Implementing Maps for XPath 4.0

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

The main purpose of this paper is to describe how the Saxon product implements maps as currently proposed for the 4.0 versions of XSLT, XPath, and XQuery. The specifications introduce new features that create a number of challenges. In describing the implementation we also have to introduce these new features.

(Saxon is an implementation of the XPath, XQuery, XSLT, and XSD specifications. Saxon 12, released in 2023, is the version in general use: it implements a few of the early features in the 4.0 drafts. Saxon 13 is being released in 2026, with much more extensive coverage of 4.0 features. SaxonJ is the version for the Java platform; SaxonCS is the version for .NET.)

1. Introduction

Maps were introduced as a new data type in XSLT 3.0 and XPath 3.1. The original motivation came from the XSLT Streaming project: if you're going to process a large document in streaming mode, then you can't look back at parts of the document that you've already skipped over. This means when you encounter a piece of data that you're going to need later, you need to remember it somewhere, and that needs a more versatile data structure than the atomic values and nodes of XSLT 2.0. Maps were first proposed for XSLT 3.0 to address this requirement.

Maps are often used in conjunction with another XSLT 3.0 feature, accumulators: accumulators provide a declarative way of associating data with particular nodes in a document, defined as a function of the value of the accumulator on previous nodes, and data found at the current node. They correspond closely with the fold operation familiar from functional programming languages, except that they are based on iteration over the nodes of a tree structure rather than the items in a linear sequence.

Later on in the development of XSLT 3.0 and XPath 3.0/3.1, interest grew in allowing XSLT and XPath to process JSON, and it was realised that maps also had a big role to play there, because the data model for XPath maps included the model for JSON objects as a subset. Features provided by XPath 3.1 maps that go beyond JSON objects include:

- The key of an entry can be any atomic item, not only a string.[1]

The associared value can be any XPath value, including for example a node, a function, or a sequence of atomic items.

Maps have proved a popular addition to the language, and the draft 4.0 specifications [2] enhance that capability in a number of ways, based on user experience. Some of those capabilities are fairly superficial, in that they provide new functionality on top of the existing data model. But others have a profound effect on the design on the underlying data structures.

(I'll use the term XPath 4.0 for convenience to refer to the full set of 4.0 specifications, including XSLT and XQuery.)

The most significant new capabilities are:

- Maps in 4.0 are ordered.

  The main reason for making maps ordered is to make the serialized output of the map human-readable. With small maps (as with attributes on an XML element) a human reader can cope with variations in the order of entries, but when maps contain large nested maps, it becomes impossible to find your way around.

  The ordering applied to XPath 4.0 maps is essentially "first in, first out": new entries are added at the end. This should not be confused with sorted maps, where entries are maintained sorted in key order. In some cases (for example a map holding events indexed by timestamp), an application can use the map order to mirror the sort order of keys.

  Making maps ordered reflects their treatment in other modern programming languages. Javascript has guaranteed the order of entries in objects since ES2015 (though its rules are not the same as those in XPath 4.0), and Python has guaranteed the order of entries in dictionaries since version 3.7.

  The fact that maps are ordered clearly impacts the way they are implemented internally; a straightforward implementation using a hash table is no longer viable, because hashing does not preserve order.

- Maps and arrays in 4.0 can be treated as a tree of JNodes, and this tree can be navigated using path expressions and axes in much the same way as XML-based trees (whose nodes are now referred to as XNodes).

  The ability to navigate the tree using the full set of XPath axes is only possible because the nodes are ordered, but an implementation that preserves ordering is actually not enough. For example, in Java if you wanted a map that reflected order of insertion you would probably turn first to the class `java.util.LinkedHashMap`. But while a `LinkedHashMap` allows you to iterate over all the entries in the map in order, it does not allow you to find a specific

[1] XPath 4.0 uses the term *atomic item* where previous versions used *atomic value*.
[2] See https://qt4cg.org/

entry by key, and then navigate to the following or preceding entries, which is required to support the `following-sibling` and `preceding-sibling` axes.

