Database Support for Image-based Electronic Editions

Alexander Dekhtyar[*]
Dept. of Comp. Sci.
Univ. of Kentucky
dekhtyar@cs.uky.edu

Ionut E. Iacob
Dept. of Comp. Sci.
Univ. of Kentucky
eiaco0@cs.uky.edu

Jerzy W. Jaromczyk
Dept. of Comp. Sci.
Univ. of Kentucky
jurek@cs.uky.edu

Kevin Kiernan
Dept. of English
Univ. of Kentucky
kiernan@uky.edu

Neil Moore
Dept. of Comp. Sci.
Univ. of Kentucky
neil@s-z.org

Dorothy Carr Porter
Research in Computing for Humanities
Univ. of Kentucky
dporter@uky.edu

ABSTRACT

Image-based electronic editions are an emerging type of a multimedia information system. They allow scholars and students to study editions of ancient texts, including a transcript, a comprehensive glossary, and full critical apparatus, in the context of their manuscript source or sources. Building image-based electronic editions poses a two-fold challenge. The humanities editor must be able to annotate manuscript images to represent a wide range of features. Because these annotations can be quite dense, with complex interrelations, the computer scientists must develop mechanisms for representing them in a way that associates the images and encoded text, enabling efficient querying of both. This paper addresses the latter problem, by proposing data structures to collect, store and query image-based annotations and to preserve image-to-text and image-to-markup mappings.

1. INTRODUCTION

Image-Based Electronic Editions of historic documents and document collections are beginning to emerge as important new resources for humanities scholars and the general public. These editions can provide at the same time any number of researchers with first-hand access to digital images of unique and fragile material that is not otherwise widely available for study.

Briefly defined for the purposes of this paper, an Image-Based Electronic Edition (IBEE) thoroughly integrates images of the manuscripts on which the text is based and provides structured access to all aspects of the encoded edition. The edition thus consists of three parts: first, the collection of images of the manuscript; second, a transcript of the manuscript; and third, an edition of the transcript, based on the images. The edition is built by annotating all relevant features of the images and transcript, from the codicological and paleographical aspects of the manuscript, to the formatting of the text, to the variant readings, emendations, restorations, glossary and notes of the edited text.

The problems studied in this paper have arisen in our work on the ARCHway Project, a collaboration of humanities scholars and computer scientists with the aim to solve the technological problems that arise from the highly complex encoding required by the seamless integration of images and text. Our examples of IBEEs in this paper derive from extensive practical experience building these editions from famous Old English manuscripts, which were damaged by fire in the eighteenth century and are now in the British Library[1].

Figure 1 depicts the ARCHway approach to IBEEs. The software architecture consists of three layers: application, a middleware, and the data management. The top layer can be split into two parts: editorial tools of the IBEE and presentation tools that form the front-end of the prepared Electronic Edition. Thus, the ARCHway architecture can conceptually be represented in a "Y" shape, with the left "hand" forming the Edition Production Technology (EPT), while the right "hand" being the actual IBEE software. At the intersection of the Y is the middleware, which integrates the different editorial and presentation tools and provides a level of abstraction between them and the data management layer, which contains the facilities for

*The article is based on work supported in part by the National Science Foundation under Grant No. 0219924, awarded pursuant to the authority of the NSF Act of 1950 (42 U.S.C. 1861 et. seq.). It is subject to GC-1 Grant General Conditions (10/98) and is made in accordance with the provisions of NSF 98-63, "Information Technology Research". In addition, the work of the forth and the sixth authors is supported in part by a Collaborative Research Award from the National Endowment for the Humanities and the Andrew W. Mellon Foundation.

[1]Our model for an IBEE is the Electronic Beowulf[16, 2], now in version 2.0, but we are using many new editorial tools designed and developed by the Electronic Boethius[11, 3] and ARCHway[14] projects.

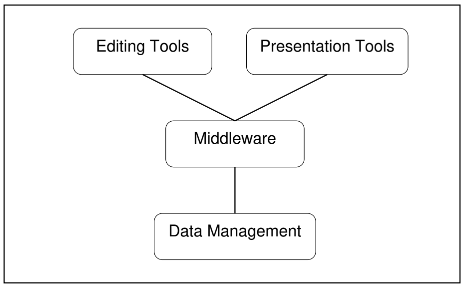

Figure 1: The high-level ARCHway project architecture for Image-Based Electronic Editions.

storage, maintenance and retrieval of the accumulated information. The Y also helps distinguish between the work of computer scientists and software developers in the stem and the work of the humanities scholars (editors), which includes the design of the editorial tools and the ultimate use of the IBEE. Both “arms” share the bottom layers of the software, while the applications residing on top of these layers are customized to perform specific editorial or presentation tasks.

This organization of IBEEs poses a number of unique technical challenges. Among them, however, the problem of proper management of continually accumulating data lies at the heart of IBEE construction. As outlined above, the data component of IBEEs consists of three types of data: (a) images, (b) transcript and (c) integrated annotations of the image and transcript to build the edition. To create an IBEE, the editor studies the images and introduces appropriate markup annotations in the transcript, transforming it in the process into an image-based edition. Because the vast majority of this markup has its origin in the images of the primary source, the manuscript, the resulting encoding must somehow incorporate the mapping between different regions of the image files and the annotated regions of the text.

