Access Support Tree & TextArray: Data Structures for XML Document Storage

Dieter Scheffner
Department of Computer Science
Humboldt-Universität zu Berlin, Germany
scheffne@dbis.informatik.hu-berlin.de

- TECHNICAL REPORT -

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. An important result of this study is the design of the Access Support Tree and TextArray (AST/TA) data structures. The basic idea of the AST/TA data structures is the separation of the (logical) structure of a document from its "visible" text content which is represented as a single contiguous string. We introduce the AST/TA data structures formally by its abstraction, namely the AST/TA model, and relate the model to the well-known XML Information Set. Moreover, we address specific issues that must not be ignored in the context of XML and that influences the design of the AST/TA structures strongly. We also compare requirements of the AST/TA approach with those found in current work. Finally, we describe those operations that take advantage of the design principles of the AST/TA data structures.

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. For this reason, we decided to design a management system for such data from scratch to provide 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 structures. They have already been implemented in main memory and provide 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 structures designed for persistent storage.
For convenience, we introduce an abstraction of these data structures, 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 in 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 languages such as the Information Retrieval Query Language (IRQL) [5].

(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 important goal was to support information retrieval aspects 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 text in the document, i.e., 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) By not fragmenting the text content, we need at most one access to fetch the complete string "Our Price: $11.96" from storage—no filtering is needed. (2) As the storage representation is not stained with markup, the search string matches directly the presentation string. Furthermore, the distances in "visible" text that a user may have in mind when formulating a query match to the distances in the text content on storage.

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. Moreover, the separation of structure and content makes it possible to present the navigation structure of a document more densely. Therefore, navigating document structures becomes more efficient, due to shorter distances within the structure presentation.

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 may insert new "markups" without considering 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 [10].
Another advantage to referring to the text content as a single contiguous string is that we may build multiple hierarchies [1] 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>

Finally, our AST/TA data structures support different views of a text, i.e., text as a sequence of characters and as a sequence of words [12]. 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 [11], thus facilitating a more efficient access to words.

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

2 Related Work

This section provides a brief overview of approaches that primarily devote to the storage of documents and of data and query models relevant to XML or structured documents, respectively. With respect to relevant criteria, we finally compare those work with our AST/TA model.

2.1 Document Storage

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 [13], 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™) [3], in HyperStorM (Hypermedia Document Storage and Modeling) [14], in the project of Structured Multimedia Document DBMS [8], and in NATIX (Native XML Repository) [6].

XML Extender. Based on the DB2 object-relational DBMS, the XML Extender manages XML documents in two different ways, namely XML Column and 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 in a system coupling an object-oriented DBMS and an information retrieval system [14]. 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 (SMD DBMS), an object-oriented MMDBS was developed for storage of SGML documents [8] in the presence of DTDs. The text content of documents is stored as a contiguous text string as a whole rather than being fragmented [9]. 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. However, this approach is not well documented.

**NATIX.** NATIX is a repository for the storage and management of large tree-structured objects, preferably XML documents [6]. 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 may 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 those of our AST/TA approach. All approaches support update operations. In Table 1 contiguous text content means, either the bare text content is stored contiguously or it is left embedded in its original markup. Therefore, we consider the text content of documents as fragmented only in XML Collections of the XML Extender and in NATIX. The hybrid approach of HyperStorM allows to store contiguous portions of documents that may contain markup to a degree. In XML Columns complete documents are stored, including their markup. The SMD DBMS and the AST/TA model propagate to store structure and text content separate from each other. Based on dismantling, structure and text content are separated in XML Collections as well. XML Collections and NATIX do not address issues of content based search. Generic document storage in the absence of schema information is supported by XML Column, NATIX, and the
AST/TA model. Only our AST/TA model explicitly addresses the support of text mining tasks; therefore, it provides an own text normalization strategy.