Before describing how we address these requirements in Saxon, it will be useful to describe how we implement XPath 3.1 maps (and why).

2. Implementing maps in XPath 3.1

XPath is a functional language, and its data structures must therefore be immutable. Adding an entry to a map does not modify the original, it creates a new map and leaves the old one unchanged. For this to happen efficiently, it needs to use a data structure where entries can be added or removed incrementally without making a copy of the entire map. Data structures that enable this are variously referred to as *immutable* or *persistent* data structures, or sometimes *functional* data structures. I don't find any of these terms very satisfactory, because they all have different meanings in different contexts: let's instead call them *progressive* data structures. By a *progressive* data structure, I mean one that allows modification, but that returns the modified version as a new value while retaining the old value intact; and with the implication that such modifications can be achieved without the cost of making a full copy.

Saxon's primary implementation for XPath 3.1 maps, from Saxon 9.6 until Saxon 12, was the class `net.sf.saxon.ma.map.HashTrieMap`. On Java it was implemented using a third-party open source structure called `ImmutableMap` developed by Michael Froh; on .NET it was underpinned by Microsoft's `System.Collections.Immutable.ImmutableDictionary`. These both provide very similar functionality (though I've no idea how similar they are internally), so we needn't dwell on the differences. The basic idea of these structures is that the data is held in an internal tree, and when an entry is added or removed, a new tree is created that shares all the parts of the old tree that haven't changed (which usually means most of it). The cost of adding or changing a single entry is therefore independent of the size of the tree.

The underlying structure (on Java, Froh's `ImmutableMap`) isn't actually a map from XPath atomic items to XPath sequences, as one might expect. That's because the XPath 3.1 model requires us to treat two keys as being equal while also being able to distinguish them. For example, two QNames can be equal despite having different prefixes, and two `dateTime` values can be equal despite having different timezones, so we need to ignore the prefix or timezone when comparing keys, but to retain it when enumerating keys using the `map:keys()` function. It's also worth noting that neither the Froh nor the Microsoft implementations of the structure allow externally supplied `equals()` or `hashcode()` callbacks. In addition, the rules for comparing keys in an XPath map are not the same as the rules for the `eq` operator: comparison of map keys needs to be transitive, error-free, and context-free, and the `eq` operator has none of these properties. For

all these reasons, the key held in the underpinning `ImmutableMap` is not the XPath atomic item itself, but an `AtomicMatchKey` derived from it, chosen so that the `equals` and `hashcode` methods behave the right way for XPath maps. [3]

So much for the key part of the underlying `ImmutableMap`. The corresponding value is an instance of `net.sf.saxon.ma.map.KeyValuePair`, which as the name suggests is a pair of objects, an atomic item and a sequence. When we access a map to retrieve values by key, we're only interested in the sequence, but when we iterate over the entries in a map (for example, when evaluating `map:keys()`), we return the atomic item.

The `HashTrieMap` isn't the only implementation of XPath 3.1 maps in Saxon 12. There are other more specialised implementations, all conforming to the same Java interface so they can be used interchangeably. The most interesting variant is `DictionaryMap`, which handles cases where (a) all the keys are instances of `xs:string` (which means we don't need to store a type label with every entry), and (b) incremental modification using `map:put()` and `map:remove()` is unlikely. This is used, for example, for the maps that result from parsing JSON objects. If the application does decide to use `map:put()` or `map:remove()` on such a map, we take a hit by converting the `DictionaryMap` into a `HashTrieMap`.