The central concern in developing solutions to this problem is efficiency of processing of information requests. A number of straightforward solutions to the problem, such as the use of image-based “bounding-boxes” for each annotation, do not capture in an efficient manner different complexities that inevitably arise. Two dimensional folio images represent one-dimensional structure - the manuscript transcript, which leads to situations where a specific feature (e.g., a sentence in the text) has to be represented as a number of disjoint regions on the image.

Also important is the encoding used to represent image- and text-based annotations. The Extensible Markup Language (XML) [6] is a widely accepted technology for annotating textual literary documents [19]. XML is designed to represent ordered hierarchical structures, such as a codex consisting of a collection of folios, which are themselves made up of lines, which themselves contain characters. In XML, these features can be annotated by *markup elements*, which nest to form a hierarchical structure—an annotated tree. Adding to the complexity of dense textual markup is the editor’s need to encode multiple *concurrent hierarchies* of XML markup, which are not easily collapsed into a single hierarchy. The problem of encoding multiple hierarchies of markup is known as *concurrent markup*; one solution is an extension of XML known as *concurrent XML* or CXML [7, 13], which treats markup elements as potentially overlapping intervals spanning ranges of text.

In this paper, we describe two structures for storing and maintenance of image-based concurrent XML markup for electronic editions. Section 4 describes the use of segment trees to store and process concurrent hierarchies of markup in main memory. Our second structure (Section 5) is an R-tree derivative data structure, which we call the *folio R-tree*, designed specifically to store folio image-to-text mapping in secondary storage. For both solutions we describe range query algorithms which determine the text/XML encodings found in the specified region of a manuscript folio image. We start, with describing the work of IBEE editors in more detail (Section 2 and formalizing the problem of image-to-text-to-XML mapping (Section 3). Finally, Sections 4 and 5 describe the segment tree and folio R-tree data structures, respectively.

2. ELECTRONIC EDITIONS: WHAT’S IN AN IMAGE?

There are many open problems associated with encoding manuscripts in XML based on images, and with maintaining and efficiently retrieving the relationship between images and text. Images of medieval manuscripts, especially ones that are badly damaged by fire, contain both more and less than what we think of as the text. If we consider a single image of a page of a manuscript, we can begin to see how complex its relationship is to a simple transcript of the text on that page. As a physical object, a manuscript page is one side of a folio, and one-half of a folded sheet in a group of folded sheets called a quire or gathering. When these gatherings are bound together they form a manuscript, or codex. In a damaged manuscript that has lost its binding we often have to under-

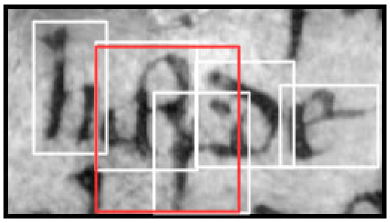

Figure 2: hæfde

stand the relationship of a folio to its gathering, and to the manuscript as a whole, in order to restore the text accurately. With the manuscripts we are editing the situation is made more complicated by the fact that each damaged folio is framed in heavy paper, and these paper frames often contain relevant information about the text they surround. Moreover, the fire damage to the manuscripts we are editing has often left gaping holes and scorched vellum where the text is no longer legible. In many cases special lighting techniques, in particular ultraviolet fluorescence and fiber-optics, can restore illegible text, but then the nature of the image has changed, and an edition must meticulously encode the sources of these different views. While there are often secondary sources, such as another manuscript of the same text or an early transcript, to help restore the text or place what remains of it in an understandable context, an image-based edition must encode the relationship between the image and the edited text. There are in fact many layers of information embedded in images, including, for example, physical damages, palimpsests, technological restorations, grammatical information, textual notes, marginalia on the manuscript and the paper frames, page layout, scribal features, editorial emendations and conjectural restorations, all of which must be encoded in XML.

We can observe some of the complexity of XML markup on a microcosmic level in Figure 2, an image of the Old English word *hæfde*. The three Old English letters, *æfd*, overlap one another, making it impossible to isolate a single letter, as the word is represented in a text file. In the image the e-tongue of the *æ*, instead of sticking out like the final *e* of the word, turns down to form the shaft of the *f*, while the ascender of the following *d* invades the space over the *f*. A text file cannot reflect these paleographical features of the handwriting. From this example we can see that integrating the XML encoding with the images of these three letters is a more complicated matter than it first seems, and it should not be hard to see that mapping the XML markup to the images is a highly desirable goal for anyone interested in minute features of manuscript images.

3. SUPPORT FOR IBEES

From the data management perspective the uniqueness of Image-based Electronic editions lies in the way images guide the creation of the markup. In this section we describe the issues that need to be resolved in order to achieve this and outline our approach to dealing with them.

Preservation of the text-to-image and XML-to-image mappings is facilitated by a number of the editorial tools that reside in the application layer of the EPT (see Figure 1). Among these tools are a document-centric XML editor and image management tools. To be able to keep track of the mapping, the image management tool provides the editor with the functionality to highlight a region on the image. This information is then shared with the XML editor for the explicit purpose of associating the image coordinates with the tagging task underway.