<table>
<thead>
<tr>
<th>criterion</th>
<th>XML Extender</th>
<th>HyperStorM</th>
<th>SMD DBMS</th>
<th>NATIX</th>
<th>AST/TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>support of update operations</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>contiguous text content</td>
<td>✓</td>
<td>–</td>
<td>to a degree (hybr. approach)</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>storage of text without markup</td>
<td>–</td>
<td>✓</td>
<td>to a degree (hybr. approach)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>separation of doc. structure &amp; content</td>
<td>–</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>support of cont.-based search</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>support of generic document storage</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>support of text normalization</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>operational support of text mining</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 1: Overview of criteria and approaches

2.2 Data and Query Models for Documents

The Document Object Model. The Document Object Model (DOM) is a logical model that defines the logical structure of documents, including document type information, and the way a document is accessed and manipulated. It provides a language-independent and implementation-neutral application programming interface (API) for valid HTML and well-formed XML documents, such that applications may dynamically access and update the content and structure of such documents. Since the object model specifies classes or interfaces, "[t]he Document Object Model is not a set of data structures" [16]. Furthermore, this model is seen as an API to the XML Information Set [18]. DOM Level 2 supports XML namespaces.

The DOM model uses a "tree-like" representation for modeling documents. In a DOM representation, attribute and entity nodes do not have any parent nodes, i.e., documents that come with attributes or entities are represented by DOM "forests" rather than a single tree. These trees are ordered partially and may consist of different types of nodes. For example, for representation purposes of an XML document element, the following types of nodes are used: element, comment, processing instruction, text, CDATA section, entity reference, and entity. In addition to these node types, the DOM provides a document node as the "root" of the complete document.

The text content of documents may spread over several nodes, thus the "visible" text may be considered to be fragmented in a DOM representation. DOM requires XML processors to replace character references and references to pre-defined entities with the replacements
of their corresponding entities in document texts. According to the string representation in DOM, documents are assumed being normalized by processors, additionally.

**The XQuery 1.0 and XPath 2.0 Data Model.** The XQuery 1.0 and XPath 2.0 Data Model [20] is purely conceptual. It does not address any specific implementation issues of representing or accessing the data. This data model is based on the XML Information Set [18] and supports XML Schema [19] types and XML namespaces [15].

The basic concept of the model is to represent XML documents as ordered trees. Such trees are made up of labeled nodes having an identity and a type assigned to. There exists exactly one root node that represents the document itself. Unlike the requirements of the XML Information Set, this document node disregards document type declarations. The children of a document node comprise exactly one element node (the document element) and arbitrary sequences of processing instruction or comment nodes. Element nodes may have children of type element, processing instruction, comment, reference and text. Reference nodes are specific to the model and are designated for implementation purposes of a query system; they do not denote entity references. In addition to their children, element nodes may have the following properties: name, namespaces, attributes, schema type of the content, and declaration (for details see [20]).

In this model, entity and character references are considered to be expanded. Furthermore, this model provides the concept of string-values for accessing the concatenated character data referred to by nodes in documents. However, the text content still is distributed among several nodes. To generate new document nodes and to control access of tree components, the model provides both a constructor concept and accessors. Although, this model provides these two concepts, update operations are not directly supported. That is, the model does not allow, e.g., to remove attributes from element nodes.

**The Proximal Nodes Model.** The Proximal Nodes model is a model for querying document databases on both content and structure [7]. It comes with a data model for text databases, which is complemented by query languages providing operations on both the document structure and the text.

The structure (of an arbitrary structured document) is modeled as a set of independent hierarchies that have their own corresponding trees and types of nodes. The types of nodes used in a tree may be implied by, e.g., SGML markup. The text is modeled as a (long) sequence of symbols (characters, words, etc.). The model postulates that markup, if present, should be filtered out from the text. Furthermore, every node in a tree has an associated segment of contiguous underlying text. A pair of numbers expresses the association between nodes and their corresponding text segments. This concept is analogous to the concept used in the Structured Multimedia Document DBMS [9].