There's one other little requirement that affects the design. What happens when you pass a map to a function that declares the required type of the argument as, say, `map(xs:integer, xs:QName)`? In a naive implementation, we would have to scan all the entries in the map checking that the keys are all integers and the associated values are all QNames. In some cases static typing might enable us to bypass this check, but those cases are probably rare. To handle this we do a little bit of optimization: as part of a `HashTrieMap`, we maintain fields for the "known key type" and "known value type". To check whether the map is an instance of a particular map type, we first check these values; if the known key type is a subtype of the required key type, and the known value type is a subtype of the required value type, then we need do no more. If not, we scan the whole map to establish a new known key type and known value type. The `map:put()` and `map:remove()` operations adjust the known type of the new map as necessary.

3. Implementing maps in XPath 4.0

We've discussed some of the new features of maps in 4.0: how does our implementation need to change?

[3] Wrapping an `AtomicValue` object in an `AtomicMatchKey` wrapper is potentially very space-expensive for a large map. However, for the common case of string keys, it enables an optimization: we can actually drop a layer of wrappers. Inside every `net.sf.saxon.value.StringValue` is a `net.sf.saxon.str.UnicodeString` object, and our `UnicodeString` interface implements `AtomicMatchKey` directly.

The first thing we did was to see whether we could find an existing implementation of immutable ordered maps that we could build on. Initially we thought we had found what we needed in the VAVR[4] project. This offers a `LinkedHashMap` that at first sight fits the requirement. Unfortunately we hit a performance bug[5] that made this a non-starter. One of the contributors to the VAVR project had fixed this with a redesigned class in a different way, and had made this available in a forked project, but there was little sign of the bug being fixed on the main branch. For a while we used the forked version, though we were a little uneasy about issues of support. But then the XPath spec introduced JNodes, and with it, the requirement to move from an entry to following and preceding siblings, and there seemed to be no immediate prospect of finding a third-party library that met that requirement, so we decided to think again.

Another factor here was that we needed solutions both for Java and C#. If VAVR didn't quite fit the bill in the Java world, we couldn't find anything remotely close for .NET.

It might be worth an aside here about the software engineering constraints involved in using open source component software. There have been some high-profile cases in the last year or two of big companies being hit by software vulnerabilities in open source libraries, and our customers are becoming increasingly anxious to seek evidence that we manage these risks, not just in our own software, but in our upstream supply chain. How do you go about reassuring yourself, let alone your customers, that a library you've downloaded from GitHub contains no malicious code? VAVR includes contributions from over 100 developers scattered around the world, and it contains about 50K lines of code. Where do you start? If the software had actually met requirements, we would have made the effort. But it didn't.

So, we started thinking about a do-it-yourself implementation.

Some observations that might influence the design (some of these have already been noted):

- Many maps are never modified after they are first built. A design that optimizes for this use case seems appropriate.

- More surprisingly, many maps are never used for retrieval by key. For example, when transforming JSON, one might parse a file containing thousands of JSON objects, and in most cases the vast majority of these will be either copied unchanged to the output, or discarded entirely. So there might be benefits to be obtained by lazy construction.

[4] https://github.com/vavr-io/vavr

[5] https://github.com/vavr-io/vavr/issues/2727#issuecomment-2567809521

- Most maps use string-valued keys. In many cases all the keys are of type `xs:string` (and not a subtype), so maintaining a type label for each entry is an unnecessary overhead.

- Most maps are small. Most of the options maps supplied to functions like `fn:xml-to-json()` contain only one or two entries. For a small map, it's quite acceptable to do a linear search to find an entry. In fact the 4.0 rules for options arguments say that it's an error to include an entry that's not defined in the spec, so the implementation has to scan the entire map anyway: indexing it adds no value.

- It's common for many maps to share the same set of keys. This is particularly true with the introduction of record types in XPath 4.0 (a record type defines a fixed set of keys, together with the required types of the corresponding values; records are maps and all map operations apply to records). But it also arises with other data, for example the `fn:parse-json()` function might well process a file containing a thousand maps each of which has exactly the same keys. Optimizing for this case would be useful.

