THE XML QUERY EXECUTION ENGINE (XEE)

Dieter Scheffner
Humboldt-Universität zu Berlin, Dept of Computer Science, 10099 Berlin, Germany
scheffne@dbis.informatik.hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin, Dept of Computer Science, 10099 Berlin, Germany
freytag@dbis.informatik.hu-berlin.de

Abstract

The characteristics of XML documents require new ways of storing and querying such documents. In this paper, we introduce the concept of our XML Query Execution Engine (*XEE*), its components, and its current implementation. *XEE* provides a testbed for our Access Support Tree and TextArray data structure of which the basic idea is to separate the (logical) structure of a document from its "visible" text content. Based on this concept, we bring together database and information retrieval technology to improve storage, retrieval, and querying of large XML document collections, in particular with respect to updates. We further explain our current approach of efficiently implementing Access Support Trees for secondary storage and reveal issues that arise when implementing such a system.

Keywords: XML, data structures, database management system

1. Introduction

XML has become widely accepted for both data representation and exchange of information over the Internet. For this reason, the amount of data in XML format is rapidly growing, making appropriate systems for their handling necessary. Conventionally, the management of large numbers of data is sensibly left with database management systems (DBMSs), which are well-known for their efficiency in handling immense amounts of data. However, DBMSs are designed for structured rather than for semi-structured or unstructured data. XML documents or data are semi-structured beside other characteristics. For this reason, it is hard for conventional DBMSs to handle any kind of XML data

efficiently. Thus, systems are required that integrate well established features of DBMSs with the ability to manage large numbers of data that greatly vary in structural and access-related requirements in an efficient and reliable way.

In this paper, we introduce our approach of a system capable of storing, retrieving, and querying XML documents, namely the *XML Query Execution Engine* (*XEE*). We describe concepts used and the current state of our implementation. In contrast to other similar systems, the design of *XEE* is based on an alternative approach for managing XML documents and data. That is, we build the *XEE* system on top of an appropriate *XML data type*—the *Access Support Tree* (*AST*) and *TextArray* (*TA*) data structure [10]. To the best of our knowledge, the storage concept, on which our data structure is based, has not been investigated yet in detail. Therefore, one important goal of our *XEE* system is to provide a testbed for the AST/TA data structure. We advocate two things with this approach. First, we integrate established database and information retrieval technology into such systems. Second, we support XML document management already on the physical system level to enable efficient implementations for storing, retrieving, and querying XML documents.

For the motivation of our approach in detail and a discussion of conventional approaches for managing XML documents, we refer, due to space limitations here, to our technical report (see [11]).

2. The AST/TA Approach for Managing XML Documents

In the following, we refer to *generic* XML documents only, i.e., documents having no constraints other than *well-formedness*, thus consisting of an *XML document element* only. We further consider XML documents to consist of both structure and content. The content of a document is the "visible" text as a human reader sees the document when using a browser. We illustrate this situation in Fig. 1, where the "visible" text content of the corresponding XML document is shown on the right hand side. We recognize that the entire "visible" text content of an XML document is "in document order" and is not interspersed with markups. The visible portion of the text is a user has in mind when formulating a query about the text content of documents.

Based on this observation, we designed our AST/TA data structure [10]. We distinguish the Access Support Tree and the TextArray of a *document element*. The entire logical structure of the document element, including element attributes, comments etc., is incorporated and stored in an ordered tree—the Access Support Tree (AST). The TextArray (TA) stores 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. In our approach XML documents are represented by AST/TA pairs in physical storage.

XML document

```
<story>
  <header>
    <author from="1832" to="1908">Wilhelm Busch</author>
    <title><published year="1865"/>Max und Moritz</title>
  </header>
  <body>
    <section>
      <heading>Vorwort</heading>
      <verses>
        <verse>Ach, was muß man oft von bösen</verse>
        <verse>Kindern hören oder lesen!</verse>
        <verse>Wie zum Beispiel hier von diesen,</verse>
        <verse>Welche Max und Moritz hießen;</verse> ...
      </verses>
    </section> ...
  </body>
</story>
```

The text content of the document

```
Wilhelm Busch
Max und Moritz


Vorwort

Ach, was muß man oft von bösen
Kindern hören oder lesen!
Wie zum Beispiel hier von diesen,
Welche Max und Moritz hießen;
...
...
```

Figure 1. A sample XML document and its "visible" text content

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. When we apply this linkage concept to documents that do not change their text content in length, text surrogate values alone are 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.

