Xindice 1.1 Internals Guide

$Revision: 511427 $

by James Bates, Kevin O'Neill

NOTICE:

This document describes the internal organization and operation of the Xindice native XML database engine. It is important reading for those who intend to contribute to Xindice's core engine.

Table of contents

Warning:

This documentation is a work in progress and is only applicable to the CVS version of Xindice. Its content is based mainly on close inspection of existing source code and some reverse engeneering. Some of the information may be inaccurate, or at least misleading with respect to the intent of the original core developers (which isn't always obvious from the source code). You have been warned.

1. 1. Overall Xindice architecture

Xindice is a native XML database engine that is written entirely in *Java*. As such it must always be hosted by a *Java Virtual Machine* (JVM).

When running, a Xindice instance stores a number of data items in Java objects inside the JVM, the most important of which are:

- Object representation of Collection hierarchy
- Client connection sate information
- Various cached data items

In addition, Xindice needs access to disk files containing the XML data, and related meta-data. The files are stored inside a diskfile directory hierarchy, that starts somewhere called the *database root*.

1.1. 1.1. Access modes

Xindice can be set up to run in a JVM in two different ways, depending on how clients will want to use Xindice.

In *embedded* mode, a complete Java application will set up a Xindice instance in *its own* JVM. Only that one Java application is able to access and manipulate the data in Xindice. Clients using the XML:DB API will use something called the *embedded driver* to access the Xindice instance that is running inside the same JVM as the host application.

In *server* mode, Xindice is run as a standard J2EE *web application*, in some *web application container*, such as Apache Tomcat. In this mode, the JVM hosting Xindice, is in fact the JVM running the web application container. Clients connect to Xindice from different JVM's possibly located on different machines, using *XML-RPC*, a *Remote Procedure Call* standard designed to work on top of *HTTP* (which is why Xindice is packaged as a *web application* in this mode).

2. 2. Organization of Collections

Logically, all XML data stored in Xindice is organized into a hierarchy of *collections*. A collection is exactly what its name suggests: it contains any number of XML documents, and can in addition contain its own child collections, thus providing a hierarchy.

The "root" collection is also called the *Database*. It is special in that:

1. It has no parent.
2. It can contain no XML documents of its own. It only has child collections.

Each collection in the database is represented in Java by an object of class `org.apache.xindice.core.Collection`. As with many Java objects inside Xindice, it is initialized using an *XML configuration description*, a piece of XML describing the properties of the collection. This XML configuration is modelled in Java as an object of class `org.apache.xindice.util.Configuration`. To set up the configuration of a collection, Xindice calls the collection's `setConfig()` method, passing it an appropriately obtained `org.apache.xindice.util.Configuration` object.

2.1. 2.1. The Database

The database, or "root" collection is the Java object that provides the link to everything else used by the Xindice instance. When Xindice first starts, its first act is to create and initialize an object of class `org.apache.xindice.core.Database`, which extends `org.apache.xindice.core.Collection`.

The database object is initialized using an XML configuration file that is obtained from outside the database (i.e. it is stored simply as a file somewhere). In the case of an embedded Xindice instance, the file is referenced using the Java property `xindice.configuration`, whereas in server mode, the file is referenced by the parameter `xindice-configuration` in the Xindice web application's `web.xml` file.

The format of the XML configuration file used to initialize the database object is as follows:

```
<xindice>
   <root-collection dbroot="./db/" name="db">
      <queryengine>
         <resolver autoindex="false"
class="org.apache.xindice.core.query.XPathQueryResolver" />
         <resolver
class="org.apache.xindice.core.xupdate.XUpdateQueryResolver" />
      </queryengine>
   </root-collection>
</xindice>
```

In fact, if during initialization, the XML configuration file cannot be found for some reason, rather that throw an error, Xindice will *assume* a default configuration file, which is exactly the one shown above.

The important elements in this configuration file are:

- the `dbroot` attribute of the `root-collection` element. It is a directory path that can be absolute or relative. If absolute, Xindice will use it right away, if relative, Xindice will try to find a system property called `xindice.db.home` and append to it the value of the "dbroot" attribute. If such system property is not set, Xindice will try to use the `/WEB-INF` webapp directory (under Tomcat it will be something like `[TOMCAT_HOME]/webapps/xindice/WEB-INF` as the parent (be aware that in case of a webapp upgrade it's possible to lose the whole database). If even that fails (e.g. because the WAR file was not unpacked), an Exception will be thrown and Xindice will fail to initialize with a proper log message suggesting a solution.
- the `name` attribute of the `root-collection` element. The root collection (or database) in Xindice is not called simply "/" as you may have expected. It actually has a name; this name is specified here.
- the various `resolver` elements in the `query-engine` element. These point to the *query engines*, and their Java implementations that this Xindice instance should support. Out of the box, Xindice supports two query styles, *XPath* and *XUpdate*. They are detailed in a subsequent chapter.

