

# **Xpiori XML Document Storage and Indexing Engine Architecture**

**by Chris Brandin**

**Release 1.1**

**Xpiori, LLC  
2864 S. Circle Dr.  
Ste. 1200  
Colorado Springs, CO 80906  
(719) 527-1315  
[www.xpiori.com](http://www.xpiori.com)**

© 2007 by Xpiori, LLC. All rights reserved.

**Version 1.1**

**Copyright © Xpiori, LLC All Rights Reserved**

**Xpiori 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.**

**The information in this white paper has been provided by Xpiori, LLC. To the best knowledge of Xpiori, it contains information concerning the current state of information processing technology. Xpiori, LLC disclaims any and all liabilities for and makes no warranties, expressed or implied, with respect to products described in this paper, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose. No specific reliance should be made on the material provided herein without thorough investigation of the technology and its proposed application to specific circumstances. Product and technology information is subject to change without notice.**

## Introduction

This paper discusses a method for storing, managing, and indexing XML documents using Digital Pattern Processing (DPP) technology. It is assumed that the reader has a working knowledge of DPP.

There are two methods generally employed to process XML data: SAX and DOM. The SAX method assumes that you have a predetermined set of signatures to search for, and that the XML data is temporal. As the signatures are encountered in an XML data stream, events are triggered to respond to the presence of those signatures. SAX is typically used to parse XML data "on-the-fly". The DOM method is almost the architectural antithesis of SAX; it assumes that you have a block of XML data and wish to make ad-hoc queries against it – more like what one would do with a database.

The method disclosed here goes well beyond either SAX or DOM. In order to have a viable XML data repository, what is really needed is a "super-DOM", along with several features typically found in database management systems, and powerful pattern matching capabilities.

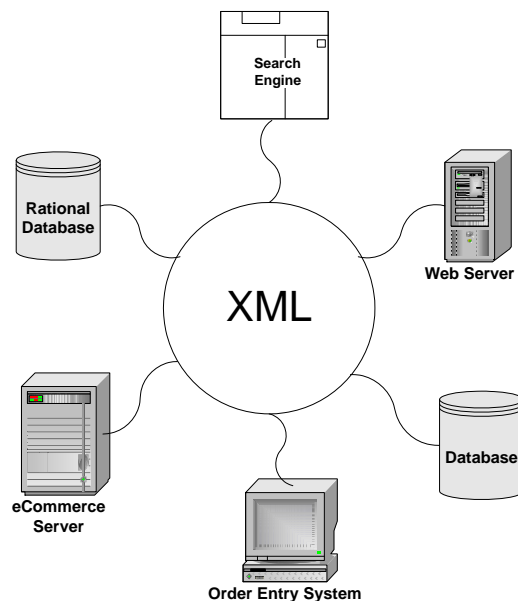
Xpiori has endeavored to produce such an XML data repository server. In the interest of clarity, this paper discusses significant architectural issues without delving too deeply into implementation topics.

This paper is divided into two sections. The first – "The Nature of Information and XML" – discusses XML from an information theory standpoint. The second – "Managing XML Data Using Digital Pattern Processing and Hierarchical Vector Annealing" – discusses architectural issues surrounding the implementation of an XML data repository server.

## **Section One: The Nature of Information and XML**

XML was developed as a standard for representing information. As a simplified descendant of SGML, it has enjoyed virtually universal acceptance as a means to exchange information between computers. The fundamental characteristic of XML that has made this possible is that it expresses data in a self-describing manner; that is, data is accompanied by context (or "metadata"). XML consists of "Tags" (metadata) and Data Elements (data). Although other components are defined (attributes and namespaces, for example), they are all fundamentally either data or metadata. It is reasonable to define information, in general, as being composed of both data and metadata. A Database Management System (DBMS) requires that all metadata be created in advance, and then data can be accommodated dynamically. Although DBMS's have served well for most data processing tasks up to now, for many applications they are fundamentally inadequate. Imagine if I were to walk up to you and simply blurt out the word "Alice". You would have no idea what I meant. If I said, "My wife is named" (metadata) "Alice" (data), you would know exactly what I meant. The problem with the DBMS paradigm is that all metadata has to be defined in advance (usually using rows and columns). If one looks at information as data coupled with metadata, it becomes apparent that DBMS's only solve half of the problem. As long as you know half of the information in advance, the other half can be managed later. Clearly, this is not adequate for many modern applications.

