8-2021

MetaXMorph: Hierarchical Transformation of Data with Metadata

Shubham Airan
Utah State University

METAXMORPH: HIERARCHICAL TRANSFORMATION

OF DATA WITH METADATA

by

Shubham Airan

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Curtis Dyreson, Ph.D. Major Professor	Dan Watson, Ph.D. Committee Member
Steve Petruzza, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2021

ABSTRACT

MetaXMorph: Hierarchical Transformation
of Data with Metadata

by

Shubham Airan, Master of Science
Utah State University, 2021

Major Professor: Curtis Dyreson, Ph.D.
Department: Computer Science

When data is annotated with metadata, the semantics of the metadata potentially impacts the transformation. In this thesis, we describe how to account for the metadata in a transformation. We extend XMorph, a system for transforming hierarchical data, to encompass data annotated with metadata. We show how to support both temporal and probabilistic metadata as example metadata domains. We demonstrate that the XMorph extension has low overhead.

(40 pages)

PUBLIC ABSTRACT

MetaXMorph: Hierarchical Transformation
of Data with Metadata
Shubham Airan

This research is about transforming data. Data comes in different shapes; it can be structured as a graph, a tree, a collection of tables, or some other shape. In this thesis, we focus on data structured as a tree, which is known as hierarchical data. The same data could be structured in many different tree shapes. Previously it was shown how to transform data from one tree shape, one hierarchy to another without losing any information. But sometimes the pieces of the hierarchy are annotated or associated with metadata, that is, with data about the data itself. The metadata can have special semantics that must be preserved when the data is transformed. Previous research also sketched how to transform hierarchical data annotated with metadata without losing information while preserving the semantics of the metadata. In this thesis, we implement the research on transforming data with metadata by extending XMorph, a data transformation language. And we evaluate the extension showing that the overhead is modest.

To my family

ACKNOWLEDGMENTS

Foremost, I would like to express my gratitude to my advisor Dr. Curtis Dyreson, for the continuous support of my Master's study and research, for his patience, motivation, enthusiasm, and immense knowledge. The door to Dr. Dyreson's office was always open whenever I ran into a trouble spot or had a question about my research or writing. His guidance helped in all the time of research and writing of this thesis. I could not have imagined a better advisor and mentor for my Master's study.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Dan Watson and Dr. Steve Petruzza for their encouragement, insightful comments, and hard questions.

My sincere thanks also go to Kaitlyn Fjeldsted, Genie Hanson, Vicki Anderson, and Cora Price for providing constant help and support whenever I ran into departmental issues, these are the people who bind the Computer Science department at Utah State University together.

Last but not the least, I am extremely grateful to my family and friends for their continuous support towards my education.

Shubham Airan

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Hierarchies are a popular way to model data. In the 1960s influential DBMSs, such as IBM's IMS [1], managed hierarchical data. The rise of XML in the 90s led to a renewed interest in hierarchical models, *c.f.,* [2], which continues today with research in JSON, *c.f.,* [3, 4] and "nested" data, *c.f.,* [5].

XML data has a tree-like model. Data in the tree is arranged in a particular shape as described by, for instance, a data guide. About 40 years ago E.F. Codd observed that this kind of data model has a problem. Queries in tree-like data models utilize path expressions that are necessarily tightly coupled to shape of data.

This problem can be solved if we could transform the data to shape needed by query, for instance, using XMorph, a shape transformation system. Previously it was shown how to transform XML data with time metadata [6]. In this thesis, we extend this previous research to implement support in the XMorph transformation system [7, 8].

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 thesis on temporal data, which is data annotated with time metadata, and probabilistic data, which is data annotated with probabilities. To transform data annotated with metadata the metadata together with the data should be adapted to what the query needs. The transformation must ensure that the semantics of the annotating metadata is observed, for example, sequenced semantics for temporal metadata [9].

Consider an example using the data shown in Figure 1.1 which focuses on temporal metadata. A common way to represent temporal hierarchical data is as a tree in which each node has a timestamp. Figure 1.1 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. For instance, the timestamp for publisher `p1` in Figure 1.1 indicates

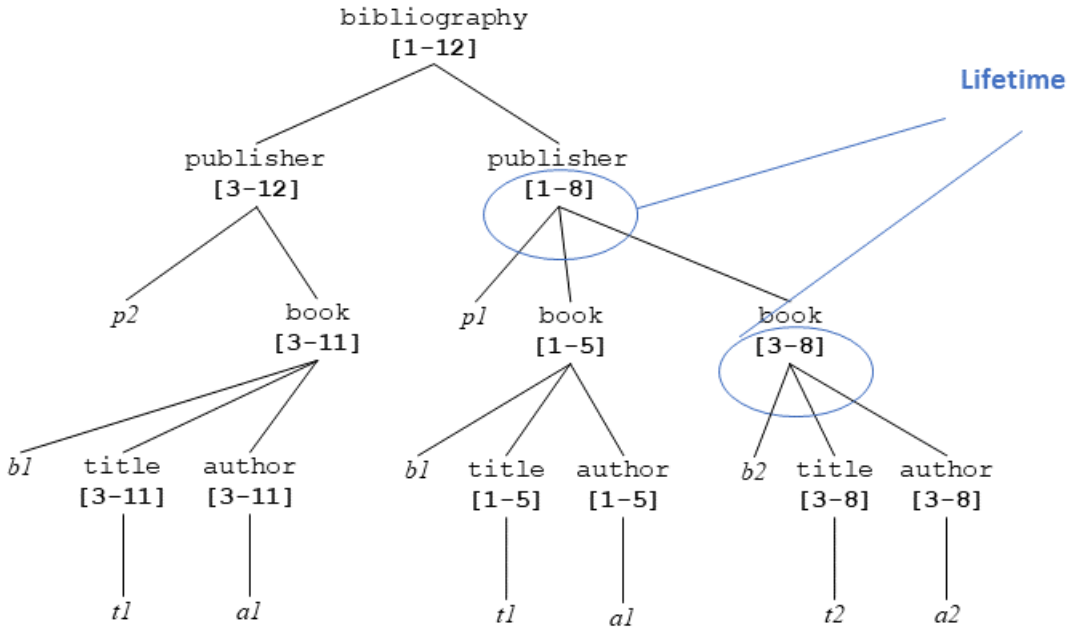

Fig. 1.1: Input data annotated with temporal metadata

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, the transformation language XMorph) and then compute the timestamp for each node in the target. The resulting tree is shown in Figure 1.2. 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 `b1` book in Figure 1.2 is [1-11], which is the union of [1-5] and [3-11]. When constrained by its parent's timestamp, a node's timestamp may need to "shrink." For example, the `p1` publisher in Figure 1.2 has a timestamp of [3-8] even though the timestamp of its counterpart in Figure 1.1 is [1-8] because its lifetime is temporally constrained by that of

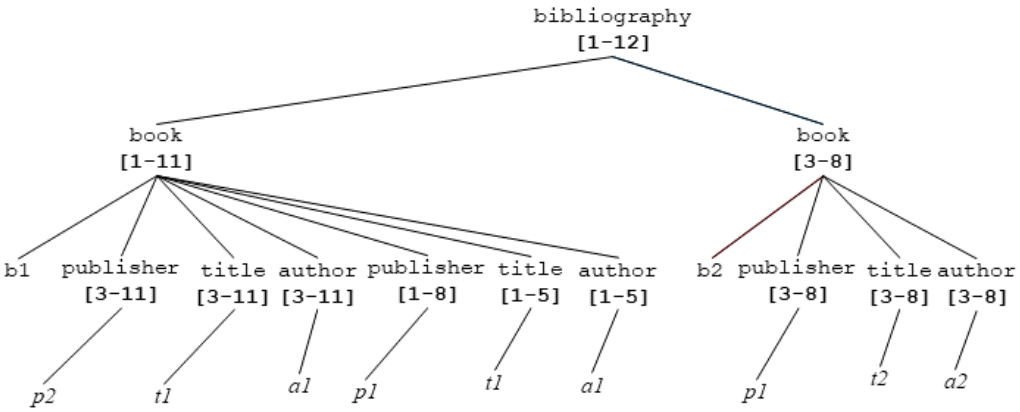

Fig. 1.2: An incorrect XML transformation

its parent's, [3-8]. But this simple approach is flawed because the transformation is not temporal information-preserving, in particular, fails to follow sequenced semantics.

The `p1` publisher and the `b1` book are related only at time [1-5] in Figure 1.1 but in Figure 1.2 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.

This thesis makes the following contributions.