Figure 2 shows the concept of representing an XML document by an AST/TA pair. The AST vertices (r, ..., x) represent 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. Text vertices such as v and x carry no labels. 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"`. Expressions in parenthesis, e.g., `(14,14)`, depict text surrogate values; expressions in square brackets such as `[0]` denote offsets. 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. In our example, all offsets are equal to 0, since we assume the document not being updated yet. For conceptual details of our AST/TA data structure, we refer to [10]. The AST/TA data structure is especially designed for update functionality, a feature that is often neglected in other approaches.

The AST/TA may be characterized as a "hybrid approach", since we separate structure from content at the same time managing both parts as an atomic unit. However, our approach differs from the *hybrid approach* described earlier by Böhm et al. in [1]. The advantage of our "hybrid approach" is that the decomposition of documents remains always the same. There is no administration overhead.

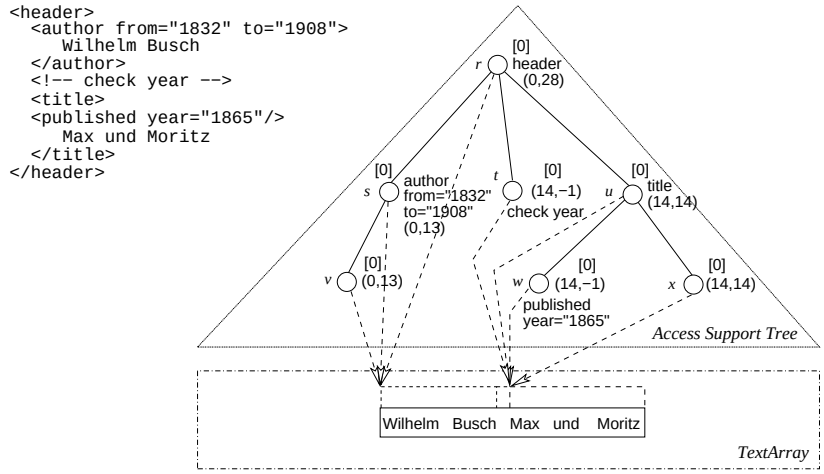

Figure 2. The simplified concept of AST/TA

3. System Overview

Efficient operations in databases require special support at the physical level. Therefore, we decided to design and to build a system based on an appropriate data structure, namely our AST/TA data structure, to overcome performance limitations as they appear in systems based on, e.g., relational, object-relational, or object-oriented approaches. The objective of the XML Query Execution Engine (*XEE*) is to provide a testbed for our AST/TA data structure whose the storage concept has not been investigated yet in all details. Since our focus is on such issues, we do not consider concurrency control, recovery, parallelism, distribution etc. at the current state of the system.

3.1 The Overall Architecture

The system architecture of our *XEE* system is based on the fundamental requirement for DBMSs to achieve both physical and logical data independence. For this reason, we implement suitable query languages for accessing the documents stored with the system. Additionally, we provide a comprehensive logical view of all documents by mapping the internal data representation (AST/TA data structures, indexes etc.) to appropriate data models. The latter is important, because query languages, such as *XQuery*[15], are based on their

corresponding model. Hence, our *XEE* system integrates the principles of the *ANSI/X3/SPARC* architecture. That is, the system architecture consists of three layers: the internal, the conceptual, and the external layer (see Fig. 3).

