High Performance XML Theory & Practice

XML Prague 2009

Alex Brown,
Director,
Griffin Brown Digital Publishing Ltd

Agenda

- Background
- XML and memory bloat: how bad is it?
- Underlying causes
- A proposed new model
- Implementation experiences
- Features of the model
- Taking it further?

Background

- Developing Java applications for processing XML – Schematron-ish.
- Why Java? – well …
- Working with documents (publishing) with models we didn’t devise and don’t like
- Read-only XML (so, not so hard)

Stating the problem

- Processing big XML documents is too slow
- And/or takes too much memory
- … in circumstances where we have to have an in-memory representation *
  - Tree representations are a reality of XML processing: expect their significance to grow

* probably ;-)

A test document

- What does “big” mean?
- Used to use one from a customer ...
- But now we have Ecma 376-1
  - aka DIS 29500
- A good test document of the “fairly big” class
- Approx 60 MB

Quantifying the problem

Benchmarks for operations on 60 MB XML document

	Time taken	Memory required
Build a DOM Document	14.1 s	231 MB
XSLT Identity Transform	40.7 s	237 MB
Parse (SAX)	5.7 s	< 2 MB

Challenges

- Can we improve on this?
- What is the root of the problem?
  – Does it even have a single “root”?
- Is there a ‘classic’ speed/memory trade-off that will thwart us?
- Even if we solve the problem, can we still use a familiar API?

Trade-offs?

“ It has been my experience […] that reducing a program’s space requirements also reduces its run time ”

- Jon Bentley

Observations

Bloaty implementations?
The trouble with Java

```
class Objs
{
        public static void main( String[] args )
        {
                // create one million small Strings
                String[] objs = new String[ 1000000 ];
                for( int i = 0; i < 1000000; i++ )
                {
                        objs[ i ] = ( "" + i );
                }
        }
}
```

50 MB

The Object overhead?

- We can reckon every java.lang.String costs at least 40 bytes
- And Objects have creation/destruction overheads too
- So a naïve implementation of an XML object model is going to be costly, right away
- But, 1 million bytes costs … 1 million bytes ☺

The trouble with DOM (etc.)

- DOM interfaces commit us to an Object-heavy implementation
- org.w3c.dom.Node declares17 methods that return an Object
- More generally, a tree-based implementation commits us to an Object-heavy experience if we use references to refer to Objects (e.g. parents/children)
- Difficult to use “standard” APIs here

Premises

- Beware Object!
  - byte[] is your friend
- Falling-back to a more primitive form of Java programming, avoiding large number of Objects
- Or - Java, but not as we (generally) know it

So what might a more primitive storage model for XML look like?

XML document as a stream

```
<root a='value'>
 <e>foo</e>
 <e>bar</e>
 <e>zxc</e>
</root>
```

Start document	
Start element	*root*
Attribute	*a*
Attribute Value	*value*
Character data	*{whitespace}*
Start element	*e*
Character data	*foo*
End element	
Character data	*{whitespace}*

