

# The Application of Digital Pattern Processing (DPP™) to Storage and Retrieval of XML Data

by Chris Brandin

Release 1.1

**NOTE: In October 2003, Xpiori, 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 Xpiori, LLC.**

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

**Digital pattern processing (DPP)** is a term coined to describe a technology that is well-suited to the high speed indexing, searching, identification and manipulation of large volumes of data based on arbitrary criteria. It was developed and patented by Xpiori, LLC, based in Colorado Springs, Colorado.

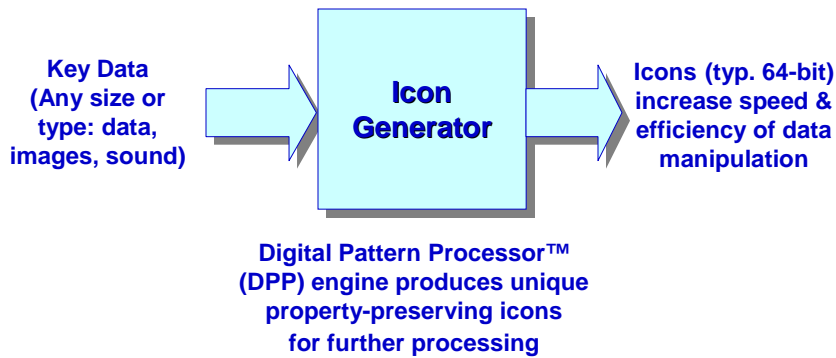
Potential applications of this technology are extremely varied. Content scanning, security, data mining and computer vision are only a few of the application areas that can benefit from DPP. In general, DPP offers a new approach to solving computational problems that are typically beset by scaling difficulties, making it possible to implement applications that may hitherto have been considered impractical.

The first application-level product developed by Xpiori, based on DPP technology, is the NeoXML Server. This presentation will describe the salient features of DPP and its application to XML storage and retrieval. Some details of the implementation are omitted or simplified for ease of understanding; the basics of the design are adequately captured.

## What is Digital Pattern Processing?

Digital Pattern Processing (DPP) is rooted in the notion that information may be processed symbolically, with a representation of a data item standing in for the data item itself. If this representation is chosen appropriately, data identification and even manipulation can be accomplished more efficiently using the data item's symbolic representation rather than actual data.

In DPP, the symbolic representation of a data item is called its **icon**. Icons live in icon stores. Within a given icon store, all icons are of a fixed length, typically 32 or 64 bits, although they can be larger or smaller depending on the nature of the data represented. Thus, arbitrarily formatted data items of arbitrary length are each represented by a single, fixed-length icon, produced by an **icon generator**. The icon generator is the logical construct that accepts variable-length key data in, and generates a fixed field icon for that data.



**Figure 1: Icon Generator**

### ***Properties of Icons***

Icons have some properties that are of particular interest:

- They are fractal in nature
- They have chaotic distribution properties
- They can be manipulated in a form of *icon algebra* that is consistent with the original data.

### ***Icon-based Libraries***

NeoCAM- to be completed  
 NeoData- to be completed  
 NeoSlider- to be completed  
 NeoFilter- to be completed

## Overview of the NeoXML Server

XML storage issues are fairly well-known. Self-description provides the power of XML, but it is typically expensive in terms of both memory and processing time. RDBMS-based techniques are cumbersome and generally inadequate to the task of representing extensible data, and most implementations rely on extensive string manipulation to support the kinds of capabilities database users have come to expect.

Xpiori has adapted and extended its technology to develop a server that stores, retrieves and modifies XML data. It also allows new XML fragments to be inserted into existing documents. The approach taken helps to mitigate many of the issues generally associated with XML repositories.

