# Querying Virtual Hierarchies using Virtual Prefix-Based Numbers

Curtis E. Dyreson
Dept. of Computer Science
Utah State University, USA
Curtis.Dyreson@usu.edu

Sourav S. Bhowmick
School of Computer Engineering
Nanyang Technological University, Singapore
assourav@ntu.edu.sg

Ryan Grapp
Dept. of Computer Science
Utah State University, USA
Ryan.Grapp@aggiemail.usu.edu

## ABSTRACT

*Prefix-based numbering* (*PBN*) is a popular method for numbering nodes in a hierarchy. But PBN breaks down when a node's location within a hierarchy changes, such as when XML data is queried after being transformed by an XSLT program or when data is reformatted in the `return` clause of an inner FLWR expression in a nested XQuery program. A query on *transformed data* cannot be evaluated as efficiently since the extant PBN numbers cannot be used (unless the data is materialized and then renumbered, which can be expensive). In this paper we present a novel strategy to *virtually transform* the data without instantiating and renumbering. Our method, which we call *virtual prefix-based numbering*, couples each PBN number with a *level array* that locates the node in the numbering space of the virtual hierarchy. The virtual numbering space preserves the property that location-based relationships between nodes can be determined by comparing (virtual) numbers.

## Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing - XML*

## General Terms

Design, Languages

## Keywords

XML; virtual hierarchy; prefix-based numbering; view query

## 1. INTRODUCTION

About 40 years ago, E. F. Codd observed that the hierarchical data model has a problem: queries must use *path expressions* to locate data in the hierarchy, thereby *tightly coupling* queries to hierarchies [5]. Codd explained that there are myriad natural hierarchies for any tree-like data collection and that tightly coupling path expressions to just one hierarchy (or even a subset of the potential hierarchies) prevents queries from being ported to new collections that have similar data organized in a different hierarchy, and also increases the cognitive burden on query writers since they have to know the hierarchy to write queries. Moreover, a query on data with a hierarchy different than what the query expects will not always fail with an obvious error, rather the query usually will run to completion and yield an empty or partial answer.

Today, path expressions are ubiquitous since data continues to be organized in hierarchies in many areas of computer science, e.g., file systems, the semantic web, and databases. Databases, in particular, have embraced the management and querying of data represented in JSON or XML, which has a hierarchical data model and is the focus of this paper.

To solve the problem he posed, Codd changed the data model, but three other distinct solutions have been researched.

1. *Rewrite the data* - Physically transform the data to the desired hierarchy [7, 16, 23]. But it can be excessively expensive to transform a data collection, especially when a query uses only a fraction of the data.

2. *Rewrite the query* - Evaluate the query through a (data transformation) view [1, 9, 14, 21, 22, 24]. The chief drawback is that a view is specific to a hierarchy, so each hierarchy needs its own view.

3. *Reinterpret the query* - Relax or change the query to explore a range of "close" hierarchies [2, 6, 13, 19, 20, 29]. But since query evaluation does not transform the hierarchy, the result is formatted in the source data's hierarchy.

In this paper, we propose a new approach: *reinterpret the data*. We develop a reinterpretation technique using a popular node numbering system for hierarchical data called *prefix-based numbering* (*PBN*) (also called *containment encoding*, *Dewey order*, *Dewey numbering*, and *Dynamic-level numbering*) [4, 11, 12, 15, 17, 26, 30]. PBN is primarily used in XML management systems to support fast XQuery evaluation and efficient XML search [25], but can be used for any hierarchical model. In our approach a user sketches a *virtual hierarchy* for the data. Data is interpreted to be located where it is specified in the virtual hierarchy; the data itself is not physically moved. This paper makes the following contributions.

- We introduce a new numbering system for virtual hierarchies called *virtual prefix-based numbering* (*vPBN*). A *v*PBN number can be moved to a new location within a hierarchy, yet be used just like a PBN number to determine location-based relationships in the context of the new location. In effect, *v*PBN


*SIGMOD'14,* June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610506.

*virtually* rather than *physically* transforms data and supports query evaluation in the transformed data space.

- We describe how to construct *v*PBN numbers. A *v*PBN number combines a PBN number with a *level array* that locates it in the transformed space. We give an algorithm for computing level arrays. A transformation can also change a node's *value*, *i.e.,* the subtree rooted at the node, and we show how to compute the transformed value.
- We add a *virtualDoc* function to XQuery to allow the specification of a virtual hierarchy.
- We both analyze and empirically measure the cost of *v*PBN, and show the cost to be modest.

## 2. TRANSFORMING DATA BREAKS PBN

The essence of PBN is that it encodes the hierarchy in the numbering. A node is numbered by combining its parent's number (as a prefix) with the ordinal of the node's position in sibling order. The primary benefit of PBN is that location-based relationships between nodes (*e.g.,* whether a node is a descendant of another) can be determined from just the numbers. This is critical to fast query evaluation since nearly all queries involve a query language *axis* (*e.g.,* ancestor, descendant, child, etc.) to locate nodes. While encoding the hierarchy in the numbering makes PBN ideal for quickly and easily determining how nodes are related, if the hierarchy changes, for instance in a nested XQuery query when an outer FLWR expression traverses the data produced by an inner FLWR expression, the numbering breaks down in two critical ways.

1. **Location-based relationships change** - The PBN number for a node is rendered obsolete when a node is moved to a new location in a hierarchy. For instance the ancestor-descendant relationship between a pair of nodes could be inverted during a transformation. In the transformed instance the node that moves from being a descendant to an ancestor would **not** have a node number that is a prefix of its new descendant (previously its ancestor).
2. **Values change** - The value of a node in the hierarchy is built from its descendants, *i.e.,* every node which is prefixed with the node's number is part of the node's value. But this property does not hold for transformed data. A node that was previously a child may move away, a non-child may move to become a child, or the order of the children may change. These kinds of changes alter the value of a node.

As a concrete example assume that Sam writes an XQuery query to list for each book, its title and the list of authors for that book. The query is shown in Figure 1. The `return` clause in the query relates a `<title>`, `$t`, with a list of `<author>`s, `$a`, through a `<book>` ancestor. When Sam runs his query on the XML data model instance shown in Figure 2, the query will produce the data model instance shown in Figure 3.

Sam's query is a kind of *data transformation*, *i.e.,* it transforms the data into a new hierarchy. The new hierarchy is given in the `return` clause where `<author>` elements are placed as children of `<title>` elements, as long as the authors are related to the title through a (least common) `<book>` ancestor.

Suppose that Rhonda wants to count the number of authors for each title. She would like to reuse Sam's query because it makes it easier for her to write the query to compute the count. For instance, Rhonda could embed Sam's query as an "inner" query in a nested query as shown in Figure 4. Alternatively, Rhonda could use Sam's

```
for $t in doc("book.xml")//book/title
let $a := $t/../author
return <title>{$t/text()} {$a} </title>
```

Figure 1: Sam's query to list the authors for each title

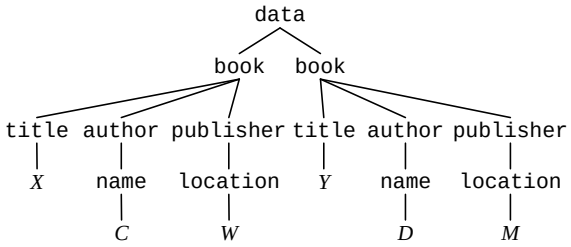



Figure 2: Input data model instance for Sam's query

query as a *view*, and use a query rewriting technique to combine her query with the view [1, 9, 14, 21, 22, 24].

There are three complications with writing views. First, **data transformation views can be complex**. It is surprisingly difficult and tedious to express even simple data transformations in XQuery as either a view or nested query. The culprit is the transformed *value*, which must be meticulously constructed element by element in a query's `return` clause. Consider for instance a query to transform the data to be the *other* `<book>` information (excluding `<title>` and `<author>`). To do so, one would have to build a list of a `<book>`'s children, remove from that list `<title>` and `<author>` nodes, place the nodes in the resulting list as children of a `<book>`, and then capture the attributes of a `<book>` by building a string that looks like the `<book>` element by concatenating all of the pieces as shown in Figure 5. And this assumes that none of the children of a `<book>` have transformed values! When used as a view and combined with a query a query rewriting strategy will produce a long, complicated query that is difficult to optimize. Second, **data transformations construct new node types**. Often a data transformation will need to change the hierarchy rooted at a node. It can only do so by explicitly constructing the node's value piece-by-piece. The resulting node is a *new* type, one not present in the original data. For example, the `<title>` element in Figure 1 is a constructed element, and is distinct from any `<title>` element originally in the data. These new elements make it problematic for a query rewriting technique to *push* `WHERE` clause constraints (*e.g.,* choose books where the book publisher is "Addison-Wesley") from an outer query into the view query. Third, **data transformation views are tightly-coupled to a hierarchy** so each hierarchy potentially needs a different view.