Thus, our goal is to provide efficient data management support for a number of tasks, in particular: storage of document-centric XML based on an existing (and potentially large) text file; the insertion of image-based XML encoding; preservation of the image-to-XML and image-to-text mapping established in the editorial tools; and efficient retrieval of information related to the text/XML-to-image mapping. To that extent, we consider two possible scenarios. In the first scenario, all data storage and data management occurs in main memory, while in the second we choose to store all information in the secondary storage. Both scenarios are viable and, in fact, complimentary. Main-memory processing is bound to be more efficient, but has certain limitations on the size of the objects. Even when the XML constructed for an electronic edition is large enough that it precludes storing all the data in main memory at the same time, in-memory processing can be used on buffered fragments. The secondary storage processing ensures the scalability of the proposed approach and relieves the problem of information volatility.

The methods described in Sections 4 and 5 assume the following properties of the XML markup of IBEEs:

- The smallest (non-empty) unit of XML markup is a single character of the manuscript text (and appropriate 2D bounding box on the image).

- Any feature on the image is encoded in relation to the transcribed text.

- Empty XML markup elements have uniquely identifiable positions between the characters of the manuscript text.

Our approaches to storing data in main memory and on disk are somewhat different. For main-memory processing (Section 4), we adopt the well-known in computational geometry segment tree data structure [17, 4], which will, in essence, replace the traditional DOM tree in storing XML, with the additional benefit of linking XML and text to the document images. The segment tree **insert** algorithm presented in the next section demonstrates how this structure can be efficiently manipulated. To illustrate how the proposed way of preserving the mapping can be used to efficiently answer range queries: queries that find text/XML inside a given region on an image, and find the region(s) on images that contain specific text/markup, we describe the **stab** algorithm.

In contrast to the all-in-one approach of segment trees to managing image-based XML, our secondary storage data structures (Section 5) separate the storage of XML from the index structure designed specifically to preserve the text-to-image mapping. At present time, representation of XML in relation databases is an area or active research[21, 20, 9, 5], however, different representations have been shown to work better in different situations. In this paper, we factor out the problem of representing XML in a relational database, by taking the "black box" approach: we assume that XML is stored in a manner that allows us to process efficiently the following queries:

- Given a range in the text content, find all XML elements whose content is a subset, superset or overlaps the range;

- Given an XML element node, find its full content.

If a full relational database system is not necessary, a persistent form of the segment tree structure can be used to answer these queries.

Adopting this assumption allows us to shift our attention to the problem of preserving the text/XML-to-image mapping in the secondary storage database. We address the latter problem by introducing a data structure called *folio R-tree*, an R-tree derivative, tailored for the specifics of the manuscript images. The details of both approaches are shown in the next two sections.

4. USING XML SEGMENT TREES

A collection of data structures supporting successful markup for the class of manuscripts that we are considering must satisfy several requirements. First, it must be able to represent multiple hierarchies of markup elements [13] in a way that supports natural markup queries. Second, it must provide an effective representation of the geometry of manuscripts in a way that allows for an intuitive user interface. Furthermore, the data structures must be dynamic; that is, they must efficiently support the addition, deletion, and modification of markup. Finally, the structures must support conversion of data to and from XML, to allow the use of standard XML processing utilities.

The image-based approach to the tagging process suggests the use of geometric structures, see [18]. The choice of structures depends on the particular needs of the editorial process, and can be viewed (quite literally) in various dimensions. Specifically, the shape and spatial distortion of the manuscript page are three-dimensional features; damage and restoration, as well as marginalia, are best viewed in two dimensions; and the text and its linear structure can be considered in one-dimensional space. In this paper we will focus on this linear view of the text and briefly discuss how 2-dimensional features can be incorporated into it. In the rest of this section we will describe and demonstrate how the segment tree [4] structure can be adopted to this end.

4.1 The tagging process

In order to lay the groundwork for formal presentation, we need to make a number of assumptions about the structure of the document and the process of adding new annotations encoded as concurrent XML (called "tagging"). We will consider markup that applies primarily to the manuscript text, which for our purposes is treated as lines of letters. Using a tool called the *line tracer*, the editor divides the folio into lines. In its user interface, the line traces displays the delineations as thin curves known *line traces* (Figure 3).

The tracer tool automatically divides each line horizontally into a number of elementary (or *atomic*) segments. The number of such segments depends on the needs of the manuscript. For example, if we wish our coordinate system to remain independent of the resolutions and scales of individual images, we may choose to have elementary segments 1/72 of an inch in width. In other cases it may be more natural to make each elementary segment a certain number of pixels wide. At other times, it may simplify matters to have each line, regardless of its length, contain the same number of elementary

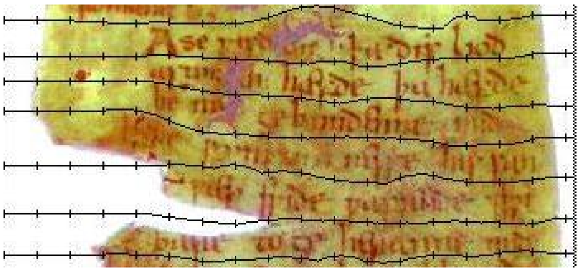

Figure 3: Fragment of a folio image with line traces

segments.

Once elementary segments have been established, the tracer conceptually arranges to elementary segments into a linear order by identifying the end of each line with the beginning of the next. This establishes a one-dimensional coordinate system over the entire folio. If w_n is the number of elementary segments making up line n, the first line of the page extends from segment 0 to segment w_1, the second from w_1 to $w_1 + w_2$, and so forth. By furthermore identifying the end of each page with the beginning of the next, we can extend this coordinate system to cover the entire manuscript. Because lines and folios are connected according to the normal flow of text, a single interval can represent a textual feature which wraps across from one lines or pages to another. Furthermore, the positions and shapes of the line traces can be used to map each elementary segment, and hence each interval, back to the corresponding two-dimensional region of the manuscript image, providing a text-to-image mapping which complements that of the folio R-tree (Section 5).