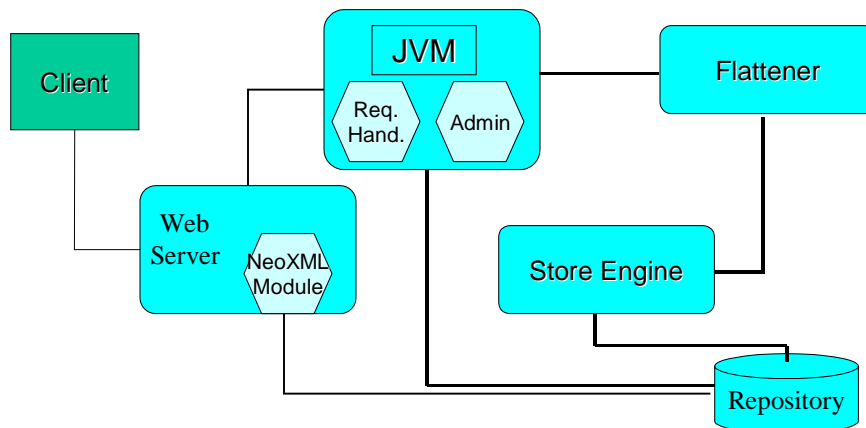
The premise adopted by the NeoXML server is based on the fact that computers handle integers extremely efficiently, while string handling is relatively painful. The cornerstone of the approach is to generate symbolic representations of the XML data, in the form of icons plus other numeric data, and to manipulate these symbols rather than the text itself as much as possible. There are specific query types and circumstances that do require text manipulation, but these are handled on an exception basis, and only after candidate XML elements have already been extracted from the database. The resulting implementation is inherently higher in performance than conventional DOMs, using less memory and operating orders of magnitude faster than DOM accesses.

In contrast to an RDBMS approach to XML data storage, the NeoXML server is schema-independent. The dynamic nature of XML is preserved- no prior definition is required before a user can store and access arbitrary XML data, and retrieve it in arbitrarily-sized chunks, from single elements to complete documents to multiple documents.

Since the server does not simply index XML, but actually stores it in an internal format, cross document queries are easily supported. For example, multiple purchase order documents can be accessed simultaneously to retrieve elements across them all. Modifications, insertions and deletions are also supported across documents, so that a single request can effect changes on multiple documents of the same or different schemas.

Another feature supported by the server is aliases. Aliases allow different XML tags with the same semantics to be treated together. For example, the XML from one auto parts manufacturer may tag an item with the name "AutomotiveLevitationDevice", while another might tag the same type of item with the name "VehicalElevationDevice". Internally, your business may prefer to refer to it simply as "Jack". Both the manufacturers' tags can be aliased to "Jack", enabling you to effectively standardize the XML tag nomenclature and use a parlance more congenial to your business to query and update the documents.

## Architecture of the NeoXML Server



**Figure 2: Process Architecture of the NeoXML Server**

The current server implementation is shown in Figure 1. Requests to access or create XML data come in from a client over an http interface to an Apache web server. Requests can be one of the following:

- Create a new XML document in the store;
- Insert a well-formed XML fragment into an existing document(s);
- Modify the textual contents of an existing attribute or text element(s);
- Delete an existing XML fragment(s) or document(s);
- Copy an existing XML document;
- Query an XML document(s) or fragment(s).

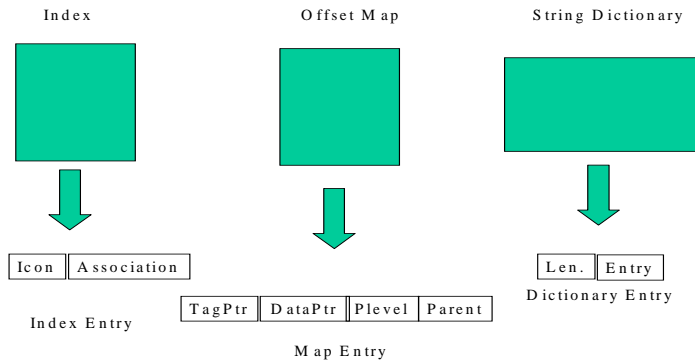
All queries are processed through a web server module, while all other operations, including administrative requests, are passed into a JVM implementing a servlet architecture. There are two flavors of servlet: a generic request handler servlet, and an administrative servlet. The use of a module that plugs directly into the web server optimizes query performance. As more experience is gained with the use of modules, the module implementation may entirely replace the servlet engine.

The current version of the server runs on both Windows 2000 and Solaris 2.7 platforms. All accesses to the repository are performed via a DLL (Wintel platforms) or shared library (Unix). Thus, specialized applications requiring ultra-high speed access to XML data and running on the same box as the server can link in the libraries directly. A JNI interface to the library functionality is also supported.

## Storage Architecture of the Server

In order to achieve the high query performance that characterizes the NeoXML server, XML data is stored in an optimized internal format made up of three main components:

- A NeoData-based index;
- An offset map;
- A string dictionary.



**Figure 3 Primary Storage Structures**

Figure 2 shows the primary NeoXML server storage structures and a somewhat simplified summary of their contents. The index is composed of index entries, the offset map of map entries, etc.

