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

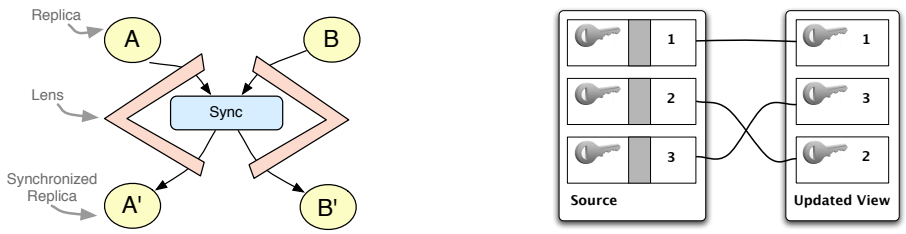
Replication increases the availability of data in mobile and distributed systems. For example, if we copy calendar data from a web service onto a mobile device, the calendar can be accessed even when the network cannot. In peer-based data sharing systems, maintaining a copy of the shared data on a local 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 and 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, 23], but only a few existing systems (e.g., AutoMed [11]) explicitly 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, and are used to handle heterogeneous replicas in the Harmony synchronization framework [22, 12]. They use a simple form of provenance to express the complex update policies needed to correctly handle ordered data. The second case study describes ORCHESTRA [16, 18], a collaborative data sharing system [21]. In ORCHESTRA, data is distributed across tables located on many different peers, and the relationship between connected peers is specified using GLAV [15] 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 between chunks 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 takes the entire state of the updated view as an argument.)

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 the *get* when the update to the view is a no-op, and the *put* function must propagate all of the information in the view back to the updated source (see [13] for a detailed 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 [13, 14, 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 is to enrich lenses with a simple mechanism for tracking provenance: programmers describe how to divide the source into *chunks* and how to generate a *key* for each chunk. These induce an association between pieces of the source and view, and this association 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          get      "Jean Sibelius, Finnish
 Aaron Copland, 1910-1990, American          ⟶        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 ($\star$) 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 nth line in the abstract structure are combined with the years from the nth line in the concrete structure (using a default year range to handle cases where the view has more lines than the source.) For simple updates—e.g., when individual entries have been edited but not reordered—this policy does a good job. On other examples, however, the behavior of this *put* function is highly unsatisfactory. If the update to the abstract string breaks the positional association between lines in the concrete and abstract strings, the output will be mangled—e.g., when the update to the abstract string swaps the order of the second and third lines, then combining

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

yields a mangled source

```
"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 the entry for Britten, and vice versa. What we want, of course, is for the *put* to align the entries in the concrete and abstract strings by matching up lines with identical name components, as depicted in Figure 1(b). On the inputs above, this *put* function would produce

```
"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, lenses need to be able to track the provenance of lines in the source and view. This is accomplished by introducing two new primitives for specifying the *chunks* of the source and a *key* for each chunk, and retooling *put* functions to lookup chunks by key rather than by position. Operationally, these retooled *put* functions work on dictionary structures where each chunk is stored under its key, rather than raw strings.

Here is a lens that gives us 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` in `comps` 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.

Key-based provenance alone does not solve the view update problem completely—although it constrains the *put* function to handles chunks in a reasonable way, it does not specify what to do when the update adds new chunks or modifies the contents of chunks. Provenance provides the vocabulary need to give succinct descriptions of the complicated update policies that are needed when data is ordered. The design of lenses however—i.e., finding natural syntax that describes views and update policies together—remains important.

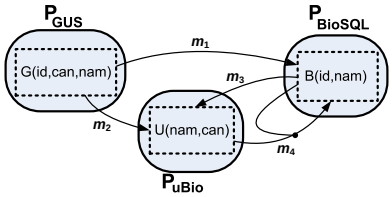
Key-based provenance is simple, but we have found it useful for expressing many update policies that arise in practice. One way that programmers can control the update policy is through their choice of keys. These "keys" are not actually required to be keys in the strict database sense—i.e., we do not demand that the key of each chunk be unique. 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. In particular, this means that if a *put* function that operates positionally is desired, it can be obtained by defining the key component of a lens to be a constant. An extension of key matching currently being investigated uses "fuzzy" metrics like edit distance to align chunks. This relaxed form matching is useful when processing data such as documents that have no obvious notion of key.

