Large XML Documents

— A Model for Managing Complexity

Abstract

XML offers several advantages over word processors for authoring and maintaining documents. There is one advantage, however, that has remained largely unexploited – the management of large documents.

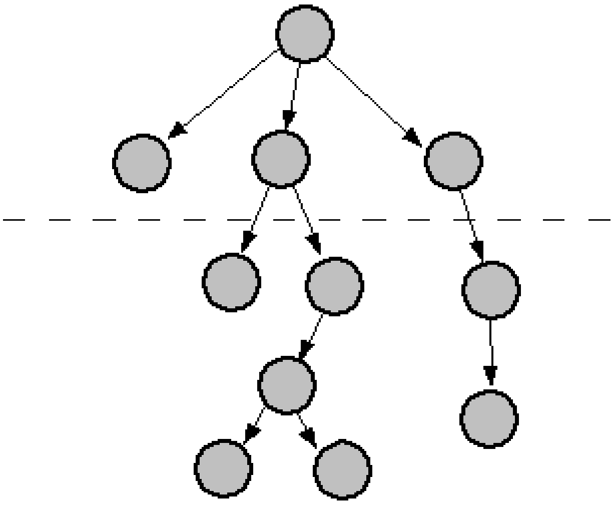
Most applications manage large XML documents by decomposing them into parts small enough to be conveniently edited. It is arguable that this practice is both inefficient and makes it difficult to realise the full value of XML data. This talk discusses some of the issues relating to decomposing XML and whether there is a processing model that would make this unnecessary. A demonstration using one such model will be shown.

Introduction

The advantages of structured (XML) over unstructured (word processor) documents have been thoroughly discussed, though perhaps not yet widely realised. A structured document is more valuable, flexible and durable because information can be reliably extracted by an automatic process. In other words, processing does not require understanding. This characteristic also offers advantages for handling large documents – a structured document can be easily decomposed into smaller parts and reconstituted without any risk of losing information. The same cannot be said, in general, of an unstructured document.

Techniques for Decomposing XML

There is an intuitive way of decomposing XML documents:



- represent the document as a tree, with the root at the top and the leaves at the bottom
- draw a horizontal line through the tree
- adjust the position of the line so that all of the sub-trees below it are small enough to be processed as a unit, but large enough to stand alone

The line in our diagram partitions the elements in the document into two sets. Above the line are the structural elements that provide the “skeleton” of the document. Below the line are the content elements that form the “meat”.

Once we have partitioned the document there are several ways to represent it. We could save it in a filesystem using the directory structure to mirror the document structure, or we could put it into a content management system (CMS) using a relational database (RDBMS) to organise the data. Both of these methods suffer from the disadvantage that we no longer have a “pure” XML document we can pass to an XML processor.

If we wish to work within the XML standards, we could use entities to store the content objects and entity references to link them, or we could use the XInclude standard to connect multiple independent documents. The latter is more flexible, but not part of the basic XML standard, and therefore not implemented by all XML processors.

The following table summarises the options:

Method	Advantages	Disadvantages
File System	No extra infrastructure required	Hard to ensure consistency
CMS/RDBMS	Security controls and search methods	Cannot process document using a standard XML processor
Entities	Part of basic XML standard	Requires an editor to parse the entire document
XInclude	Each document can be processed independently	Not widely implemented

In short, while decomposing an XML document may be easy, deciding how to represent it in its decomposed form may not be!

Disadvantages of Decomposing

While decomposing XML is easy and reliable, it does suffer from a number of disadvantages:

1. Deciding how to decompose documents is a skill distinct from the skills used in authoring content. Because it can be critical in allowing effective access to the information, resources must be allocated to make sure it is carried out in a timely and reliable manner.
2. Authors need to be aware of the decomposition method in order to locate and change information, which may involve additional skills and/or training.
3. The separation of elements into two classes may make certain processing more difficult. One of the chief advantages of the XML model is self-similarity – the fact that any part of a document can be represented by a tree structure, regardless of where it appears in the document. Using more than one method to store different parts of the document may compromise this. This is particularly true when an RDBMS is used.