- We may want to provide a range of possible implementations, and the best time to choose an implementation is when all the keys and values are known. This reinforces the point that it would be a good idea to provide a map builder that first simply gathers a list of key-value pairs, and only then decides what kind of map to construct.

So, whenever possible, we construct maps by supplying a list of key-value pairs to a mutable map builder; during this phase of processing all that the map builder does is to accumulate the list of key-value pairs.

For example, the first stage of building a map such as:

```
map{"LHR": "London",
    "LAX": "Los Angeles",
    "CBR": "Canberra",
    "YUL": "Montreal",
    "CIA": "Rome"}
```

is to construct a list of key-value pairs like this:

	Keys	Values
1	LHR	London
2	LAX	Los Angeles
3	CBR	Canberra
4	YUL	Montreal
5	CIA	Rome

Figure 1. Output of Map Builder

(Implementation details: the builder constructs two lists, an `ArrayList<AtomicValue>` for the keys and an `ArrayList<GroundedValue>` for the associated values. On completion, these are converted into two arrays, of type `AtomicValue[]` and `GroundedValue[]`.)

Before the map can be used, the only essential operation is to check the keys for duplicates. This can be done very cheaply by passing the keys through a Bloom filter. A Bloom filter typically maintains a bit array of say 65536 bits, and has a hash function that returns three independent numbers in the range 0..65535 for any key value[6]. Having applied this hash function to an input key, the Bloom filter checks whether all three bits in the bit array are set; if so, we have a potential duplicate. If not, we know that it cannot be a duplicate. If we're unlucky, and a possible duplicate is indicated, then we build an index of all the keys to make sure. [7]

At this point operations that scan over the entries in the map can be evaluated simply by iterating over the list of key-value pairs constructed by the builder. Many operations are now possible on the map without needing to build an index. For example:

- Functions such as `map:keys()`, `map:entries()`, or `map:for-each()` can be evaluated by scanning the list of key-value pairs.
- The value of `map:size()` is known.
- The map can be serialized using the JSON output method.
- It is possible to ascertain whether the map is an instance of a type such as `map(xs:string, node())`.

But suppose that the next thing that happens is a `map:get()` operation. We now need to build an index (except in the case where this was already built when

[6]The literature on Bloom filters gives formulae for optimizing the size of the bitmap and the number of bits that should be set. Our approach in comparison is somewhat rough and ready.

[7]Note that duplicate keys in a map under construction are not necessarily an error. There are various ways they can be handled, for example by taking the first duplicate, or the last, or by combining them into a sequence of values.

the Bloom filter detected possible duplicates). The index is an internal map from `AtomicMatchKey` keys (derived from the atomic items in the key list) to the integer offset of the key and value in the key-value-pair list constructed by the builder.

The structure now looks like this:

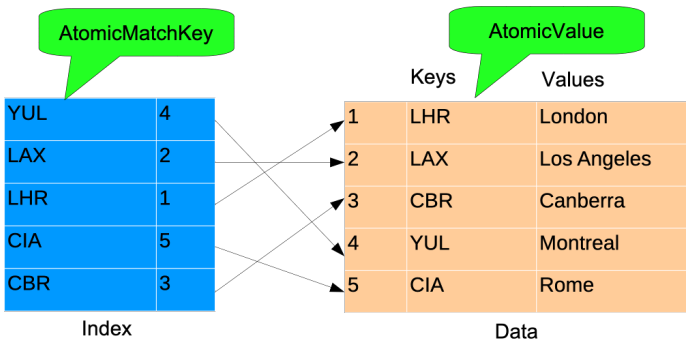

Figure 2. Solid Map with Index

Note that the index itself contains no ordering information, but the order of entries is maintained in the list of key-value pairs.

A `map:get` operation computes the `AtomicMatchKey` corresponding to the required atomic value, uses this to look up an integer offset, and then returns the value at that offset in the list of key value pairs.