The authors of [7] emphasize that this model is purely logical. Although they briefly discuss the full index and the partial index—these indexes are used for storing hierarchy trees or parts of them, respectively—they do not address specific issues of how to store neither structure nor text. Moreover, this model is designed for structure and text to be more or less static, i.e., updates are considered to be performed on documents rarely. Finally, the model does not address issues specific to XML, e.g., how to deal with document type declarations, character and entity references, entities, namespaces, processing instructions, comments, empty elements, element attributes etc., which are in the focus of the XML Information Set [18].
Comparing Models. Table 2 summarizes the characteristics of the models introduced previously, comparing those with the ones of our approach. Unlike the DOM, the XQuery and XPath Data Model, and the Proximal Nodes model; the AST/TA model represents data structures, i.e., it mandates an implementation for an XML document rather than it constitutes a conceptual model for querying documents. As the Proximal Nodes model is not specific to XML, it does not address issues of supporting the XML Information Set, expanding entity and character references, and supporting XML namespaces. By using a tree built over linear text, both the Proximal Nodes and our AST/TA model provide index functionality. On the contrary, the DOM models a document consisting of multiple trees (a forest); the XQuery and XPath Data Model abstracts a document in a single tree. This implies that the structure and content of documents is not separated in either of these models. In the Proximal Nodes model or AST/TA model respectively, text content is seen or implemented as contiguous string. The XQuery and XPath Data Model and the DOM do not support content based search explicitly. The DOM and the AST/TA are the only models addressing updates.

<table>
<thead>
<tr>
<th>criterion</th>
<th>DOM</th>
<th>XQuery Model</th>
<th>Proximal Nodes</th>
<th>AST/TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>conceptual model</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>mandates an implementation</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>support of XML Information Set</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>expands entity and character references</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>support of namespaces</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>tree structure</td>
<td>forest</td>
<td>single tree</td>
<td>tree over text</td>
<td>tree over text</td>
</tr>
<tr>
<td>provides index functionality</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>separation of doc. structure &amp; content</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>contiguous text content</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>support of cont.-based search</td>
<td>–</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>support of update operations</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2: Overview of criteria and models
3 The Access Support Tree/TextArray Model

3.1 Motivation

The AST/TA model is the abstraction of the AST/TA data structures that are relevant and are used for representing XML documents in physical storage. The AST/TA model describes and summarizes the components of XML documents and their relationship to one another with regard to their storage representation rather than to their conceptual models as provided by query models like the Document Object Model [16], the XQuery 1.0 and XPath 2.0 Data Model [20], or the Proximal Nodes Model [7]. 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] each. XML declarations and XML document type declarations are optional and provide additional information constraining XML document elements. Basically, XML declarations specify the XML version and the character set being used. XML document type declarations provide the schema information of documents. As XML Schema [19] provides a means of specifying schemata in the form of XML documents themselves, XML document type declarations may be reduced to just references to other XML documents. With this concept applied to, both XML documents and their associated schemata may be shaped and handled in the same way.

Considering only XML document elements, the AST/TA model is independent from the use of any schema specification. Thus, the model supports generic XML documents, i.e., documents that have no constraints other than well-formedness.

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 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, i.e., when updating a surrogate value, in most cases, either the start position or the length must be updated only.

\footnote{We use the notation introduced in XML Information Set specification [18].}
3.2.1 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 $\wedge$ and $\#$ act as substitutes for word separators. The normalization of TAs is addressed in the next section.

The sample AST consists of seven vertices ($r, \ldots, x$) representing their corresponding document components, i.e., elements, a comment, and text components. For example, vertex $r$ labeled with $\text{header}$ refers to the document element itself. Vertex $t$ represents a comment. We label such vertices with their textual content only, e.g., $\text{check year}$, 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 $s$ ($\text{author}$ 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 $\text{author}$ element is a direct part (child) of the $\text{header}$ element. Thus, the AST tree matches the logical structure of the document element.

![Diagram of AST and TextArray](image-url)

**Figure 1: AST and normalized TextArray**

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 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.

### 3.2.2 Definitions

Our definition of the AST/TA data structures is based on the well-known XML *Information Set (Infoset)*. Infoset defines an abstract data set that provides a consistent set of definitions referring to the information of well-formed XML documents [18]. By relating Infoset to the components of the AST/TA model, we provide the necessary semantics for our model.