Given the assumptions about the markup stated in Section 3, we can choose our data structures for the representation of markup and content. If we have established a one-dimensional order on the manuscript, as described above, each letter and each markup element is represented as an interval in that coordinate system, possibly spanning lines or even folios. One data structure that efficiently represents dynamic collections of intervals is the segment tree [4], which we will describe in section 4.2. We can use the segment tree to perform some types of queries directly; other queries can be performed by first constructing a document tree model (for example, using the document object model, DOM[1]) containing the relevant markup [8] [7], and using standard document-processing tools. In addition, the ability to define the size of elementary segments based on the manuscript images makes the segment tree structure more flexible and better suited for the image-based markup process than structures whose region boundaries are determined by the markup data.

In our approach, the tagging process takes place in a number of steps. First, images are combined to form manuscript pages; for example, one page may consist of the blended overlay of a daylight image and an ultraviolet-enhanced image of the same leaf. These pages are arranged in textual order, and each page has its lines marked with traces. Once these steps have been performed, the editor selects ranges of the image and describe them as content (letters) or as markup elements, assigning attributes as appropriate.

4.2 Segment trees for XML Markup

Our XML segment tree data structure[12, 13] is based on segment trees. The segment tree was introduced in [4, 17] as a geometric dynamic data structure to represent and to perform a number of update (e.g., insert and delete) and query operations on a set of segments. It is assumed that the coordinates of all the endpoints for the potential segments, but not the segments themselves, are known in advance. This corresponds in our case to marking entities along a given line, where each point on the line has its coordinate measured in some unit (pixels, points, millimeters, etc.), and selecting in the marking process intervals that bracket those entities.

The underlying structure for the segment tree is a balanced binary tree with leaves representing atomic (or smallest, indivisible) segments. Each node represents the union of the atomic segments rooted in this nodes. For example, the root of the tree represents all the atomic segments. Intervals that belong to the collection represented in the segment tree are associated with nodes of the tree and satisfy the following property: a node v stores s if the union of its atomic segments is contained in s but the union of the atomic segments associated with the parent of v do not. Thanks to this property each segment is associated with at most $\mathcal{O}(\log n)$ nodes because it needs to be represented at most twice on any level in the tree [17].

In our adaptation, each node N of the segment tree contains:

- The interval $N.\mathbf{I} = [\alpha, \omega]$ covered by N.
- A list $N.$**elements** of those elements and characters stored in N (as described above). We need to traverse the list in sorted order; because of this, $N.$**elements** may be better represented as some form of binary search tree.

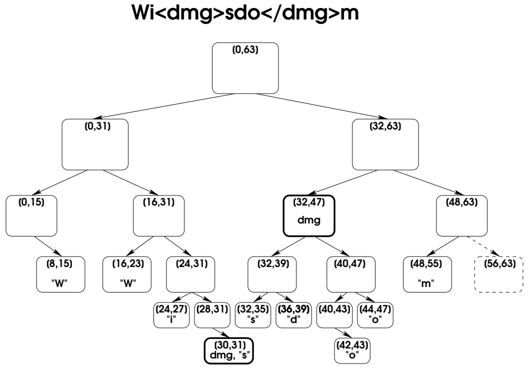

Figure 4: Segment tree

- Pointers N.**left** and N.**right** to the left and right subtrees, respectively, of N. If a subtree has not yet been constructed, or if N represents an atomic segment, one or both of these pointers may be empty (which we represent by the symbol $\emptyset$).

For our application, each atomic segment's endpoints correspond to two consecutive coordinates in the selected positioning system for horizontal lines in an image (in some cases we will use points as the atomic segments, as well). Segments correspond to the tagged ranges and they are named with the marking tags. A sample segment tree with atomic segments $(0, 1), \ldots, (62, 63)$ is depicted in Figure 4. These coordinates are established by the line tracer (Section 4.1), based on the division of the folio into elementary segments. To save space, only nodes that are marked or whose descendants are associated with marked ranges, exist in the tree. New nodes are added to the tree through `insert` operation if needed. One such nodes, which corresponds to a potential union of atomic segments but is not actually present in the tree, is depicted with dashed lines in Figure 4. This figure describes a set of intervals corresponding to the XML "`Wi<dmg>sdo</dmg>m`". Each letter and each markup element is represented by a segment, which typically spans a number of atomic segments. From the tree we can readily find the span of each range. For example, the <`dmg`> tag covers a range from 30 to 47.

Insertion of a new interval s into segment tree is a simple operation of marking some nodes along the insertion path in the tree (see next section). Basically, we want to mark the smallest number of nodes whose union of the corresponding atomic segments equals s. This is done by placing the beginning and ending points of s among the nodes of the segment tree in a fashion similar to the binary search, and marking the topmost nodes that lie between them. In Figure 5, we show the segment tree after inserting a range for <`rep`> which spans the interval from 24 to 41; the affected nodes are highlighted. This could be represented in pseudo-XML as "`W<rep>i<dmg>sd</rep>o</dmg>m`". Note that elements located in the same node of the segment tree are listed with the longer range first. As discussed earlier, this is needed for the correctness of the traversal procedures to generate pseudo-XML documents from the segment tree.

