eXist: An Open Source Native XML Database

Wolfgang Meier

Darmstadt University of Technology
meier@ifs.tu-darmstadt.de

Abstract. With the advent of native and XML enabled database systems, techniques for efficiently storing, indexing and querying large collections of XML documents have become an important research topic. This paper presents the storage, indexing and query processing architecture of eXist, an Open Source native XML database system. eXist is tightly integrated with existing tools and covers most of the native XML database features. An enhanced indexing scheme at the architecture's core supports quick identification of structural node relationships. Based on this scheme, we extend the application of path join algorithms to implement most parts of the XPath query language specification and add support for keyword search on element and attribute contents.

1. Overview

eXist (http://exist-db.org) is an Open Source effort to develop a native XML database system, which can be easily integrated into applications dealing with XML in a variety of possible scenarios, ranging from web-based applications to documentation systems running from CDROM. The database is completely written in Java and may be deployed in a number of ways, either running as a stand-alone server process, inside a servlet-engine or directly embedded into an application.

eXist provides schema-less storage of XML documents in hierarchical collections. Using an extended XPath syntax [2, 8], users may query a distinct part of the collection hierarchy or even all the documents contained in the database. Despite being lightweight, eXist's query engine implements efficient, index-based query processing. An enhanced indexing scheme supports quick identification of structural relationships between nodes, such as parent-child, ancestor-descendant or previous-/next-sibling. Based on path join algorithms, a wide range of path expression queries is processed only using index information. Access to the actual nodes, which are stored in the central XML document store, is not required for these types of expressions.

The database is currently best suited for applications dealing with small to large collections of XML documents which are occasionally updated. It has been designed with document-centric documents in mind. Document-centric as opposed to data-centric documents typically contain larger sections of text, a lot of mixed content and less machine-readable data. However, querying these types of documents is not very well supported by XML query languages like XPath. eXist thus provides a number of extensions to standard XPath to efficiently process fulltext queries, including keyword searches, queries on proximity of search terms or regular expressions.

A growing number of developers is actively using the software in a variety of application scenarios. Applications show that eXist – despite its relatively short project history – is alread able to address true industrial system cases (see section on applications below). Main contributions of this article are:

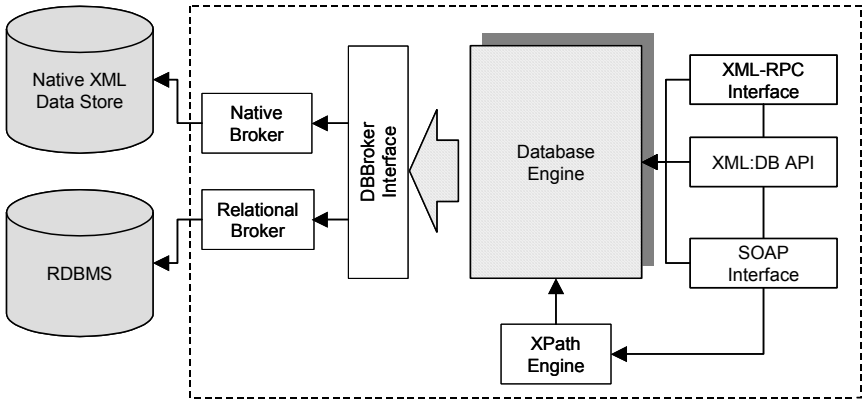
1. We provide a detailed description of the data structures, the indexing architecture and the query processing aspects implemented in eXist.
1. We show how an enhanced numbering scheme for XML nodes at the architecture's core could be used to implement efficient processing of complex path expression queries on large, unconstrained document collections. The numbering scheme supports quick identification of relationships between nodes as well as navigation through the document tree. Contrary to other proposals, which focus on the efficient processing of parent-child and ancestor-descendant relationships, our indexing scheme supports all axes of navigation as required by the XPath specification.
2. While previous contributions have indicated the superiority of path join algorithms over conventional tree traversals for a limited set of expressions [15, 21, 23], we extend the application of path joins to implement large parts of the XPath query language and add support for keyword search on element and attribute content.

The paper is organized as follows: We provide a general overview of eXist's architecture and its features in the next section. Section 3 presents some details on the indexing and storage of XML documents as implemented in eXist. We will introduce the numbering scheme used at the core of the database engine and describe the organization of index and data files. Section 4 will then explain how the numbering scheme and the created index structures are used in query processing. In Section 5 we present some experimental results to estimate the efficiency and scalability of eXist's indexing architecture and query engine. Section 6 will finally point out some example applications and ongoing projects which are based on eXist.

2. Database Architecture

While the term "native XML database" is currently applied to many products available on the market (please refer to [4] for a comprehensive list), it is not very well defined. In this article the term is used according to the definition proposed by the vendor independent XML:DB initiative [24]. As proposed there a native XML database is a database which defines a logical model for XML documents, stores and retrieves documents according to that model and has an XML document as its fundamental unit of storage. The definition does not put any constraint on the underlying physical storage model.

Fig. 1. Architecture Overview



Though eXist provides an optional relational storage backend, it has been designed from the start to be a pure native XML database. Documents are stored according to the W3C's document object model (DOM) [14]. Backend implementation details are completely hidden from eXist's core (see figure 1). All calls to the storage backends are handled by database broker classes implementing the abstract class DBBroker. These classes provide a rather limited set of basic operations, like adding, removing or retrieving a document or fragment. Additionally, there are methods to access available index-structures, for example, to retrieve the set of element nodes corresponding to a given qualified name. The XPath query engine has also been implemented as a separate component, which relies entirely on the methods exposed by DBBroker.

2.1. Storage Backends

As a proof of concept for the indexing and storage architecture, the first public releases in 2001 were entirely based on the relational backend. These versions served as a testing platform to explore if the basic design would meet our expectations. Beginning with version 0.6, the relational backend has been supplemented by a new storage backend implementation, which has been written from scratch in Java. Switching between the internal XML store and an external RDBMS is done in the configuration file.

However, eXist has not been designed to achieve best performance with a relational database as storage backend. Efficient storage of XML in an RDBMS has been explored by other research contributions [10, 12, 19, 23]. Instead development has been led by a generic approach, which could be applied to different backend implementations. For many applications, the native XML data store will yield much better performance than the relational backend.