- We implement an algorithm for correctly transforming hierarchical data annotated with metadata.
- We describe the changes to the XMorph data transformation system that are needed to correctly store and retrieve the metadata associated with each node.
- We implement metadata-specific transformations for two kinds of metadata: time and probability.

- We evaluate the implementation to demonstrate that it adds only a modest overhead.

CHAPTER 2

Background: Review of XMorph

In this chapter, we describe how XMorph performed XML transformations without taking metadata into account.

XMoprh implements the concept of a *query guard*, which turns an ordinary query into a plug-and-play query [7, 8, 10–14]. 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. The data could be physically [11] or *virtually* transformed [13].

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.

Research on query guards focuses on matching and transforming the *shape* of the data.

It is agnostic about the *semantic* matching of labels between the guard and the source, *e.g.,* does `person` in the guard mean the same as `person` in the data, because the *semantic matching problem* is already being researched by other communities, *e.g.,* work on ontologies in the Semantic Web community. The focus of query guard research is on the shape of the data and because the problem is orthogonal; Semantic Web solutions can be added to plug-and-play queries to address the problem of semantic mismatch.

A DBMS can be engineered to efficiently support query guards [13, 14]. There are a multitude of commercial and research systems to process, store, manage and query hierarchical data. Many of these systems rely on special techniques to number the nodes in a hierarchy such a way that queries can be evaluated efficiently. Among the various numbering techniques, *prefix-based numbering* (DLN) (also called *containment encoding*, *Dewey order*, *Dewey numbering*, and *Dynamic-level numbering*) is popular [15–20]. Dyreson *et. al* developed a new numbering technique called *virtual prefix-based numbering* that continues to use the prefix-based numbers originally present in the data when evaluating a query on the result of a data transformation [13].

A data transformation potentially changes the value of a node. Dyreson *et. al* also showed how virtual prefix-based numbers can be used to efficiently compute a transformed value [13]. The pieces of the transformed value could be non-contiguous and out of order with respect to the transformed value. An overview of the strategy to compute transformed values is depicted in Figure 2.1. The strategy has two steps: 1) *Preprocessing* and 2) *Rendering*. Preprocessing identifies a "container" string that contains all the pieces of the transformed value, reads the container from disk, and constructs *children queues* that are populated with the pieces that will form the value. The second step, Rendering, use the queues to construct the transformed value from the pieces.

2.1 DLN, the underlying data structure behind everything

To understand the research in this thesis it is important to give an overview of Dewey level numbering (DLN). DLN is the key to efficient processing of nodes in XMoprh. DLN is used in storing the nodes, identifying the closest nodes, and transforming from one XML

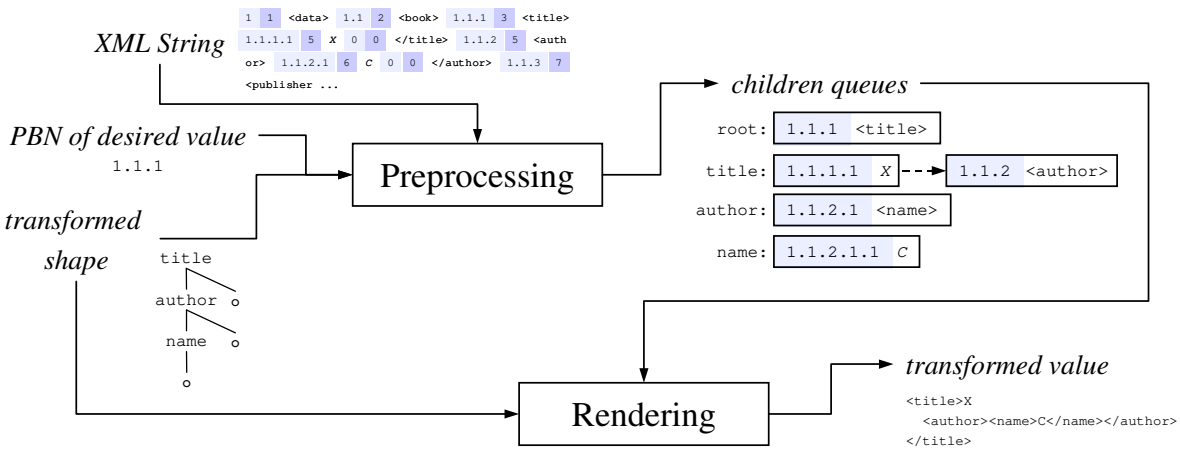

Fig. 2.1: Two-step process for computing a transformed XML string

structure to another. DLN's are hierarchical ids, which borrow from Dewey's decimal classification. Examples for node ids: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3. In this case, 1 represents the root node, 1.1 is the first node on the second level, 1.2 is the second, and so on. To support efficient insertion of new nodes between existing nodes, we use the concept of sub-level ids. Between two nodes 1.1 and 1.2, a new node can be inserted as 1.1/1, where the / is the sub-level separator. The / does not start a new level. 1.1 and 1.1/1 are thus on the same level of the tree.

2.2 Implementation of XMorph

In this section, we cover some details relevant to our project about the implementation of XMorph.

2.2.1 Parsing and storing the input

We first make a root node (supported by DLN), which is document root. We also push it to idStack stack. We also push it to parent stack, where we will store all parent ids. After that when we start to parse the input XML we use method startElement(). Below are the key functionalities of this method.

1. We get the parent ID from the top of stack.

2. We check if it is in a map, this map stored parent, child nodes stored previously.

   (a) if this parent already has a child, then we create next sibling of the previous child, using method of DLN.

   (b) if it did not have any child previously then, we will create new child of this parent.

3. after that we will store this parent, child in map. here replacement of previous child works because we only use it to create new siblings of child nodes.

4. after that we add this new child in idStack, and parent to parent id stack.

2.2.2 Query Compilation

We give input as string to query and get a query object. It is useful in generating a list of shape nodes of output.

we get a hierarchy of node types. For example, for the query in Figure 2.2, we build an object which has information that it has book is group type and has child publisher. Here we also set some shape nodes that will be dynamically grouped. This information will be very useful when we start to build the graph.

2.2.3 Generate the resulting shape nodes

Every Shape Node has two key information, the type ID of node, which is whether it is book or publisher or something else. and it also has previous DLN info, which helps locate where in the hierarchy it was present before.

Based on the query, we first build list of shape nodes. For example, if we wanted to restructure the XML in Figure 2.2 such that we group books with child publisher, then the structure of output should be as follows.

```
document root
  bibliography
    book
```

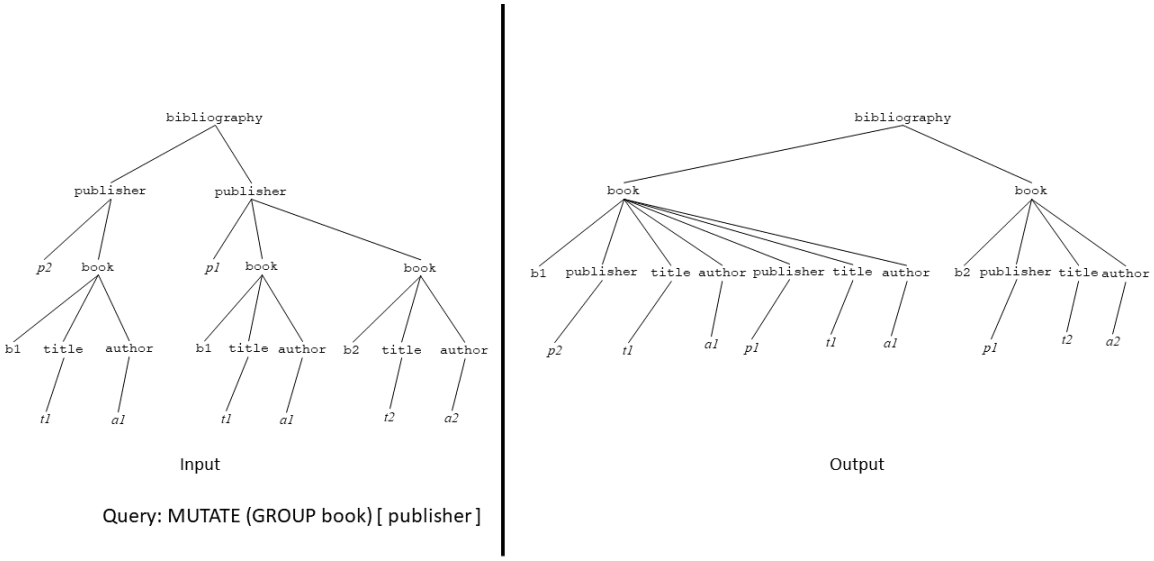

Fig. 2.2: XML Transformation without metadata

```
publisher
title
author
```

This hierarchy in the result is very important because now we will only use the closest distance metric on DLN along with this hierarchy to build resulting graph

2.2.4 Traversing over the stored input and build the graph

Now, we start to build the graph backed by two Hashmaps. first is edges, where we will store parent as key and list of child nodes. Second is groupMap, which has same node as key and value, but it is important since we do not want to add new parent for same grouped node in edges map. For example, for Figure 2.2, we do not want to add two b1 book nodes, in edges, rather we add more children to existing b1 book node. One key thing to note is that we build graph from bottom to top because we are calling this method by children of nodes first.

