Author-prepared version. Limited to noncommercial access and personal use.

How to Edit Gigabyte XML Files on a Mobile Phone with XAS, RefTrees, and RAXS

Tancred Lindholm
Helsinki Institute for Information Technology
P.O. Box 9800
FIN-02015 TKK, Finland
tancred.lindholm@hiit.fi

Jaakko Kangasharju
Helsinki University of Technology
P.O. Box 5400
FIN-02015 TKK, Finland
jkangash@cc.hut.fi

ABSTRACT

The Open Source mobility middleware developed in the Fuego Core project provides a stack for efficient XML processing on limited devices. Its components are a persistent map API, advanced XML serialization and out-of-order parsing with byte-level access (*XAS*), data structures and algorithms for lazy manipulation and random access to XML trees (*RefTree*), and a component for XML document management (*RAXS*) such as packaging, versioning, and synchronization. The components provide a toolbox of simple and lightweight XML processing techniques rather than a complete XML database. We demonstrate the Fuego XML stack by building a viewer and multiversion editor capable of processing gigabyte-sized Wikipedia XML files on a mobile phone. We present performance measurements obtained on the phone, and a comparison to implementations based on existing technologies. These show that the Fuego XML stack allows going beyond what is commonly considered feasible on limited devices in terms of XML processing, and that it provides advantages in terms of decreased set-up time and storage space requirements compared to existing approaches.

Categories and Subject Descriptors

I.7.1 [**Document and Text Processing**]: Document Management; E.1 [**Data Structures**]: Trees

General Terms

Design, Experimentation, Performance

Keywords

XML, mobile, lazy, tree, parsing, serialization

1. INTRODUCTION

Interoperability and openness are fundamental motivators behind the Extensible Markup Language (XML) [34]. Application development and deployment becomes easier as existing software components for processing XML may readily be used. When data is stored in an open XML format it becomes accessible to other applications, which may then provide functionality beyond what was originally intended.

The number of mobile phones and other limited devices on the Internet capable of hosting generic applications is rapidly rising, and may soon outgrow the number of regular computers [15]. In terms of software development, these devices are challenging due to their significantly lower processing capabilities compared to desktop systems. Thinking that faster processors will make this go away is a fallacy: as these devices are battery-powered, saving CPU cycles translates into longer operation on a single charge, which is often a key selling point. Thus, in this domain, saving processing cycles is of paramount importance.

Storage on limited devices, on the other hand, is another business, as it typically does not consume any power when idle. During recent years, we have seen rapid growth in storage capacity on limited devices. This has led to an increasing mismatch between storage and processing power. For instance, the mobile phone we use can accommodate a 1 GB XML file. However, it will take the phone some 9 hours just to parse that file. This is around 270 times slower than the same task on our current desktop PC.

In the Fuego Core project series (2002–2007) at the Helsinki Institute for Information Technology[1] we are developing a set of middleware services based on the ideas of interoperability, openness, and computational simplicity. This *Fuego mobility middleware* [28] uses XML as its primary format for data storage and manipulation. The middleware is available under an Open Source license on the Web.[2]

In this paper we present the *Fuego XML stack*, which is the collective name for the XML processing components in the Fuego middleware. The stack provides efficient read and write random access to ordinary XML files, small and large alike. The distinguishing features of the stack are: i) the ability to process significantly larger XML files on limited devices than what is commonly considered feasible, and ii) extensive use of verbatim XML for on-disk storage. Our stack enables application developers to use large XML data files on mobile devices, without expensive format conversions, and in a very interoperable and open manner.

The components of the stack are the XAS API for efficient XML parsing and serialization, the RefTree API for lazy XML tree manipulation, and the Random Access XML Store (RAXS) API for XML document management such as packaging, versioning, and synchronization. The efficiency

[1] http://www.hiit.fi

[2] http://hoslab.cs.helsinki.fi/homepages/fuego-core/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiQuitous 2008, July 21 - 25, 2008, Dublin, Ireland.
Copyright © 2008 ICST ISBN 978-963-9799-27-1.

of our stack stems from the use of lazy data structures and the advanced parsing and serialization capabilities of XAS.

When storing data on disk, we use the original XML in its verbatim format as far as possible. That is, we avoid having to import and export XML documents to and from a store format, and instead use the XML file directly. This way, our store is open to XML processing applications, and we can avoid costly and sometimes error-prone format conversion.

We demonstrate the use of the stack by building a Wikipedia viewer and editor, which we are currently able to run successfully on a real mobile phone with an input file whose size is 1 GB. We also wrote two additional parallel implementations of the viewer, where one stores data in application-specific files, and the other uses an SQL database. We present measurements on the viewer and the implementations, based on which we provide a performance analysis, and a comparison showing the strengths and weaknesses of our software stack versus other prevalent storage solutions.

Our stack is not an XML database, but a database could potentially be built on top of it. However, Stonebraker et al. [24,25] observed that an application-specific data storage may be significantly faster than a generic one. Since we think that this observation is relevant for limited devices as well, our stack opts to provide a toolbox of APIs for XML access rather than a complete database.

We start by describing the components of the Fuego XML stack in Section 2, and then move on to the Wikipedia viewer in Section 3, where we present background on Wikipedia, and describe how the APIs of the XML stack are used to implement the editor. The experimental results and their analysis are in Section 4. We review related work, mainly in the areas of lazy XML and XML versioning, in Section 5. This is followed by concluding remarks in Section 6.

2. THE FUEGO XML STACK

Consider such applications as an address book, a collection of media files with metadata, a rich text editor, or a high-score list for a game. All of these need to store some data persistently. They read it in a linear or random-access fashion, potentially using some narrowly-defined search criteria (e.g., the data for a specific entry in the address book, the metadata for a particular media file). Changes are typically made to a limited subset of the data, often one record at a time. Data display, too, may be limited to a subset of the data, e.g., one screen of the text document, or the topmost entries for the high-score list.

Using XML as the storage format is a popular choice that provides significant interoperability out-of-the-box, a desirable feature that is as useful on the limited device as it is on the desktop. Desktop computers typically have enough resources to parse the XML data into memory, where random access and updates are performed, after which it is written back as a whole. However, on limited devices we frequently find that the available processing and memory resources are insufficient for such an approach. The Fuego XML stack addresses this by providing random-access read and write functionality to verbatim XML files, without requiring the files to be fully parsed, nor fully loaded into memory. The maximum size of a manageable XML document becomes mostly dependent on the subset of the data the application needs to access, rather than the full document.