**Prerequisites.** In this paragraph, we describe the AST/TA model components and relate them to their corresponding symbols used in the formal definitions following next. We use the same printing for *information items* and *properties* as in Infoset. The tree structure of an AST formally consists of vertices $\mathcal{V}$ and edges $\mathcal{E}$:

- $\mathcal{V}$ includes exactly one (root) vertex corresponding to the *[document element]* of the document information item. Vertices represent such parts of XML documents that relate to types in $T$, which we specify later.
- $\mathcal{E}$ denotes the relationships between *[parent]* and *[children]* vertices. The order among *[children]* vertices correspond to the order given by the *[children]* list of an element information item.

Every vertex in the AST/TA model has a type in $T$:

- $T$ specifies the types of vertices in $\mathcal{V}$. Every type refers to the XML Information Set as follows. The tokens for these types, e.g., element, are taken from the syntax of the *XML Standard* [17].
- element corresponds to the element information item.
- PI corresponds to the processing instruction information item.
- Comment corresponds to the comment information item.

$T$ integrates three additional types, namely CharData, CDSect, and EntityRef, related to character data. Generally, character data correspond to a sequence of one or more contiguous character information items. As Infoset does not provide suitable definitions for those types, we must fall back to definitions and syntax rules of the XML Standard [17].

CharData refers to the syntax rule [14] CharData. That means, CharData refers to character data not containing the string "]]>" and the characters '<' and '&' in their literal forms. We do not address character references, since 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 by XML processors.

CDStart is based on the CDATA Sections definition, in which the permissible character sequences are constrained by syntax rule [20] CDATA. That is, CDStart refers to CDATA sections with their syntactic additives <! [CDATA[ and ]]> removed.

EntityRef refers to the entity reference definition and syntax rule [68] EntityRef. This means, EntityRef describes entity references in their native form, e.g., &entity-ref;. In contrast to unexpanded entity reference information items, we extend the [name] of the reference by the delimiters & and ; in the AST/TA model. Nonetheless, we assume that validating XML processors expand entity references, whenever it is possible.

Vertices of the types element, PI, and Comment carry labels of L:

L corresponds to the set of labels containing the following strings:

Qualified Names of Elements: element labels are qualified names with respect to the syntax rule [6] QName of the XML namespace specification [15]. That is, labels may have, e.g., the form my:author.

Content of PIs: PI vertices carry labels made up of the [target] followed by a single space (#x20) followed by the [content] of a processing instruction item.

Content of Comments: Comment vertices receive labels that are strings referring to [content] of a comment information item.

In addition, element vertices may have (optional) attributes, which we refer to as set A:

A corresponds to the set of [specified] attributes referring to attribute information items. In the AST/TA model, the following properties of attributes are taken into consideration: their qualified names (see previous paragraph), their [normalized value], and their [attribute type]. For a single element vertex v, a,v corresponds to the subset of A representing all attributes of the [owner element] v.

We express the order among siblings of one parent element vertex by $\ll$ or $\prec$ respectively:

$\ll$ corresponds to the order in the [children] list of an element information item in general.

$\prec$ corresponds to the order of two neighbor siblings in the [children] list of an element information item, in which one sibling is the successor of the other one.

For our AST/TA model, we omit considering [character code] of character information items. Following the XML Standard, we assume a character being an atomic unit of text as specified, e.g., by ISO/IEC 10646 [4].
Definitions. Based on the semantics introduced in the previous paragraphs, we summarize our AST/TA model by the following definitions of TextArray, Normalized TextArray, and Access Support Tree. The definition of TextArray provides the basis for the more specific definition of Normalized TextArray.

Definition 1 (TextArray)

A TextArray is the contiguous sequence \( \tau \) of characters with respect to character information items in document order that represents all parts of an XML document referring to the type set \( \mathcal{T}_{PCDATA} := \{ \text{CharData}, \text{CDSect}, \text{EntityRef} \} \). \( \tau_i \) is the \( i \)th character of \( \tau \), where \( i \in \{0, \ldots, \text{length}(\tau) - 1 \} \).

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 \( <! [ \text{CDATA} [ \text{and } ] ] > \), and (3) entity references are stored as specified by the type of EntityRef referring their native form.

\( \square \)

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 [17]. For implementation reasons, we make markup boundaries explicit at the end of the TextArray, however, not at its beginning. This last separator does not contribute to the length of the TextArray.

\( \square \)

Before defining the Access Support Tree in Definition 3, we first present the main ideas of the AST on a more informal level.

