Access Support Tree & TextArray: A Data Structure for XML Document Storage & Retrieval

Dieter Scheffner Johann-Christoph Freytag

Department of Computer Science
Humboldt-Universität zu Berlin, Germany
{*scheffne* | *freytag*}*@dbis.informatik.hu-berlin.de*

Abstract

The characteristics of XML documents require new ways of storing and querying such documents. Queries on both textual content and structural aspects must be supported efficiently. For this reason, we examined existing work on both document storage approaches and models for querying documents deriving requirements that are essential for the storage of XML documents. As a result of our study, we designed the Access Support Tree and TextArray (AST/TA) data structure. The important idea of the AST/TA data structure is the separation of the (logical) structure of a document from its "visible" text content, which is represented as a single contiguous string, at the same time providing a tight integration to guarantee consistent changes. We introduce the AST/TA data structure formally by its abstraction, namely the AST/TA model and compare requirements of our AST/TA approach with those found in current work. Finally, we demonstrate the superiority of the AST/TA model by describing those operations that take advantage of the design principles of the AST/TA data structure.

1. Introduction

XML has become widely accepted for both data representation and exchange of information over the Internet. The amount of such data is rapidly growing. Thus, the demand increases for systems that are able to manage large numbers of such data which greatly vary in structural and access-related requirements in an efficient and reliable way. Since XML is a language to express semi-structured data, it is hard for conventional DBMSs to handle any kind of XML documents efficiently. In conventional DBMSs such data are mapped to physical storage layout that is designed for structured data in the first place, making query processing inefficient. For this reason, we decided to design a management system from scratch on basis of a data structure that is suitable for providing both efficient operations on XML documents and more flexibility for many applications.

In this paper, we introduce the first step of our ongoing research on physical storage implementation for XML documents, namely the Access Support Tree (AST) and TextArray (TA) data structure. It has already been implemented in main memory and provides our base concept for maintaining XML documents on persistent storage. We intend to build our XML query execution engine (*XEE*) based on the AST/TA data structure designed for persistent storage.

For convenience, we introduce an abstraction of our data structure, which we refer to as the *AST/TA model*. The AST/TA model takes advantage of merging well-known and established concepts from different fields in order to support search and update operations on large collections of XML documents, e.g., web sites of shops etc. The requirements for the physical storage design are mainly influenced by the requirements of our *XEE* system as follows:

(1) Query evaluation is supported by integrating both the concept of database query languages and the concept of information retrieval, thus supporting such concepts of, e.g., the Information Retrieval Query Language (IRQL) [3] and the Proximal Nodes model [6].

(2) The idea of separating the layout from the structure and content of a document is extended to the separation of structure and content, thus taking over the basic DBMS concept of partitioning data and meta data.

(3) The structure of documents to be stored is not necessarily constrained by any schema, i.e., it must be possible to store such *generic* documents even if they do not come with any DTD.

(4) Efficient operations on documents must be supported, especially while updating the structure and the content of documents.

Our goal was to support information retrieval and structural queries equally, and to accomplish a separation of structure and content. Thus, we decided to keep the entire "visible" text content of an XML document as a single contiguous string maintaining the original order of

the text in the document. The ”visible” text content is what a user may have in mind when formulating a query. Consequentially, the text content is neither fragmented nor interspersed with markup. Consider the following sample query put to an electronic shop: ”*Retrieve documents containing* `Our Price: $11.96`”. Assuming that the price is tagged with markup, e.g., `... Our Price: <price>$11.96</price> ...`, in our desired XML document, we recognize the following advantages: (1) As the storage representation is not stained with markup, the search string matches directly the presentation string—no filtering is needed. Furthermore, the distances in the ”visible” text of documents match to their distances on storage. (2) By not fragmenting the text content, we need at most one access to fetch the complete string ”`Our Price: $11.96`” from storage. Consequently, we are able to implement efficient search and browse operations on the ”visible” text.

In addition, queries on document structure only, e.g., *”Are there any priced books?”*, may be handled without looking at the text content at all. We experienced that the predominant part of XML documents—far more than 50%—consists of text content only. Thus, the separation of structure and content makes it possible to present the navigation structure of documents more densely. That is, with respect to an implementation of this concept on persistent storage, we expect to map document structures to much less pages than complete documents require. For this reason, navigating document structures becomes more efficient, due to shorter access paths within the page presentation of such structures.

By maintaining (unstructured) text content as a single contiguous string, we are able to provide operations that refine the structure of a document in a state of flux. That is, we add new ”markups” to the structure rather than we insert real markups into the text content of the document. For example, a shop administrator realizes that it might reasonable to make the price of a product explicitly available to customers by tagging it. Such a change only affects the structure of the document. Thus, the impact on the overall system is much smaller. Technically, this means, our approach supports *tag insertion* which is addressed as the *tag insertion problem* and is an important part of text mining [18].

Another advantage of referring to the text content as a single contiguous string is that we may build multiple or independent hierarchies [6] of logical structures on top of the same text content. For example, when we consider the sentence ”`Max wears a hat.`”, we may add structure as in `<txt><name>Max</name>wears a hat.</txt>` or as in `<txt><subj>Max</subj><pred>wears</pred><obj>a hat</obj>.</txt>`.