Notice that this external configuration file does *not* specify the *child collections* of the database. As we will see shortly, these are configured in XML somewhere *inside* the database, which we can now access (since we know where the database data is stored).

2.2. 2.2. The database root directory

As indicated, this directory contains all data and meta-data for the XML content of the database.

The directory structure inside this database root directory reflects the child collection structure of the database. So if there is a child collection named `mycol` in the database, the database root directory will contain a subdirectory named `mycol` and so on.

Each collection's directory contains at the minimum a file with extension `.tbl` that contains *all* the XML documents stored in that collection. The file is *not* human-readable. Its format is explained in subsequent chapters.

2.3. 2.3. The System Collection

One special collection, called `system`, always exists within a Xindice database. When the Xindice database is initialized, it automatically also loads the system collection, as this known to always exist. The structure of the system collection is simple: it contains no documents of its own, but contains two child collections: `SysConfig` and `SysSymbols`.

The `SysConfig` collection contains exactly one document called `database.xml`.

`SysSymbols` contains various documents that are in fact the *Symbol tables* used for storage of the element and attribute names of all XML content in the database. The symbol tables are detailed in a subsequent section.

The `database.xml` is the XML configuration file that is used to initialize all other collections in the database. It is located in the database itself, because it obviously needs to be updated each time collections are added or removed from the database.

Its structure is as shown below: (you can check your own configuration by issueing the command-line tool invocation: `xindice rd -c /db/system/SysConfig -n database.xml`)

```
<database name="db">
    <collections>
        <collection compressed="true" name="james">
            <filer class="org.apache.xindice.core.filer.BTreeFiler" />
            <indexes>
                <index class="org.apache.xindice.core.indexer.ValueIndexer"
name="myidx" pattern="sub" />
            </indexes>
            <collections>
                <collection compressed="true" name="sub">
                    <filer class="org.apache.xindice.core.filer.BTreeFiler"
/>
                    <indexes />
                </collection>
            </collections>
        </collection>
        <collection compressed="true" name="james_sub">
            <filer class="org.apache.xindice.core.filer.BTreeFiler" />
            <indexes />
        </collection>
    </collections>
</database>
```

As you can plainly see, it is here that the XML configuration for all remaining collections is stored. Please note that the `system` collection is not mentioned here. The XML configuration for the `system` is hard-coded into Xindice as follows:

```
<collection name="system">
    <!-- No filer for system collection: it contains no doucments itself
-->
    <collections>
        <collection name="SysSymbols" compressed="true">
            <filer class="org.apache.xindice.core.filer.BTreeFiler" />
            <symbols>
                <symbol name="symbols" id="0" />
                <symbol name="symbol" id="1" />
                <symbol name="name" id="2" />
                <symbol name="id" id="3" />
                <symbol name="nsuri" id="4" />
            </symbols>
        </collection>
        <collection name="SysConfig" compressed="false">
            <filer class="org.apache.xindice.core.filer.BTreeFiler" />
        </collection>"
    </collections>"
</collection>
```

The only way to modify this configuration is to change the Xindice source code (`org.apache.xindice.core.SystemCollection` class)and recompile.

2.4. 2.4. Other Collections

The XML Configuration data used to initialize collection objects in Java (of class `org.apache.xindice.core.Collection`) is, as shown in the examples above, located in an XML document in the system collection, or, in the case of the system collection itself, hard-coded into Xindice.

The important aspects of this configuration data are:

- A collection is represented by a `collection` configuration element. The `name` attribute indicates the collection's name. The `compressed` attribute, usually `true` indicates how XML data should be encoded by the collection's *filer*. More on this in a subsequent chapter.
- Each `collection` element contains a `filer` element. This element basically tells the collection the name of a Java class that it can use to read its own *data file*. (The file with a `.tbl` extension mentioned earlier). `org.apache.xindice.core.filer.BTreeFiler` is the most common, and indeed the standard filer class used in Xindice. If a collection has no filer (because the `filer` element is missing, or because the Java class it points to couldn't be loaded), then it won't be able to store any XML documents. That's the case for example of the database (root collection), and the system collection.
- A `collection` element *may* contain a `collections` element, which contains one `collection` element per child collection of the collection under discussion.

3. 3. Data storage

The XML data contained in the XML documents of a collection is stored in one single data file with extension `.tbl` that is located in the collection's directory somewhere in the database root directory. A special Java class called a *filer* is responsable for reading and writing XML data to such a data file.

In this chapter and the next, we will examine the most common filer in Xindice, implemented by the `org.apache.xindice.core.filer.BTreeFiler` class. The mechanism is somewhat complex, but it is broken down into a number of superimposed *layers*, each layer implementing some abstract data structure designied to improve overall performance.