An overview of the components of the stack is shown in Figure 1. On the bottom we have the fundaments: XML parsing and serialization and a persistent map for opaque binary objects (BLOBs). XML parsing and serialization is handled by the XAS module, which distinguishes itself by allowing XML documents to be parsed out-of-order, and by giving access to the raw byte input and output streams of the XML parser and serializer.

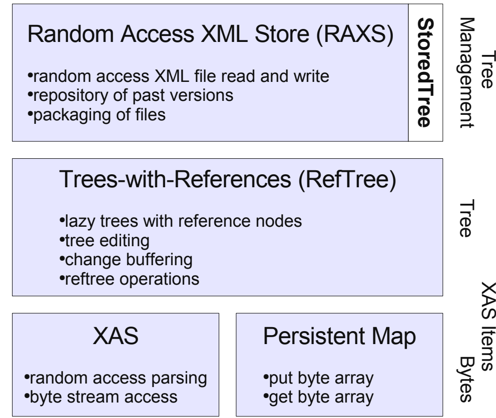

Figure 1: Overview of the Fuego XML Stack

XML	XAS Item	
<p>	ST(p)	ST(x) = Start Tag x
<i>	ST(i)	
Hi	T(Hi)	T(x) = Text x
</i>	ET(i)	ET(x) = End Tag x
	ST(b)	
LyX	T(LyX)	
	ET(b)	
</p>	ET(p)	

Figure 2: XML document and XAS items.

On top of these we have the trees-with-references (RefTree) API for manipulating XML documents using a tree model. RefTree provides a fashion of lazy trees, which allows loading XML documents partially into memory. The API features efficient in-memory buffering of tree changes, and algorithms for manipulating the lazy aspects of trees.

At the highest layer we have the Random Access XML Store (RAXS), which provides overall XML document management operations, such as change transactions, versioning, and synchronization. RAXS functions as a container for an XML document along with any auxiliary files and indices.

2.1 XAS

XAS is designed to be a general-purpose XML processing system that includes sufficient extensibility to permit clean and efficient implementation of many different XML processing tasks. We focus here on the parts of XAS that are relevant to the RefTree and RAXS implementations, but the full system contains much more functionality.

The basic concept in XAS is the *item* that represents an atomic piece of XML data, and roughly corresponds to an event in the SAX API [4]. An XML document is then viewed as a sequence of items. XAS provides both a streaming model for parsing and serializing XML and an in-memory item list model for XML processing applications. An example XML document and the corresponding XAS items are shown in Figure 2.

The item stream and list models of XAS are familiar from other APIs, but XAS also includes lower-level access to the actual bytes comprising the XML document. The main users of this functionality are expected to be applications that transform XML, so this byte stream access is provided also when serializing an XML document. By reading and writing bytes directly, incoming subtrees need to be parsed only if they need to be changed.

While byte stream access is extremely beneficial in some cases, it also places a burden on applications using it. Since the application essentially takes the place of the XML processor, it must also take care that, after the byte stream processing is complete, the application is in the same subtree and at the same level as it was prior to the byte processing. This ensures that the state of the XML processor remains valid, so it can continue its processing.

The concept of the processor's state is formalized in XAS by the notion of *processing context*. This is a stack maintained by the XAS processor, which consists of start tags of elements that are ancestors of the current node being processed. Thus, an application using byte stream access needs to finish the access with the same processing context as it was in the beginning.

The formalization of the processing context gives rise to another technique, that of the *seekable parser* which to our knowledge is unique to XAS. When the XML document being parsed comes from a random-access storage such as a file, a seekable parser can be constructed on top of the input. Such a parser allows repositioning of the underlying input to an application-specified location. To ensure correct parser functionality, the application must also set the processing context correctly for the repositioning.

The repositioning offset and processing context at that offset combine to form the *parsing state* of the position. The seekable parser interface contains operations to access and set the current parsing state. Usually, when setting the parsing state, the new state will be one previously accessed, but since the offsets are simply bytes, correct parsing states can also be computed. However, the application needs to take care that the processing context is suitable for the offset.

A seekable parser makes it possible to parse XML documents out of order or lazily. This will require the application to pre-build an index for the parsing states of the relevant subtrees, usually including the byte lengths of the subtrees as well. Then, when reading the document the next time, if the application does not wish to parse a subtree, it can use the indexed parsing state to seek after the skipped subtree. To parse lazily, the application can simply insert a placeholder item containing the indexed parsing state so that the subtree can be later parsed correctly if needed.

2.2 Tree-with-References (RefTree)

A basic strategy for dealing with data structures that do not fit in main memory is to load into memory only those parts of the structure that are needed for the task at hand. Our rendition of this strategy is the tree-with-references, abbreviated *reftree*, data model with accompanying methods to manipulate such trees.

A reftree is an ordered tree with two types of nodes: *ordinary* and *reference* nodes. A reference node acts as a placeholder for a single node or, more importantly, an entire subtree. In cases when processing does not require inspection of a subtree, we may replace that subtree with a reference node, and manipulate the reference node rather than the subtree, yielding savings in memory and processing time. Reference nodes can be thought of as placeholders for subtrees whose evaluation has been delayed.

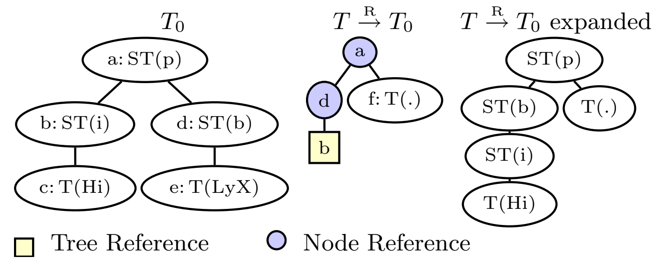

Figure 3: RefTree holding XAS Items and the corresponding XML.

A reftree T along with the reftree T_B its reference nodes point to is denoted $T \xrightarrow{R} T_B$, with T being the *referencing* tree and T_B the *referenced* tree. We do not allow a reftree to reference nodes from more than one tree.

Every tree node has a *key*, by which it is uniquely identified inside the reftree, and an application-specific *content* object. The ability to identify and address nodes by their unique key is an important part of the reftree tree model, as will be evident further on.

Reference nodes have a *target* as their content. The target holds the key of the node being referenced. There are two variants of reference nodes: *tree* references and *node* references. The former references the *subtree* rooted at its target, and the latter the *target node only*.

For a tree reference, the descendants of the target node in the referenced tree are said to be *virtual* in the referencing tree. Thus, there are in total four *kinds* of nodes: ordinary, node reference, tree reference, and virtual.

