A note on Editor performance

A story on how the performance of Fonto came to be what it is, and how we will further improve it

Stef Busking
FontoXML
<stef.busking@fontoxml.com>

Martin Middel
FontoXML
<martin.middel@fontoxml.com>

Abstract

This paper will discuss a number of key performance optimizations made during the development of Fonto, a web-based WYSIWYM XML editor. It describes how the configuration layer of Fonto works and what we did to make it faster. It will also describe how the indexing layer of Fonto works and how we improve it in the future.

1. Introduction

1.1. How does Fonto work?

Fonto is a browser-based WYSIWYM[1] editor for XML documents. It can be configured for any schema, including many DITA specializations, JATS, the TEI, docbook and more. Fonto configuration consists of three parts:

1. How do elements look and feel (the *schema experience*)
2. How can they be mutated (the *operations*)
3. The encompassing *user interface* of Fonto

The schema experience is specified as a set of rules that assign specific properties to all nodes matching a corresponding selector. These selectors are expressed in XPath.

Operations also make use of XPath in order to query the documents. Effects are defined either as JavaScript code, or using XQuery Update Facility 3.0.

The user interface of Fonto has several areas (e.g., the toolbar, sidebar and custom dialog boxes) in which custom UI can be composed from React components. These can observe XPath expressions to access the current state of the documents

[1]What You See Is What You Mean

and be updated when it changes. The documents themselves are rendered recursively by querying the schema experience for each node and generating HTML appropriate for the resulting configuration.

1.2. What is performance?

When a single key is pressed, Fonto needs to update the XML and then update all related UI. This includes updating the HTML representation of the documents, recomputing the state of all toolbar buttons based on the applicability of their operation in the new state, and updating any other UI as necessary.

Typically, such updates involve looking up the values of various configured properties for a number of nodes (by re-evaluating the associated XPath selectors against those nodes) and/or executing other types of XPath / XQuery queries. In order to keep the editor responsive, these updates need to be implemented in a way that scales well with respect to both the complexity of the configuration as well as the sizes of the documents being edited. In order to keep Fonto easy to configure, we should not place too many requirements on the shape of this configuration. This means Fonto has to deal with a wide range of possibilities regarding the number of selectors etc.

When we started Fonto, we considered documents of around 100KB to be ‘pretty big’, and these could be pretty slow to work with. After heavy optimization, we now have workable editors that load documents of multiple megabytes[2], using (automatically updating) cross references, (automatic) numbering of sections and more. This paper details a few of the most significant optimizations we have applied in order to get to that point.

2. Accessing schema experience configuration

As described in the introduction, Fonto uses XPath selectors to apply a set of properties to nodes. We call the combination of a selector and a value a declaration.

Example of the look and feel configuration of the ‘p’ element:

```
configureAsBlock(sxModule, 'self::p');
```

This configuration does the following internally:

[2]Using just-in-time loading to only load a small subset, this even scales to working in collections totaling in the hundreds of megabytes, but that could be considered cheating.

Table 1. Summary of properties set for a paragraph

Property	Value
Automergable	false
Closed	false
Detached	false
Ignored for navigation	false
Removable if empty	true
Splittable	true
Select before delete	false
Default Text Container	none
Layout type	block
Inner layout type	inline
… (a total of 23 properties, plus optionally up to 35 more that are not set automatically)	...

There are about 23 properties being configured for a single paragraph, each specifying whether the paragraph may be split, how it should interact with the arrow keys, how it behaves when pressing enter in and around it, etcetera.

2.1. Orthogonal configuration

A number of these properties can be set individually, such as the background color or the text alignment of an element. This allows for a drastic reduction in the amount of selectors. Previously, when configuring some paragraphs to have a different background color compared to the 'generic' paragraph, all of the 'the same' properties also needed to be configured. By adding a way to configure single properties, reductions of more than three quarters of the configuration were seen.