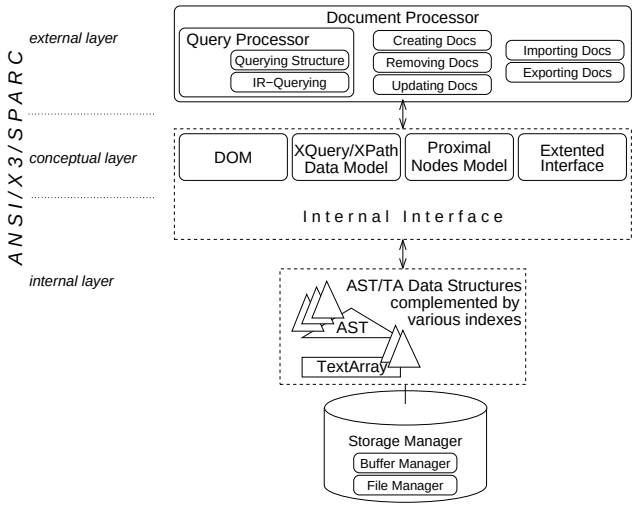

Figure 3. Architectural Overview

3.2 Storage Management

The storage manager—the buffer manager in cooperation with the file manager—hides the underlying storage layout. It is responsible for retrieving, writing, and caching data blocks or pages on behalf of their client applications. In our *XEE* system, the storage manager maps requests for individual pages to the related data blocks in the corresponding files.

3.3 The TextArray

As the Access Support Tree (AST) and TextArray (TA) are independent from each other, the text content of XML documents is managed by and stored in TAs only. We store the data of TAs as *large objects* on disk; TAs reside in separate files. For this purpose, it is necessary to fragment the byte sequences of TAs into disk pages. The storage manager is responsible for retrieving and writing those pages.

TAs are designed for retrieving and writing arbitrary sequences of characters or words from or to the "visible" text content of XML documents [10]. To facilitate text or information retrieval support appropriately, TAs must provide both sequential and random access to text segments of documents they manage. As TAs are designed for *non-normalized* text (sequences of characters) and *normalized* text (sequences of words)—see [10], we provide generic access to

TAs by a suitable interface. Furthermore, TAs have no fixed length. Due to possible document updates, TAs may change their size during their lifetime. We must take into account this requirement when implementing TAs. Therefore, TAs are *large objects* based on *positional B+-tree* indexes, supporting both efficient sequential and random access and efficient updates. In particular, the AST/TA integrates specific needs with respect to the combination of XML and the different data models we support in our system.

3.4 The Access Support Tree

ASTs manage and store the logical structure—the "markup"—of XML documents in one file. In contrast, all text content (cmp. Fig. 1) is managed by TAs. Therefore, ASTs are qualified for accessing the structure or meta data of XML documents. Unlike the implementation of TAs, the implementation of ASTs is far more challenging.

We face the following main challenges when implementing an AST based on secondary storage:

1 AST vertices have *text surrogate values* adjusted by *offset*s assigned to. It is important to update these values efficiently.

2 ASTs must be serialized and fragmented into byte sequences having a maximum size of one disk page.

3 We must decide which parts or subtrees of an AST are stored on the same page.

4 ASTs consist of vertices that are different in size and may change their size in updates.

We propose to follow the idea of Kha et al. to address the problems (1), (2), and (3). In [6] the authors describe a data structure for indexing XML documents based on *Relative Region Coordinate*s (*RRC*s). Region coordinates describe the location of content data in XML documents; they refer to start and end positions of text sequences in XML documents. RRCs are region coordinates being adjusted by offsets relative to the corresponding region coordinate of the parent node in the index structure. In our AST data structure, we use *text surrogate values* combined with *offsets* in analogy to RRCs. When updating the text content of an XML document that is stored as AST/TA pair, we must update a number of text surrogate values and offsets in the AST (see [10]). This problem is analogous to the *RRC-update problem* described in [6].

Kha et al. designed their index structure to perform RRC updates efficiently. Their idea is to store index nodes whose coordinates are likely to be changed at the same time on the same disk page [6]. Hereby, the number of disk I/Os is kept as small as possible when updates occur. Such a group of nodes, which