In this chapter, we will concern ourselves not with XML storage directly, but the more abstract process of storage of (key,value) pairs. In essence, a Xindice file is no more that a performance-oriented format for storing (key,value) pairs. We will see later on, that this data format is used not only for the storage of XML data itself, but also for the storage of *indexes*.

The `org.apache.xindice.core.filer.BTreeFiler` class breaks up data storage into two layers. The bottom-most layer provides a *paged file* implementation. The code for this layer is located in `org.apache.core.filer.Paged`. On top of the paging layer sits a *B+-Tree* implementation: a balanced tree data structure (which allows storage of (key,value) pairs) specially optimized for disk access.

3.1. 3.1. Paged file

Paging provides efficient access to a random-access file by allowing parts of the file (*pages*) to be "mapped" to main memory for easy access. Pages have a fixed length. If data that must be stored is longer than the length of one page, subsequent pages in the file can be "linked" to the first.

As shown in the diagram, a paged file consists of a file header, followed by a list of fixed-length pages. The file header is 4kb long, and each page is, by default, 4kb long. (These values can be modified in the `org.apache.core.filer.Page` source code).

Each page contains a 64-byte header, followed by actual data. Pages are numbered. Whenever a particular page, say page `n` is needed, but not yet loaded into memory, the Java code can calculate the start address of the page as:

```
offset = fileHeaderSize + (n * pageSize)
```

At this address, it will then find the header of the wanted page, and 64 bytes further, the start of the page's data.

3.1.1. 3.1.1. Paged file header

The paged file header consists of a number of fixed-length fields. Fields which are longer than one byte, are *always* stored in Big Endian format, which means the most significant byte is written at the lowest address. This is regardless of the type of architecture the server process is running on, so your data files are portable between architectures.

The meaning of the various fields in the file header, whose structure is shown above, is as follows:

- header size (2 bytes): set to 4096 (0x1000), the size of this header.
- page size (4 bytes): set to 4096 (0x00001000), the page size.
- page count (8 bytes): this field is not used consistently. It is present mainly for historical reason.
- total page count (8 bytes): total number of pages present in this file.
- first free page (8 bytes): page number of the first unused page in this file. (see below)
- last free page (8 bytes): page number of the last unused page in this file. (see below)
- page header size (1 byte): size of each page header. Set to 64 (0x40) by default.
- max key size (2 bytes): see below.
- record count (8 bytes): number of *records* stored in this file. (see below)

Classes, such as the `org.apache.xindice.core.filer.BTree` class, which extend the paged class can add further fields of their own to this header.

3.1.2. 3.1.2. Pages and records

As indicated in the introduction, the idea of using a *paged* file is that data values can be stored that are longer that available page length. A stored data value is called a *record*. If the length of data in a record fits into the available space in a single page, nothing special happend. But if the record data doesn't fit into a single page, then the first page is filled with the start of the record, and subsequent pages are used to store the remainder. In this situation storage of a record uses several pages, which are linked together with appropriate fields in the header (see next section). Below is an example of a record stored in 4 different pages:

Unused pages which exist in the data file, but are free to be used for data, are all linked together into the *unused page list*, in the same manner (i.e. using the `next page` field in page headers) as pages used to store a record. The first page number in this unused page linked list is stored in the file header `first free page` field, and the last page in the `last free page` field.

3.1.3. 3.1.3. Page header

Each page has its own header, whose structure is shown above. The meaning of the various fields is as follows:

- status (1 byte): Pages in the data file are either *used* or *unused*. *Used* pages contain actual data.
  When a data value (called *record* in paging terminology) has to be written to the paged file that is longer than the page size will allow (i.e. longer than the page size - page header size), a new page is allocated for the part of the value that doesn't fit in the first page. The header of the first page's next page field is then set to the page number of this new page. If the data is still too long for both pages, a third one is allocated and pointed

to by the second, and so on.
When a page is unused, its status is set to `UNUSED`, encoded by the value 0x00. If a page is used to store the *first* part of a record, (or only part if the record is short enough), the status is set to a value that is used by the B-Tree algorithm. If, on the other hand, the page contains the *remainder* of some record started in some other page, its status is set to `OVERFLOW`, encoded by the value 0x7E.

- key length (2 bytes): pages have the possibility of storing a *key* just before their actual data. The maximum length of such a key is set in the file header's max key size field to 256 (0x0100). The *actual size* of the key located in this page, if any, is set in this field. Data for this page thus begins `64 + (key_length)` bytes beyond the start of the page.
  If no key is used in this page, this field is 0.
- key hash (4 bytes): As the name suggests, this field stores a 32-bit hash value calculated from the key. Used to optimize searches mainly.
- data len (4 bytes): The length of the data stored *in this page*. If some of the data of the record stored here continues into a subsequent page, this field contains the length of the data stored in *this page* only.
- record len (4 bytes): the total length of the data record of which part is stored in this page.
- next page (8 bytes): page number of the page that contains subsequent data for the record stored in this page, if more data is available.
  If this is the last page that stores data for the record, this field contains -1 (0xFFFFFFFF).

