Plug-and-Play Queries for Temporal Data Sockets

Curtis E. Dyreson[1] and Sourav S Bhowmick[2]

[1] Department of Computer Science
Utah State University, USA
Curtis.Dyreson@usu.edu
[2] School of Computer Engineering
Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Abstract. Plug-and-play queries are portable, reliable, and easier to code. When a plug-and-play query is plugged into a data socket, the socket transforms the data to the shape needed by the query. If data is annotated with metadata, the semantics of the metadata potentially impacts the transformation. In this paper we describe how to account for the metadata in a transformation. We focus on temporal metadata and show how a transformation can preserve temporal semantics. We also show how the transformation can be driven by the metadata, for instance, the temporal metadata could be used to create data versions.

1 Introduction

A *plug-and-play* query is akin to a plug-and-play device which can be plugged into any kind of socket and used. Plug-and-play queries have a specification of what kind of data they need in order to be evaluated. A *data socket* for a plug-and-play query uses this specification to *transform data* to what is needed for the query to evaluate. Hence, a query writer can code a plug-and-play query for a simple, easy-to-understand schema, plug the query into a data socket, and rely on the socket to automatically adapt the data to the schema needed by the query. The data socket can inform the user whether the transformation is possible or the data is insufficient for producing a reliable answer, and can give precise information about what the data lacks. The benefits of plug-and-play queries are that they are *portable*, you can take a plug-and-play query to any data source and evaluate it, more *reliable*, the query checks the data environment to determine if it can be safely and correctly evaluated, and *easier to code* for complex data since a query writer can write the query with respect to a simple view of the data, abstracting away the data's real complexity.

We previously researched plug-and-play queries for hierarchical data sockets [5–10, 27]. Hierarchies are a popular way to model data. In the 1960s influential DBMSs, such as IBM's IMS [21], managed hierarchical data. The rise of XML in the 90s led to a renewed interest in hierarchal models, *c.f.,* [15], which continues today with research in JSON, *c.f.,* [19, 26] and "nested" data, *c.f.,* [22]. In this paper we extend our previous research to cover temporal, hierarchical plug-and-play queries and data sockets. Data sits in a milieu of descriptive and proscriptive metadata. Examples include a schema, character sets, privacy annotations, and security restrictions. We focus in this paper

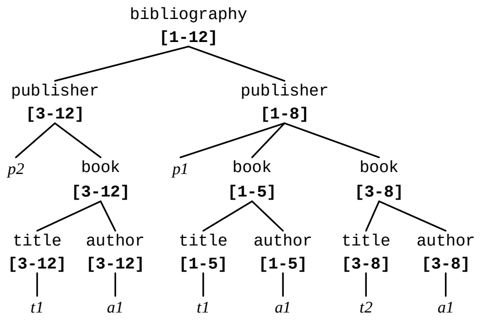

Fig. 1: Books within publishers

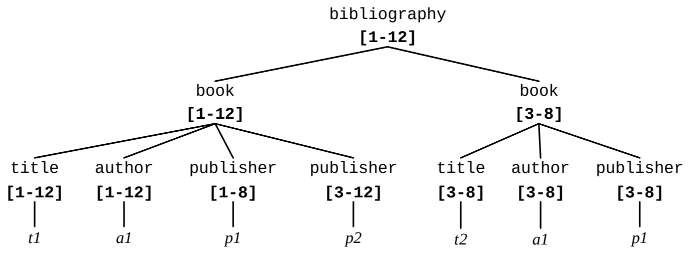

Fig. 2: An incorrect temporal transformation

on temporal data, which is data annotated with time metadata. To plug a query into a temporal data socket, the time metadata together with the data should be adapted to what the temporal query needs. For a socket to correctly transform such data, the transformation must ensure that the semantics of the annotating metadata is observed, in particular *sequenced semantics* for temporal metadata [25].

