

Index-Driven XQuery Processing in the eXist XML Database

Wolfgang Meier

wolfgang@exist-db.org

The eXist Project

XML Prague, June 17, 2006

Outline

- 1 Introducing eXist
- 2 Node Identification Schemes and Indexing
 - Introduction
 - Node Identification Schemes in eXist
 - Updating Nodes via XUpdate and XQuery Update
- 3 XQuery Processing
 - Index Usage and Structural Joins
 - Consequences for Query Processing
- 4 Outlook

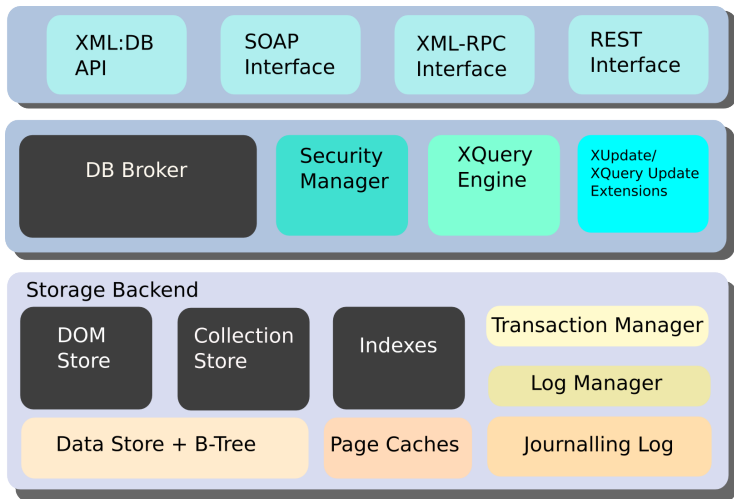
Project History

- Summer 2000: eXist is born!
- First experiments to implement an indexing scheme for XML on top of a relational DBS

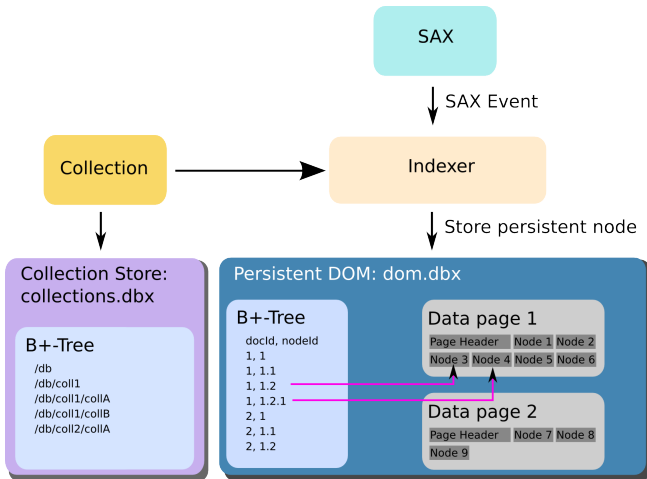
Inspired by:

D. Shin, H.Jang, H.Jin: “Bus: An Effective Index and Retrieval Scheme in Structured Documents”. In Proceedings of the 3rd ACM International Conference on Digital Libraries, 1998, Pittsburgh, PA.

Architecture Overview



Document Storage



XQuery Implementation

- eXist provides its own XQuery implementation
- Efficiency depends on the indexing system
- Basic processing logic is sometimes quite different from in-memory XQuery processors

eXist's approach to XQuery can not be understood without knowing the indexing system and vice versa!

Outline

- 1 Introducing eXist
- 2 Node Identification Schemes and Indexing
 - Introduction
 - Node Identification Schemes in eXist
 - Updating Nodes via XUpdate and XQuery Update
- 3 XQuery Processing
 - Index Usage and Structural Joins
 - Consequences for Query Processing
- 4 Outlook

Node Identification Schemes

- Every node in the tree is labelled with a **unique identifier**
- Quick identification of **structural relationships** between a set of given nodes
- Direct access to nodes by their unique identifier
- **Reduce I/O** operations by deciding XPath expressions based on node IDs and indexes

Basic Operations

Decision: for two given nodes, decide if they are in a specific relationship, e.g. parent/child, ancestor/descendant, preceding-sibling/following-sibling.

Reconstruction: for a given node, determine the IDs of nodes in its neighbourhood, e.g. parent, next-sibling, first/last child. Some identification schemes don't support reconstruction very well.