2.2. Managing Documents in Collections

Inside eXist, documents are managed in hierarchical collections, comparable to storing files in a file system. The XML store is schema independent: collections are not bound to a predefined schema, so the number of document types used by documents in one collection is not constrained. Arbitrary documents may be mixed inside the same collection. Also, documents are not required to have an associated schema or DTD at all, i.e. they may be well-formed only, though the indexer will validate a document against a schema or DTD if one has been defined.

2.3. Deployment

eXist offers several modes of deployment: the database engine may

1. run as a stand-alone server process, providing HTTP and XML-RPC [22] interfaces for remote access.
2. be embedded into other applications, which have direct access to the database via the XML:DB API [26].
3. run inside a servlet engine like Apache's Tomcat. Servlets or Java Server Pages (JSP) running in the same web-application context can directly access the database. XML-RPC, SOAP [27] and WebDAV [11] access is provided by servlets.

Using the third alternative, eXist can be distributed as a single web-application archive (WAR-file), which is simply copied into the web-applications folder of a standard-compliant servlet-engine. No further installation is required.

2.4. Application Development

Depending on how the database engine has been deployed, eXist offers several interfaces for remote access to the database as listed above. XML-RPC is a simple protocol for remote procedure calls using HTTP as transport protocol and XML for message encoding [22]. The Simple Object Access Protocol (SOAP) has recently attracted much attention as an XML-based messaging protocol for the development of so-called Web Services [27]. WebDAV is an HTTP-based protocol to support distributed authoring [11]. We will not discuss these protocols here. Please refer to the available online documentation material for a description.

The preferred way to access eXist from Java applications is to use the XML:DB API [26]. The vendor independent XML:DB initiative has proposed a common interface for access to native or XML-enabled databases. The XML:DB API allows developers to write portable applications, which will work with different products implementing a standard-compliant database driver for the API. Several applications and tools have already been developed by other open source projects. This includes, for example, a browser and editor which retrieves and stores documents in an XML database and a WebDAV interface.

eXist's integration with Cocoon also relies on the XML:DB API. Cocoon provides a powerful application server environment for the development of XML-driven web applications. Features include configurable transformation pipelines and XML server pages (XSP) for dynamic content generation. Cocoon may use eXist as a data store, reading XML files from the database instead of the file system. Resources stored in eXist may be directly referenced in Cocoon's site configuration file. The combination of eXist and Cocoon enables developers to create complex web applications entirely based on XML and related technologies.

3. XML Indexing and Storage

This section takes a closer look at the indexing and storage architecture implemented in eXist. We will first provide some background information and then introduce the numbering scheme used at the core of the database.

3.1. Background

XML query languages like XPath [2, 8], Quilt [5] or XQuery [6] use path expressions to navigate through the logical, hierarchical structure of an XML document, which is modelled as an ordered tree. A path expression locates nodes within a tree. For example, the expression

```
book//section/title
```

will select all "title" elements being children of "section" elements which have an ancestor element named "book". The double slash in subexpression "book//section" specifies that there must be a path leading from a "book" element to a "section" element. This corresponds to an ancestor-descendant relationship, i.e. only "section" elements being descendants of "book" elements will be selected. The single slash in "section/title" denotes a parent-child relationship. It will select only those titles whose parent is a "section" element.

XPath defines additional node relationships to be specified in a path step by an axis specifier. Among the supported axes are ancestor, descendant, parent, child, preceding-sibling or following-sibling. The / and // are abbreviations for the child and descendant-or-self axes. For example, the expression "//section" is short for "/descendant-or-self::node()/child::section".

According to XPath version 2.0 [2], which is contained as a subset in XQuery, the result of a path expression is a sequence of distinct nodes in document order. The selected node sequence may be further filtered by predicate expressions. A predicate expression is enclosed in square brackets. The predicate is evaluated for each node in the node sequence, returning a truth value. Those nodes in the sequence for which the predicate returns false are discarded. For example, to find all sections whose title contains the string "XQuery" in its text node children, one may use the expression:

```
book//section[contains(title, 'XQuery')]
```

The predicate subexpression specifies a value-based selection, while the subexpression "book//section" denotes a structural selection. Value-based selections can be specified on element names, attribute names/values or the text strings contained in an element. Structural selections are based on the structural relationssships between nodes, such as ancestor-descendant or parent-child.

Quite a number of different XML query language implementations are currently available to XML developers. However, most implementations available as Open Source software rely on conventional top-down or bottom-up tree traversals to evaluate path expressions.

Despite the clean design supported by these tree-traversal based approaches, they become very inefficient for large document collections as has been shown previously [15, 21, 23]. For example, consider an XPath expression selecting the titles of all figures in a collection of books:

```
/book//figure/title
```

In a conventional, top-down tree-traversal approach, the query processor has to follow every path beginning at "book" elements to check for potential "figure" descendants, because there is no way to determine the possible location of "figure" descendants in advance. This implies that a great number of nodes not being "figure" elements have to be accessed to test (i) if the node is an element and (ii) if its qualified name matches "figure".

Thus index structures are needed to efficiently perform queries on large, unconstrained document collections. The indexing scheme should provide means to process value-based as well as structural selections. While value-based selections are generally well supported by extending traditional indexing schemes, such as B+-trees, structural selections are much harder to deal with. To speed up the processing of path expressions based on structural relationships, an indexing scheme should support the quick identification of such relationships between nodes, for example, ancestor-descendant or parent-child relationships. The necessity to traverse a document subtree should be limited to special cases, where the information contained in indexes is not sufficient to process the expression.

3.2. Numbering Schemes

A considerable amount of research has been carried out recently to design index structures which meet these requirements. Several numbering schemes for XML documents have been proposed [7, 13, 15, 20, 21, 23]. A numbering scheme assigns a unique identifier to each node in the logical document tree, e.g. by traversing the document tree in level-order or pre-order. The generated identifiers are then used in indexes as a reference to the actual node. A numbering scheme should provide mechanisms to quickly determine the structural relationship between a pair of nodes and to identify all occurrences of such a relationship in a single document or a collection of documents.

