# XML Tree Models for Efficient Copy Operations

Michael Kay
*Saxonica*
<mike@saxonica.com>

**Abstract**

*A large class of XML transformations involves making fairly small changes to a document. The functional nature of the XSLT and XQuery languages mean that data structures must be immutable, so these operations generally involve physically copying the whole document, including the parts that are unchanged, which is expensive in time and memory. Although efficient techniques are well known for avoiding these overheads with data structures such as maps, these techniques are difficult to apply to the XDM data model because of two closely-related features of that model: it exposes node identity (so a copy of a node is distinguishable from the original), and it allows navigation upwards in the tree (towards the root) as well as downwards. This paper proposes mechanisms to circumvent these difficulties.*

## 1. Introduction

*An XSL transform takes linear time*
*If the input and output are almost the same.*
*Though the changes you make may be local and small*
*You still pay the price of transforming it all.*

Many XML transformations, whether expressed in XSLT or XQuery, copy large chunks of the input directly to the output, without change. This is typically an expensive operation, requiring both time and memory proportional to the size of the subtree being copied. This cost is particularly painful when a transformation operates incrementally, making many passes over the input, each of which only makes small changes.

For an example of such a problem, see [2]. In that paper I explored the possibility of writing an XSLT optimizer in XSLT. Optimization is essentially a series of transformations applied to an expression tree, so it is in principle a task to which XSLT should be well-suited; but my conclusion in that paper was that it wasn't feasible to achieve adequate performance, largely because each transformation step involved copying the large parts of the tree that remained unchanged. Recently in Saxonica we have been revisiting this problem (see [4]) because we

are interested in making the entire XSLT compiler portable across platforms, and the classic way of achieving this is to write the compiler in its own language. This led me to look again at the efficiency of transformations that make small changes to a large tree.

The fact that copying a subtree is expensive is a consequence of two particular rules in the XDM data model: (a) nodes have identity (which means that the expression `copy-of(X) is X` must return false – a copy of a node is not the same thing as the original), and (b) nodes have parents (which means that `exists(copy-of(X)/..)` must return false – when a node is copied, the copy is parentless). Any implementation of a copy operation that retains these properties without performing a physical copy of the subtree is going to be complicated.

In the XDM data model, the maps and arrays used to represent JSON structures do not have this property. In the tree representation of JSON, there is no way of navigating from an object to its parent; and there is no way of distinguishing two copies of the same object. This means that subtrees can be shared, which makes copying logically unnecessary (or to put it another way, producing a logical copy does not require producing a physical copy).

Saxon[1] implements maps and arrays using what I will call "versioned" data structures. (The names "immutable" and "persistent" are also used, but both adjectives have alternative meanings, so I will avoid them). In a versioned data structure, after any update operation, both the old and the new values are available for further processing, yet the new value shares memory with the old for those parts of the data that have not changed. Appendix A describes briefly how versioned maps and arrays work. A versioned data structure for XML trees is more difficult to achieve, because of the problems of node identity and parent navigation.

There's no intrinsic difference between XML and JSON at the lexical level that accounts for this deep difference between the way that XDM models the two cases. We could explore what happens when we add parent pointers to JSON trees, or we could explore what happens when we remove them from XML trees. This paper does the latter.

The ability to navigate from a node to a parent (and therefore, implicitly, to its siblings) is extremely useful, because it makes it possible to identify nodes of interest by their context as well as their content. In [3] I showed some use cases where XSLT 3.0 is applied to the task of transforming JSON, and the inability to access this contextual information proved a constant obstacle, to the extent that I concluded the easiest way to accomplish many JSON transformations was to convert the data to XML, transform the XML, and then convert it back to JSON.

[1]Some statements made here about the Saxon product refer to code that is implemented and tested but not yet released.

*Now the freedom to navigate upwards and down,*
*to parents, descendants, children and peers,*
*Means the rule for transforming a node in your text*
*Can refer to the context in which it appears.*
*So if* `xml:lang` *says your paragraph's Dutch,*
*This may affect formats for numbers and such,*
*But to determine what language applies at each point*
*You must know the container in which it is found.*

In the latest incarnation of the Saxon-JS product, we are using a JSON-based model internally to represent the interpreted expression tree, but we have found it necessary to introduce parent pointers to allow access to context information such as the static base URI and in-scope namespaces of an expression. This doesn't cause any problems in this case because the expression tree, by the time the compiler is finished with it, never changes.