Table 2. Orthogonal configuration

Without using Orthogonal Configuration	With Orthogonal configuration
`configureAsBlock( sxModule, 'self::p', 'paragraph' );` `configureAsBlock( sxModule, 'self::p[@align="right"]', 'paragraph with right alignment', {align: 'right'} );`	`configureAsBlock( sxModule, 'self::p', 'paragraph' );` `configureProperties( sxModule, 'self::p[@align="right"]', { markupLabel: 'paragraph with right alignment', align: 'right' } );`
2 x 23 properties, plus one, for the alignment = 47	23 properties, plus one, for the alignment, makes 24

For a property like how an element behaves when computing the plain text value from it, the registry may look like this, for the p element. Note that multiple of these selectors are automatically generated.

Table 3. Properties defined for a paragraph

Selector	Plain text behavior	Priority
`self::p and parent::*[( self::list-item ) and parent::* [ self::list[ @list-type="simple"]]]`	interruption	2
`self::p and parent::*[(self::list-item) and parent::*[ self::list[ @list-type="roman-upper" and @continued-from ] ]]`	interruption	2
<18 rows omitted for clarity>		
`self::p[parent::def]`	interruption	0
`self::p`	interruption	0
`self::*[parent::graphic]`	removed	0

2.2. Selector buckets

As shown earlier, selectors are used extensively in the configuration layer. For some selectors, it is quite obvious to see that a given node will never match a given selector. For example, the selector `self::p` may never match the `<div />` element.

We leverage this knowledge by indexing the selectors that are used in configuration by a hash of the kind of nodes they may 'match' their 'buckets'. We currently use node type buckets - derived from the nodeType values defined in the DOM spec[8] - and node name buckets derived from the qualified names of elements.

Table 4. Buckets

Selector	Bucket
`self::p`	name-p
`self::element()`	type-1
`@class`	type-1 (only elements may have attributes)
`self::p or self::div`	type-1
`self::comment()`	type-8
`self::*`	No bucket: both attributes and elements may match to this selector

Note that some of these selectors could also be expressed as a list of more specific buckets. For example, `self::*` could be stored under both the bucket for `type-1` as well as the one for `type-2` For simplicity, and to keep lookups by bucket as efficient as possible, we have currently limited our implementation to a single bucket per selector. We may revisit this decision in the future.

We then group the selectors that configure a certain property by their bucket. By computing the same hash(es) that may apply for a node, we drastically reduce the amount of selectors that need to be tested against any given node.

2.3. Selector priority / optimal order of execution

2.3.1. Conceptual Approach

An application may have the following configuration for the 'italic font' property:

Table 5. Italic font per selector

Selector	Value
`self::cursive`	true
`self::quote`	true
`self::plain-text`	false
<default>	none

The ordering of selectors is defined using a *specificity* system inspired by CSS: We group and count the amount of 'tests' in an selector: a selector with two attribute tests is more important than one with a single attribute test. Additionally, we allow applications to define explicit *priorities*. Specificity is used only if priorities are omitted or to break ties when priorities are equal.

The selectors defined by this piece of configuration will be evaluated in order and the value of the first match will be returned. In this example, a <p /> element will have no configuration for the 'slant' property, while the <quote /> will set it to 'true' and the <plain-text /> will set it to 'false'.

2.3.2. Optimization

The ordering of declarations does not mean all of these selectors have to be executed in that specific order. In the table defining the properties set for a paragraph, all of the high-priority selectors have a very low probability of matching. The much simpler `self::p` selector is more likely to match. To generalize this problem, we use a Bayesian predictor for the likeliness of whether a selector will match a given node.

The hypothesis (H) is that this selector matches. Evidence (E) is the hash assigned to the node. This is configurable, but usually the name of the element we input. We want to compute the probability of H given E: the selector matches for this hash. Bayes theorem gives us that $P\left(H\middle|E\right) = \frac{P(E|H) \quad P(H)}{P(E)}$ where $P(E|H)$ is the percentage of matches of this selector that match this hash. Basically, this is the amount of times the selector matched a similar element, continuously approximated based on previous results. $P(H)$ is the percentage of matches of this selector overall, and $P(E)$ is the percentage of results of any selector for a node with this hash. Because we will compare these scores for the same hash, the $P(E)$ part is constant and can be omitted.