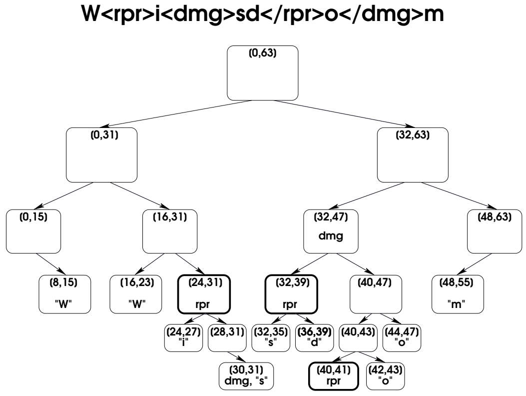

Figure 5: Modified segment tree

4.3 Operations on segment trees

Segment trees support a number of operations for updating and querying the tree. They include basic operations to update the tree, operations to query the tree and operations to support concurrent XML markup and its various representations. Here we will describe **insert**(I), which stores the interval I in the tree, and **stab**(x), which returns a collection of all the stored intervals which contain the point x. In the following pseudocode, r refers to the root of the segment tree. For an interval $I = [\alpha, \omega]$, mid(I) is the point $\lfloor (\alpha + \omega)/2 \rfloor$. Given a node ν, we write mid(ν) for mid$(\nu$.**interval**$)$, $\alpha(\nu)$ for the left endpoint of ν's interval, and $\omega(\nu)$ for the right endpoint of ν's interval.

```
insert(Node ν, Element e)
    if ν.interval ⊆ I(e)
        ν.elements ← ν.elements ∪ {e}
    else
        if α(e) ≤ mid(ν)
            if ν.left = ∅
                ν.left ← new Node(α(ν), mid(ν))
            insert(ν.left, e)
        if ω(e) > mid(ν)
            if ν.right = ∅
                ν.right ← new Node(mid(ν) + 1, ω(ν))
            insert(ν.right, e)
```

The **insert** procedure takes as arguments a segment tree node ν (initially the root of the segment tree) and the element e to be stored in the tree. If ν's interval is contained in e's interval, e is inserted into the **elements** list of ν (line 3). In this case, no descendant of ν will store e, so **insert** does not recurse into either subtree.

If, on the other hand, ν's interval is not contained in e's, the procedure recurses into zero or more of ν's children. If e's interval intersects the left half of ν's interval (line 5), e is recursively inserted into ν's left subtree (line 8). Similarly, if e's interval intersects the right half of ν's interval (line 9), e is recursively inserted into the right subtree (line 12). If the element will be located in an uninstantiated portion of the tree (lines 6 and 10), the necessary nodes are first instantiated (lines 7 and 11).

After **insert**(e) (when e fits into the segment tree), e is stored in those nodes ν (and only those nodes) where ν's interval is contained in I(e) but ν's parent's interval is not contained in I(e). Furthermore, **insert**(e) terminates in finite time $\mathcal{O}(\log M)$, where M is the size of the search space (i.e., the number of atomic segments).

Another important segment tree operation is the stabbing query. This query returns a collection of all the stored intervals which contain a given point x.

```
stab(Integer p) :   Set
    if p ∈ r.interval
        return stab-node(r, p)
    else return ∅

stab-node(Node ν, Integer p)
    if p ≤ mid(ν) ∧ ν.left ≠ ∅
        return r.elements ∪ stab-node(ν.left)
    else if p > mid(ν) ∧ ν.right ≠ ∅
        return r.elements ∪ stab-node(ν.right )
    else return r.elements
```

The search, guided by mid(ν), is performed recursively, with logarithmic cost plus the size of the output. The **stab** operation, and slightly modified versions thereof, can be used to implement many other queries, such as markup overlap and containment.

5. USING FOLIO R-TREES FOR INDEXING IMAGE CONTENT

In this section we present data structures and algorithms, that facilitate maintenance and querying of IBEE data (image and text) using secondary storage. In addition to the maintenance issues, we consider the following two Range Query problems:

Text Range Query Given a manuscript folio image and a rectangular box, find all the markup and manuscript text found inside the box.

Image Range Query Given manuscript markup, find the corresponding folio image region(s).

These two problems correspond respectively to two questions that arise naturally for IBEEs: (i) "what are the features contained in a given image region?" and (ii) "what image regions reflect the information encoded in an XML fragment?". Under the assumptions about the IBBE markup stated in Section 3, the Range Query problems introduced above can be solved as follows:

1. Text Range Query: Find the XML encodings in a folio image region:
   (a) find all text in the given box (the Text Range Query proper problem);
   (b) find all XML markup associated with this text (an *XML Retrieval* problem proper).
2. Image Range Query: Find the folio image region(s) for some folio encodings:
   (a) find all text in the encodings (an *XML Retrieval* problem);
   (b) find the folio image region(s) corresponding to this text (the Image Range Query problem proper).

In the rest of the paper, we concentrate on solving the problem of going from image to text and from text to image. The complimentary problem of XML-to-text mapping is dealt with by the XML storage "black box" whose existence we assumed in Section 3. We note here that the retrieval of text given an XML element and retrieval of XML elements that are in a specific relationship (subset, superset, overlap) with given text can be done efficiently for all traditional approaches to storing XML in relational databases [21, 20, 9, 5]. To process text and image range queries, we propose to use a data structure called *folio R-tree*.