Detour – Once Were Files

At one point in the history of computing a typical application was designed around a number of files, each of which had an access method determined by how it was physically stored (e.g. sequential, random or indexed). The advent of relational databases offered a new way to store data that replaced the concept of a file with the abstraction of a table. Although it is more efficient to manage files directly (at least for small applications), this is more than offset by the advantages databases offer:

1. Administration of files is simplified because it is centralised, and the same skills can be used to administer data for any application.
2. Applications do not need to know how data is physically stored. Code can be made much simpler because a high-level abstraction is used.
3. Because data is centralised and accessed in a common way, there are opportunities to use it in ways that are not part of the original application design (e.g. ad hoc queries).

These advantages are not necessarily things that a designer of a file-based system would consider to be problems. Nevertheless, they have proven to be sufficiently important to make the RDBMS a foundation of modern application development.

The disadvantages of the current methods for decomposing XML are in some ways a mirror of the advantages provided by a database. This is not to say that databases are a better way of storing XML. Rather, this demonstrates that using an abstraction to model a system can offer benefits that outweigh the concomitant loss of efficiency.

The question is: “what is a better model for large XML documents?”.

Traditional Models for Managing Complexity

There are many examples of large systems that have been organised in ways that make them easy to use:

- geographical locations use an addressing scheme that is organised into countries, regions, cities and streets. Each address on a street is allocated a number.
- libraries are organised by an hierarchical numbering system. Once the user finds the appropriate shelf they can scan they books allocated to a specific number ordered by the author’s name.
- finding an entry in a telephone book is usually done by applying an heuristic to determine the approximate region, then scanning forwards or backwards from that point.

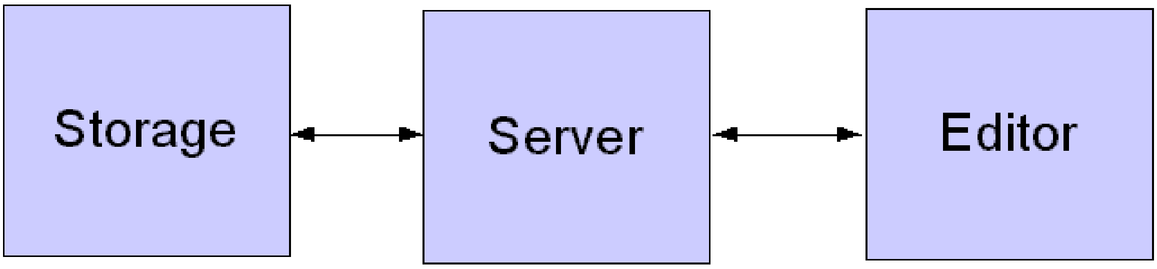
All of the above have one thing in common. There is a top-level hierarchy that partitions the data into manageable pieces. Each of these pieces is then processed in a simple (often linear) way.

There are obviously similarities with the scheme described above for decomposing XML data, but there is also a crucial difference. The traditional schemes allow for some freedom in determining when processing changes from hierarchical to linear. This is important in order to accommodate changes in the volume and distribution of the data, as well as variations in the skill and experience of the users.

The next section proposes a new model for decomposing XML that includes this flexibility.

A New Model for Managing XML

The new model is described by way of a demonstration system, the components of which are shown in this diagram:



The XML data is stored in a repository that preserves its tree structure (a persistent DOM). The server responds to the requests of the editor to supply and store parts of the document. The editor supplies the user interface to allow the user to read and alter the data.