We use the statistical probability of the selectors we will evaluate to determine an optimal order of execution of selectors. If we evaluate all selectors in order of decreasing likeliness, we only need to check selectors with higher priority *but a different value* in case of a match. In pseudocode, this becomes:

```
Let declarations be all declarations that may match the input, based on
buckets.
Sort declarations based on their priority, their specificity and lastly
on order of declaration.
Let skippedDeclarations be an empty list.
Let declarationsInOrderOfLikeliness be declarations, sorted using the
Bayesian predictor from most likely to least.
For likelyDeclaration of declarationsInOrderOfLikeliness do:
  If (likelyDeclaration.selector does not match input) continue;
  // We have a likely match, see whether it was the 'good' one
  For declaration of declarations do:
    If (declaration.selector is equal to likelyDeclaration.selector)
      // The likely declaration is the most matching one
      Return likelyDeclaration.value;
    If (declaration.value is equal to likelyDeclaration.value)
      // No need to evaluate this selector now,
      // it would result in the same value
      Add the declaration to skippedDeclarations, continue;
    // This higher-priority declaration would result in a different value
    If (declaration.selector does not match the input) continue;
    // This declaration applies, unless one of the skipped declarations
(with higher priority) matches as well
    For skippedDeclaration of skippedDeclarations do:
      If (skippedDeclaration.selector matches input)
        Return likelyDeclaration.value
    // We have no declaration that is deemed more important
    Return declaration.value
```

Fonto ends up querying a large number of declarations for all nodes in the loaded documents as a result of rendering and other initial processing. This means that the initial set-up will make sure that the Bayesian predictor is sufficiently trained by the time the user starts editing.

2.3.3. Performance impact

Worst case: This algorithm has the same worst-case performance as the implementation without it. The worst case will be triggered when the most likely match is also the least important one, and all preceding declarations point to another value. In this case, the algorithm will be forced to evaluate every preceding selector.

Best case: The most likely selector is preceded by a large amount of more complex selectors, which point to the same value. The algorithm will only evaluate a single selector: the most likely one. Because these selectors are prefiltered by their bucket, this is the more likely case: it is more common to configure a number of paragraphs to have the same declared value in for instance enter behaviour than all having different values.

2.3.3.1. Measurement

We conducted a performance test of the initial render of a JATS document of 721KB, containing 18826 nodes in the configuration that was highlighted in the table describing the properties set for a paragraph. These performance tests measured how long it took to render all of the content to html elements using Chrome 81 in Fonto 7.9.0. Tests were repeated 4 times.

Table 6. Performance of the Bayesian predictor

	Amount of XPaths evaluated
Without the Bayesian predictor	121575
With the Bayesian predictor	109964

With the optimization, we see a 9.5% reduction in the amount of XPaths that are being evaluated. The total load time is reduced by three seconds. This is a significant improvement over the old situation.

Furthermore, we measured how many times certain XPath expressions were executed. The following expressions stood out:

Table 7. XPaths with a fewer executions with the Bayesian predictor:

Selector	Execution count without predictor	Total time spent executing this expression	Execution count with predictor	Total time spent executing this expression

`self::*[`
` parent::*[`
`  self::term-sec[`
`  not(ancestor::abstract or`
`   ancestor::boxed-text)]]]`
`and not (`
` self::node()[`
`  not(self::sec or`
`   self::term-sec)]`
`)`
	1797	93 ms	900	42 ms

`self::label`	79	2ms	1662	47 ms

`self::label[`
` parent::abstract]`
	79	3 ms	1	(Too low to measure)

`self::label[parent::fn]`	1733	60 ms	1091	42 ms

`self::named-content[`
` @vocab="unit-category"]`
	2173	160 ms	1174	96 ms

`self::named-content[ @vocab="specification"]`	325	22 ms	538	56 ms

From this table, the label selectors stand out the most: the `self::label` selector grew both in execution count and in the total spent. This effect is explained by the next selector: `self::label[parent::abstract]`. This selector is part of a set of twelve similar selectors that went for 79 executions to a single one. The Bayesian predictor learned that the `self::label` select is more likely to match than the `self::label[parent:abstract]` and prevents executing it.

