# Provenance and Data Synchronization

J. Nathan Foster
University of Pennsylvania
jnfoster@cis.upenn.edu

Grigoris Karvounarakis
University of Pennsylvania
gkarvoun@cis.upenn.edu

## 1 Introduction

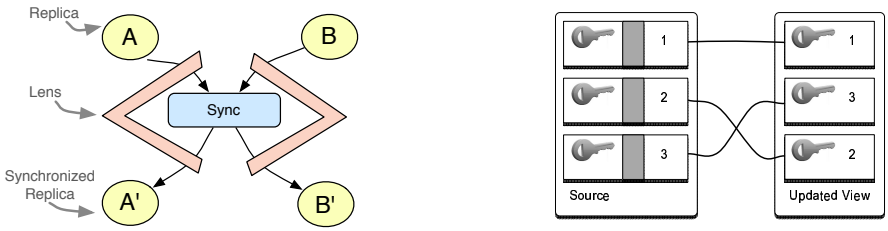
Replication increases the availability of data in mobile and distributed systems. For example, if we copy calendar data from a web service onto a mobile device, the calendar can be accessed even when the network cannot. In peer-based data sharing systems, maintaining a copy of the shared data on a local node enables query answering when remote peers are offline, guarantees privacy, and improves performance. But along with these advantages, replication brings complications: whenever one replica is updated, the others also need to be refreshed to keep the whole system consistent. Therefore, in systems built on replication, synchronization mechanisms are critical.

In simple applications, the replicas are just that—carbon copies of each other. But often the copied data needs to be transformed in different ways on each replica. For example, web services and mobile devices represent calendars in different formats (iCal vs. Palm Datebook). Likewise, in data sharing systems for scientific data, the peers usually have heterogeneous schemas. In these more complicated systems, the replicas behave like views, and so mechanisms for updating and maintaining views are also important.

The mapping between sources and views defined by a query is not generally one-to-one. This loss of information is what makes view update and view maintenance difficult. It has often been observed that *provenance*—i.e., metadata that tracks the origins of values as they flow through a query—could be used to cope with this loss of information and help with these problems [5, 6, 4, 24], but only a few existing systems (e.g., AutoMed [12]) use provenance in this way, and only for limited classes of views.

This article presents a pair of case studies illustrating how provenance can be incorporated into systems for handling replicated data. The first describes how provenance is used in *lenses* for ordered data [2]. Lenses define updatable views, which are used to handle heterogeneous replicas in the Harmony synchronization framework [23, 13]. They track a simple, implicit form of provenance and use it to express the complex update policies needed to correctly handle ordered data. The second case study describes ORCHESTRA [17, 19], a collaborative data sharing system [22]. In ORCHESTRA, data is distributed across tables located on many different peers, and the relationship between connected peers is specified using GLAV [16] schema mappings. Every node coalesces data from remote peers and uses its own copy of the data to answer queries over the distributed dataset. Provenance is used to perform incremental maintenance of each peer as updates are applied to remote peers, and to filter "incoming" updates according to *trust conditions*.


**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**




(a) (b)

Figure 1: (a) Synchronization architecture for heterogeneous replicas. (b) Correspondence induced by keys.

## 2 Lenses

A *lens* is a bidirectional program. When read from left to right it denotes an ordinary function that maps sources to views. When read from right to left, the same lens denotes an "update translator" that takes a source together with an updated view and produces a new source that reflects the update.

In the context of data synchronization, lenses are used to bridge the gap between heterogeneous replicas. To synchronize two replicas represented in different formats, we first define lenses that transform each source format into a common "abstract" format, and then synchronize the abstract views. For example, to synchronize iCal and Palm Datebook calendars, we use the forward direction of two lenses to transform the files into abstract calendars, discarding the low-level formatting details and any other data specific to each replica. After synchronization, we then propagate the changes induced by the synchronizer back to the original formats using the reverse direction of the same lenses. The architecture of a synchronizer for heterogeneous data assembled in this way is depicted in Figure 1(a).