3.2. 3.2. The B-Tree storage format

3.2.1. 3.2.1. B-Trees

The B-Tree (as used by Xindice) is a data structure that allows for disk-efficient organization of keys. The B-Tree is such that keys can be rapidly retrieved from the tree, allong with associated data.

A *node* in a B-Tree consists of a set of values. *Between* each pair of values in a node is a *pointer* to a child node. The idea is that when searching for a key, starting from the root node, at each node the sought key is compared to the values in that node, and a path is chosen immediately to the right of the smallest value that is equal to or greater than the sought key. This process continues until a leaf node (which has no pointers to other nodes), in which case the value is present or not. If not present, the value doesn't exist in the tree.

Note that searches *always* terminate on a leaf node. This means that we can determine which keys are present in the B-Tree simply by looking at all the leaf nodes. This is not the case of all B-Trees, but Xindice B-Trees are always organized this way.

For a given tree, there is a maximum and minimum number of values that should be present in each non-leaf node. The desired number of values is trade-off between the number of steps required by the search algorithm, and the size of each node (which is read entirely into memory at the appropriate step). In the example shown above, the maxium chosen was 4, and the minimum 2.

When inserting and deleting values from the tree, some complicated operations may be necessary, such as *splitting* and *collapsing* non-leaf nodes. The manner in which these are done greatly influences the performance of updates to B-Tree files.

Any good database book should provide a much more detailed explanation of the theory behind B-Trees than this short introduction paragraph.

3.2.2. 3.2.2. Storing B-Trees in a paged file

Xindice implements B-Trees using the paged file, described in a previous section. Each node in the B-Tree is stored in a record, spanning one or more pages.

In addition to the nodes of the B-Tree, Xindice's B-Tree filer also stores *data* that is associated with some key. This data is stored in separate records (also spanning one or more pages). Leaf nodes in the B-Tree (recall that the set of all leaf-nodes in the tree contains *all* keys used) contain pointers to the records that contain the data associated with the keys (in the leaf nodes).

The B-Tree filer (implemented by both `org.apache.xindice.core.filer.BTree` and `org.apache.xindice.core.filer.BTreeFiler`) thus deals with three kinds of records in the paged file that it manages:

- non-leaf nodes, called *branch* nodes of the B-Tree. Such nodes contain a list of key values interwoven with pointers to other B-Tree nodes.
- leaf nodes of the B-Tree. Such nodes contain a list of key values, and associated with each a pointer to a data record, which contains the data associated with the key.
- data records. These records are not part of the B-Tree, but the leaf nodes in the tree *point* to them.

The B-Tree implementation adds some extra fields to both the paged file header and page headers for its internal organization. The details are as follows:

3.2.2.1. 3.2.2.1. B-Tree filer paged file header

The two extra fields (in Big Endian byte order, as usual) are:

- `root node page number` (8 bytes): The page number of the page where the record that holds the root node of the B-Tree starts.

- `total bytes` (8 bytes): the size, in bytes of the file holding the B-Tree node- and data-records.

3.2.2.2. 3.2.2.2. B-Tree filer page header

The extra fields (in Big Endian byte order, as usual) are:

- `status` (1 byte): Some extra values are possible here, above those used in the standard page file. The field is used in the first page of a record to indicate which of the three kinds of record it is:
  - a record used to store a branch node of a B-Tree will have status `0x01`.
  - a record used to store a leaf node of a B-Tree will have status `0x02`.
  - a record used to store data will have status `0x14`.
- `val_count` (2 bytes): If the record is used to store a node of the B-Tree, then this field indicates how many (key) values there are in the node.
- `created` (8 bytes): If the record is used to store data, then the time (in UNIX long format) when the data record was created is stored here.
- `modified` (8 bytes): If the record is used to store data, then the time (in UNIX long format) when the data record was last modified is stored here.

3.2.3. 3.2.3. B-Tree node storage

The last remaining question is: how is a B-Tree node stored in its record in the paged file? Recall that a B-Tree node contains a certain number of key values, and an associated set of pointers (in fact page numbers of the start of related records).

In the case of branch nodes, there is one more pointer than there are values; the pointers point to other nodes in the B-Tree. In the case of leaf nodes, there are exactly as many pointers as key values, and each pointer points to the data record associated with one of the key values.