2.3.3.2. Comparison to another approach

In order to verify the results of the Bayesian predictor, we compared it to another, similar approach. Instead of using the predictor as the main sorting function, use the 'complexity of a selector. In other words, consider 'simpler' expressions to be more likely to match than 'complex' selectors. In order to approximate the 'complexity' of a selector, use the specificity algorithm as described in an earlier section.

This gave us the following results for the selectors mentioned in the previous chapter:

Total amount of XPaths executed: 111864.

Table 8. Performance metrics of using selector specificity as likeliness

Selector	Execution count without predictor	Total time spent executing this expression
`self::*[ parent::*[ self::term-sec[ not(ancestor::abstract or ancestor::boxed-text)]]] and not ( self::node()[ not(self::sec or self::term-sec)] )`	899	43 ms
`self::label`	1684	46 ms
`self::label[ parent:abstract]`	0	-
`self::named-content[ @vocab="unit-category"]`	2165	175 ms

The table gives interesting results: the `self::label` selector is executed many times, but the `self::label[parent:abstract]` selector is never executed at all.

However, the moderately complex `self::named-content[@vocab="unit-category"]` selector is evaluated way more often than when using the Bayesian predictor.

When going through the configuration of this editor, this can be explained. The 'normal' `<named-content />` element is expected to never occur in the editor in question. It is configured to never be rendered. However, the more special `<named-content />` elements that have additional attributes set are expected to occur, and are given a number of additional visualization properties, such as widgets, additional options for a context menu etcetera.

In essence, the `self::named-content` occurs few times in the total configuration, while the specific versions occurs many times. However, some specific versions of this element occur more than others; the Bayesian predictor takes advantage of this while this approach can not hold it into account.

2.4. Deduplication of duplicate property values

This best case is further leveraged by deduplicating duplicate values. In some cases, the configuration API allows one to input instances of functions. We rewrote these APIs to allow for better memoization: all function factories attempt to return the same function when called multiple times with the same arguments.

2.5. Related work

While the selector-to-value configuration in Fonto looks like how XSLT links up selectors to templates, they differ on a fundamental point: Templates in XSLT are usually unique to a selector; they see little reuse. The value space of a configuration variable in Fonto is usually small: they mostly consist of booleans and any non-discrete data is grouped nonetheless by the deduplication mechanisms described earlier. This makes optimizations like the Naive Bayes optimization work out of the box.

Lumley and Kay present optimizations for the XSLT case. In particular, they highlight the common use of DITA-class-substring selectors in DITA cases. However, such selectors and associated optimizations are not as applicable in Fonto. While Fonto does offer an abstraction[3] over the dita-class infrastructure for DITA-based editors, we advise against using it for configuration. This is because the class hierarchy usually produces unwanted results when used directly in our orthogonal configuration hierarchy and doing so may introduce a lot of complexity in the configuration as specific values frequently need to be overridden for specific sub-classes.

An example of this problem is found in specializing the list item element: not all specializations of the list item should be rendered or behave like list items:

[3] https://documentation.fontoxml.com/api/latest/fonto-dita-class-16324219.html

Take for example the `<consequence />` element in the Dita hazard domain. These elements should not be rendered like lists and they should not be indentable using the tab key, nor splittable using enter.

Because of these reasons, and because the dita inheritance structure does not give any pointers on how to create those elements, the configuration is most often denormalized to simply using node names.

3. Processing XML at interactive speeds

3.1. General XPath performance

The main performance bottleneck of Fonto is the performance of running XPath queries. XPaths is not only used to retrieve the schema experience configuration, but also to run generic queries. In order to speed up most queries, most of the optimizations described in the work of Kay[5] are implemented.

3.1.1. Outermost

Furthermore, a number of specific optimizations are implemented. One of the strongest optimizations regards the 'outermost' function, which returns the 'highest level' nodes from an input set. An example usage in Fonto is find and replace, which runs a query similar to following to determine the searchable text of an element:

```
descendant::node()[
  self::text() or
  (self::paragraph or self::footnote)
] => outermost()
```

This query returns all textnodes that are directly in a 'block', and any elements that are also a 'block'. Consider the following XML:

```
<xml>
  <paragraph>
    A piece of text
    <footnote>text is a string of characters</footnote>
    with a footnote in it
  </paragraph>
</xml>
```

When evaluating this query in a naive way, the path expression will result in a list of all descendants that match its filter, including all of the descendants that will be removed by the 'outermost' function.