Another option is to express Sam's query in a dedicated data transformation language [7, 16, 23]. Rhonda can the then write her query to use the result of the transformation. The problem with this option is that it needs (at least) *two passes over the data*, one to transform the data and a second to evaluate the query on the transformed data. This strategy is inefficient for large data collections when a query uses only a small portion of the transformed data.

We propose a new strategy in this paper that *virtually* transforms the data to a desired hierarchy. We illustrate our new approach with an example. A virtual data transformation can be expressed using a *structural summary*, *DataGuide* [10], or XMORPH program [7] to describe the desired (virtual) hierarchy for the data. The DataGuide for the result of Sam's transformation is specified below. In the DataGuide the children of an element type are listed within braces.

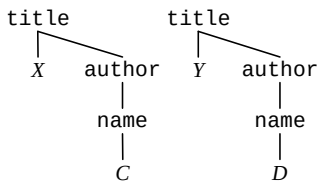



Figure 3: Output data model instance for Sam's query

```
for $t in (...Sam's query...)//title
return <title>{$t/text()}
         {count($t/author)}
       </title>
```

Figure 4: Rhonda's query to count the authors for each title

```
title { author { name } }
```

The specification is a *virtual DataGuide* (*v*DataGuide) since it describes the desired hierarchy rather than the data's physical hierarchy. In the virtual hierarchy <title> elements contain <author> children, and <name> grandchildren.

To use a *v*DataGuide in an XQuery query, we introduce a *virtualDoc* function which is like the *doc* function but includes the *v*DataGuide specification as a string parameter. The rest of the query is logically evaluated with respect to the *v*DataGuide. An example is shown in Figure 6. The query specifies that Rhonda's query is to be evaluated on the document described by the *v*DataGuide for the result of Sam's query. No data is physically transformed, only the hierarchy of the data is changed so that nodes appear in the location they should be *after* the transformation. The query is subsequently evaluated in the transformed space. This allows query writers to craft queries using any *v*DataGuide they desire. The challenge addressed in this paper is *how to efficiently evaluate a query on data described by a vDataGuide*.

# 3. RELATED WORK

Previous research has focused on *front-end* or *query language-level* solutions to the problem of querying transformed data. Much of this research effort has been devoted to discovering the best way to relax the tight coupling of path expressions in a query to the hierarchy of the data. Approaches include techniques to approximately match a path to a hierarchy [2, 3, 13], apply XML *search* [6, 20, 28], or systems to relax, reinterpret, or rewrite the path expressions in a query [19, 27, 29, 31]. But these approaches do not investigate how to transform the XML *values* in the data; it is the values in the transformed hierarchy rather than the source hierarchy on which queries in the pipeline should be evaluated.

Research in XML data transformation languages is more relevant [7,16,23], but these approaches are inefficient since two passes are needed: one to transform the data, the second to query the transformed data. The most relevant front-end research is to combine an XML query with a view [1, 9, 14, 21, 22, 24]. Views that transform data are cumbersome to write, and element types constructed in the `return` clause of a view are distinct from seemingly similar element types referred to in path expressions in a query; they potentially have different values which must be first constructed before being queried. In other words, the view must be (temporarily) materialized and then queried. In contrast, our idea is support queries over data transformation views by manipulating the node numbering system rather than by query rewriting. The primary advantage is that we *virtually transform only data actually used in the query*. Note however that virtual hierarchies only construct views that are data transformations (which are a common, important kind of view), query rewriting is still necessary for views, in general.

```
for $b in //book
let $v := $b/* except $b/title except $b/author
return (fn:concat( "<book ",
   {return for $att in $b//@*
       return (fn:concat(
            name($att), "=", "'", $att, "'"))
   }, ">",
   {return for $ele in $v return {$ele}},
 "</book>")
```

Figure 5: Including "other" book information in Sam's transformation of Figure 1

```
for $t in
  virtualDoc("x.xml",
             "title { author { name } }"
            )//title
return <title>{$t/text()}
         {count($t/author)}
       </title>
```

Figure 6: Combining Sam's transformation with Rhonda's query

There are strategies for efficiently modifying PBN after an update [12,18,25,30]. *Update renumbering* is orthogonal to and quite different from virtual numbering. Update renumbering physically changes the PBN number for every node in an edit. In contrast, *v*PBN does not change any physical node numbers, instead it logically renumbers the data, re-using the extant physical numbers. Adapting update renumbering to support virtual hierarchies would be very expensive since all of the nodes in a data collection would have to be individually, physically renumbered at query time.

# 4. BACKGROUND

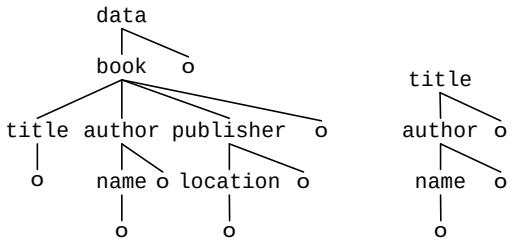
This section introduces terminology we use in the paper and reviews PBN.

## 4.1 Terminology

A DataGuide describes the parent/child relationships among the *types* in a data collection [10]. We assume a DataGuide, $S = (T, E)$ is a forest of $T$ types and $E$ edges, that connect a parent type to a child type. The type of a node in an XML data instance can be thought of as the concatenation of element/attribute names on the path from the root (a URI) to that node (as described more concretely in the **typeOf** function below). Note that this means that for a *recursive schema type*, each level of recursion is a different (actual) type. Also, note that the type includes the URI, so DataGuide's for different URIs have different sets of types. Figure 7 displays the DataGuide for the data instances of Figure 2 and Figure 3. The DataGuides are essentially the same size as the data instances since the instances are small. In general a DataGuide for a data collection will be much smaller than the data.

We assume the following helper functions for a DataGuide, $S$, on an XML data instance, $D$.

- **roots**$(S) = \{t \mid (t, \_) \in S \wedge \neg\exists v((t, v) \in S)\}$ is the set of types in $S$ that have no incoming edges.
- **name**$(S, v)$ - If $v \in D$ is an attribute or element then its name is the label of the corresponding type in $S$, otherwise $\circ$ ($v$ is a text node, for brevity we ignore other kinds of nodes).




**(a) Original shape** **(b) Transformed shape**

Figure 7: The DataGuides for the original data (Figure 2) and transformed data (Figure 3)

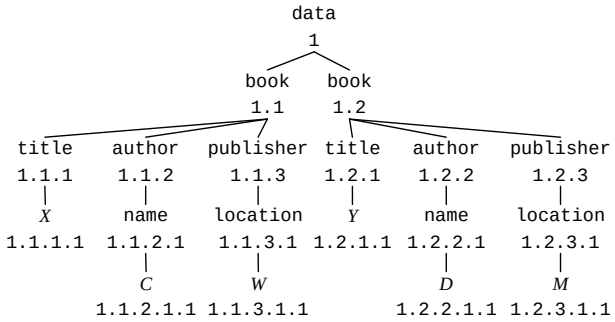



Figure 8: PBN numbers for the data model instance of Figure 2

- **typeOf**$(S, v)$ - Yields the *type* of a node, $v \in D$. Each node in the data has a well-defined type that is specified as a concatenation of the names of the elements on the path from a data root to the node, $v_k$, *e.g.,* **name**$(S, v_0)$**.name**$(S, v_1)$**.** ... **.name**$(S, v_k)$ where $v_i$ is a node at level $i$ on the path. The type is a node in $S$.
- **lcaTypeOf**$(S, v, w)$ - For $v, w \in D$, the lowest common ancestor type in $S$ between $\mathbf{typeOf}(S, v)$ and $\mathbf{typeOf}(S, w)$ or *null* if there is no least common ancestor (the nodes are in different trees in the forest).
- **length**$(S, v)$ - The length of the type of $v$ (number of names in the path).
- **originalTypeOf**$(S', S, v)$ - The original type of $v$, that is, $v$ is in a *v*DataGuide, $S'$, but we want the type of $v$ in the original Dataguide, $S$.

For example, the **typeOf** `author` in Figure 7(b) is `title.author`, and it has a length of 2. Its **originalTypeOf** is `data.book.author`. The **lcaTypeOf** of `title.author` and `title` is `title`.

As this paper is about prefix-based numbering, we use a simple specification of a *v*DataGuide. A *v*DataGuide can be specified using the following grammar.