Consider the following example. A common way to represent temporal hierarchical data is as a tree in which each node has a timestamp [11, 12, 14]. Figure 1 shows a temporal version of some publisher data where the timestamp (shown below an element) indicates the database lifetime of the node, *i.e.,* the *transaction time* [16]. For instance, the timestamp for publisher *p1* in Figure 1 indicates that data about *p1* was inserted at time 1 and is current until time 8. Suppose we want to transform the data so that books are above publishers in the hierarchy. A possible strategy is to first transform the timestamp-stripped source tree (using, for example, our transformation language XMorph [10]) and then compute the timestamp for each node in the target. The resulting tree is shown in Figure 2. Computing the timestamps for the target is straightforward. The timestamp of a node in the target is the union of the timestamps of all the nodes in the source it corresponds to; for example, the timestamp of the *t1* book in Figure 2 is [1-12], which is the union of [1-5] and [3-12]. When constrained by its parent's timestamp, a node's timestamp may need to "shrink." For example, the *p1* publisher in Figure 2. has a timestamp of [3-8] even though the timestamp of its counterpart in Figure 1 is [1-8], because its lifetime is temporally constrained by that of its parent's, [3-8] [4].

But this simple approach is flawed, because the transformation is **not** *temporal information-preserving*, in particular it fails to follow sequenced semantics [2]. Sequenced semantics is pictured in Figure 3, where the temporal transformation semantics is defined in terms or reduces to a non-temporal transformation semantics. A temporal hierarchy can be thought of as a sequence of snapshots. Each snapshot is the *slice* of temporal data current at an instant. Individually, each snapshot can be transformed by a previously defined and well-known non-temporal transformation. Sequenced semantics stipulates that the meaning of a temporal transformation is *snapshot-equivalent* (or *snapshot reducible* [24] or *S-reducible* [3]) to the (non-temporal) transformation of each slice. So if we perform a temporal transformation and then slice the data, the set of snapshots should be the same as what we would obtain by first slicing the data and then transforming each slice.

The transformation described above fails to observe sequenced semantics. The *p1* publisher and the *t1* book are related only at time [1-5] in Figure 1 but in Figure 2 we can see they are related at time [1-8]. A transformation that observes sequenced semantics should relate them at time [1-5] exactly, not including time [6-8]. The timestamps introduce additional semantic constraints that need to be properly taken care of to ensure that the transformation is (temporal) information-preserving.

Our motivating example illustrates that when hierarchical data is annotated with metadata, special techniques are needed to ensure the preservation of the metadata's semantics in a transformation. The metadata can also *explicitly influence the transformation*. Consider for example a transformation that produces "versions" of the data, where a version is defined as a change in the children of a node. Different transformations will induce different versions; so the versions must be constructed dynamically.

This paper makes the following contributions.

- We describe a reversible, temporal transformation technique for data sockets and show how to detect information loss in the transformation, *i.e.,* to determine whether sequenced semantics is preserved.
- We show how the time metadata can drive a transformation.
- We describe how our technique extends to other kinds of metadata.

2 Background: Review of Plug-and-Play Queries

We previously investigated plug-and-play queries for hierarchical data. We introduced the concept of a *query guard*, which turns an ordinary query into a plug-and-play query [7]. A query guard is a lightweight reusable specification of the hierarchy that a query needs. It protects the query by testing whether the data can be transformed (without losing information) to the hierarchy given in the guard, and transforms the data if needed. A query guard focuses only on the *structure, not the semantics,* of the data because semantic web technologies, *e.g.,* ontologies, already address the orthogonal semantic matching problem.

A query guard allows the query to couple to any hierarchy that can be converted, cast, or coerced to the type (hierarchy) needed by the query. The guard protects the query by checking whether the data can be transformed, without losing information,