In this section we will briefly introduce three alternative numbering schemes, which have been recently proposed. We will then discuss the indexing scheme used at

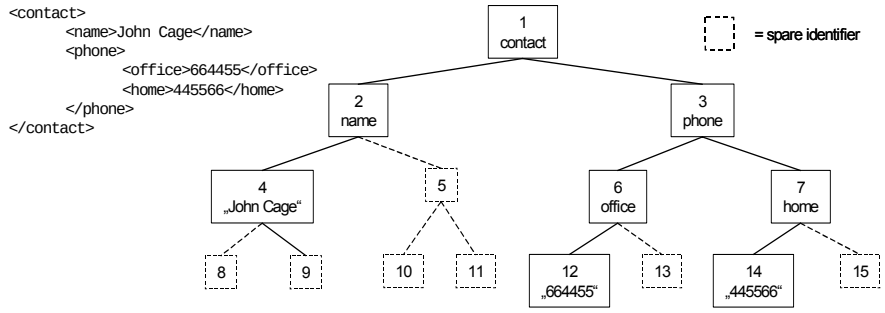
the core of eXist, which represents an extension to the level-order numbering scheme presented below.

An indexing scheme which uses document id, node position and nesting depth to identify nodes has been proposed in [23] (also discussed in [21]). According to this proposal, an element is identified by the 3-tuple (document id, start position:end position, nesting level). Start and end position might be defined by counting word numbers from the beginning of the document. Using the 3-tuples, ancestor-descendant relationships can be determined between a pair of nodes by the proposition: *A node x with 3-tuple (D_1, S_1:E_1, L_1) is a descendant of a node y with 3-tuple (D_2, S_2: E_2, L_2) if and only if $D_1 = D_2$; $S_1 < S_2$ and $E_2 < E_1$.*

The XISS system ([15], also discussed in [7]) proposes an extended preorder numbering scheme. This scheme assigns a pair of numbers *<order*, *size>* to each node, such that: (i) for a tree node *y* and its parent *x*, *order(x) < order(y)* and *order(y) + size(y) ≤ order(x) + size(x)* and (ii) for two sibling nodes *x* and *y*, if *x* is the predecessor of *y* in preorder traversal, *order(x) + size(x) < order(y)*. While *order* is assigned according to a pre-order traversal of the node tree, *size* can be an arbitrary integer larger than the total number of descendants of the current node. The ancestor-descendant relationship between two nodes can be determined by the proposition that *for two given nodes x and y, x is an ancestor of y if and only if order(x) < order(y) <= order(x) + size(x)*.

The major benefit of XISS is that ancestor-descendant relationships can be determined in constant time using the proposition given above. Additionally, the proposed scheme supports document updates via node insertions or removals by introducing sparse identifiers between existing nodes. No reordering of the document tree is needed unless the range of sparse identifiers is exhausted. This feature is used in [7] to assign durable identifiers keeping track of document changes.

Fig. 2. Unique identifiers assigned to the nodes of a sample document by the level-order numbering scheme



Lee et al. [13] proposed a numbering scheme which models the document tree as a complete k-ary tree, where k is equal to the maximum number of child nodes of an element in the document. A unique node identifier is assigned to each node by traversing the tree in level-order. Figure 2 shows the identifiers assigned to the nodes of a very simple XML document, which is modelled as a complete 2-ary tree. Because the tree is assumed to be complete, spare ids have to be inserted at several positions.

The unique identifiers generated by this numbering scheme have some important properties: from a given identifier one may easily determine the id of it's parent, sibling or possible child nodes. For example, for a *k*-ary document tree we may obtain the identifier of the parent node of a given node whose identifier is *i* by the following function:

$$parent_i = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor \quad \textbf{(1)}$$

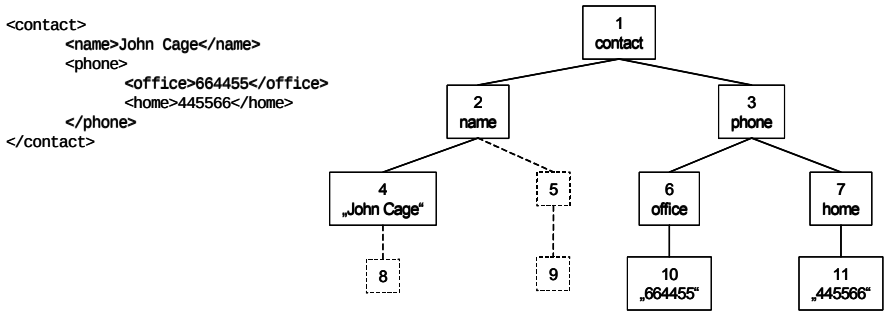
To get the unique identifier of the *j-th* child of the node with identifier *i* we use the function:

$$child_{i,j} = k(i-1) + j + 1 \quad \textbf{(2)}$$

However, the completeness constraint imposes a major restriction on the maximum document size to be indexed by this numbering scheme. For example, a typical article will have a limited number of top-level elements like chapters and sections while the majority of nodes consists of paragraphs and text nodes located below the top-level elements. In a worst case scenario, where a single node at some deeply structured level of the document node hierarchy has a the largest number of child nodes, a large number of spare identifiers has to be inserted at all tree levels to satisfy the completeness constraint, so the assigned identifiers grow very fast even for small documents.

The numbering scheme implemented in eXist thus provides an extension to this scheme. To overcome the document size limitations we decided to partially drop the completeness constraint in favour of an alternating scheme. The document is no longer viewed as a complete k-ary tree. Instead the number of children a node may have is recomputed for every level of the tree, such that: for two nodes *x* and *y* of a tree, *size(x) = size(y)* if *level(x) = level(y)*, where *size(n)* is the number of children of a node *n* and *level(m)* is the length of the path from the root node of the tree to *m*. The additional information on the number of children a node may have at each level of the tree is stored with the document in a simple array. Figure 3 shows the unique identifiers generated by eXist for the same document as above.

Fig. 3. Unique node identifiers assigned by the alternating level-order numbering scheme



Our approach accounts for the fact that typical documents will have a larger number of nodes at some lower level of the document tree while there are fewer elements at the top levels of the hierarchy. The document size limit is raised considerably to enable indexing of much larger documents. Compared to the original numbering scheme, less spare identifiers have to be inserted.