$S \leftarrow$ `label` $P$
$P \leftarrow$ `{` $L$ `}` $\mid \epsilon$
$L \leftarrow D\ L \mid \epsilon$
$D \leftarrow$ `*` $\mid$ `**` $\mid$ `label`

In the grammar, the terminal `label` is a name or type in the original DataGuide. The label can be fully qualified to disambiguate and uniquely name a type, e.g., `x.y` specifies a different type than `x.z.y` is the (source) DataGuide. The terminal `*` represents the children (which are not mentioned elsewhere in the *v*DataGuide) for the specified `label` in the original DataGuide while `**` represents descendants. So for example, a *v*DataGuide to express the identity transformation (from and to the *v*DataGuide specified in Figure 7(a)) can be given as

```
data {
 book {
  title author { name } publisher { location }
 }
}
```

or simply as

```
data { ** }
```

Other issues relating to specifying data transformations, such as richer expression of transformations and reasoning about potential information loss, are orthogonal to this paper and have been researched elsewhere [7, 16, 23]. Our focus is on node numbering.

## 4.2 Prefix-based Numbering

In this section we review prefix-based numbering (PBN). PBN encodes the hierarchy in the numbering. In PBN each node in a data model instance is numbered as follows: a node is numbered $p.k$, where $p$ (the prefix) is the number of its parent and $k$ represents that it is the $k^{\text{th}}$ sibling in the order of the children.

An example can help clarify PBN. Figure 8 shows the PBN numbers for the data model instance of Figure 2. The PBN numbers are shown below the element and text nodes. The node numbered `1.2.2` (`<author>`) represents the second child in document order relative to its parent (which has a PBN number of `1.2`).

There are strategies for packing PBN numbers into as few bits as possible, making PBN numbers relatively concise [11]. PBN numbers are also very efficient for queries. Given two PBN numbers we can quickly compute (by comparing the numbers) a specific relationship (child, parent, ancestor, descendant, following sibling, preceding) of one PBN number relative to another. For instance `1.1.2` can be compared to `1.2`. Since `1.1.2` is neither a prefix nor a suffix of `1.2`, it is not a child, parent, ancestor, or descendant. The PBN number `1.1.2` precedes `1.2` in document order, but is not a preceding sibling since the parent of `1.1.2` (`1.1`) is different from that of `1.2` (`1`). The efficiency of PBN in quickly determining these relationships makes it useful for query processing in XML DBMSs and search engines.

There are also techniques for efficient update [18, 30]. This paper is orthogonal to fractional PBN numbers or other renumbering strategies.

## 4.3 After a data transformation

A data transformation renders PBN numbers useless. Consider the transformation of the data in Figure 8 using the *v*DataGuide clause from the query of Figure 6. The result is shown in Figure 9. The *original* PBN number for each node is shown below the node. The PBN numbers cannot be used to compute that `<title>` $Y$ (PBN number `1.2.1`) is a parent of `<author>` $D$ (PBN number `1.2.2`) since $Y$'s PBN number is not a prefix of $D$'s.

A transformation could produce two kinds of changes to the location of a node. 1) **Level change** - The level of a node may change. For example, `<title>` $Y$ has moved from level 2 in Figure 8 to level 1 in Figure 9. 2) **Parent change** - A node's parent may change. For example, `<author>` $D$ has switched from the second `<book>` of Figure 2 to the `<title>` $Y$ in Figure 9.

A transformed data model instance can be *renumbered* by reparsing or traversing the instance and assigning a new PBN number to each node. But renumbering is potentially expensive. First, the transformed data instance may be large, requiring some of the transformed data to be stored temporarily, thereby increasing disk

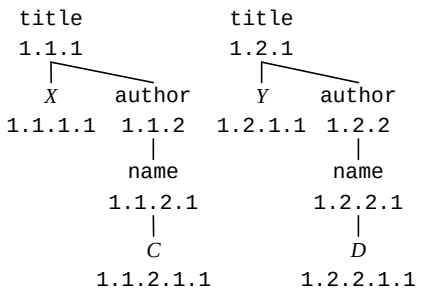



Figure 9: Incorrect, original PBN numbers for the result of Sam's transformation

I/O and slowing query evaluation. Second, an XML management system often has several indexes to improve query performance. For example, there will usually be an index to quickly look up nodes of a given type (*e.g.,* find all the `<author>` elements). In these indexes, when nodes are referenced, it is common to use the PBN number as a logical key. But the numbers are those in the original data instance, not the transformed instance. When the transformed data is renumbered, the indexes have to be recreated as well to use in query evaluation. Finally, it may be the case that when a view physically constructs data, more data than is needed is produced. The query may use only a small fraction of the data in the view specification. Query rewriting tries to optimize the view by pushing constraints from the query into the view. In contrast our approach is to virtually transform only the data needed by the query by applying the transformation at the level of the node numbers used in the query.

Since renumbering data is expensive we develop a strategy to virtually transform the PBN, which we call *virtual PBN* (*vPBN*). There are two parts to the technique: 1) computing virtual relationships between node pairs and 2) computing virtual node values. We consider each part in turn in Sections 5 and 6.

# 5. VIRTUAL RELATIONSHIPS

Virtual PBN maps each PBN number to a *virtual PBN number* (*vPBN number*). A *v*PBN number is like a PBN number, but adds a *level array*. The level array records the tree level of each component in a PBN number. Figure 10 depicts the level array below each PBN in the transformed instance of Figure 9. The leftmost `<title>` has a level array of `[1,1,1]` indicating that each component in the PBN number is on level `1`. The leftmost `<name>` has a level array of `[1,1,2,3]` indicating that the first two components represent the ancestor at level `1`, the next at level `2`, and the last component is at level `3`. The level array together with a PBN number forms a *v*PBN number.

*v*PBN numbers can be compared to determine location-based relationships. The relationships of interest are listed below. In this list for a *v*PBN number, $x$, let $x_a$ denote the level array, $x_n$ denote the PBN, and $\mathbf{max}(x_a)$ be the maximum number in the level array $x_a$. Note that for each location-based relationship, there is one additional constraint: the relationship must hold for the *types* of $x$ and $y$ in the *v*DataGuide, $V$. That is, node $x$ is a descendant of node $y$ if only if **vAncestor**$(x, y)$ holds and in the *v*DataGuide, **ancestor**(**typeOf**$(V, x)$, **typeOf**$(V, y)$). We assume that PBN is used to number the types in a DataGuide and quickly determine relationships in the DataGuide.

- **vSelf**$(x, y)$ - $x$ is the virtual self $y$ iff

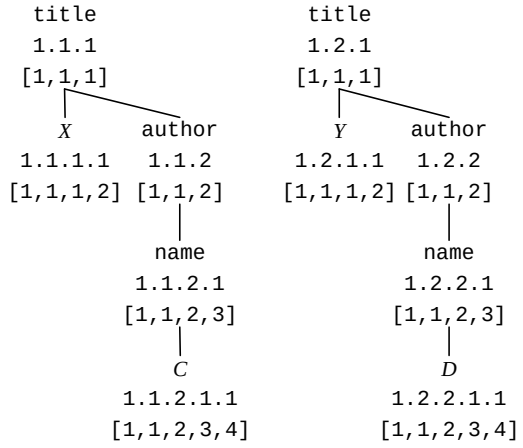



Figure 10: *v*PBN numbers for the result of Sam's transformation

$x_a = y_a \ \wedge \ x_n = y_n$
$\wedge$ **self**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vAncestor**$(x, y)$ - $x$ is a virtual ancestor of $y$ iff

$\mathbf{max}(y_a) > \mathbf{max}(x_a)$
$\wedge \ \forall i \Big[ 1 \leq i \leq \mathbf{max}(x_a)$
$\wedge \ y_a[i] = x_a[i] \Rightarrow x_n[i] = y_n[i] \Big]$
$\wedge$ **ancestor**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vParent**$(x, y)$ - $x$ is a virtual parent of $y$ iff

**vAncestor**$(x, y) \ \wedge \ \mathbf{max}(y_a) + 1 = \mathbf{max}(x_a)$
$\wedge$ **parent**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vDescendant**$(x, y)$ - $x$ is a virtual descendant of $y$ iff

$\mathbf{max}(x_a) > \mathbf{max}(y_a)$
$\wedge \ \forall i \Big[ 1 \leq i \leq \mathbf{max}(y_a)$
$\wedge \ x_a[i] = y_a[i] \Rightarrow x_n[i] = y_n[i] \Big]$
$\wedge$ **descendant**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vChild**$(x, y)$ - $x$ is a virtual child of $y$ iff

**vDescendant**$(x, y) \ \wedge \ \mathbf{max}(y_a) + 1 = \mathbf{max}(x_a)$
$\wedge$ **child**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vDescendant-or-self**$(x, y)$ - $x$ is a virtual descendant or self of $y$ iff

**vDescendant**$(x, y) \ \vee$ **vSelf**$(x, y)$
$\wedge$ **desendant-or-self**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vPreceding**$(x, y)$ - $x$ precedes $y$ in virtual document order iff

$\neg$**vAncestor**$(x, y) \ \wedge \ \neg$**vSelf**$(x, y)$
$\wedge \ \exists i \Big[ i \leq \mathbf{max}(x_a)$
$\wedge \ \forall j \leq i \Big[ x_a[j] = y_a[j] \Rightarrow x_n[j] = y_n[j] \Big]$
$\wedge \ i \neq \mathbf{max}(x_a) \Rightarrow x_n[i] < y_n[i] \Big]$
$\wedge$ **preceding**(**typeOf**$(V, x)$, **typeOf**$(V, y)$).