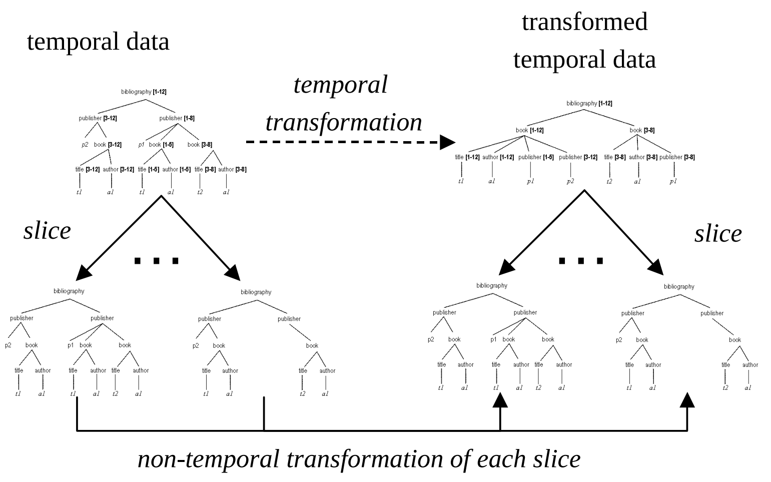
temporal data
transformed temporal data
temporal transformation
slice
slice
non-temporal transformation of each slice

Fig. 3: Sequenced semantics

to the type (hierarchy) needed by the query. Some transformations lose information when the hierarchy is manipulated, but the guard can alert the programmer to lossy transformations. The data could be physically [7] or *virtually* transformed [8].

As an example of querying data with a query guard, assume that we want to extract the book publishers using the XQuery query given below.

```
<data> {
    for $b in doc("x.xml")//book
    return <book>{$b/title} {$b/publisher}</book>
 } </data>
```

Suppose that the query is applied to the hierarchy depicted in Figure 1. The query will *fail* to produce the desired result because the path expression in the query do not match the shape of the data. The failure will not generate an error, rather the query will run to completion and yield an empty or partial answer. On the other hand, if the query is run on the data in Figure 2 it will produce the desired result.

We developed a language called XMorph to express query guards for plug-and-play queries and data sockets [5, 7, 10]. A guard for the example XQuery query is given below.

```
MORPH bibilography [
        (GROUP book [title]) [
          title author publisher
        ]
      ]
```

The guard specifies that a `<bibliography>` has as children `<book>`s, and each `<book>` is grouped by `<title>` and has `<title>`, `<author>`, and `<publisher>` children, that is, it is the hierarchy of Figure 2. With the guard the query can now be run successfully on the data in Figure 1 and Figure 2, or other book mini-world hierarchies since the socket will transform the data to what is needed for the query to evaluate.

Some transformations potentially lose information. Consequently, it is important for a query guard to identify and report a *lossy* transformation. It is not readily apparent in the aforementioned example whether the guard is "good" in the sense that it protects the query by neither manufacturing nor discarding data. This issue is vital to a user. If the transformation specified by a guard is lossy then the subsequent query evaluation will be similarly lossy and inaccurate. Let's introduce terminology to more precisely describe what we mean by a *good* guard. This terminology is adapted from the vocabulary of type systems in programming languages since a guard plays a role similar to a data type in a programming language, *i.e.,* it defines how the data is structured or encoded. A guard is *narrowing* if it ensures that data is not created, *widening* if it ensures that no data is lost, *strongly-typed* if it both narrowing and widening, *weakly-typed* if it neither narrowing nor widening, or has a *type mismatch* if the guard mentions a type that is absent from the source. A query guard can provide detailed feedback about which part of a guard is lossy. A programmer can use this feedback to add syntax to a query guard to indicate that the loss is acceptable, *e.g.,* most narrowing transformations will be fine, just as a C++ programmer might add a `cast()` to transform the result of an expression to a suitable type.

3 A Temporal Hierarchical Model

A temporal hierarchy can be modeled as a labeled tree.

Definition 1 (Temporal Hierarchy).
Let T be a set of chronons which forms a discrete image of a continuous time-line. A temporal hierarchy is a tuple (V, E, Σ, L, T, S), where

- *V is a set of nodes,*
- *$E : V \times V$ is a tree edge set of the form (p, c), where p is the parent and c is the child,*
- *Σ is an alphabet of labels and text values,*
- *$L : V \rightarrow \Sigma$ is a label/text value function that maps a node to a label/text value,*
- *T is the chronon set, and*
- *$S : V \rightarrow 2^T$ is a timestamp function that maps a node to a timestamp (a set of chronons).*

The hierarchy is said to be temporally-consistent iff $\forall p, c[(p, c) \in E \Rightarrow S(c) \subset S(p)]$.

