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, 6], 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. eXist 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. For developers, access through HTTP, XML-RPC, SOAP and WebDAV is provided. Java applications may use the XML:DB API [18], a common interface for access to native or XML-enabled databases, which has been proposed by the vendor independent XML:DB initiative.

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, for example, as a core retrieval component in a multi-lingual technical documentation publishing system, containing technical maintenance documentation for several car models produced by an italian manufacturer.

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.
2. 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. Contrary to other proposals, which focus on the efficient processing of ancestor-descendant relationships, our indexing scheme supports all axes of navigation as required by the XPath specification.
3. While previous contributions have indicated the superiority of path join algorithms over conventional tree traversals for a limited set of expressions [10, 15, 16], 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: The next section 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 3 will then explain how the numbering scheme and the created index structures are used in query processing. In Section 4 we finally present some experimental results to estimate the efficiency and scalability of eXist's indexing architecture and query engine.

2. 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.

2.1. Background

XML query languages like XPath [2, 6] or XQuery [4] 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" ele-

ment. 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 relationsships 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 [10, 15, 16]. 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.

2.2. 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 [5, 8, 10, 14, 15, 16]. 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 [16] (also discussed in [15]). 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 (D1, S1:E1, L1) is a descendant of a node y with 3-tuple (D2, S2: E2, L2) if and only if D1 = D2; S1 < S2 and E2 < E1.

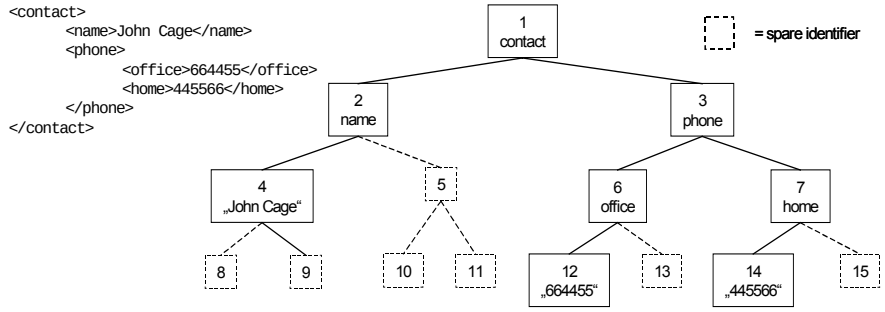
The XISS system ([10], also discussed in [5]) proposes an extended preorder numbering scheme. This scheme assigns a pair of numbers <order, size> to each node, such that: (i) for a tree node y and its parent x, order(x) < order(y) and order(y) + size(y) ≤ order(x) + size(x) and (ii) for two sibling nodes 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 [5] to assign durable identifiers keeping track of document changes.

Lee et al. [8] 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 1 shows the identifiers assigned to the nodes of a very simple XML document, which is modelled as a complete 2-ary tree.

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.

Fig. 1. Unique identifiers assigned by the level-order numbering scheme



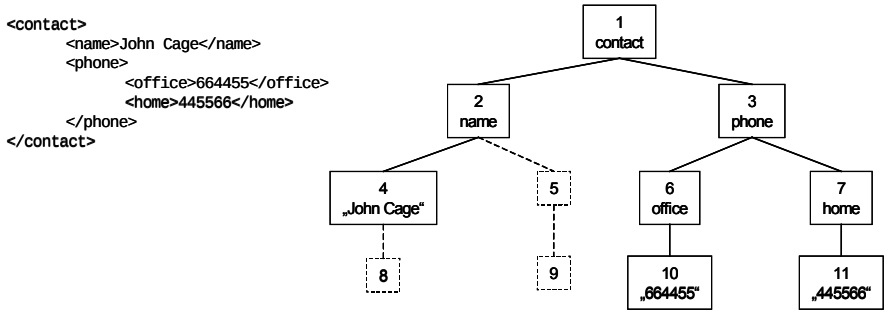
The unique identifiers generated by this numbering scheme have some important properties: from a given identifier one may easily determine the id of it's parent, sibling or possible child nodes. For example, for a k-ary document tree we may obtain the identifier of the parent node of a given node whose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

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

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 2 shows the unique identifiers generated by eXist for the same document as above.