- **vPreceding-sibling**$(x, y)$ - $x$ virtually precedes $y$ and is a virtual sibling iff

$$\begin{aligned}&\textbf{max}(x_a) = \textbf{max}(y_a)\\ &\wedge\ \textbf{vPreceding}(x, y)\ \wedge\ \neg\textbf{vSelf}(x, y)\\ &\wedge\ \forall i\Big[1 \leq i \leq \textbf{max}(x_a) - 1\\ &\qquad\wedge\ x_a[i] = y_a[i] \Rightarrow x_n[i] = y_n[i]\Big]\\ &\wedge\ \textbf{preceding-sibling}(\textbf{typeOf}(V, x), \textbf{typeOf}(V, y)).\end{aligned}$$

- **vFollowing**$(x, y)$ - $x$ follows $y$ in virtual document order iff

$$\begin{aligned}&\neg\textbf{vAncestor}(x, y)\ \wedge\ \neg\textbf{vSelf}(x, y)\\ &\wedge\ \exists i\Big[i \leq \textbf{max}(x_a)\\ &\qquad\wedge\ \forall i\Big[j \leq i\\ &\qquad\qquad\wedge\ x_a[j] = y_a[j] \Rightarrow x_n[j] = y_n[j]\Big]\\ &\qquad\wedge\ i \neq \textbf{max}(x_a) \Rightarrow x_n[i] > y_n[i]\Big]\\ &\wedge\ \textbf{following}(\textbf{typeOf}(V, x), \textbf{typeOf}(V, y)).\end{aligned}$$

- **vFollowing-sibling**$(x, y)$ - $x$ virtually follows $y$ and is a virtual sibling iff

$$\begin{aligned}&\textbf{max}(x_a) = \textbf{max}(y_a)\ \wedge\ \textbf{vFollowing}(x, y)\\ &\wedge\ \neg\textbf{vSelf}(x, y)\ \wedge\\ &\wedge\ \forall i\Big[1 \leq i \leq \textbf{max}\\ &\qquad\wedge\ x_a) - 1(x_a[i] = y_a[i] \Rightarrow x_n[i] = y_n[i]\Big]\\ &\wedge\ \textbf{following-sibling}(\textbf{typeOf}(V, x), \textbf{typeOf}(V, y)).\end{aligned}$$

Let's consider several examples using the *v*PBN numbers in Figure 10. In the figure, the original PBN number is shown below the element name or text value. Below each PBN number is the level array for the number in the *v*DataGuide. The leftmost `<name>` is a virtual descendant of the leftmost `<title>` since its prefix at level `1` is `1.1`, which matches the prefix at level `1` of `<title>` (`1.1`). But is not a virtual descendant of the rightmost `<title>`; that `<title>` has has a prefix of `1.2` at level `1` which does not match `1.1`. Text node $C$ `1.1.2.1.1` virtually precedes `<author>` `1.2.2` since $C$ is not a virtual ancestor or self of `<author>`, and at level `1` $C$ has a prefix of `1.1` which is less than `<author>`'s prefix at level `1` (`1.2`). Finally $C$ is not a virtual following-sibling of $D$ since though they are at the same level, they do not have the same virtual parent (their prefixes differ at level `1`).

So *v*PBN has the property that a location-based relationship can be computed just by comparing the numbers for two nodes, just like PBN. *v*PBN slightly increases the space cost, at worst doubling the size of a number compared to PBN, though we will see in the next section that the level arrays do not have to be stored with the numbers since the level array can be stored with each type (it is the same array for each element of that type).

## 5.1 Sibling ordinals computed dynamically

The final component of a PBN number is a sibling ordinal, *e.g.,* the $n^{\text{th}}$ sibling is numbered $n$. The ordinal can be used in some path expressions, for instance, XPath has a node test for the ordinal, though data-centric applications tend not to use ordinals in queries (data is assumed to be unordered). While *v*PBN does preserve document order, it does not not compute sibling ordinals. Instead, if an ordinal is needed, it must be computed dynamically, *e.g.,* by queueing the siblings.

## 5.2 Assigning *v*PBN numbers

Each node in a transformed data instance must have a *v*PBN number, that is, a PBN number and a level array. The PBN number is always that of the node in the original data instance, but the level array must be constructed. Fortunately it is not necessary to assign a level array to each node individually, rather the level array is the same for each *type* in a *v*DataGuide.

We now present an algorithm (Algorithm 1) to build a map (a hash table) that maps each type to a level array. The algorithm traverses the *v*DataGuide to build the map, and also uses the original DataGuide. For each node that it visits, the algorithm extends the level array of the previous level. There are three cases to consider, which are depicted in Figure 11 and Figure 12. We explain the figure below.

*Case 1: The transformation moves a descendant in the original document to become a child in the virtual hierarchy* - In Figure 11(a), $Y$ is a descendant of $X$. The transformation moves $Y$ to become $X$'s child as depicted in Figure 11(b). Assume that $X$ is at level `n`, so $Y$ ends up at level `n+1`. Moreover, assume that $X$'s level array is $a$. $Y$'s level array is constructed by concatenating $X$'s level array with `[n+1, ... ,n+1]`, which has one array position for each component, $\texttt{z}_1\texttt{. ... .z}_\texttt{m}\texttt{.y}$, indicating that all these components are at level `n+1`. As an example, consider constructing the level array for `name` in Figure 7(b). The level of its parent is `2`, its parent's level array is `[1,1,2]` (which can be seen in Figure 10). Since `name` is a child of `author` in Figure 7(a) only one new value is concatenated to the level array: `[1,1,2]` • `[3]`, yielding `[1,1,2,3]` for the level array of type `title.author.name`.

*Case 2: The transformation moves an ancestor to a child* - In Figure 11(a), $X$ is an ancestor of $Y$. The transformation moves $X$ to become $Y$'s child as depicted in Figure 11(c). Assume $Y$ is at level `n`, so $X$ will move to level `n+1`. $Y$'s level array is the level array of its incoming least common ancestor, $a$, concatenated with the array `[n, ... ,n]`, which has one array position for each component, $\texttt{x.z}_1\texttt{. ... .z}_\texttt{m}\texttt{.y}$, indicating that all these components are at level `n`. Since $X$'s PBN number lacks the components $\texttt{z}_1\texttt{. ... .z}_\texttt{m}\texttt{.y}$, it omits those components from its level array. Instead $X$'s level array is that of the least common ancestor, $a$, concatenated with the array `[n,n+1]` indicating that component `x` is at level `n`, and there is one more level with no corresponding component in its PBN number. So $X$'s level array is one larger than its PBN number. As an example, consider inverting `name` and `author` in Figure 7(b). In the original DataGuide shown in Figure 7(a) the least common ancestor of `name` and `title` is `book`, which has a PBN number of length 2, so the least common ancestor's level array is the first two positions of the level array for `title`, *i.e.,* `[1,1]`. The level array for `name` would then be `[1,1]` • `[2,2]`. The first two numbers in its PBN number correspond to its parent in the first level, while the last two locate its position on the second level. The level array for `author`, the new child of `name` would be `[1,1]` • `[2,3]`.