5.1 Folio R-Trees

R-trees [10] have proved to be efficient and flexible in storing information about objects in multidimensional spaces. The defining feature of an R-tree is that the regions described by two sibling nodes in the tree overlap. While this makes retrieval operations, such as range queries, follow multiple paths through the tree, this is well compensated by the flexibility R-trees afford in storing objects.

Informally, a *folio R-tree*, a data structure for text-to-image and image-to-text mapping, consists of three layers of nodes. Nodes at the top layer, called *folio nodes* index individual folios. Since each folio is represented by a separate image, then each folio record stored on these nodes corresponds to a two-dimensional space of its own. Nodes at the second layer, *line nodes*, index lines found on a specific folio. Finally, the third layer of nodes, *character nodes* indexes characters of a single line. The manuscript folios are sorted in ascending order, and each folio is associated to a single entry in one folio node. This entry contains a pointer to the page that indexes the lines of the folio (the page of line nodes). Each entry in a line node stores information about a single line of a folio. The entry contains the id of the line (folio number, line number), the *rectangular bounding box* for the line and the pointer to the character node which stores information about the characters from the line. Each character node entry contains the id of the character: (folio number, line number, position in line), character itself and its bounding box on the folio image. The bounding box of the line is maintained as the smallest rectangle that encloses all bounding boxes for (currently known) characters in the line.

More formally, we define the data structures for the folio R-trees as follows:

A *folio record* is of the form $\langle FolioID, ImageFile, LinesPage, LinesRecord\rangle$, where $FolioID$ stores the name of a specific manuscript folio, $ImageFile$ is the name of the image file associated with the folio and $LinesPage$ and $LinesRecord$ are a disk page address and a record slot id on the disk page respectively.

A *line record* is of the form $\langle LineNumber, ULX, ULY, LRX, LRY, CharPage, CharRecord\rangle$, where $LineNumber$ is the number (id) of the line on its folio, ULX, ULY, LRX, and LRY are the coordinates (upper left and lower right corners) of the bounding box of the line on the folio image, and $CharPage$ and $CharRecord$ are a disk page address and a record slot id on the disk page respectively.

A *character record* is a tuple $\langle CharID, CharValue, ULX, ULY, LRX, LRY\rangle$, where $CharID$ is the identifier for the character, $CharValue$ is the character value itself[2], and ULX, ULY, LRX, and LRY are the coordinates (upper left and lower right corners) of the bounding box of the character on the folio image. Disk pages consisting of folio, line and character records are called *folio*, *line*, or *character pages* or *nodes* respectively.

The folio R-tree, then, can be defined as follows. Let $F = (f_1, \ldots, f_n)$ be a collection of manuscript folios and S be the text contained on these folios. By $S[i]$, $(1 \leq i \leq n)$ we denote the substring of S contained on folio f_i. A folio R-tree T representing the manuscript $\langle F, S\rangle$ is a collection of folio, line and character nodes, such that

[2]We assume that the character value is represented in Unicode.

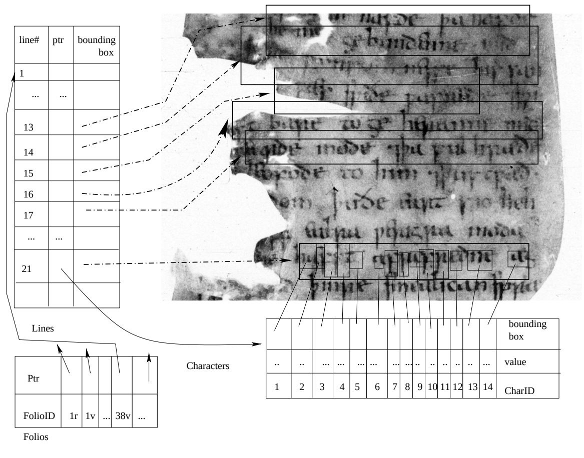

Figure 6: Folio R-trees for manuscript pages.

1. There exists a folio record for each folio $f_i \in F$; there exists a line record for each line l_{ij} of each folio f_i and there exists a character record for each character s_k of S.
2. The bounding box of any line l is exactly the minimal bounding box for all characters from this line.
3. All folio records are sorted in the order prescribed by F; all line records for a specific folio occupy consecutive records on consecutive line pages; all character records for a specific line occupy consecutive records on consecutive character pages.
4. In each folio record, $LinesPage$ and $LinesRecord$ point to the first line record for the given folio; in each line record $CharPage$ and $CharRecord$ point to the first character record for the given line.

A folio R-tree satisfying conditions 2,3 and 4 above, but not condition 1 is called an incomplete folio R-tree for the manuscript $\langle F, S\rangle$.