(Implementation details: the index is a Java `HashMap<AtomicMatchKey, Integer>`. In this example, where the keys are strings, the `AtomicMatchKey` will be a `UnicodeString`; for strings consisting entirely of Latin-1 characters, the `UnicodeString` will be implemented as a `Twine8`, which wraps a `byte[]` array holding one character per byte.)

Now, if we're unlucky, someone will decide to do a `map:put` or `map:remove` operation. What happens now?

In our first iteration of the design, the first `map:put` or `map:remove` led to a bulk reorganization of the data structure. The list of key-value pairs was replaced by a progressive list (one that allows low-cost modifications without changing the original). The index from keys to offsets was replaced by a progressive map implementation (the same `HashTrieMap` that we used in Saxon 12). In order to avoid changing any integer offsets, `map:remove` did not actually remove the entry, but rather added its offset to a list of logically-deleted entries.

This design worked well for the vast majority of use cases, but we found it performed chronically badly in one or two applications. Specifically, we found an application that after constructing a map, added one single entry to it many times over. So we refined the design to avoid the bulk copy.

In the final design, a map in general consists of two parts: a *solid* part constructed by the builder which never changes once constructed, and a *fluid* part containing subsequent changes. The solid part is the structure we have already

seen: it consists of a list of key value pairs and an index to offsets in this list, neither of which will ever change, so it can be implemented using regular Java or C# structures (A `HashMap<AtomicMatchKey, Integer>`, an `AtomicValue[]` array, and a `GroundedValue[]` array).

The fluid part is essentially a list of changes that need to be applied to the solid part. It consists of the following components:

- A removals list: a set of integers, the offsets of entries that are no longer present because of `map:remove` operations.
- A replacements list: a map from integer offsets to values, identifying entries where the value associated with a key differs from the original value in the solid part.
- An additions list: this is similar to the solid part of the map: it contains a map from `AtomicMatchKeys` to integer offsets in a list of key-value pairs. The difference is that both the map and the list are progressive data structures, so incremental changes are possible.

The data structures used in the fluid part of the map are all progressive data structures, so they can be efficiently modified. The solid part represents the map as it was immediately after the initial build; the fluid part aggregates all subsequent changes.

Retrieval operations need to look in both parts. Update operations (`map:put` and `map:remove`) only affect the fluid part.

So if we add another airport to our example map, the structure will look like this:

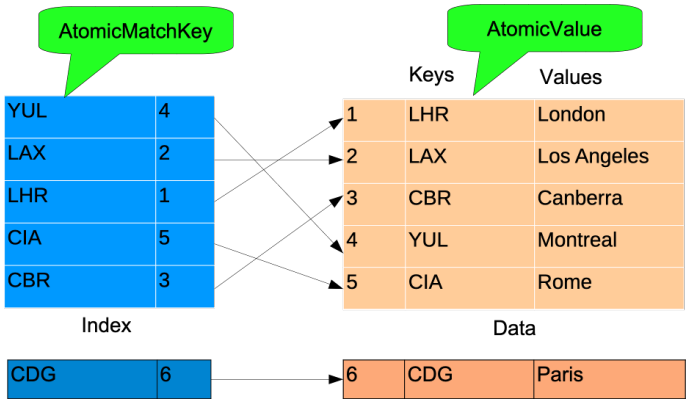

Figure 3. Modified Map

Retrieval operations now have to consider both parts of the map:

- `map:get()` looks first in the fluid part of the map; if it finds nothing, it looks in the solid part.

- Scanning operations such as `map:keys()` look first in the solid part of the map, skipping any keys that are also present in the fluid part; then the entries in the fluid part are appended.

There are a couple of subtleties here. The specification says that when `map:put` replaces the value for an existing key, the position of that entry in the map ordering does not change. So when we are scanning, then for each key-value pair that we encounter, we must skip it if it is present in the removals list, and we must return the replacement key and value if it is present in the replacements list. The specification leaves it implementation-defined whether a `map:put` operation replaces the key as well as the value (consider the case where `map:put` changes the value associated with a given QName, but supplies as a key value a QName with a prefix that differs from the original). The Saxon implementation retains the original key value.