In order for computers to be able to exchange information, an independent self-describing means of representing data must be used to bridge the gap between the ways different computers characterize data elements. This is accomplished with XML by defining a common way to describe data, and by using mapping means to translate to the respective DBMS's syntax. The diagram below shows XML as the connecting fabric between a variety of data processing devices:



The curious thing about this diagram is that the XML circle in the middle doesn't exist physically. Consider the following:

In order to provide the means for all people to be able to communicate, a new language is created. This language is capable of expressing the sum total of everything that can be expressed in all languages. Furthermore, it can be used to express new things that no other language has the means to accommodate. Now, is it rational to suppose that

this new language will forever be devoid of literature, and that no information will ever be originated in it?

Clearly, even if this new language was not intended to be anything but an intermediate translation means, it would eventually be used on its own because it can be used to express what no other language can. For the same reason, XML will become its own domain.

## XML Syntax

Considering the impact it has had on the computer industry, XML has a strikingly simple syntax. Fundamentally, there are three types of elements: Tags, Data Elements, and Attributes. Additionally, there are a few other data types that are not germane to understanding the basics of XML, so they will not be discussed here. A typical XML fragment might look like the following:

```
<Xpiori>
  <Employee division = Corporate>
    <Name>
      <Last> Brandin </Last>
      <First> Chris </First>
    </Name>
    <Name>
      <Last> Dix </Last>
      <First> Tim </First>
    </Name>
  </Employee>
</Xpiori>
```

The items surrounded by angle-brackets ("`<>`") are Tags. The items outside the brackets are Data Elements ("Brandin", "Chris", "Dix", and "Tim" in this example). Tags can also contain Attributes ("Division = Corporate", in this example). Basically, Tags are metadata and Data Elements are data. Attributes are actually another metadata/data pair that are contained within a Tag – sort of a fractal representation of Tag/Data pairs. The effect of this is that Attributes "belong" to everything in the hierarchy below them, whereas Data Elements only "belong" to themselves and everything in the hierarchy above.

Fundamentally, that's what XML amounts to. Inside of this very simple construct, however, lies no less than an information revolution. Now we have a means to express information in its totality, not just the "data" portion. In order to take full advantage of the extensibility feature (the self-description) of XML, we must also devise a way to store and manage XML information. Now the XML circle in the diagram above becomes something physical – an XML data repository server.

## XML Information

Although XML is highly structured, it is not pre-structured. New data types can be added to an XML document simply by adding the appropriate Tag/Data information. In the DBMS world, there is no efficient way to do this because data types must be pre-defined. Object-relational database systems have attempted to mitigate this, but they have proven inadequate. Because of new business model requirements (like workflow management), it is becoming increasingly important to be able to deal with not only dynamic data, but *dynamic data types* as well. In order to deal with new data types in real time, a new model for information management had to be developed – “Information Couplets”.

XML is very simple and elegant, but it is not a natural way to express information. Tags do not constitute a complete form of metadata. The Tag – “<Last>” – doesn’t mean anything by itself. The metadata that belongs with “Brandin” is actually all the tags in the hierarchy above “Brandin”; that is “Xpiori + Employee + Name + Last”. Representing Tags in a hierarchical manner is linguistically convenient, but not necessarily optimal from an information standpoint. Because metadata provides a context for data, it must be complete. For example, although the phrase “last name” consists of two words, it is actually a singular piece of metadata; the use of two words is a matter of linguistic convention.

An “Information Couplet” is a metadata/data pair, where the metadata is a complete context. The XML fragment shown above, becomes the following when “flattened” into Information Couplets:

	Couplet Metadata	Data	ELevel
1.	Xpiori>Employee>@division>	Corporate	3
2.	Xpiori>Employee>Name>Last>	Brandin	4
3.	Xpiori>Employee>Name>First>	Chris	2
4.	Xpiori>Employee>Name>Last>	Dix	4
5.	Xpiori>Employee>Name>First>	Tim	2

The ELevel parameter is used to aid in the reconstruction of correctly formed XML and is discussed later. The “@” character indicates an Attribute instead of a Data Element. All information about a metadata/data pair is contained in each Information Couplet, and the information therein can be directly mapped into a DBMS (assuming the metadata maps to a pre-defined data type).