Semantically, a lens $l$ is just a pair of functions, which we call *get* and *put*. The *get* component maps sources to views. It may, in general, discard some of the information from the source while computing the view. The *put* component therefore takes as arguments not only an updated view, but also the original source; it weaves the data from the view together with the information from the source that was discarded by the *get* component, and yields an updated source. (Note that lenses are agnostic to how the view update is expressed—the *put* function works on the entire state of the updated view.)

The two components of a lens are required to fit together in a reasonable way: the *put* function must restore all of the information discarded by *get* when the view update is a no-op, and the *put* function must propagate all of the information in the view back to the updated source (see [14] for a comparison of these requirements to classical conditions on view update translators in the literature.) In a lens language, these requirements are guaranteed by the type system; in implementations, they are checked automatically [14, 15, 3, 2].

### 2.1 Ordered Data

Recent work on lenses has focused on the special challenges that arise when the source and view are ordered [2]. The main issue is that since the update to the view can involve a reordering, accurately reflecting updates back to source requires locating, for each piece of the view, the corresponding piece of the source that contains the information discarded by *get*. Our solution to this problem is to enrich lenses with a simple mechanism for tracking provenance: programmers describe how to divide the source into *chunks* and generate a *key* for each chunk. These induce an association between pieces of the source and view that is used by *put* during the translation of updates—i.e., the *put* function aligns each piece of the view with a chunk that has the same key.

To illustrate the problem and our solution, let us consider a simple example from the string domain. Suppose that the source is a newline-separated list of records, each with three comma-separated fields representing the name, dates, and nationality of a classical composer, and the view contains just names and nationalities:

```
"Jean Sibelius, 1865-1957, Finnish          "Jean Sibelius, Finnish
 Aaron Copland, 1910-1990, American    get    Aaron Copland, American
 Benjamin Britten, 1913-1976, English"  ⟶    Benjamin Britten, English"
```

Here is a lens that implements this transformation:

```
let ALPHA = [A-Za-z ]+
let YEARS = [0-9]{4} . "-" . [0-9]{4}
let comp = copy ALPHA . copy ", "
         . del YEARS . del ", "
         . copy ALPHA
let comps = copy "" | comp . (copy "\n" . comp)*
```

The first two lines define regular expressions describing alphabetical data and year ranges using standard POSIX notation for character sets (`[A-Za-z ]` and `[0-9]`) and repetition (`+` and `{4}`). Single composers are processed by `comp`; lists of composers are processed by `comps`. In the *get* direction, these lenses can be read as string transducers, written in regular expression style: `copy ALPHA` matches `ALPHA` in the source and copies it to the view, and `copy ", "` matches and copies a literal comma-space, while `del YEARS` matches `YEARS` in the source but adds nothing to the view. The union (`|`), concatenation (`.`), and iteration (`*`) operators work as usual. The *get* of `comps` either matches and copies an empty string or processes a each composer in a newline-separated list using `comp`. (For formal definitions see [2].)

The *put* component of `comps` restores the dates to each entry positionally: the name and nationality from the $n$th line in the abstract structure are combined with the years from the $n$th line in the concrete structure (using a default year range to handle cases where the view has more lines than the source.) For some simple updates this policy does a good job. For example, suppose that the update changes Britten's nationality, and adds a new composer to the end of the list. The *put* function combines

```
"Jean Sibelius, Finnish
 Aaron Copland, American
 Benjamin Britten, British
 Alexandre Tansman, Polish"
```

with

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, English
 Benjamin Britten, 1913-1976, English"
```

and yields an updated source