etc

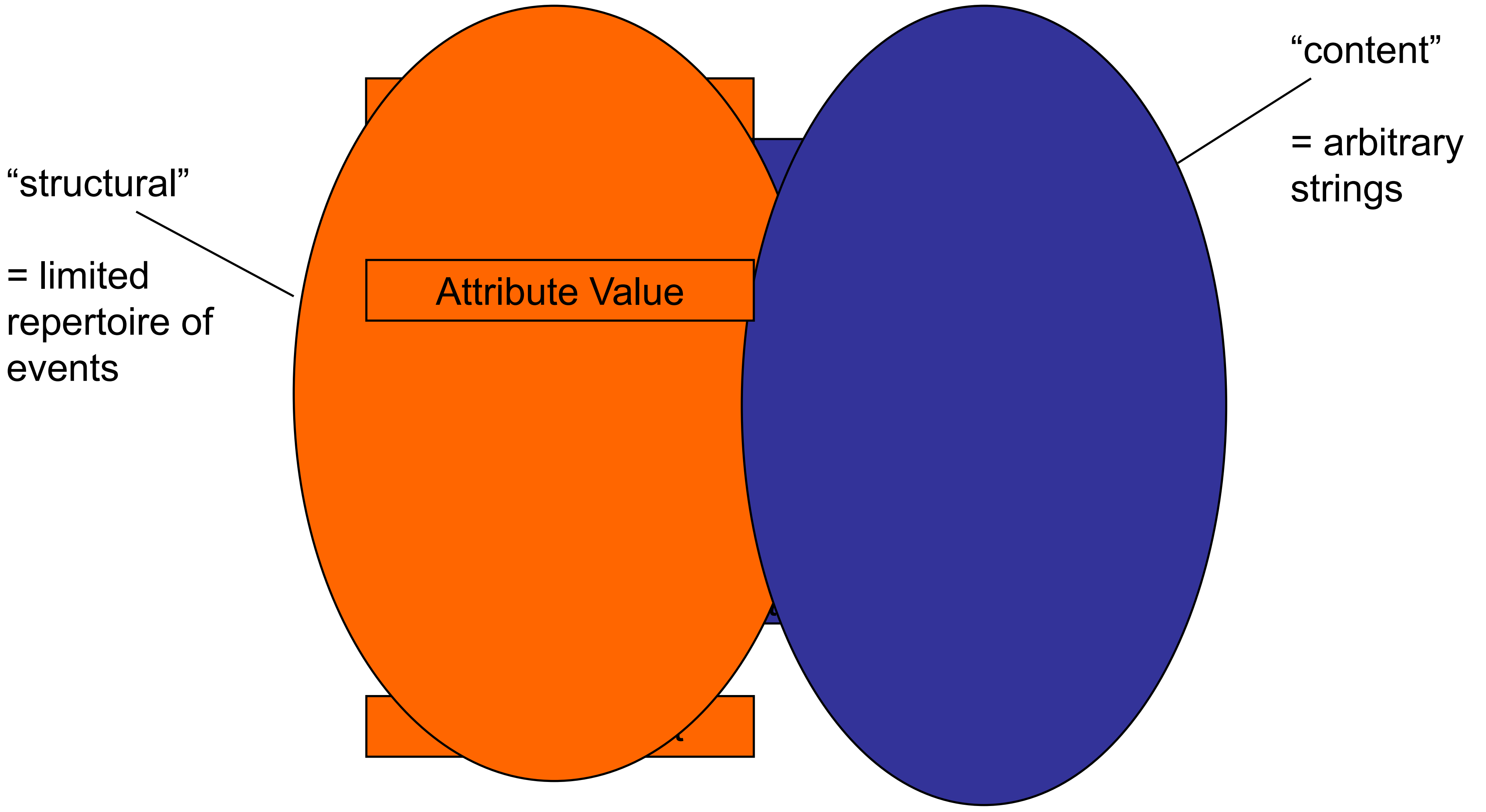
End document

Stream features

Start document	
Start element	*root*
Attribute	*a*
Attribute Value	*value*
Character data	*{whitespace}*
Start element	*e*
Character data	*foo*
End element	
Character data	*{whitespace}*
etc	
End document	

- Not a SAX stream
- Persistent
- More finely-grained
- “Piano roll”

Two types of phenomenon

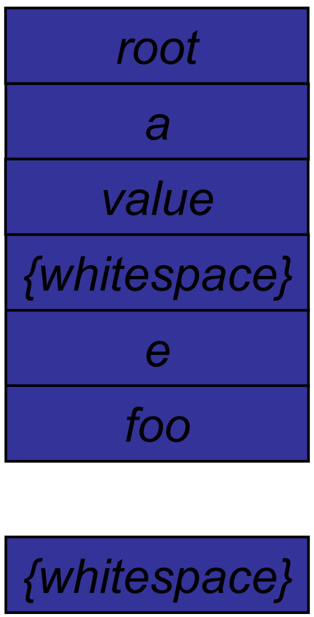


Representing structural phenomena with single bytes

Start document	0x80
Start element	0x81
Attribute	0x82
Character data	0x83
Start element	0x81
Character data	0x83
End element	0x84
Character data	0x83
etc	
End document	0xFF