Another way to control the update policy is in the definition of the chunks themselves. Many examples are naturally processed using a single level of chunks, as in the composer lens. But chunks may also be nested within each other, which has the effect of stratifying matching into levels: top-level chunks are matched globally across the entire string, subchunks are then aligned within each chunk, and so on. This is useful in cases where the source also has nested structure—e.g., we use it in our lens for LaTeX documents. We have also implemented an extension in which chunks are "tagged" and matching is limited to chunks having the same tag.

As these examples illustrate, simple key-based provenance is capable of expressing many useful update policies. We have used it 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 databases of protein sequences represented in the ASCII SwissProt format and XML. Ongoing and future work is focused on using more sophisticated forms of provenance in lenses, and extending dictionary lenses to richer structures such as trees.

3 ORCHESTRA

ORCHESTRA is a *collaborative data sharing system* (abbreviated CDSS) [21], i.e., a system for data sharing among heterogeneous peers related by a network of schema mappings. Each peer has a locally controlled and



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

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

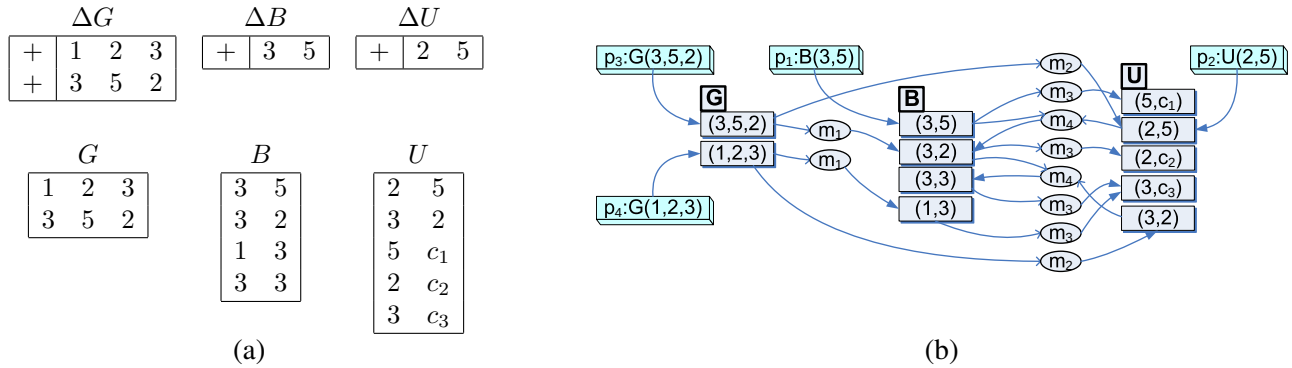


Figure 3: Example of update exchange and resulting provenance graph

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 [10, 9] in data sharing, and they are equivalent to so-called *global-local-as-view* or *GLAV* mappings [15, 20]. Some examples are shown in the right part of Figure 2. 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}.
- 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 [8, 1], as explained in [16]. 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 [16, 17]). 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* [8, 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 [16] 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 [19] 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 [19]. 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 [19], 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 [16] 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 provenance can be used to improve solutions to traditionally challenging problems such as view update and view maintenance, even when it does not provide enough information to solve them on its own.

There is burgeoning interest in provenance, and more sophisticated models are being actively developed. Whereas early notions such as *lineage* [8] 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, 17] 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 (Todd Green, Zachary Ives, Val Tannen, Nicholas Taylor) projects; the case studies are based on papers co-authored with them. 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] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] L. Wang, E. A. Rundensteiner, and M. Mani. U-filter: A lightweight xml view update checker. page 126. IEEE Computer Society, 2006.