For simplicity, we discuss a data model with only one time dimension. The temporally-consistent property formally specifies that a child's timestamp has to be included in its parent's timestamp. Note that each edge implicitly has the same timestamp as that of the child node. This is because the two nodes are related only when the child exists. The model ignores some features of XML's DOM such as sibling order, node types, *e.g.,* attribute, processing instruction, comment, element, and text nodes, and labels and values are used for element names and text nodes, respectively. While these aspects could be modeled, the simpler model is sufficient for our purposes.

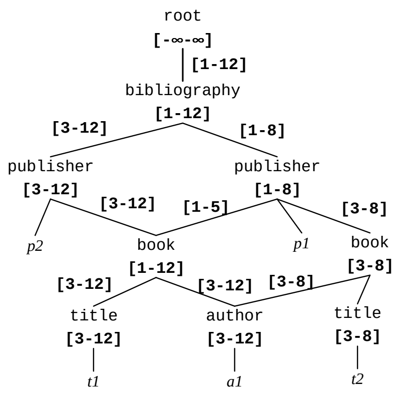

Fig. 4: The grouped, temporal hierarchy for the data of Figure 1

For this paper, we assume that the labels are unambiguous, *e.g.,* a `<title>` element is the title of a book rather than an author. A method to disambiguate labels is given elsewhere [10].

To support grouping in data transformations we extend the tree model to a partial order by explicitly adding groups as defined below.

Definition 2 (Grouped Temporal Hierarchy).
A grouped, temporal hierarchy is a tuple $(G, V_G, E_G, \Sigma, L, T, S_G, X_G)$*, constructed from a temporal hierarchy,* (V, E, Σ, L, T, S)*, where*

- $G : V \rightarrow V_G$ *is a group identity function, which maps a node to a group node (there is a single group node for each group),*
- V_G*: is a set of nodes,* $V \bigcup \{root\}$*,*
- $E_G : V_G \times V_G$ *is the edge set where* $E_G = \{(G(v), G(w)) \mid (v, w) \in E\}$*,*
- $S_G : V_G \rightarrow 2^T$ *is a node timestamp function that maps a group node,* $v \in V_G$*, to a timestamp, which is which is computed as*

$$S_G(g) = \bigcup_{\forall v[G(v)=g]} [S(v)]$$

and
- $X_G : E_G \rightarrow 2^T$ *is an edge timestamp function that maps an edge,* $(v, w) \in E_G$*, to a timestamp, which is computed as*

$$X_G((v, w)) = \bigcup \{S(y) \cap S(z) \mid G(y) = v \ \wedge \ G(z) = w \ \wedge \ (y, z) \in E\}$$

As an example, consider Figure 4 which is the grouped temporal hierarchy corresponding to the temporal hierarchy in Figure 1. Each edge in the figure is timestamped. There are groups for the book with title *t1* and author *a1*, for the author `a1`.

4 Transforming Temporal Data

A non-temporal transformation transforms data by finding relationships between nodes. Our technique used *closest* relationships, which are nodes connected by a path that is the shortest for all nodes of the node *types* at the start and end of the path. A node type is the concatenation of label on a path from the root to a node. For instance, the type of the `<book>` nodes in Figure 4 is `bibliography.publisher.book`. Suppose that our XMorph query guard makes `<publisher>`s children of `<book>`s. Then the shortest path between `<publisher>` and `<book>` types has length 1. The `<book>` with title *t2* then is closest to only the `<publisher>` *p1* since the shortest path to that `<publisher>` is of length 1 but the shortest path to `<publisher>` *p2* is 3.

4.1 Closest Lifetimes

Closest relationships have lifetimes.