Also inserting a node at a deeper level of the node tree has no effect on the unique identifiers assigned to nodes at higher levels. It is also possible to leave sparse identifiers between existing nodes to avoid a frequent reordering of node identifiers on later document updates. This technique has been described in [7] and [15]. However, eXist does currently not provide an advanced update mechanism as defined, for example, by the XUpdate standard [25]. Documents may be updated as a whole, but it is not possible to manipulate single nodes with current versions of eXist. Support for dynamic document updates is planned for future versions, but currently eXist is best suited for more or less static documents which are rarely updated.

Using an alternating numbering scheme does not affect the general properties of the assigned level-order identifiers. From a given unique identifier we are still able to compute parent, sibling and child node identifiers using the additional information on the number of children each node may have at every level of the tree.

There are some arguments in favour of the numbering scheme currently implemented. Contrary to our approach, the alternative indexing schemes discussed above concentrate on a limited subset of path expression queries and put their focus on efficient support for the child, attribute and descendant axes of navigation. Since eXist has been designed to provide a complete XPath query language implementation, support for all XPath axes has been of major importance during development. For example, consider an expression which selects the parent elements of all paragraph elements containing the string “XML”:

```
//para[contains(., 'XML')]/..
```

The “..” is short for “parent::node()”. It will select the parent element of each node in the current context node set. Using our numbering scheme, we may easily compute the parent node identifier for every given node to evaluate the above expression. We are also able to compute the identifiers of sibling or child nodes. Thus all axes of navigation can be implemented on top of the numbering scheme.

This significantly reduces the storage size of a single node in the XML store: saving soft or hard links to parent, sibling, child and attribute nodes with the stored node object is not required. To access the parent of a node, we simply calculate its unique identifier and look it up in the index. Since storing links between nodes is not required, an element node will occupy no more than 4 to 8 bytes in eXist's XML store.

Additionally, with our indexing scheme any node in an XML document may serve as a starting point for an XPath expression. For example, the nodes selected by a first XPath expression can be further processed by a second expression. This is an important feature with respect to XQuery, which allows multiple path expression queries to be embedded into an XQuery expression.

3.3. Index and Data Organization

In this section we provide some implementation details concerning the index and data organization. We will then explain how the numbering scheme and the created index structures are used in query processing.

Currently, eXist uses four index files at the core of the native XML storage backend:

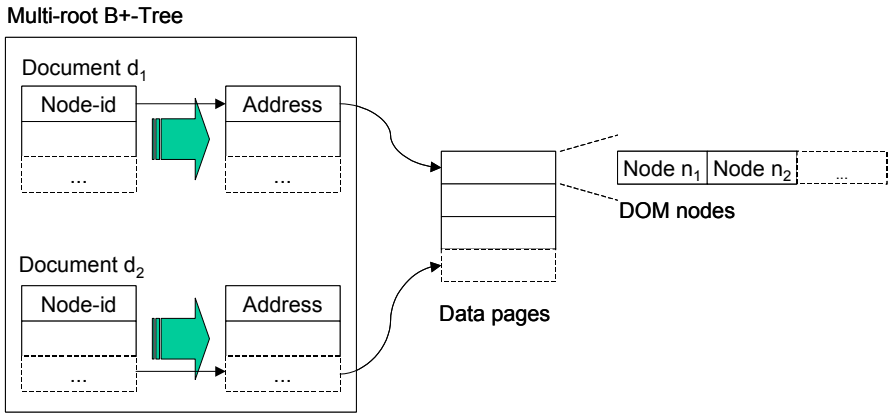
1. `collections.dbx` manages the collection hierarchy
2. `dom.dbx` collects nodes in a paged file and associates unique node identifiers to the actual nodes
3. `elements.dbx` indexes elements and attributes
4. `words.dbx` keeps track of word occurrences and is used by the fulltext search extensions

All indexes are based on B+-trees. An important point to note is that the indexes for elements, attributes and keywords are organized by collection and not by document. For example, all occurrences of a "section"-element in a collection will be stored as a single index entry in the element's index. This helps to keep the number of inner B+-tree pages small and yields a better performance for queries on entire collections. We have learned from previous versions that creating an index entry for every single document in a collection leads to decreasing performance for collections containing a larger number (>1000) of rather small (<50KB) documents.

The index file `collections.dbx` manages the collection hierarchy and maps collection names to collection objects. Due to performance considerations, document descriptions are always stored with the collection object they belong to. A unique id is assigned to each collection and document during indexing.

The XML data store (`dom.dbx`) represents the central component of eXist's native storage architecture. It consists of a single paged file in which all document nodes are stored according to the W3C's document object model (DOM) [14]. The data store is backed by a multi-root B+-Tree in the same file to associate a unique node identifier in a given document to the node's storage address in the data section (see figure 4).

Fig. 4. XML Data Store Organization

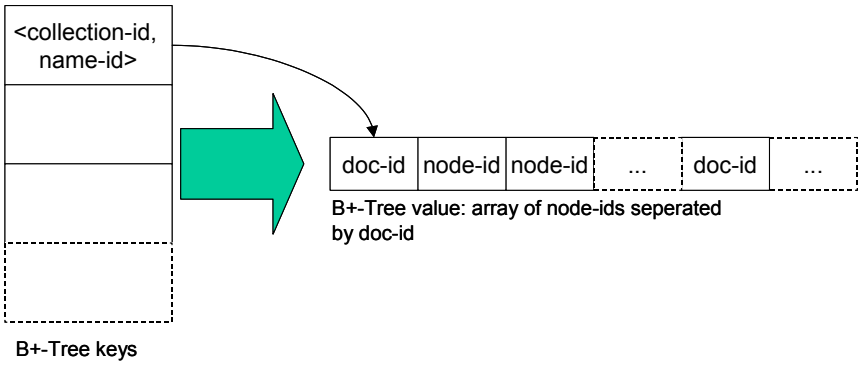


Please note again that it is not necessary to keep track of links between nodes, e.g. by using pointers to the next sibling, first child or parent. The DOM implementation completely relies on the numbering scheme to determine node relationships. For example, to get the parent of a node, the parent's unique identifier is calculated from the node's identifier and the corresponding node is retrieved via an index lookup.