In either case, the node is encoded as follows:

- For each key value, two fields are written. Keys in the Xindice implementation are not numbers, as in the image of a B-Tree above, but rather strings of character data. For each of these strings, Xindice encodes the string in utf-8, resulting in a byte array, and writes:
  - The length of the resulting utf-8 byte array. This length is written over two bytes.
  - The bytes representing the key value (a string), in utf-8.

  The number of values present is known from the page header.
- Immediately following the key data come the pointers. These are written simply one after the other, each occupying 8 bytes. (Each pointer is the page number of the first page holding the record being pointed to). The number of pointers is known from the number of key values, and whether the node is a leaf- or branch node.

4. 4. XML storage

As we saw in the preceding chapter, the B-Tree file format allows for the efficient storage of (key, value) pairs. In this chapter we concern ourselves with using such a (key, value) storage facility to store the XML content of all XML documents in a collection.

The principle Xindice uses is deceptively simple here: for every XML *document*, Xindice will calculate something called the *compressed DOM*. This is an array of bytes which can be used to reconstruct the complete XML document at any time. An XML document is then stored as a (key, value) pair in the B-Tree, where the key is the name given to the XML document, and the value is the calculated Compressed DOM.

The remaining mechanism to investigate is thus how to construct the Compressed DOM of a document.

4.1. 4.1. The symbol tables

In order to store the XML content in a space-efficient manner, Xindice uses something called a *Symbol table*. This is an XML file which associates a 16-bit number with any (QName,namespace URI) pair used as element or attribute name in XML *all* XML files stored in a collection. (i.e. there is *one* symbol table per collection).

Consider the following XML document, to be added to a Xindice collection:

```
<?xml version="1.0"?>
<p:person xmlns:p="http://www.xindice.org/Examples/PersonData"
          gender="female"
          xml:lang="fr">
    <p:first-name>Susanne</p:first-name>
    <p:last-name>Carpentier</p:last-name>
    <p:e-mail active="yes">scarpentier23@freenet.fr</p:e-mail>
</p:person>
```

When this document is stored into an empty Xindice collection, the following symbol table is created:

```
<?xml version="1.0"?>
<?xindice-class org.apache.xindice.xml.SymbolTable?>
<symbols>
    <symbol name="p:first-name"
nsuri="http://www.xindice.org/Examples/PersonData" id="4" />
    <symbol name="p:e-mail"
nsuri="http://www.xindice.org/Examples/PersonData" id="6" />
    <symbol name="p:last-name"
nsuri="http://www.xindice.org/Examples/PersonData" id="5" />
    <symbol name="gender" id="2" />
    <symbol name="xml:lang" id="3" />
    <symbol name="p:person"
nsuri="http://www.xindice.org/Examples/PersonData" id="0" />
    <symbol name="active" id="7" />
    <symbol name="xmlns:p" nsuri="http://www.w3.org/2000/xmlns/" id="1" />
</symbols>
```

As you can see, the symbol table is itself an XML document which contains an element for every (QName, namespace URI) pair used in element and attribute names in the XML documents of the collection. The `id` attribute is the 16-bit number that Xindice has assigned to the (QName, namespace URI) pair.

As more documents are added to the collection using different element and attribute names, entries are added to the collection's symbol table.

Usually, a collections's symbol table is stored as any other XML document in the Xindice database. All symbol tables stored in Xindice are in the `system/SysSymbols` collection using as name the path of the collection, with underscores (_) subsituted for the /'s in the collection path.

Being a collection in Xindice, `system/SysSymbols` itself has a symbol table too. It is:

```
<symbols>
    <symbol name="symbols" id="0" />
    <symbol name="symbol" id="1" />
    <symbol name="name" id="2" />
    <symbol name="id" id="3" />
    <symbol name="nsuri" id="4" />
</symbols>
```

Normally, this symbol table should be stored in an XML document named `system_SysSymbols` in the `system/SysSymbols` collection. Doing so however would create an endless loop, as `system/SysSymbols`'s symbol table is needed to read itself! This particular symbol table is therefore hardcoded (`org.apache.xindice.core.SystemCollection` class) into the Xindice source code.

For any other collection, you can always request the symbol table yourself by issuing the Xindice command-line invocation:

```
xindice rd -c /db/system/SysSymbols -n [your_collection_path]
```

4.2. 4.2. The Compressed DOM

Now that we understand symbol tables, we can take a look at the way in which Xindice generated a byte string from any given XML document.

The trick is to understand that Xindice simply runs through the XML document recursively, building a byte sequence for a particular node in the tree representation of the XML. This will contain the byte data for the children of the node, and these sub-sequences contain the data for their children etc...

Xindice thus starts by generating the byte sequence for the document node, which will set off generation for the whole XML document. The code that handles this is located in the `org.apache.xindice.xml.dom.DOMCompressor` class.