A common optimization in functional languages like XPath is to perform lazy evaluation. We implemented this using a generator pattern inspired by LINQ[4]. However, lazy evaluation alone is not enough in this case. To further optimize outermost, we pass a hint into the generator for the descendant axis, indicating whether it should traverse the descendants of the previous node returned or skip to the next one.

Consider the expression above, which consists of three parts: a descendant part, a filter part and the outermost function. Using lazy evaluation, we start at the outermost function, which requests the first node from the expression that feeds it. To compute this, the filter expression requests nodes from the descendant expression until it finds one that matches the filter, which is returned to the outermost function. The outermost function is not interested in the descendants of this node, so it now passes the "skip descendants" hint when requesting the next node. This hint, passed through the filter expression to the descendant expression, prevents the latter from traversing the subtree of the matching node and instead skips to the following node.

As find and replace recursively applies the query for text nodes and sub-blocks, this optimization basically changes the performance of that from $O(n\log(n))$ complexity to $O(n)$, as every subtree is now only traversed once instead of for each ancestor.

3.2. Schema validity

Fonto checks the validity of XML to the schema by converting each content model into a nondeterministic finite automaton (NFA), similar to the approach described by Thompson & Tobin[7]. We perform several optimizations to ensure this validation can happen quickly enough to not seriously impact editor performance.

Before a schema is loaded in Fonto, it is pre-processed in an offline compilation step. This converts the schema to a JSON format and simplifies the content model expressions. We first remove any indirections such as substitution groups and redefinitions. We then apply a number of rewrite rules to reduce these content models to equivalent but simpler models. For example, if any item within an `<xs:choice>` is optional, the entire choice can be considered optional, and all items within can be marked as required (`minOccurs="1"`). If multiple items within the choice were optional, this reduces the number of empty transitions that have to be created in the resulting NFA.

For example, the schema structure:

```
<xs:choice minOccurs="1">
  <xs:element name="employee" type="employee" minOccurs="0"/>
  <xs:element name="member" type="member" minOccurs="0"/>
</xs:choice>
```

Is equivalent to:

[4] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

```
<xs:choice minOccurs="0">
  <xs:element name="employee" type="employee" minOccurs="1"/>
  <xs:element name="member" type="member" minOccurs="1"/>
</xs:choice>
```

When compiling the reduced schema to an NFA we apply a few optimizations over the Thompson & Tobin algorithm in order to further reduce the size of the resulting automaton. Firstly, all branches of a choice that each process a single node (such as the "employee" and "member" branches in the example) are represented as a single transition. Large choices between multiple single-element options are a fairly common occurrence in schemata we've seen used in Fonto. This optimization reduces the number of possible paths in the NFA, reducing the memory and execution time costs for computing possible paths during validation. In real-world schemata, such optimizations may be more significant. For example, the content model of paragraphs usually consists of a repeating choice between a number of inline elements.

Secondly, and again to reduce the size of the NFA, any repetition of a term T with `minOccurs="1"` and `maxOccurs="unbounded"` is compiled to the automaton for T followed by an additional empty transition back to the start. The original Thompson & Tobin algorithm would build an NFA containing the automaton for T twice (once required, once optional repeating).

Our implementation for applying the resulting NFAs to XML content makes heavy use of pre-allocated typed arrays to store all state during traversal. Being a garbage-collected language, manual memory management is not commonly considered in JavaScript applications. However, validation being a very hot code path, preventing allocations serves both to avoid the performance overhead associated with them, as well as the later cost of having garbage collection reclaim those allocations. Ignoring the schema and NFA optimizations, manual memory management alone has led to a significant performance improvement compared to our implementation before these changes: applying a test NFA similar[5] to `<xs:any minOccurs="0" maxOccurs="unbounded" />` to a sequence of 10000 children went from around 111ms to just 17ms.

3.3. Indices

Many operations in Fonto applications require traversing parts of the DOM using XPath queries. While most of these traversals are limited to a reasonably local subset of nodes, there are some types of queries that have to traverse large numbers of nodes. In our experience, these most commonly take one of two shapes. One is to find a specific element or set of elements based on the value of some of their attributes, for instance, finding the target of a reference based on its `xml:id`.