In our XML stack we use reftrees for in-memory storage of XML. The reftree structure mirrors the structure of the XML document, and the XAS items from the document are stored in the tree as content of ordinary nodes. As an example, consider the tree T_0 in Figure 3, which holds the XAS items of the XML document in Figure 2. Note that we have assigned a key to each node (a, b, c, ...) and that end tags are not stored in the tree structure, as these can be inferred. $T \xrightarrow{R} T_0$ is a reftree that references nodes in T_0: a and d are referenced using node references, and b using a tree reference. In $T \xrightarrow{R} T_0$, the node f is ordinary, whereas a and d are node references, b is a tree reference, and c is a virtual node. Rightmost is $T \xrightarrow{R} T_0$ with all reference nodes expanded.

If two reftrees $T \xrightarrow{R} T_B$ and $T' \xrightarrow{R} T_B$ become identical when the reference nodes in both trees have been replaced with the nodes they reference, we say that T and T' are *reference equal*. We denote this $T \stackrel{R}{=} T'$.

A reftree may be interpreted as the *cumulative state change* to the referenced tree induced by some edits. More precisely, assume we apply edit operations to a tree T to get T'. We may now use reftrees to express T' using references to T, i.e., express the state change as $T'' \xrightarrow{R} T$ so that $(T'' \xrightarrow{R} T) \stackrel{R}{=} T'$. When we want to emphasize that a reftree expresses such a state change we use the term *change tree*. For instance, we can interpret $T \xrightarrow{R} T_0$ in Figure 3 as a change tree that edits T_0 by moving d below a, deleting e, and inserting f.

Reference nodes are not auto-evaluating, i.e., they do not appear to be, or get automatically replaced with, the nodes they reference. This is to avoid costly computation that would potentially happen "behind the back" of the application developer. Instead, reference nodes are explicitly manipulated. For this the RefTree API provides a set of operations for common reftree manipulation tasks.

2.2.1 RefTree Operations

In the *reference expansion* operation a selected set of reference nodes are replaced with the nodes they reference. The *application* operation is used to combine two "chained" reftrees of the form $T \overset{\text{R}}{\rightarrow} (T' \overset{\text{R}}{\rightarrow} T_0)$ into a single reftree $T \overset{\text{R}}{\rightarrow} T_0$. In particular, using repeated application, we may combine a list of reftrees of the form $T_n \overset{\text{R}}{\rightarrow} T_{n-1} \overset{\text{R}}{\rightarrow} \ldots \overset{\text{R}}{\rightarrow} T_0$ into $T_n \overset{\text{R}}{\rightarrow} T_0$. The intuitive meaning of application is to combine several change trees into one change tree that expresses the cumulative change.

In the *reference reversal* operation, the roles of the referencing and the referenced trees are reversed, i.e., we compute $T'_B \overset{\text{R}}{\rightarrow} T$ from $T \overset{\text{R}}{\rightarrow} T_B$ so that $T'_B \overset{\text{R}}{=} T_B$. When thinking of a reftree as a change tree from state a to b, reference reversal produces the change tree from state b to a. We note that reference reversal is a special case of a more general operation, *reference normalization* [18].

Note that we want the algorithms to yield results that use reference nodes whenever possible in order to ensure processing and memory efficiency (both application and reference reversal have trivial solutions where all nodes are expanded). Our reftrees implementation produces results that are efficiently computable and honor this requirement.

From an implementation point of view, we found that it makes sense to categorize reftrees depending on the ability to traverse and change the tree. First, there are the basic immutable reftrees, or just reftrees, which may be traversed along the parent and child axes. The *addressable* reftree adds the ability to look up nodes by their key in a random-access fashion. Finally, a *mutable* reftree is an addressable reftree that can be changed by subtree insert, delete, update, and move operations.

The *change buffer* is a special mutable reftree that wraps around an addressable reftree T_a. Initially the change buffer appears identical to T_a. Any subsequent change operations modify the buffer as any other mutable reftree. However, behind the scenes, the change buffer maintains a mutable reftree $T_c \overset{\text{R}}{\rightarrow} T_a$, which is a change tree that expresses the cumulative change from T_a. To maintain the change tree, we expand the nodes involved in each operation in the change tree, and then execute the operation on the expanded tree. The change buffer can be queried for the change tree at any point with the getChangeTree() method.

The change buffer thus computes from a list of edit operations the cumulative state change as a change tree. The RefTree API also includes an algorithm for the reverse, i.e., to construct a list of edit operations from a change tree. Using this algorithm, a mutable reftree T_m may be edited into the state indicated by some change tree $T \overset{\text{R}}{\rightarrow} T_m$.

The API allows casting an immutable reftree to an addressable reftree by the means of a wrapping tree that maintains an index of keys and nodes. An addressable reftree, in turn, may be cast to a mutable reftree by wrapping it with a change buffer. This functionality along with the aforementioned categories let the application developer implement the minimum level of functionality that is required to accomplish a task in a sufficiently efficient manner. For instance, for a small tree, only the basic reftree needs to be implemented, as the API methods for casting to more able types are typically efficient enough.

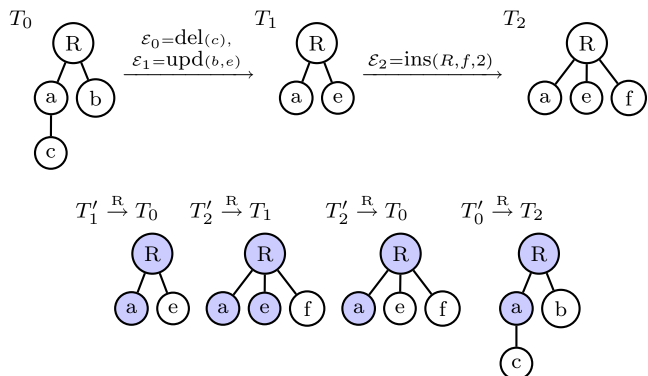

Figure 4: Different reftrees.

A way to conserve memory is to instantiate individual nodes on-demand, rather than the full tree structure, and discard nodes from memory immediately when no longer needed. We call nodes that are used in this manner *transient*. When implementing transient nodes, it is easy to accidentally waste memory if one uses object pointers between nodes, as the pointers may prevent the node from being freed (i.e., garbage collected) by making it reachable through some other node that is in use. However, as nodes have unique keys in our API, this problem can be avoided by using these instead of object pointers to link nodes.