Our data structure also supports different views of text efficiently, i.e., text as a sequence of characters and as a sequence of words [10]. Considering text as a sequence of characters is a very simple and flexible way for manipulating and accessing texts. Nowadays, well-known representatives of such texts are descriptions of, e.g., protein sequences in molecular biology. However, for natural languages, such as German or English, it is advantageous if we consider text as a sequence of words, because we may avoid repetition of characters which only indicate word boundaries and layout. For this purpose, we take into account text normalization similar to the normalization in PAT [8], thus facilitating a more efficient access to words.

Eventually, our approach may be useful for replacing or supplementing existing storage structures, respectively. The latter alternative means, that those parts of XML documents that conventional systems cannot handle efficiently are stored in our data structure. That is, the AST/TA data structure might be plugged into such systems as an ”XML data type”.

The concept we use in our approach, i.e., considering the text content as contiguous string and representing the logical structure of documents in a separate hierarchy, has already been introduced earlier. To our best knowledge, however, this was done only twice, that is, in the framework of the *Structured Multimedia Document DBMS* (*SMD DBMS*) [2] and in the *Proximal Nodes* model [6]. With regard to SMD DBMS, only few implementation issues are discussed in [7]. The Proximal Nodes model is, according to its authors, a purely logical model for querying document databases. Therefore, it does not mandate any specific implementation. Neither of these approaches refers to XML documents directly. Hence, we are convinced, it is worthwhile to investigate deeper implementation details based on this concept, in particular with regard to XML documents.

2. Related Work

XML or SGML documents are often considered to be objects and thus stored and managed by OODBMSs. Alternatively, documents are ”forced” into relational or object-relational DBMSs by managing documents as data in one or more tables. Moreover, we experienced that, according to the *ANSI/X3/SPARC* architecture model [11], most of these approaches more or less rely on the conceptual level rather than on the physical level, thus leaving the physical organization to the DBMS.

In this section, we briefly examine storage and management concepts of documents in the *XML Extender* (IBM® DB2® Extenders™) [4], in *HyperStorM* (Hypermedia Document Storage and Modeling) [12], in the project of *Structured Multimedia Document DBMS* [2], and in NATIX (Native XML Repository) [5].

XML Extender. Based on the DB2 object-relational DBMS, the XML Extender manages XML documents in two different ways, namely as *XML Column* and as *XML Collection*. XML Column supports the storage of complete and marked up documents in a single table column. *User Defined Functions* give access to documents and parts of documents. To search the structure of a document efficiently, the user must create indexed side tables referring to well-chosen elements and element attributes of the document. In contrast to an XML Column, an XML Collection is based on the idea of "dismantling" documents. Elements or element attributes are mapped into columns of one or more tables. The text content of elements and the values of element attributes become table values. *Document Access Definitions* take care of the structural glue of documents. Dismantling and reconstructing documents is accomplished with the help of *Stored Procedures*.

HyperStorM. The objective of the HyperStorM project was to build an application database framework for storing structured documents [12] in a system coupling an object-oriented DBMS and an information retrieval system. In this approach we distinguish three strategies to represent SGML documents in an object-oriented database: (1) a completely structured database-internal representation of documents, i.e., each logical document component corresponds to a database object, (2) documents are stored as BLOBs in the database, and (3) the hybrid approach of (1) and (2), i.e., some "non-flat" elements are represented by individual database objects in an object hierarchy while "flat" elements represent parts of the document in their native form (text interspersed with markup). It is an administration task to decide which element becomes a "flat" or a "non-flat" element.

Structured Multimedia Document DBMS. Within the framework of the Structured Multimedia Document DBMS (SMDDBMS), an object-oriented MMDBS was developed for storage of SGML documents [2] in the presence of DTDs. The text content of documents is stored as a contiguous text string as a whole rather than being fragmented [7]. The logical structure of a document is represented by an object hierarchy of which the objects that refer to text content reference to "their text" with so-called *annotations*. An annotation is a logical reference that indicates the first and the last character of the substring to which the concerning object refers.

NATIX. NATIX is a repository for the storage and management of large tree-structured objects, preferably XML documents [5]. Essentially, the idea of NATIX is to map the logical structure of an XML document directly into the corresponding physical structure. Unlike the logical structure, the physical tree structure is equipped with additional nodes that help manage large trees, such that the tree might be split up among several pages in storage. Since the materialized tree is the direct mapping of the logical structure of the document, the text content is fragmented and is interspersed with structural data like element and attribute names, etc. NATIX may store generic documents that do not depend on any schema.

Comparing Approaches. Table 1 summarizes the properties of the approaches introduced in the previous paragraphs and compares them with the requirements for our approach.

3. The Access Support Tree/TextArray Model

3.1. Motivation

The AST/TA model is the abstraction of the AST/TA data structure that is relevant and is used for representing XML documents in physical storage. The AST/TA model describes and summarizes the components of XML documents and their relationship to each other with regard to their storage representation rather than to their conceptual models as provided by query models like the *Document Object Model* [13], the *XQuery 1.0 and XPath 2.0 Data Model* [17] or the *Proximal Nodes Model* [6]. The AST/TA model provides a base concept for maintaining XML documents on persistent storage. Based on the concept of an persistent AST/TA model, we intend to implement the DOM, the XQuery 1.0 and XPath 2.0 Data Model, or the Proximal Nodes model on secondary storage.