Fig. 2. 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 [5] and [10]. However, eXist does currently not provide an advanced update mechanism as defined, for example, by the XUpdate standard [17]. 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. We have already started to simplify the generated index structures (see below) as a prerequisite for a future XUpdate implementation.

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.

2.3. Index and Data Organization

In this section we provide some implementation details concerning 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:

- `collections.dbx` manages the collection hierarchy
- `dom.dbx` collects nodes in a paged file and associates unique node identifiers to the actual nodes
- `elements.dbx` indexes elements and attributes
- `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.

Users will usually query entire collections or even several collections at once. In this case, just a single index lookup is required to retrieve relevant index entries for the entire collection. This results in a considerable performance gain for queries spanning multiple collections. We provide some details on each index file in the following paragraphs:

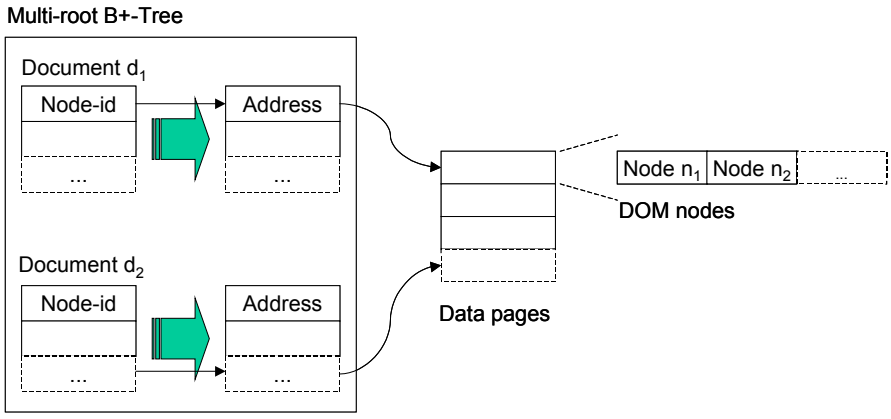
The index file `collections.dbx` manages the collection hierarchy and maps collection names to collection objects. Due to performance considerations, document descriptions are always stored with the collection object they belong to. A unique id is assigned to each collection and document during indexing.

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) [9]. The data store is backed by a multi-root B+-Tree in the same file to associate the unique node identifiers of top-level elements in a given document to the node's storage address in the data section (see figure 3).

Only top-level elements are indexed by the B+-tree. Attributes, text nodes and elements at lower levels of the document's node hierarchy are just written to the data pages without adding a key in the B+-tree. Access to these types of nodes is provided by traversing the nearest available ancestor found in the tree. However, the cases where direct access to these nodes is required are very rare. The query engine will process most types of XPath expressions without accessing dom.dbx.

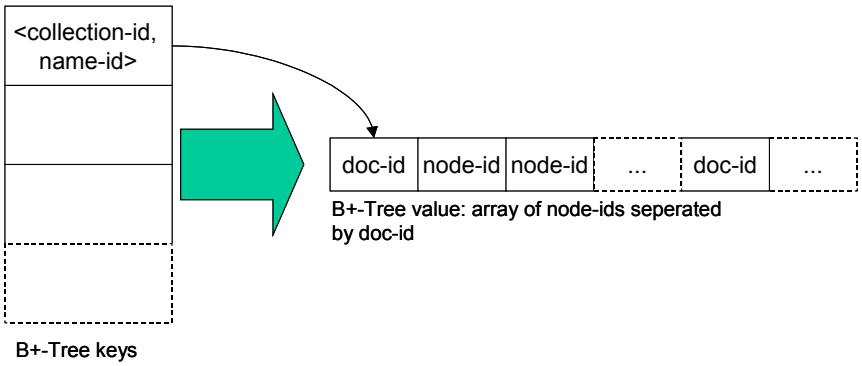
Fig. 3. XML Data Store Organization



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. As a result, the storage size of a document in dom.dbx will likely be smaller than the original data source size for larger documents.

Since nodes are stored in document order, only a single initial index lookup is required to serialize a document or fragment. eXist's serializer will generate a stream of SAX [12] events by sequentially walking nodes in document order, beginning at the fragment's root node.

