Xpriori Technical Paper

Virtual Document Ordering and Set Operations using Couplet Hierarchical Vectors DPP™

by Harry Direen, Ph.D.

Release 1.2

NOTE: In October 2003, Xpriori, LLC acquired NeoCore Holdings, LLC including all technology and patents. Any references to Neo, NeoCore or NeoCore Holdings, LLC technology or patents as such are now the property of Xpriori, LLC.

Xpriori, LLC
2864 S. Circle Dr.
Ste. 401
Colorado Springs, CO 80906
(719) 425-9840
www.xpriori.com

Version 1.2

Xpriori technology is protected by the following patents:
US Patent #5,742,611 (21 Apr 98)
US Patent #5,942,002 (8 Aug 99)
US Patent #6,157,617 (5 Dec 00)
US Patent #6,167,400 (26 Dec 00)
US Patent #6,324,636 (27 Nov 01)
US Patent #6,493,813 (10 Dec 02)
US Patent #6,792,428 (14 Sept 04)

Other U.S. and international patents pending.

Introduction

The hierarchical structure of XML documents may be broken down into information couplets. The information couplet is a metadata/data pair where the metadata is the context of the data, derived from the XML tag structure. Chris Brandin's paper, [1] "NeoCore XML Document Storage and Indexing Engine Architecture", explains couplets and the strong informational and architectural reasons for using them. Couplet hierarchical vectors (CHVs) provide a means of tracking, operating on, and reconstructing the hierarchical structure of XML documents from a set of couplets.

This paper assumes the reader has a working knowledge of Chris Brandin's paper [1]. Concepts in that paper will be expanded on in this paper.

Couplet Hierarchical Vectors

The best way to understand CHVs is to start with an example. We will take a simple XML document, break it down into its couples, and then define the CHVs from the couplets. Each couplet will have an associated CHV. Along the way we will review the *Parent* and *P-Level* numbers associated with the couplets. These two numbers, along with the couplet line number and depth may be used to construct the CHVs.

Appendix 1 provides an example XML document followed by the flattened couplet structure. Carefully study the flattened couplet structure along with the *Parent* and *P-Level* numbers provided. Each couplet has a unique line number associated with it. In this particular case, the couplet line numbers are generated by numbering the couplets as they are extracted from the original XML document. Once additions, deletions and modifications are made, the couplet line numbers will not appear so nicely related to the original XML document. The key is that each couplet will retain a unique line number.

The *Parent* and *P-Level* numbers encode the hierarchical structure of the XML document in the flattened couplet space. As can be seen, each couplet captures the entire tag structure up to the given data element. As such, a given flattened line may contain the parent structure of several lines of the original XML document. The *Parent* number is the line number of the parent of the given couplet. The sticky issue here is that since a couplet contains the entire parent tag structure up to the data item, which parent are we referring to? Lets see if we can clarify this issue.

First, a few definitions and nomenclature. We will refer to the depth that a given item resides at. Look at line three of the flattened XML document. In the couplet tag structure, "Phonebook" is at depth 1, this is the shallowest depth and may be referred to as the root. Items go deeper from here. "Listing" is at depth 2, "Name" at depth 3, and so on out to the data item "Brandin", which is at depth 5. The couplet depth is the depth of the deepest item of the couplet, in this case it is 5. In the case there is no data item associated with the couplet, Null data, the depth of the couplet will be the depth of the tag structure plus 1. In most cases, this is a moot point. The parent of "Name" is "Listing", and the children of "Name" are "Last" and "First".

Now lets walk through the creation of the *Parent* number for several lines of the flattened XML document. Line one contains the root node of the document, "Phonebook". A parent does not exist for this item, so the *Parent* number points back to itself, 1, i.e. the parent of "Phonebook" is "Phonebook". This will be true only for the root node of a document: the *Parent* number will point back to itself. On line two, we see the tag "Listing". The parent of "Listing" is "Phonebook" which resides on line one, so *Parent* is 1. On line three, the parent of "Last" is