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 the successor relation \((\prec)\) 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: \( T := T_{\text{labeled}} \cup T_{\text{PCDATA}} \) with \( T_{\text{labeled}} := \{ \text{element,PI,Comment} \} \) and \( T_{\text{PCDATA}} := \{ \text{CharData, CDSect,EntityRef} \} \). 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 \( \vartheta \in T_{\text{labeled}} \) receives a label \( \lambda \in L \). Vertices of type \( \vartheta \in T_{\text{PCDATA}} \) carry the empty word \( \varepsilon \) as label. Vertices of type \( \text{element} \) may have a set of corresponding attributes \( a_v \) assigned to it, for which is \( a_v \subseteq A \).

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 \( \sigma_p \) and \( \sigma_l \), 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 \texttt{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 \( \sigma_p \) of a text surrogate value, such that vertices reference their text segments via their text surrogate values correctly after updates.

\textbf{Definition 3 ( Access Support Tree (AST) )}

Let:

\( \mathcal{V} \) - the set of all vertices
\( \mathcal{E} \) - the set of all edges, so that: \( \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \)
\( T \) - the set of vertex types; \( T := \{ \texttt{element, PI, Comment, CharData, CDSect, EntityRef} \} \)
\( \vartheta \) - the mapping: \( \vartheta : \mathcal{V} \rightarrow T \) assigning a type \( t \in T \) to a vertex \( v \in \mathcal{V} \)
\( L \) - the set of all vertex labels
\( \lambda \) - the mapping: \( \lambda : \mathcal{V} \rightarrow L \cup \{ \varepsilon \} \) assigning a label \( l \in L \) to a vertex \( v \in \mathcal{V} \)
\( A \) - the set of all element attributes
\( \alpha \) - a relation: \( \alpha \subseteq \mathcal{V} \times A \) assigning the subset of element attributes \( a_v \subseteq A \) to the corresponding vertex \( v \in \mathcal{V} \) (defined in (3) of this definition)
\( \ll \) - an ordering relation: \( \ll \subseteq \mathcal{V} \times \mathcal{V} \) (defined in (6) of this definition)
\( \prec \) - a successor relation: \( \prec \subseteq \ll \) (defined in (7) of this definition)
\( \sigma \) - a mapping: \( \sigma : \mathcal{V} \rightarrow \mathbb{N}_0 \times \mathbb{Z} \) assigning a text surrogate value \( \sigma \in \mathbb{N}_0 \times \mathbb{Z} \) to a vertex \( v \in \mathcal{V} \) (defined in (8) of this definition)
\( \omega \) - a mapping: \( \omega : \mathcal{V} \rightarrow \mathbb{Z} \) assigning an offset \( \omega \in \mathbb{Z} \) to a vertex \( v \in \mathcal{V} \) (defined in (9) of this definition)

\( AST = (\mathcal{V}, \mathcal{E}, T, \vartheta, L, \lambda, A, \alpha, \ll, \prec, \sigma, \omega) \) is an \textbf{Access Support Tree}, if and only if \( AST \) is an ordered tree with the following properties:

1. Every vertex has a type assigned to it:
   \[ \forall v \in \mathcal{V} \ (\vartheta(v) \in T) \]

2. Vertices of type \( T_{label} := \{ \texttt{element, PI, Comment} \} \) receive a label;
   vertices of type \( T_{PCDATA} := \{ \texttt{CharData, CDSect, EntityRef} \} \) receive the empty word \( \varepsilon \) as label:
   \[ \forall v \in \mathcal{V} \ (\vartheta(v) \in T_{label} \implies \lambda(v) \in L) \]
   \[ \forall v \in \mathcal{V} \ (\vartheta(v) \in T_{PCDATA} \implies \lambda(v) = \varepsilon) \]

3. The relation \( \alpha \) has the following properties:
   \[ \alpha(v) := \begin{cases} 
   a_v \subseteq A : & \text{if } \vartheta(v) = \texttt{element} \\
   \emptyset : & \text{else.} 
   \end{cases} \]