Now that information has been converted to a “complete” form, the issue of data-bloat needs to be addressed. A simple way to reduce storage overhead is to tokenize the metadata. Because metadata tends to be highly redundant, an indexed dictionary can be used. The dictionary for the example above would contain three entries:

Token	String	ELevel
#1	Xpiori>Employees>@division>	3
#2	Xpiori>Employee>Name>Last>	4
#3	Xpiori>Employee>Name>First>	2

The ELevel parameter can be dealt with in a couple of ways: it can be contained in the “flattened” Information Couplet document, or it can be folded into the dictionary (as shown above). Once metadata has been tokenized, the Information Couplet document above becomes:

Couplet	Metadata	TokenData
1.	#1	@division
2.	#2	Brandin
3.	#3	Chris
4.	#2	Dix
5.	#3	Tim

Data can also be tokenized, and will usually take up less space that way. The data element dictionary would contain the following entries:

Data Token	String
#1	Corporate
#2	Brandin
#3	Chris
#4	Dix
#5	Tim

Now the Information Couplet document becomes:

Couplet	Metadata	TokenData	Token
1.	#1		#1
2.	#2		#2
3.	#3		#3
4.	#2		#4
5.	#3		#5

The Information Couplet document is now a highly compressed, fixed-format, representation of the XML document. From this, the original XML can be quickly reconstructed.

So far, we have not made any use of DPP. If we want to do anything more with XML (like querying and extracting information), indexing and navigation capabilities must be added. By using DPP, this can be accomplished with very little overhead.

## Indexing and Navigating XML Documents

Managing and querying against a large heterogeneous XML document repository requires a combination of indexing and navigation features. Navigation is typically accomplished by using a DOM; they are very slow and occupy a *huge* amount of memory. Using DPP, a *numerical DOM* can be built that uses almost no additional memory at all. The key to this is to add DOM capability to the Information Couplet document instead of treating it as a separate module. To accomplish this, two new fields are added to the Information Couplet document:

Couplet	Metadata	TokenData	Token	Parent	PLevel
1.	#1		#1	1	1
2.	#2		#2	1	2
3.	#3		#3	2	4
4.	#2		#4	1	2
5.	#3		#5	4	4

The Parent parameter points to the current Couplet's parent (the Couplet containing the parent Tag). PLevel indicates the start of all the Tags that correspond to parent tags. Because the XML document has been flattened to fewer lines, some lines contain their own parent Tags. To clarify this we will examine an expanded version of the document above (the underlined portions correspond to parent tags indicated by PLevel):

Couplet	Metadata	Data	ELevel	Parent	PLevel
1.	<u>Xpiori&gt;Employee&gt;@division&gt;</u>	Corporate	3	1	1
2.	Xpiori> <u>Employee&gt;Name&gt;Last&gt;</u>	Brandin	4	1	2
3.	Xpiori>Employee>Name> <u>First&gt;</u>	Chris	2	2	4
4.	Xpiori> <u>Employee&gt;Name&gt;Last&gt;</u>	Dix	4	1	2
5.	Xpiori>Employee>Name> <u>First&gt;</u>	Tim	2	4	4

The combination of Parent and PLevel are used to accomplish set intersection convergences using a technique called "Hierarchal Vector Annealing"; this will be discussed in detail in the second section of this paper.

When an XML document is parsed, it is necessary to locate dictionary entries via an index. A NeoData Store is used to do this using DPP technology. Aside from offering an efficient means to index the dictionaries, the use of a NeoData Store offers additional advantages. Icon Algebra can be used to decompose and reconstruct different Tag name combinations, aiding in translation and aliasing.

The ELevel parameter can be easily calculated by looking at the next line's PLevel parameter - taking account of Parameters and always setting the last line to an ELevel of 1 (this is always done anyway). Thus, because PLevel parameters have been added to the Information Couplet document, the ELevel parameters can be eliminated. The result is that the Information Couplet document can now accomplish all navigational tasks without having to access the dictionaries at all. This amounts to a numerical DOM, where navigational tasks are completed using only small, fixed-length Information Couplets. Furthermore, fewer navigation steps are necessary because hierarchies are collapsed. Data set intersections always take the same amount of time, irrespective of document size (unlike conventional DOMs, where document size severely degrades performance). Performance is radically improved - often by six or more orders-of-magnitude.

In order to make the numerical DOM work, all Information Couplet entries have to be indexed. In a conventional DOM, navigation starts at parent nodes and ripples up through child nodes. With a numerical DOM, convergence happens in the opposite direction; navigation starts with tag/data Couplets anywhere in the document and they are converged at their parents. Using DPP, indices are built using very little memory. The index overhead is a flat 96-bits per unique entry, and 40-bits for each duplicate. Using Icon Algebra, it is possible to directly access any node in any XML document - using both fully and partially qualified queries – instantly. Complex queries are serviced by a combination of direct indexing and numerical DOM navigation.