Now: the main thrust of this paper is to show that providing the ability to navigate from a node to its parent does not necessarily imply that the stored tree needs to include parent pointers. There's another way to enable access to the parent, which is to remember, when you get to a node, how you got there. There's no way of getting to a node without going via its parent, so in principle you can always retrace your steps. Knowledge of the parent can thus be part of the information returned when a node is retrieved, even if the information is not actually stored with the node. Equally, the identity of a node (affecting the result of the XPath `is` operator) can be a function of how the node was reached: if we treat the identity of a node as a list comprising the identities of all its ancestors, that is, a path to the node from the root, then it does not matter if a physical subtree is shared by several logically separate XML documents: a node can be reached by more than one path, but it is the full path that establishes the node identity, so such a node has multiple identities depending on how you got there. With this insight, we can see that it should be possible to provide full XPath navigation capability on a tree with no stored parent pointers and no built-in notion of node identity.

The KL-tree described in section 3 is an implementation of this concept.

## 2. Push and Pull Processing

The semantics of XSLT 1.0 were written in terms of instructions such as `xsl:element` *writing nodes to the result tree*: the narrative was written assuming a push model where instructions push data to a destination. By contrast, for XQuery and later XSLT versions, the specification uses *pull* language: element constructors are expressions that return a result to their caller, namely a newly constructed element. The pull model implies bottom-up tree construction, where

leaf nodes are constructed first, and then grafted onto their new parents: this inevitably involves copying the node to give it a new parent, at each level of construction.

A processor that implements this literally as written is going to be very inefficient, because of the amount of tree copying needed. In practice there are a number of ways the repeated copying can be avoided:

- The implementation can use a push model internally. What happens here is that an instruction like `xsl:element` starts by emitting a *startElement* event to an output receiver; it then processes its child instructions (also in push mode), and finally emits a corresponding *endElement* event. If the output receiver is a serializer writing lexical XML, this approach means that the result nodes are never actually constructed in memory at all: the processor emits a sequence of events which are translated into lexical XML markup by the serializer. If the output is an intermediate tree, the receiver will be a tree builder that uses the stream of incoming events to construct an in-memory tree, but none of the intermediate nodes will ever need to be copied. This is the approach used by the Saxon-Java product.
- The implementation can use a bottom up model, and attempt to recognize where leaf nodes do not need to be copied, but can instead be directly grafted to their new parent by updating a parent pointer. This relies on being able to recognize that the leaf node exists solely for the purpose of creating the content of the next container, and will never be used as a parentless node in its own right. This isn't quite as effective as the push strategy, because it involves materializing the result tree prior to serialization, but it can still perform well. This is the approach used in Saxon-JS.

This paper suggests that a third approach might be possible, which is simply to ensure that copying a tree of nodes is extremely fast: ideally it should cost nothing. This is achieved by creating a virtual copy of the tree: a separate tree in terms of XDM node identity, but sharing the same underlying storage as the original. Our first attempt at this is the KL-tree, described in the next section.

## 3. The KL-Tree

This section describes the KL-Tree, an experimental implementation of the XML part of the XDM data model.

The data that physically exists in memory is the K-Tree, and its nodes are called K-nodes. Putting attributes and namespaces to one side for the moment, we have five node kinds: documents, elements, text nodes, comments, and processing instructions. Since comments and processing instructions behave just like text nodes, we can ignore them for the purpose of this discussion.

So, as a first approximation, the K-tree contains:

- Document nodes, which contain a sequence of child nodes
- Element nodes, which have a name, a sequence of child nodes, plus attributes and namespaces
- Text nodes, which contain a string value

K-nodes do not contain enough information to enable navigation to ancestors or siblings, or to enable sorting of nodes into document order. To achieve that, any navigation through the K-tree returns not the K-node itself, but an L-node; an L-node contains a reference to the K-node, plus additional information. Specifically, the additional information in an L-node comprises a reference to its parent L-node (with null used to indicate that the L-node is the root of the L-tree), plus the position of the L-node among its siblings in the L-tree. With this additional information, navigation from an L-node using any of the 13 XPath axes becomes possible, as does sorting into document order.