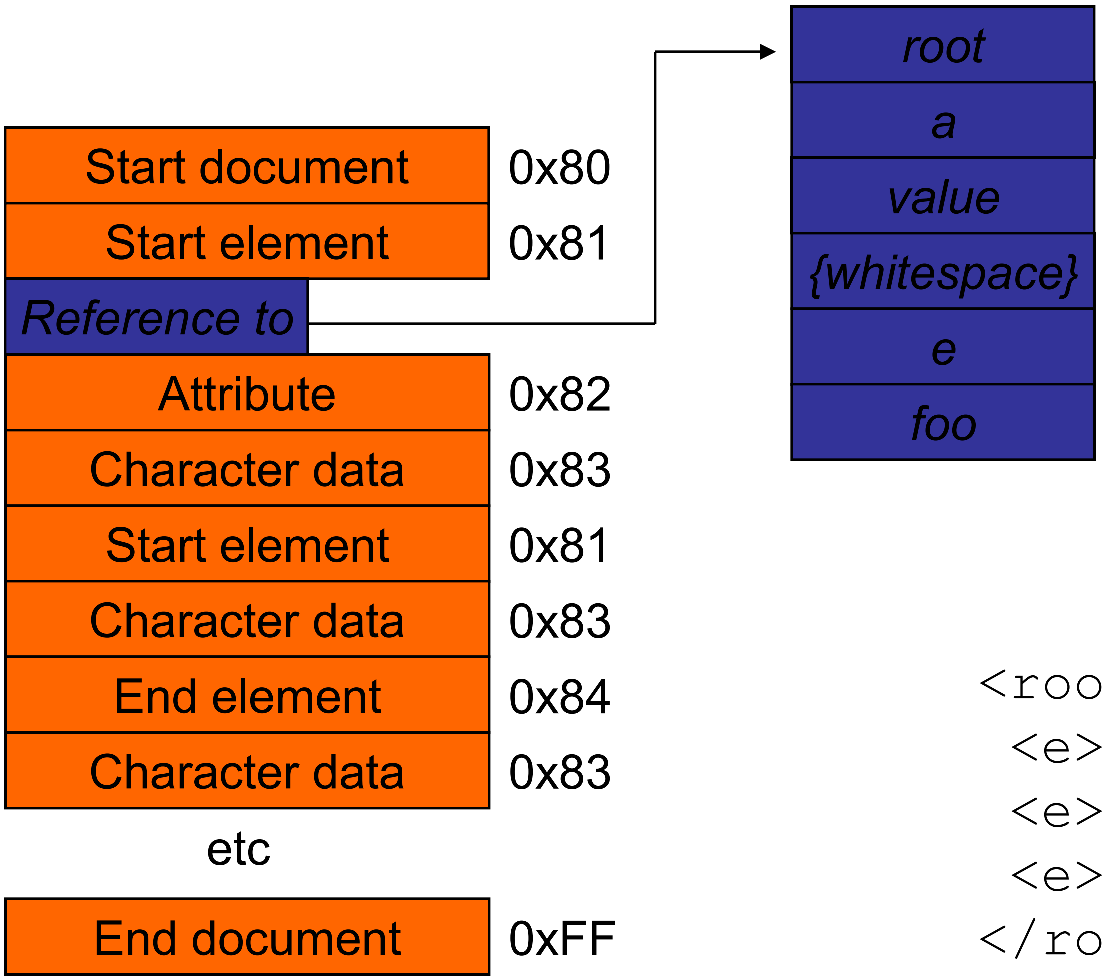
- Actual values are unimportant
- But notice the high bit is set for all these values
- And that we'll have plenty of high-bit values not taken by our usual infoset repertoire

String storage (1)