## Memory Use

It is impossible to specify a compression ratio applied to XML documents using this numerical DOM method because compression and navigation functions have been collapsed into one system. Generally, the numerical DOM (which, unlike a conventional DOM, contains the XML documents) is considerably smaller than the original XML, and the indices are too. When all elements are combined, the total memory footprint is much smaller than with other methods of managing XML documents.

## Aliasing and Translation

Aliasing and translation are related in that they both involve changing Tags. Consider the circumstance where a query against multiple documents needs to be made: find all documents that pertain to the customer named "John Smith". In one document, customers may be called "client", and in another "customer". An aliasing mechanism makes it possible to find all tags that are equivalent to each other.

Another feature supported by Xpiori's aliasing mechanism involves searching for data based on unqualified queries. The primary indices built to support a numerical DOM consist of an index of complete metadata ("Xpiori>Employee>Name>Last>", for example), an index of complete couplets ("Xpiori>Employee>Name>Last>Dix"), and optionally an index of just data ("Dix"). An aliasing mechanism is used in order to service partially qualified tag combinations ("\*Employee>\*Dix"). This is achieved by building an alias index against the metadata dictionary, rather than indexing every permutation of the XML information. The metadata dictionary is typically very small, so the number of entries necessary to alias every permutation of unqualified Tags remains small. Basically the process consists of looking up every fully qualified tag combination that corresponds to the unqualified query. A list of fully qualified metadata Tags is returned, and a query is constructed using those fully qualified Tag combinations. This process can be encapsulated in the XML server, so the user need have no knowledge of the process.

Translation can be accommodated by adding one level of indirection to the dictionary pointers (Tokens) in the Information Couplet document. The Tokens contained in Couplets now point to an array of pointers that point to dictionary entries. Multiple dictionaries are built (with their own pointer arrays), one for each language. The user then selects the appropriate dictionary according to the language desired. In this mechanism, Couplets contain a symbolic representation of information that is language independent. The additional processing required to support this feature is negligible.

## Content Scanning

DPP technology is used to support full three-dimensional content searching in the XML data repository server. Content scanning takes on a somewhat different character in XML documents. Normally, one looks at content scanning as a process where signatures are located in a big buffer of data. XML documents rarely contain large blocks of data as data elements. Typically, they consist of a collection of smaller units of data where signatures do not span data elements. This offers the opportunity to approach wildcard searches against data elements in a couple of different ways.

If anything is known about where the data elements to be searched might be, a targeted string search can be executed. Xpiori's three-dimensional content scanning can accept a list of data buffers to be searched. If, for example, the query consisted of anybody named "Smith\*" (meaning Smith, Smithe, Smithson, etc...), one would assume that the string search should be applied only to data elements that are names. The aliasing mechanism is used to restrict the string searching process to only the appropriate data elements.

When the wildcard search is completely unqualified (find "\*\*Smit\*" anywhere in any document, for example), a full three-dimensional content scan is executed against the data dictionary, not the XML or Information Couplet documents. Because dictionaries only contain one instance of each data element (no duplicates), a complete search doesn't require seeing all the data even once. An interesting feature of the Xpiori three-dimensional content scanning technology is that any number of signatures can be searched for simultaneously. That means the time it takes to complete a search is always the same (according to dictionary size), irrespective of how many searches are being serviced at once.

Content scans can also be executed against the metadata dictionary. As these files tend to be very small (metadata is highly redundant), outrageous wild-card searches can be executed against Tags almost instantaneously.

## **Section Two: Managing XML Data Using Digital Pattern Processing and Hierarchical Vector Annealing**

This section discusses the mechanics of building an XML data repository server using DPP. The major components are described, along with a step-by-step description of how Hierarchical Vector Annealing works.

### **Indexing Instructions**

In order to control the behavior of a Xpiori XML Server, documents may be accompanied by a block of XML data specifying how the data is to be indexed and (optionally) how the document should be stored:

```
<Xpiori_XML_Index_Server>
  <NoIndex [start=##] [end=##]> Tag\...Tag[\] </NoIndex>
  <Index [start=##] [end=##] [duplicates=##]> Tag\...[Attribute_]Tag[\] </Index>
</Xpiori_XML_Index_Server>
```

The "Index" and "NoIndex" elements have optional attributes called "start" and "end". These attributes indicate which element or attribute names and (optionally) whether the data element should be included in the item. Usually the "start" value is a negative number indicating how many names in the hierarchy to include. A 0 corresponds to the end of the current name, so a 0 excludes all names, a -1 includes the current name; -2 includes the parent element name as well. The "end" attribute works the same way except it can also be a positive value; a 0 indicates the index entry ends with the current name, a 1 indicates the data element should be included. Note that there is a difference between using unqualified names in the description and using the "start" and "end" attributes against a fully qualified tag string. Using attributes will apply the rule only against the fully qualified tag string, even if the resulting index entry is not fully qualified. Using a partially qualified tag string will apply the rule to any name combination that matches.