```
"Jean Sibelius, 1865-1957, Finnish
 Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, British
 Alexandre Tansman, 0000-0000, Polish"
```

(The year range `0000-0000` is the default; it is generated from the regular expression `YEARS`.) On other examples, however, the behavior of this *put* function is highly unsatisfactory. For example, suppose instead that the update to the abstract string swaps the order of the second and third lines. Then the *put* function takes the following view (and the same source as above)

```
"Jean Sibelius, Finnish
 Benjamin Britten, English
 Aaron Copland, American"
```

and yields

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1910-1990, English
 Aaron Copland, 1913-1976, American"
```

where the year data has been taken from the entry for Copland and inserted into into Britten's, and vice versa! What we want, of course, is for the *put* to align the entries in the concrete and abstract strings by *matching* lines with identical name components, as depicted in Figure 1(b). On the same inputs, this *put* function yields

```
"Jean Sibelius, 1865-1957, Finnish
 Benjamin Britten, 1913-1976, English
 Aaron Copland, 1910-1990, American"
```

where the year ranges are correctly restored to each composer.

## 2.2 Provenance for Chunks

To achieve this behavior, the composers lens needs to be able to keep track of the association between lines in the source and view even when the update involves a reordering—i.e., it need to track *provenance*.

One way to do this would be using explicit *provenance tokens*. On this approach, each line of the source would be annotated with a unique identifier, and the *get* function would propagate these annotations from source to view. The disadvantage of this approach is that the view is no longer an ordinary string, but a string with annotations. This means that applications that take views as input, such as the data synchronizer described above, need to operate on annotated structures, which can be cumbersome.

Lenses use a simpler mechanism that eliminates the need to handle annotated structures. The set of lenses is enhanced with two new primitives for specifying the *chunks* of the source and a *key* for each chunk, and *put* functions are retooled to work on structures where the source is organized as a dictionary of chunks indexed by key, rather than the strings themselves. We call these *dictionary lenses*. Here is a dictionary lens that has the desired behavior for the composers example:

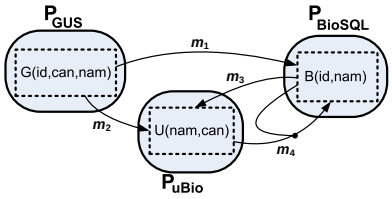
```
let comp = key ALPHA . copy ", "
         . del (YEARS . ", ")
         . copy ALPHA
let comps = "" | <comp> . ("\n" . <comp>)*
```

Compared to the previous version, the two occurrences of `comp` are marked with angle brackets, indicating that these subexpressions are the reorderable chunks, and the first `copy` at the beginning of `comp` has been replaced by the special primitive `key`. The lens `key ALPHA` copies strings just like `copy ALPHA`, but also specifies that the matched substring is to be used as the key of the chunk in which it appears—i.e., in this case, that the key of each composer's entry is their name.

The association induced by keys approximates the association that would be obtained using explicit provenance tokens. Indeed, when the keys are unique and when the view update does not modify the names, the two coincide. The idea of using keys to guide view update is not new: similar approaches have been studied in the relational setting [**?**]. However note that the "keys" used in dictionary lenses are not required to be keys in the strict database sense. When several pieces of the view have the same key, the *put* function pulls chunks out of the dictionary in the order that they originally appeared in the source. This gives the option of obtaining other useful update policies via the choice of key. For example, if a *put* function that operates by position is desired, it can be programmed as a lens whose key component returns a constant.

Another way to control the update policy embodied in a dictionary lens is via the definition of chunks. Many examples can be processed using one level of chunking, as in the composer lens. But chunks may also be nested, which has the effect of stratifying matching into levels: top-level chunks are matched globally across the entire string, subchunks are aligned locally within each chunk, and so on. This is useful in cases where the source has nested structure—e.g., it is used in a lens for LaTeX sources.