Definition 3 (Closeness time).
In a grouped, temporal hierarchy, $(G, V_G, E_G, \Sigma, L, T, S_G, X_G)$*, let* $v, w \in V_G$ *be a pair of closest nodes, and* P *be the set of shortest paths between* v *and* w*. Then* v *is closest to* w *at time* t *where*

$$t = \bigcup_{p \in P} [\bigcap_{e \in p} X_G(e)]$$

The definition says that the lifetime of a closest relationship is the union of the times of the paths between the nodes, where the time of a path is the intersection of the times on the path's edges. Consider the lifetime of the closest relationship between the `<author>` *a1* and `<publisher>` *p1* in Figure 4. There is one shortest path through each `<book>` node. The leftmost path has a lifetime of [3-12] $\bigcap$ [1-5] = [3-5] while the rightmost path's lifetime is [3-8] $\bigcap$ [3-8] = [3-8]. So the lifetime is [3-5] $\bigcup$ [3-8] = [3-8].

4.2 Computing Closest Lifetimes

We now show how to compute the timestamp as data is transformed. The (non-temporal) transformation determines how to place children beneath parents as described elsewhere [7]. The lifetime of a node in the transformed data can be computed as the child is created. Let $v = \{(v_1, t_1), \ldots, (v_n, t_n)\}$ where t_i is the time of node v_i, and $w = \{(w_1, s_1), \ldots, (w_m, s_m)\}$ be a pair of closest, grouped nodes where v is the parent and w is the child in the transformed data. Then the time for w in the transformed data is

$$\bigcup \{t_i \cap s_k \mid (v_i, t_i) \in v \wedge (w_m, s_m) \in w \wedge v_i \text{ is closest to } w_k\}$$

Because a temporal hierarchy is temporally-consistent (a child's lifetime is a subset of a parent's lifetime), a shortest path always pases through a least common ancestor, and the lifetime along a path is computed by intersection, the lifetimes for every edge in the path are not needed, but can be inferred from the lifetime of the source and sink nodes. As an example, consider the task of placing publisher *p1* beneath the grouped book with title *t1*. There are two book nodes in the group, $\{(b_1, [3-12]), (b_2, [1-5])\}$, and one node in the publisher group $\{(p_1, [1-8])\}$. b_1 is not closest to p_1 (it is closer to publisher *p2*) so the lifetime is the intersection of b_2 and p_1, which is [1-5].

To analyze the time cost of the lifetime computation, let N be the number of nodes in a grouped node and c be the cost of determining if a pair is close, then the time cost is $O(cN^2)$.

4.3 Information Loss

We can also compute information loss with low cost. Observe that a transformation may "shrink" the metadata (through temporal intersection) but will never increase it (the lifetime of a grouped node will never be increased, rather the temporal union just pieces together the grouped lifetime from the members of the group). A *reversible* transformation retains information, and the ability to reverse the transformation, while a *narrowing* transformation loses closest relationships present in the data.

Theorem 1. *A temporal transformation is reversible if it is based on a reversible non-temporal transformation, that is, if it preserves all of the non-temporal closest relationships, and if all nodes in the transformed data (modulo grouping) have the same lifetime as the node in the source data (modulo grouping).*

Proof. Assume a temporal hierarchy, (V, E, Σ, L, T, S) which is transformed to temporal hierarchy, $(V', E', \Sigma', L', T', S')$. If the non-temporal part of the transformation is reversible then we know that we can reverse the transformation to obtain the original set of edges, nodes, label function, etc. (this proof is given elsewhere [27]). So we need to show that we can obtain S from S' Assume $v \in S$ and $S(v) = t$. We know that there exists $v \in V'$ because the transformation is (non-temporal) reversible. Assume $S'(v) = t'$. Then there are three cases.

$t \subset t'$ - Information has been added. There exists at least one snapshot in the transformed data that contains v which does not exist in the source. The transformation, therefore, must be *widening* and is not necessarily reversible.

$t' \subset t$ - Information has been lost. There exists at least one snapshot in the source data that contains v which does not exist in the source. The transformation is *narrowing*.