Well-formed XML documents represent the actual instances of documents. They actually consist of only an XML **[document element]**[1] each. XML declarations and XML document type declarations are optional and provide additional information constraining XML document elements. Since *XML Schema* [16] provides means for specifying schemata in the form of XML documents themselves, XML document type declarations may be reduced to just references to other XML documents. The AST/TA model is designed to support *generic* XML documents, i.e., documents having no constraints other than well-formedness. For this reason, the AST/TA model devotes to XML document elements only.

3.2. The static AST/TA Model

An important goal of our approach is to separate meta data from data. Therefore, we distinguish the Access Support Tree from the TextArray of a document element. The

[1] We use the notation introduced in XML Information Set specification [15].

Table 1. Overview of criteria and approaches

criterion	XML Extender		HyperStorM	SMD DBMS	NATIX	AST/TA
	XML Column	XML Collection				
support of update operations	✓	✓	✓	✓	✓	✓
contiguous text content	✓	–	to a degree (hybr. approach)	✓	–	✓
storage of text without markup	–	✓	to a degree (hybr. approach)	✓	✓	✓
separation of doc. structure & content	–	✓	–	✓	–	✓
support of cont.-based search	✓	–	✓	✓	–	✓
support of generic document storage	✓	–	–	–	✓	✓
support of text normalization	–	–	–	–	–	✓
operational support of text mining	–	–	–	–	–	✓

✓ : criterion applies to the approach. – : criterion does not apply.

entire logical structure of the document element, including element attributes, comments etc., is incorporated in an ordered tree—the AST—whereas the TA keeps the entire text content as a single contiguous string with regard to its original order in the document. Thus, the XML document element is represented by an AST/TA pair in physical storage.

The relationship between AST and TA is established by logical references called *text surrogate values*. A text surrogate value is a vector indicating the start position and the length of a text segment referenced to in a TA. Every vertex of the logical structure receives a text surrogate value, thus text segments might be reached from any vertex and ASTs become indexes. If we apply this linking concept to documents that do not change their text content in length, text surrogate values alone might be sufficient. We introduce *offsets* that adjust the text surrogate values after updates changing the length of the text in TAs. Hereby, the number of vertices to be modified is kept as small as possible. Otherwise, all values that span or follow the location of update in the logical structure would have to be modified, making such updates inefficient. The offset concept is also applied to text surrogate values by specifying the length rather than the end position of text segments. That is, when updating a surrogate value, in most cases, either the start position or the length must be updated only.

An AST/TA Example. Figure 1 shows an example of an AST/TA pair with its corresponding document element. This example is based on a *normalized* TA. The symbols ␣ and # act as substitutes for word separators. We address the normalization of TAs in the next section.

The sample AST consists of seven vertices (r, ..., x) representing their corresponding document components, i.e., elements, a comment, and text components. For example, vertex r labeled with `header` refers to the document element itself. Vertex t represents a comment. We label such vertices with their textual content only, e.g., `check year`, leaving out syntactic additives as `<!--` and `-->`. On the contrary, text vertices such as v and x have no labels. They act as place holders for the text components they represent in the AST tree. In addition to labels, element vertices may have an attribute set assigned to. Thus, vertex s (`author` element) comes with the refinements of `from="1832"` and `to="1908"` representing the attributes `from` and `to` with their corresponding values `"1832"` and `"1908"`.

Edges (solid lines) link vertices of the tree. They refer to the immediate part-of relationship of the document components. For example, the `author` element is a direct part (child) of the `header` element. Thus, the AST tree matches the logical structure of the document element.

Expressions in parenthesis, e.g., `(14,14)`, depict text surrogate values. That is, the vertices u and x both refer to the text segment that starts at position `14` in the TA and has a length of `14`. Arrows give a notion of text surrogate values in the figure pointing to the start of corresponding text segments in the TA.

For an efficient implementation of update operations, all vertices do have a defined start position in their surrogate values, thus aligning vertices to text segments. Therefore, the comment vertex t has the defined start position of `14`

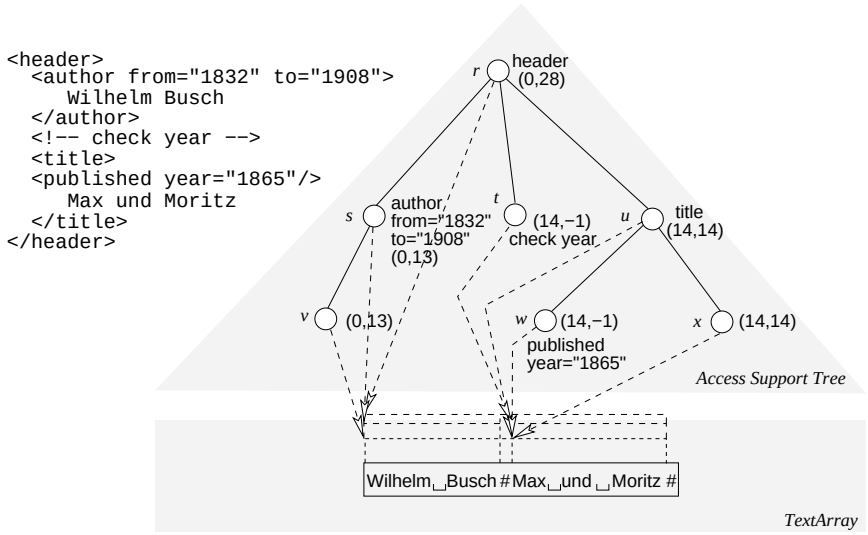

Figure 1. AST and normalized TextArray

in its surrogate value. Vertices representing processing instructions and comments and vertices that stand for *true empty elements* acquire a length of -1 in their surrogates. Addressing the problem of empty elements, we must distinguish elements that do not span over any text content in a TA—*true empty elements*—and "PCDATA elements" that refer to empty word content and, therefore, acquire a length of 0 in their surrogates. Providing these different measures of length, the AST model satisfies the different semantics of such elements.