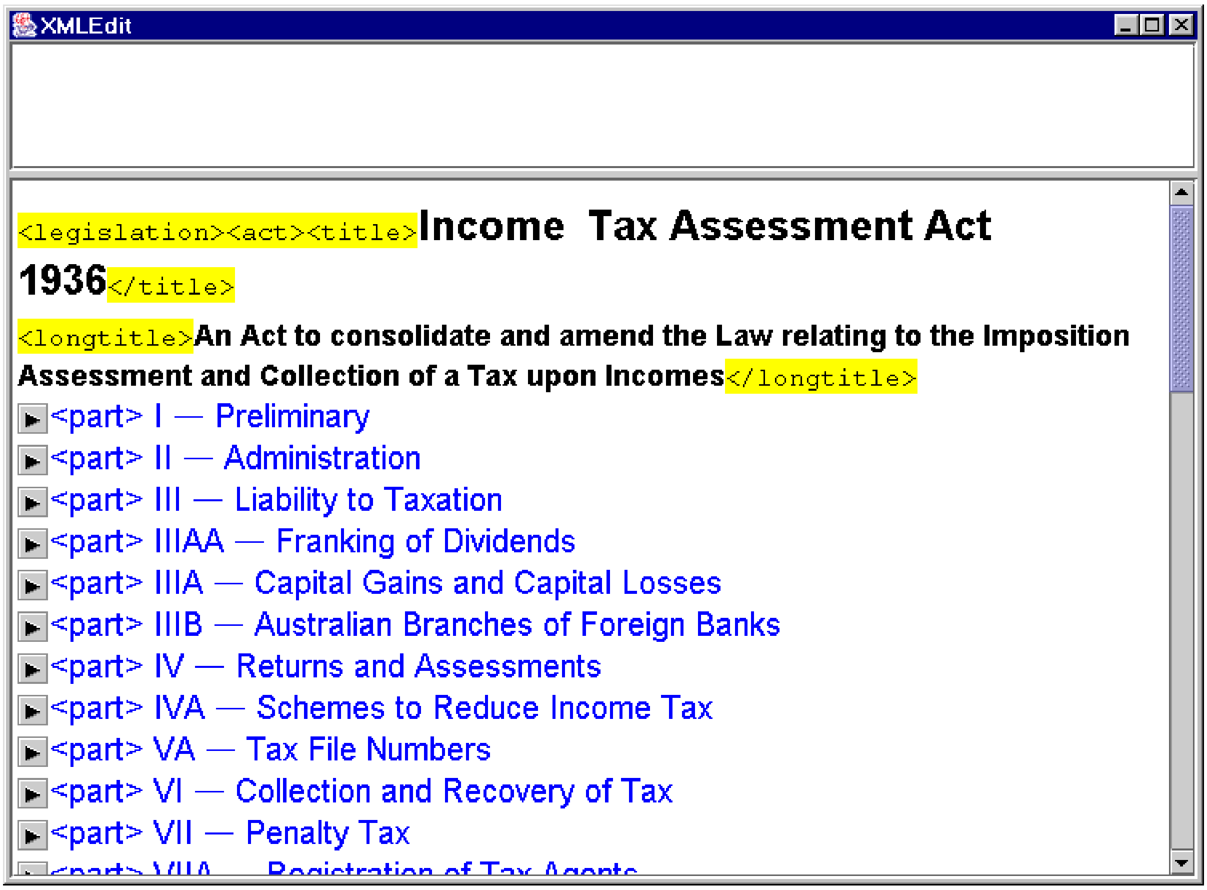
This is a conventional model used in a number of existing systems. For managing large documents, a number of extra features are added to the components:

- The repository stores an extra property for each node that gives the size of the sub-tree rooted at this node. (The precise definition of "size" may vary – the number of characters used is one obvious measure.)
- The server maintains a property that controls the size of the data returned as the result of a request by the editor (the "maximum edit size"). This value is arbitrary and can be varied according to application and user preference.
- The editor has a means of indicating which part of the document the user is currently editing, as well as other controls as discussed below.