[5]This particular test also involves determining all possible minimal traces through the NFA. Fonto can use this information to synthesize[6] missing elements.

Another is to find all descendant nodes of a certain type, often under some ancestor node, for instance, finding all footnotes in the document.

To prevent the full DOM traversal in answering these queries, it can help to perform some of the work ahead of time. To this end, Fonto allows defining specialized indices, which are then made accessible to XPath queries as functions that return associated data given some key. Fonto currently has three types of index:

- The *attribute index* can be defined for any attribute name (local name and namespace URI), and maps a given value to the set of nodes that have the attribute set to that value.
- The *bucket index* can be defined for any bucket, as discussed in an earlier section, and tracks all nodes matching that bucket that are currently part of any loaded document
- The *descendant index* tracks the set of descendant nodes matching a given selector under a specified ancestor. To make updates efficient, this selector is currently severely limited in terms of the parts of the DOM it may refer to.

Internally, Fonto makes heavy use of mutation observers (as defined in the DOM standard) and the resulting mutation records to represent changes in any of the loaded documents. Indices interpret these mutation records to determine which changes affect their data, and then update that data accordingly only if such changes are found.

In our current implementation, all indices should be explicitly defined by the application developer. We have considered automatically generating indices, such as attribute indices for attributes using the xs:ID type, but found that many schemata do not actually assign this type for their identifier attributes.

3.3.1. Indexing arbitrary computations

In addition to these indices, mutation records can be used to invalidate the cached results of any DOM-based computation, including XPath evaluation[4]. This requires tracking that computation's data dependencies in terms similar to the relations described by the mutation records. While not an index in the traditional sense, the similarity in terms of implementation and integration with the indices described above have led us to refer to this system as the *callback index*.

Summarizing from our earlier work, we use a facade between the computation and all DOM access to intercept these events and track corresponding dependencies in terms of the corresponding mutation record type (either *childList*, *attributes* or *characterData*). When mutation records are processed, we match them against these dependencies and signal (potential) invalidation of a computed value when the data depended on has changed. To avoid unnecessary work, re-computation is not performed automatically, but only on demand. This usually happens when the UI using the result is ready to update, instead of updating these values many more times than could ever be observed by any user. It also avoids work in cases the UI decides not to re-issue the computation, for instance based on the result on another. For instance, the title of some figure in the document outline does not need to be recomputed if the entire section containing that figure is removed from the outline tree.

Both mutation records and raw DOM access operations can sometimes present a rather coarse-grained view of changes / dependencies. For instance, looking for a child element of a specific type may require visiting and examining all children of the parent node. This means that the corresponding computation may be invalidated unnecessarily if a node of a different type is inserted under the same parent. We use two mechanisms to reduce such unwanted invalidations.

First, dependencies registered by the DOM facade can specify a test callback in addition to the mutation record type. This test is evaluated against the changed document if a mutation record is processed with the matching type. If the test does not pass, the mutation record is ignored. We use this, for instance, to check whether a *childList* change affected the "parent node" relation for a given node.

Second, for most axis traversals in XPath we pass the *bucket* of the corresponding node test to the DOM facade. The resulting bucketed dependencies only invalidate the computation result for changes that match the bucket in question. For instance, the *childList* dependency for the selector `child::p` only triggers invalidation if a *childList* mutation record adds or removes `<p>` elements, not when only other nodes are added or removed.

3.3.2. Indexing and overlays

In Fonto, both the DOM and indices use a system of overlays to represent a future state of the DOM without actually mutating the original. For indices, these overlays are only initialized and then updated at the moments when the indices are actually used. As Fonto computes many possible future states at any time (for example, to determine the states of buttons in the toolbar), this avoids a significant amount of work for operations that do not use indices.