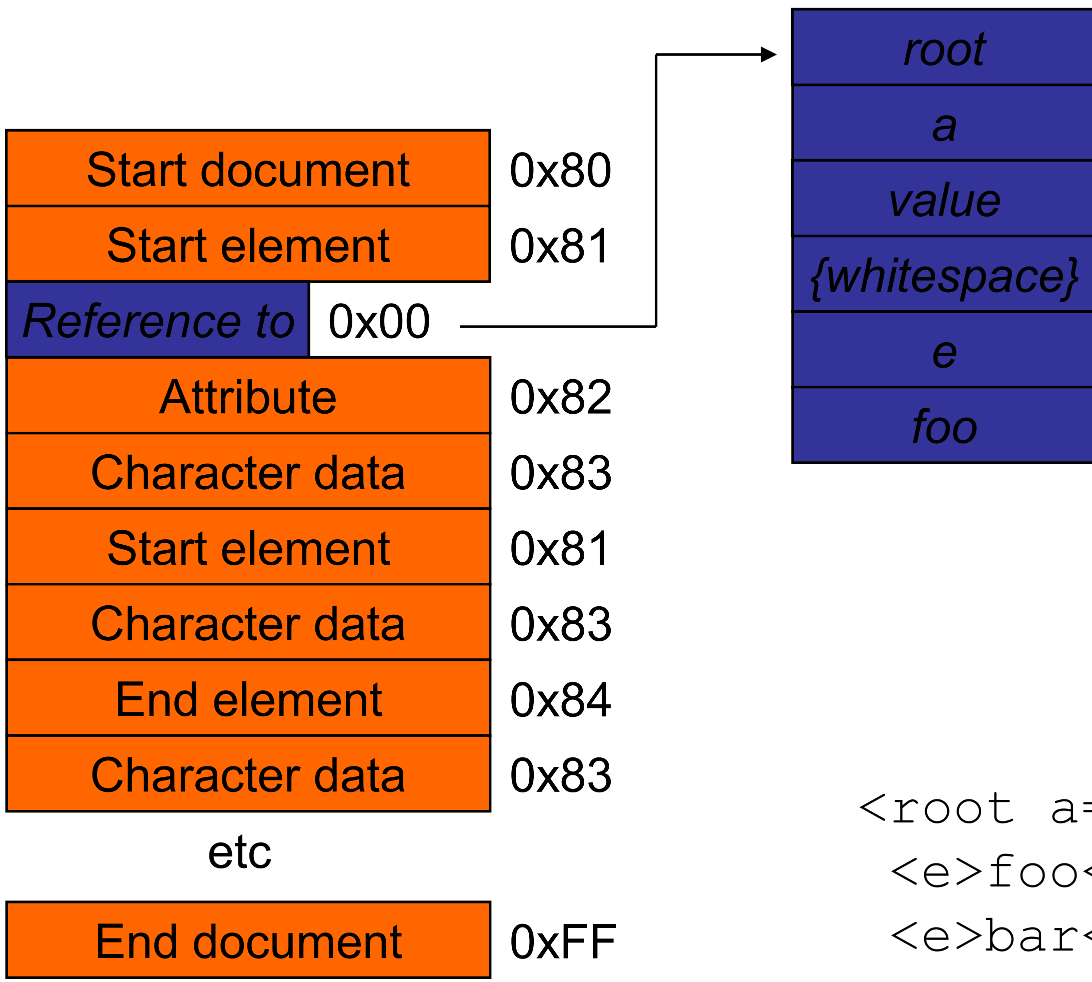
- Strings are after all, the most important things in your document!
- Use a dictionary
- Refer to strings by index
- XML documents always have at least one duplicate string!
- Often, lots
- So, normalisation would seem sensible

String Storage (2)




```
<root a='value'>
  <e>foo</e>
  <e>bar</e>
  <e>zxc</e>
</root>
```

String Storage (3)



- String events are always delimited by structural events
- We never set the high bit for string lookup values
- And use as many 7-bit numbers as we need to encode the lookup value

```
<root a='value'>
 <e>foo</e>
 <e>bar</e>
 <e>zxc</e>
</root>
```

Bitwise representation

High bit identifies structural events

0x80	0000000	START_DOCUMENT
0x81	0000001	START_ELEMENT
0x00	0000000	(ref to string)
0x82	0000010	ATTRIBUTE
0x01	0000001	(ref to string)
0x83	0000011	ATTRIBUTE_VALUE
0x02	0000010	(ref to string)
0x84	0000100	TEXT
0x03	0000011	(ref to string)

etc

Encoding larger values

- Say we have a text node that references string 200_{10}

0x84	10000100	TEXT
0x48	0[redacted]	(part of ref to string)
0x01	0[redacted]	(part of ref to string)
0x85	10000100	END_ELEMENT

Encoded as sequence of 7-bit numbers

etc

Alternative Serialisations ?

```
##############################

# string table (0 indexed)

root
a
val

# etc

##############################

STD      # start document
STE 0    # start element named as for string 0
ATT 1,2  # attribute named as for string 1, value of string 2
TXT 3    # text event
STE 4    # etc
```

Implementation Experience

Early Implementation

- Used a SaxReader to create the stream
- Used a HashMap of Strings for the string table (so, not optimal)
- Did not handle all of the infoset
- But, looked promising ….. so …. we went ahead and implemented it

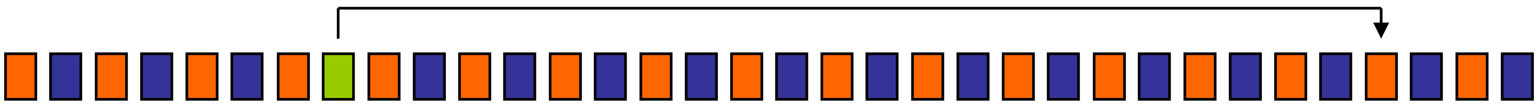
Issues

The demon of scanning

- The model as outlined so far is memory-efficient, but very slow to query
- Poor ‘random access’ performance to parts of the XML document, as compared with tree model
- Especially for operations like finding following-sibling or parent nodes