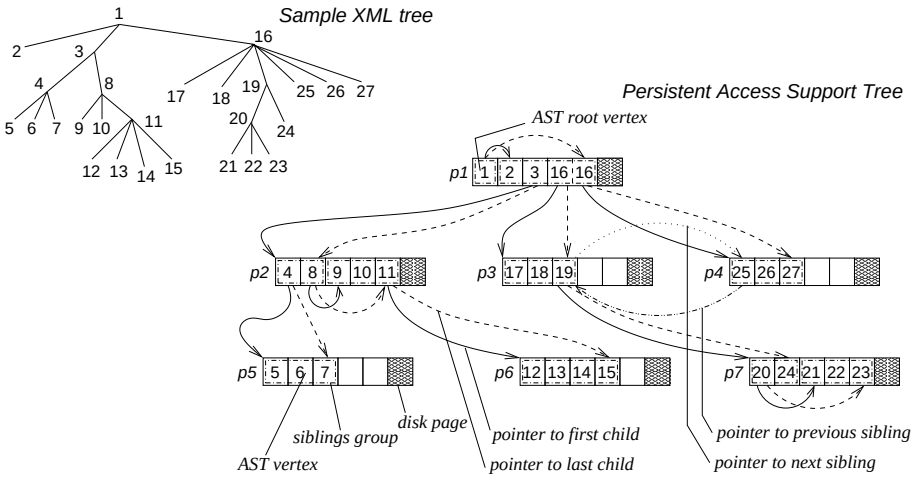

Figure 4. Sample XML tree and its corresponding Persistent AST

must fit on a disk page, is called *Block Subtree* (*BST*). The nodes in a BST are organized as a tree representing a fragment of an *XML tree*. The individual BSTs are linked together, such that they represent the complete tree. Thus, every BST has (exactly) one root node, which is linked to a leaf node of the parent BST—if a parent BST exists. For details, we refer to [6].

We adopt the approach of mapping BST index structures to secondary storage, because of both (a) enabling efficient updates of text surrogate values and their offsets and (b) having disk pages arranged into trees, i.e., every disk page is linked to at most one parent page. The latter feature is the key to our approach for implementing efficient offset handling, which we explain later.

Figure 4 illustrates our approach of mapping the sample XML tree (introduced earlier in [6]) to our *Persistent Access Support Tree* (*AST*$^{\mathcal{P}}$). We store all children of a vertex as a siblings group on a disk page. Consider, e.g., the siblings group 5, 6, 7, which is stored on page *p5* and represents the children of vertex 4 (page *p2*). Parent and child vertices are connected by pointers. A parent vertex has a pointer to its first and a pointer to its last child. Vice versa, a siblings group has a pointer its parent vertex. (For clarity reasons, we left such references out in Fig. 4.) A disk page can store more than one siblings group, e.g., see page *p2*. Here, we find two groups, 4, 8 and 9, 10, 11, which make up a tree of siblings groups. Group 4, 8 is the root group for this page. In general, every page has exactly one siblings group representing the root for this page. If a parent vertex for such a root group exists, it is stored on a different page—the parent page. Hence, our AST$^{\mathcal{P}}$ incorporates the BST index concept. Following the BST index concept further, we divide groups of siblings, in case they do not fit on a single page. For example, the siblings 17, 18, 19, 25, 26, 27 are split up in two pages (*p3* and *p4*). According to [6], we provide a copy of the

parent vertex (16) in the parent page for each child page. We may establish additional links between siblings fragments for only performance reasons on different pages as shown in Fig. 4 between *p3* and *p4*. We neglect such links in our further considerations, since they are not urgent and would violate the tree concept here.

Kha et al. also describe in [6] how to create and manipulate such tree structures. We may easily transfer these algorithms to our AST$^{\mathcal{P}}$. However, there are still open issues we must consider. For example, we must take a look at algorithms for splitting and for merging pages when overflows or underflows occur, respectively. This contemplation is especially important, because we must use a page layout different from that used for BST indexes—ASTs consists of vertices of variable and changing size. We present ideas on our page layout later.

AST$^{\mathcal{P}}$s build trees based on links between parent and child pages. We require text surrogate values being immediately recalculated rather than just aligned by offsets, assuming that pages loaded into main memory can be processed fast. We note, such recalculations take place when offset updates happen; thus, the corresponding pages must be written anyway. For this reason, the only offsets we still need are offsets for disk pages (*page offsets*).