We left out the offsets in our example, since we assume the document not being updated yet. Therefore, all offsets are equal to 0.

Definitions. The Definitions 1, 2, and 3 summarize our AST/TA model. The identifiers we use in our definitions, namely `element`, `PI`, `Comment`, `CharData`, `CDSect`, `EntityRef`, `Attribute`, and `Name` refer to the left-hand sides of the corresponding productions in the *XML Standard* [14]. As character references are only used for overcoming limitations of the character encoding of documents and for "escaping" characters that would be otherwise interpreted as markup, we postulate character references to be expanded. Furthermore, we assume that validating XML parsers expand entity references, whenever it is possible. Otherwise, entity references are handled in their native form, e.g., `&entity-ref;`, as "normal" text of type #PCDATA. In Def. 3 we use the term *textual content* to refer to processing instructions and comments leaving out the syntacticle additives `<--`, `-->`, `<?`, and `?>`.

Formal definitions of our AST/TA data structure and its relationship to the well-known *XML Information Set* [15] can be found in [9].

Definition 1 (TextArray) *A TextArray is the contiguous sequence τ of characters in document order that represents all parts of an XML document referring to the type set T_{PCDATA} := {`CharData`, `CDSect`, `EntityRef`}. τ_i is the i^{th} character of τ, where $i \in \{0, \ldots, length(\tau)-1\}$.*

The following properties are valid for TextArrays: (1) all character references are expanded in TextArray segments that refer to the type of `CharData`*, (2) CDATA sections are stored as specified by the type of* `CDSect` *leaving out the syntactic additives* `<![CDATA[` *and* `]]>`*, and (3) entity references are stored as specified by the type of* `EntityRef` *referring to their native form.* □

Definition 2 (Normalized TextArray) *A normalized TextArray is a TextArray with the following additional properties. Sequences of white spaces (*`#x20`*,* `#x9`*,* `#xD`*,* `#xA`*) are reduced to a single space (*`#x20` – word separator*) or they are removed if they appear next to markup boundaries or next to the beginning of the TextArray. Markup boundaries are represented by the character* `#x0` *(*separator*). We may use the character* `#x0` *for separators, since it is not part of any XML document [14]. For implementation reasons, we make markup boundaries explicit at the end of the TextArray, however, not at its beginning.* □

Definition 3 (Access Support Tree) *An Access Support Tree is an ordered tree $(\mathcal{V}, \mathcal{E})$. A vertex $v \in \mathcal{V}$ represents the corresponding structure component of an XML document element. An edge $e \in \mathcal{E}$ expresses the relationship between* parent *and* child *vertices in $\mathcal{V}$, thus $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$.*

Siblings of one parent vertex are ordered with respect to a successor *relation for any two neighbor siblings. Therefore, they may be counted from left to right starting with 0.*

Every vertex has a type $\vartheta \in T$ assigned to it. We may separate T in two subsets: $T := T_{labeled} \cup T_{PCDATA}$ with $T_{labeled} := \{$`element`*, *`PI`*, *`Comment`*$\}$ and $T_{PCDATA} := \{$*`CharData`*, *`CDSect`*, *`EntityRef`*$\}$.*

There is exactly one root *vertex of type* `element`*. All vertices, except the* root *vertex, have a parent vertex of type* `element`*.*

Every vertex of type $\vartheta \in T_{labeled}$ receives a label $\lambda \in L$ representing the `Name` *or the* textual content *of the corresponding component. Vertices of type $\vartheta \in T_{PCDATA}$ carry the empty word ε as label. Vertices v of type* `element` *may have a set of corresponding* `Attributes` *a_v, including their names and values, assigned to it.*

Every vertex obtains a text surrogate value $\sigma \in \mathbb{N}_0 \times \mathbb{Z}$. Text surrogate values are vectors consisting of the two components σ_p and σ_l, which refer to the start position *and the* length *of the text segment in a TextArray referenced to by a vertex. Thus, text surrogate values are logical references. Vertices of type $\vartheta \in T_{PCDATA}$ reference their corresponding character sequence in the TextArray that they represent in a document. Vertices of type* `element` *reference the text segment that is the "concatenation" of all text segments to which their descendant vertices of type $\vartheta \in T_{PCDATA}$ reference.*

Offsets $\omega \in \mathbb{Z}$ adjust the position *σ_p of a text surrogate value, such that vertices reference their text segments via their text surrogate values correctly after updates.* □

In the following, we discuss issues of integrating our proposed text normalization into the AST/TA model. For the integration of other relevant XML features such as handling of entities and XML namespaces, we refer to our technical report [9].