Tag names are separated by a "\". If the tag hierarchy can lead into a data element a "\" should be added to the end. Parameter entries are signified by a leading "@". For example if an XML snippet looked like this:

```
<Employee>
  <Name>
    <First> Chris </First>
```

The entry to create a fully qualified index entry (including the data element) would be: "<Index start=-3 end=1> Employee\Name\First\ </Index>". If you wanted to not include the data element ("Chris") the "end" attribute would be set to 0. If you wanted to partially qualify the index entry excluding "Employee", the "start" attribute would be set to -2.

The default behavior of the XML Index Server is to index all data elements if no "Index" item is specified. If it is desired to not index a data element, it can be excluded with the "NoIndex" tag.

The "Index" tag indicates that an index entry should be created according to the given attribute and tag string. If an "Index" entry is specified, then the XML Index Server's default behavior changes: data elements will not be indexed unless specified. The "duplicates" attribute governs how duplicate entries are handled (discussed in another paper). A OR'ed bitmap is used where

1=method 1, 2 = method 2, and 4 = method 3. All methods can be supported in any combination. When no "duplicates" attribute is specified, method 1 is used.

### Original XML Document for Examples

```
1      <Phonebook country=USA>
2          <Listing category=Residential>
3              <Name>
4                  <Last> Brandin </Last>
5                  <First> Chris </First>
6              </Name>
7              <Address>
8                  <Number> 1234 </Number>
9                  <Street> Main Street </Street>
10                 <City> Colorado Springs </City>
11                 <State> CO </State>
12                 <Zip> 80909 </Zip>
13             </Address>
14             <Telephone>
15                 <Areacode> 719 </Areacode>
16                 <Number> 555-1206 </Number>
17             </Telephone>
18         </Listing>
19         <Listing category=Residential>
20             <Name>
21                 <Last> Brandin </Last>
22                 <First> Alice </First>
23             </Name>
24             <Address>
25                 <Number> 1234 </Number>
26                 <Street> Main Street </Street>
27                 <City> Colorado Springs </City>
28                 <State> CO </State>
29                 <Zip> 80909 </Zip>
30             </Address>
31             <Telephone>
32                 <Areacode> 719 </Areacode>
33                 <Number> 555-1061 </Number>
34             </Telephone>
35         </Listing>
36         <Listing category=Business>
37             <Name> Xpiori </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 an intermediate format used to derive the internal files to store and index XML data. The underlined portions are metadata, and the non-underlined portions are data elements.

Line	Metadata/Data	ELevel
1	<u>Phonebook</u> > <u>@country</u> >USA	2
2	<u>Phonebook</u> > <u>Listing</u> > <u>@category</u> >Residential	3
3	<u>Phonebook</u> > <u>Listing</u> > <u>Name</u> > <u>Last</u> >Brandin	4
4	<u>Phonebook</u> > <u>Listing</u> > <u>Name</u> > <u>First</u> >Chris	3
5	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Number</u> >1234	4
6	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Street</u> >Main Street	4
7	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>City</u> >Colorado Springs	4
8	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>State</u> >CO	4
9	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Zip</u> >80909	3
10	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Areacode</u> >719	4
11	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Number</u> >555-1206	2
12	<u>Phonebook</u> > <u>Listing</u> > <u>@category</u> >Residential	3
13	<u>Phonebook</u> > <u>Listing</u> > <u>Name</u> > <u>Last</u> >Brandin	4
14	<u>Phonebook</u> > <u>Listing</u> > <u>Name</u> > <u>First</u> >Alice	3
15	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Number</u> >1234	4
16	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Street</u> >Main Street	4
17	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>City</u> >Colorado Springs	4
18	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>State</u> >CO	4
19	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Zip</u> >80909	3
20	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Areacode</u> >719	4
21	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Number</u> >555-1061	2
22	<u>Phonebook</u> > <u>Listing</u> > <u>@category</u> >Business	3
23	<u>Phonebook</u> > <u>Listing</u> > <u>Name</u> >Xpiori	3