In Figure 4, we see a tree T_0 that is first edited into a tree T_1, which in turn is modified into T_2. The edit of T_0 into T_1 expressed as a change tree is $T'_1 \overset{\text{R}}{\rightarrow} T_0$, and $T'_2 \overset{\text{R}}{\rightarrow} T_1$ expresses the edit of T_1 into T_2 using references to nodes in T_1. To obtain the cumulative change between T_0 and T_2 we can combine these two reftrees using the application operation, yielding $T'_2 \overset{\text{R}}{\rightarrow} T_0$. Note that since neither e or f appears in T_0, we cannot use references to these in $T'_2 \overset{\text{R}}{\rightarrow} T_0$.

Suppose T_0 is an immutable tree, but we nonetheless want to edit it as shown. We can then wrap T_0 with a change buffer, which will allow us to perform the required edits by buffering them into memory as a change tree. In particular, if we query the change buffer for the current change tree after application of the edits $\mathcal{E}_0, \mathcal{E}_1$, we get $T'_1 \overset{\text{R}}{\rightarrow} T_0$, and after $\mathcal{E}_2$ we would get $T'_2 \overset{\text{R}}{\rightarrow} T_0$. Finally, we may want to swap the roles of T_0 and T_2 so that we express T_0 using references to T_2. This is done by reference reversal of $T'_2 \overset{\text{R}}{\rightarrow} T_0$, and yields the tree $T'_0 \overset{\text{R}}{\rightarrow} T_2$ shown rightmost. By expanding the references in $T'_0 \overset{\text{R}}{\rightarrow} T_2$ with nodes from T_2, we see that $T'_0 \overset{\text{R}}{\rightarrow} T_2$ and T_0 are indeed identical, i.e., $T'_0 \overset{\text{R}}{\rightarrow} T_2 \overset{\text{R}}{=} T_0$.

While space does not permit a detailed description of how the reftree operations are implemented, we can report that implementations that scale as $O(n \log n)$ or better are possible, where n is roughly the combined sized of the input and output trees, and excluding virtual nodes. This is based on our current implementation of the operations, which in each case performs an amount of work bounded by $O(\log n)$ for each input and output node.

2.3 Indexing with Persistent Maps

We provide a *persistent map* for maintaining indexes. The map is one-way, and between arbitrary keys and values, and it is stored as binary files that grow dynamically with the size of the map. We currently use an implementation of the dynamic hash algorithm in [16] provided by the Solinger Sdbm project[3], mainly because it is simple and lightweight.

The precise format of the keys used to identify nodes is not dictated by the RefTree API, but in practice we have found it useful to use Dewey keys [30], i.e., keys consisting of a path of integers from the root where the key $i_0 i_1 \ldots i_k$ selects the i_k^{th} child node of the node selected by the key $i_0 i_1 \ldots i_{k-1}$. The benefits of these are mainly that they can denote any node in the XML document and that determining structural relationships between two Dewey keys is straightforward.

In the common case where we key XML data in a reftree by Dewey keys, a persistent map of Dewey keys to XAS parsing states turns out to be a very useful index. The structural properties of Dewey keys allow parent and sibling keys to be inferred, so such a map suffices for performing the lookup and tree traversals needed by an addressable reftree.

A useful key-related construct is the *key map*, which is a bi-directional map on node keys. Key maps may be used to, e.g., control which nodes align during reftree operations, or to re-map the structure of a tree. A particularly useful key map implementation is our Dewey-keyed mutable reftree, which is able to map between the Dewey keys of the referenced and referencing trees.

2.4 The Random Access XML Store (RAXS)

At the top level of the Fuego XML stack, we have the Random Access XML Store (RAXS). RAXS ties together data files, a reftree model, and tree mutability. The API allows opening an XML file as a reftree, editing that tree, and writing the changes back to the file in a transactional manner. A RAXS can be thought of as a limited XML "database", which is stored as a main XML file with an accompanying set of files storing indexes and related binary objects.

To the application developer a RAXS instance exposes its data as an addressable reftree T. This tree is obtained with the getTree() method, which can also be used to obtain previous versions of T by passing a version number as argument. A store is edited by calling the getEditableTree() method, which returns the store contents as a mutable reftree, and also flags the start of an editing transaction. The mutable reftree is generated by constructing a change buffer T' on top of T. Thus, any changes made to the store are buffered as a change tree $T' \xrightarrow{\text{R}} T$ in memory.

The edited data is committed by calling the commit() method with T' as an argument, which also ends the edit transaction and returns the mutable tree to the store. If the changes should be discarded instead of committed, the change buffer is simply reset.

Setting versioning aside for a moment, the main task of commit() is to update T to T'. One way to do this is to rewrite T, and another is to use a *forward delta* from T to T'. In both cases we use the change tree $T' \xrightarrow{\text{R}} T$ maintained by the change buffer.

In the rewrite approach we traverse $T' \xrightarrow{\text{R}} T$, emitting nodes to the new tree as we go along. Given an appropriate index to the XML text for T we may serialize any tree reference nodes by using the XAS byte streaming abilities to copy bytes from the original T to the new tree data file. That is, unchanged subtrees of T will be written to the new file by performing byte stream copy, rather than using slower XML parsing and serialization.

The forward delta approach can be used when it is infeasible to rewrite T, e.g., if it is too large even for stream copying, or it is on read-only media. In this case $T' \xrightarrow{\text{R}} T$ is simply stored in an auxiliary file, which will then effectively contain a forward delta from the original T to T'. When we want to recall T' we wrap T from the original file with a change buffer, and apply the reftree in the auxiliary file to the change buffer, hence yielding T'. A feature of this approach is that all changes may be flushed by removing the auxiliary file. A drawback of the forward delta approach is a memory overhead corresponding to the size of the change tree expressing the cumulative edits from the original tree.

On each commit we assign a version number to the store. For the purpose of recalling past versions we store *reverse deltas*, which allow recalling version $v-1$ from version v. As the delta we use a reftree that expresses T using references to T', and store it on disk. Since we know $T' \xrightarrow{\text{R}} T$, we can get $T \xrightarrow{\text{R}} T'$ using reference reversal. We note that the ability to recall past versions is useful when synchronizing data [19].

Each RAXS instance is connected to an addressable reftree that provides the tree content through an interface called StoredTree. In addition to providing the store with a tree, interface implementors also provide the tree update method. We have found that this interface nicely separates the concerns of RAXS from those of the storage layer.