4. There is exactly one root vertex, which is of type \texttt{element}:
   \[ \exists ! r \in \mathcal{V} \ (\forall p \in \mathcal{V} \ ((p,r) \notin \mathcal{E}) \land \vartheta(r) = \texttt{element}) \]
(5) All vertices, except the root, have a parent vertex of type element:
\[
\forall v \in \mathcal{V} (\exists p \in \mathcal{V}(p, v) \in \mathcal{E} \implies \vartheta(p) = \text{element})
\]

(6) All children of a vertex are ordered with respect to the ordering relation \( \ll \); counting starts with 0:
\[
\forall v, v_i, v_j \in \mathcal{V} ((v, v_i), (v, v_j) \in \mathcal{E} \implies (i < j \iff v_i \ll v_j)) i, j \in \mathbb{N}_0
\]

(7) A child vertex \( v_j \) is the successor of the vertex \( v_i \) (\( v_i \prec v_j \)):
\[
\forall v_i, v_j \in \mathcal{V} (v_i \prec v_j \iff v_i \ll v_j \land \nexists m \in \mathcal{V} (v_i \ll m \land m \ll v_j))
\]

(8) Every vertex obtains a text surrogate value:
\[
\forall v \in \mathcal{V} (\sigma(v) \in \mathbb{N}_0 \times \mathbb{Z})
\]

\( \sigma_p \) and \( \sigma_i \) represent the vector components of \( \sigma \) referring to the start position and the length of the text segment in a TA referenced to by a vertex. The following property is given for an AST over a non-normalized TextArray and in its initial state. The latter means, all offsets of the vertices \( v \in \mathcal{V} \) (defined in (9) of this definition) are equal to 0.

(a) If \( \vartheta(v) \in T_{FC\text{DATA}} \) then:
\[
\sigma_i(v) := [\text{length of the character sequence the vertex } v \text{ refers to}]
\]

Otherwise:
\[
\sigma_i(v) := \begin{cases} 
\sum_{u \in \text{children}(v) \land \sigma_i(u) \geq 0} \sigma_i(u) & : \exists v_j \in \text{children}(v) (\sigma_i(v_j) \geq 0) \\
-1 & : \text{else.}
\end{cases}
\]

(b)
\[
\sigma_p(v) := \begin{cases} 
0 & : \nexists u \in \mathcal{V} (u \ll v) \land \text{parent}(v) = \emptyset \\
\sigma_p(\text{parent}(v)) & : \nexists u \in \mathcal{V} (u \ll v) \land \text{parent}(v) \neq \emptyset \\
\sigma_p(u) & : \exists u \in \mathcal{V} (u \prec v \land \sigma_i(u) = -1) \\
\sigma_p(u) + \sigma_i(u) & : \exists u \in \mathcal{V} (u \prec v \land \sigma_i(u) \geq 0)
\end{cases}
\]

(9) Every vertex obtains an offset value:
\[
\forall v \in \mathcal{V} (\omega(v) \in \mathbb{Z})
\]

Offsets adjust the position \( \sigma_p \) of a text surrogate value, such that vertices reference their text segments via their text surrogate values correctly after updates.

\( \square \)

In Definition 3 we did not take surrogate values for normalized TextArrays into consideration, because the calculation of these surrogate values is slightly more complex. Hence, we present a detached definition for them in the following.

**Definition 4 (Text Surrogate Values on Normalized TextArrays)**

By this definition, we redefine (8) of Definition 3. We take over terms and symbols with their meaning given by Definition 3. Furthermore, we introduce virtual vertices of type text, which represent neighbor vertices of type \( T_{FC\text{DATA}} \) as one single text vertex. That is, there exist no two neighbor siblings of type text in the AST. The corresponding text segment in a TA of such a virtual text vertex
is the concatenation of its individual text segments. After concatenating such segments, the combined text is normalized in the same way as text of ordinary vertices of type $T_{PCDATA}$.

Again, $\sigma_p$ and $\sigma_i$ represent the vector components of $\sigma$ referring to the start position and the length of the text segment in a TA referenced to by a vertex. The following property is given for an AST over a normalized TextArray and in its initial state. The latter means, all offsets of the vertices $v \in V$ (defined in (9) of Definition 3) are equal to 0.