The index is a NeoData index and as such is composed of fixed-length entries. The association portion of the index entry contains an offset value to an entry in the offset map. Offset map entries, also fixed in length, in turn point to the tag and data strings associated with a given “flattened” line of XML. Plevel is a special number that characterizes the structure of the data, while Parent points within the offset map to the parent of the current line in flattened space.

In order to fully understand the storage architecture and the role of each of these structures in the server, the following examples will illustrate what happens when a document is processed for storage, and what happens when a document is queried. The roles of the different structures will then, it is hoped, be apparent.

### **Example: Storing A Document**

Consider the following small XML document<sup>1</sup>:

```
<Example>
  <Header>Sample Document</Header>
  <EmptyTag><EmptyTag>
    <Item>First Item</Item>
    <Item number="2">Second Item</Item>
</Example>
```

<sup>1</sup> Note that the author originally had a much more interesting and occasionally scatological document as a sample, but decided that discretion was, indeed, the better part of valor.

The first step in processing this document is to generate its “flattened” representation. Appropriately enough, this is done in the flattener process. Our sample document flattens as follows:

Line	Offset		Plevel	Parent
0	0	Example>Header> Sample Document	1	0
0	1	Example>EmptyTag>	2	0
0	2	Example>Item> First Item		2
0	3	Example>Item>@number> 2		2
3	4	Example>Item>Second Item		3

Note that a line is generated only if a)it has data values associated with it or b)it is an empty tag.

Two numbers are also generated when the file is flattened. The first, *plevel*, is the depth of the shallowest tag that appears for the first time in the line. In Line 0, Example is the shallowest tag that appears for the first time. The plevel of the line is thus 1. In Line 3, the plevel is 2, corresponding to Item. In line 4, since all the tags have already appeared once, the plevel is equal to the tag depth plus 1. In this case, the plevel of Line 4 is three.

The second number is the *parent* of the line in flat space. A given element in an XML document is represented by one or more flattened lines. A line’s parent in flat space is either the number of the first line of the element of which the subject line is a part, or, if the line is itself the first line of an element, its parent will be the first line of its containing element. In the sample document:

- Line 1’s parent is Line 0;
- Line 2’s parent is Line 0;
- Line 3’s parent is Line 0;
- Line 4’s parent is Line 3 (Line 3 is the first line of Item).

Both parent and plevel are critical in navigating XML documents in response to queries and in calculating set convergences.

After the line has been flattened and both parent and plevel generated, the document is then stored in the offset map and string dictionary. This portion of the processing is performed by the StoreEngine process.

The offset map contains exactly one fixed-length entry for each flattened line. It maintains the lines in document sequence order, so that the entire document may be reconstructed by accessing and interpreting sequential entries in the offset map<sup>2</sup>. The string dictionary, by contrast, contains variable length entries. Each dictionary entry stores either a tag string or a data string.

When the store engine receives a flattened line, it identifies the tag and the data portions of it. It then checks the dictionary, via a small NeoData index not shown in Figure 2, to see if either of the strings has already been entered in the dictionary. If it has, the process picks up a byte offset pointer into the dictionary to the entry from the index; if it has not, it adds the string to the dictionary and updates the index to reflect the update. The offset map stores the dictionary offset to the tag portion of the flattened line (TagPtr) and the dictionary offset to the data portion (DataPtr). Depending on the nature of the specific data, the use of a string dictionary can result in greatly compressed storage of the the XML document. The offset map also stores

<sup>2</sup> Note that insertion of fragments within a document is handled as a special case.

the plevel and, based on its current position and the value of the parent field, an offset to the map entry that contains the parent of the current flattened line. Thus, two of the entry's offsets point to a location in the string dictionary (TagPtr & DataPtr), while one (Parent) points to a logically previous entry in the offset map itself.

Returning to our "small document", the stored representation would look something like the following (note this representation cheats a little, using entry offsets rather than byte offsets for ease of understanding):

Dictionary:		Offset Map:					
Offset	String	Offset	TagPtr	DataPtr	Plevel	Parent	
0	Example>Header>	0	0	1	1	0	
1	Sample Document		1	2	Null	2	0
2	Example>EmptyTag>		2	3	4	2	0
3	Example>Item>		3	5	6	2	0
4	First Item		4	3	7	3	0
5	Example>Item>@number						
6	2						
7	Second Item						

### *Generating the Index*

The final step in the generation of the stored representation of the document is to generate the index entries. Index creation is deferred to the end of the store process to allow easier recovery from an aborted document. The index used in the XML server is essentially a NeoData index. The current server implementation supports two indexing modes: a minimal index, and a maximal index. Custom indexing will be supported in a future release.