24	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Number</u> >2864	4
25	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Street</u> >South Circle Drive	4
26	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Suite</u> >1200	4
27	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>City</u> >Colorado Springs	4
28	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>State</u> >CO	4
29	<u>Phonebook</u> > <u>Listing</u> > <u>Address</u> > <u>Zip</u> >80906	3
30	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Areacode</u> >719	4
31	<u>Phonebook</u> > <u>Listing</u> > <u>Telephone</u> > <u>Number</u> >576-9780	1

## Dictionary File Contents

A dictionary file is an array of strings. Each unique tag string and data element string from the flattened XML is stored in the appropriate dictionary. Tokens are shown below as sequential numbers for clarity; they are actually the locations of strings in the dictionary file and are not stored.

### Metadata Dictionary:

Token	Associated String
#01	Phonebook>@country>
#02	Phonebook>Listing>@category>
#03	Phonebook>Listing>Name>Last>
#04	Phonebook>Listing>Name>First>
#05	Phonebook>Listing>Address>Number>
#06	Phonebook>Listing>Address>Street>
#07	Phonebook>Listing>Address>City>
#08	Phonebook>Listing>Address>State>
#09	Phonebook>Listing>Address>Zip>
#10	Phonebook>Listing>Telephone>Areacode>
#11	Phonebook>Listing>Telephone>Number>
#12	Phonebook>Listing>Name>
#13	Phonebook>Listing>Address>Suite>

### Data Dictionary:

Token	Associated String
#01	USA
#02	Residential
#03	Brandin
#04	Chris
#05	1234
#06	Main Street
#07	Colorado Springs
#08	CO
#09	80909
#10	719
#11	555-1206
#12	Alice
#13	555-1061
#14	Business
#15	Xpiori
#16	2864
#17	South Circle Drive
#18	1200
#19	80906
#20	576-9780

## Dictionary nStore Contents

The dictionary nStore is used to locate string entries in the dictionary file above. The Token entry is a pointer into the dictionary file. The Count value indicates how many times each string is used – it is used for maintenance purposes.

### Metadata nStore:

Indexed Item (Iconized)	Association	
	Token	Count
Phonebook>@country>	#01	1
Phonebook>Listing>@category>	#02	3
Phonebook>Listing>Name>Last>	#03	2
Phonebook>Listing>Name>First>	#04	2
Phonebook>Listing>Address>Number>	#05	3
Phonebook>Listing>Address>Street>	#06	3
Phonebook>Listing>Address>City>	#07	3
Phonebook>Listing>Address>State>	#08	3
Phonebook>Listing>Address>Zip>	#09	3
Phonebook>Listing>Telephone>Areacode>	#10	3
Phonebook>Listing>Telephone>Number>	#11	3
Phonebook>Listing>Name>	#12	1
Phonebook>Listing>Address>Suite>	#13	1

### Data nStore:

Indexed Item (Iconized)	Association	
	Token	Count
USA	#01	1
Residential	#02	2
Brandin	#03	2
Chris	#04	1
1234	#05	2
Main Street	#06	2
Colorado Springs	#07	3
CO	#08	3
80909	#09	2
719	#10	3
555-1206	#11	1
Alice	#12	1
555-1061	#13	1
Business	#14	1
Xpiori	#15	1
2864	#16	1
South Circle Drive	#17	1
1200	#18	2
80906	#19	1
576-9780	#20	1

## Tokenized XML Document with Level Indicators and Parent Pointers

The Metadata and Data values are pointers into the dictionary file. The Parent value points to the line in the tokenized XML document from where a regular XML document can be reconstructed including the current entry's parent tags (with attributes) and, optionally, its siblings. The PLevel value indicates the lowest level tag for the current Couplet that is a parent. The Parent and PLevel values are used to achieve set convergence based on query parameters.

Couplet	Metadata	TokenData	Token	Parent	PLevel
1	#01		#01	1	1
2	#02		#02	1	2
3	#03		#03	2	3
4	#04		#04	3	4
5	#05		#05	2	3
6	#06		#06	5	4
7	#07		#07	5	4
8	#08		#08	5	4
9	#09		#09	5	4
10	#10		#10	2	3
11	#11		#11	10	4
12	#02		#02	1	2
13	#03		#03	12	3
14	#04		#12	13	4
15	#05		#05	12	3
16	#06		#06	15	4
17	#07		#07	15	4
18	#08		#08	15	4
19	#09		#09	15	4
20	#10		#10	12	3
21	#11		#11	20	4
22	#02		#14	1	2
23	#12		#15	22	3
24	#05		#16	22	3
25	#06		#17	24	4
26	#13		#18	24	4
27	#07		#07	24	4
28	#08		#08	24	4
29	#09		#19	24	4
30	#10		#10	22	3
31	#11		#20	30	4

## XML Index nStore

The index nStore is used to directly access items in the XML Information Couplet document. Indexed items may, optionally, be converted to lower-case before being iconized. The data element portions of index entries may also have white-space characters processed or removed.

Indexed Item (Iconized)	Association Couplet
Phonebook>@country>USA	1
Phonebook>Listing>@category>Residential	2
Phonebook>Listing>Name>Last>Brandin	3
Phonebook>Listing>Name>First>Chris	4
Phonebook>Listing>Address>Number>1234	5
Phonebook>Listing>Address>Street>Main Street	6
Phonebook>Listing>Address>City>Colorado Springs	7
Phonebook>Listing>Address>State>CO	8