3. VIEWING AND EDITING WIKIPEDIA

In recent years, the simple style of text markup used in Wikis [17] has become popular for collaborative Web authoring. The basic idea is to use visually non-intrusive markup on plain text to facilitate easy editing. For instance, surrounding a word with two single quotes (''word'') may produce emphasis in the formatted view, and prefixing lines with a hash sign (#) may produce an enumerated list. While different Wikis typically use slightly differing syntax, this basic idea is common to all.

We use the term *wikitext* for text marked up according to some Wiki syntax. One of the most well known corpora of wikitext is the English-language Wikipedia[4], an online encyclopedia to which anyone with a Web browser may contribute. XML dumps of the Wikipedia corpus are publicly available on the Wikipedia web site.

To put our architecture to test, we elected to implement an offline viewer and editor for Wikipedia based on the available XML dumps, as we think it would make for a realistic use case. We did not use the complete English Wikipedia (some 8.5 GB[5] at the time of writing), but rather a file consisting of the first 1 GB of the XML dump dated March 3, 2006. This corpus is also known as the Hutter Prize enwik9 data set[6]. It has 243 419 articles encompassing a wide range of sizes: the 10th, 50th, 99th, and 99.9^{th} percentiles are 26, 1 685, 35 128 and 68 274 bytes of wikitext, respectively.

[3] http://sourceforge.net/projects/solinger/

[4] http://en.wikipedia.org

[5] We follow the convention to use base 10 prefixes for files, and base 2 for RAM. Hence, 1 GB=10^9 bytes for files.

[6] http://prize.hutter1.net/

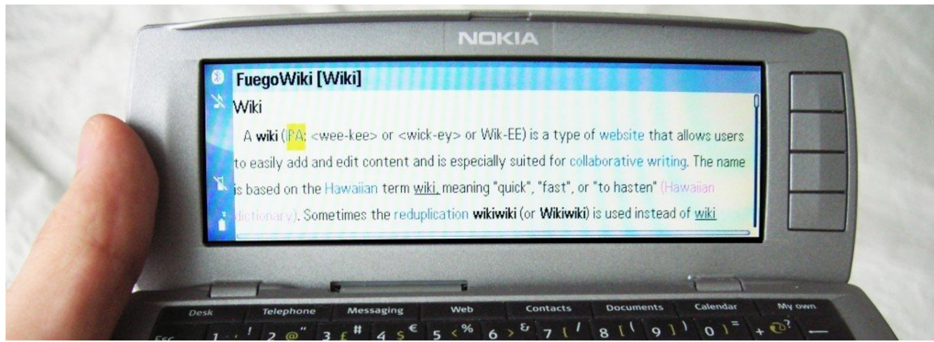

Figure 5: The editor on a Nokia 9500 showing the article on the word "Wiki"

The individual articles in a Wikipedia XML dump are contained in child elements of the document root. Each article has a title along with some other metadata in its child elements, and the actual article text inside a <text> element. The article text is wikitext rather than marked up as XML, i.e., from an XML point of view, it is a single text node. While the use of wikitext is regrettable from a pure XML perspective, it is a good example of the kind of practical XML usage that our architecture should accommodate.

In the following, we describe the Wikipedia viewer implementation, along with central code snippets that show how the Fuego XML API is put to use.

3.1 The Wikipedia Editor

For running the viewer we used the Nokia 9500 Communicator, which is a smartphone for the GSM cellular network. The Nokia 9500 ships with a Java implementation supporting the Java Personal Profile (PP) specification [26]. It also supports the more common Mobile Information Device Profile (MIDP) [27], but that is insufficient for our purposes, as it lacks the necessary APIs for random access to files. Our unit supports up to 1 GB of solid state storage on industry standard reduced-size multimedia card (RS-MMC) memory. The viewer uses RAXS to access the articles that are in the enwik9 file, which is stored on the MMC memory card on the phone. The Java heap was limited to 4 MB.

Figure 5 shows the Wikipedia editor running on the 9500 and showing the article on the word "Wiki". Hyperlinks between Wiki articles are shown in blue (e.g., IPA in the Figure), and may be navigated with the arrow keys and followed by pressing Enter. The currently selected link (IPA) is shown on yellow background. Articles may be looked up by entering a partial article name, after which the user may pick one of the 10 alphabetically closest titles.

To edit the current article, the user enters edit mode. In this mode, the user is able to view and change the article wikitext in a plaintext editor widget. Figure 6 shows the wikitext editor view along with the corresponding formatted view for a sample article.

3.2 Implementation

The starting point for the implementation is the enwik9 XML dump file. We designed our editor to use the XML dump verbatim without any extra data transformation stages in between. To manage the file we implement a Wikipedia RAXS, to which we provide the XML file as an addressable reftree through the StoredTree interface. Thus, the first

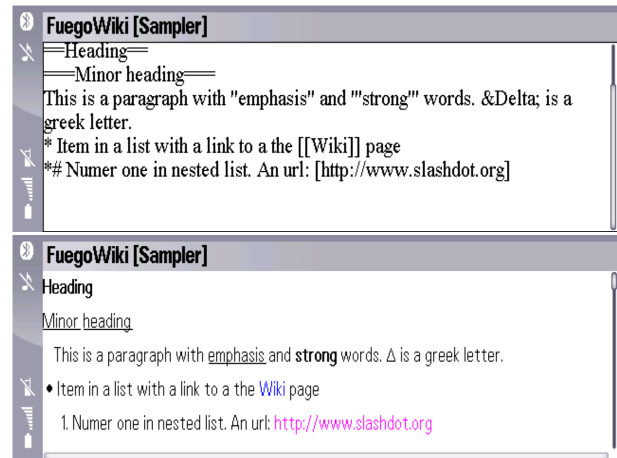

Figure 6: Editor and formatted views for an article.

step is to implement an addressable reftree T_W holding the content of enwik9, with RAXS then providing editing and versioning on top of this tree. The nodes of T_W should be transient due to the size of the file.

We construct T_W so that it mirrors the structure of enwik9, uses Dewey keys, and stores the corresponding XAS items at its nodes. We generate the nodes of T_W on-the-fly using XAS's seekable XML parser. When a node is requested, we seek to the corresponding position in enwik9, parse the XAS item at that position, and finally wrap the item along with its Dewey key as a reftree node.

```
class WikiTree implements AddressableRefTree, StoredTree {
  RefTreeNode getNode(DeweyKey k) {
    ParserState p = lookup(k);
    parser.setState(p);
    return new RefTreeNode(k,parser.nextItem());
```

Since nodes are typically requested in an out-of-order fashion, we need to be able to look up the position and length in enwik9 corresponding to a given Dewey key. For this purpose we build an index I_K, which is a map of Dewey keys to the corresponding parsing states in enwik9. Thus, the node lookup in the code snippet above is implemented as

```
class WikiTree { ...
  ParserState lookup(DeweyKey k) {
     I_K.lookup(k);
```

Due to the size of enwik9 the index will not fit into working memory. Instead we build it in advance, and store it in the persistent map provided by our API. Indexing is typically done on faster hardware, or as a background job on the target device prior to use. This is since to index an XML document we need to parse through it – and just parsing 1 GB of XML takes some 9 hours on the 9500.

