbf{SumBlue}$  have instance-level where-lineage pointing to an expensive FireBird car in  $\mathbf{Autos}$ . Mistakenly,  $\mathbf{Autos}$  has the color of the FireBird recorded as blue instead of red, its actual color. Having found this error in the FireBird's color information, we could then flag any output tuples in the transformation graph that depended on the FireBird tuple in  $\mathbf{Autos}$  as questionable.

## 5.2 Tradeoffs

- *Eager-lazy hybrid.* Recall from Section 3 the distinction between eager and lazy lineage. In contrast to a lazy system, an eager system uses extra storage space and preprocessing time to enable faster answers to lineage queries. The storage cost of instance-level where-lineage in a fully eager system may be too high in certain situations [11]. Even so, it might make sense to do some preprocessing for faster answers to lineage queries. An intermediate solution could be used as a compromise between an eager and a lazy system.

As an example, let us revisit  $\mathbf{SumBlue}$ . Suppose our aggregation  $\alpha$ , in addition to computing total sales by date, also computed total sales by month. ( $\alpha$  here is a "cube-like" operator, but unlike CUBE,  $\alpha$  does not store a copy of tuples from  $\mathbf{BlueSales}$ .) For each tuple  $o \in \mathbf{SumBlue}$ , an eager system would

record, together with  $o$ , pointers to all tuples from **BlueSales** that contributed to  $o$ . A lazy system would record no extra information, and would instead, when asked for  $o$ 's instance-level where-lineage, run a tracing algorithm to create it, such as those in [3, 4]. A hybrid system would record extra information, but less than an eager system would record, to aid the retrieval of  $o$ 's instance-level where-lineage.

For example, our hybrid system might record, for each daily tuple  $o_d \in \mathbf{SumBlue}$ , pointers to all tuples from **BlueSales** that contributed to  $o_d$  (same as eager). For each monthly tuple  $o_m \in \mathbf{SumBlue}$ , the hybrid system might record pointers to all daily tuples  $o_d \in \mathbf{SumBlue}$  in  $o_m$ 's month. To retrieve  $o_m$ 's instance-level where-lineage, the hybrid system would first follow  $o_m$ 's lineage pointers to its daily tuples  $o_d$ , then follow  $o_d$ 's lineage pointers to find the tuples  $t \in \mathbf{BlueSales}$  that belong to  $o_m$ 's instance-level where-lineage. Since this hybrid system has to follow an extra level of pointers, it takes longer than an eager system to find  $o_m$ 's instance-level where-lineage, but the amount of lineage information it has to store for  $o_m$  is significantly smaller.

The hybrid example described here takes advantage of the structure of our “cube-like” aggregation. However, creating hybrid systems for general relational queries and transformations is a challenging problem.

### 5.3 “Richer” Lineage Queries

- *Querying updates together with lineage.* “Update lineage” connects newer versions of modified data to older versions [5]. When we have conventional lineage together with update lineage, we can ask queries like: “Find all output tuples whose instance-level where-lineage contains a tuple whose value at some point was equal to a given value”. The paper on curated databases [1] studied update lineage, but it focused on reducing storage costs as opposed to enabling efficient lineage queries. In addition, the paper on curated databases only considered a limited set of transformations, not relational queries or general transformations.
- *Pointing to old versions of data.* Reference [5] considers lineage for the case where all corrections to base data are immediately propagated to output data. In a lineage system where corrections to base data are not immediately (or perhaps never) propagated, the lineage of output data can point to old versions of base data. In such a system, a user may want to ask for all *stale data* (output data that points to old base data), or request that all stale data be recomputed using new base data. We do not know of any work that fully supports stale data and its lineage.

- *Unifying relational queries and general transformations.* Most systems focus on either relational queries or general transformations. However, there are situations in which a system that supports both would be useful. For example, consider an information extraction system that first used extractors to retrieve structured data from web pages and then used relational queries to query the structured data. Suppose we then wanted to find the web pages that led to the creation of an output tuple. To support this lineage query, a system would have to track both relational queries and transformations. For eager systems, the key difference between relational queries and transformations is that the relational structure of queries allows the system to automatically create instance-level where-lineage [10, 12]. In contrast, general transformations often require the user to specify how the lineage should be created. In lazy systems, the structure of relational queries [4] makes the tracing problem easier than it is for general transformations [3]. Supporting lineage for both relational queries and general transformations in a unified system is an open problem.
- *Combining where and how.* If we had a system that supports both where- and how-lineage, we could issue “mixed” lineage queries, such as which input tuples used by a particular transformation type led to the creation of a given output tuple. The only paper we surveyed that supported both where- and how-lineage [8] supported schema-level lineage only. The only lineage information this system stored was the workflow graph, describing input and output relationships between data sets and transformations. Creating a query language and an efficient implementation for a system that supports both where- and how-lineage at the instance level is an open problem.

## 6. REFERENCES

- [1] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD*, 2006.
- [2] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.
- [3] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, 2001.
- [4] Y. Cui, J. Widom, and J. Weiner. Tracing the lineage of view data in a warehousing environment. In *TODS*, 2000.
- [5] A. Das Sarma, M. Theobald, and J. Widom. Data modifications and versioning in Trio. Technical report, 2008.
- [6] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *VLDB*, 1978.
- [7] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and

- applications. In *IEEE Data Engineering Bulletin*, 18(2), 1995.
- [8] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD*, 2008.
- [9] J. Huang, T. Chen, A. Doan, and J. Naughton. On the provenance of non-answers to queries over extracted data. In *VLDB*, 2008.
- [10] C. Re and D. Suciu. Approximate lineage for probabilistic databases. In *VLDB*, 2008.
- [11] M. Stonebraker, J. Becla, D. DeWitt, K. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for science data bases and SciDB. In *CIDR*, 2009.
- [12] J. Widom. Trio: A system for data, uncertainty, and lineage. To be in *Managing and Mining Uncertain Data*.