When the user starts editing a document, the editor sends a request to the server to return the root node of the document. The data that is actually returned depends on the value of the maximum edit size, as follows:

- If the size of the sub-tree under this node is less than the maximum edit size, the entire sub-tree is returned.
- Otherwise, the children of this node are sorted into descending order of size of their sub-trees.
- The size of the node sub-tree is reduced by the size of each child sub-tree in turn until the size is less than the maximum edit size. Each child sub-tree used in this way is marked as "collapsed".

When the data is returned, each "collapsed" node is represented by a marker. Each marker contains a pointer to the node in the repository. Note that the size calculations above can be done very efficiently, since the repository stores a size value for each node.



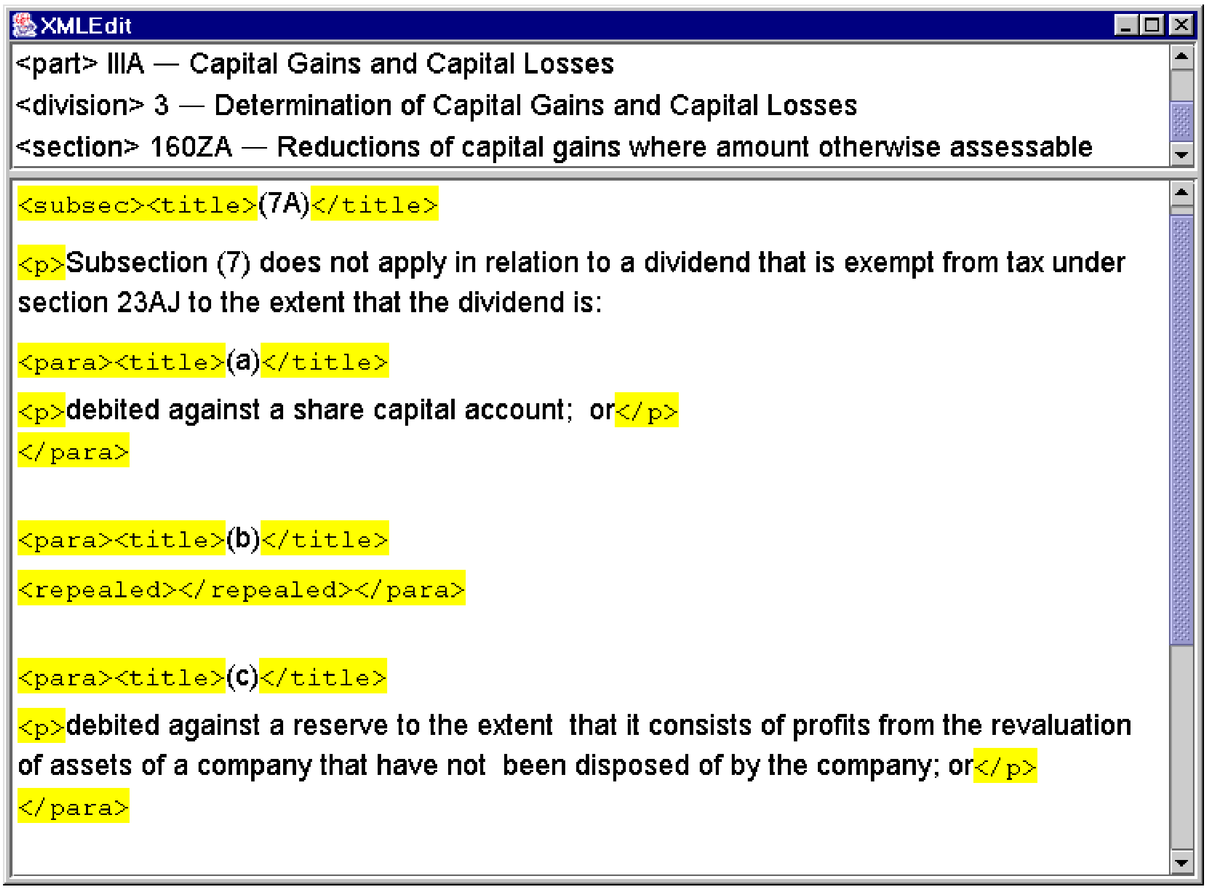
<legislation><act><title>Income Tax Assessment Act 1936</title>