$t' = t$ - The source and transformed data contain the same snapshots, hence no information is lost.

So it is necessary for the lifetimes to be the same to guarantee that the transformation preserves the timestamp function. ■

The temporal information loss can be quantified as follows. For each edge, (p, c), in the transformed data, compute the timestamp shrinkage, $v_I = S(c) - S(p)$. Let $V_I = \bigcup v_I$ for all v in the transformed data and let r be the root of the hierarchy. Then the amount of information loss is how many snapshots are lost vis-a-vis the number of snapshots in the document, $|(S(r) - V_I)|/|S(r)|$.

4.4 Other Temporal Semantics

Other temporal semantics, such as earliest and latest semantics, have been described [13], but as they generalize sequenced semantics, the techniques developed in this paper apply to those semantics.

A more interesting kind of transformation utilizes the metadata in the transformation. In a *version transformation* each combination of children creates a distinct version of a node. A versioned temporal hierarchy would help answer queries such as "select the latest version of each book".

Definition 4 (Versioned temporal hierarchy).
Let $D = (V, E, \Sigma, L, T, S)$ be a temporal hierarchy. The versioned hierarchy of D, denoted $D_V = (V_V, E_V, \Sigma, L, T, S_V)$, is defined as follows.

- $V_V = V \bigcup \{v_1, \ldots, v_m\}$ *where v_i is a* `<version>` *node, there is one version node for every change in the children of a node.*
- $E_V = E_1 \bigcup E_2$ *where*
  - $E_1 = \{(v, v) \mid (v, c) \in E \ \wedge \ v \text{ is a version node } \wedge \ S_V(v) \subseteq S(v)\}$*, and*
  - $E_2 = \{(v, c) \mid (v, c) \in E \ \wedge \ v \text{ is a version node } \wedge \ S_V(v) \subseteq S(c)\}$
- $S_V = S \bigcup \Upsilon$ *where Υ is a function that maps a version node to a timestamp that represents the lifetime of the version (maximal time when the children remain the same).*

As an example, Figure 5 is the versioned hierarchy for the data in Figure 1.

Versioning can be computed during a transformation by sorting the children of a transformed node by their timestamps and chopping them into versions. While this increases the runtime cost by $O(N \log N) + O(V)$ where N is the number of transformed nodes and V is the number of versions. Since there can be at most two versions per (interval) timestamp, since only the endpoints represent a change in the existence of a child, $O(V) = O(N)$, the overall worst-case time cost is the cost of the sort.

4.5 Other Kinds of Metadata

The techniques developed in this paper can be generalized to apply to other kinds of metadata by using different functions to compute the metadata along a path and combine metadata from paths in grouping, which for temporal data are basically intersection and union, respectively. Consider Bayesian probabilistic metadata, and assume probabilistic independence. Figure 6 shows the data of Figure 1 but with probabilities for metadata. Each node has two probabilities in the figure. The top number is probability that the node is a child of the parent, that is, the probability of the node's existence, which we will call the *existence probability*. The bottom number is the conditional probability that the node exists, computed as the conditional probability of its parent's existence times the probability of its existence. So for example, the `<book>` for `<publisher>` *p2* has an existence probability of .1 (the top number), and a conditional probability of .09 (the bottom number) which is computed as its parent's conditional probability, .9, times its probability, .1. The probability along a path is computed using multiplication. Paths are combined using Bayesian addition (assuming independence). The transformed data, with a Bayesian independence assumption, is shown in