CHAPTER 3

Extending XMorph

From the example in chapter 1, we understood why modifications in XMorph are important so that it takes account of the metadata in a transformation. In this chapter, we describe the modifications that we made to XMorph to implement support for metadata. The modifications are listed below.

- Meta interface: This interface, defines the common functionalities every metadata exhibits. If We wanted to add support for any new metadata, all we need to do is just implement the methods of this interface and everything will be taken care of automatically. This is currently implemented by temporal and probabilistic metadata types.

- MetaDLN: To do transformation while taking metadata into account, we need a way to store metadata of each node as well. This data structure uses capabilities of old DLN class along with list of metadata at each node.

- Metadata generators: We need some way to add metadata to plain XML. Metadata generators are used to generate metadata for each node. We use this to add metadata to each node when we parse the XML.

3.1 New Data Structure

We have to add some new data structures to XMorph to implement support for metadata. This section describes the new data structures.

3.1.1 Meta Interface

We have created an interface, which defines the methods needed by any metadata. Below are the description of the interface methods.

- **union:** This method defines union of metadata with the other metadata.
- **intersection:** This method defines intersection of metadata with the other metadata.
- **addToTupleInput:** This method builds metadata from the input from the DB.
- **addToTupleOutput:** This method writes data to DB from the metadata information.
- **isEmpty:** This method determines, if node should exist based on the condition of metadata, for example, empty time in our case means, there is no valid time, a node can exist, so it should be removed from tree.
- **addMetaData:** This method is important in managing metadata at various stages in MetaDLN.
- **toString:** This method is used to print out the metadata information.

Below are the two implementations of the interface.

- **TimeElement:** This implementation is backed by Tree Set of Time Object(Time object has start time and end time as attributes and various methods necessary for it).
- **Probabilistic:** This implementation shows the probability of existence of node. It has probability as its attribute.

3.1.2 MetaDLN, underlying data structure behind an XML Transformation with Metadata

This class extends DLN class and has list of different kinds of metadata. So it is a DLN plus a list of metadata.

3.1.3 Metadata generators

These generators are useful because we can use them while parsing and test any XML input with metadata if no metadata was present before. We have also created metadata generators, which returns list of metadata with 4 options:

- static metadata: this gives us metadata of all time and probability 1
- random metadata: this gives us metadata of random time and probability between 0 and 1
- static metadata bounded by parent: this gives us new metadata object with same metadata information as the parent
- random metadata: this gives us metadata of random time and probability bounded by parent

3.2 Parsing and storing the input

We do parsing and storing similar to previous version of XMorph. The only additional thing we do is adding random or static metadata to each node. If it is root node, we generate metadata without need of parent, else we generate and add metadata to nodes bounded by parent.

The reason for that is that a node can not exist when its parent did not exist.

For storage, we have replaced DLN with MetaDLN because we need to store metadata information as well.

3.3 Compilation of Query and Generate the resulting shape nodes

We did not need to modify it, since metadata addition has nothing to do with the desired hierarchy of types of nodes.

This metadata information affects transformation only while we do actual transformation and build the output XML.

3.4 Traversing over the stored input and build the graph

we build the graph backed by two Hashmaps. first is edges, where we will store parent as key and list of child nodes as value. Second is groupMap, which has DLN and type information as key and grouped node as value.

One key thing to note is that we build graph from bottom to top because we are calling this method by children of nodes first.

In addition to those steps, we also have to make sure that metadata is right for each of the transformed nodes. We also get rid of nodes, which have empty metadata, because for example in case of temporal metadata, we can not have node, which does not have any lifetime or in other words empty temporal metadata.

Below are the steps, which show how we take care of metadata correctly.

1. We first clone the child node along with its edges. This step is very important because in XML transformation it is highly probable that a node becomes child of multiple other nodes in modified XML, this cloning saves the original metadata of original node and does force same metadata of all those child nodes.

2. We then intersect metadata of child with the parent and assign it to child, so that child's metadata is always less than or equal to parent's metadata. we do it before union of metadata of parent because in that case, we will lose sequence semantics or in other words, we want that much metadata to child node, which is related to that parent-child relationship.

3. We then check if parent or child is part of grouping node, if it is then we do union of metadata of this node with previously-stored grouping node.

4. We then check if this parent node is part of tree, if it is, then we just add new child node to previously present parent, else we add parent and then new child.

CHAPTER 4

Example of XML transformation in modified XMorph

In this chapter, we give a concrete example of a transformation with metadata. The example illustrates how our modified XMorph does XML transformation from input to output for a given grouping query as shown in Figure 4.1. We will see how we add edges to the modified tree structure of XML, and how the metadata is processed preserving its semantics. In this example, blue color shows parent node and red color shows child node being added in modified XML.

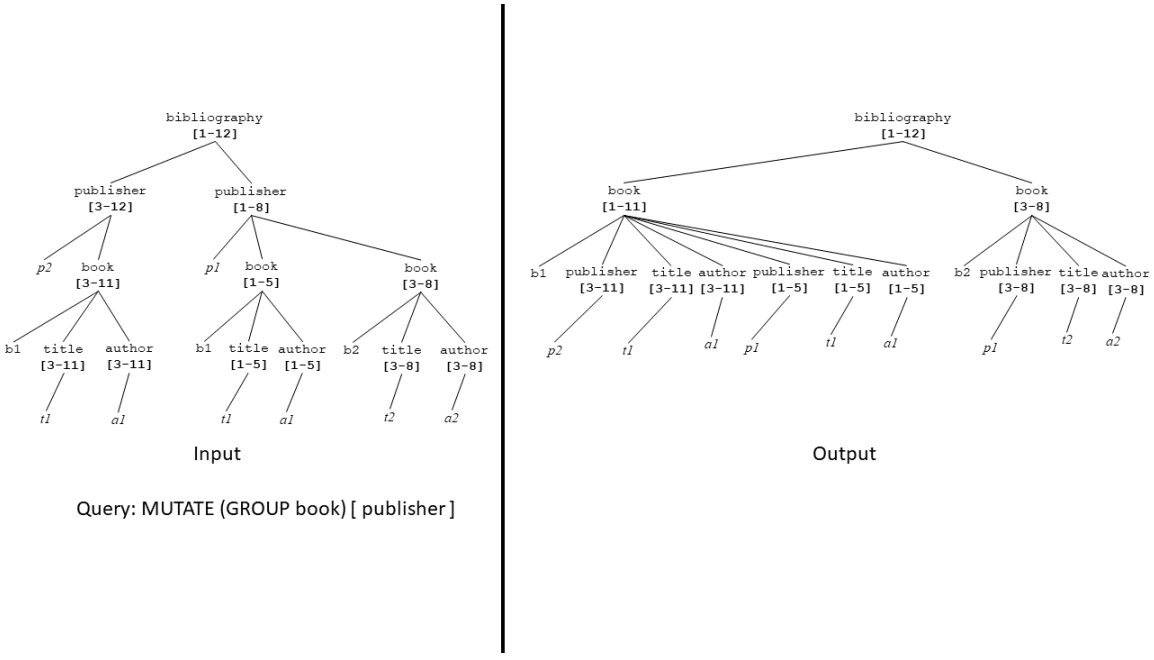

Fig. 4.1: Input and correct output for the given grouping query

In Figure 4.2, we start building our transformed XML. First, we make clone of child node p2. Then we do intersection of metadata of new child node from parent node and assign the resulting metadata to new child node. This way, our new child node will always have metadata smaller than or equal to parent's metadata. After that, we add this parent and new child node as parent-child in edges. We also put b1[3-11] as parent and child into

Grouped map, because book type was marked as grouping node when we created Query object from given query.