In addition to (8) of Definition 3, we introduce the value $S(x,y)$ for any two vertices $x$ and $y$ for which: $x \prec y$. The calculation of this "separator" value $S(x,y)$ is found in Table 3. The definition of text surrogate values is then as follows:

(a) If $\varnothing(v) \in \text{text}$ then:

$$\sigma_i(v) := [\text{length of the normalized text the vertex $v$ refers to}]$$

Otherwise:

$$\sigma_i(v) := \begin{cases} 
\sigma_i(u) : & \text{if } |\text{children}(v)| = 1 \land \\
& u \in \text{children}(v) \land \sigma_i(u) \geq 0 \\
\sum_{v_i \in \text{children}(v) \land \sigma_i(v_i) \geq 0} \sigma_i(v_i) + \sum_{v_i,v_{i+1} \in \text{children}(v)} \sigma_i(v_i) \geq 0 \\
\sum_{v_i,v_{i+1} \in \text{children}(v)} \sigma_i(v_i) \geq 0 \\
-1 : & \text{else.}
\end{cases}$$

(b) $\sigma_p(v) := \begin{cases} 
0 : & \text{if } \exists u \in V(u \ll v) \land \text{parent}(v) = 0 \\
\sigma_p(\text{parent}(v)) : & \text{if } \exists u \in V(u \ll v) \land \text{parent}(v) \neq 0 \\
\sigma_p(u) + S(u,v) : & \exists u \in V(u \prec v \land \sigma_i(u) = -1) \\
\sigma_p(u) + \sigma_i(u) + S(u,v) : & \exists u \in V(u \prec v \land \sigma_i(u) \geq 0)
\end{cases}$

\[\square\]

\begin{tabular}{|c|c|c|c|}
\hline
$x$ & $y$ & $E$ & $C$ \\
\hline
$E$ & 1 & $\{0,1\}^T$ & 1 \\
$C$ & 0 & 0 & 0 \\
$T$ & 1 & $\{0,1\}^T$ & \bot \\
\hline
\end{tabular}

$E$: type element (no true empty element)

$C$: types Comment, PI, and true empty elements

$T$: type text (virtual text vertices)

$\bot$: undefined value

$x \prec y$ must be true.

\[\uparrow 1, \text{ if } \exists z (y \ll z \land \sigma_i(z) \geq 0); \text{ otherwise this value is 0.}\]

\textbf{Table 3: Calculating the number of separators between two vertices $x$ and $y$}

Table 3 is based on the observation that, whether a separator is found in a TA or not, depends on the sequence of types of sibling vertices. For example, we must introduce a separator between an element vertex $e$ and a text vertex $t$; therefore $S(e,t) = 1$. Sequences in which a vertex of type $C$ follows a vertex of type $E$ or $T$ respectively, result in a non-determinism. In both cases, calculating the value $S$ is context-sensitive. That is, the value $S$
for such a sequence is equal to 1, if and only if there exists some sibling of the $C$ type vertex and that sibling has a non-negative length in its surrogate value. Otherwise, the value $S$ of such sequences is equal to 0. This distinction ensures that all text surrogate values in an AST have their start positions in the valid range, i.e., from 0 to the length of the TextArray.

In the following, we discuss issues of integrating the proposed text normalization and relevant XML features into the AST/TA model.

3.2.3 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:  \texttt{<author>Wilhelm Busch</author><title>Max und Moritz...}
TextArray: \texttt{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:  \texttt{<author>Wilhelm Busch</author><title>Max und Moritz...}
TextArray: \texttt{Wilhelm\_Busch\_#Max\_und\_Moritz...}

There is no problem in removing the string helmBu, 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:  \texttt{<author>Wilhelm Bu</author><title>x und Moritz...}
TextArray: \texttt{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 that 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.