*Case 3: The transformation creates a parent, child relationship between a pair of nodes previously related only through a least common ancestor* - In Figure 12(d), $X$ is related to $Y$ through least common ancestor $Z$. The transformation moves $X$ to become $Y$'s parent as depicted in Figure 12(e). Assume $X$ will move to level `n`, so $Y$ ends up at level `n+1`. $X$'s level array is the level array of its incoming least common ancestor, $a$, concatenated with the array `[n, ... ,n]`, which has one array position for each component, $\texttt{v}_1\texttt{. ... .v}_\texttt{m}\texttt{.x}$, indicating that all these components are at level `n`. $Y$'s level array is the level array of its incoming least common ancestor, $a$, concatenated with the array `[n+1, ... ,n+1]`, which has one array position for each component, $\texttt{w}_1\texttt{. ... .w}_\texttt{k}\texttt{.y}$, indicating

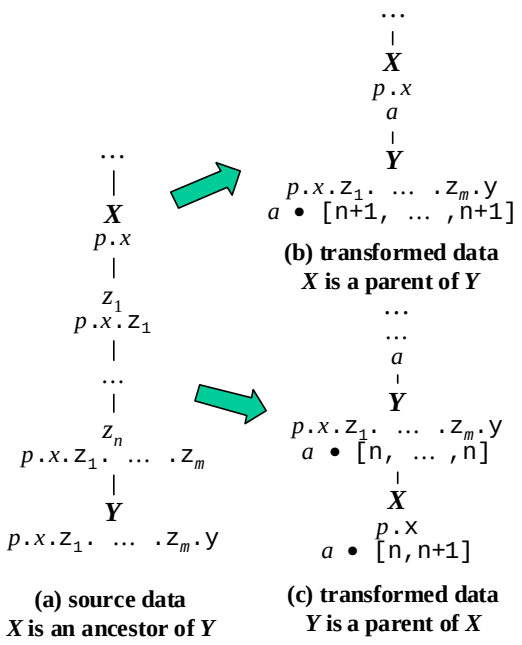



Figure 11: Case 1 and Case 2 for extending the level array

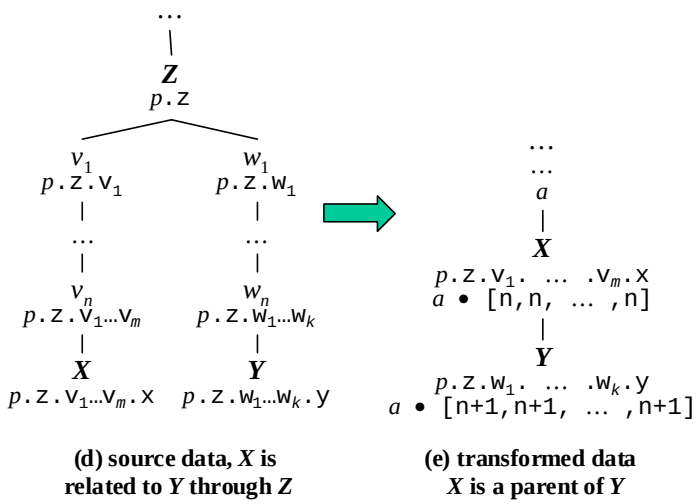



Figure 12: Case 3 for extending the level array

that all these components are at level n+1. As an example, consider constructing the level arrays for title and author in Figure 7(b). In the original DataGuide shown in Figure 7(a) the least common ancestor of title and author is book, which has a PBN number of length 2, so the least common ancestor's level array is the first two positions of the level array for title, *i.e.,* [1,1]. The level array for title would then be [1,1] • [1]. All three numbers in its PBN number place it on the first level. The level array for author, the new child of title is [1,1] • [2].

Consider building the level arrays for the nodes in Figure 10. Algorithm 1 first builds the level array for the root type in the *v*DataGuide of Figure 7(b). The only root type is title. The algorithm allocates an array of length 3 (the length of the original path for <title> in Figure 7(a). It fills in the array by assigning level 1 to every cell. Next it recursively descends the *v*DataGuide to build the level arrays for level 2. It uses case 1) given above (a descendant becomes a child) for the text node type, and case 3) (types are related through a least common ancestor) for the author. The algorithm then recursively descends the *v*DataGuide repeatedly applying case 1 to construct the rest of the level arrays.

Algorithm 1 has a worst-case time complexity of $O(cN)$ where $N$ is the size of the *v*DataGuide and $c$ is the deepest level (longest PBN number) in the tree. The algorithm visits each cell in the *v*DataGuide, costing $O(N)$. When it visits a cell it allocates and fills an array of at most size $c$, and also might compute a least common ancestor. This can be done at a cost of $c$ by numbering the DataGuide using PBN. The least common ancestor type can be computed by finding the shared prefix in a pair of PBN numbers.

The worst-case space complexity is also $O(cN)$ since the algorithm computes a map with $N$ key/value pairs each of size $O(c)$.

## 5.3 Properties of *v*PBN

We now prove that virtual prefix-based numbers can be used to compute relationships in the space described by a *v*DataGuide. For brevity, we focus only on the descendant relationship (the other relationships are variants).

THEOREM 1. *Let $V_D$ be a virtual document created from a document, $D$, using a vDataGuide, $V_G$, then* ***vDescendant***$(w, q) \Leftrightarrow w$ *is a descendant of $q$ in $V_D$.*

PROOF. ($\Leftarrow$) Assume false. Then $w$ is not a descendant of $q$ in $V_D$ but **vDescendant**$(w, q)$ holds. Since **vDescendant**$(w, q)$ we know that **typeOf**$(V_G, w)$ must be a descendant of **typeOf**$(V_G, q)$ and $q$'s level array has fewer levels than $w$'s, so for $w$ to not be a descendant of $q$ it must be the case that for some $i$, $w_a[i] = q_a[i] \wedge w_n[i] \neq q_n[i]$. Is this possible? There are three cases.

Case 1) $w$ is a descendant of $q$ in $D$, hence $q$'s PBN number is a prefix of $w$'s, so $w_n[i] = q_n[i]$ for all $i \leq$ **max**$(q_a)$.

Case 2) $w$ is an ancestor of self of $q$ in $D$. Algorithm 1 will create level arrays for $w$ and $q$ that are the same up to $w$ and $q$'s least common ancestor, which is $w$. Hence, $w_n[i] = q_n[i]$ for all $i \leq$ **max**$(q_a)$.

Case 3) $w$ follows (or precedes) $q$ in $D$. Algorithm 1 will create a level array for $q$ that is the same up the level array for $w$ to the point where they diverge at their least common ancestor, $z$, as depicted in Figure 12. Said differently, let $p.z$ be the PBN number for $z$ and $z_a$ be its level array. $q$'s level array will be $z_a$ padded with extra levels up to level $n$. $w$'s level array will be $z_a$ padded with extra levels strictly greater than $n$. Hence, $w$ and $q$ must have the same PBN number's up to $z$, so $w_n[i] = q_n[i]$ for all $i \leq$ **max**$(z_a)$. And for all **max**$(z_a) < j \leq$ **max**$(q_a)$, $w_a[j] \neq q_a[j]$.

Hence, $w_a[i] = q_a[i] \wedge w_n[i] \neq q_n[i]$ cannot be satisfied.

($\Rightarrow$) Assume false. Then it is not the case that **vDescendant**$(w, q)$ holds, even when $w$ is a descendant of $q$ in $V_D$. If $w$ is a descendant of $q$ then we know that **typeoOf**$(V_G, w)$ must be a descendant of **typeOf**$(V_G, q)$. Moreover, it must be the case that $q$'s *v*PBN is a (virtual) prefix of $w$'s. So it must be the case that $w_a[i] = q_a[i] \Rightarrow w_n[i] = q_n[i]$ for all $i \leq$ **max**$(q_a)$. Hence, **vDescendant**$(w, q)$ must hold. □

# 6. COMPUTING TRANSFORMED VALUES

Suppose that an XML DBMS stores the source XML data as a long string. Then the *value* of each kind of node is a specific substring. For instance, the value of an element is the substring beginning with the starting tag for that element and continuing to the ending tag. Consider the value of the first <author> element in Figure 2. It is the following string (with whitespace stripped).

<author><name>C</name></author>

**Algorithm 1:** Algorithm to build level arrays

**Input**: Shape $D$, Shape $T$
$D$ is the DataGuide of the original data
$T$ is the *v*DataGuide
**Output**: $M$
$M$ maps a Type $T$ to a level array $A$: $T \rightarrow A$

**Procedure** build(Shape $D$, Shape $T$, int $n$, int[] $L$, Type $v$, Map $M$)
$n$ is the current level
$L$ is the current level array
$v$ is the current lowest common ancestor type
**begin**
  // iterate through the roots of the *v*DataGuide
  **forall the** $r \in \mathbf{roots}(T)$ **do**
    // Find the length of the lca in $D$
    $x \leftarrow \mathbf{originalTypeOf}(S, D, r)$
    **if** $v \neq$ *null* **then**
      $x \leftarrow \mathbf{lcaTypeOf}(D, v, x)$
    $k \leftarrow \mathbf{length}(x)$
    // Allocate an array of sufficient length
    $s \leftarrow \mathbf{length}(\mathbf{originalTypeOf}(S, D, r))$
    $r_a \leftarrow$ new array$[s+1]$
    // Copy the current level array
    **for** $i \leftarrow 1 \ldots k$ **do**
      $r_a[i] \leftarrow L[i]$
    // Extend the new array with the current level
    **if** $k+1 = s$ **then**
      $r_a[s+1] = n$
    **else**
      **for** $i \leftarrow k+1 \ldots s$ **do**
        $r_a[i] \leftarrow n$
    // Add to the map
    $M(\mathbf{typeOf}(r)) \leftarrow r_a$
    // Do not recurse if $r$ is a text node type
    **if** $r \neq \circ$ **then**
      **forall the** $(r, c) \in \mathbf{edges}(S)$ **do**
        // Recursively call build
        build$(D, c, n+1, r_a, x, M)$

// Call build initially
$M \leftarrow$ new Map()
build$(D, T, 1, [\,], null, M)$

The value is an important part of XQuery processing. For instance, in the evaluation of the `return` clause each variable that is bound to a node evaluates to the value of that node.