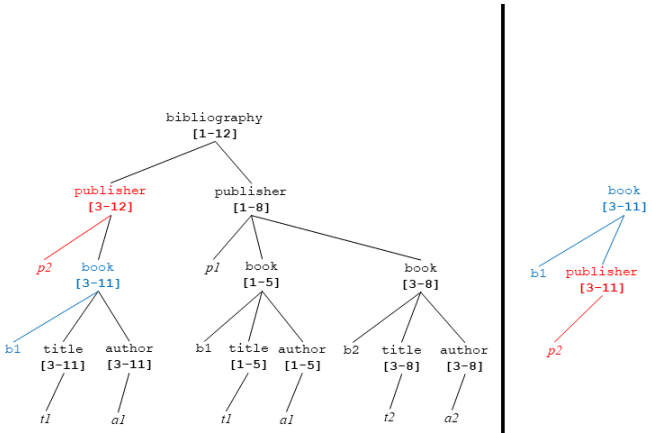

Fig. 4.2: Addition of b1[3-11] as parent and p2[3-12] as child

Then, in Figure 4.3, we first do cloning of child node. Then we do intersection of metadata of t1 and b1 and assign it to t1. After that, we check that if this parent has been added previously, and then we retrieve the list of children and adds this new child node to the list of children of that parent. In this way, we always add new child node to existing parent if it exists. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to parent node. IN this way our metadata for b1 will always be up to date.

Figure 4.4 shows cloning of a child node. Then we do intersection of metadata of a1 and b1 and assign it to a1. After that, we check that if this parent has been added previously, and then we add this newly child node to list of children of b1. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to parent node.

The next step is shown in Figure 4.5. This figure illustrates that we first do cloning

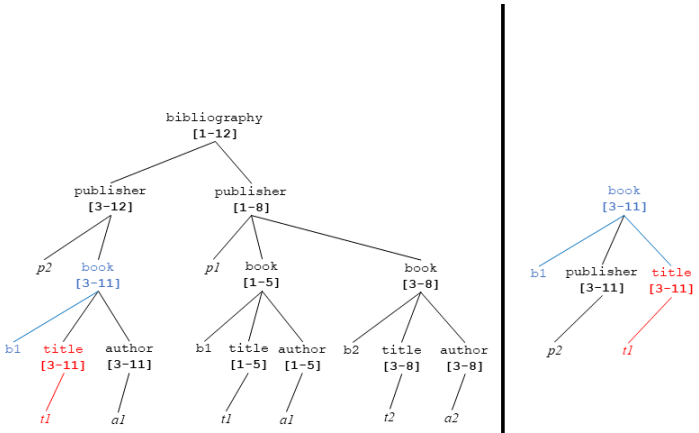

Fig. 4.3: Addition of b1[3-11] as parent and t1[3-11] as child

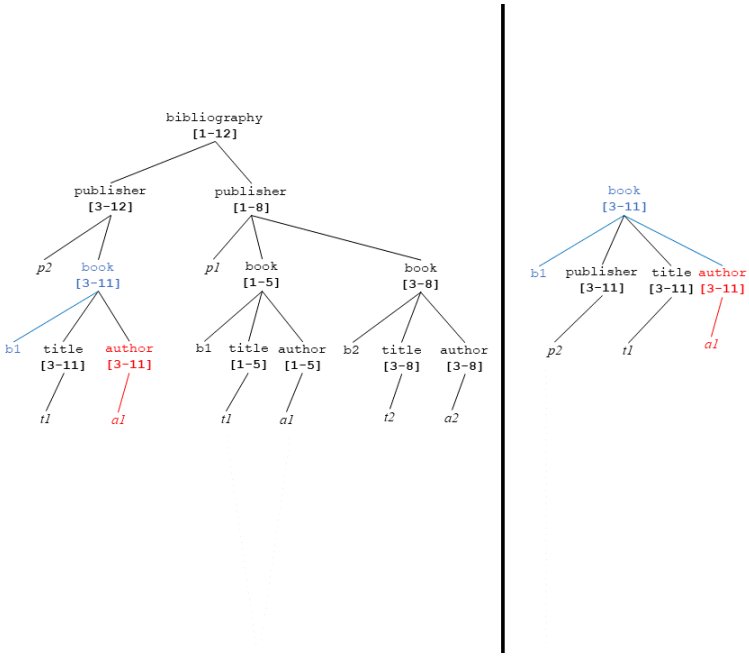

Fig. 4.4: Addition of b1[3-11] as parent and a1[3-11] as child

of child node. Here one more thing to notice is that if we only do cloning of child node, then we will miss the already added children of b1. To avoid that, we always check that if that child node has any children, if they exist, we add this new child as parent and those children as children of this new child node in our edges map. In this, way we do not lose children of the b1 in case of cloning.

Then we do intersection of metadata of bib and b1 and assign it to b1. After that, we check that if this parent has been added previously. In this case, it does not, so we simply add this parent and new child node as parent-child in edges. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to child node.

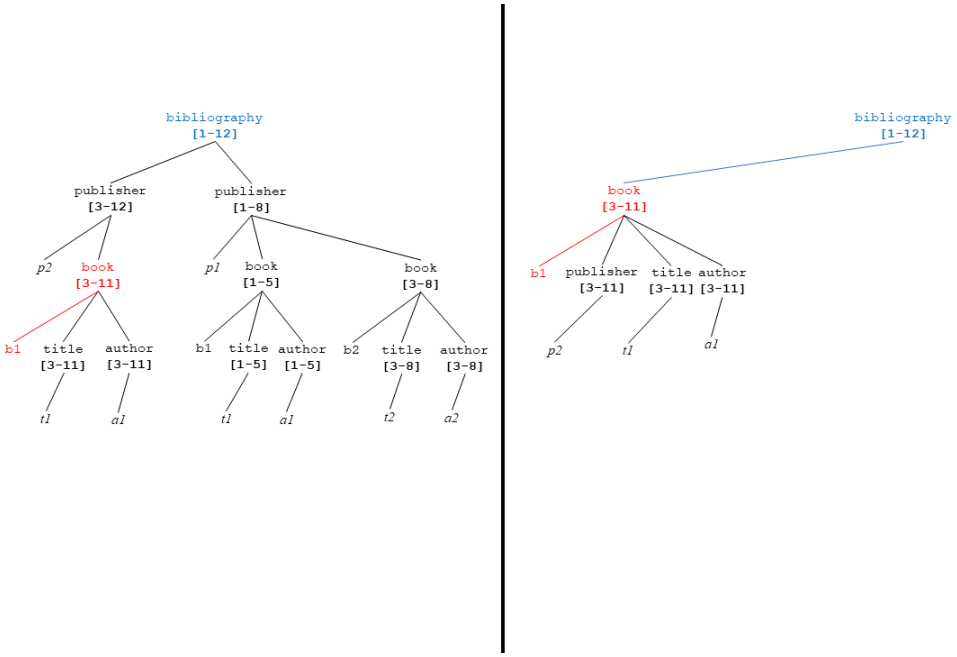

Fig. 4.5: Addition of bibliography[1-12] as parent and b1[3-11] as child

In Figure 4.6, we first do cloning of p1 and then we do intersection of metadata of p1 and b1 and assign it to p1. Notice that in this situation, metadata of original p1 node is not affected as we do intersection and assignment on new child node.

Preservation of original child node p1 is important here as, we will see that p1 will become child of b2 book, also. So, we will need original metadata of p1 node. This is

the main reason, why we were doing cloning of child node every time because, in XML transformation, we never know when a child node will become child node of 2 or more nodes.

After that, we do union of metadata of previously-stored grouped node b1[3-11] and b1[1-5], which eventually makes its metadata [1-11]. Then, we add p1[1-5] to b1[1-11].