Text Normalization. It is easy to handle text content as a sequence of characters, because the text remains in its original form. However, when considering text content as a sequence of words, we must be aware of *separators* and *word separators*. If we disregard separators, words would merge in a TA. The following example shows this effect—the symbol "␣" indicates single word separators:

```
Instance:   <author>Wilhelm Busch</author
            ><title>Max und Moritz...
TextArray:  Wilhelm␣BuschMax␣und␣Moritz...
```

`Busch` and `Max` merged into `BuschMax` within the TA, thus `Max` may not be recognized as an independent word any longer.

Furthermore, we have to take into consideration different semantics of separators and word separators in a TA, if we intend to remove character sequences that cross markup boundaries. Let us take a look at the following example—for better reading, the symbol "#" replaces the `#x0` character here, indicating separators:

```
Instance:   <author>Wilhelm Busch</author
            ><title>Max und Moritz...
TextArray:  Wilhelm␣Busch#Max␣und␣Moritz...
```

There is no problem in removing the string `helm␣Bu`, since the removal happens directly within the `author` element. However, if we want to remove, e.g., `sch#Ma`, that crosses markup boundaries, we expect the markup boundary to be maintained:

```
Instance:   <author>Wilhelm Bu</author
            ><title>x und Moritz...
TextArray:  Wilhelm␣Bu#x␣und␣Moritz...
```

The separator must remain in the TA as long as the markup `</author><title>` remains part of the instance.

To handle these and other challenges which notably arise in text normalization, we encapsulate the access to TextArrays on behalf of both views of text in our implementation. Thus, we provide a generic interface for the operations that the AST/TA model specifies.

3.3. Operations of the AST/TA Model

The AST/TA model provides operations with respect to both the *Data Definition Language* and the *Data Manipulation Language* similar to DBMSs. The AST/TA model provides operations both to *generate* an AST/TA pair from an XML document and to *restore* an XML document from an AST/TA pair. The AST/TA DML includes operations such as *search*, *insert*, and *delete* that primarily work either on the AST, on the TA, or on both, respectively.

The DDL operations. We assume an XML processor to *generate* both the AST and the TA from an XML document at the same time. Furthermore, we expect an XML processor to do any necessary normalization, e.g., the carriage-return and line-feed and the attribute-value normalization described in the XML Standard [14]. Additionally, the processor may perform the text normalization we proposed in Definition 2. During the process of generating an AST/TA pair, all character references are expanded; if schema information is available, entity references are expanded as well. Beyond that, text surrogate values are calculated and all offsets are set being equal to 0.

The reverse operation of generating AST/TA pairs is *restoring* XML documents from AST/TA pairs. This process is mainly ruled by the serialization of the tree structures coming up with AST/TA pairs. With performing different kinds of normalization while generating AST/TA pairs, the restored XML document differs from its original source.

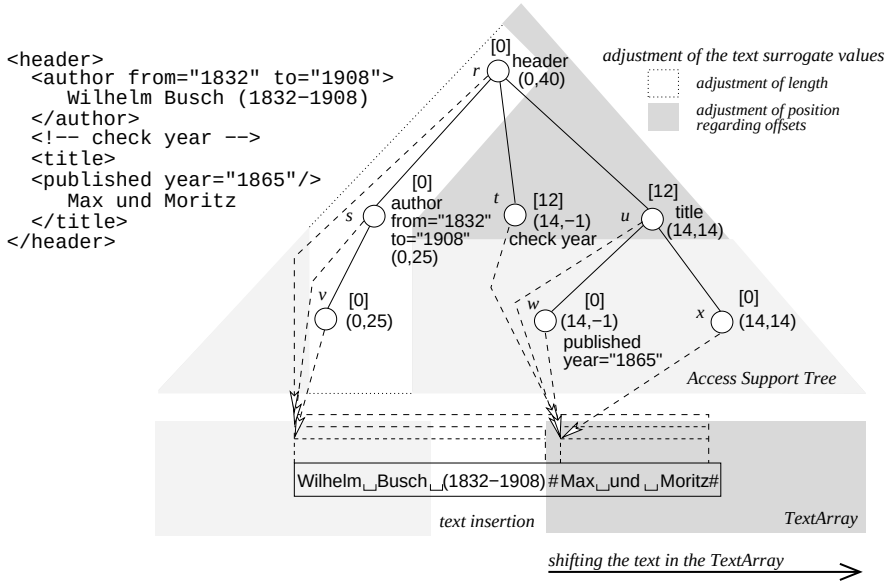

Figure 2. Inserting text into the TextArray

Manipulating TextArrays. Searching the TextArray means retrieving an arbitrary sequence of characters or words from the "visible" text content of an XML document. The TA enables random access to arbitrary segments of the text content. For this purpose, we need to know the length of the desired text segment and its start position in the TA. Alternatively, the TA allows to scan its text.

By representing the text content of an XML document as a single contiguous string, the TA provides efficient operations both for random access to and for scanning the text. As we do not need to rearrange the text, random access is performed in constant time, whereas scanning needs time depending on the length of the TextArray.