"Name", but "Name" occurs for the first time on the same line. It would be of little value to have *Parent* point back to itself. So we go back one level and look at "Name". The parent of "Name" is "Listing". "Listing" opened up on a previous line, so we will set *Parent* to the line "Listing" opened up on, line 2. On line four, the parent of "First" is "Name". Since "Name" opened up on a previous line, *Parent* will be 3. Looking at one more line, line five, lets start at the data depth, 5. "1502" is contained in "Number" which appears for the first time on line 5. So we look at "Number" whose parent is "Address", which also appears for the first time on this line. The parent of "Address" is "Listing", so *Parent* will be 2. It is worth your time to go through the rest of the document and see if you can justify each *Parent* number. Notice that as one "Listing" closes out in the XML document and another "Listing" opens, the *Parent* reference numbers change in the flattened document. By simply looking at the flattened document without the *Parent* number, it would be impossible, in general, to tell when these changes occur.

The *P-Level* works in conjunction with the *Parent*. The *P-Level* tells us at which depth a new tag hierarchy takes effect or opens up. Look at line two of the flattened document in Appendix 1. The *P-Level* is 2. This indicates that "Listing" appears for the first time or opens up on this line. On line three, the *P-Level* is 3 because "Name" opened up on this line and "Name" is at depth 3. The *P-Level* is defined as the depth of the first item on the couplet that opens up on that line. The *P-Level* captures the change in the hierarchy structure. If you lay the original XML document on its side, with the shallowest tags up, it can be seen that the *P-Level* captures the peaks of the opening tags. Now go through the rest of the flattened document and see if you can justify each of the *P-Level* numbers. Notice on line twelve that *P-Level* dropped to 2, indicating that "Listing" re-opened on this line.

Now we come to the Couplet Hierarchical Vector (CHV). The CHV is a vector of numbers that gives the entire heritage of a couplet in terms of the couplet line number of itself and all of its ancestors. The size of the CHV will be the depth of the couplet. For example, the CHV of the flattened line number three is:

$$chv_3 = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix} \tag{0.1}$$

The size of chv_3 is 5 because the depth of couplet three is 5. The first item in chv_3 represents the root node, "Phonebook", which opened on line 1. The second item represents "Listing", which opens up on line 2. The third item represents "Name", which opens up on line 3. Item 4 represents "Last", which opens up on line 3, and finally, item 5 represents "Brandin", which exists on line 3 also. Look at the next two examples from the flattened lines 15 and 26. See if you can justify the vector numbers.

$$chv_{15} = \begin{bmatrix} 1 \\ 12 \\ 15 \\ 15 \\ 15 \end{bmatrix} \qquad chv_3 = \begin{bmatrix} 1 \\ 22 \\ 24 \\ 26 \\ 26 \end{bmatrix}$$

The CHV can be built in a recursive manor using the *P-Level*, *Parent*, and couplet line number information. Start by setting the index d to the depth of the given couplet at line number n. The

set $chv_n[d] = n$. Decrement d. If $d <$ *P-Level* then set $chv_n[d] =$ *Parent* and go to the parent's flattened line. Else, set $chv_n[d] = n$. Repeat the process until the root node is reached, i.e. when $d = 1$. In this fashion, the CHV for a given couplet can always be constructed from the *Parent* and *P-Level* information, which means the CHV for couplets does not have to be stored.

Properties of Couplet Hierarchy Vectors

In this section we will establish a number of the properties of CHVs. This will allow us to determine virtual document order and to perform set manipulations based on CHVs. First we will note a couple of fundamental properties of XML documents that we will take advantage of.

The ordering of elements in an XML document at a given hierarchy level does not matter. We will consider two XML documents equivalent if they only differ by the ordering of elements within a given hierarchy. For example, the following two XML documents are considered equivalent:

Document 1:

```
<A>
        <B>
                <C1>c-1</C1>
                <C2>c-2</C1>
        </B>
        <D>
                <C1>c-3</C1>
                <C2>c-4</C1>
        </D>
</A>
```

Document 2:

```
<A>
        <D>
                <C2>c-4</C1>
                <C1>c-3</C1>
        </D>
        <B>
                <C2>c-2</C1>
                <C1>c-1</C1>
        </B>
</A>
```

The tag names uniquely identify elements within a hierarchy level so ordering imparts no new or implied information.