<longtitle>An Act to consolidate and amend the Law relating to the Imposition Assessment and Collection of a Tax upon Incomes</longtitle>

▶<part> I — Preliminary
▶<part> II — Administration
▶<part> III — Liability to Taxation
▶<part> IIIAA — Franking of Dividends
▶<part> IIIA — Capital Gains and Capital Losses
▶<part> IIIB — Australian Branches of Foreign Banks
▶<part> IV — Returns and Assessments
▶<part> IVA — Schemes to Reduce Income Tax
▶<part> VA — Tax File Numbers
▶<part> VI — Collection and Recovery of Tax
▶<part> VII — Penalty Tax
▶<part> VIIA — Registration of Tax Agents

Editor showing root node

The editor displays the markers as links. The text of the link identifies the node (in the example, the text is the content of the child <label> and <title> elements; a more general solution would allow an XSLT or XPath expression to be used). If the user activates the link, the editor sends a request to the server to display the sub-tree rooted at that node and the above process repeats (exploiting the self-similarity of XML). This process can be repeated until a sub-tree small enough to be shown in full is chosen.



<part> IIIA — Capital Gains and Capital Losses
<division> 3 — Determination of Capital Gains and Capital Losses
<section> 160ZA — Reductions of capital gains where amount otherwise assessable

```
<subsec><title>(7A)</title>
<p>Subsection (7) does not apply in relation to a dividend that is exempt from tax under section 23AJ to the extent that the dividend is:
<para><title>(a)</title>
<p>debited against a share capital account;  or</p>
</para>

<para><title>(b)</title>
<repealed></repealed></para>

<para><title>(c)</title>
<p>debited against a reserve to the extent  that it consists of profits from the revaluation of assets of a company that have not  been disposed of by the company; or</p>
</para>
```

Editor showing a terminal node

The editor also displays the context of the current node in the top pane as a list of ancestor nodes. Each line is a link (using the same text described above) that can be activated to display the corresponding node.

The main advantage of this model for the user is that navigation and content editing are integrated in a natural way. It is common practice to have a separate navigation window showing the document as a tree, from which the user selects the node they wish to work on. Experience has shown that not all authors are comfortable with this paradigm (a fact that programmers tend to overlook).

Managing Concurrency

The above explains how the model allows a user to navigate in a document, but we also must consider the ability to make changes. The most important aspect of this is how we manage concurrency – making sure that two users can't make changes that interfere in a destructive way.

Some XML repositories allow locking by node, but this can cause problems. In practice it can require locking all of the descendents of a node, which can easily lead to unnecessary conflicts.

The new model can handle concurrency in a more efficient way. Only nodes that are returned “in full” need to be locked by the server. The only restrictions required for collapsed nodes are:

- A collapsed node cannot be deleted it there are any locks on it (in practice, deletions are the only operations that would require a lock on a complete sub-tree).
- The element name of a node cannot be changed if it currently appears anywhere as a collapsed node.

If these restrictions are in force, the editor can validate the node being edited without having to inspect the content of collapsed nodes (for this to work, all content models need to be context-independent, as they are in DTDs – a more complex approach would be required if content models were allowed to vary depending on context, as they are in some of the other schema languages).

Conclusion

The model discussed here, or something similar, may provide the means for XML data to be used as a generic platform for applications, in the same way that relational databases have done for tabular data. It is an incremental advance on current technologies, so its implementation is unlikely to be difficult. The chief challenge it presents is the relatively tight binding between editor and repository. The next step is a standard protocol that will facilitate multi-vendor solutions. Any takers?