The TA facilitates insertions and deletions of arbitrary sequences of characters or words respectively, to change the text content of an XML document. Finding an insertion or deletion position is done in constant time. However, inserting or deleting a sequence of characters or words may take much more time, because all text that follows the corresponding position in a TA must be shifted, i.e., $length(\tau)/2$ bytes on average. To improve the performance, we implement TAs as B*-trees as, e.g., described by Carey et al. in [1]. Insertions into or deletions from normalized TAs are additionally more complex, because the normalized state of those TAs must be maintained.

Deletions and insertions are similar to each other. In contrast to insertions, we must specify a valid deletion length referring to TAs. For example, we may delete up to 20 characters beginning at position 7 in the TA of Figure 1. If we delete 14 characters, the remaining TA is `Wilhelm#Moritz#`. Again, normalized TAs must be left in a normalized state afterwards. That is, if we specify deleting 13 characters, 14 characters must actually be deleted; otherwise, the TA is in an ill-formed state: `Wilhelm#␣Moritz#`. The separator sequence "#␣" violates the normalization. Adjusting the deletion length is integrated into our deletion algorithm.

After changing the length of TAs, it is necessary to adjust text surrogate values and offsets in ASTs. Figure 2 shows an example in which the text "`(1832-1908)`" was inserted after the word `Busch`. The example refers to a normalized TextArray. Thus, we must insert an additional space at the beginning of the text insertion causing an effective insertion length of 12. With respect to the example in Figure 2, we perform the AST adjustment as follows: We start in the root vertex r and increment its text surrogate length by 12. In the next step, we search for the child vertex of r that spans over the text insertion and increment its length also by 12. The offsets of all siblings vertices (t and u) that follow to the right of that child vertex (s) are incremented by the insertion length of 12. (In Figure 2 we use square brackets to constitute offsets, e.g., `[12]`.) This procedure continues until we reach the text vertex v (the leaf) that directly spans over the insertion. By using offsets rather than none to adjust text surrogate values, we do not need to visit all of the vertices that follow the location of an update within an AST. Thus, if C is the maximum number of child vertices and d is the maximum depth in an AST, we must visit no more than $C \times d$ vertices, using this approach.

As for deletions, updates of offsets and text surrogate values are analogous to updates after insertions. However, we need a more sophisticated algorithm when deleting char-

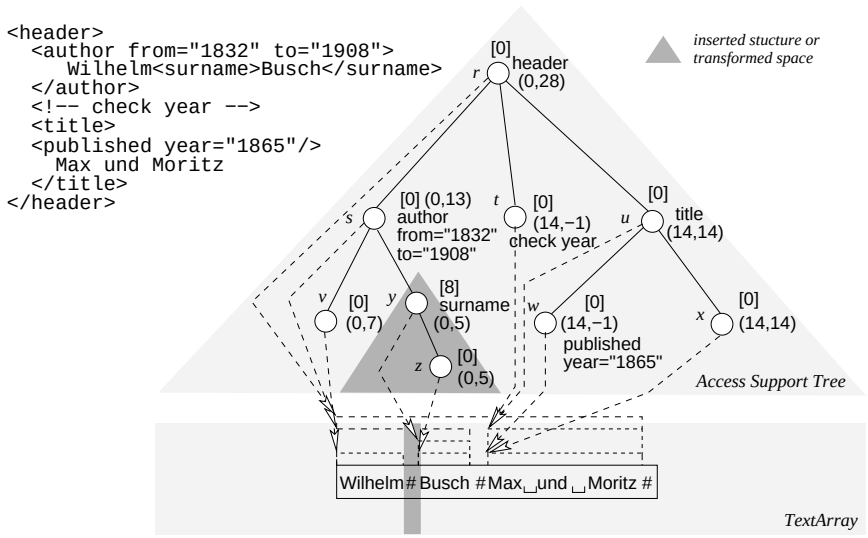

Figure 3. Inserting structure into the AST

acter or word sequences that go beyond markup boundaries. Unlike at insertions, the adjustment procedure for ASTs may reach a non-leaf vertex of which some children span only over parts of the deleted text. Therefore, this subtree must be traversed to adjust text surrogate values and offsets of the vertices according to their contributions.

Manipulating ASTs. The manipulation of ASTs refers to the structure of XML documents. For searching the structure of a document, the AST provides operations navigating the tree to find vertices having specific characteristics. For example, we are able to search for vertices with specific element names, attribute names, attribute values, specific positions to siblings, or references to specific text segments according to their corresponding text surrogate value and offsets.

Insertions referring to ASTs are operations that refine the structure of documents only, therefore enhancing the structure of ASTs. For example, we might apply some refinement to our sample document, such that `<author from="1832" to="1908" >Wilhelm Busch</author>` becomes `<author from="1832" to="1908">Wilhelm<surname>Busch</surname></author>`. Such an insertion of a logical structure into a document is a *tag insertion* that might be the result of a text mining task [18]. Figure 3 illustrates the previous example based on a normalized TextArray. The AST receives two additional vertices y and z with y of type `element` representing the new `surname` markup. Vertex y is the parent of z which references the refined text segment `Busch`. Since the underlying TA is normalized, the former word separator (␣) prefixing the word `Busch` must be converted into a separator (#), making the new markup boundary explicit in the TA. This is not necessary with non-normalized TAs, because there are no separators to consider.