*Our initial invention to answer this question*
*Was a tree in which nodes pointed down but not up.*
*Elements reference children and text nodes;*
*The link is one way: you can only descend.*
*But now when a query selects a descendant,*
*We remember the path for retracing our steps.*
*The pointer to parent becomes now redundant*
*We can find a container, whatever our depth.*

The L-nodes are created on demand, when a node is retrieved in the course of navigation, and they are garbage-collected as soon as they are no longer needed. With a little bit of optimization, it is possible in many cases to avoid creating L-nodes that aren't needed, for example with an XPath expression `child::title`, we can arrange only to create L-nodes for those K-nodes that match the required name.

Two L-nodes are identical (in the sense of the XPath `is` operator) if their parents are identical and they have the same sibling position; so it's only root nodes that have intrinsic identity. It doesn't matter whether the two L-nodes are represented by the same Java object, or whether the K-nodes that they reference are represented by the same Java object: one Java object can represent several nodes, and several Java objects can represent the same node.

Similarly, sorting of L-nodes into document order can be achieved from knowledge of the parent nodes and sibling positions.

## 4. KL-tree Performance

Appendix A summarizes the execution time of various important operations, comparing the KL-tree implementation with Saxon's standard TinyTree imple-

mentation as well as the more conventional LinkedTree model, and (for completeness) the other tree implementation models supported by Saxon.

What these figures show is that the KL-tree is dramatically faster for one particular operation, that of grafting a tree into a new containing tree, but it is a little bit slower than the existing TinyTree implementation for many other operations. In particular, searching the KL tree is about 4 times slower. The KL-tree also uses more memory. This is not because the model is intrinsically inefficient; it just fails to reproduce some of the optimizations implemented in the TinyTree. The TinyTree achieves much of its fast search time by using arrays of data rather than linked objects to represent nodes, and because scanning an array is faster than following pointers in a linked list, it is hard for any implementation using linked lists to achieve comparable performance.

Sadly, this appears to be a show-stopper as far as incorporation into Saxon is concerned. The number of stylesheets that show an overall performance improvement from the KL-tree is small, and moreover, it's difficult to recognize them by static analysis. This means that the feature is only viable as a user-selected option, and we know from experience that only a very small number of users who stand to benefit from tweaking such features will actually understand the feature sufficiently well to take advantage of it. If only 5% of stylesheets stand to gain, and if only 5% of the authors of those stylesheets recognize the fact, then adding the feature will not create enough happiness in the user community to make it worth the trouble.

So let's throw this idea out of the window for the time being (Prague being a popular place for defenestration) and try something else. Since the TinyTree is delivering good all-round performance, let's see if we can use that as our baseline, and make incremental improvements.

## 5. The TinyTree

At this stage we need to explain the workings of the TinyTree, which is Saxon's default tree implementation. Although the data structure has been around for many years, and has changed very little, the only published information is the low-level internal Javadoc `https://www.saxonica.com/documentation/index.html#!javadoc/net.sf.saxon.tree.tiny/TinyTree`, plus a slightly out-of-date blog article [1].

The data structure consists of a set of arrays, held in the TinyTree object. The arrays are in three groups, where in each group the arrays can be considered to represent columns in a table. Using Java arrays to represent the columns of the table, rather than the conventional approach of using one Java object per row, accounts for much of the space saving benefits, and also provides for fast tree construction and navigation.

*The TinyTree structure makes no use of pointers;*
*Its content instead is arranged using vectors.*
*One holds the depth, a second the node kind,*
*A third holds the names, coded as numbers.*
*A search for descendants will step through these vectors*
*Comparing the node kind and name for a match.*
*With no pointer chasing, and no string comparing,*
*The search for a node is impressive to watch.*

The principal table contains one row for each node other than namespace and attribute nodes. These rows are in document order. The following information is maintained for each node:

- the depth in the tree
- the name code
- the index of the next sibling
- two fields labelled *alpha* and *beta*, described below
- the type annotation that results from schema validation (this array is absent for untyped trees)
- the index of the preceding sibling. This array is created lazily only when needed, the first time that the preceding-sibling axis is used for any node in this tree.

The meaning of *alpha* and *beta* depends on the node kind. For text nodes, comment nodes, and processing instructions these fields index into a string buffer holding the text. But for element nodes, *alpha* is an index into the attributes table, and *beta* is an offset into the namespaces table. Either of these may be set to -1 if there are no attributes or namespaces.