To be able to look up an article by its title we use an additional persistent map I_T that maps an integer key to a $\langle \text{title}, \text{Dewey key} \rangle$ pair, so that the key of the pair is the position of the title in the sorted list of titles. Each index requires around 16 MB of additional storage on disk.

3.2.1 Implementing Editability

To store the changes persistently we need to augment T_W with a tree update method that gets called by the RAXS instance. Recalling different alternatives from Section 2.4, we find that in this case a forward delta from the unmodified enwik9 file is more appropriate, as rewriting the file on the target device is both slow and strenuous on storage capacity. Thus, when the user saves his edits, we simply store onto the file system the change tree that RAXS maintains and passes on commit:

```
class WikiTree { ...
  updateTree(ChangeBuffer editTree) {
    RefTree changes = editTree.getChangeTree()
    serialize(changes,"changes.xml")
```

There is, however, a scalability issue with this approach and the way the enwik9 data set is structured. Consider what the change tree for T_W will look like when a single article has been edited. We cannot use a tree reference for the entire tree (as it has changed), which means that we will have to use a node reference for the root, and tree references for each of the unchanged articles in its child list. The problem is that there are 240 000 articles, which implies at least that many nodes in the change tree. We cannot afford to store that many nodes in memory on the 9500.

A solution in the spirit of the reftree model is to introduce artificial tree levels in the structure of T_W compared to enwik9, where the nodes on these levels represent increasingly larger groups of articles. We may thus reference a group of nodes, rather than individual nodes, which reduces the size of the change tree significantly. This can be done by re-mapping the keys (and structure) of T_W using an appropriate key map m: T_W `= new MappedRefTree(` T_W`,` m`)`

3.2.2 Streaming Wikitext

Some articles weigh in at tens of kilobytes of wikitext and above, and expand to roughly twice the size in working memory if we use Java strings. As the editor needs to load into memory the forward delta, which holds the full wikitext of each changed article, we would like to minimize the memory footprint of nodes with wikitext. In addition, the XAS text items holding the article wikitexts are rarely accessed (except when the particular article is viewed). Thus, nodes holding wikitext are good targets for optimization.

To save memory we make these text items lazy, i.e., they do not load their content from the XML file unless accessed. To further improve performance in cases when the content is read or copied to another file, we use the byte stream capabilities of XAS to access the content as a raw byte stream. This allows us to bypass the XML parser when no parsing is needed (as in the case of wikitext), and to eliminate unnecessary text buffering.

Laziness is implemented with a special StreamedText XAS item. This item only stores the parser state at the text item, which allows us to load the actual text when needed. When the item is accessed, it can transparently load its text, or alternatively, provide an input stream to the text. Using streamed text, the serialize() method looks as follows:

```
class WikiTree { ...
 serialize(RefTreeNode n, XasSerializer s) {
   if ( n.content instanceof StreamedText )
    copyStream(n.content.inputStream, s.outputStream )
   else normalOutput(n.content, s)
   for ( child: n.children ) serialize(child, s)
```

Our wikitext formatter is StreamedText-aware, and constructs the display primitives directly from the input stream, without any intermediary buffers. Using streamed wikitext in the formatter and in forward deltas enables viewing wikitexts larger than the available working memory. Thus, our approach scales both on the level of XML document nodes, and on the level of the content size of individual nodes.

4. MEASUREMENTS

To validate our approach, we performed two experiments. In the first, we measured memory usage and the responsiveness perceived by the end user during use of the application, and in the second we compared versions of the viewer that used storage back-ends based on alternate approaches to the approach based on the Fuego XML stack.

4.1 Responsiveness and Memory Usage

To test the responsiveness of the editor we measured the time elapsed for 4 basic tasks on articles of varying size:

Lookup The article was looked up using the Find functionality by typing the article name and executing a lookup. We measured the time from starting lookup until a list of the 10 closest matching articles was ready.

Hyperlink A Wiki hyperlink from a dummy article to the test article was followed. We measured the time from activating the hyperlink until the text of the article was visible on screen.

Edit Edit mode was entered from article view mode. We measured the time from edit activation to having loaded the wikitext into the text edit widget.

Save The article was saved in edit mode. We measured the time from executing save until the updated article text was visible on screen. The save test was executed in two variants: Save 1 and Save n, where 1 denotes the first save of the article, and n a later save. We expect Save 1 to be slower than n due to reference node expansion in the change buffer at the first save (on subsequent saves, all nodes are already expanded).

The tasks were run on 4 articles, whose wikitexts serialized as XML were 1 kB, 4 kB, 16 kB, and approximately 64 kB.

The results are shown in Table 1, which lists the median time over 5 repetitions (4 for Save 1 due to a faulty test run), along with the maximum deviation from the median in parentheses. The times for the lookup and hyperlink tests

Table 1: Responsiveness by task and document

Time[ms]	1k	4k	16k	64k
Lookup	203 (15)	203 (15)	203 (16)	235 (5)
Link	1 938 (156)	2 156 (32)	2 047 (109)	1 984 (155)
Edit	250 (47)	438 (1)	1 406 (109)	4 734 (109)
Save 1	11 553 (244)	11 438 (188)	12 008 (40)	12 844 (235)
Save n	5 703 (79)	5 828 (141)	6 844 (110)	9 953 (688)

Table 2: Editor RAM usage and size of delta files

Space [kB]	1k	4k	16k	64k
Max memory	473	491	565	688
Size of delta	4.3	7.2	19.4	62.1

are satisfactory, especially considering that repainting the screen, which is a part of the hyperlink and save tests, takes about 500–800 ms.

The times to enter editing and saving range from reasonable (some 6 s to save an already edited small document) to a bit on the slow side (13 s for saving a large document for the first time). However, we note that the performance is similar to that of the 9500's native applications. A quick test yielded that opening an unformatted 64 kB text file with the native Documents application, which ships with the phone, takes some 7 s, and saving the same document around 5 s.