4.2.1. 4.2.1. Document node

The document node, which can at most contain some processing intructions, comments, and exactly one element, is encoded as follows:

The 4-byte `record_length` field, which is always encoded in Big Endian order, i.e. the most significant byte at the lowest address, indicates the total length, in bytes, of the encoded document byte array, *excluding* the 4 bytes used by `record_length`.

The child data is formed by concatenating the byte arrays generated by the processing instruction, comment and element nodes inside the document, in the order they appear in the document.

4.2.2. 4.2.2. Element nodes

An element node is encoded as shown in the diagram below:

All multibyte fields are always encoded in Big Endian order, i.e. the most significant byte is at the lowest address. The meaning of the various fields is as follows:

- The signature (1 byte) is composed of several 1- or 2-bit fields whose meaning is:
  - `Attribute Count Type` (bits 0-1). This is a code that indicates the length of the `attribute_count` field. The code works as follows:
    - value `00` (binary): the `attribute_count` field is zero, and thus *absent* from the byte array
    - value `01` (binary): the `attribute_count` field is 4 bytes (32 bits) long.
    - value `10` (binary): the `attribute_count` field is 2 bytes (16 bits) long.
    - value `11` (binary): the `attribute_count` field is 1 byte (8 bits) long.
  - `Record Length Type` (bits 2-3). This is a code that indicates the length of the `record_length` field. The code works exactly as the `Attribute Count Type` code above, except it cannot be `00`, as the `record_length` field is never zero.
  - `C` (bit 4). Tells whether the element has any non-attribute children (child elements, text or comments). `0` means it hasn't, `1` means it has.

    - `A` (bit 5). Tells whether the element has any attributes. `0` means it hasn't, `1` means it has.
    - signature type (bits 6-7). Always set to `01` for elements. It tells Xindice that what follows is an element signature.

- `record_length` (`x`: number of bytes inidicated by `RLT` code in signature): the length, in bytes, of the byte string representing this signature. That is including the signature byte, this `record_length` field, the `symbol_id` field and any child & attribute data.
- `symbol_id` (2 bytes): This is the 16-bit identifier that is associated with the element *name* (QName and namespace URI) of this element in the containing collection's symbol table.
- `attribute_count` (`y`: number of bytes indicated by `ACT` code in signature): the number of attributes this element has. This allows Xindice to start reading the child & attribute data, knowing that the first `attribute_count` nodes will represent attributes.
- child & attribute data: Attribute data is written first; it is obtained by generating byte arrays for all attribute nodes in this element and concatenating them together. Immediately following the attribute data come the byte sequences obtained by compressing the comment, text and child element nodes of this element. They are processed in the order they appear in the XML document. Again, the byte sequences thus generated by child nodes are concenated together.

4.2.3. 4.2.3. Attribute nodes

An attribute node is encoded as follows:

Once again, both 2-byte fields are Big Endian.

- The `symbol_id` (2 bytes) is the identifier that is associated with the attribute *name* (QName and namespace URI) of this attribute in the containing collection's symbol table.
- The `length` (2 bytes) is the length, in bytes of the `data` field that follows. Note the length of `length` is fixed at 2 bytes, so an attribute's data can be at most 65536 bytes long.
- The `data` is the byte array obtained by encoding the character data that is the attribute value in UTF-8. Note that the number of bytes will in general be more than the number of characers encoded by them, as UTF-8 is a variable character-width encoding scheme.

4.2.4. 4.2.4. Text nodes

A text node is encoded as follows:

The meaning of the various fields is as follows:

- `signature` (1 byte): is composed of several bit fields as follows:
  - bits 0-1: unused, and always set to `00`
  - Record Length Type (bits 2-3): used to indicate the length of the `rec_length` field exactly as for elements.
  - bits 4-5: unused, and always set to `00`
  - signature type (bits 6-7): for text nodes, these are set to `00`. These allow Xindice to know that the encoded node is a text node.
- `record_length` (x: as indicated in `RTL`): length of the complete byte array for this text node, *including* the signature and this `record_length` field. As this field could be 4 bytes long, there is not the same restriction on text node length as there is on attribute length.
- `data`: This is the byte array obtained by encoding the character data of the text node in UTF-8.

4.2.5. 4.2.5. Comment nodes

A comment node is encoded as follows:

The meaning of the various fields is as follows:

- `signature` (1 byte): fixed at 0xE0 for comments
- `record_length` (4 bytes): length of the complete byte array for this comment node, *including* the signature and this `record_length` field.
- `data`: This is the byte array obtained by encoding the character data of comment node in UTF-8.