A name code is an integer value that indexes into the *NamePool* object: it can be used to determine the prefix, local name, or namespace URI of an element or attribute name. Name codes enable searching for elements and attributes using fast integer comparisons rather than string comparisons.

The attribute table holds the following information for each attribute node:

- a pointer to the attribute's parent element
- prefix
- name code
- attribute type
- attribute value

Attributes for the same element are adjacent.

The namespace table holds one entry per namespace declaration or undeclaration (*not* one per namespace node). The following information is held:

- a pointer to the element on which the namespace was declared or undeclared
- namespace prefix
- namespace URI

The links between elements and attributes/namespaces are all held as integer offsets. This reduces size, and also makes the whole structure relocatable. All navigation is done by serial traversal of the arrays, using the node depth as a guide.

Saxon attempts to remember the parent of the current node while navigating down the tree, and where this is not possible it locates the parent by searching through the following siblings; the last sibling points back to the parent. In the case where there is a large number of siblings, occasional parent pointers are inserted as pseudo-nodes to reduce the length of this search.

## 6. Virtual Copy

Existing Saxon releases include an optimization whereby an expression of the form

```
<xsl:variable name="x">
    <xsl:copy-of select="$doc//a/b/c"/>
</xsl:variable>
```

creates a virtual copy of the selected <c> element nodes, rather than doing a physical copy.

Rather like an L-node in the KL-tree model, the virtual copy is a wrapper node that points to the original node of which it is a copy. Many of the properties of the virtual node (for example, the name, type, and string value) are identical to the corresponding properties of the original. The mechanism can also handle some variation, for example there is scope for the original data to be schema-typed, while the copy is untyped. Navigating around the virtual tree is done, by and large, by navigating around the underlying physical tree, and then wrapping the resulting node. The main way in which the virtual copy differs from the original (apart from having a different identity) is that XPath navigation never strays outside the subtree that has been copied. Navigation from any node in the tree to its ancestors stops when it hits the root of the virtual copy; navigation from the root to siblings or parent returns an empty sequence.

Unlike the KL-tree, the virtual copy cannot be shared as a child of multiple parents. In fact, a virtual copy is always a parentless copy of some original tree or subtree: the original node may or may not have a parent, but the virtual copy never has. This gives it limited usefulness. Indeed, one could argue that it is only ever used to ameliorate code that was badly written in the first place, because it is essentially used only to eliminate copying that was never necessary. The relevant variable could equally well have been written as:

```
<xsl:variable name="x" select="$doc//a/b/c"/>
```

with no copying needed.

Although this mechanism has limited usefulness in its current form, it turns out not to be difficult to extend it. In particular, we can extend it so that:

- A virtual copy *V* is identified by a pair of nodes (*R*, *P*), typically in different trees. *P* is referred to as the grafting host: we are effectively grafting the tree rooted at *R* to a new parent *P*.
- *V* is deep-equal to *R*: they have isomorphic subtrees that share the same storage
- The parent of *V* is *P*, which in general is not the parent of *R*.

When navigating *V*, the result of any navigation within the subtree is a wrapper node (like the L-node described earlier) which remembers that the parent of *V* is *P* rather than *R*; and any navigation that strays from the subtree (which in practice will always reduce to a call on `V.getParent()`) uses this information to return to the tree containing the grafting host.

We can now look at extending the TinyTree model so that an element node in the model (which in the normal way would be immediately followed by all its descendant nodes in document order) can be replaced by a reference to an external element node which is deemed to be copied at the relevant position.

So the tree representation produced by the following construct:

```
<xsl:variable name="x">
    <out>
        <xsl:copy-of select="$doc//a/b/c"/>
    </out>
</xsl:variable>
```

would contain an entry for the document node at level 0, then an entry for the wrapper `<out>` element at level 1, then a number of "external element" nodes at level 2, each containing some kind of reference to one of the selected `<c>` nodes. For convenience, we'll call this the "host tree", and we'll refer to the trees containing the `<c>` nodes as "grafted trees". Of course, the process can be nested arbitrarily deep, in that grafted trees can themselves contain external element nodes to further copied trees.

How do we navigate such a structure?

Firstly, if we are processing the descendant nodes in the outer tree in document order, then when we hit an external element node, we have to remember where we are on some stack, and continue by processing the descendants of the grafted node. When we've finished scanning the grafted subtree, we pop the stack. This part isn't too difficult.