However, ordered sequential access to the data is desirable in many cases, e.g. to serialize a document or fragment from the internal data model back into it's XML representation. To achieve this, nodes are stored in document order, physically located in subsequent data pages.

Thus only a single initial index lookup is required to serialize a document or fragment. eXist's serializer will generate a stream of SAX [17] events by sequentially walking nodes in document order, beginning at the fragment's root node. Any XML tool implementing SAX may be used to post-process the generated stream.

Fig. 5. Index Organization for Elements and Attributes



Element and attribute names are mapped to unique node identifiers in file `elements.dbx` (see figure 5). To save storage space, node names are not directly used as keys. Instead, the collection object associates the qualified element and attribute names with integer ids in a name table. Each entry in the index consists of a key – being a pair of <collection-id, name-id> - and an array value containing an ordered list of document ids and node ids, which correspond to elements and attributes matching the qualified name in the key. To find, for example, all chapters in a collection of books, the query engine will need a single index lookup to retrieve the complete set of node identifiers pointing to chapter elements.

Finally, the file `words.dbx` corresponds to an inverted index as found in many traditional information retrieval systems. The inverted index represents a common data structure and is typically used to associate a word or phrase with the set of documents in which it has been found and the exact position where it occurred [18]. eXist's inverted index differs from traditional IR systems in that instead of storing the word position, we use unique node identifiers to keep track of word occurrences. By default, eXist indexes all text nodes and attribute values by tokenizing text into keywords. In `words.dbx`, the extracted keywords are mapped to an ordered list of document and unique node identifiers. The file follows the same structure as `elements.dbx`, using <collection-id, keyword> pairs for keys. Each entry in the value list points to a text or attribute node where the keyword occurred. It is possible to exclude distinct parts of a given document type from fulltext-indexing or switch it off completely.

4. Query Language Implementation

Given the index structures presented above, we are able to access distinct nodes by their unique node identifier, retrieve a list of node identifiers matching a given qualified node name or a specified keyword. The DBBroker interface has method definitions for each of these basic operations. In this section, we will explain how the available index structures are used by the query engine to efficiently process path expression queries.

eXist currently contains an experimental XPath query language processor. XPath represents a core standard for XML query processing, since it is embedded into a number of other XML query language specifications like XSLT and XQuery. eXist's XPath processor implements major parts of the XPath 1.0 standard requirements, though – at the time of writing – it is not yet complete. However, the existing functionality covers most of the commonly needed XPath expressions. Additionally, several extensions to standard XPath are available, which will be described below.

4.1. Path join algorithm

Based on the features provided by the indexing scheme, eXist's query engine is able to use fast path join algorithms to efficiently process path expressions. Several path join algorithms have been proposed in recent research: Zhang et al. [23] explored

the efficiency of traditional merge join algorithms as used in relational database systems for XML query processing. They proposed a new algorithm, multi-predicate merge join, which could outperform standard RDBMS joins.

Two families of structural join algorithms have also been proposed in [21]: Tree-merge and stack-tree. While the tree-merge algorithm extends traditional merge joins and the new multi-predicate merge join, the stack-tree algorithm has been especially optimized for path joins as used in XML query processing.

General path join algorithms based on the extended pre-order numbering scheme of XISS have been proposed and experimentally tested in [15]. Three algorithms are assigned to distinct types of subexpressions: Element-Attribute Join, Element-Element Join and Kleene-Closure Algorithm.

eXist's query processor will first decompose a given path expression into a chain of basic steps. Consider an XPath expression like

```
/PLAY//SPEECH[SPEAKER='HAMLET']
```

We use the publicly available collection of Shakespeare plays for examples [3]. Each play is divided into ACT, SCENE and SPEECH sections. A SPEECH element includes SPEAKER and LINE elements. The above expression is logically split into subexpressions as show in figure 6.

Fig. 6. Decomposition of Path Expression



The exact position of PLAY, SPEECH and SPEAKER elements is provided in the index file `elements.dbx`. To process the first subexpression, the query engine will load the root elements (PLAY) for all documents in the input document set. Second, the set of SPEECH elements is retrieved for the input documents via an index lookup from file `elements.dbx`. Now we have two node sets containing potential ancestor and descendant nodes for each of the documents in question. Each node set consists of <document-id, node-id> pairs, ordered by document identifier and unique node identifier. Node sets are implemented using Java arrays.

To find all nodes from the SPEECH node set being descendants of nodes in the PLAY node set, an ancestor-descendant path join algorithm is applied to the two sets. eXist's path join algorithm is quite similar to the algorithms presented in [15]. However, there are some differences due to the used numbering scheme.

Our ancestor-descendant-join algorithm is show in figure 7. The function expects two ordered node sets as input: the first contains potential ancestor nodes, the second potential descendants. Every node in the two input sets is described by a pair of <document-id, node-id>. The function recursively replaces all node identifiers in the descendant set with the node id of their parent node using function `get_parent_set` in the outer loop. The inner loop then compares the two node sets to find equal pairs of nodes by incrementing either `ax` or `dx` depending on the comparison. If a pair of nodes with equal document and node identifiers is found, the ancestor node and the original descendant node are copied to output. The algorithm terminates if `get_parent_set` returns false which indicates that the descendant list contains no more valid node identifiers.

Fig. 7. Ancestor-descendant Join

```
Algorithm ancestor_descendant_join(al, dl)
  /*
    Input:
      al: ordered set of potential ancestor nodes defined
          by <doc-id, node-id> pair.
      dl: ordered set of potential descendant nodes de-
fined
          by <doc-id, node-id> pair.
    Output: set of <ancestor, descendant> pairs.
  */
  dl_orig = copy of dl;

  // get_parent_set replaces each node-id
  // in dl with the node-id of its parent.
  while(get_parent_set(dl)) {
    ax = 0;
    dx = 0;
    while(dx < dl.length) {
      if(dl[dx] == null)
        dx++;
      else if(dl[dx] > al[ax]) {
        if(ax < al.length - 1)
          ax++;
        else
          break;
      } else if(dl[dx] < al[ax])
        dx++;
      else {
        output(al[ax], dl_orig[dx]);
        dx++;
      }
    }
  }
```

The outer loop is repeated until all ancestor nodes of descendants in dl are checked against the ancestor set. This way we ensure that extreme cases of element-element joins are properly processed, where a single node is a descendant of multiple ancestor nodes. The parent_child_join algorithm differs from ancestor_descendant_join in that the outer loop is only repeated once.