4.2.6. 4.2.6. Processing Instruction nodes

Contrary to intuition, procedding instruction nodes are not stored using the sytmbol table for the PI target. In stead a signle text value is constructed by concatenating the PI target, one single space (U+0020) and the PI data. This text is then used to encode the PI node as follows:

The meaning of the various fields is as follows:

- `signature` (1 byte): fixed at 0x80 for PI nodes
- `record_length` (4 bytes): length of the complete byte array for this pi node, *including* the signature and this `record_length` field.
- `data`: This is the byte array obtained by encoding the concatenated character data calculated earlier, in UTF-8.

5. 5. Queries

Queries in Xindice provide functionality for looking up and retrieving documents or parts of documents using a variety of query languages. Updates of the database itself are also possible using queries.

The query architecture is quite modular, and so will first inspect it at an abstract level before delving into the details of Xindice's two native query languages, XPath and XUpdate.

5.1. 5.1. The Query interfaces

When Xindice first initializes itself, it creates amongst other things a *Query engine*, implemented by `org.apache.xindice.core.query.QueryEngine`. The principal purpose of this query engine is to setup one or more *Query resolvers*. Query resolvers are Java classes that handle all querying functionality related to one particular query language, e.g. XPath or XUpdate. The source code refers to these as *query styles*.

A query resolver is represented by the Java interface `org.apache.xindice.core.query.QueryResolver`. Two implementations, one for XPath and one for XUpdate of this interface exist within Xindice, but you could add your own implementations to support other query languages. As usual, loading and initializing the appropriate query resolvers is done by parsing a piece of XML configuration data (The `query-engine` element of the database configuration file).

The `org.apache.xindice.core.query.QueryResolver` interface provides two important methods: `query()` and `compileQuery()`. Both model a complete invokation of a query *on a collection*. They accept the following parameters:

- `org.apache.xindice.core.Collection` **`context`**: This is the collection in which the query is to be invoked. All results will be fragments of XML from documents in this collection. Cross-collection queries are never possible in Xindice.
- `java.lang.String` **`query`**: This is the query expression to be invoked. The syntax of the expression depends on the query language being implemented by the resolver.
- `org.apache.xindice.xml.NamespaceMap` **`nsMap`**: The query expression will generally contain element names. Both XPath and XUpdate allow using element names in their syntax. Often these element names will be *qualified* by a namespace prefix, which needs to be *resolved* by the query resolver in order to find the exact namespace uri being represented. This namespace map allows a query resolver to do just that.
- `org.apache.xindice.core.data.Key[]` **`keys`**: This parameter, if not `null` provides a set of *doccument names* (called keys internally) in the collection that should be considered by the query invokation. If the parameter *is* `null`, then *all* documents in the collection are considered.

The difference between both methods is that `query()` invokes the query immediately and returns results, whereas `compileQuery()` encapsulates the invokation into a `org.apache.xindice.core.query.Query` object which can be used several times later on to invoke the query.

We now turn our attention to the two provided query resolvers XPath and XUpdate.

5.2. 5.2. XPath Queries

The `org.apache.core.query.XPathQueryResolver` class implements a query resolver for the XPath language. Recall that a query resolver provides two query methods: one for immediate execution and another for storing (after compiling) invokations for later use.

Internally, the xpath query resolver *always* compiles queries into a `org.apache.core.query.XPathQueryResolver.XPathQuery`. It then may or may not execute the query immediately depending on which method was called on the resolver.

Analyzing and compiling the query is actually handled by Xalan, not Xindice. Xalan contains XPath manipulation and evaluation classes, and Xindice uses these.

When an XPath query needs to be evaluated, a set of candidate documents is selected from the collection. Then the XPath is evaluated using the Xalan classes against each of these documents in turn: the document is loaded into a DOM tree using the compressed DOM and B-Tree filer classes, and the XPath is evaulated against this DOM tree. The results of all evaluations are aggregated and returned.

It follows from the above that there is absolutely no performance gain in using Xindice to evaluate an XPath with respect to a document. Using a parsed XML file, exactly the same performance would result. Xindice's main contribution is in searching through a large collection of documents, as in this case, it can use indexes to intelligently select a set of candidate documents, as we will now see.

5.2.1. 5.2.1. Selecting candidate documents from a collection

When performing a query, recall it is posible to specify which documents should be considered. If this is done, then Xindice will use the provided set as the candidate set, and execute the XPath query against *each* of them, reading them into memory first.

If however no explicit set of documents is speficied, Xindice will try to locate an appropriate index based on the XPath query. This index can then provide an intelligent set of candidate documents, and the XPath is evaluated against only these documents. The details of indexes are explained in the next chapter.

If no appropriate index is found, Xindice resorts to "brute force": it evaluates the XPath expression against *every* document in the collection, thus effectively reading in, parsing and searching each document.