In the following, we focus on ”simple” structural insertions as in our example that refine texts referenced by single text vertices rather than higher level structures in ASTs. For these simple insertions, we perform the following steps: After having found our desired sequence of characters or words that is to be marked by a new tag, we know the position and the length of this sequence. We search the AST—beginning from the root—for the text vertex that spans over this sequence. As all vertices of an AST come with text surrogate values, we follow exactly one path to the target vertex. This text vertex must be ”split” in such a way that the AST integrates its additional structure correctly. That is, the root vertex of the new structure becomes a sibling of this text vertex and the text surrogate values and offsets must be adjusted accordingly. With respect to Figure 3, the text vertex v receives a new text surrogate length of 7 and the offset of the root vertex y is set to 8, thus aligning all text surrogate values of the newly inserted structure.

We note that structural insertions do not have any impact on the overall AST; changes in the AST are rather limited to the location of insertion. Beyond that, if TextArrays hold sequences of characters rather than of words, there is no need to access TAs at all.

We also allow the user to delete structure from an AST. That means, vertices may be removed from ASTs. Removing a vertex is simple. All children of the vertex to be removed become children of its grand parent. Only, if the corresponding TextArray is normalized, separators might have to be exchanged by word separators.

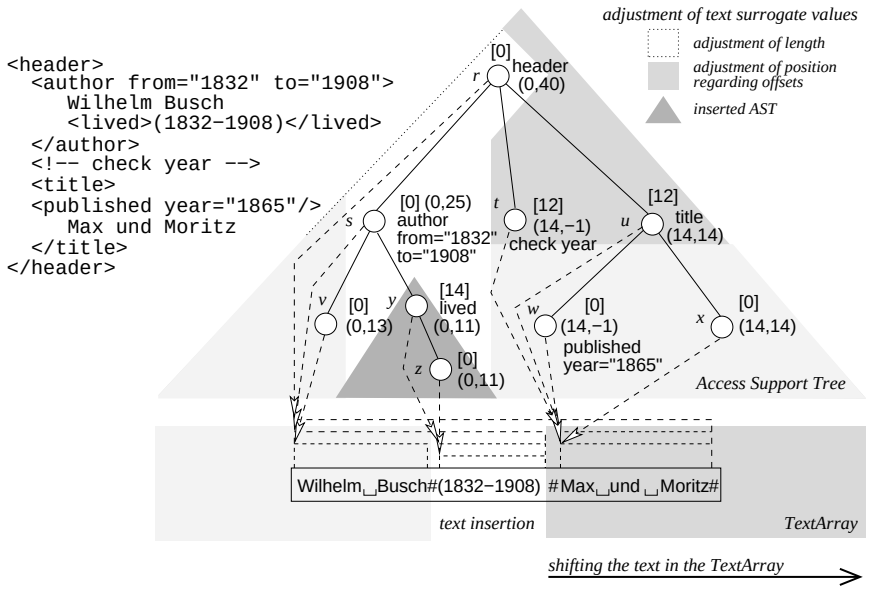

Figure 4. Insertion of an AST/TA pair into an AST/TA pair

Structural updates of ASTs also involve updates of element names, attribute names, attribute values etc. These operations are simple operations mainly based on structural search, therefore we do not address them here.

Manipulating AST/TA pairs. In the previous sections, we considered the manipulation of ASTs and TAs separately. We now focus on manipulating ASTs and TAs jointly.

A simple operation on AST/TA pairs is the search for the minimal enclosing element of an arbitrary text segment of a TA. For example, in Figure 1, the minimal enclosing element of the string `Busch#Max` is the `header` element. Based on text surrogate values, this search needs to follow exactly one path from the root vertex to the desired element vertex in the AST. If C is, again, the maximum number of child vertices and d is the maximum depth in an AST, at most $C \times d$ vertices must be processed.

Inserting an AST/TA pair into another AST/TA pair is equivalent to inserting one XML document into another. The position of such insertions depends on different characteristics of the AST/TA pair in which to insert. On the one hand, insertion positions may depend on text positions. For example, consider the query: "*Insert* `<b>y</b>` *into* `<a>xz</a>` *after* `x`" that results in `<a>x<b>y</b>z</a>`. On the other hand, insertion positions may depend on structural properties as in the query: "*Insert* `<b>y</b>` *into* `<a>xz</a>` *as the first child of* `a`" resulting in `<a><b>y</b>xz</a>`.

Figure 4 describes an example of the latter type of insertions. Here, the AST/TA pair representing the XML element `<lived>(1832-1908)</lived>` is inserted as last child into vertex s (`author` element). The algorithm for non-normalized TextArrays works as follows: First, the vertex s in which the insertion is to occur is searched. During the search the length parameter of the surrogate values of r and s and the offsets of the siblings that follow the search path are incremented by the length of the additional text content "`(1832-1908)`". Finally, the root vertex y of the AST to be inserted is placed as the last child of vertex s; the new text is added to the TA. This kind of insertion needs to pass through the path from the root to the destination vertex exactly once.

Unfortunately, the insertion requires a second walk through the search path, when we insert into AST/TA pairs that are built on normalized TAs as shown in Figure 4. We do not know the real insertion length in the TA at the beginning; it might change, because of considerations of separators. Thus, we cannot update the text surrogate values while searching the destination vertex. Therefore, the corresponding text surrogate values and offsets must be updated after the insertion of text into the TA requiring a second walk through the search path.