The result returned by the algorithm will become the context node set for the next subexpression in the chain. Thus the resulting node set for expression PLAY//SPEECH becomes the ancestor node set for expression SPEECH[SPEAKER], while the results generated by evaluating the predicate expression SPEAKER=”HAMLET” become the descendant node set.

To evaluate the subexpressions PLAY//SPEECH and SPEECH[SPEAKER], eXist does not need access to the actual DOM nodes in the XML store. Both expressions

are entirely processed on basis of the unique node identifiers provided in the index file. Additionally, the path join algorithm determines ancestor-descendant relationships for all candidate nodes in all documents in one single step.

Yet to process the equality operator in the predicate subexpression SPEAKER=”HAMLET”, the query engine will have to retrieve the actual DOM nodes to determine their value and compare it to the literal string argument. Since a node’s value may be distributed over many descendant nodes, the engine has to do a conventional tree traversal, beginning at the subexpression’s context node (SPEAKER).

This could be avoided by adding another index structure for node values. However, for many documents addressing human users, exact match query expressions could be replaced by corresponding expressions using the fulltext operators and functions described in the next section. We have thus decided to drop the value index supported by previous versions of eXist to reduce disk space usage.

4.2. Query Language Extensions

The XPath specification only defines a few limited functions to search for a given string inside the character content of a node. This is a weak point if one wants to search through documents containing larger sections of text. For many types of documents, the provided standard functions will not yield satisfying results.

eXist offers two additional operators and several extension functions to provide access to the fulltext content of nodes. For example, to select the scene in the cavern from Shakespeare’s Macbeth:

```
//SCENE[SPEECH[SPEAKER &= ’witch’ and LINE &= ‘fenny
snake’]]
```

`&=` is a special text search operator. It will select context nodes containing *all* of the space-separated terms in the right hand argument. To find nodes containing *any* of the terms the `|=` operator is used. The order of terms is not important. To match more complex string patterns, regular expression syntax is supported through additional functions.

To impose an order on search terms the `near(node set, string, [distance])` function selects nodes for which each of the terms from the second argument occur near to each other in the node's value and in correct order. A maximum distance of words may be specified by an optional third argument. To get the SPEECH fragment where Hamlet exactly uses the phrase ``cursed spite'' one may write:

```
//SPEECH[SPEAKER&='HAMLET' and near(LINE, 'cursed
spite')]
```

All fulltext-search extensions use the inverted index file `words.dbx`, which maps extracted keywords to an ordered list of document and unique node identifiers. To evaluate the near-subexpression above, the query engine will look up each of the keywords “cursed” and “spite” and retrieves a set of node identifiers for all text nodes containing the keyword. Since there is more than one search term, the obtained node

sets are merged by generating the intersection (for the &= operator and near-function) or union set (for the |= operator). Finally, the resulting set of text nodes is joined with context node set (LINE) by applying an ancestor-descendant path join algorithm as described above. Thus, while the equality operator as well as standard XPath functions like contains require eXist to perform a complete scan over the contents of every node in the context node set, the fulltext search extensions rely entirely on information stored in the index.

5. Performance and Scalability

To estimate the efficiency of eXist's indexing and query processing some experimental results are provided in this section. We compare overall query execution times for eXist, Apache's Xindice [1] and an external XPath query engine [16] which is based on a conventional tree-traversal based approach. In a second experiment we process the same set of queries with increasing data volumes to test the scalability of eXist.

We have chosen a user-contributed data set with 39.15 MB of XML markup data containing 5000 documents taken from a movie database. Each document describes one movie, including title, genre, ratings, complete casts and credits, a summary of the plot and comments contributed by reviewers. Document size varies from 500 bytes to 50 KB depending on the number of credits and comments. Experiments were run on a PC with AMD Athlon 4 processor with 1400 MHZ and 256 MB memory running Mandrake Linux 8.2 and Sun's Java Development Kit 1.4.

We formulated queries for randomly selected documents which might typically be of interest to potential users. For example, we asked for the titles of all western movies or films with certain actors or characters.

The Jaxen XPath engine [16] has been selected to represent a conventional, top-down tree-traversal based query engine. Jaxen builds on a generalized tree walking model to support processing of XPath expressions for several document object models. For our experiment, Jaxen runs on top of eXist's persistent DOM implementation using standard DOM methods to navigate the document tree. Additionally, we have processed the same set of queries with an alternative Open Source native XML database, Apache's Xindice. Since Xindice requires manual index creation, we have defined an index on every element referenced by our queries. Our test client used the XML:DB API to access Xindice as well as eXist.

Each query in the set has been repeated 10 times for each test run to allow B+-Tree page buffers to come into effect. This corresponds to normal database operation where the database server would run for a longer period of time with many users doing similar queries with respect to input document sets and element or attribute selections. Xindice and eXist use the same B+-Tree code base. Running on top of eXist's persistent DOM, Jaxen equally benefits from eXist's page buffering mechanisms. Repeated tests are also required because of high variation between single test runs due to optimizations done by the Java virtual machine.

As described above, eXist does not create an index on element and attribute values. We have thus replaced all exact match expressions by equivalent fulltext search expressions and made a second test run. For example, the XPath expression

//movie[.//credit='Gable, Clark'] has been reformulated as follows: //movie[near(.//credit, 'Gable, Clark')]. Both sets of queries are equivalent in that they will generate the same number of hits for our data set.

Average query execution times for selected queries are shown in table 1. Execution times for retrieving result sets have not been included. They are the same for the eXist-based approaches. Retrieving results merely depends on the performance of eXist's serializer, which has no connection to the query engine.