We have used dictionary lenses to build lenses for a variety of textual formats including vCard, CSV, and XML address books, iCal and ASCII calendars, BibTeX and RIS bibliographic databases, LaTeX documents, iTunes libraries, and protein sequence data represented in the SwissProt format and XML. These examples demonstrate that a simple notion of implicit provenance formulated using keys is capable of expressing many useful update policies. Current work is focused on an extension to key matching that uses "fuzzy" metrics such as edit distance to align chunks. This relaxed form of matching is useful when processing data with no clear key such as documents, and for handling cases where the update changes a key. We are also studying primitives that incorporate explicit metadata (e.g., source string locations) into the keys, and on developing dictionary lenses for richer structures such as trees and graphs.




$$
\begin{aligned}
m_1: &\quad G(i,c,n) \rightarrow B(i,n)\\
m_2: &\quad G(i,c,n) \rightarrow U(n,c)\\
m_3: &\quad B(i,n) \rightarrow \exists c\, U(n,c)\\
m_4: &\quad B(i,c) \wedge U(n,c) \rightarrow B(i,n)
\end{aligned}
$$

Figure 2: Example collaborative data sharing system for bioinformatics sources. For simplicity, each peer ($P_{GUS}, P_{BioSQL}, P_{uBio}$) has one relation. Schema mappings, given at the right, are indicated by labeled arcs.

# 3 ORCHESTRA

ORCHESTRA is a *collaborative data sharing system* (abbreviated CDSS) [22], i.e., a system for data sharing among heterogeneous peers related by a network of schema mappings. Each peer has a locally controlled and edited database instance, but wants to ask queries over related data from other peers as well. To achieve this, every peer's updates are translated and propagated along the mappings to the other peers. However, this *update exchange* is filtered by *trust conditions*, expressing what data and sources a peer judges to be authoritative, which may cause a peer to reject another's updates. In order to support such filtering, updates carry *provenance* information. ORCHESTRA targets scientific data sharing, but it can also be used for other applications with similar requirements and characteristics.

Figure 2 illustrates an example bioinformatics CDSS, based on a real application and databases of interest to affiliates of the Penn Center for Bioinformatics. GUS, the Genomics Unified Schema, contains gene expression, protein, and taxon (organism) information; BioSQL, affiliated with the BioPerl project, contains very similar concepts; and a third schema, uBio, establishes synonyms and canonical names for taxa. Instances of these databases contain taxon information that is autonomously maintained but of mutual interest to the others. Suppose that a BioSQL peer, $P_{BioSQL}$, wants to import data from peer $P_{GUS}$, as shown by the arc labeled $m_1$, but the converse is not true. Similarly, peer $P_{uBio}$ wants to import data from $P_{GUS}$, along arc $m_2$. Additionally, $P_{BioSQL}$ and $P_{uBio}$ agree to mutually share some of their data: e.g., $P_{uBio}$ imports taxon synonyms from $P_{BioSQL}$ (via $m_3$) and $P_{BioSQL}$ uses transitivity to infer new entries in its database, via mapping $m_4$. Finally, each peer may have a certain *trust policy* about what data it wishes to incorporate: e.g., $P_{BioSQL}$ may only trust data from $P_{uBio}$ if it was derived from $P_{GUS}$ entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent peers' administrators.

The arcs between peers are sets of *tuple-generating dependencies* (tgds). Tgds are a popular means of specifying constraints and mappings [11, 10] in data sharing, and they are equivalent to so-called *global-local-as-view* or *GLAV* mappings [16, 21]. Some examples are shown in the right part of Figure 2. For instance, $m_1$ says that, if there is a tuple in $G$ about an organism with id $i$, canonical name $c$ and name $n$, then an entry $(i, n)$ should be inserted in $B$. Another mapping, $m_4$, ensures that, if there is an entry in $B$ associating id $i$ with a name $c$, and - according to $U$ - $n$ is a synonym of $c$, then there is also an entry $(i, n)$ in $B$. Observe that $m_3$ has an existential variable. For such mappings, update exchange, also involves inventing new "placeholder" values, called *labeled nulls*. Figure 3(a) illustrates update exchange on our running example: assuming that the peers have the local updates shown on the top, (where '$+$' signifies insertion), the update translation constructs the instances shown on the bottom (where $c_1, c_2, c_3$ are labeled nulls).