Distinguishing Features of Node ID Schemes

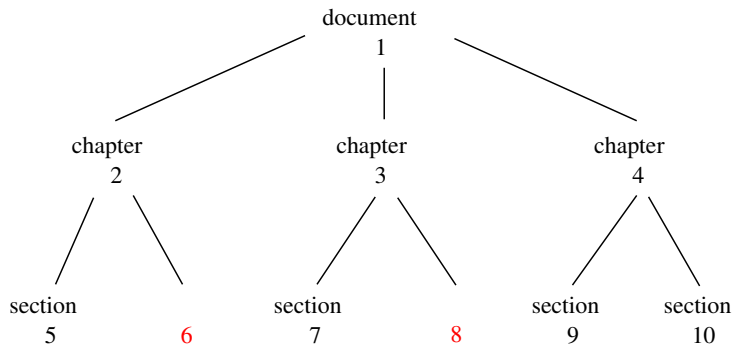
- Numeric vs. path-based IDs
- Fixed vs. variable size
- Supported axes for decision and reconstruction
- Frequency of reindex operations after tree changes (update-friendly vs. static)

Outline

- 1 Introducing eXist
- 2 Node Identification Schemes and Indexing
 - Introduction
 - **Node Identification Schemes in eXist**
 - Updating Nodes via XUpdate and XQuery Update
- 3 XQuery Processing
 - Index Usage and Structural Joins
 - Consequences for Query Processing
- 4 Outlook

Extended Level-Order Numbering

Figure: Node IDs assigned by extended level-order numbering



Benefits

- Simple arithmetic computation
- Determine the relationship between any two given nodes (decision)
- From a given ID we can reconstruct the IDs of all neighbours
- Works for all XPath axes, including child, descendant, ancestor, parent . . .
- Space efficient: as all IDs can be reconstructed, we don't need to store node IDs in the DOM

Disadvantages

- Sparse encoding: need to insert “virtual nodes” due to completeness constraint
- eXist did already raise the document size limit considerably, but there’s still a:

Document size limit: if the resulting tree is rather imbalanced, the indexer may run out of available IDs at some point, even when using 64bit integer IDs

- Document size limit depends on tree structure and is hard to predict
- Not **update friendly**: node insertions may trigger a complete renumbering of the tree

Switching to a New Scheme

- Work to replace level-order numbering started in Dec. 2005
- Sponsored by the University of Victoria node of the Canadian TAPoR consortium
- Redesign of the indexing core of eXist
- Basic query logic remains the same, so XQuery implementation is only partially affected
- First public release scheduled for July 1

Dynamic Level Numbering (DLN)

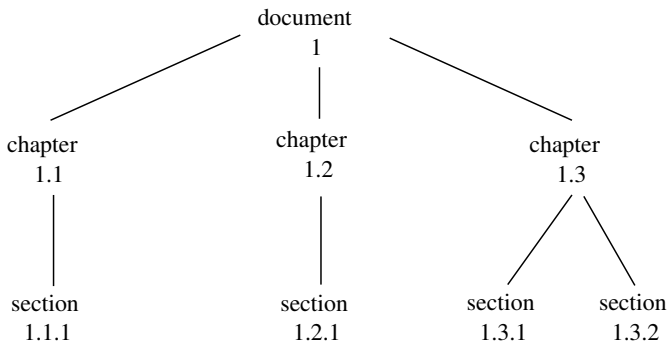
- Path-based identification scheme
- Hierarchical, variable length IDs
- Inspired by Dewey's decimal classification
- Node IDs consist of the ID of the parent node as prefix and a level value: 1, 1.1, 1.2, 1.2.1 ...

Article:

Böhme, T.; Rahm, E.: Supporting Efficient Streaming and Insertion of XML Data in RDBMS. Proc. 3rd Int. Workshop Data Integration over the Web (DIWeb), 2004

Dynamic Level Numbering

Figure: Node IDs assigned by DLN



Main Benefits

- No conceptual limit on document size due to variable-size IDs
- Deciding the relation between any two given nodes is a trivial operation
- DLN is a very **update-friendly** scheme (contrary to most other schemes, see below)

Problems

- Depending on nesting depth, IDs can become very long
- Main challenge: find an efficient binary encoding

Example: IDs Picked From a Real TEI Document

```

1 <div6 exist:id="1.46.9.7.8" xml:id="JG10229">
2   <head exist:id="1.46.9.7.8.7">
3     <name exist:id="1.46.9.7.8.7.3" type="Dramenfigur">FAUST</name>
4     <stage exist:id="1.46.9.7.8.7.4">unruhig</stage>
5     <stage exist:id="1.46.9.7.8.7.6"> auf seinem Sessel am Pulten</stage>
6   </head>
7   <sp exist:id="1.46.9.7.8.8" xml:id="JG10230">
8     <lg exist:id="1.46.9.7.8.8.4">
9       <l exist:id="1.46.9.7.8.8.4.7" n="1" part="N">Hab nun ach die Philosophie</l>
10      <l exist:id="1.46.9.7.8.8.4.8" part="N">Medizin und Juristerey ,</l>
11      <l exist:id="1.46.9.7.8.8.4.9" part="N">Und leider auch die Theologie</l>
12      <l exist:id="1.46.9.7.8.8.4.10" part="N">
13        Durchaus<note exist:id="1.46.9.7.8.8.4.10.4" place="unspecified" anchored="yes">
14          <hi exist:id="1.46.9.7.8.8.4.10.4.4">Durchaus]</hi> Vollständig.</note>
15          studirt mit heisser Müh.
16        </l>
17      </lg>
18    </sp>
19 </div6>