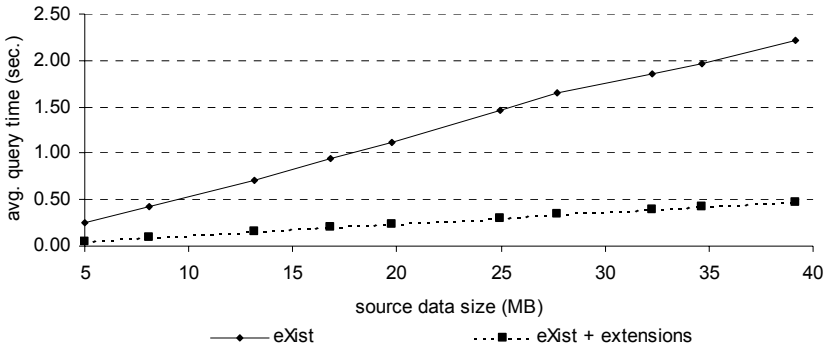
Table 1. Avg. query execution times for selected queries (in seconds)

XPath Query	eXist	eXist + extensions	Xindice	Jaxen
/movie[.//genre='Drama']//credit[@role='directors']	3.44	1.14	10.62	21.86
/movie[genres/genre='Western']/title	0.79	0.23	1.39	7.58
/movie[languages/language='English']/title	1.45	0.97	34.18	8.50
/movie[.//credit/@charactername='Receptionist']	3.12	0.21	27.04	51.48
/movie[contains(.//comment, 'predictable')]	2.79	0.20	25.75	31.49
/movie[.//credit='Gable, Clark']	4.47	0.35	0.38	33.72
/movie[.//languages/language='English']/title[starts-with(.,'42nd Street')]	1.63	0.32	17.47	32.64
/movie[languages/language='English' and credits/credit='Sinatra, Frank']	5.16	0.58	0.11	13.26

Our results show that eXist's query engine outperforms the tree-traversal based approach implemented by Jaxen by an order of magnitude. This supports previous research results indicating the superiority of path join algorithms [15, 21, 23]. It is also no surprise that search expressions using the fulltext index perform much better with eXist than corresponding queries based on standard XPath functions and operators. Results for Xindice are rather hard to interpret, because we observe high variation between different queries. Further experiments would be needed to explore this effect.

In a second experiment, the complete set of 5000 documents was split into 10 subcollections. To test scalability we added one more subcollection to the database for each test sequence and computed performance metrics for eXist with the standard XPath and extended XPath query sets. Thus the raw XML data size processed by each test cycle increased from 5 MB for the first collection up to 39.15 MB for 10 collections. As before, each query has been repeated 10 times. Average query execution times for the complete set of queries are shown in figure 8.

Fig. 8. Avg. query execution times by source data size



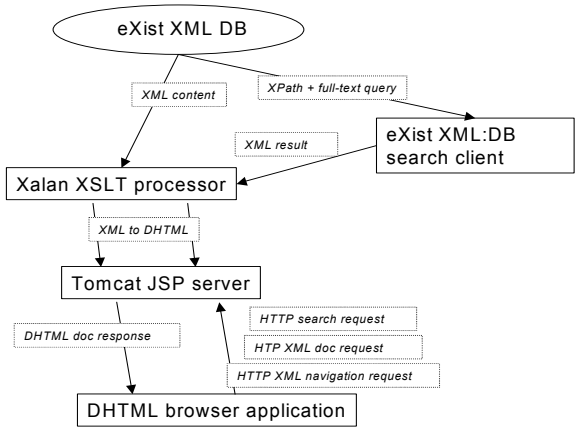
We observe for both sets of queries that query execution times increase at least linearily with increasing source data size. Thus our experiment shows linear scalability of eXist's indexing, storage and querying architecture.

6. Applications

Despite it's relative short project history, eXist's user community has been growing continously. A number of projects proves that eXist may be used effectively to implement XML based applications and services in a wide range of application scenarios, dealing with different types of XML documents and data volumes. XML document types range from highly structured technical documentation to literary works encoded in TEILite. In this section we would like to point out some example applications, showing the variety of application scenarios:

The italian automobile manufacturer, FIAT, currently deploys eXist as part of a **multi-lingual technical documentation publishing system** (GlobalGate) which covers technical maintenance documentation for several car models by FIAT, ALFA ROMEO and LANCIA. The system, developed by Xerox Global Services, is targeted at technical maintenance operators of authorized dealers in Europe. The maintenance documentation produced by FIAT technical writers and translated into most European languages is stored into eXist to generate a static database instance.

Fig. 9. GlobalGate Web-Publishing Architecture



The generated database and the application logic are deployed in a number of ways: it might either be served by existing webservers, installed on a local hard disk or written to CD. The application has been implemented using JSP, XSLT and the XML:DB API for access to eXist (see figure 9). For the CD version, eXist and a Tomcat webserver run entirely from CD. Hardware requirements are low: Pentium 200 MHZ, 64 MB RAM, WIN 95.

The raw XML data size for each car model and language is 12 MB on average. 10 languages are available on one CD, stored in different collections. On the minimal hardware platform query response times are reported to range from 3 to 5 seconds. FIAT has forecast a production of 15.000 CDs to be delivered to authorized dealers.

The **Anglo-Norman On-Line Hub** [28], a project funded by the United Kingdom Arts and Humanities Research Board and in cooperation with the Universities of Swansea and Abertystwyth, will provide access to a digitised Revision of the Anglo-Norman Dictionary, including the corpus on which this dictionary is based as well as scholary works related to Anglo-Norman. eXist is used as the core indexing and retrieval mechanism for the entire project. A first experimental interface to access the scholary articles is available to users. A more sophisticated user interface is currently restricted to the editors and project participants. It allows to search the actual dictionary entries by citation source, useage domain etc.. XML markup conforms to TEI Guidelines.

As a final example, a **Privacy Management Architecture** (PRIMA) [9] has been developed by the Information Technology Transfer Office at Darmstadt Technical University. PRIMA currently provides a personal data manager and data recorder, implemented as proxy, which records all user interactions with service providers. The user can view and manage the personal data he has given to service providers. PRIMA is based on open Internet standards like HTTP, XML, SSL and P3P. eXist is used as the database backend to store P3P data.