## 3.1 Using Provenance for Trust Policies

In addition to schema mappings, which specify the relationships between data elements in different instances, a CDSS supports *trust policies*. These express, for each peer $P$, what data from update translation should be trusted and hence accepted. Some possible trust conditions in our CDSS example are:

- Peer $P_{BioSQL}$ distrusts any tuple $B(i, n)$ if the data came from $P_{GUS}$, and trusts any tuple from $P_{uBio}$.

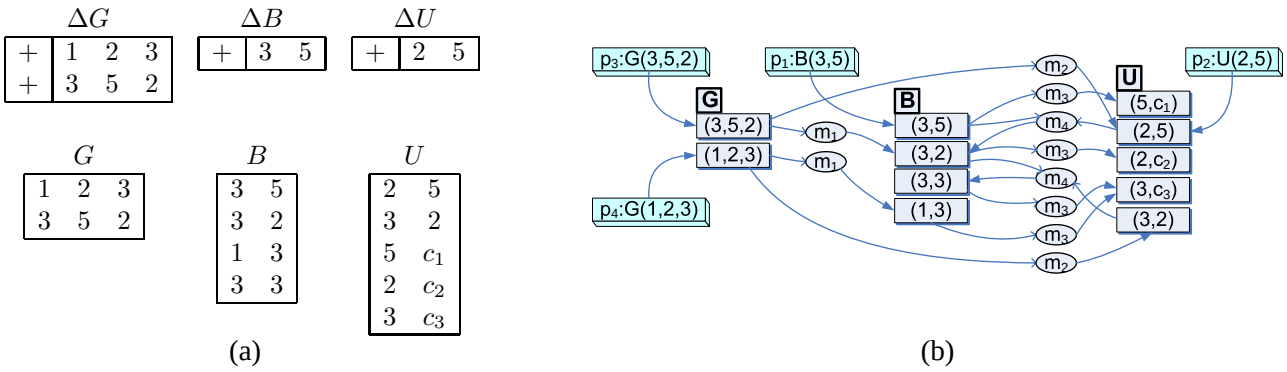



Figure 3: Example of update exchange and resulting provenance graph

- Peer $P_{BioSQL}$ distrusts any tuple $B(i, n)$ that came from mapping $(m_4)$ if $n \neq 2$.

Since the trust conditions refer to other peers and to the schema mappings, the CDSS needs a precise description of how these peers and mappings have contributed to a given tuple produced by update translation, i.e., *data provenance*. Trust conditions need a more detailed provenance model than why-provenance [6] and lineage [9, 1], as explained in [17]. Informally, we need to know not just from which tuples a tuple is derived, but also *how* it is derived, including separate alternative derivations through different mappings.

Figure 3(b) illustrates the main features of our provenance model with a graphical representation of the provenance of tuples in our running example (a more formal description can be found in [17, 18]). The graph has two kinds of nodes: tuple nodes (rectangles), and mapping nodes (ellipses). Arcs connect tuple nodes to mappings that apply to them, and mapping nodes to tuples they produce. In addition, we have nodes for the insertions from the local databases. This "source" data is annotated with its own id (unique in the system) $p_1, p_2, \ldots$ etc. (called a *provenance token*), and is connected by an arc to the corresponding tuple entered in the local instance.