In a pathological case, for example a map holding timed events where new events are continuously added and old events are continuously removed, we might get to the point where every single entry in the solid part has been removed or replaced, and in that case we can reorganize the map.

4. Performance Measurements

The main performance objective in this exercise was to ensure that introducing ordered maps did not cause a significant performance regression between Saxon 12 and 13. We have run a few simple performance measurements to convince ourselves of this; this doesn't claim to be a comprehensive analysis, and there are many external factors that might distort the result. But from the figures given below, the results give confidence.

First I have measured map construction using a MapBuilder versus incremental construction using successive calls of `map:put`. This was done using direct calls of the Java implementation classes, not by means of XPath expressions, in order to avoid extraneous factors such as XPath parsing cost intruding on the measurements.

For map sizes of 1000, 10,000, and 100,000 entries, with Saxon 13 timings in microseconds for bulk construction using a MapBuilder were 193, 478, and 9210μs respectively. Corresponding timings for incremental construction using a sequence of `map:put` operations were significantly slower: 240, 1949, and 29,368μs. (The test case built the map and then did a single `map:contains` operation to force indexing.)

In Saxon 12 the equivalent figures for bulk construction were 214, 2122, and 37,699μs, and for incremental construction 138, 1527, and 20,561μs.

So at first sight, bulk construction in Saxon 13 with the new design (which is likely to be the dominant mode of processing) is significantly better than in Saxon 12, while incremental construction is perhaps a bit slower, but not worryingly so.

In Saxon13, making one million `get` requests on a map with one thousand entries took 29,423µs. The corresponding figure with Saxon 12 was 29,245µs - effectively identical.

In Saxon13, evaluating `map:keys` on a map with one hundred thousand entrues took 9092µs. The corresponding figure with Saxon 12 was 40,978µs. Scanning a map sequentially can be expected to be considerably faster with the new data structure, since the keys are held in a single list, and have good memory proximity leading to good CPU caching.

The above figures are all for SaxonJ. There is no reason to expect a significant difference with SaxonCS.

5. Shaped Maps

When many maps share the same sequence of keys, but with different values, then it makes sense to hold the sequence of keys once, rather than repeating it in every map.

Saxon 13 implements a structure called a `ShapedMap` that works this way. The list of keys is called a `Shape`, and the `ShapedMap` consists essentially of a reference to the `Shape`, plus an array of slots holding values of the entries.

A `ShapedMap` does not have any kind of index. Most such maps are small, and a sequential search for the key can be expected to perform well. In addition, lookup expressions that reference the key statically (for example `$person?name`) are very common, and these expressions use a simple caching mechanism: if in one evaluation `$person` is a `ShapedMap`, and in the next evaluation `$person` is another `ShapedMap` with the same `Shape`, then it knows that the offset in the array of values will also be the same as last time.

Saxon 13 uses shaped maps only for records, corresponding to either a built-in record type defined in the language specification, or a user-declared record type. There is scope to extend their use to other use cases, notably to the maps that result from JSON parsing. This requires the parser to recognize that multiple maps are using the same structure.

In fact, the general design for 4.0 maps, with its use of a list of key-value pairs, can easily be extended to combine with the general idea of shaped maps. Since the general structure uses a HashMap from keys to integer offsets, and the keys are the same in multiple maps, this HashMap can be shared. This is a potential development beyond Saxon 13.

6. Typing and Coercion

XPath allows an expression of the form `$map instance of map(K, V)`, testing whether all the keys in a map are of type `K` and all the values are of type `V`. Such a test is also performed implicitly when a map is passed as an argument

to a function that expects a map of a particular type. This test is potentially very expensive, involving a full scan of the entries in the map.