Each element in an XML document has a unique parent, and, if two elements have the same parent, then the rest of the ancestry for the two elements is the same. These two statements are obvious from the structure of an XML document.

Containment of elements within the same hierarchy in an XML document implies a relationship between the elements. The containment of "Last" and "First" within the "Name" element in our phone book example implies "Chris" and "Brandin" are related. In the same sense, an attribute element modifies the elements contained within the given hierarchy. "Residential" within the

"Listing" hierarchy implies the "Chris Brandin" is a resident of Colorado Springs and not a "Business" in Colorado Springs.

CHV Property 1: For any two CHVs and a given depth, *d*,

$$\begin{aligned} &\text{If} \quad chv_1[d] = chv_2[d] \quad \text{then} \\ &\qquad chv_1[i] = chv_2[i] \quad \text{for} \quad i = 1 \text{ to } d \end{aligned} \tag{0.2}$$

This property comes from the XML document property that if two elements have the same parent then the rest of the ancestry is the same. We will also say that when condition 1.2 is met, chv_1 and chv_2 are contained within the same hierarchy element at level *d* if the depths of the two CHVs are greater than *d*.

CHV Property 2: For any two CHVs whose depths are greater than *d*,

$$\begin{aligned} &\text{If} \quad chv_1[d] = chv_2[d] \quad \text{AND} \quad chv_1[d+1] \neq chv_2[d+1] \\ &\qquad \text{Then} \\ &chv_1 < chv_2 \quad \text{if} \quad chv_1[d+1] < chv_2[d+1] \\ &\qquad \text{Else} \\ &chv_1 > chv_2 \end{aligned} \tag{0.3}$$

If the depth of chv_1 equals *d* and the depth of chv_2 is greater than *d*,

$$\begin{aligned} &\text{and} \quad chv_1[d] = chv_2[d] \\ &\qquad \text{then} \\ &chv_1 < chv_2 \end{aligned} \tag{0.4}$$

Property two allows us to order a set of couplets based on the virtual XML document order. Virtual document order is the order that maintains the hierarchical structure of the XML document. The key to determining the order of two CHV's is to find the deepest depth, *d*, for which $chv_1[d] = chv_2[d]$. Then the depth, $d + 1$ will determine the order. If the root nodes are different, then the two CHV's come from different documents and $chv[1]$ may be used to determine document order. If chv_1 is not as deep as chv_2 and $chv_1[d] = chv_2[d]$ at the depth *d* of chv_1, then chv_2 must come after chv_1 and we say that chv_2 is contained within the chv_1 element.

Demonstrate with a couple of insertions into the example Document

We can see how the ordering of CHVs work by looking at what happens when we insert a couple of items into our phone book document. Suppose we decided to add middle initials for Chris and Harry and also the postfix Jr. for Harry. The flattened lines with somewhat arbitrary insertion point line numbers are:

58	Phonebook>Listing>Name>MidInitial>G	For Harry
97	Phonebook>Listing>Name>MidInitial>L	For Chris
126	Phonebook>Listing>Name>Post>Jr	For Harry

We can insert these lines in the proper virtual document order by setting the CHVs to:

$$chv_G = \begin{bmatrix} 1 \\ 12 \\ 13 \\ 58 \\ 58 \end{bmatrix} \qquad chv_L = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 97 \\ 97 \end{bmatrix} \qquad chv_{Jr} = \begin{bmatrix} 1 \\ 12 \\ 13 \\ 126 \\ 126 \end{bmatrix}$$

Notice that if we attempted to order these vectors at level 4 or 5 without taking into account the shallower levels, we would not maintain proper document order. Also notice that the vectors will not place the middle initial between the "Last" and "First" name within the hierarchy. This is not required. Remember that the tag structure contains the full description or context of the data item. The CHVs provide the hierarchical structure information.

CHV Property 3 (Containment): If chv_1 has depth d and chv_2 has a depth greater than d, then $chv_2 \in chv_1$ (chv_2 is an element of or is contained in chv_1) if $chv_2[d] = chv_1[d]$.