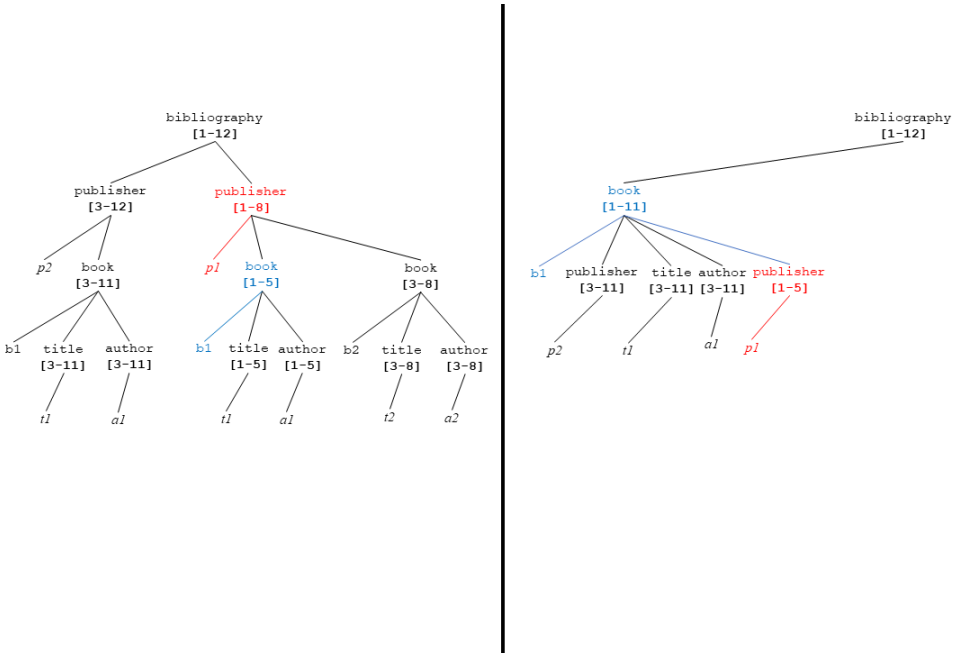

Fig. 4.6: Addition of b1[1-5] as parent and p1[1-8] as child

In Figure 4.7, we first do cloning of child node. Then we do intersection of metadata of t1 and b1 and assign it to t1. After that, we check that if this parent has been added previously, and then we add this newly child node to list of children of b1. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to parent node.

Then, in Figure 4.8, we first do cloning of child node. Then we do intersection of metadata of a1 and b1 and assign it to a1. After that, we check that if this parent has been added previously, and then we add this newly child node to list of children of b1. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to parent node.

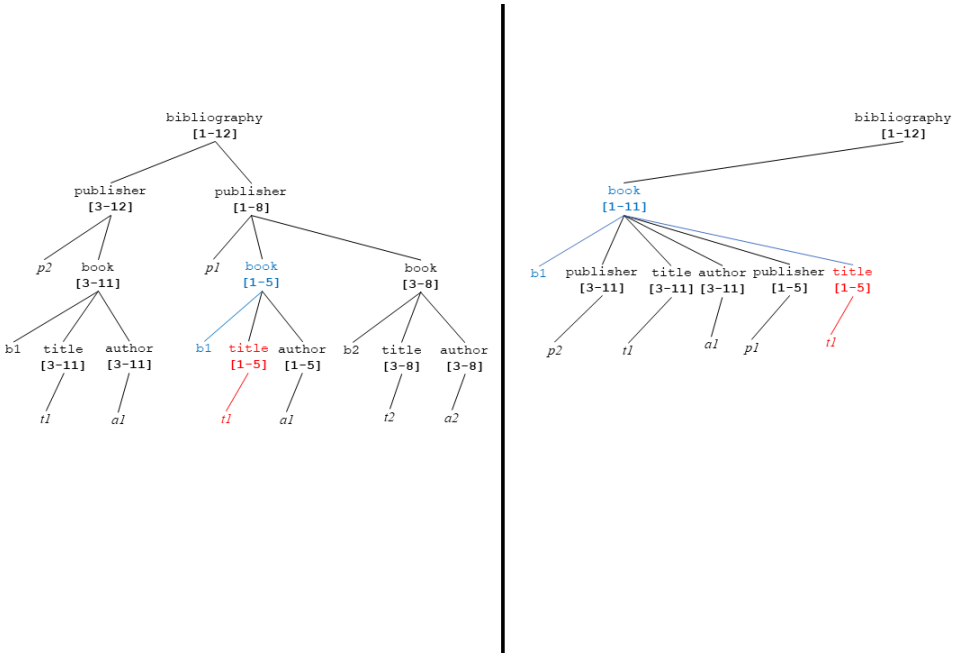

Fig. 4.7: Addition of b1[1-5] as parent and t1[1-5] as child

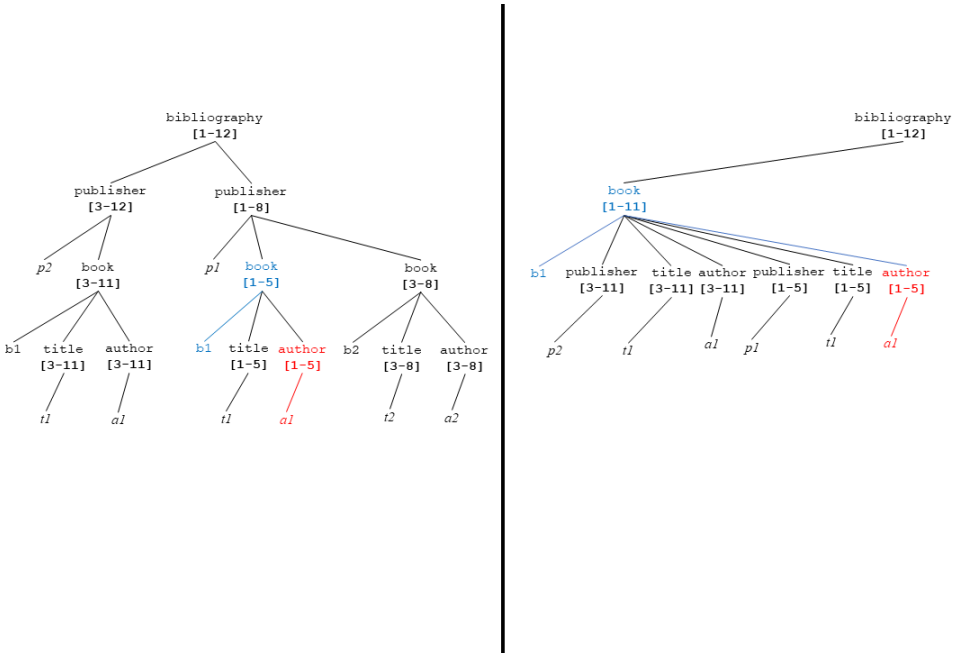

Fig. 4.8: Addition of b1[1-5] as parent and a1[1-5] as child

In Figure 4.9, we first do cloning of child node. Then, we check that if that child node has any children, if they exist, we add this new child as parent and those children as children of this new child node in our edges map. In this, way we do not lose children of the b1 in case of cloning.

Then we do intersection of metadata of bib and b1 and assign it to b1. After that, we check that if this parent has been added previously. In this case, it is, so we simply add this parent and new child node as parent-child in edges. After, that we also do union of metadata of previously-stored grouped b1 node and this b1 node and assign it to child node.

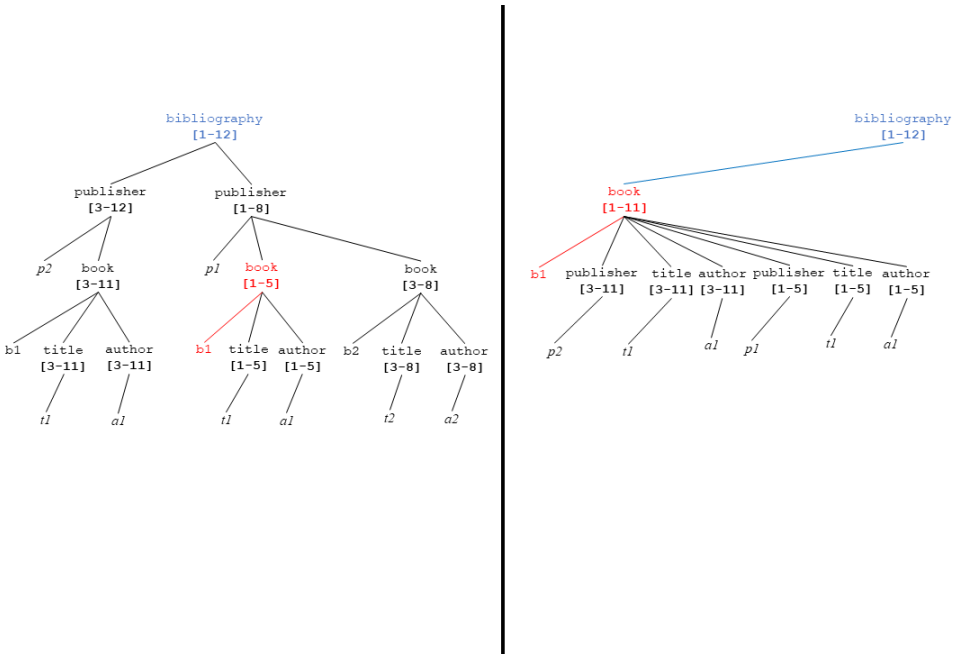

Fig. 4.9: Addition of bibliography[1-12] as parent and b1[1-5] as child

In Figure 4.10, First, we make clone of child node p1. Here, we can see and understand that if we did not make clone of p1 before, then we will take p1[1-5] as child rather than p1[1-8].