3.2.4 White Space Handling

The XML Standard provides for White Space Handling [17]. Editing an XML document, a user may set apart markup for greater readability in XML sources by inserting additional spaces, tabs, or blank lines. Such formatting often violates the content model of corresponding elements. As this formatting is not part of the document content, our AST/TA model does not support White Space Handling in its current state. However, we easily may enable White Space Handling by introducing an additional vertex type.
3.2.5 Handling Entities

Entities refer to physical components rather than to the logical structure of XML documents. For this reason, we do not address them in the AST/TA model. However, the information about entities in a document may be of importance for applications. Therefore, we advocate mapping the information about replacement texts of parsed entities to additional structures—entity indexes. Simple entries for such indexes may look like as follows: `<name of the entity> <text surrogate value>`, in which the text surrogate value refers to values in the corresponding AST. Let’s have a look at the following sample entity, given by its replacement text:

```
front text<A>text of A</A>back text
```

Entities are expanded with regard to the AST/TA model, thus we find the text of this entity in the TA. Therefore, this entity may be clearly located by the start position and the length of its text content in the TA. However, we need additional information in an index with respect to the following example of an entity replacement:

```
<B><A>text</A></B>
```

A and B elements have same text surrogate values. If this entity replacement represents the only child of another B element, the information of an index entry must be suitably extended, e.g., by supplying the ID of the vertex in which the entity immediately occurs.

3.2.6 Handling XML Namespaces

XML namespaces are used for qualifying element and attribute names [15]. Namespace attributes are element attributes qualified by `[prefix]` that declare namespaces for all elements in the subtree starting with the corresponding element. Avoiding redundancies, the AST/TA model handles namespaces attributes in the same way as any other attributes. For speeding up queries related to namespaces, we may introduce namespace indexes similar to the entity indexes. An index entry may consist of the following components:

```
<[prefix]> <[namespace name]> <text surrogate value> <vertex ID>
```

where `[prefix]` and `[namespace name]` refer to properties in Infoset. Prefixes or namespaces names may become the search keys. The vertex ID describes the root element from which the namespace declaration is valid. Based on inclusion relationships, text surrogate values help to find the relevant namespace declarations for an arbitrary AST vertex quickly.

3.2.7 Handling Entity References and CDATA Sections

In analogy to the indexes introduced previously, we may apply indexes to entity references and CDATA sections as well. If the access to entity references and CDATA sections is handled by indexes, we might do without having particular vertex types for them. That means, we then need only one type for character data in the AST/TA model.
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.

3.3.1 The DDL operations

We assume an XML processors 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 [17]. Additionally, the processor may perform the text normalization 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 an AST/TA pairs is restoring XML documents from an 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.

3.3.2 Manipulating TextArrays

Searching the TextArray means retrieving an arbitrary sequence of characters or words from the text content of an XML document. The TA enables random access to arbitrary text segments of the text content, in case we already know the length of the desired text segment and its start position in the TA. For reasons that we do not know these parameters, 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 the insertion and the deletion of arbitrary sequences of characters or words respectively, to change the text content of an XML document. We may insert an arbitrary amount of characters to a valid position within a TA with respect to both normalized and non-normalized TAs. We find the position of insertion in constant time. However, inserting a new sequence of characters or words may take much more time, because all text that follows the insertion position in a TA must be shifted, i.e., $\text{length}(\tau)/2$ bytes on average. Insertions into normalized TAs are additionally more complex, because the normalized state of those TAs must be maintained. To improve the performance, we implement TAs as B*-trees as, e.g., described by Carey et al. in [2].

Deletions are performed similar to insertions. However, 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 Bush. 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 may 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 character 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.

![Figure 2: Inserting text into the TextArray](image)
3.3.3 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 may 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 values 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 [10]. 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. If the corresponding TextArray is normalized, separators might have to be exchanged by word separators.

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.

3.3.4 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 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 $z$" that results in $<a>xb</b>y</b></b>zz</a>$. On the other hand, insertion positions may depend on structural properties as in the query: "Insert $<b>y</b>$ into $<a>zz</a>$ 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 structures in main memory. We carefully review the interdependence between the AST and the TA for possible improvements. For example, the AST/TA data structures rely on relative text surrogate values, which enable efficient update operations. However, we need absolute text surrogate values 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 to the already proposed indexes—namespace indexes, entity indexes etc.—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.

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 structures 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, the AST/TA data structures enable 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 structures 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, e.g., handling processing instructions and comments, white spaces handling, handling of entities, providing different views of text content etc. The AST/TA data structures immediately support 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 structures facilitate additional aspects of interest, e.g., tag insertions.

References