A critical component in the implementation of an XML DBMS that uses PBN is a *value index* to quickly find the value of a node given its PBN number. The index maps a node's PBN number to a range of characters in the source data string that forms its XML value. For the node corresponding to the first `<author>` element the value index would map its PBN number, `1.1.2`, to the range 29-60 indicating that the XML value starts at the 29th character in the source data string and ends at the 60th character.

The character positions are not necessarily this simple, rather they are usually some combination of a disk block number and offset within the block to facilitate fast retrieval from disk [4]. The XML string may also be modified for storage. Header information for each node, *e.g.,* the kind of node (text, element, etc.) is often inserted into the XML string stored on disk. When retrieving the value from disk, the header information for each node is read, processed, and removed from the value. We will assume that the header information has a PBN number and a *Type ID*, which is

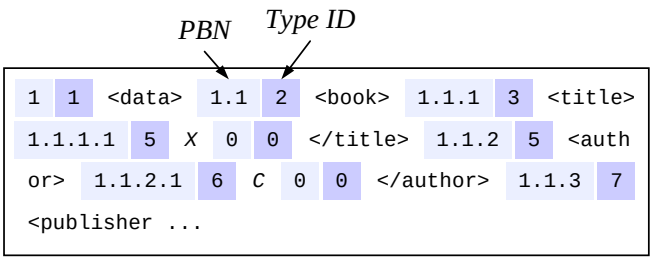



Figure 13: XML string with node header information on disk

an integer field that names the *source data type*, *e.g.,* a `<title>` would have an integer that corresponds to the source data type `data.book.title`. An example of the string layout on disk is shown in Figure 13. The header information for each node, which consists of a PBN number and a Type ID, is shaded in the figure.

A data transformation, unfortunately, potentially changes the value of a node. The pieces of the transformed value could be non-contiguous and out of order with respect to the transformed value. So getting the value of a node has to be implemented differently.

For example, consider computing the transformed value of the first `<title>` in Figure 10. One of its descendants is an `<author>`, which should be part of its transformed value. But Figure 13 shows that the text for `<author>` *starts* after the closing tag for `<title>`; it is not originally part of `<title>`'s value.

In this section we describe how to compute transformed values. An overview of the strategy is depicted in Figure 14. (As an aside, the problem of computing a transformed value is similar to that of computing a node constructor for the transformed space, *e.g.,* a `getChildren` constructor builds a sequence of PBN numbers, so we use the same strategy for node constructors.) 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 string from disk, and constructs *children queues* that are populated with the pieces that will form the value. The second step, Rendering, uses the queues to construct the transformed value from the pieces.

## 6.1 Preprocessing

The first step is to read from disk a string that is large enough to contain all of the pieces of a transformed value. This container string is the source data value of the node or one of its ancestors. The node that contains the pieces of the value in the source data is described by the intersection of the level arrays for the node and its descendants in the *v*DataGuide.

Consider preprocessing the value of the first `<title>` in Figure 10, which has a PBN number of `1.1.1`. We want to determine the PBN number of the node that has a value which contains all of the pieces that are a part of the `<title>`'s value. The intersection of the level arrays of the node and all its descendants in Figure 10 is the level array `[1,1]`. (In effect, the intersection computes the level array of the least common ancestor of the types in the original DataGuide.) So the container string for node `1.1.1` is the string returned by the source data value index look-up for `1.1` (which is the value of a `<book>` element).

Next, the string is read and processed. For each piece of the value string, the header information is processed. If the header indicates that the piece does not have a type that should be in the *v*DataGuide, then the piece is discarded. Otherwise, the piece is added to one of the *children queues*. There is one queue for each element type in the *v*DataGuide. The queue holds, in document order, the PBN

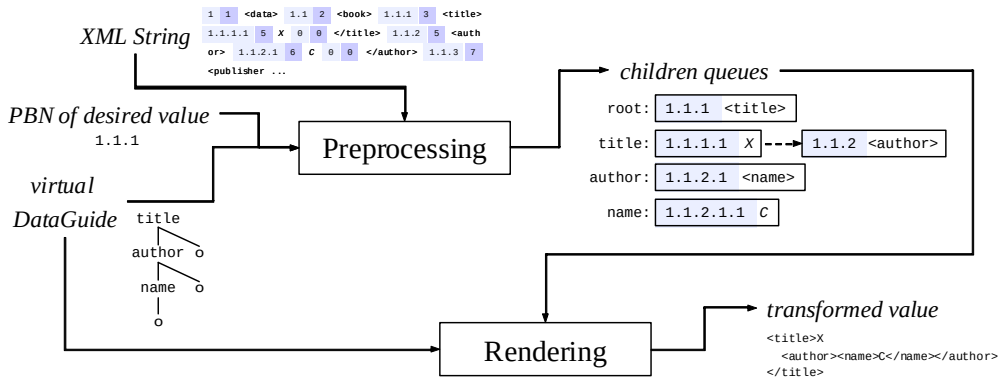



Figure 14: Two-step process for computing a transformed XML string

number of each potential child for a node of the indicated type and the value of its piece, *e.g.,* the `title` queue holds the PBN numbers and (partial) values of children for a `<title>` in the transformed value as shown in Figure 10.

## 6.2 Maintaining Document Order

Note that the nodes in each child queue are sorted in document order by PBN number. Since the container string is processed in document order, the sorted order is maintained at no additional cost; each new piece is simply appended to the appropriate queue.

## 6.3 Rendering

The transformed value is created by recursively descending the *v*DataGuide and concatenating, in the appropriate order, the text pieces. Algorithm 2 gives the algorithm. Starting with the root, the algorithm visits each child queue in the *v*DataGuide. It iterates through the child queue, recursively building the transformed value of a child. Depending on the *v*DataGuide, several passes may be required over a child queue, so an optional, helpful strategy is to cache the computed transformed value, and look-up the value when needed.

As an example, consider building the transformed value for PBN number `1.1.1`. Figure 14 shows the children queues for this PBN number. The algorithm starts with the `root` queue. It selects the first node, and since it is an element type node, constructs its value by recursively calling **Function buildValue**. The recursive call traverses the `title` queue. The first node in that queue is a text node descendant of `1.1.1` and the transformed string becomes "*X*". Next, the second child in the queue is visited. It too is a descendant (in the transformed space), and so its value is constructed by recursively visiting the `author` and `name` queues. These visits build the string "`<name>`*C*`</name>`". That string is wrapped with `<author>` tags, appended to the string "*X*", and returned to the top level invocation of **Function buildValue** which wraps `<title>` tags around the result yielding the transformed value given below.

```
<title>X<author><name>C</name></author></title>
```

We analyze the cost of each step in the algorithm separately. Preprocessing has a worst-case time complexity of $O(M)$ where $M$ is the length of the container string. It scans the container string at a cost of $O(M)$ and puts the desired pieces into the children queues. We reasonably assume constant time queue insertion cost, and constant time queue selection (using a hash table). Preprocessing has a space complexity of $O(M)$ since the children queues contain at most the pieces of the processed string.

Though the complexity appears modest, note that $M$ could be the size of the entire data collection (the smallest least common ancestor of the types in the *v*DataGuide could be the root of the data collection, *i.e.,* the pieces of the transformed value are spread throughout the data). The on-the-fly preprocessing step however can be replaced by a static preprocessing step. The XML string can be shredded to queues once when it is first inserted into the data store. One queue is needed for each each type in the original DataGuide. The source queues are (logically) merged at query evaluation time, in sorted order, into a single queue for the children of a particular type (since the children are not known until the data is virtually transformed). The queue contains all of the pieces for that type and the rendering algorithm has to quickly skip to the cluster of descendants for any given PBN number. Such a queue can be efficiently implemented using a $B^+$-tree. This strategy costs $O(D)$ space to store the queues for each type, where $D$ is the size of the data collection, and lowers the amount of memory needed at run-time from $O(M)$ to $O(v)$ where $v$ is the size of the longest branch in the *v*DataGuide.

Rendering has a worst-case time complexity of $O(M)$. It visits each node in each child queue at most once, and there are at most $M$ pieces (usually far fewer). However, it has a worst-case space complexity of $O(T)$ where $T$ is the length of the transformed value since the transformed value is constructed on-the-fly. It also maintains a cache of size $O(M)$. Strategies to reduce the cache size and stream output of virtual values are beyond the scope of this paper.