More tricky is that when we're processing the grafted tree, we have to remember where we came from. The rules are similar to those for the current VirtualCopy described in the previous section, but with some key differences:

- The parent of the root node in the grafted tree is no longer absent, it is the parent of the external element node in the outer tree.
- Similarly, the siblings of the root node in the grafted tree are the siblings of the external element node.

With these changes, a `copy-of()` operation on a TinyTree node creates a parentless VirtualCopy (as today), and an operation that attaches such a VirtualNode to a new parent, in the course of building a new TinyTree, adds a reference to the VirtualCopy. Both cases now take constant time independent of the tree size.

A minor refinement: for very small trees, for example those consisting of a single element node with a single text node child, it may turn out to be cheaper to perform a physical copy of the node.

Unfortunately, though, most of the implicit copying that happens during a transformation isn't done with explicit `copy-of()` operations, it is done using the recursive shallow copy implicit in the built-in template rules. So the next section studies how we can make recursive shallow copy equally efficient.

## 7. Shallow Copy and the Identity Template Pattern

Returning to the original use case, stylesheets that make small changes to large trees are often written to use a design pattern with a fallback rule that shallow-copies an element, overridden by higher-priority rules to make specific changes. For example, a stylesheet to delete all `<Note>` elements might be written like this in XSLT 3.0:

```
<xsl:mode on-no-match="shallow-copy"/>
<xsl:template match="Note"/>
```

In earlier XSLT releases the default action would be spelled out explicitly, for example:

```
<xsl:template match="node()|@*">
    <xsl:copy>
        <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
</xsl:template>
<xsl:template match="Note"/>
```

The problem here is that the code is copying elements from the source tree to the result tree one node at a time, which makes it difficult to take advantage of a fast deep copy.

The first thing we have to do is to detect that this pattern is in use: this is clearly easier to do with the declarative XSLT 3.0 approach. It's harder when the identity template is written out explicitly, because there are many variations on how it is written; however it should be possible to detect the common cases.

When we detect that the template rules for a mode use shallow-copy as the fallback action, with specific actions for a small number of match patterns, we can attempt an optimization: if there is no explicit template rule for an element, or for any of its descendants (or for their attributes, if applicable) then the element can be deep-copied to the result, with no further processing of the subtree.

*The identity pattern in XSLT*
*is troublesome mainly because you can't see*
*which nodes have subtrees that don't change one jot*
*and would benefit greatly from copying the lot.*

If the stylesheet is schema-aware then there is potential to recognize statically that there are some elements that will always be deep copied. Sadly, however, writing schema-aware stylesheets seems to have remained a minority interest, so this approach on its own won't get us very far. However, there might still be benefits from doing a dynamic check.

Specifically, if the following situation arises dynamically:

- `xsl:apply-templates` selects a node *N* for which there is no matching template rule
- the current mode (explicitly or implicitly) uses `on-no-match="shallow-copy"`
- *N* is a node in a TinyTree
- the instruction is evaluated in push mode
- the current output destination is a TinyTree builder

then it may be worth scanning the descendants of N to see whether any of them matches an explicit template rule; and if not, performing a deep copy-of() operation rather than a recursive tree walk.

The approach could be further improved with a learning strategy: if (say) the first ten nodes with a particular name M have been found to have descendants matched by an explicit template rule, then it's probably not worth considering this approach for any subsequent nodes named M.

An important caveat is that this tactic will never be used in the common case where transformation results are being written directly to a serializer. The cost of producing serialized XML will always increase with document size. The benefit comes when a pipeline of transformation phases pass data to each other in the form of in-memory trees (and it applies whether these transformation phases are written as a sequence of separate XSLT stylesheet executions or as a single XSLT execution).

## 8. Use in XQuery Update

In XQuery, the code for copying a tree with minor changes to selected nodes is somewhat tedious to write: essentially, the `xsl:apply-templates` mechanism

needs to be simulated with a recursive function that switches on the type of the supplied node using a `typeswitch` expression, each branch typically containing a recursive call to process child nodes using the same logic. In principle, the same optimizations could be applied as for the XSLT shallow copy pattern; but it is probably harder to detect this pattern in XQuery because there is more scope for minor variation in the code.