An example of vectors contained in other vectors which shows the hierarchy of the vectors is:

$$chv_G = \begin{bmatrix} 1 \\ 12 \\ 13 \\ 58 \\ 58 \end{bmatrix} \in \begin{bmatrix} 1 \\ 12 \\ 13 \\ 58 \end{bmatrix} \in \begin{bmatrix} 1 \\ 12 \\ 13 \end{bmatrix} \in \begin{bmatrix} 1 \\ 12 \end{bmatrix} \in \begin{bmatrix} 1 \end{bmatrix}$$

This brings up the interesting point that a single flattened line may represent several layers of the hierarchy and therefore may have several CHVs associated with it. For example, look at line 5 of the flattened XML document in appendix 1. Both "Address" and "Number" open up on this line and the data element "1502" is contained on this line. The related CHVs for this line are:

$$chv_5 = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 5 \\ 5 \end{bmatrix} \in \begin{bmatrix} 1 \\ 2 \\ 5 \\ 5 \end{bmatrix} \in \begin{bmatrix} 1 \\ 2 \\ 5 \end{bmatrix}$$

In most instances, we will say that the CHVs for this line are distinct and therefore not equal. There are some instances where we will not make a distinction between the different CHVs and therefore call them equal. This is a fine point that must be kept in mind when working with CHVs and when defining set operations with them.

Convergence, Duplicates and Comparing CHVs

Couplet hierarchy vectors are typically compared at a shallower depth than their defined depth. This is due to the nature of the hierarchical documents and what we are looking for within them. Suppose we are searching through the phonebook for the listing of Chris Brandin on East Pikes Peak Avenue. The three associated CHVs are:

$$chv_{Chris} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 4 \end{bmatrix} \quad chv_{Brandin} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 3 \\ 3 \end{bmatrix} \quad chv_{PikesPeak} = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \\ 6 \end{bmatrix}$$

Clearly these are distinct vectors and cannot be compared at their full depth of 5. If we compare these vectors at the "Listing" level, depth 2, we see that each vector is equal at depth 2, and so they are all from the same listing and therefore we have a match at that level.

One of the vector operators we will use is convergence. Convergence truncates a vector to a shallower depth, *d*. When we converge a vector to a depth, *d*, we are effectively tossing out or ignoring deeper level information (all values in the CHV greater than *d*). The examples seen in the last section (property 3 on containment) shows examples of CHVs being converged one level at a time. Notice how a converged vector contains the vector being converged. This operation is used often when working with documents.

Indexes

All items within the NeoCore XMS are found via indexes. A tag index for street number would contain all the line items for:

Phonebook>Listing>Address>Number

The index in this case would contain:

5, 15, and 24

Since the example phone book document was built in order, this index ordering follows virtual document order. If we used this method for middle initials, we might have the index order as:

58 and 97

While this orders the index by leaf node order, these two items are not in virtual document order, which is the ordering we would get if we used the CHVs. Set operations on hierarchical documents are most efficient if the sets are in virtual document order.

Since many of the sets are derived from the indexes, it is important that the indexes be in virtual document order.

Set Operations

Ordered sets allow set operations such as set intersection, set union, and other set operations that will be defined later, to be carried out much more efficiently than if the sets are not ordered. For instance, a binary search algorithm may be performed on an ordered set whereas a binary search cannot be performed on an unordered set.

We will call ***A*** an ordered set of CHVs if:

$$A = \{a_1 \quad a_2 \quad a_3 \quad \cdots \quad a_n\}$$
$$\text{where the elements are such that } a_j \geq a_i \text{ for all j>i} \tag{0.5}$$

A will be a properly ordered set if $a_j > a_i$ for all j>i .

If we have a properly ordered set ***A*** of CHVs, converging each element to a shallower depth will in general create duplicates within the set making the set simply ordered. This must be kept in mind during set operations. Often we will start with two properly ordered sets, and elements of one or both sets will be converged to lower levels when creating duplicates within that set. These duplicates often must be removed from the final output set.