Basically, each index entry consists of an icon representing all or a portion of a flattened line, and an association which contains the offset into the offset map of the offset entry that corresponds to that line.

In the current implementation, all queries start at the index. If no index entry can be found to match a query string, the query will fail. Minimal indexing is designed to support *fully qualified queries*, that is, those queries in which the full structure of a document is known a priori. Maximal indexing is designed to support *partially qualified queries*, also known as *descendent axis queries* in XPath parlance.

### **Querying the NeoXML Server**

At its most basic, querying involves looking the key up in the index, accessing the offset stored in the association, and reconstructing the XML from the offset map and dictionary. Simple as that may sound, implementing complex queries and ensuring that all cases are covered is not a task for the faint of heart.

### ***XPath in a Nutshell...***

The query interface to the NeoXML server is a subset of the W3C XPath Query Language Recommendation.

Briefly summarized, an XPath query consists of two parts: a target, and an optional predicate. Each XPath expression identifies a node or set of nodes in an XML document. Slashes are used to separate items in a path, much like a file system path. `"/Example"` would select the entire document in our simple example. The XPath expression `"/Example/Item"` would select both Item

nodes in the document. Modification with a predicate can choose one or the other: `"/Example/Item[.="First Item"]` would select the Item child of Example whose text value equals "First Item". Note that square brackets are used to enclose predicates. Attributes are denoted by the @ sign: `"/Example/Item[@number="2"]` would select the second Item in the document.

The examples of XPath given so far use what is called the "child axis". Each node in the path is the child of its predecessor. This is indicated by the use of the "/" between node names. Another axis is the descendent axis, denoted by "//". For example, `//Item` would designate all nodes of type "Item", regardless of their parentage. The use of descendent queries relieves the questioner from knowledge of the complete structure of the XML document.

### ***Indexing to Support Queries***

In order to support XPath queries at different levels of a document, each level must be indexed. In minimal indexing, the following policy is used:

Starting from the plevel tag of the flattened line, each left-to-right subtag is indexed. The full tag and the full line, including data, are also indexed. For our flattened sample, then, the first flattened line would generate index entries for `Example>`, `Example>Header>` and `Example>Header> Sample Document`. The next line would generate an entry for `Example>EmptyTag>`. The next line would generate entries for `Example>Item>`, and `Example>Item>First Item`. The following are all the unique index entries generated in a minimal index for our small sample document. Starred items indicate that an index stores multiple associations<sup>3</sup>:

```
Example>
Example>Header>
Example>Header>Sample Document
*Example>Item>
Example>Item> First Item
Example>Item>@number>
Example>Item>@number>2
Example>Item> Second Item
```

Note that the mapping between index entries and offset map entries is complex in terms of its multiplicity. Specifically, many index entries may point to a single offset map entry, and, conversely, a single index entry may point to many offset map entries.

### ***A Simple Query***

Let's now consider what happens when a user enters the following XPath query: `/Example/Item`. The intent of this query is to return the XML fragments corresponding to all Item entries in our sample document.

First, the XPath is parsed onto an execution stack. In the case of this query, the stack depth is one, so it is pretty simple. The index is accessed and the associations attached to `"Example>Query>"` are returned. In this case, the return from the index will be the set {2,3}, based on the offset map in the previous section. The offset map is then navigated sequentially from each of these points until the plevel encountered is less than or equal to the plevel of the initial entries. This marks the end of the element being retrieved. The XML is then reconstructed

---

<sup>3</sup> The intricacies and options involved in storing duplicates will not be addressed here. Duplicate handling forms the subject of a paper all on its own.

for each fragment and returned wrapped in <Query> ...</Query> tags to complete a well-formed XML document, which is returned to the user.

Note that no strings are accessed in this process until the appropriate elements have been accessed through the offset map. The actual strings that compose the XML document are not touched until the fragments are reconstructed. From the perspective of symbolic processing, the offset map is thus a symbolic representation of the actual document, and is processed in lieu of the actual document. Thus, the offset map extends the Xpiori concept of symbolic processing into the XML domain, resulting in tremendous processing efficiencies.

### **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™"

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"

# # #