Note that, when the mappings form cycles, it is possible for a tuple to have infinitely many derivations, as well as for the derivations to be arbitrarily large; nonetheless, this graph is a finite representation of such provenance. From the graph we can analyze the provenance of, say, $B(3, 2)$ by tracing back paths to source data nodes — in this case through $(m_4)$ to $p_1$ and $p_2$ and through $(m_1)$ to $p_3$. This way, we can detect when the derivation of a tuple is "tainted" by a peer or by a mapping, i.e., if all its derivations involve them, or not, if there are alternative derivations from trusted tuples and mappings. For example, distrusting $p_2$ and $m_1$ leads to rejecting $B(3, 2)$ but distrusting $p_1$ and $p_2$ does not.

### 3.2 Using Provenance for Incremental Update Exchange

One of the major motivating factors in our choice of provenance formalisms has been the ability to *incrementally maintain* both the data instances at every peer and the provenance associated with the data. Similarly to the case of trust conditions, the provenance model of ORCHESTRA is detailed enough for incremental maintenance, while *lineage* [9, 1] and *why-provenance* [6] are not, intuitively because they don't identify alternative derivations of tuples. We represent the provenance graph *together* with the data instances, using additional relations (see [17] for details). Schema mappings are then translated to a set of datalog-like rules (the main difference from standard datalog being that $Skolem$ functions are used to invent new values for the labeled nulls). As a result, incremental maintenance of peer instances is closely related to incremental maintenance of recursive datalog views, and some techniques from that area can be used. Following [20] we convert each mapping rule (after the relational encoding of provenance) into a series of *delta rules*.

For the case of incremental insertion, the algorithm is simple and analogous to the incremental view maintenance algorithms of [20]. Incremental deletion is more complex: when a tuple is deleted, we need to decide whether other tuples that were derived from it need to be deleted; this is the case if and only if these derived tuples have no alternative derivations from base tuples. Here, ORCHESTRA's provenance model is useful in order to identify tuples that have no derivations and need to be deleted. A small complication comes from the fact that there may be "loops" in the provenance graph, such that several tuples are mutually derivable from one another, yet none are derivable from base tuples. In order to "garbage collect" these no-longer-derivable tuples, we can also use provenance, to test whether they are derivable from trusted base data; those tuples that are not must be recursively deleted following the same procedure.

Revisiting the provenance graph of Figure 3(b), suppose that we wish to propagate the deletion of the tuple $B(3, 5)$. This leads to the invalidation of mapping nodes labeled $m_3$ and $m_4$. Then, for the tuples that have incoming edges from the deleted mapping nodes, $U(5, c_1)$ has to be deleted, because there is no other incoming edge, while for $B(3, 2)$ there is an alternative derivation, from $G(3, 5, 2)$ through $(m_1)$, and thus it is not deleted. We note that a prior approach to incremental view maintenance, the DRed algorithm [20], has a similar "flavor" but takes a more pessimistic approach. Upon the deletion of a set of tuples, DRed will pessimistically remove all tuples that can be transitively derived from the initially deleted tuples. Then it will attempt to re-derive the tuples it had deleted. Intuitively, we should be able to be more efficient than DRed on average, because we can exploit the provenance trace to test derivability in a goal-directed way. Moreover, DRed's re-derivation should typically be more expensive than our test for derivability, because insertion is more expensive than querying, since the latter can use *only* the keys of tuples, whereas the former needs to use the complete tuples; when these tuples are large, this can have a significant impact on performance. Experimental results in [17] validate this hypothesis.

In the future, we plan to add support for bidirectional propagation of updates over mappings. In this case, we have to deal with a variation of the view update problem, and we expect provenance information to be useful in order to identify possible update policies for the sources and *dynamically* check if they have side-effects on the target of the mappings.

# 4 Discussion

These case studies describe some first steps towards applying provenance to problems related to data replication. In particular, they demonstrate how tracking provenance, either implicitly as in lenses or explicitly as in ORCHESTRA, can improve solutions to traditionally challenging problems such as view update and view maintenance.