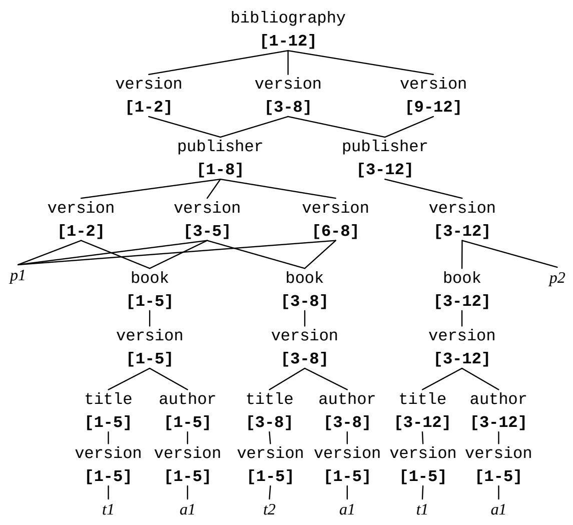

Fig. 5: Versioned books within publishers

Figure 7. The leftmost <book> has as its existence probability the formula "1 - the probability that neither <book> in the group exists". The group has two <book>s, which exist with probability .09 and .4, respectively (as computed in Figure 6). So the existence probability of the grouped <book> is $1 - (1 - .09) * (1 - .4)$. Other kinds of metadata, security, provenance, etc., will use other operations to enforce a semantics.

5 Related Work

Previous hierarchy-related research in querying data with the wrong shape can be broadly classified into several categories. **Query relaxation/approximation** techniques loosen the tight coupling of path expressions to the hierarchy of data is to relax the path expressions or approximately match them to the data by exploring a space of hierarchies that are within a given edit distance, *c.f.,* [1]. **Hierarchical search engines** de-couple queries from specific hierarchies, similar to our aims, and can find data in differently-shaped hierarchies, *c.f.,* [20]) **Structure-independent querying** techniques use a least common ancestor to query data independent of its hierarchy, *c.f.,* [18]. Finally, there is research in **declarative transformation languages** [17, 23]. We extend the final approach in this paper and describe how to transform data data annotated with metadata. There is no previous research on working with data annotated with metadata in any of the above categories.

6 Conclusion

Transforming data is an important part of query evaluation. When data is annotated with metadata, the transformation has to preserve the semantics of the metadata, par-

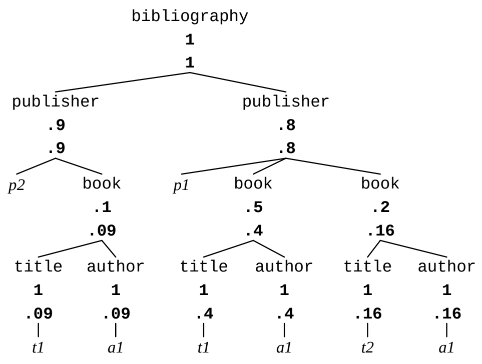

Fig. 6: Probabilistic books within publishers

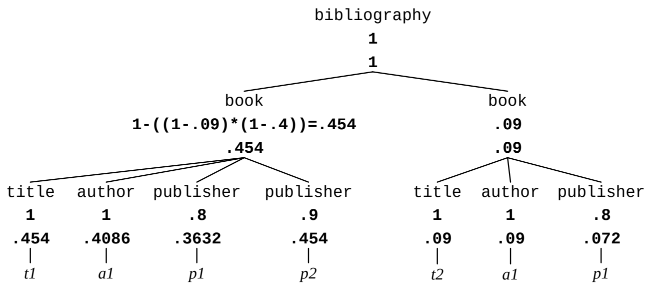

Fig. 7: Transformed probabilistic data

ticularly when the data is grouped. In this paper we investigated the transformation of data annotated with temporal metadata. We presented a sequenced transformation technique, which restructures temporal data while ensuring sequenced semantics. We also presented a versioning transformation technique that reorganizes the data into versions. We are currently implementing the transformation in XMorph. More on the project, including code, a tutorial, and a demo, can be found at `http://digital.cs.usu.edu/~cdyreson/XMorph`.