We attempt to reduce this cost through caching. If no cached data is available, we evaluate the instance-of test by examining every entry in the map. We then record the fact that the map is known to be an instance of `map(K, V)`. This information is retained (or suitably amended) in any new map constructed using `map:put()` or `map:remove()`. If there is a test for some other unrelated map type `map(K2, V2)`, then we potentially re-evaluate the condition from scratch; this is hopefully rare. But it's theoretically possible, for example, that a map that wasn't an instance of `map(K, V)` becomes valid against that type after a sequence of `map:remove()` operations, and we need to allow for this.

When a map is passed to a function that requires an argument of type `map(K, V)` then the language spec requires the system to coerce the supplied map to the required type, which may potentially involve coercing all the keys (or all the values) individually. We're looking at how to do this operation lazily to reduce the cost, but we don't yet have a really efficient solution to this requirement. The requirement is particularly onerous when the map contains function items, because in this case function coercion is needed even if the supplied function is a subtype of the required function type.

A. JNodes

As mentioned, XPath 4.0 introduces the idea of being able to navigate a tree of maps and arrays using path expressions that are very like the traditional paths used to navigate XML-based trees of elements, attributes, and text nodes. The key to making this possible is the idea of a tree of JNodes, where the JNodes encapsulate the maps and arrays and their entries and members.

JNodes are described in [Kay 2025], and more fully, of course, in the draft 4.0 specifications. But a summary might be appropriate here, since it imposes requirements on the design of maps as presented in this paper.

An array can be wrapped in a JNode. It is then possible to navigate to the children of the JNode, which correspond to the members of the array. The JNode that wraps one of these children retains not only the value of the relevant array member, it also retains a property identifying the parent JNode (wrapping the original array), and the member index within the array. This makes it possible to navigate upwards from one of these child JNodes back to the parent, and also sideways to the siblings, representing the other members of the array. If the value of the array member is a map or array, then it is also possible to navigate downwards, to the grandchildren of the original array.

Similarly, a map can be wrapped in a JNode. Navigating to the children of this JNode delivers a set of JNodes that correspond to the entries in the map (retaining order). These child JNodes contain not only the value of the map entry,

but also the key, and a reference to the parent map. So again, navigation becomes possible upwards and sideways as well as downwards.

All the other axes (with the exception of the attribute and namespace axes) are defined as combinations and closures of the basic relations of a node to its children, parent, and siblings, so all these XPath axes become equally applicable to trees of maps and arrays.

This is achieved without giving maps and arrays a persistent identity. This means that two maps with the same entries can always be treated as being the same map, which is what makes it possible for a small modification to a large map to retain the parts of the map that have not changed: small modifications to a large tree take constant time.

B. The Journey: How did we get here?

The design presented in this paper is the current stage of a long journey, many of whose steps have been documented in previous conference papers. There have been many false turns along the way. Rather than a traditional list of references, I would like to give an account of this journey, highlighting which ideas worked, which were discarded, and which were heavily amended in the light of experience.

My first exploration of this area was in [Kay2007] where I explored the possibility of writing an XSLT optimizer in XSLT. It seemed to me that optimization is largely a matter of transforming expression trees, and since transforming trees is what XSLT is all about, the idea ought to be feasible. I quickly discovered that if the trees were implemented as XNode trees then it wasn't possible to achieve anything like good enough performance. The basic reason is that the design of XNode trees (in particular the fact that XNodes have node identity, and a reference to their parent) makes it very hard to find a way of making a small modification to a large tree in constant time.

I returned to this theme in [Kay2018a], where I explored the idea of a K-Tree, a tree where nodes were materialized on demand in the course of navigation [1], enabling the result tree of a transformation to share subtrees of the source tree if they had not been modified. The practicalities of implementing this proved too complex: considerations of node identity, document order, namespace context and the like meant that no performance benefits were achieved in practice. Although the design did make some operations substantially faster, the more basic and common operations of tree navigation all became a bit slower because of the extra complexity.