A second way to make small changes to a document in XQuery is to use XQuery Update. For example a query to delete all the `Note` elements at any level could be simply written:

```
delete node //Note
```

The downside of this mechanism is that the updated document (with the `Note` elements deleted) is not visible within the same query. Instead, some external mechanism (perhaps XProc) is needed to insert the updating query into a pipeline of operations. The XQuery update specification states that modifying nodes within a tree does not affect the node identity of other nodes: in effect, the tree becomes mutable; except that there is no way within the XQuery language of comparing the node identity before and after update to see whether the claim is actually true.

If updates are to be made visible within a query, the only way to achieve this is with the "copy-modify" expression (also known as a transform expression). An example might be:

```
copy $doc2 := $doc
modify (delete node $doc//Note)
return $doc2
```

In the sadly-abandoned 3.0 version of XQuery Update, this can be replaced with the simpler syntax:

```
$doc transform with {
   delete node .//Note
}
```

The result of this expression is now a copy of the subtree rooted at `$doc` in which the `Note` elements have been deleted. Pure functional behaviour and immutability have been restored by constraining the in-situ modification to work on a tree that is created as a copy of the original, where the copy becomes accessible to the rest of the query only in its modified form.

So we are once again in the situation where the cost of making this change depends on the size of the document, and not only on the number of nodes being changed.

With a construct like this, however, we have a very much better chance of exploiting a virtual copy mechanism. The first stage in evaluating this copy-modify expression is to produce a pending update list, which is a list of actions to be

applied to the tree, together with the nodes that they affect. We can then expand this list to include all the ancestors of affected nodes. Then, when performing the copy operation to construct `$doc2`, we can implement this by means of a recursive copy on all its children, and in the course of this recursive deep copy, any node that is `not` on the expanded list of affected nodes can be virtually-copied by creating a reference to the original node in `$doc`.

*In XQuery Update it's easy to see*
*that the modified nodes form just part of the tree.*
*Thus the list of those nodes that remain unaffected*
*Is readily formed, as might be expected.*
*And with virtual copies of parts that stay constant*
*Applying small changes takes only an instant.*

This promises to be sufficiently useful that we could consider providing XSLT syntax with the same semantics: the above example might become:

```
<uxsl:update select="$doc">
  <uxsl:delete select=".//Note"/>
</uxsl:update>
```

while a more complex example might be:

```
<uxsl:update select="$doc">
  <uxsl:delete select=".//Note"/>
  <uxsl:rename select=".//Comment" to="Remark"/>
  <uxsl:replace-value select=".//Salary" by=". * 1.1"/>
</uxsl:update>
```

The result of the `uxsl:update` expression would be the updated copy of `$doc`

## References

[1] Michael Kay *Saxon: Anatomy of an XSLT Processor*, 2005. Article published at IBM DeveloperWorks. Available at `https://www.ibm.com/developerworks/library/x-xslt2/`

[2] Michael Kay: *Writing an XSLT Optimizer in XSLT.* Presented at Extreme Markup Languages: Montréal, Canada, 2007. Available at `http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html` and at `http://conferences.idealliance.org/extreme/html/2007/Kay01/EML2007Kay01.html`

[3] Michael Kay: *Transforming JSON using XSLT 3.0.* Presented at XML Prague, 2016. Available at `http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf` and at `http://www.saxonica.com/papers/xmlprague-2016mhk.pdf`

[4] John Lumley, Debbie Lockett and Michael Kay: *Compiling XSLT3, in the browser, in itself.* Presented at Balisage: The Markup Conference 2017, Washington, DC, August 1-4, 2017. In Proceedings of Balisage: The Markup Conference 2017. Balisage Series on Markup Technologies, vol. 19 (2017). Available at `https://doi.org/10.4242/BalisageVol19.Lumley01`

[5] `http://www.xml-benchmark.org`

## A. Measurements

This appendix gives measured timings (in milliseconds) for various operations on various implementations of XDM trees. The measurements were made on an experimental version of the Saxon XSLT processor; the precise measurement configuration is not significant because we are only interested in the relative numbers. The data used was a 10Mb version of the XMark dataset[5].