7. Outlook

Despite the many projects already using eXist, there is still much work to be done to implement outstanding features and increase usability and interoperability. We are currently concentrating on complete XPath support, possibly using existing implementations developed by other projects.

Some types of scholary applications also request an option to display keywords in context. This requires a more sophisticated transformation of query results, for example, as specified in the XQuery standard [6]. XQuery also supports much more complex user queries. With our current implementation, complex queries involving multiple source documents have to be split manually into several XPath expressions.

Due to index generation a document stored in eXist occupies more disk space than the original raw XML file. The average expansion factor is approx. 2.5. This is a major restriction for applications which need to run in a limited environment like a CD. Since most of the space is occupied by B+-tree pages, there is still some optimization potential, e.g. by reducing the storage size used for B+-tree keys and page pointers.

Another topic is XUpdate - a standard proposed by the XML:DB initiative for updates of distinct parts of a document [25]. eXist does currently not provide an advanced update mechanism. Documents may only be updated as a whole. While this is a minor problem for applications dealing with relatively static document collections, it represents a major limitation for applications which need to frequently update rather large documents.

As explained above, the numbering scheme could be extended to avoid a frequent reordering of node identifiers on document updates by introducing sparse identifiers between nodes [15]. Support for multiversion documents using durable node numbers has been proposed in [7].

While support for ranked keyword searches has so far not been demanded by users, term-weighting and ranking schemes compatible with our indexing approach have already been proposed in [20].

8. References

[1] The Apache Group. "Xindice Native XML Database". http://xml.apache.org/xindice.

[2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. "XML Path Language (XPath) 2.0. W3C Working Draft 30 April 2002. http://www.w3.org/TR/xpath20. Working Draft, 2002.

[3] John Bosak. XML markup of Shakespeare's plays, January 1998. http://ibiblio.org/pub/sun-info/standards/xml/eg/.

[4] Ronald Bourret. "XML and Databases". http://www.rpbourret.com/xml/XMLDatabasesProds.htm.

[5] D. Chamberlin, J. Robie, D. Florescu. “Quilt : An XML Query Language for Heterogeneous Data Sources”, Proc. Of WebDB, 2000.

[6] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Siméon and M. Stefanescu. “XQuery 1.0: An XML Query Language”. http://www.w3.org/TR/xquery. W3C Working Draft, W3C Consortium, December 2001.

[7] Shu-Yao Chien, Vassilis J. Tsotras, Carlo Zaniolo, and Donghui Zhang. “Efficient Complex Query Support for Multiversion XML Documents”. In *Proceedings of the EDBT Conference*, 2002.

[8] James Clark, and Steve DeRose. “XML Path Language (XPath) Version 1.0”. W3C Recommendation 16 November 1999. http://www.w3.org/TR/xpath. W3C Recommendation, W3C Consortium, 1999.

[9] Darmstadt University of Technology, IT Transfer Office. “PRIMA – Privacy Management Architecture”. http://www.ito.tu-darmstadt.de/PRIMA/.

[10] Daniela Florescu and Donald Kossmann. “A performance evaluation of alternative mapping schemes for storing XML data in a relational database”. Technical report, INRIA, Rocquencourt, France, 1999.

[11] Y. Goland, E. Whitehead, A. Faizi, S. Carter and D. Jonsen. “HTTP Extensions for Distributed Authoring – WEBDAV. RFC2518”. http://www.ietf.org/rfc/rfc2518.txt. Technical report.

[12] Torsten Grust, “Accelerating XPath Location Steps”. In *Proceedings of the 21st ACM SIGMOD Int’l Conference on Management of Data (SIGMOD 2002)*, June 2002, Madison, Wisconsin, USA.

[13] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. “Index Structures for Structured Documents”. In *Proceedings of the 1st ACM International Conference on Digital Libraries,* March 20-23, 1996, Bethseda, Maryland, USA. ACM Press, 1996.

[14] A. Le Hors, P. Le Hegaret, G. Nicol, J. Robie, M. Champion and S. Byrne. “Document Object Model (DOM) Level 2 Core Specification Version 1.0”. http://www.w3.org/TR/DOM-Level-2-Core/. W3C Recommendation, Nov. 2000.

[15] Quanzhong Li and Bongki Moon. “Indexing and Querying XML Data for Regular Path Expressions”. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Databases*, September 11-14, 2001, Roma, Italy.

[16] Bob McWhirter and James Strachnan. “Jaxen: Universal XPath Engine”. http://www.jaxen.org.

[17] David Megginson. “SAX: Simple API for XML”. http://sax.sourceforge.net/.

[18] G. Salton and M. J. McGill. “Introduction to Modern Information Retrieval”. McGraw-Hill, New York, 1983.

[19] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt and J. Naughton. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *Proceedings of the VLDB Conference*, Edinburgh, Scotland, 1999.

[20] Dongwook Shin, Hyuncheol Jang, and Honglan Jin. "BUS: An Effective Indexing and Retrieval Scheme in Structured Documents". In *Proceedings of the 3rd ACM International Conference on Digital Libraries*, June 23-26, 1998, Pittsburgh, PA, USA. ACM Press, 1998.

[21] Divesh Srivastava, Shurug Al-Khalifa, H.V. Jagadish, Nick Koudas, Jignesh M. Patel, and Yuqing Wu. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching". In *Proceedings of the ICDE Conference*, 2002.

[22] Dave Winer. "XML-RPC Specification". http://www.xmlrpc.com/spec. Technical report, 1999.

[23] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohmann. "On Supporting Containment Queries in Relational Database Management Systems". In *Proceedings of the SIGMOD Conference*, 2001, Santa Barbara, California, USA.

[24] The XML:DB Project. "Frequently Asked Questions". http://www.xmldb.org/faq/.

[25] The XML:DB Project. "XUpdate Working Draft". http://www.xmldb.org/xupdate/. Technical report, 2000.

[26] The XML:DB Project. "XML:DB Database API Working Draft". http://www.xmldb.org/xapi/. Technical report, 2001.

[27] XML Protocol Working Group. "SOAP Version 1.2 Specification". http://www.w3.org/2000/xp/Group.

[28] "The Anglo-Norman On-Line Hub". http://www.anglo-norman.net.