Then we do intersection of metadata of new child node p1 from b1 and assign the resulting metadata to new p1 node. After that, we add this parent and new child node as parent-child in edges. We also put b2[3-8] as parent and child into Grouped map, because

book type was marked as grouping node when we created Query object from given query.

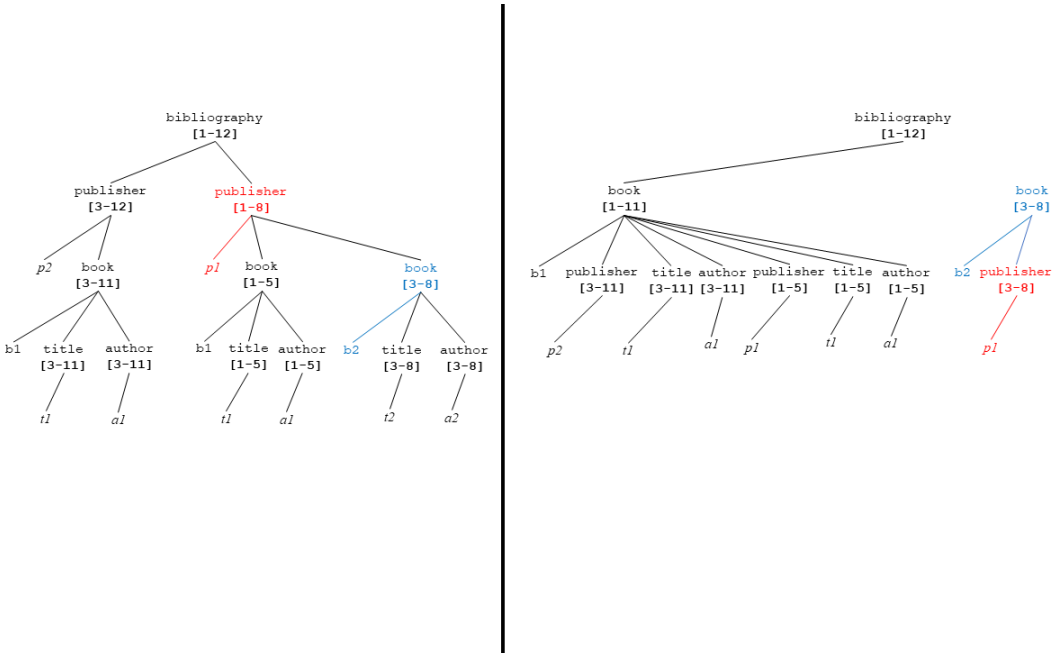

Fig. 4.10: Addition of b2[3-8] as parent and p1[1-8] as child

In Figure 4.11, we first do cloning of child node. Then we do intersection of metadata of t2 and b2 and assign it to t2. After that, we check that if this parent has been added previously, and then we add this newly child node to list of children of b2. After, that we also do union of metadata of previously-stored grouped b2 node and this b2 node and assign it to parent node.

In Figure 4.12, we first do cloning of child node. Then we do intersection of metadata of a2 and b2 and assign it to a2. After that, we check that if this parent has been added previously, and then we add this newly child node to list of children of b2. After, that we also do union of metadata of previously-stored grouped b2 node and this b2 node and assign it to parent node.

In Figure 4.13, we first do cloning of child node. Then, we check that if that child node has any children, if they exist, we add this new child as parent and those children as children of this new child node in our edges map. In this, way we do not lose children of the b2 in case of cloning.

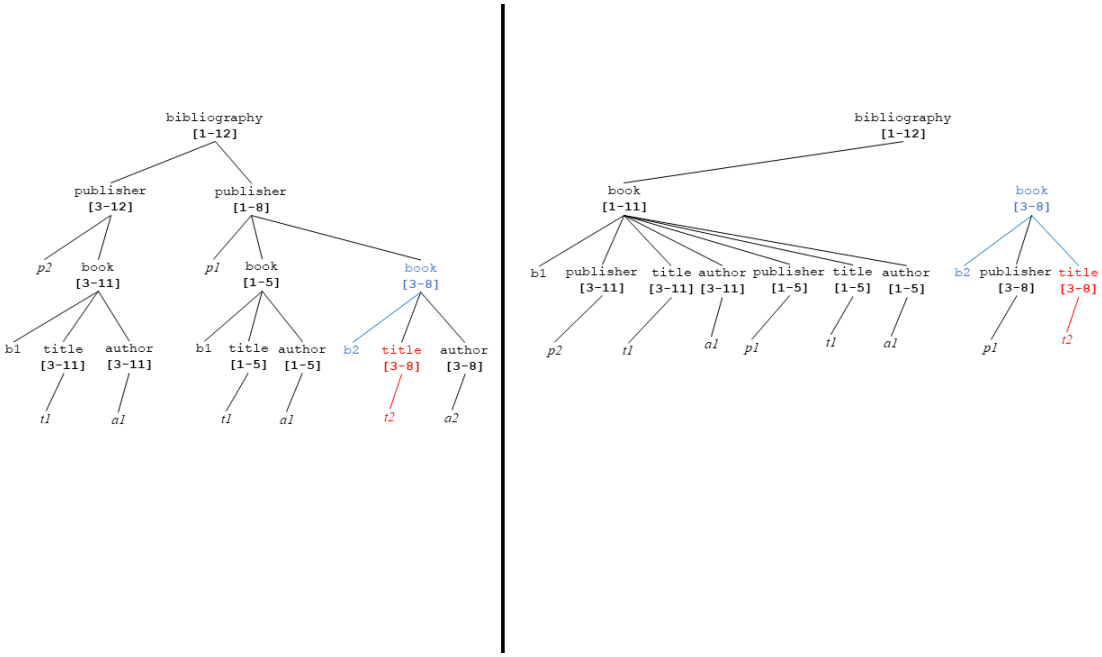

Fig. 4.11: Addition of b2[3-8] as parent and t2[3-8] as child

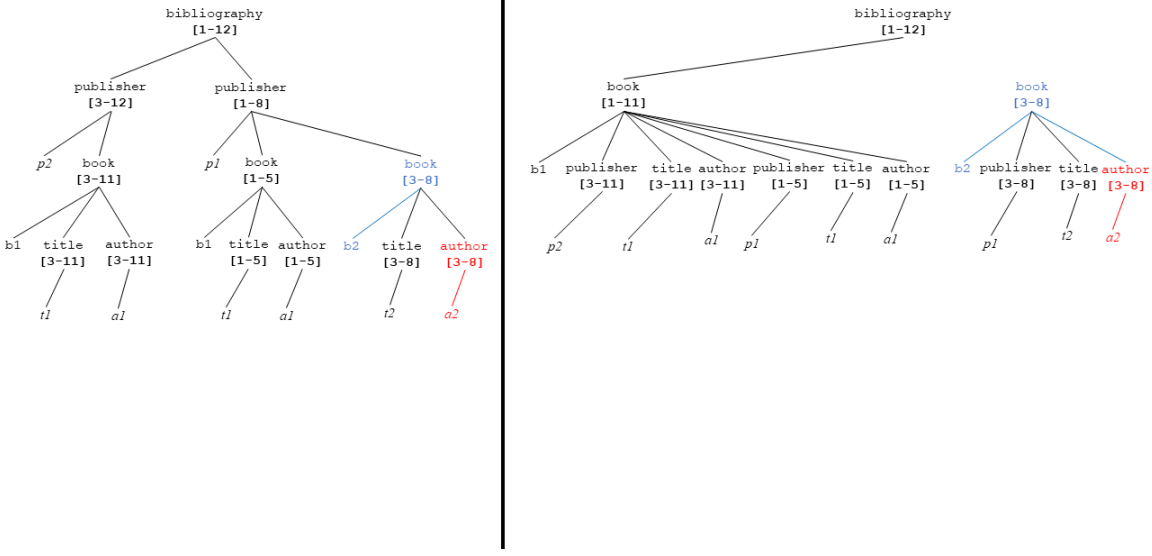

Fig. 4.12: Addition of b2[3-8] as parent and a2[3-8] as child

Then we do intersection of metadata of bib and b2 and assign it to b2. After that, we check that if this parent has been added previously. In this case, it does not, so we add this parent and new child node as parent-child in edges. After, that we also do union of metadata of previously-stored grouped b2 node and this b2 node and assign it to child node.