Phonebook>Listing>Address>Zip>80909	9
Phonebook>Listing>Telephone>Areacode>719	10
Phonebook>Listing>Telephone>Number>555-1206	11
Phonebook>Listing>@category>Residential	12
Phonebook>Listing>Name>Last>Brandin	13
Phonebook>Listing>Name>First>Alice	14
Phonebook>Listing>Address>Number>1234	15
Phonebook>Listing>Address>Street>Main Street	16
Phonebook>Listing>Address>City>Colorado Springs	17
Phonebook>Listing>Address>State>CO	18
Phonebook>Listing>Address>Zip>80909	19
Phonebook>Listing>Telephone>Areacode>719	20
Phonebook>Listing>Telephone>Number>555-1061	21
Phonebook>Listing>@category>Business	22
Phonebook>Listing>Name>Xpiori	23
Phonebook>Listing>Address>Number>2864	24
Phonebook>Listing>Address>Street>South Circle Drive	25
Phonebook>Listing>Address>Suite>1200	26
Phonebook>Listing>Address>City>Colorado Springs	27
Phonebook>Listing>Address>State>CO	28
Phonebook>Listing>Address>Zip>80906	29
Phonebook>Listing>Telephone>Areacode>719	30
Phonebook>Listing>Telephone>Number>576-9780	31

## Converging Sets and Returning an XML Fragment

Queries consist of data elements with their associated tags, and instructions indicating what should be returned. For example, a query against the above document may consist of the following:

- Selection Items:
  1. <Phonebook country=USA>
  2. <Phonebook> <Listing category=Business>
  3. <Phonebook> <Listing> <Name> Xpiori
  4. <Phonebook> <Listing> <Address> <City> Colorado Springs
  5. <Phonebook> <Listing> <Address> <State> CO
- Item to return:
  1. <Phonebook> <Listing>

The following entries are created to access the XML index nStore returning:

Indexed Item (Iconized)	Association
	Couplet
Phonebook>@country>USA	1
Phonebook>Listing>@category>Business	22
Phonebook>Listing>Name>Xpiori	23
Phonebook>Listing>Address>City>Colorado Springs	27
Phonebook>Listing>Address>State>CO	28

The tokenized XML document is accessed using the Couplet value and returns the following items:

Couplet	Metadata	TokenData	Token	Parent	PLevel
1	#01		#01	1	1
22	#02		#14	1	2
23	#12		#15	22	3
27	#07		#07	24	4
28	#08		#08	24	4

The set convergence is accomplished using a technique called "Hierarchal Vector Annealing" with the following steps:

1) The desired item to return is "<Phonebook> <Listing>". First all selection items containing these tags must converge to the same couplet that also contains these tags and has a PLevel value of 2 or lower. Items 2-5 fit these criteria.

2) Both couplets 28 and 27 have a parent value of 24; that Couplet is fetched from the XML Information Couplet document:

Couplet	Metadata	TokenData	Token	Parent	PLevel
24	#05		#16	22	3

4) This Couplet has a PLevel value of 3, so it is not used for convergence, rather its Parent value is used to locate its parent Couplet. It points to Couplet 22 (which has a PLevel of 2 or lower).

- 5) Couplet 23 also has a PLevel of 3, so its Parent value is used to locate its parent – Couplet 22.
- 6) Couplet 22 already has a PLevel value of two.
- 7) Items 2-5 have all converged at Couplet 22; this is used to fetch the XML fragment, provided the remaining selection criteria are matched.
- 8) The remaining selection item (#1) is an attribute with a tag level of 2, so its parent will have a tag level of 1. Because Couplet 1 has a PLevel of 1, it contains its own parent Tag, otherwise its Parent value would be used to fetch its parent Couplet.
- 9) The current convergence Couplet (#22) must ripple down Parent values until it reaches the first PLevel 1. Every selection item has converged so a match has been made.
- 10) Each matched Couplet at a level lower than Couplet 22's level is fetched (in this example, only Couplet 1 is at a lower level). Couplet 22 and all its children are fetched. The result is:

Couplet	Metadata	TokenData	Token	Parent	PLevel
1	#01		#01	1	1
22	#02		#14	1	2
23	#12		#15	22	3
24	#05		#16	22	3
25	#06		#17	24	4
26	#13		#18	24	4
27	#07		#07	24	4
28	#08		#08	24	4
29	#09		#19	24	4
30	#10		#10	22	3
31	#11		#20	30	4

- 11) The above is translated to:

Metadata/Data	ELevel
<u>Phonebook&gt;@country&gt;USA</u>	2
<u>Phonebook&gt;Listing&gt;@category&gt;Business</u>	3
<u>Phonebook&gt;Listing&gt;Name&gt;Xpiori</u>	3
<u>Phonebook&gt;Listing&gt;Address&gt;Number&gt;2864</u>	4
<u>Phonebook&gt;Listing&gt;Address&gt;Street&gt;South Circle Drive</u>	4
<u>Phonebook&gt;Listing&gt;Address&gt;Suite&gt;1200</u>	4
<u>Phonebook&gt;Listing&gt;Address&gt;City&gt;Colorado Springs</u>	4
<u>Phonebook&gt;Listing&gt;Address&gt;State&gt;CO</u>	4
<u>Phonebook&gt;Listing&gt;Address&gt;Zip&gt;80906</u>	3
<u>Phonebook&gt;Listing&gt;Telephone&gt;Areacode&gt;719</u>	4
<u>Phonebook&gt;Listing&gt;Telephone&gt;Number&gt;576-9780</u>	1

- 12) The reconstructed XML fragment becomes:

```
<Phonebook country=USA>
  <Listing category=Business type=Technology>
```

```

        <Name> Xpiori </Name>
        <Address>
            <Number> 2864 </Number>
            <Street> South Circle Drive </Street>
            <Suite> 1200 </Suite>
            <City> Colorado Springs </City>
            <State> CO </State>
            <Zip> 80906 </Zip>
        </Address>
        <Telephone>
            <Areacode> 719 </Areacode>
            <Number> 576-9780 </Number>
        </Telephone>
    </Listing>
</Phonebook>

```

13) Filtering of the XML document can be accomplished by a query post-processor.

The example above shows the default behavior for servicing queries. The convergence levels for selection items can be specified explicitly. This is accomplished by expressing a formula for the selection items. For example, to explicitly describe the scheme above, one might use:

```

[Phonebook>country>USA]
& (([Phonebook>Listing>@category>Business]
& [Phonebook>Listing>Name>Xpiori]
& [Phonebook>Listing>Address>City>Colorado Springs]
& [Phonebook>Listing>Address>State>CO])
@ [Phonebook>Listing])
@ [Phonebook]

```

## Scalability

The XML data repository server has been designed in a pervasively modular manner. This means that there is almost no limit to how big the servers can be. Additionally, data relationships can be built that span multiple documents on multiple computers, supporting true “many-to-many” relationships.

## Closing Comments

The example shown above shows queries being serviced by returning well-formed XML. What is actually returned is the appropriate list of Couplets, and that is parsed back into XML. Couplets can be used in their native form (in custom applications), or only the desired data element can be extracted (for a DBMS). Basically, Couplets amount to a fundamental representation of information that can be converted to accommodate any number of information representation paradigms.

In this paper, issues relating to the management of XML documents have been discussed on the core level – the XML data repository “engine”. Peripheral issues are discussed in other papers. The query protocol, for example, was disclosed as a Boolean equation. In the XML server, as deployed, the queries are actually expressed in XPath; a conversion mechanism is used to convert queries to the internal format. Other issues include using HTTP ports for communication and deployment on distributed computer systems.

## **Appendix: Related Papers**

Brandin, Chris, "A Definition of Digital pattern Processing™"

Brandin, Chris, "DPP™ Memory Management"

Brandin, Chris, "Three-Dimensional Content Scanning Using DPP™"

Brandin, Chris, "Optimized Coding Methods for Icon Generation and Manipulation In DPP™"

Brandin, Chris, "Management of Duplicate Data Elements in DPP™ Virtual Associative Memories"

Brandin, Chris, "Behavioral Set and Field Descriptor Implementations in DPP™", Xpiori

Direen, Harry and Phillips, Keith, "Finite Fields and Properties of the Xpiori Icon Generator, Associative Processing Unit, and Associative Memory Controller used in Digital Pattern Processing™"

Direen, Harry, "Duplicate Tree Structures in DPP™ Virtual Associative Memories"

**# # #**