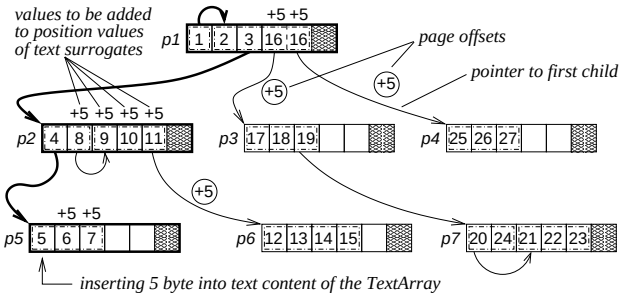

Figure 5. The page offset concept of AST$^{\mathcal{P}}$

Figure 5 shows the principle of page offsets. Suppose, we insert 5 byte of text into the text segment that the text vertex 5 spans over. Therefore, we must update the *length values* of the text surrogates along the path of vertices 1, 3, 4, and 5 on the pages *p1*, *p2*, and *p5*. All *position values* in text surrogates of sibling vertices that are to the right of this path, must be increased by +5. In our example this is valid for both vertices 16 and for the vertices 8, 6, and 7. As page *p2* must be written anyway, we additionally increase the *position values* of surrogates of the vertices 9, 10, and 11. Note, that we change values on three pages only (*p1, p2, p5*)—the update path. Three offsets for the pages *p3*, *p4*, and *p6* still must be recorded. We record page offsets in the *Offset Dictionary*, which may occupy one or more pages attached to the AST$^{\mathcal{P}}$ file. The Offset Dictionary

only needs as much entries as pages are in use in an $AST^{\mathcal{P}}$. The advantage of using only page offsets instead of vertex offsets is twofold. First, we save space on AST pages, because vertices do not need offset data anymore. Second, much less offsets must be updated on behalf of the whole tree. Furthermore, we may easily check the dictionary whether there is an entry different from 0, making unnecessary access to the Offset Dictionary superfluous.

We briefly explain the page layout. A page starts with an array of entries in which each entry has a fixed length and represents the entry point to the data of the vertex it manages. This array section is followed by additional management structures. The data of vertices such as element and attribute names etc. constitute the tail of all sections. Thus, changes affecting the length of, e.g., element and attribute names, cause much less byte moving on pages. We fill pages between 50% and 100%.

3.5 Indexing

We complement our AST/TA data structure with various kinds of indexes to improve query performance. In *XEE* we basically provide indexes on both ASTs and TAs. Hence, we distinguish indexes on both structure and content of XML documents; this is analogous to [7]. We apply indexes among other things to element names, attribute names, attribute values, comments, processing instructions, and structural paths. In addition, we use indexes known from information retrieval for content-based indexing such as, e.g., *String B-Trees* [3]. Hereby, we cover the "traditional" requirements for indexes on XML documents as demanded, e.g., in [12].

Furthermore, we provide indexes notably for *entities*, *CDATA sections*, and *XML namespaces*. Entities refer to physical components rather than to the logical structure of XML documents. We map the information about replacement texts of parsed entities to additional structures—*entity indexes*, since our AST/TA data structure refers to logical structure only. 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. We propagate this concept to *CDATA section indexes* and *XML namespace indexes* as well. XML namespaces are valid for complete subtrees. Text surrogate values help to find namespace declarations in charge of an arbitrary AST vertex very quickly based on inclusion relationships. Eventually, we also take a look at issues like, e.g., how we can make indexes on structure and on content interact with one another.

3.6 The Internal Interface and Data Models Supported

The *Internal Interface* merely brings together and combines functionality provided by the diverse data structures of AST, TA, and additional indexes. We

implement the *XQuery 1.0 and XPath 2.0 Data Model*[16] and the *Document Object Model* (*DOM*)[14] using the Internal Interface. In addition to these *de facto* data models, we experiment with the *Proximal Nodes Model*[7], which is designed for structured documents in general and integrates the aspect of querying documents with respect to both structure and content. With such a document model integrated, our system naturally enables the support of information retrieval and structural queries in equal rank. In *XEE* we support such a diversity of models to get more insight how models may interact with our data structure.

3.7 The Document Processor

The *Document Processor* provides the interface to the *XEE* system for users and external applications. The main component of the Document Processor is the *Query Processor*. The Query Processor allows the user to query both the structure and the content of XML documents. We experiment with several implementations of query processors. For example, we use a processor that may implement *XQuery*[15] for querying the structure of documents. For queries on content, we currently design a query language based on the *matching sublanguage*[7], *PAT expressions*[9], and *IRQL*[4].

Besides querying documents, it must also be possible to manage XML documents. That is, the Document Processor must provide functionality to create, remove, and update documents. Another important facility must support the import and export of documents to or from the system as usually supported by DBMSs. We map such "additional" functionality to the *Extended Interface* of our AST/TA data structure, whereas we map functionality for querying to the interfaces of the supported data models. We investigate these mappings based on our *XEE* system.