Furthermore, the lazy initialization of overlays allows computations based on the unmodified DOM to be re-used across different operations, as long as the value is computed before any modifications are made. In practice, this happens a lot. For example, tables use the callback index to derive a schema-neutral "grid model" representation from the DOM nodes. They then mutate this model, which in turn updates the schema-specific table DOM. As the entire table toolbar uses the same initial state of the DOM to compute the state of its operations, we only need to compute this grid model once. In fact, the same model has likely already been computed and cached in the callback index in order to validate the result of the previous operation, and is also used in rendering the table.

3.3.3. Fonto versus XML databases

In general, XML databases solve similar problems in terms of using indexing to make queries faster. However, the problem space differs in the following ways:

In Fonto, loading multiple megabytes of XML is a lot; we are on the web, so data needs to be small enough to download quickly and as a result will always fit in memory. In XML databases, gigabytes of XML is not rare and to be expected. In Fonto, authors on a bad internet connection don't want to wait ages for their documents to load, so larger documents are usually cut up into smaller chunks, which are loaded just in time for editing.

In Fonto, both queries and changes usually affect the same small part of the document. Furthermore, changes happen frequently. In an XML database, both changes and query subjects are often more spread-out. Because of this, our indexing approach needs to take frequent updates into account, and such updates need to happen quickly enough for users not to notice any slowdown.

In XML databases, it is acceptable to build indices during load time. In Fonto, the editor should be up and running as soon as possible. This means we can not build a large index at start-up if computing that index takes a non-trivial amount of time. Also, because Fonto usually does not run for a long time, it is probable that an index will never be used.

The current cache invalidation approach so far fits that set of requirements. A reusable result is often only computed when necessary, and forgotten when it no longer applies. A larger computation can be spread out over multiple separately indexed entries in order to make recomputation more efficient in cases where only part of these are invalidated.

4. Conclusions

Lots of tricks are possible to make user-friendly authoring of XML fast, even in JavaScript and webbrowsers. When Fonto was two years old, we received a lot of feedback on the performance of documents of 100KB of XML. Currently, we have clients working with single documents ranging into the megabytes, configured using complex schemata like JATS or the TEI standards. Using approaches like JIT loading and chunking, we have clients working with tens of thousands of documents which we are unable to even download and keep in memory simultaneously.

5. Future work

At Fonto we continue to move to declarative formats to specify the configuration, behavior and UI of the editor. We prefer to use existing standards, and continue to improve and extend our XPath, XQuery and XQUF implementations. For configuration, the closest analogue in terms of declarative formats seems to be CSS.

However, we prefer to keep using XPath for our selectors. We also have several property types that go far beyond the property values commonly found in CSS, including the way the appearance of elements is defined as a composition of visual components and widgets. It is likely we will need to develop a custom format to support this combination.

In mutating the XML DOM, moving to XQUF has the additional advantage that we can use the callback index to track the dependencies of an operation, and therefore only recompute its effect (represented as a pending update list) and state (based on the validity of the resulting DOM) when required. In addition to converting current JavaScript-based primitives, this requires allowing other bits of state to be dependency-trackable in the same way as the DOM, including the current selection.

To minimize work even further with minimal impact on the way developers configure Fonto, we wish to further expand indices and the callback index into a framework for general incremental computation. This requires dependencies between index entries (already partially implemented), which allow for memoization by isolating one computation from another. To propagate invalidations caused by DOM changes efficiently, we also need to add a way to stop this propagation when the new result for some computation equals the previous value, as that means results depending on that value can be reused.

Bibliography

[1] XQuery update facility 3.0 `https://www.w3.org/TR/xquery-update-30/`

[2] A minimalistic XPath 3.1 implementation in pure JavaScript `https://github.com/FontoXML/fontoxpath`

[3] `https://drafts.csswg.org/selectors-4/#specificity-rules`

[4] Martin Middel. Soft validation in an editor environment. 2017. `http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf`

[5] Michael Kay. XSLT and XPath Optimization `http://www.saxonica.com/papers/xslt_xpath.pdf`

[6] Martin Middel. How to configure an editor. 2019. `https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf`

[7] Thompson, Henry S., and Richard Tobin. "Using finite state automata to implement W3C XML schema content model validation and restriction checking." *Proceedings of XML Europe*. Vol. 2003. 2003.

[8] Various authors. The DOM Living Standard. Last updated 16 January 2020. `https://dom.spec.whatwg.org/`