Example 1. Consider a fragment of a folio R-tree depicted in Figure 6. It shows how the folio R-tree is used to store information about a manuscript folio. Each folio in the manuscript gets a single entry in a folio node (lower right corner) This node stores the folio Id, filename of the associated image (not shown for simplicity) and a pointer (*pageID*, *slotID*) to the first line record for the lines of the folio. Each line record, in turn, stores the bounding box for the line and the pointer to the first character record for it. The bounding boxes may overlap, see, e.g., bounding boxes for lines 15, 16 and 17 of the folio $38v$. Each character record stores the character id (in our case just the ordinal for the character in the line, but generally, it may be a unique id, such as the byte position of the character in the manuscript text), its value (i.e., the actual Old English character) and its bounding box. Bounding boxes for characters may also overlap as boxes for characters 9 and 10 in line 21 do. □

Suppose for a moment, that a manuscript folio contains 30 lines of text, each line consisting of at most 40 characters (each character is represented with 4 bytes in Unicode). We also assume that a line id consists in two integers (4 bytes total) and a character id consists in three integers (6 bytes total). Each coordinate value is an integer of 2 bytes, each pointer to a disk page occupies 4 bytes. Then the space required for a character record is: 3×2 bytes + 1×4 bytes + 4×2 bytes = 18 bytes. A line record will occupy 2×2 bytes + 4×2 bytes + 1×2 bytes + 1×2 bytes + 4 bytes = 20 bytes. For a whole folio, all character records sum up to $30 \times 40 \times 18$ bytes = 21600 bytes. To answer a Range Query for a folio, the folio record, all folio lines records, and all folio characters records are to be retrieved (in the worst case). For a disk page size $dps \geq 30 \times 20$ bytes = 600 bytes (so that all folio line records reside on a single page), a Range Query operation requires $2 + \lceil 21600/dps \rceil$ disk reads (one for the folio record, one for line records, and $\lceil 21600/dps \rceil$ for character records).

A folio R-tree update operation (including insertion and deletion) targets a character record in the folio R-tree. Update operation takes as parameters the character to be updated and its associated information and is to be performed as follows: (i) scan for the character's folio record page, (ii) search for the character's line record page and update the line record bounding box as needed, and (iii) read the character record page and update the record data. The pseudocode of the update algorithm is given below.

```
F-Tree-Update(T,folio,line,CharID,CharValue,Box)
  input:  T: pointer to the root of folio R-tree;
          folio: folio ID;
          line: line number;
          CharID: the character ID
          CharValue: the character value
          Box: character bounding box.

Scan T for record r s.t. r.FolioID = folio;
Scan r.LinesPage, starting with r.LinesRecord, to find
    line record l s.t. l.LineNumber = line
// Update line information:
  l.ULX ← min(l.ULX, Box.ULX);
  l.ULY ← min(l.ULY, Box.ULY);
  l.LRX ← max(l.LRX, Box.LRX);
  l.LRY ← max(l.LRY, Box.LRY);
Scan l.CharPage, starting with l.CharRecord, to find
    character record c s.t. c.CharID = CharID
// Update character information:
  c.CharID ← CharID; c.CharValue ← CharValue
  c.ULX ← Box.ULX; c.ULY ← Box.ULY;
  c.LRX ← Box.LRX; c.LRY ← Box.LRY;
```

5.2 Range Queries Using Folio R-trees

Let T be a folio R-tree for some manuscript $\langle F, S \rangle$. The algorithms for solving the Range Query problems are derived from the standard range query algorithm for R-trees. To simplify notation, we will use $BoundingBox$ to denote the quadruple of attributes ULX, ULY, LRX, LRY in line and character nodes of folio R-trees.

Below is the pseudocode of the algorithm for the Text Range Query problem. This pseudo-code makes a simplifying assumption that there is only one folio node in T. This assumption, however, is not necessary - if the number of folios exceeds the capacity of a single disk page, more pages can be used.

```
TextRangeQuery(T,folio,Box)
  input:   T: pointer to a folio R-tree node/record;
           folio: folio ID;
           Box: query range.
  output:  C: Set of character nodes intersecting Box.

if (T is a folio page) then
  Scan T for record r, s.t., r.FolioID = folio;
  T1 ← (r.LinesPage, r.LinesRecord);
  C ← TextRangeQuery(T1, folio, Box);
else if (T is a line page) then
  C ← ∅;
  for each l - line record for folio
    if (l.BoundingBox ∩ Box ≠ ∅) then
      T1 ← (l.CharPage, l.CharRecord);
      C ← C ∪ TextRangeQuery(T1, folio, Box);
else if (T is a character page) then
  C ← ∅;
  for each c - character record for current line
    if (c.BoundingBox ∩ Box ≠ ∅) then
      C ← C ∪ {c};
return C;
```

The algorithm proceeds as follows. At first stage (T is the folio node that is the root of the folio R-tree), it scans the folio page to find the record for the specified folio. Upon finding it, the algorithm downloads the first disk page on which the lines of the folio are stored. Starting with the first line record for the folio, the algorithm scans all line records for it (by the definition of the folio R-tree, they are stored in a sequence), downloading any new line pages as needed, and compares the input bounding box with the bounding box for each line in turn. If a non-empty intersection is determined, the algorithm downloads the disk page that stores the character records for the given line and scans its character records for intersection with the input box. All characters that are found to have such intersection are assembled in the answer set.

The Image Range Query algorithm is shown below. The algorithm takes as input a pointer to a folio R-tree node, a folio, and a *text range*, $TextRange = (startLine, startOffset, endLine, endOffset)$, specified by characters starting and ending positions (lines and offsets). It returns a set of bounding boxes for the characters in the input text range.

ImageRangeQuery(T,$folio$,$TextRange$)

`input`:	T: pointer to a folio R-tree node/record;
	folio: folio ID;
	TextRange: text range.
`output`:	B: Set of character bounding boxes.