Binary Search on an ordered set of CHVs

A binary search is a well know algorithm for finding an element within a given set. A binary search will find a given element within an ordered set of *n* elements (if it exist within the set) in less than or equal to $\log_2(n)+1$ steps. This is substantially faster than going through each element of a large set one-by-one comparing each element it to a target element. I will not give the details of a binary search here because it is such a well known algorithm. I will point out a few issues that have to do with binary searches on a set of CHVs though.

We will assume that ***A*** is an ordered set of CHVs for this discussion and chv_a is the item we are searching for. ***A*** may be properly ordered, but if we are comparing the elements of ***A*** at a depth that is shallower than the element's depths, duplicates may be introduced. The elements of ***A*** may have different depths associated with each element. The depth that each element in ***A***, a_i, is compare to chv_a at may be:

1) a defined depth *d*,
2) the depth of chv_a,
3) the depth of the element a_i.

The depth of comparison will depend on the definition of the set operation being performed. This is a place where the definition of equality between CHVs is important. See the section "Convergence, Duplicates and Comparing CHVs" above.

If $chv_a \in A$, then the binary search routine returns the index of the item found, along with a flag indicating a match was found. If there are duplicates of chv_a in ***A***, then the index of the first

duplicate element found will be returned. It will be the responsibility of the calling routine to check for duplicates around the item found if this is necessary.

If $chv_a \notin A$ then the binary search routine returns the index of the element that is just greater than or just less than chv_a, along with a flag indicating greater than or less than match. If j is the index returned, we will have one of the two cases:

$a_{j-1} < chv_a < a_j$ if *less than* is returned (a_j could be the first element of A)

$a_j > chv_a > a_{j+1}$ if *greater than* is returned (a_j could be the last element of A)

Binary search routines will be used heavily in the set operations described below.

A Fast Set Intersection and Union Algorithm

The NeoCore XMS (XML Management System) uses a variety of set operations when querying and processing XML information. The sets that are dealt with are often quite large, so efficient set operators are fundamental to NeoCore XMS. The easiest way to understand the basis of the various fast set operations is to study a simple set intersection operation.

Let ***A*** and ***B*** be two properly ordered sets:

$$A = \{a_1 \quad a_2 \quad a_3 \quad \cdots \quad a_n\} \text{ where } a_j > a_i \text{ for all } j > i$$

$$B = \{b_1 \quad b_2 \quad b_3 \quad \cdots \quad b_m\} \text{ where } b_j > b_i \text{ for all } j > i$$

For a set intersection we are looking for the properly ordered set ***C*** where:

$$C = A \bigcap B$$

First, if the sets ***A*** and ***B*** were not ordered, the set intersection process would require searching through all element of ***B*** for each element of ***A***, looking for matches. When a match is found, the element is added to set ***C***. This process would require on the order of $m \times n$ steps, which is terribly inefficient. Having ordered sets allows us to speed up this process considerably by using binary searches.

Using a binary search and comparing set ***A*** to set ***B***, for each element of ***A***, a_i, we would do a binary search of set ***B*** looking for the element a_i. If a_i is found in set ***B***, a_i will be added to set C. This process will take on the order of $n[\log_2(m)+1]$ steps. For a large set ***B***, this will be considerably faster than using unordered sets. Clearly to make this process as fast as possible, we will want to compare the smaller set against the larger set. Note that because ***A*** is a properly ordered set, the result set C will be a properly ordered set.

The next improvement comes from noting that since both ***A*** and ***B*** are ordered sets, once we perform a binary search on ***B*** looking for a_i, the next binary search can be on a smaller set. The binary search of ***B*** returns an index of ***B*** indicating a match or the closest match above or below a_i. Lets say the element found is b_j. When we go to look for a_{i+1} in the set ***B***, clearly $a_{i+1} \geq b_j$ so we may perform the binary search of a_{i+1} against the smaller set $B - \{b_1 \quad \cdots \quad b_{j-1}\}$. In other words, each binary search is preformed against a shrinking set of elements in ***B***, thus further reducing the number of steps required below $n[\log_2(m)+1]$ steps.