5.3. 5.3. XUpdate Queries

XUpdate queries are used to send update instructions to Xindice. An XUpdate query is actually a complete XML document which can contain any number of update instructions. As it is an XML document of itself, the namespace bindings for prefixes used in the XUpdate instructions can be specified in the XUpdate string itself. They can also be specified as a `NamespaceMap` when calling a query method on the resolver. This may cause a conflict if the same prefix is bound both in the XUpdate, and in the provided `NamespaceMap`. By default, Xindice will take into account the value specified in the `NamespaceMap` in this case, though the behaviour can be altered by modifying the `org.apache.xindice.core.xupdate.XUpdateImpl.API_NS_PRECEDENCE` constant in the source code and recompiling.

The `org.apache.xindice.core.xupdate.XUpdateQueryResolver` class provides a query resolver for XUpdate queries. Just as the XPath resolver, it always first compiles queries to a `org.apache.xindice.core.xupdate.XUpdateQueryResolver.XUpdateQuery`, wich it then immediately executes if the resolver's `query()` method was called.

Xindice uses the Lexus engine, produced at infozone.org to parse and execute XUpdate instructions. The `org.apache.xindice.core.xupdate.XUpdateQueryResolver.XUpdateQuery` and `org.apache.xindice.core.xupdate.XUpdateImpl` classes basically provide some glue around Lexus.

Lexus itself doesn't contain an XPath evaluator, and so Xindice needs to provide one to use Lexus. We saw earlier that Xindice itself uses Xalan for its XPath needs. To provide a valid XPath implementation to Lexus, another class `org.apache.xindice.core.xupdate.XPathQueryImpl` exists which encapsulates Xalan's XPath classes in a manner suitable for Lexus. Don't confuse this class with `org.apache.core.query.XPathQueryResolver.XPathQuery`, which also wraps around Xalan, but for use by Xindice's own XPath query resolver.

5.3.1. 5.3.1. Selecting documents to update

When an XUpdate query is submitted with a specified set of documents, the all the XUpdate modification instructions are performed on each of the documents.

If however no documents are specified, Xindice proceeds as follows:

- For each of the modification instructions, it looks up the *selector expression*, that is an XPath expression that each XUpdate modification instruction contains to indicate *which* node in a document should be modified.
- Next, a complete XPath query is performed on the collection, using the `org.apache.xindice.core.query.XPathQueryResolver` to find all nodes matching the XPath in documents in the collection, and these nodes' containing documents are retained. Note that this step *might* make use of any indexes set up for XPath, as determined by the behaviour of the XPath query resoolver.
- The XUpdate modification instruction is performed on the retained set of documents.

5.3.2. 5.3.2. Performing the updates

Whether explicitly passed as an argument to the query method, or whether determined by evaluating the selector expression on the collection, the set of documents is always processed as follows:

- Each document in the set is read in to a DOM tree in memory using the DOM Compressor and B-Tree filer classes.
- Lexus updates this in-memory DOM representation according to the modification instructions.
- Xindice writes the *entire document* back using the same DOM Compressor and B-Tree filer classes. The modification time *of the whole document* is updated appropriately.

5.3.3. 5.3.3. Result

Xindice queries must always return results. XUpdate queries always return a result with exactly one XML resource that looks like this:

```
<?xml version="1.0"?>
<src:modified
xmlns:src="http://xml.apache.org/xindice/Query">4</src:modified>
```

It essentially indicates the number of updates that the query performed. To change the format of this XML, you'll have to modify the XUpdate query resolver's source code.

6. 6. Indexes

7. 7. The XML:DB drivers

7.1. 7.1. Embedded driver

7.2. 7.2. XML-RPC driver

7.3. 7.3. Managed driver

The Managed driver (`xindice-managed`) is designed for use in environments where the life cycle of the xindice database is controlled by an external service. Like the Embeded driver accesses a database running in the local vm. Unlike the Embeded driver the Manged driver will **not** create the database instance. It will fail to load if the database instance is not available

7.3.1. 7.3.1. Configuraton

Configuration of the driver is easy, create an instance of the class. There are two constructors provided, a "no args" default constructure one that takes the name of the database to connect to. The "no args" constructor will attempt to connect a database with the name of "db" or the value of the "xindice.drivers.managed.db" system property. When the second constructor is used the database name is explicitly passed.

7.3.2. 7.3.2. ManagedServer

ManagedServer is a simple class that loads a database for access by the Managed driver. It is by no means the only way to create and register a database instance. Configuration is the same as for the embeded driver. This is suitable for simple tests and standalone applications, but managed environments should provide their own configuration implementations.

7.3.3. 7.3.3. Advantages of the managed driver

The main advantage is that the database has an explicit lifecycle. With the Emdeded driver database creation is a side effect of access to the client driver. The determination of what lifecycle methods are run when is unclear (for example when is the database closed). By making database instantiation an external task to the driver this problem disappears.