However, the experience was useful, in several ways.

[1] An idea I subsequently learned was well known in functional programming circles, and was referred to as a Zipper [Huet1997]

- Firstly, we achieved some significant applications that used XSLT to transform code.

  One was the XX compiler, a compiler for XSLT, generating SEF files for execution in the browser using SaxonJS: this is described in [Kay&Lumley2019]. Acceptable performance was achieved primarily by reducing the number of transformation phases in the compiled pipeline, and doing more work in each phase.

  Another was the transpiler that we use internally to convert the Java code of the Saxon product into equivalent C# code: the transpiler is written primarily in XSLT, and is described in [Kay2021]. This is essentially a four-pass transformation. The first pass (written in Java) converts the Java syntax to XML. The second pass constructs a digest of the source tree, building an index of all the packages, classes, and methods. The third pass analyzes this digest and decorates it with information needed to generate the C#, for example by determining which methods should be receive `override` or `virtual` modifiers. The final pass takes the entire XML representation of the syntax produced in phase 1, and serializes it as C#, using the information found in the digest.

  Although the transpiler was based on transformations of XML trees, and is used daily in production in that form, I also prototyped a redesign using maps and arrays. This case study contributed many of the ideas for the 4.0 specification of maps and arrays, ideas which I presented in [Kay2025a]. That paper preceded the introduction of JNodes, but reading the conclusions of the paper, the essential ideas for JNodes were all there, and led directly to the introduction of JNodes at [Kay2025b].
- The idea behind JNodes [Kay2025b] was strongly influenced by my work on the K-Tree design [Kay2018a], and the problems in implementing it, caused by node identity, namespace context, and parentage, and it reused the idea of building nodes with parent pointers dynamically in the course of tree navigation, enabling a modified tree to share subtrees with the original and thus allowing modification in constant time. Awareness of these problems explains why I successfully argued during the development of XSLT 3.0 and XPath 1.0 that maps and arrays should not have identity or parent pointers, and this design decision can be seen as paving the way to making JNode navigation possible.

References

[1] . Gerard Huet. *The Zipper*. Journal of Functional Programming. 7 (5): 549–554 doi:10.1017/s0956796897002864. S2CID 31179878.

[2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages, Montreal, 2007. Available at `http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html`.

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

[4] Michael Kay. *XML Tree Models for Efficient Copy Operations*. XML Prague 2018. Available at `https://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf` and at `https://www.saxonica.com/papers/xmlprague-2018mhk.pdf`.

[5] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Markup UK 2018. Available at `https://markupuk.org/2018/Markup-UK-2018-proceedings.pdf` and at `https://www.saxonica.com/papers/markupuk-2018mhk.pdf`.

[6] Michael Kay and John Lumley. *An XSLT compiler written in XSLT: can it perform?*. XML Prague 2019. Available at `https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf` and at `https://www.saxonica.com/papers/xmlprague-2019mhk.pdf`.

[7] Michael Kay. *<transpile from="Java" to="C#" via="XML" with="XSLT"/>*. Markup UK 2021. Available at `https://markupuk.org/2018/Markup-UK-2021-proceedings.pdf` and at `https://www.saxonica.com/papers/markupuk-2021mhk.pdf`.

[8] Michael Kay. *XSLT Extensions for JSON Processing*. Balisage 2022. Available at `https://www.balisage.net/Proceedings/vol27/html/Kay01/BalisageVol27-Kay01.html` .

[9] Michael Kay. *Processing JSON with Template Rules*. Markup UK 2025. Available at `https://markupuk.org/webhelp/index.html#ar04.html`.

[10] Michael Kay. *JNodes: a New Model for Navigating JSON Trees*. Balisage 2025. Available at `https://balisage.net/Proceedings/vol30/html/Kay01/BalisageVol30-Kay01.html` .