# 7. IMPLEMENTATION & EXPERIMENTS

We chose to extend eXist-db, version 1.4.1, which is an open source native XML DBMS written in Java. We chose eXist-db because it is a fully-functional XML DBMS, with continuing support and releases, and an active development community. The eXist-db project provides the best opportunity to contribute our research results to a public domain, widely-visible, and widely-used XML DBMS. We made the following changes to eXist-db.

- Modified the ANTLR grammar to parse *v*DataGuides, and added the virtualDoc() function.
- Modified the `org.exist.numbering` package to support *v*PBN as discussed in Section 4.3.
- Modified classes in the `org.exist.storage` package to add a Type ID to each node header in an XML string stored on disk.

**Algorithm 2:** Algorithm to build a transformed value

**Input**: Shape $T$, Queues $Q[\,]$
$T$ is the *v*DataGuide
$Q[\,]$ is the children queues data structure
**Output**: XMLString $s$
$s$ is the transformed value

**Function buildValue**(Shape $T$, Queues $Q[\,]$, PBN $n$, Type $t$)
$T$ is the *v*DataGuide
$Q[\,]$ is the children queues
$n$ is the parent's PBN number
$t$ is the parent's type
**begin**
  // Check to see if we've already computed the value
  **if cache.contains**$(n)$ **then**
    return cache.get($n$)
  // Initialize the string
  String $s \leftarrow$""
  // For each child in the queue
  **for** $c \in Q[t]$ **do**
    // Check if it is a child of this node
    **if** $n$ *is null or* **descendant**$(c.PBN, n)$ **then**
      **if** $c$ *is a text node* **then**
        $s \leftarrow$**concat**($s$, $c$.value)
      **else**
        // Recursively call **buildValue**
        String $v \leftarrow$**buildValue**($t$, $Q[\,]$, $c$.PBN, $c$.type)
        $s \leftarrow$**concat**($s$, $c$.startTag, $v$, $c$.endTag)
        // Cache computed value
        **cache.put**($c$.PBN, $v$)
  return $s$

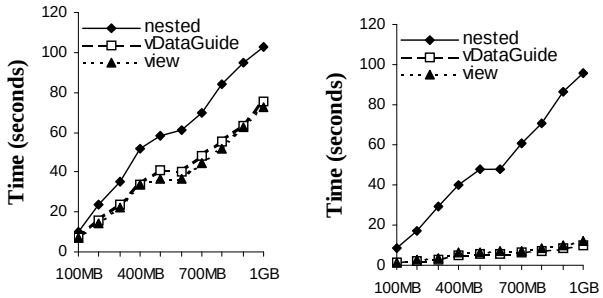
// Call **buildValue** initially
return **buildValue**($T$, $Q[\,]$, null, root)

- Added a SAX parser to parse inserted documents and build D ataGuide-related BerkeleyDB tables needed to analyze a DataGuide and construct a transformed value. Modified the existing SAX parser to invoke the actions in the new parser.
- Added the code from the XMORPH project [7, 8] to support evaluation of *v*DataGuides.
- Modified the data serializers to retrieve the virtual value of a node when needed.

The code is available from the project's home page, `cs.usu.edu/~cdyreson/XMorph`. We have not yet implemented support for XUpdate operations (which may also need to modify the *v*DataGuide tables), sibling ordinal predicates, or ordering constructors (the `ORDER` clause).

We performed several experiments which we explain below. The experiments were run on a dedicated machine with an Intel 3.40GHZ CPU, 16GB of RAM, and a 1TB, 7200RPM disk. We implemented and tested using Java 1.7, with a Xerces SAX parser, BerkeleyDB Java Edition, version 4.0.71, eXist-db, version 1.5.1, on a Linux system running Ubuntu 13.0.11. The measurements in each experiment were averaged over 5 runs. To eliminate eXist caching effects we used an "embedded" implementation of eXist-db and restarted it after each query, *i.e.,* every query starts with a cold cache. We did not clear the operating system cache between runs.

**Nested vs. virtual vs. "perfect" view experiments**: The first experiment tests the example given in Section 2. It compares the nested query given in Figure 4 against the *v*DataGuide query given in Figure 6, and a "best-case" hand-crafted view. The result is shown in Figure 15(a). For the data, we used DBLP XML data, which is 1GB in size. We converted DBLP's various formats for proceedings, articles, etc. to a single format to ensure that the entire dataset was queried (e.g., we converted `<booktitle>` to `<journal>` elements). We then sliced the data into 100MB fragments, each representing 10% of DBLP data, and coalesced the fragments to form 10 data sets from 100MB to 1GB in size. In the nested XQuery, the inner result is constructed, parsed and renumbered for evaluation in the outer query. The *v*DataGuide avoids constructing and parsing the temporary inner result, but must still do some rendering since every article appears in the result. The view query does not have to transform the data, so performs best.




(a) Placing `<author>` counts within `<title>`s

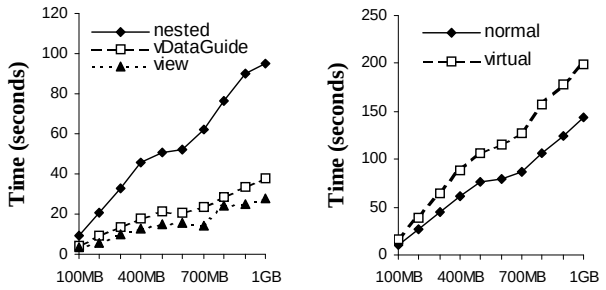
(b) Selecting one title

Figure 15: Nested query compared to *v*DataGuide and view query, *i.e.,* query in Figure 4 vs. Figure 6 vs. perfect view query

The second experiment repeats the first experiment, but adds a highly selective condition (where `<title>` equals a specific title) to the outer query. The result is shown in Figure 15(b). The nested XQuery still produces a large result, from which one title is selected. In contrast, the *v*DataGuide evaluates the query in the virtual numbering space by performing the selection, just like the hand-written view. The third experiment uses a less selective condition (where `<year>` is 2002). The result is shown in Figure 16(a).

The three experiments illuminate how the cost of *v*PBN compares to that of using views. In Section 2 we noted that a data transformation view could be complicated to write, might construct new element types with the same element names as those in the underlying data, and are tightly-coupled to hierarchies decreasing their portability. Only the second issue impacts cost. The experiments show the upper and lower bounds on evaluating a query using a view. If the view rewriting can combine the view and the query to avoid constructing a temporary `<title>` type (if it can avoid the second issue) then *v*PBN overhead will be incurred. But if the view rewriting inserts the view as a nested query (is unable to recognize that `<title>` in the view does not have to be constructed) then the view approach will devolve to the nested query approach. The selectivity of the query determines how much extra work *v*PBN has to do rendering transformed values using Algorithm 2 described in Section 6.3 vs. the perfectly written view. In either case, *v*PBN would still offer better portability and make it easier to write the data transformation.

**Overhead experiment**: The overhead experiment measures the overhead of *v*PBN compared to normal eXist-db evaluation. It uses a query on an *identity* *v*DataGuide. The identity *v*DataGuide just slows query evaluation, especially for large *v*DataGuides, without doing any "useful" work, since the transformed data is the same as the original data. Rather than quickly reading an XML value for a node from disk, each value must be constructed piece by piece by pre-processing and rendering. We designed an experiment using




(a) Selecting the year 2002 (b) Input overhead

Figure 16: Further experiments with DBLP

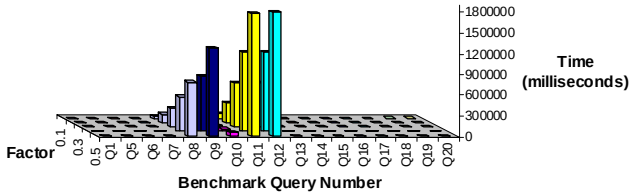



Figure 17: Raw cost of *v*DataGuide on XMark benchmark

the XMark benchmark. We generated five documents (benchmark factors .1 through .5) and evaluated the queries in the benchmark. The *v*DataGuide is big, it has 238 element types. The results are graphed in two figures. Figure 17 shows the raw cost of evaluating each query using *v*PBN on a document generated with the given benchmark factor. We removed benchmark queries Q2, Q3, and Q4 as they concern sibling ordinals, *e.g.,* get the $n^{\text{th}}$ child.[1] Some queries are far more expensive than others to evaluate because they traverse and retrieve large parts of the document. To visualize the actual overhead we plotted the ratio of the overall cost (*v*PBN/PBN) for each query and factor. The result is shown in Figure 18. A slowdown of 2 means that the query was 200% slower using *v*PBN. For many of the queries the slowdown decreases as the document size increases. The graph also shows that for the expensive queries (Q8, Q9, Q11, and Q12) the slowdown is 2 to 3 for *v*PBN. Again, the experiment was designed to uncover the overhead cost, and the overhead is only incurred when a query uses a *v*DataGuide.