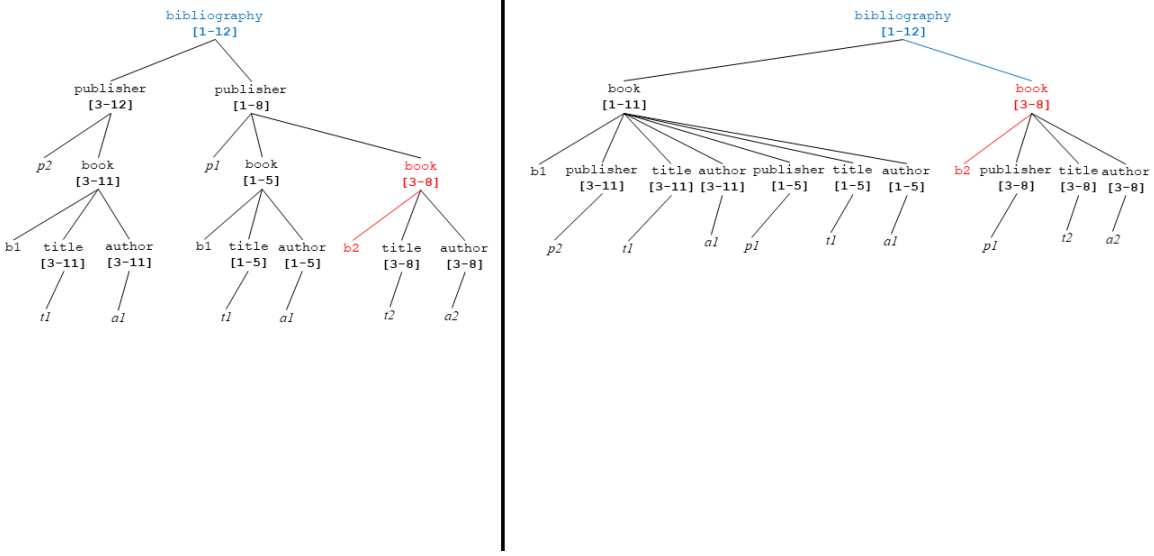

Fig. 4.13: Addition of bibliography[1-12] as parent and b2[3-8] as child

In this way, we complete our XML transformation with metadata.

CHAPTER 5

Experiments

We implemented XMorph in Java. The implementation uses ANTLR for the parser, a Xerces SAX parser for the data shredding, and BerkeleyDB Java edition for the data store. To quantify cost of experiments of a data transformation we performed several experiments. The experiments were run on Quad Processor Windows machine, with 4 intel 8565U 1.8GHZ chips, 16 GB of RAM. In each experiment, we isolated the testing machine. We ran each experiment ten times and used the mean cost. The cache was cleared for each run so the timings are "cold cache" numbers in every experiment.

XML Generation for experiment:

We generated Sample XML data as shown in fig 5.1 to test XMorph, using classes of javax.xml. We used number of employees parameters to change file size as per requirement. We also generated different employees by randomly selecting different firstname, lastname, and department from their respective lists. We kept increasing length of departments list so that when we group this data on department, we don't have much change for number of employees/department. This parameter is important for comparing the timing of grouping transformation so that we compare time of same query on similar data while performing the experiments.

Metadata generation and addition to generated XML: We generated metadata in 2 ways, with parent or without parent. To add metadata to XML correctly, we add metadata as we parse the XML. Let's see how it works with example shown in fig 5.2. First, when we parse Company element, we generate metadata randomly without parent and assign it to MetaDLN node created for this node. After that when we parse employee with id=1 as child of company, we pass parents metadata to our metadata generators, so that we get random metadata confined within parents metadata. We assign the metadata returned to MetaDLN node created for this node. IN this way, we do metadata generation

```
<company>
  <employee id="1">
    <firstname>Shivam</firstname>
    <lastname>Swami</lastname>
    <department>R</department>
  </employee>
  <employee id="2">
    <firstname>Shivam</firstname>
    <lastname>Falor</lastname>
    <department>R</department>
  </employee>
  <employee id="3">
    <firstname>Erik</firstname>
    <lastname>Falor</lastname>
    <department>S</department>
  </employee>
  <employee id="4">
    <firstname>Erik</firstname>
    <lastname>Dyreson</lastname>
    <department>S</department>
  </employee>
</company>
```

Fig. 5.1: sample input without meta data

and add it to their respective nodes.

Queries ran on sample input: To understand time taken by modified XMorph, We ran following 2 queries on our generated XML.

1. MUTATE firstname [(CLONE lastname)]
2. MUTATE (GROUP department) [employee]

The first query mutates input XML such that last name becomes child of first name. These kinds of queries are just rearrangements of input XML with some intersection and take less time to execute. Fig 5.3 shows the result of the first query ran on sample input. Fig 5.5 shows the timing result of first query ran on input XML of various sizes of same schema.

The second query groups input data on department and then add employees as its children. These kinds of grouping transformations are bit slower because we have to do some extra operations(union and cloning), while we do transformation. Fig 5.4 shows the result of the second query ran on sample input. Fig 5.6 shows the timing result of second query ran on input XML of various sizes of same schema.

```
▼<company TimeElement="[5, 15]" Probabilistic="0.6">
  ▼<employee id="1" TimeElement="[5, 15]" Probabilistic="0.5">
     <firstname TimeElement="[5, 15]" Probabilistic="0.5">Shivam</firstname>
     <lastname TimeElement="[5, 15]" Probabilistic="0.5">Swami</lastname>
     <department TimeElement="[5, 10]" Probabilistic="0.25">R</department>
   </employee>
  ▼<employee id="2" TimeElement="[10, 15]" Probabilistic="0.4">
     <firstname TimeElement="[10, 13]" Probabilistic="0.3">Shivam</firstname>
     <lastname TimeElement="[10, 12]" Probabilistic="0.2">Falor</lastname>
     <department TimeElement="[10, 15]" Probabilistic="0.4">R</department>
   </employee>
  ▼<employee id="3" TimeElement="[5, 15]" Probabilistic="0.6">
     <firstname TimeElement="[5, 13]" Probabilistic="0.6">Erik</firstname>
     <lastname TimeElement="[11, 12]" Probabilistic="0.6">Falor</lastname>
     <department TimeElement="[10, 10]" Probabilistic="0.6">S</department>
   </employee>
  ▼<employee id="4" TimeElement="[7, 8]" Probabilistic="0.5">
     <firstname TimeElement="[7, 7]" Probabilistic="0.4">Erik</firstname>
     <lastname TimeElement="[7, 8]" Probabilistic="0.45">Dyreson</lastname>
     <department TimeElement="[8, 8]" Probabilistic="0.3">S</department>
   </employee>
 </company>
```

Fig. 5.2: sample input with meta data

Cost of transformation vs data size: To understand the extra overhead added by transformation with metadata, we calculated the time taken by both XMorphMeta and XMorph on various file sizes.

From both Figure 5.5 and Figure 5.6, we can see that overhead added by inclusion of metadata in transformation is very modest. XMorphMeta also exhibits almost linear relationship of time vs file size, which will be very good as soon as we will start to increase file size of XML.

```
▼<company TimeElement="[5, 15]" Probabilistic="0.6">
  ▼<employee id="1" TimeElement="[5, 15]" Probabilistic="0.5">
    ▼<firstname TimeElement="[5, 15]" Probabilistic="0.5">
       Shivam
       <lastname TimeElement="[5, 15]" Probabilistic="0.5">Swami</lastname>
     </firstname>
     <department TimeElement="[5, 10]" Probabilistic="0.25">R</department>
   </employee>
  ▼<employee id="2" TimeElement="[10, 15]" Probabilistic="0.4">
    ▼<firstname TimeElement="[10, 13]" Probabilistic="0.3">
       Shivam
       <lastname TimeElement="[10, 12]" Probabilistic="0.2">Falor</lastname>
     </firstname>
     <department TimeElement="[10, 15]" Probabilistic="0.4">R</department>
   </employee>
  ▼<employee id="3" TimeElement="[5, 15]" Probabilistic="0.6">
    ▼<firstname TimeElement="[5, 13]" Probabilistic="0.6">
       Erik
       <lastname TimeElement="[11, 12]" Probabilistic="0.6">Falor</lastname>
     </firstname>
     <department TimeElement="[10, 10]" Probabilistic="0.6">S</department>
   </employee>
  ▼<employee id="4" TimeElement="[7, 8]" Probabilistic="0.5">
    ▼<firstname TimeElement="[7, 7]" Probabilistic="0.4">
       Erik
       <lastname TimeElement="[7, 7]" Probabilistic="0.4">Dyreson</lastname>
     </firstname>
     <department TimeElement="[8, 8]" Probabilistic="0.3">S</department>
   </employee>
 </company>
```