As for deletions, we provide operations on AST/TA pairs similar to insertions.

4. Future Work

Currently, we have implemented the AST/TA data structure in main memory. We carefully review the interdependence between the AST and the TA for possible improvements. For example, the AST/TA data structure relies on relative text surrogate values, which enable efficient update operations. However, we need absolute text surrogate val-

ues for vertices in ASTs to reference text segments in TAs correctly. For this reason, we must take into account the issue of providing absolute text surrogate values for arbitrary vertices of ASTs in an efficient way. Another issue of interest is to provide a suitable interface that enables convenient access to XML documents represented by AST/TA pairs.

Currently, we move our main memory implementation of AST/TAs into secondary storage. Thus, we provide persistent AST/TAs for handling large scale XML document collections within our XML query execution engine (*XEE*). In the framework of *XEE*, we intend both to implement a persistent DOM based on persistent AST/TAs and to map query languages to the AST/TAs. As for the mapping of query languages, the integration of both the concept of database query languages and the concept of information retrieval data plays an important role. In addition, we take a look at optimizations with respect to AST/TAs and plan, therefore, to apply various kinds of indexes to both the AST and the TA to improve the performance. For this purpose, we have already proposed some indexes in [9], e.g., for XML namespaces, entities etc.

5. Conclusion

The AST/TA model introduces our approach for storing and retrieving *generic* XML documents without any DTD. The basic idea of the AST/TA data structure is the separation of meta data from data. This resulted in the design of two independent structures of the Access Support Tree for taking the meta data and of the TextArray for taking the data of XML documents. Moreover, TextArrays maintain the data in ”one piece” in document order. Based on this concept, our AST/TA data structure enables efficient access to documents based on both content and structure, thus facilitating information retrieval and database-like querying equally. Although we pursue a generic approach, among other things, of supporting different views of text, we are able to provide the AST/TA data structure with efficient update operations. For example, the number of vertices processed in an textual insertion is reduced considerably by using offsets for text surrogate values.

We addressed relevant aspects of our approach referring to XML. In addition, our approach incorporates further aspects specific to XML that must not be ignored, e.g., the XML Information Set [15], white spaces handling, and handling of XML namespaces and entities. Due to space limitations, we could not address such issues here; hence, we refer to [9]. The AST/TA data structure immediately supports some of these aspects, namely by our proposed scheme to handle processing instructions and comments and by the text normalization we proposed. Furthermore, the AST/TA data structure facilitates additional aspects of interest, e.g., tag insertions.

References

[1] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of VLDB, Kyoto, Japan*, pages 91–100, 1986.

[2] Database Systems Research Group (University of Alberta). *Multimedia Data Management*. `http://www.cs.ualberta.ca/~database/multimedia/multimedia.html`, 1998.

[3] A. Heuer and D. Priebe. IRQL – Yet Another Language for Querying Semi-Structured Data? Technical Report Preprint CS-01-99, Universität Rostock, 1999.

[4] International Business Machines Corporation. *XML Extender (Administration and Programming)*. IBM, 2000.

[5] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proceedings of ICDE, San Diego, California*. IEEE Computer Society, 2000.

[6] G. Navarro and R. A. Baeza-Yates. Proximal Nodes: A Model to Query Document Databases by Content and Structure. *Information Systems*, 15(4):400–435, 1997.

[7] M. T. Özsu, D. Szafron, G. El-Medani, and C. Vittal. An Object-Oriented Multimedia Database System for a News-on-Demand Application. *Multimedia Systems*, 3(5-6):182–203, 1995.

[8] A. Salminen and F. W. Tompa. PAT Expressions: An Algebra for Text Search. *Acta Linguistica Hungarica*, 41(1-4):277–306, 1992-93.

[9] D. Scheffner. Access Support Tree & TextArray: Data Structures for XML Document Storage. Technical Report HUB-IB-157, Humboldt Universität zu Berlin, 2001.

[10] F. W. Tompa. Views of Text. Digital Media Information Base (DMIB ’97), November 1997.

[11] D. Tsichritzis and A. C. Klug. The ANSI/X3/SPARC DBMS Framework Report of the Study Group on Dabatase Management Systems. *Information Systems*, 3(3):173–191, 1978.

[12] M. Volz, K. Aberer, and K. Böhm. An OODBMS-IRS Coupling for Structured Documents. *Data Engineering Bulletin*, 19(1):34–42, 1996.

[13] World Wide Web Consortium. Document Object Model (DOM) Level 2 Core Specification, Version 1.0. Technical Report REC-DOM-Level-2-Core-20001113, W3C, November 2000.

[14] World Wide Web Consortium. Extensible Markup Language (XML), Version 1.0 (Second Edition). Technical Report REC-xml-20001006, W3C, October 2000.

[15] World Wide Web Consortium. XML Information Set. Technical Report REC-xml-infoset-20011024, W3C, October 2001.

[16] World Wide Web Consortium. XML Schema Part 1: Structures. Technical Report PR-xmlschema-1-20010330, W3C, March 2001.

[17] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. Technical Report WD-query-datamodel-20010607, W3C, June 2001.

[18] S. Yeates and I. H. Witten. On Tag Insertion and its Complexity. In *Proceedings of PRICAI 2000: International Workshop on Text and Data Mining, Melbourne, Australia*, pages 52–63, 2000.