The operations that we measured are as follows:

- Build: the time taken to build the tree by parsing a 10Mb source document.
- Scan: time taken to scan the tree in descendant order, sending SAX-like events to a sink receiver that immediately discards the data.
- Graft: time taken to build a new tree consisting of a document node, a wrapping element node, and a copy of the 10Mb source tree.
- Scan Grafted: time taken to scan the tree that results from the graft operation, again sending SAX-like events to a sink receiver that immediately discards the data.
- Search Grafted: time to execute the XPath query `count(//*[@id])` (which returns 6075 elements) on the tree that results from the graft operation.

These operations were timed with the following implementations of the XDM tree model:

- TinyTree (old): the TinyTree as available in the released Saxon product, without special support for virtual copying
- TinyTree (new): the TinyTree modified as described in this paper, to allow grafting of a virtual copy of a subtree
- Linked Tree: the Saxon Linked Tree, a conventional tree implementation where nodes are Java objects with pointers to children and parent nodes
- KL-Tree: the experimental KL-Tree model described in this paper
- DOM: the implementation of the W3C DOM interface packaged in the Oracle JDK
- Domino: a hybrid tree implementation introduced in Saxon 9.8, consisting of a DOM supported by additional indexes for fast searching

- XOM: see `https://xom.nu`
- JDOM2: see `https://www.jdom.org`
- DOM4J: see `https://dom4j.github.io`
- AXIOM: see `https://ws.apache.org/axiom/`

Here are the measurements:

**Table A.1. Measurements of various operations on various tree implementations**

| Model | Build | Scan | Graft | Scan Grafted | Search Grafted |
|---|---|---|---|---|---|
| TinyTree (old) | 120 | 6.6 | 85 | 9.2 | 6.3 |
| TinyTree (new) | 120 | 7.2 | 0.8 | 6.8 | 7.4 |
| LinkedTree | 125 | 19 | 65 | 25 | 15 |
| KL-Tree | 118 | 20 | 0.005 | 20 | 58 |
| DOM | 133 | 127 | 216 | 120 | 37 |
| Domino | 211 | 51 | 114 | 56 | 14 |
| XOM | 184 | 55 | 197 | 90 | 42 |
| JDOM2 | 133 | 64 | 164 | 86 | 30 |
| DOM4J | 134 | 96 | 188 | 108 | 205 |
| AXIOM | 145 | 82 | 158 | 98 | 127 |

How it was measured: using a microbenchmarking environment in Java, calling low-level interfaces to simulate the run-time activity of an XSLT or XQuery processor. Each test was run repeatedly for 10 seconds or more to warm up the Java hotspot compiler; this was then repeated for another 10 seconds to get an average execution time, and only the figures for the second run were recorded.

## B. Versioned Maps and Arrays

Given a map such as

```
let $old := map{"a":1, "b":2, "c":3}
```

it is possible in XPath 3.1 to perform an operation such as

```
let $new := map:put($old, "b", 17)
```

whose result is a new map,

```
map{"a":1, "b":17, "c":3}
```

After this operation, $old still refers to the original map, while $new refers to the new map. But the map:put() operation does not copy parts of the map that have

not been changed: the cost of the map:put() operation is essentially independent of the size of the map that is being modified.

Various data structures can be used to achieve this effect. The one that Saxon uses is a hash trie.

To simplify the actual implementation, we can consider that for each possible key value there is a hash code which can be viewed as a sequence of seven 5-bit integers. A tree of depth 7 with a fan-out of 32 can then be used to locate any value: an entry in the leaf nodes of this tree is a list of key-value pairs, where the keys are those sharing the same hash code.

Modifying the entry for one particular key then involves replacing 7 nodes in the hash trie, one for each level corresponding to the 7 components of the hash code. This will always involve a replacement for the root of the tree. All nodes in the tree other than these 7 can be shared between the old tree and the new. Modification thus has a constant cost: whether the map contains one entry or a billion, the put() operation creates exactly 7 new nodes in the tree.

In practice the actual hash trie implementation has optimizations that reduce the cost of handling very small maps, because these are quite common. For example a map of less than five entries is implemented as a simple list of key-value pairs.

A similar solution is used for arrays. In concept, an array is simply implemented as a map whose keys are integers. But because the structure needs to handle operations other than get() and put() efficiently (notably, retrieval of entries in key order), and because integers are not limited to 35 bits, the actual trie structure used internally is different.

See also: Wikipedia, *Persistent Data Structures*, `https://en.wikipedia.org/wiki/Persistent_data_structure`