Stratagem #1: Pseudo-events

- Introduce pseudo events into the byte stream
- Informally stating e.g.: "following-sibling is 5,000 bytes this-a-way"



- Our reserved hi-bit values can be used
- This *is* a classic memory/speed tradeoff
- They can be placed arbitrarily

Signpost Events

- following sibling information
- preceding sibling information
- parent information
- … all specify new stream positions

(Other events)

- CDATA sections
- Line numbers
- Column numbers
- … customers value these pesky things

Stratagem #2: Better string representation

- Used a plain HashMap in proof-of-concept
- Not optimal for reasons noted earlier in this talk
- Instead better to use a sequence of chars and index into that
- (N.B. biting the two-bytes-per-char bullet)

Dynamic container woes

- Most Java containers (and our custom ones):
  - Resize when they need to (d'oh)
  - Double their capacity at that moment
  - Generally sane behaviour
  - But can lead to memory waste

Stratagem #3: Document Sniffing

- Parse the document once before building the tree
- Collect stats
- Precisely allocate structures necessary to hold that document's representation
- Remember - the importance of the transient memory use figure

Benchmarks

Benchmarks for operations on 60MB document

	Time taken	**Memory required**
Build a DOM Document	14.1 s	231 MB
Make Frozen Stream	11 s	117 MB
With physical locators	14.5 s	217 MB

Making it Useful

Just a Thought - An API?

- Do we *really* want/need another XML API?
- Nature of the ‘frozen stream’ suggests an iterator-based (cursor-based) API. Avoiding Objects.
- To correspond to something recognisable from the XML world, why not use XPath axes?

Making it XPath-queryable

- XPath is a sane way to interact with XML in code
- And enables Schematron implementation
- (Which is what we are interested in)
- Jaxen: stable, high-performance, conformant XPath library http://jaxen.codehaus.org/

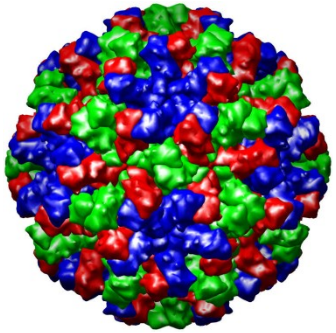
Integration with other XML libraries

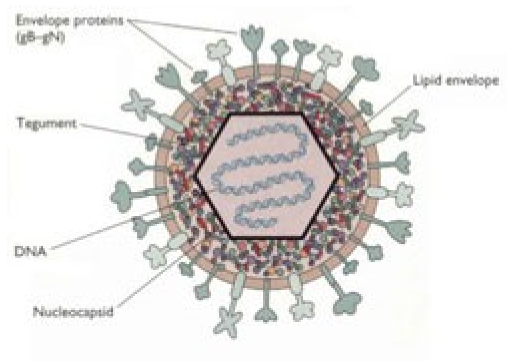
- Unfortunately, not if they expect a “tree of Nodes” model and/or Objects
- However Jaxen works with “any” model which can provide Axis iterators
- So theoretically we “just” need to provide XPath axis iterators on top of our frozen streams

Jaxen integration

- But:
  - Jaxen too is predicated on the representation of nodes as Objects
  - So now we “just” need to re-write Jaxen around arrays of ints (representing event indexes into frozen stream)
  - Some time later …

Preliminary Results





- Promising: 2x speed of Saxon/XSLT ISO Schematron, *but* using +30% memory
- Tunable to be leaner/slower
- Code to be released under GPL licence as “Probatron”.

Thinking Aloud

Other optimisations ?

- Use assembly language !
- Leverage parallel pipelines and multi-core features of modern chips ?
- Note Intel work in this area

Using other storage

- Frozen streams are highly amenable to being paged to disc
- Or split across machines

Extreme optimisations ?

- Similarities between our ‘frozen stream’ and multimedia streams? Use multimedia hardware? Blitting?
- Design custom hardware for stream processing

Conclusions

- In memory XML trees are still expensive
  - But real progress in past 36 months
- Saxon pretty much ticks all boxes; hard to beat!
- 100% streaming remains the holy grail
- Users may value the ability to choose good speed or memory-use performance
- Maybe scope for extreme optimisations

Thank you for listening