Fig. 4. Index Organization for Elements and Attributes



Element and attribute names are mapped to unique node identifiers in file `elements.dbx`. 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 requires a single index lookup to retrieve the complete set of node identifiers pointing to chapter elements.

Since the sequence of document and node ids consists entirely of integer values, it is stored in a combination of delta and variable-byte coding to save storage space.

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 [13]. 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.

3. 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. 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.

3.1. Path join algorithm

Based on the features provided by the indexing scheme, eXist's query engine is able to use path join algorithms to efficiently process path expressions. Several path join algorithms have been proposed in recent research: Zhang et al. [16] 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 [15]: 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 [10]. 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 5.

Fig. 5. 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 algorithms are quite similar to those presented in [10]. However, there are some differences due to the used numbering scheme.

We concentrate on the ancestor-descendant-join as shown in figure 6. 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 id of their parent using function get_parent_set in the outer loop. The inner loop then compares the two sets to find equal pairs of nodes by incrementing either ax or dx depending on the comparison. If a matching pair of nodes is found, ancestor and 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. 6. Ancestor-Descendant Join

```
Algorithm ancestor_descendant_join(al, dl)
  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 generated node set 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 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, 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.

3.2. 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 near(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. Both operators support wildcards in the search terms. 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. To match more complex string patterns, regular expression syntax is supported through additional functions.

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. 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.

4. 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 [11] 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 [11] has been selected to represent a conventional, top-down tree-traversal based query engine. For our experiment, Jaxen runs on top of eXist's persistent DOM implementation. Additionally, we processed the same set of queries with an alternative native XML database, Apache's Xindice. Since Xindice requires manual index creation, we 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 page buffering mechanisms.

As described above, eXist does not create an index on element and attribute values. For a second test run, we thus replaced all exact match expressions by equivalent fulltext search expressions. For example, the expression //movie[.//credit='Gable, Clark'] has been reformulated as follows: //movie[near(.//credit, 'Gable, Clark')]. Both sets of queries are equivalent with respect to the generated 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)

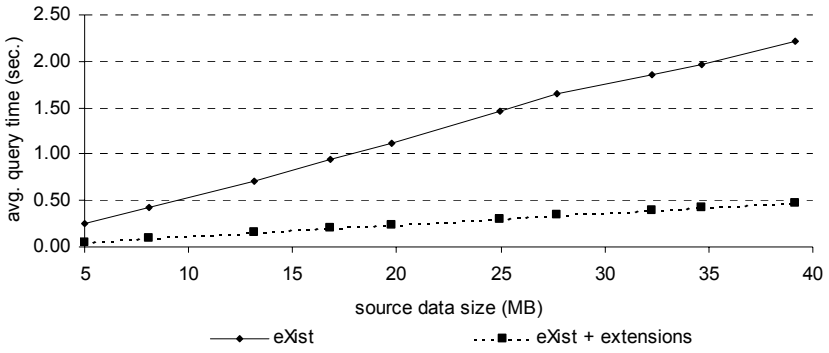
XPath Query	eXist	eXist + extensions	Xindice	Jaxen
/movie[.//genre='Drama']//credit[@role='directors']	3.44	1.14	10.62	21.86
/movie[genres/genre='Western']/title	0.79	0.23	1.39	7.58

/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 [10, 15, 16]. 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 show that selections on the descendant axis (using the // symbol) are not very well supported by their XPath implementation. Contrary to Xindice, eXist handles these types of expressions efficiently.

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 7.

Fig. 7. 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.

5. 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. Some of eXist's weak points – namely indexing speed and storage requirements - have already been subject to a considerable redesign. We are currently concentrating on complete XPath support, possibly using existing implementations developed by other projects.

Another important topic is XUpdate - a standard proposed by the XML:DB initiative for updates of distinct parts of a document [17]. 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 portions of 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 [5, 10]. The necessary changes to handle sparse identifiers have already been implemented. We have also started to simplify the created index structures, making them easier to maintain on node insertions or removals. However, some work remains to be done on these issues.

Additionally, support for multiversion documents using durable node numbers has been proposed in [5]. The scheme described there could also be implemented for eXist.

Being an open source project, eXist strongly depends on user feedback and participation. Interested developers are encouraged to join the mailing list and share their views and suggestions.

6. 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] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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