The next optimization is best described by looking at a concrete example. Consider the two sets:

$$A = \{5,6,7,9,15,18,47,48,49,86,105,107\}$$

$$B = \{9,10,11,12,48,91,92,93,97,105,133,138,150\}$$

We will start by looking for 5 in set ***B***. Since 5 is less than 9, 5 is not in ***B***, the search will return the index 1 with a *less than* flag (we did not even need a binary search to determine this). We could continue looking for 6 and 7 in set ***B***, or we can switch our search order and look for 9 in set ***A*** – {5} with a binary search which will return the index 4 (for element 9) with a match flag. Notice that by switching the search order, we were able to skip two searches that would not have produced matches.

At this point we can look for 10 in set ***A*** or 15 in set ***B***. Since our last search was an element of ***B*** against the set ***A***, we will continue with this process until there is a reason to change. So we look for 10 in the set:

$$A - \{5,6,7,9\} \quad \text{or the set:} \quad \{15,18,47,48,49,86,105,107\}$$

which returns the index for element 15 with a *less than* flag. This is our indication to reverse the search order again. We now look for 15 in the set ***B*** – {9,10} with a binary search which will result in finding either element 12 with a *greater than* flag, or 48 with a *less than* flag depending on implementation details of the binary search algorithm. If we land on 48 with a *less than* flag, or next search will be element 48 from set ***B*** against the reduced set ***A***. We continue this process until one of the sets runs out of elements.

Let's count up the number of steps for this example. A step consists of comparing an element of one set against the first element of the other set, a possible binary search and a reversal decision. The enumerated steps are:

1. Compare A5 against B9, reverse search order
2. Compare B9 against A6, Binary Search, find A9
3. Compare B10 against A15, reverse search order
4. Compare A15 against B11, Binary search find B48 <, reverse search order
5. Compare B48 against A18, Binary search, find A48
6. Compare B91 against A49, Binary search, find 86 <, reverse search order
7. Compare A105 against B92, Binary search, find B105
8. Compare A107 against B133, Stop Search.

The whole process took 8 steps and only 5 binary searches. This is less steps than the smaller of the two sets. Set ***A*** contains 12 elements. There were less than half as many binary searches required by this process than elements in the smaller of the two sets.

It is clear that by this process of reversing the search order, we will jump through the intersection of the two sets with the minimal number of steps and binary searches. In fact, we will use less binary searches than the number of elements in the smaller of the two sets, and the sets that we are doing binary searches against become smaller and smaller as the process continues. This makes for a very fast algorithm for set intersection.

A Fast Set Union Algorithm

The same process that was used to speed up set intersection may be used to speed up a set union. Here we will define the union of two properly ordered sets ***A*** and ***B*** to be a properly ordered set ***C*** than contains all of the elements of both ***A*** and ***B*** without including any duplicates.

$$C = A \bigcup B$$

If you think about it, the primary difference between the set intersection and the set union is what is kept. In the set intersection process, we searched through both sets looking for common elements and kept only one copy of the common elements and tossed all non-common elements. In set union, we have to search though both sets looking for common elements, keeping only one copy of the common elements, plus we keep (instead of tossing) all non-common elements. With this in mind, it is easy to see how to modify the fast intersection algorithm to perform the fast set union algorithm. Using the above example again,

$$A = \{5,6,7,9,15,18,47,48,49,86,105,107\}$$
$$B = \{9,10,11,12,48,91,92,93,97,105,133,138,150\}$$

this time taking the union, the steps are:

1. Compare A5 against B9, add A5 to C, reverse search order.
2. Compare B9 against A6, Binary Search, find A9, add A6 – A9 to C.
3. Compare B10 against A15, add B10 to C, reverse search order.
4. Compare A15 against B11, Binary search find B48 <, add B11 – B12 to C, add A15 to C, reverse search order.
5. Compare B48 against A18, Binary search, find A48, add A18 – A48 to C.
6. Compare B91 against A49, Binary search, find 86 <, add A49 – A86 to C, add B91 to C, reverse search order.
7. Compare A105 against B92, Binary search, find B105, add B92 – B105 to C.
8. Compare A107 against B133, add A107 to C, add B133 – B150 to C. Stop Search.

Notice that the union operation on the same sets resulted in the same number of steps and the same number of binary searches as the intersection operation. The only difference in the two processes is that the union operator kept more items.