$B \leftarrow \emptyset$;
Scan T for record r, s.t., r.**FolioID** = *folio*;
Scan r.**LinesPage**, starting with r.**LinesRecord**, to find
  line record l s.t. l.**LineNumber** = *TextRange*.**startLine**;
Scan l.**CharPage**, starting with l.**CharRecord**, to find
  character record c s.t. c.**CharID** = *TextRange*.**startOffset**;
`while` c is in *TextRange*
  $B \leftarrow B \cup \{(c.\mathrm{ULX}, c.\mathrm{ULY}, c.\mathrm{LRX}, c.\mathrm{LRY})\}$;
  $c \leftarrow$ next character record;
`return` B;

The algorithm performs a search for the record of the first character in the input text range, then retrieves the bounding boxes of each character in the text range. Due to the relative small sizes of folio records and line records, searching for the first character record is performed using few disk read operations. Retrieving bounding boxes for all characters in the input text range requires a number of disk readings proportional to the text range and the number of character records per disk page.

6. REMARKS AND CONCLUSIONS

The data structures described in this paper support, in different ways, the image-based XML encoding of manuscripts used in our EPT software. Our XML-segment tree data structure provides convenient support for non-hierarchical XML markup that originates from an image and lends itself to efficient implementations for IBEEs.

While segment trees are used as an all-in-one solution for manipulating image-based markup in main memory, folio R-trees are designed to store on disk information linking text to image. Folio R-trees complement traditional methods for storing XML in relational databases [21, 20, 9] and folio R-tree algorithms, such as Text/Image Range Query work in combination with XML query processing algorithms on those structures to deliver results that find XML markup associated with regions on the manuscript folio images.

7. REFERENCES

[1] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood (Eds.). Document Object Model Level 1, *W3C Consortium Recommendation*, 1998 http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[2] British Library Cotton MS Vitellius A. xv.

[3] British Library Cotton Otho A. vi.

[4] Bentley, J. L., and D. Wood. An optimal worst-case algorithm for reporting intersections of rectangles, *IEEE Trans. on Computers* C-29 (1980) 571-577.

[5] A. Bonifati, S. Ceri, Comparative Analysis of Five XML Query Languages , *SIGMOD Record*, pp. 68-79 March 2000,

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler (Eds.), Extensible Markup Language (XML) 1.0 (2nd Edition), *W3C Recommendation*, 2000, http://www.w3.org/TR/REC-xml.

[7] A. Dekhtyar, I.E. Iacob, A Framework for Management of Concurrent XML Markup, *Proc. Workshop on XML Data and Schemas (XSDM'03), LNCS*, Vol. 2814, pp. 311-322, 2003.

[8] P. Durusau, M.B. O'Donnel, Declaring Trees: The Future of the Evolution of Markup?, *Proc. Conference on Extreme Markup Languages*, 2002.

[9] D. Florescu, D. Kossmann, A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database, *INRIA Technical Report #3680*, 1999.

[10] A. Guttman, R-trees: A dynamic index index structure for spatial searching, in *Proc. ACM SIGMOD*, 1984, pp. 47-54.

[11] K.C. Hawley, K. Kiernan. An Image-Based Electronic Edition of Alfred the Great's Old English Version of Boethius's Consolation of Philosophy., in Proc. Joint International ALLS-ACH Conference, 2003.

[12] J.W. Jaromczyk, N. Moore. Geometric data structures for multihierarchical XML tagging of manuscripts, in *Proc., 20th European Workshop on Computational Geometry*, 2004.

[13] J.W. Jaromczyk, N. Moore. Geometric data structures for multihierarchical XML tagging of manuscripts. *U. of Kentucky Technical Report TR 404-04*, 2004.

[14] K. Kiernan, J.W. Jaromczyk, A. Dekhtyar, D.C. Porter, K. Hawley, S. Bodapati, I.E. Iacob, The ARCHway Project: Architecture for Research in Computing for Humanities through Research, Teaching, and Learning, forthcoming in *Literary and Linguistic Computing*,2004.

[15] K. Kiernan, Digital Facsimiles in Editing: Some Guidelines for Editors of Image-based Scholarly Editions, forthcoming in *Electronic Textual Editing*, J. Unsworth, K. O'Brien O'Keeffe, and L. Burnard (eds.), 2004.

[16] K. Kiernan, A. Prescott, E. Solopova, D. French, L. Cantara, M. Ellis (Eds.), C.J. Yuan, Electronic Beowulf, *British Library Publications, London*, 1999, 2003,http://www.uky.edu/~kiernan/eBeowulf/guide.htm.

[17] Preparata, F. P. and M. I. Shamos. *Computational Geometry: an Introduction*. New York: Springer-Verlag, 1985.

[18] Samet, H.. Multidimensional Data Structures. In *Algorithms and Theory of Computation Ha ndbook*, ed. M. J. Atallah. Boca Raton: CRC Press, 1999.

[19] C.M. Sperberg-McQueen, L. Burnard, Text Encoding Initiative (TEI) Guidelines, *TEI Consortium*, 2001, http://www.tei-c.org/P4X/

[20] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, J. F. Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities, in *Proceedings, VLDB*, pp. 302-314, 1999.

[21] F. Tian, D.J.DeWitt, J. Chen, C. Zhang, The Design and Performance Evaluation of Alternative XML Storage Strategies , *SIGMOD Record*, Vol 31, No. 1, pp. 5-10, March 2002.