There is burgeoning interest in provenance, and more sophisticated models are being actively developed. Whereas early notions such as *lineage* [9] and *why-provenance* [6] only identified which source values "contribute to" the appearance of a value in the result of a query, more recent models [7, 18] also describe *how* those source values contributes to the value in the result. We believe that as these richer models are developed, they will increasingly be applied at all levels of systems including in mechanisms for creating, maintaining, and updating views, for debugging schema mappings [7], and for curating and synchronizing replicated data.

**Acknowledgements** The systems described in this article were developed in collaboration with members of the Harmony (Aaron Bohannon, Benjamin Pierce, Alexandre Pilkiewicz, Alan Schmitt) and ORCHESTRA (Olivier Biton, Todd Green, Zachary Ives, Val Tannen, Nicholas Taylor) projects; the case studies are based on papers co-authored with them. We wish to thank Peter Buneman for helpful comments on an early draft. Our work has been supported by NSF grant IIS-0534592 (Foster), and NSF grants IIS-0477972, 0513778, and 0415810, and DARPA grant HR0011-06-1-0016 (Karvounarakis).

## References


[1] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB 2006, Proceedings of 31st International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea*, pages 953–964, 2006.

[2] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, Jan. 2008. To appear.

[3] A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Chicago, Illinois*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[4] P. Buneman, A. Chapman, J. Cheney, and S. Vansummeren. A provenance model for manually curated data. In *International Provenance and Annotation Workshop (IPAW), Chicago, IL*, volume 4145 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2006.

[5] P. Buneman, S. Khanna, and W. C. Tan. Data provenance: Some basic issues. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS), New Delhi, India*, volume 1974 of *Lecture Notes in Computer Science*, pages 87–93. Springer, 2000.

[6] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In J. V. den Bussche and V. Vianu, editors, *Database Theory — ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*, pages 316–330. Springer, 2001.

[7] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. In *VLDB 2006, Proceedings of 31st International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea*. ACM Press, 2006.

[8] G. Cong, W. Fan, and F. Geerts. Annotation propagation revisited for key preserving views. In *ACM International Conference on Information and Knowledge Management (CIKM), Arlington, VA*, pages 632–641, 2006.

[9] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[10] A. Deutsch, L.Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[11] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.

[12] H. Fan and A. Poulovassilis. Using schema transformation pathways for data lineage tracing. In *BNCOD*, volume 1, pages 133–144, 2005.

[13] J. N. Foster, M. B. Greenwald, C. Kirkegaard, B. C. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4):669–689, June 2007. Extended abstract in *Database Programming Languages (DBPL)* 2005.

[14] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Extended abstract in *Principles of Programming Languages* (POPL), 2005.

[15] J. N. Foster, B. C. Pierce, and A. Schmitt. A logic your typechecker can count on: Unordered tree types in practice. In *Workshop on Programming Language Technologies for XML (PLAN-X), Nice, France, informal proceedings*, Jan. 2007.

[16] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *Proceedings of the AAAI Sixteenth National Conference on Artificial Intelligence, Orlando, FL USA*, pages 67–73, 1999.

[17] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB 2007, Proceedings of 32nd International Conference on Very Large Data Bases, September 25-27, 2007, Vienna, Austria*, 2007.

[18] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*, 2007.

[19] T. J. Green, N. Taylor, G. Karvounarakis, O. Biton, Z. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *SIGMOD 2007, Proceedings of the ACM International Conference on Management of Data, June 11-14, 2007, Beijing, China*, 2007. Demonstration description.

[20] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 157–166. ACM Press, 1993.

[21] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 505–516. IEEE Computer Society, March 2003.

[22] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR 2005: Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA*, pages 107–118, January 2005.

[23] B. C. Pierce et al. Harmony: A synchronization framework for heterogeneous tree-structured data, 2006. `http://www.seas.upenn.edu/~harmony/`.

[24] L. Wang, E. A. Rundensteiner, and M. Mani. U-filter: A lightweight XML view update checker. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE), Atlanta, GA*, page 126. IEEE Computer Society, 2006.