We note that there is a peculiar discrepancy between the increases in save time for the Save 1 and n tasks: in the former case, there is very little increase, whereas in the latter case the save time almost doubles between 1 kB and 64 kB. This has to do with mixed usage of Text and StreamedText items: in the former case the wikitext is copied as a byte stream from one file to the other, in the latter case a Java string with the wikitext is converted to UTF-8 and serialized as bytes, a process which we have found to be rather slow.

We also measured maximum Java memory usage over various points in the view-edit-save-view cycle, as well as the size of the forward delta holding the changed text. These results are in Table 2. We see that memory consumption remained well below 1 MB during all tasks, and that the forward deltas are quite compact, with only some 3 kB of XML for reference nodes in addition to the article text.[7]

4.2 Alternate Implementations

To validate the use of the Fuego XML stack, we implemented alternate versions of the viewer that used more traditional approaches for storing the wikitext. These versions, listed below, were written following good engineering practice in order to provide a fair comparison.

File store (FS) In this version, each article is stored in a separate XML file, yielding some 240 000 files on disk. As is customary, the files were dispersed over several subdirectories to improve file lookup speed.

SQL store (SQL) The enwik9 dump was imported into a MySQL[8] database, using the default Wikipedia database schema. Articles were then fetched using SQL statements, which were prepared before execution in order to improve performance.

[7]The 64 kB article shrunk below 64 kB because we serialized some entities differently.
[8]http://www.mysql.com

Table 3: Performance comparison between Fuego, SQL, and FS implementations

	Fuego	SQL	FS
Import time (ext2)	170 s	1620 s	456 s
Read time (ext2)	7123 ms	7736 ms	6952 ms
Space (ext2)	1017 MB	1789 MB	1629 MB
Space (FAT16)	1017 MB	1790 MB	≈14000 MB

For each version, we obtained three measurements

Import The time to import the enwik9 file into the store. This time includes I_K index building, splitting into files, etc. I_T building, which would roughly double import time, was disabled during the comparison since that indexing functionality was not available in the alternate implementations. If the I_T index were added to the alternate implementations, their import time would increase as well.

Random read The time to read a sequence of articles. The same random ordering was used for all implementations. We disabled streaming text nodes in the Fuego version during this test to make our approach more similar to the others. Read time was measured with a warmed file system cache.

Space The amount of disk space required by the store implementation. The test was run on both ext2 and FAT file systems. FAT was included as it is commonly used on solid state storage devices. We used the FAT16 variant, as several contemporary devices (e.g., the 9500 Communicator) do not yet support the newer FAT32.

All tests were run on a 3 GHz Pentium 4 desktop running Ubuntu Linux 6.10 with 2 GB RAM. We could not run these tests on the 9500 since the alternate versions were not feasible on that platform due to storage limitations and lack of a suitable SQL engine.

The results are in Table 3. Looking at import time, we find that our approach is the fastest, followed by FS at more than twice the time. SQL setup is significantly slower, even though we measure only the time to run the import, not the actual conversion from XML to SQL.

In the read test there are only minor differences between the approaches, with some minor overhead in the Fuego API over the FS case, and still some more overhead for using SQL. If we enable streaming in the Fuego case to save memory, this adds some 1300 ms to execution time due to additional seeks in the input file. Note that this option to reduce memory usage is not available in the other implementations.

Looking at space overhead we find that our approach saves some 40% compared to the SQL and FS on ext2 approaches. With FAT16 the FS approach becomes infeasible, as needed space surpasses the 4 GB limit of that file system, and we have to resort to an estimate based on extrapolation. This happens because FAT16 allocates space per file in increments of 32 kB. This issue should be alleviated as more modern file systems become prevalent on solid state storage, but it is a concern with current devices.

In summary, we see that the Fuego approach provides comparable performance to other common approaches, while providing significantly lower space overhead and faster import times. Furthermore, we note that these results alone

already rule out the FS approach on the 9500 due to space requirements.

5. RELATED WORK

Delayed, also known as lazy, evaluation is a basic technique in algorithmic design. In the reftrees API we use explicit rather than implicit evaluation of delayed structures, akin to Scheme-style delayed evaluation [2], and less like the Haskell-style "invisible" laziness [22]. This latter form of laziness is used in the "lazy" mode of the Xerces XML parser[9], as well as in the lazy XML parsing and transformation research presented by Noga, Schott, et al., in [21, 23]. Their positive findings on the performance improvements that a lazy approach may offer should be pertinent to reftrees.

A form of delayed evaluation is also behind the XML projection technique [20] where an in-memory model of an XML document is constructed only as much as is needed for answering a given query. The described technique based on projection paths should be directly applicable to obtain the set of nodes to expand in a reftree. With reftrees, data may be expanded beyond the initial projection, thus permitting more dynamic and flexible access to the result.

Inflatable XML trees [13] are close in functionality to tree reference nodes combined with lazy parsing. Like these, inflatable trees are based on the idea of parsing only what is needed by the application, and they also support efficient serialization by direct copying of bytes. The main difference is that inflatable trees are not index-based but store the unparsed bytes in memory, so they would not be suitable for our use where usable RAM is very limited.

Compact representation of XML trees in memory has also been achieved by sharing of subtrees [5], effectively making the tree a DAG. This technique is very useful in saving memory, but it requires the separation of content and structure, and this requirement is not addressed clearly. Busatto et al. [7] keep the content in a string buffer, and during traversal determine the indices. Neither technique seems amenable to arbitrary traversals of a document in memory.

The reftrees update model has two features that stand out: support for the move operation and the construction of a change tree based on the cumulative effect of edits. An update model based on insert and delete operations, as was done in early work of Tatarinov et al. [29], is still often used as the basic model. However, we and others argue [9, 12] that the move is a valuable addition, as it can capture *structural* changes – and XML is indeed much about structure. The lazy XML update mechanisms proposed in [8], and the versioned documents in [31] both keep a log of updates, as opposed to our approach, which maintains a state change.

In RAXS, we use reftrees for computing XML deltas and maintaining versioned documents. References across trees are used in the same manner in the Reference-Based Version Model (RBVM) [10] to implement versioned documents. [11] addresses querying, and finds RBVM to provide an excellent basis for queries on versioned documents.

The Xyleme XML warehousing project uses a versioned XML repository [19] for storage and retrieval of XML. New document versions are received as snapshots of their state, based on which a delta between the new and present versions is computed. The deltas are *complete* in the sense that they can be used both in the backward and forward directions; past versions are retrieved by following a chain of deltas starting at the current version. The set of edit operations is the same as those used with our mutable reftree.