Couplet Hierarchical Vector Set Operations

Set operations involving CHVs are similar to normal set operations, but they must be more carefully defined. The key issue is that the CHVs contain more information than simple numbers. The level that the CHV's are compared at must be defined and the level the CHV's are returned at must be defined. These levels depend on the underlying NeoCore XMS operation.

Get Index Set (Get Single Set)

This operation creates a properly ordered set ***A*** by retrieving an index set, converging the CHV's from the index set to a given depth, and removing all duplicate items.

As noted above in the section on indexes, an index contains a list of all line numbers (map offsets in NeoCore XMS) for something like a tag:

Phonebook>Listing>Address>Number: 5,15,24

If our interest in Address-Numbers is at the listing level, these items (5,15, and 24) would be converged back to the Listing level giving:

$$A = \{2,12,22\}$$

In this particular case, there were no duplicates introduced during the convergence process. In general though, a converge may introduce duplicate items. Because the indexes are established in virtual document order removing duplicates is a simple process. Before adding the next item to the set ***A***, the previous item in set ***A*** is compared, at the converged depth, to the new item. If the two items are equal, the new item is tossed, if they are not equal, the new item is added to ***A***.

CHV Set Intersection

This operation creates a properly ordered set C from the intersection of two properly ordered sets A and B of CHVs:

$$C = A \bigcap_{chv} B$$

The elements of ***A*** may be defined at various depths and the elements in ***B*** may also be defined at various depths. The elements $a \in A$ and $b \in B$ are considered equal if they are both defined as having the same depth, *d*, and $a[d] = b[d]$, otherwise they are not considered equal. The intersection of sets ***A*** and ***B*** may be carried out by the process given above using this definition.

CHV Set Union

This operation creates a properly ordered set C from the union of two properly ordered sets A and B of CHVs:

$$C = A \bigcup_{chv} B$$

Once again, the elements of ***A*** may be defined at various depths and the elements in ***B*** may also be defined at various depths. The elements $a \in A$ and $b \in B$ are considered equal if they are both defined as having the same depth, *d*, and $a[d] = b[d]$, otherwise they are not considered equal. The union of sets ***A*** and ***B*** may be carried out by the process given above using this definition.

Hierarchical Vector Correlation

This operation creates a properly ordered set C from the intersection of two properly ordered sets ***A*** and ***B*** of CHVs:

$$C = A \bigcap_{hvc} B$$

The elements of ***A*** may be defined at various depths. The elements in ***B*** may also be defined at various depths with the condition that depths of elements in ***B*** are greater than or equal to the depths of corresponding elements in ***A***. An element, *b*, of ***B*** corresponds to an element, *a*, of ***A*** if **b** is a child or sibling of *a* or could logically be a child or sibling of *a*. An example of this is that "City" is a child of "Address" in terms of the document structure.

The elements of C will be all elements of ***B*** that correlate with an element of ***A***. This is a set intersection process whereby the elements of ***B*** are compared with the elements of ***A*** at the depth of the elements of ***A***. If $b \in B$ matches $a \in A$ at *a*'s depth, then *b* is added to ***C***. The set intersection process defined above may be used with this definition to perform the hierarchical vector correlation.

Various other set operations may and have been defined using couplet hierarchical vectors. The set operations are variations of set intersection and set union operations where the depth of comparing the CHVs must be carefully defined along with the depth of the elements added to the resultant set. The above set operations provide examples of the type of operations that can be performed.

Conclusions

Couple hierarchical vectors provide effective and efficient mechanism to order, work with and manipulate hierarchical objects such as XML documents. Without virtual document ordering via CHV's, efficient set operations on these hierarchical objects are impossible. Binary search algorithms (or other similar algorithms) may be defined and used on sets of ordered CHV's. This makes a wide variety of set operations much faster.

A very efficient algorithm has been given for performing operations on ordered sets. This algorithm may provide orders of magnitude speed improvements over other set intersection and union methods depending on the structure of the items in the sets.

Several set operations have been defined based on hierarchical objects using CHV's. The defined set operations provide examples of the type of operations that may be performed on hierarchical objects such as XML documents.