Fig. 5.3: Mutate Query Result on sample input

```
▼<company TimeElement="[5, 15]" Probabilistic="0.6">
  ▼<department TimeElement="[5, 15]" Probabilistic="0.4">
     R
    ▼<employee id="1" TimeElement="[5, 10]" Probabilistic="0.25">
       <firstname TimeElement="[5, 10]" Probabilistic="0.25">Shivam</firstname>
       <lastname TimeElement="[5, 10]" Probabilistic="0.25">Swami</lastname>
     </employee>
    ▼<employee id="2" TimeElement="[10, 15]" Probabilistic="0.4">
       <firstname TimeElement="[10, 13]" Probabilistic="0.3">Shivam</firstname>
       <lastname TimeElement="[10, 12]" Probabilistic="0.2">Falor</lastname>
     </employee>
   </department>
  ▼<department TimeElement="[8, 8] [10, 10]" Probabilistic="0.6">
     S
    ▼<employee id="3" TimeElement="[10, 10]" Probabilistic="0.6">
       <firstname TimeElement="[10, 10]" Probabilistic="0.6">Erik</firstname>
     </employee>
    ▼<employee id="4" TimeElement="[8, 8]" Probabilistic="0.5">
       <lastname TimeElement="[8, 8]" Probabilistic="0.45">Dyreson</lastname>
     </employee>
   </department>
 </company>
```

Fig. 5.4: Grouping Query Result on sample input

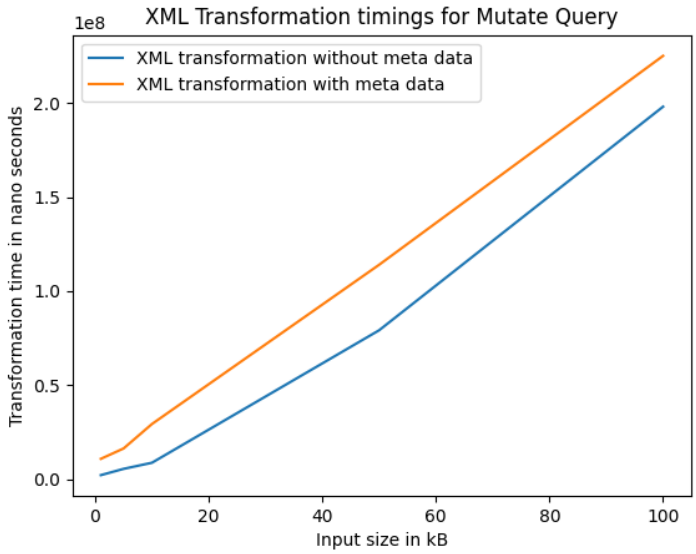

Fig. 5.5: XMorph vs XMorphMeta timing results on various file sizes in case of normal transformation

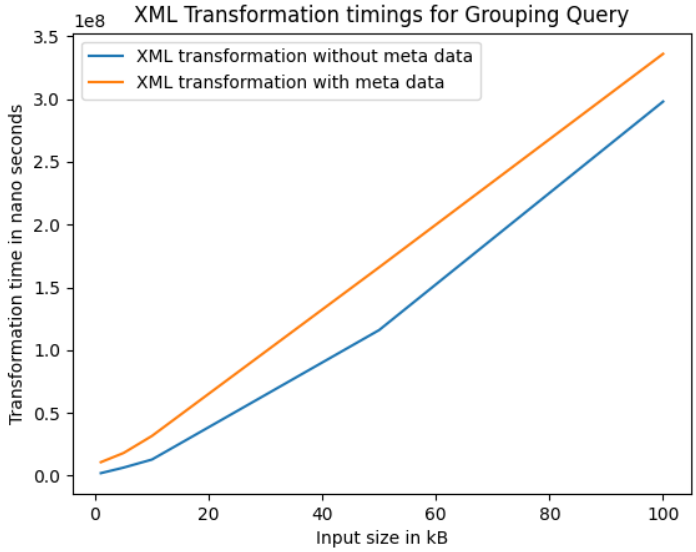

Fig. 5.6: XMorph vs XMorphMeta timing results on various file sizes in case of grouping transformation

CHAPTER 6

Conclusions and Future Work

In this thesis first, we implemented an algorithm for correctly transforming hierarchical data annotated with metadata. The challenge in supporting such a transformation is preserving the semantics of the metadata. We showed how a single operation within a metadata definition could be utilized to correctly preserve the semantics. Second, we described the changes to the XMorph data transformation system that are needed to correctly store and retrieve the metadata associated with each node. We extended the core object, a DLN object, to include metadata creating a MetaDLN object. We implemented the serialization and deserialization of the object to the database. We extended the parser to process and calculate metadata as the data is input. Third, we implemented metadata-specific transformations for two kinds of metadata: time and probability. These are common kinds of metadata in databases and we showed how the special semantics of each kind could be supported. Fourth and most importantly we evaluated the implementation to demonstrate that it adds only a modest overhead.

Future work is to utilize the metadata to drive a transformation. An example is to transform a temporal hierarchy into a hierarchy that identifies versions of each node in the data. Such a transformation would push the metadata into the data, for instance by embedding the timestamps as node attributes. The second line of future work is to improve parsing. As we parse the XML, We generate metadata randomly or statically restricted by parent node. In future work, we can envision a format for metadata that is input and parsed along with the data. Finally, we could implement yet other kinds of metadata such as provenance and security/privacy.

Bibliography

[1] W. C. McGee, "The Information Management System IMS/VS Part I: General Structure and Operation," *IBM Systems Journal*, vol. 16, no. 2, pp. 84–95, 1977.

[2] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A native XML database," *VLDB J.*, vol. 11, no. 4, pp. 274–291, 2002.

[3] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon, "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*, 2014, pp. 1247–1258.

[4] D. Tahara, T. Diamond, and D. J. Abadi, "Sinew: a SQL System for Multi-Structured Data," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 815–826.

[5] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Commun. ACM*, vol. 54, no. 6, pp. 114–123, 2011.

[6] C. E. Dyreson and S. S. Bhowmick, "Plug-and-play queries for temporal data sockets," in *Flexible Query Answering Systems - 12th International Conference, FQAS 2017, London, UK, June 21-22, 2017, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10333. Springer, 2017, pp. 124–136. [Online]. Available: https://doi.org/10.1007/978-3-319-59692-1_11

[7] C. Dyreson, S. Bhowmick, A. Jannu, K. Mallampalli, and S. Zhang, "XMorph: A Shape-polymorphic, Domain-specific XML Data Transformation Language," in *ICDE*, 2010, pp. 844–847.

[8] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli, "Using XMorph to Transform XML Data," *PVLDB*, vol. 3, no. 2, pp. 1541–1544, 2010.

[9] R. T. Snodgrass, Ed., *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[10] C. Dyreson and S. Zhang, "The Benefits of Utilizing Closeness in XML," in *DEXA Work.*, 2008, pp. 269–273.

[11] C. E. Dyreson and S. S. Bhowmick, "Querying XML Data: As You Shape It," in *ICDE*, 2012, pp. 642–653.

[12] B. Q. Truong, S. S. Bhowmick, and C. E. Dyreson, "SINBAD: Towards Structure-Independent Querying of Common Neighbors in XML Databases," in *DASFAA (1)*, 2012, pp. 156–171.

[13] C. E. Dyreson, S. S. Bhowmick, and R. Grapp, "Querying virtual hierarchies using virtual prefix-based numbers," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 791–802.

[14] ——, "Virtual exist-db: Liberating hierarchical queries from the shackles of access path dependence," *PVLDB*, vol. 8, no. 12, pp. 1932–1943, 2015.

[15] C. Li and T. W. Ling, "An Improved Prefix Labeling Scheme: A Binary String Approach for Dynamic Ordered XML," in *DASFAA*, 2005, pp. 125–137.

[16] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang, "Storing and Querying Ordered XML using a Relational Database System," in *SIGMOD Conference*, 2002, pp. 204–215.

[17] J. X. Yu, D. Luo, X. Meng, and H. Lu, "Dynamically Updating XML Data: Numbering Scheme Revisited," *World Wide Web*, vol. 8, no. 1, pp. 5–26, 2005.

[18] T. Böhme and E. Rahm, "Supporting Efficient Streaming and Insertion of XML Data in RDBMS," in *DIWeb*, 2004, pp. 70–81.

[19] G. Gou and R. Chirkova, "Efficiently Querying Large XML Data Repositories: A Survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 10, pp. 1381–1403, 2007.

[20] H.-K. Ko and S. Lee, "A binary string approach for updates in dynamic ordered xml data," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 4, pp. 602–607, 2010.