4. Related Work

This section gives a brief overview of existing management concepts for structured documents that are related to our work: *Structured Multimedia Document DBMS* (*SMD DBMS*)[2], *HyperStorM* (Hypermedia Document Storage and Modeling)[13], the *Proximal Nodes Model*[7], and NATIX (Native XML Repository)[5].

The concept we use in our AST/TA 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 the best of our knowledge, this approach was only used twice, i.e., in the framework of SMD DBMS and in the Proximal Nodes Model.

An object-oriented MMDBS was developed for storage of SGML documents in the presence of DTDs within the framework of the SMD DBMS [2]. Their

approach is not well documented with respect to storage design; only few implementation issues are discussed in [8].

The Proximal Nodes Model is a model for querying document databases on both content and structure [7]; it regards documents as being static objects and does not refer to XML documents directly. Since the PN model is a purely logical model, it does not mandate any specific implementation.

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 [13]. The authors of [1] advocate the *hybrid approach* as the favorite strategy for representing SGML documents in object-oriented databases. The hybrid approach is a trade-off between storing each logical document component as an individual database object and storing complete documents as BLOBs in databases.

NATIX is a repository for the storage and management of large tree-structured objects, preferably XML documents [5]. NATIX stores generic XML documents that do not depend on any schema information. The main idea of NATIX is to map the logical structure of an XML document directly into the corresponding physical structure. Such trees may be split up among several pages in storage. Hereby, the text content of documents is interspersed with structural data, e.g., element and attribute names and is thus fragmented within pages.

5. Conclusion

This paper introduces the XML Query Execution Engine (*XEE*), which we currently design and implement. The focus of the *XEE* system is to provide a testbed for the storage and retrieval approach for XML documents that is brought in by our AST/TA data structure. That is, we investigate a storage and retrieval approach, in particular with respect to updates, that has been neglected so far in the past. We are convinced this is a promising approach for bringing together database and information retrieval technology to improve storage, retrieval, and querying of large XML document collections.

We give insight into the concept of the *XEE* components and the current state of the system by introducing the system architecture. For example, we introduce our concept of efficiently implementing $\text{AST}^{\mathcal{P}}_{\mathcal{S}}$ for persistent storage based on the design of our main memory ASTs. Moreover, we carefully review the interdependence between the AST and the TA data structures for possible improvements, namely we apply diverse indexes to these structures. We propose indexes such as the *entity index* and the *XML namespace index* that are specific to our system. We provide efficient handling of large scale XML document collections on secondary storage. Another issue of interest is to provide a suitable interface that enables convenient access to XML documents managed by the *XEE* system. Therefore, we implement a persistent DOM based on

persistent AST/TAs and map query languages to the AST/TAs. We provide access to XML documents both by an API for programming languages and by query languages. 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 the framework of *XEE*.

References

[1] Klemens Böhm, Karl Aberer, and Wolfgang Klas. Building a Hybrid Database Application for Structured Documents. *Multimedia Tools and Applications*, 8(1):65–90, January 1999.

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

[3] Paolo Ferragina and Roberto Grossi. The String B-Tree: A New Data Structure for String Search in External Memory and its Applications. *J. of the ACM*, 46(2):236–280, 1999.

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

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

[6] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. An XML Indexing Structure with Relative Region Coordinates. In *Proceedings of ICDE, 2001, Heidelberg, Germany*, pages 313–320. IEEE Computer Society, 2001.

[7] 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.

[8] M. Tamer Özsu, Duane Szafron, Ghada El-Medani, and Chiradeep Vittal. An Object-Oriented Multimedia Database System for a News-on-Demand Application. *Multimedia Systems*, 3(5-6):182–203, 1995.

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

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

[11] Dieter Scheffner and Johann-Christoph Freytag. The XML Query Execution Engine (XEE). Technical Report HUB-IB-158, Humboldt Universität zu Berlin, 2002. avail. at: `http://dbis.informatik.hu-berlin.de/publications/techreports.html`.

[12] Harald Schöning. Tamino - A DBMS designed for XML. In *Proceedings of ICDE, 2001, Heidelberg, Germany*, pages 149–154. IEEE Computer Society, 2001.

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

[14] 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.

[15] World Wide Web Consortium. XQuery 1.0: An XML Query Language. Technical Report WD-xquery-20011220, W3C, December 2001.

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