Appendix 1

Example XML Document

```
1   <Phonebook City=Colorado Springs>
2       <Listing category=Residential>
3           <Name>
4               <Last> Brandin </Last>
5               <First> Chris </First>
6           </Name>
7           <Address>
8               <Number> 1502 </Number>
9               <Street> East Pikes Peak Avenue </Street>
10              <City> Colorado Springs </City>
11              <State> CO </State>
12              <Zip> 80909 </Zip>
13          </Address>
14          <Telephone>
15              <Areacode> 719 </Areacode>
16              <Number> 630-1206 </Number>
17          </Telephone>
18      </Listing>
19      <Listing category=Residential>
20          <Name>
21              <Last> Direen </Last>
22              <First> Harry </First>
23          </Name>
24          <Address>
25              <Number> 2750 </Number>
26              <Street> North Gate Rd </Street>
27              <City> Colorado Springs </City>
28              <State> CO </State>
29              <Zip> 80921 </Zip>
30          </Address>
31          <Telephone>
32              <Areacode> 719 </Areacode>
33              <Number> 495-0589 </Number>
34          </Telephone>
35      </Listing>
36      <Listing category=Business>
37          <Name> NeoCore </Name>
38          <Address>
39              <Number> 2864 </Number>
40              <Street> South Circle Drive </Street>
41              <Suite> 1200 </Suite>
42              <City> Colorado Springs </City>
43              <State> CO </State>
44              <Zip> 80906 </Zip>
45          </Address>
46          <Telephone>
47              <Areacode> 719 </Areacode>
48              <Number> 576-9780 </Number>
49          </Telephone>
50      </Listing>
51  </Phonebook>
```

Flattened XML Document

This is the flattened couplet version of the above XML document. The underlined portions are metadata, and the non-underlined portions are data elements.

Line	Couplet (Metadata/Data)	Parent	P-Level	Depth
1	Phonebook>@City>Colorado Springs	1	1	3
2	Phonebook>Listing>@category>Residential	1	2	4
3	Phonebook>Listing>Name>Last>Brandin	2	3	5
4	Phonebook>Listing>Name>First>Chris	3	4	5
5	Phonebook>Listing>Address>Number>1502	2	3	5
6	Phonebook>Listing>Address>Street>East Pikes Peak Avenue	5	4	5
7	Phonebook>Listing>Address>City>Colorado Springs	5	4	5
8	Phonebook>Listing>Address>State>CO	5	4	5
9	Phonebook>Listing>Address>Zip>80909	5	4	5
10	Phonebook>Listing>Telephone>Areacode>719	2	3	5
11	Phonebook>Listing>Telephone>Number>630-1206	10	4	5
12	Phonebook>Listing>@category>Residential	1	2	4
13	Phonebook>Listing>Name>Last>Direen	12	3	5
14	Phonebook>Listing>Name>First>Harry	13	4	5
15	Phonebook>Listing>Address>Number>2750	12	3	5
16	Phonebook>Listing>Address>Street>North Gate Rd.	15	4	5
17	Phonebook>Listing>Address>City>Colorado Springs	15	4	5
18	Phonebook>Listing>Address>State>CO	15	4	5
19	Phonebook>Listing>Address>Zip>80921	15	4	5
20	Phonebook>Listing>Telephone>Areacode>719	12	3	5
21	Phonebook>Listing>Telephone>Number>495-0589	20	4	5
22	Phonebook>Listing>@category>Business	1	2	4
23	Phonebook>Listing>Name>NeoCore	22	3	4
24	Phonebook>Listing>Address>Number>2864	22	3	5
25	Phonebook>Listing>Address>Street>South Circle Drive	24	4	5
26	Phonebook>Listing>Address>Suite>1200	24	4	5
27	Phonebook>Listing>Address>City>Colorado Springs	24	4	5
28	Phonebook>Listing>Address>State>CO	24	4	5
29	Phonebook>Listing>Address>Zip>80906	24	4	5
30	Phonebook>Listing>Telephone>Areacode>719	22	3	5
31	Phonebook>Listing>Telephone>Number>576-9780	30	4	5

# # #