References

1. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree Pattern Relaxation. In: EDBT. pp. 496–513 (2002)
2. Böhlen, M.H., Jensen, C.S.: Sequenced Semantics. In: Encyclopedia of Database Systems, pp. 2619–2621 (2009)
3. Böhlen, M.H., Jensen, C.S., Snodgrass, R.T.: Temporal Statement Modifiers. ACM Trans. Database Syst. 25(4), 407–456 (2000)
4. Currim, F., Currim, S., Dyreson, C.E., Snodgrass, R.T., Thomas, S.W., Zhang, R.: Adding temporal constraints to XML schema. IEEE Trans. Knowl. Data Eng. 24(8), 1361–1377 (2012)

5. Dyreson, C., Bhowmick, S., Jannu, A., Mallampalli, K., Zhang, S.: XMorph: A Shape-polymorphic, Domain-specific XML Data Transformation Language. In: ICDE. pp. 844–847 (2010)
6. Dyreson, C., Zhang, S.: The Benefits of Utilizing Closeness in XML. In: DEXA Work. pp. 269–273 (2008)
7. Dyreson, C.E., Bhowmick, S.S.: Querying XML Data: As You Shape It. In: ICDE. pp. 642–653 (2012)
8. Dyreson, C.E., Bhowmick, S.S., Grapp, R.: Querying virtual hierarchies using virtual prefix-based numbers. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. pp. 791–802 (2014)
9. Dyreson, C.E., Bhowmick, S.S., Grapp, R.: Virtual exist-db: Liberating hierarchical queries from the shackles of access path dependence. PVLDB 8(12), 1932–1943 (2015)
10. Dyreson, C.E., Bhowmick, S.S., Mallampalli, K.: Using XMorph to Transform XML Data. PVLDB 3(2), 1541–1544 (2010)
11. Dyreson, C.E., Grandi, F.: Temporal xml. In: Encyclopedia of Database Systems, pp. 3032–3035 (2009)
12. Dyreson, C.E., Mekala, K.G.: Prefix-based node numbering for temporal xml. In: WISE. pp. 172–184 (2011)
13. Dyreson, C.E., Rani, V.A., Shatnawi, A.: Unifying sequenced and non-sequenced semantics. In: 22nd International Symposium on Temporal Representation and Reasoning, TIME 2015, Kassel, Germany, September 23-25, 2015. pp. 38–46 (2015)
14. Dyreson, C.E., Snodgrass, R.T., Currim, F., Currim, S.: Schema-mediated exchange of temporal xml data. In: ER. pp. 212–227 (2006)
15. Jagadish, H.V., Al-Khalifa, S., Chapman, A., Lakshmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: TIMBER: A native XML database. VLDB J. 11(4), 274–291 (2002)
16. Jensen, C.S., C. E. Dyreson (editors): A Consensus Glossary of Temporal Database Concepts - February 1998 Version. In: Temporal Databases: Research and Practice, Lecture Notes in Computer Science 1399. pp. 367–405. Springer-Verlag (1998)
17. Krishnamurthi, S., Gray, K.E., Graunke, P.T.: Transformation-by-Example for XML. In: PADL. pp. 249–262 (2000)
18. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: VLDB. pp. 72–83 (2004)
19. Liu, Z.H., Hammerschmidt, B.C., McMahon, D.: JSON data management: Supporting schema-less development in RDBMS. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. pp. 1247–1258 (2014)
20. Liu, Z., Walker, J., Chen, Y.: XSeek: A Semantic XML Search Engine Using Keywords. In: VLDB. pp. 1330–1333 (2007)
21. McGee, W.C.: The Information Management System IMS/VS Part I: General Structure and Operation. IBM Systems Journal 16(2), 84–95 (1977)
22. Melnik, S., Gubarev, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. Commun. ACM 54(6), 114–123 (2011)
23. Pankowski, T.: A High-Level Language for Specifying XML Data Transformations. In: ADBIS. pp. 159–172 (2004)
24. Snodgrass, R.T.: The Temporal Query Language TQuel. ACM Trans. Database Syst. 12(2), 247–298 (1987)
25. Snodgrass, R.T. (ed.): The TSQL2 Temporal Query Language. Kluwer (1995)
26. Tahara, D., Diamond, T., Abadi, D.J.: Sinew: a SQL System for Multi-Structured Data. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014. pp. 815–826 (2014)
27. Zhang, S., Dyreson, C.E.: Symmetrically Exploiting XML. In: WWW. pp. 103–111 (2006)