In [6] Buneman et al. propose a method for archiving scientific XML data, and observe that it is important that the deltas equate elements properly over versions, i.e., that the "same" element can be tracked through versions. This is an issue especially in the case that document snapshots, rather than a log of edits is used, as is the case in Xyleme and the archive of Buneman et al. We note that in our XML stack element tracking is straightforward if changes are made through the mutable reftree interface.

For delta-based approaches, version recall has been optimized by the use of techniques for co-locating referenced nodes [10] and by storing full intermediate versions [32]. The latter approach is immediately applicable to RAXS.

Experience shows that generic DBMS solutions do not necessarily fit all application scenarios [24, 25]. In particular, with large data sets, a customized solution can be up to one or two orders of magnitude faster than a generic one. On mobile devices, the data sets may not be extraordinarily large, but mobile devices are typically a few orders of magnitude slower than desktops, thus creating a similar situation.

Space usage concerns could be alleviated by using compression or an alternate serialization format[10]. However, document editing requires random access [33], so standard compression algorithms would not work that well. Furthermore, common alternate formats usually tokenize names starting from the beginning, so while they preserve the structure of the XML file, the XAS parsing state would need to be extended to include the tokenization in effect. While our system includes support for alternate formats, we do not yet have a seekable parser for them. Finally, using an alternate format would entail transformations from textual XML that would not be feasible on a mobile device for large files.

6. CONCLUSIONS

We presented the Fuego XML stack, which is a toolbox of APIs for efficient reading, lookup, writing, and management of XML documents that utilizes out-of-order parsing and lazy data structures. Using the toolbox, it becomes feasible to process even gigabyte-sized XML files on weak devices, such as mobile phones. We demonstrated this by implementing a multiversion editor for Wikipedia XML dump files. We measured the performance of the editor on a Nokia 9500 Communicator smartphone when editing a 1 GB XML file, and found it to be sufficient for real-world usage. We also compared our approach to alternate implementations, and found that our approach allows much shorter set-up time and requires significantly less space on disk. Finally, the editor demonstrates that our approach scales both with the number of document nodes as well as with the size of the individual nodes in the XML document.

The key to short set-up time is that we read from and write to verbatim XML files: the XML file *is* the data store. No document shredding or other conversion operations are required. This way, we maintain maximum compatibility with other XML processing tools. Our stack is also an example of the usefulness of a *rich* XML API, which is an XML data management API that goes beyond simple parsing and serialization, but without becoming a full-blown XML database.

[9] `http://xerces.apache.org/`

[10] `http://www.w3.org/XML/Binary`

7. REFERENCES

[1] *ACM Symposium on Document Engineering*. ACM Press, Nov. 2002.

[2] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[3] P. M. G. Apers, P. Atzeni, S. Ceri, et al., editors. *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., Sept. 2001.

[4] D. Brownell. *SAX2*. O'Reilly, Sebastopol, California, USA, Jan. 2002.

[5] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In Freytag et al. [14], pages 141–152.

[6] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. *ACM Transactions on Database Systems*, 29(1):2–42, Mar. 2004.

[7] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In *Tenth International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 199–216, Aug. 2005.

[8] B. Catania, B. C. Ooi, W. Wang, and X. Wang. Lazy XML updates: Laziness as a virtue of update and structural join efficiency. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 515–526, June 2005.

[9] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37, May 1997.

[10] S.-Y. Chien, V. J. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In Apers et al. [3], pages 291–300.

[11] S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Supporting complex queries on multiversion XML documents. *ACM Transactions on Internet Technology*, 6(1):53–84, Feb. 2006.

[12] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18th International Conference on Data Engineering*, pages 41–52, Feb. 2002.

[13] R. Fernandes and M. Raghavachari. Inflatable XML processing. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*, volume 3790 of *Lecture Notes in Computer Science*, pages 144–163, Nov. 2005.

[14] J. C. Freytag, P. C. Lockemann, S. Abiteboul, et al., editors. *29th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, Sept. 2003.

[15] S. Keshav. Why cell phones will dominate the future Internet. *Computer Communication Review*, 35(2):83–86, 2005.

[16] P.-Å. Larson. Dynamic hashing. *BIT Numerical Mathematics*, 18(2):184–201, 1978.

[17] B. Leuf and W. Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet.* Addison-Wesley Professional, 2001.

[18] T. Lindholm, J. Kangasharju, and S. Tarkoma. A hybrid approach to optimistic file system directory tree synchronization. In *Fourth International ACM Workshop on Data Engineering for Wireless and Mobile Access*, June 2005.

[19] A. Marian, S. Abiteboul, G. Cobéna, and L. Mignet. Change-centric management of versions in an XML warehouse. In Apers et al. [3], pages 581–590.

[20] A. Marian and J. Siméon. Projecting XML documents. In Freytag et al. [14], pages 213–224.

[21] M. L. Noga, S. Schott, and W. Löwe. Lazy XML processing. In *ACM Symposium on Document Engineering* [1], pages 88–94.

[22] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[23] S. Schott and M. L. Noga. Lazy XSL transformations. In *ACM Symposium on Document Engineering*, pages 9–18, Nov. 2003.

[24] M. Stonebraker and U. Çetintemel. "One size fits all": an idea whose time has come and gone. In *21th International Conference on Data Engineering*, pages 2–11, Apr. 2005.

[25] M. Stonebraker, U. Çetintemel, M. Cherniack, et al. One size fits all? — part 2: Benchmarking results. In *Third International Conference on Innovative Data Systems Research (CIDR)*, Jan. 2007.

[26] Sun Microsystems Inc. *JSR 62: Personal Profile Specification*, Sept. 2002.

[27] Sun Microsystems Inc. and Motorola Inc. *JSR 118: Mobile Information Device Profile Version 2.0*, Nov. 2002.

[28] S. Tarkoma, J. Kangasharju, T. Lindholm, and K. Raatikainen. Fuego: Experiences with mobile data communication and synchronization. In *17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, Sept. 2006.

[29] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 413–424, May 2001.

[30] I. Tatarinov, S. D. Viglas, K. Beyer, et al. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, June 2002.

[31] R. K. Wong and N. Lam. Managing and querying multi-version XML data with update logging. In *ACM Symposium on Document Engineering* [1], pages 74–81.

[32] R. K. Wong and N. Lam. Efficient re-construction of document versions based on adaptive forward and backward change deltas. In *Proceedings of the 14th International Conference on Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 266–275, Sept. 2003.

[33] World Wide Web Consortium. *XML Binary Characterization Properties*, Mar. 2005. W3C Note.

[34] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*, 4th edition, Aug. 2006. W3C Recommendation.