**Heap experiment**: We also monitored memory use as part of the overhead experiment using Java's verbose garbage collection option. Figure 19(a) shows the results for XMark's benchmark query Q8 at a benchmark factor of 0.5. The graph plots average heap size (after generational and full garbage collection events) over the lifetime of the run. For instance, at about 37% of the run, *v*PBN maxed out at 16GB of memory, triggering a full garbage collection event. The graph shows that *v*PBN consumes more heap space, at a faster rate, leading to more garbage collection events. The extra memory use is part of the overhead illustrated in the previous experiment.

**Input experiment**: This experiment measures the additional time for data input with *v*PBN. A separate SAX parser builds the *v*DataGuide when an XML document is stored. We used the DBLP collection from the first experiment. The results are plotted in Figure 16(b). For a 1GB document, the overhead is about 20%.

[1]Sibling ordinals in PBN are known statically, while *v*PBN must compute them dynamically, which we have not yet implemented. Interestingly eXist-db computes ordinals dynamically because it uses fractional PBN numbers.

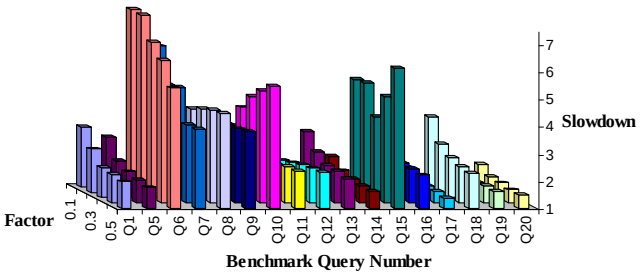


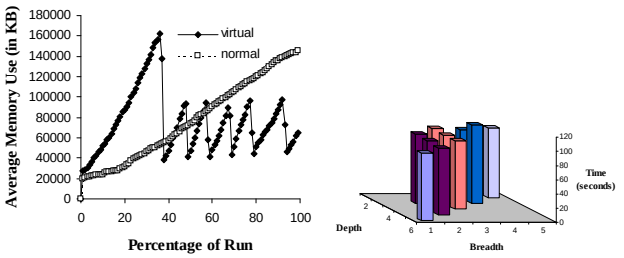

Figure 18: Relative slowdown

**Permuting *v*DataGuide experiment**: The permutation experiment measures the impact of changing the virtual hierarchy. We created ten different *v*DataGuides that use the same number of labels by varying the depth of nesting from 2 (a relatively flat structure) to 6 levels (richer nesting), and permuted the order of the labels. We ensured that the size of the virtual document was roughly the same for each *v*DataGuide by nesting only singleton elements (e.g., we nested `<year>` inside `<journal>`). Each query outputs the entire virtual document. We tested using the 1GB DBLP dataset from the first experiment. The result is shown in Figure 19b. The cost remains flat across the various shapes showing that the structure of the *v*DataGuide has little impact on performance.

## 8. SUMMARY

Prefix-based numbering supports efficient query evaluation of hierarchical data through the fast evaluation of location-based node relationships. But the numbering is rendered obsolete when nodes change location in a data transformation. This paper introduces virtual prefix-based numbering (*v*PBN) which allows nodes to appear as though they were moved to a new location without physically moving the nodes. *v*PBN is **concise**. It slightly increases the size of a PBN number. Each PBN number needs an additional integer-sized field to name a type; the type maps to the needed level array to convert a PBN number to a *v*PBN number. The level arrays are stored with the element types. *v*PBN supports **efficient querying**. We showed that *v*PBN maintains the property that node relationships, *e.g.,* is this node an ancestor of another, can be determined by simply comparing the numbers for each node, as is currently done using PBN. **Update is also efficient**. *v*PBN numbers are dynamically constructed, and so do not add to the cost of data update (other than updating the tables to analyze the DataGuides). Finally, *v*PBN is **practical**. We gave an overview of the changes needed to implement *v*PBN in eXist-db.

In future we plan to automatically generate *v*DataGuides. Relying on programmers to specify *v*DataGuides has two problems. First, a programmer may change the query but forget to change the *v*DataGuide. Second a programmer may give an incorrect *v*DataGuide, resulting in data in the wrong hierarchy for the query. The best way to solve both problems is to take *v*DataGuide construction out of the hands of programmers and instead automatically infer a *v*DataGuide from an XQuery query. Another approach to making it even easier to construct *v*DataGuides is to develop a GUI for visualizing a DataGuide and supporting drag and drop operations to construct the desired *v*DataGuide. Adding features to the *v*DataGuide specification language, such as *v*DataGuide composition and value-based conditional guides, would also help. We also plan to implement a strategy to store nodes indexed by PBN number. This will lower the cost of rendering virtual values since a related node can be found by a fast index-lookup rather than by a slow




(a) Average memory use (b) Permuting the *v*DataGuide

Figure 19: Heap and permutation experiments

traversal of the containing subtree. Finally, we plan to examine the open problem of creating a virtual hierarchy from data annotated with metadata, especially proscriptive metadata (*e.g.,* security) that restricts use of the data. The metadata may limit or modify data as it is virtually transformed to fit in a virtual hierarchy.

## 9. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1144404 entitled "III: EAGER: Aspect-oriented Data Weaving." Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We wish to thank Shuohao Zhang for his insights in starting this research.

## 10. REFERENCES

[1] F. N. Afrati, R. Chirkova, M. Gergatsoulis, B. Kimelfeld, V. Pavlaki, and Y. Sagiv. On rewriting XPath queries using views. In *EDBT*, pages 168–179, 2009.

[2] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *EDBT*, pages 496–513, 2002.

[3] N. Augsten, M. H. Böhlen, and J. Gamper. The *q*-gram distance between ordered labeled trees. *ACM Trans. Database Syst.*, 35(1), 2010.

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

[5] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6):377–387, 1970.

[6] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.

[7] C. E. Dyreson and S. S. Bhowmick. Querying XML Data: As You Shape It. In *ICDE*, pages 642–653, 2012.

[8] C. E. Dyreson, S. S. Bhowmick, and K. Mallampalli. Using XMorph to Transform XML Data. *PVLDB*, 3(2):1541–1544, 2010.

[9] M. El-Sayed, E. A. Rundensteiner, and M. Mani. Incremental Maintenance of Materialized XQuery Views. In *ICDE*, page 129, 2006.

[10] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.

[11] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[12] M. P. Haustein, T. Härder, C. Mathis, and M. Wagner. Deweyids - the key to fine-grained management of xml documents. *JIDM*, 1(1):147–160, 2010.

[13] Y. Kanza and Y. Sagiv. Flexible Queries over Semistructured Data. In *PODS*, 2001.

[14] A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD Conference*, pages 565–576, 2012.

[15] H.-K. Ko and S. Lee. A binary string approach for updates in dynamic ordered xml data. *IEEE Trans. Knowl. Data Eng.*, 22(4):602–607, 2010.

[16] S. Krishnamurthi, K. E. Gray, and P. T. Graunke. Transformation-by-Example for XML. In *PADL*, pages 249–262, 2000.

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

[18] C. Li, T. W. Ling, and M. Hu. Efficient Updates in Dynamic XML Data: From Binary String to Quaternary String. *VLDB J.*, 17(3):573–601, 2008.

[19] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.

[20] Z. Liu, J. Walker, and Y. Chen. XSeek: A Semantic XML Search Engine Using Keywords. In *VLDB*, pages 1330–1333, 2007.

[21] D. Luo, T. Chen, T. W. Ling, and X. Meng. On View Transformation Support for a Native XML DBMS. In *DASFAA*, pages 226–231, 2004.

[22] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, pages 972–983, 2011.

[23] T. Pankowski. A High-Level Language for Specifying XML Data Transformations. In *ADBIS*, pages 159–172, 2004.

[24] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *SIGMOD Conference*, pages 455–466, 1999.

[25] V. Sans and D. Laurent. Prefix based numbering schemes for xml: techniques, applications and performances. *PVLDB*, 1(2):1564–1573, 2008.

[26] 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*, pages 204–215, 2002.

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

[28] B. Q. Truong, S. S. Bhowmick, C. E. Dyreson, and A. Sun. MESSIAH: Missing Element-Conscious SLCA Nodes Search in XML Data. In *SIGMOD Conference*, pages 37–48, 2013.

[29] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 537–538, 2005.

[30] J. X. Yu, D. Luo, X. Meng, and H. Lu. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web*, 8(1):5–26, 2005.

[31] S. Zhang and C. E. Dyreson. Symmetrically Exploiting XML. In *WWW*, pages 103–111, 2006.