```

Binary Encoding of a Level Value

- Variable-length encoding using fixed-size units (currently: 4 bits)
- Efficient for streamed data: fan-out does not need to be known in advance
- Start with a minimal number of bits, further units are added as needed
- Highest 1-bits indicate number of units used

Stream Encoding of a Single Level Value

Units	Bit pattern	ID range
1	0XXX	1..7
2	10XX XXXX	8..71
3	110X XXXX XXXX	72..583
4	1110 XXXX XXXX XXXX	584..4679
5	1111 0XXX XXXX XXXX XXXX	4680..37447
6	1111 10XX XXXX XXXX XXXX XXXX	37448..299591

DLN Encoding

A DLN is encoded as a sequence of stream-encoded level values separated by a 0-bit.

ID	Bit string	Bits
1.3	0001 0 0011	9
1.80	0001 0 1100 0000 1001	17
1.10000.1	0001 0 11110001010011001001 0 0001	30

Node Set Processing

- Higher **computational costs**: comparing two DLN IDs is slower than comparing two level-order IDs (long integer)
 - Many operations need to parse the bit string
- ⇒ Be careful with ID comparisons; avoid unnecessary sort operations

Processing Example: DLN.isChildOf()

```
1 public boolean isChildOf(NodeId parent) {  
2     DLN other = (DLN) parent;  
3     if (!startsWith(other))  
4         return false;  
5     int levels = getLevelCount(other.bitIndex + 2);  
6     return levels == 1;  
7 }
```

Outline

- 1 Introducing eXist
- 2 Node Identification Schemes and Indexing
 - Introduction
 - Node Identification Schemes in eXist
 - Updating Nodes via XUpdate and XQuery Update
- 3 XQuery Processing
 - Index Usage and Structural Joins
 - Consequences for Query Processing
- 4 Outlook

Node Updates

- Using level-order numbering, inserting/updating a node is very slow
 - We need to renumber and reindex all nodes following the insertion point
 - Using more spare IDs does just delay the reindex
- ⇒ **Bad scalability**: reindex takes longer and longer as the document grows

Benefits of DLN

Using DLNs, we can now completely avoid renumbering the document tree!

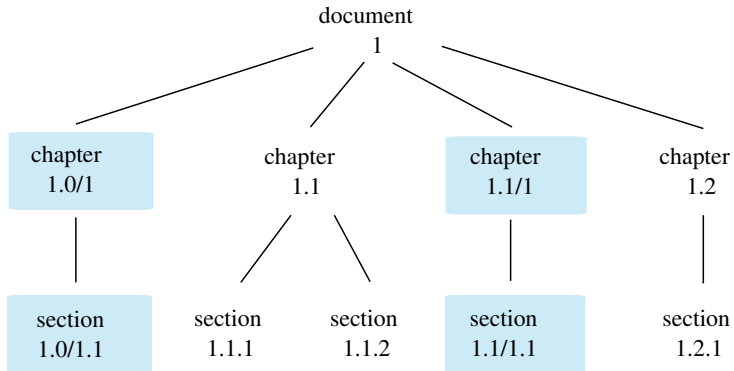
- DLN introduces subvalues in addition to the level values
- Between two nodes 1.1 and 1.2, a new node can be inserted as 1.1/1
- 1.1 and 1.1/1 are on the same level of the tree
- To insert a node before 1.1, we assign level-value 0, i.e. 1.0/1
- The next node inserted after 1.0/1 then gets ID 1.0/0/1

Applying an XUpdate

```
1 <xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
2   <xu:insert-before select="/document/chapter[2]">
3     <xu:element name="chapter">
4       <xu:element name="section"/>
5     </xu:element>
6   </xu:insert-before>
7
8   <xu:insert-before select="/document/chapter[1]">
9     <xu:element name="chapter">
10      <xu:element name="section"/>
11    </xu:element>
12  </xu:insert-before>
13 </xu:modifications>
```

Node Update Example

Figure: Node tree after updates



= inserted nodes

Binary Encoding of Subvalues

- To store a DLN with subvalue, we use a 1-bit to separate subvalues
- Level values are separated with a 0-bit as before
- Example: 1.1/7 is encoded as 0001 0 0001 1 1000

Update Problems

- Repeatedly insert a node in front of the first child of an element
- IDs grow very fast: 1.1/0/1, 1.1/0/0/1, 1.1/0/0/0/1
- To handle this edge case, eXist triggers a defragmentation run after several insertions
- Defragmentation is required here anyway to reduce the growth of dom.dbx

Outline

- 1 Introducing eXist
- 2 Node Identification Schemes and Indexing
 - Introduction
 - Node Identification Schemes in eXist
 - Updating Nodes via XUpdate and XQuery Update
- 3 XQuery Processing
 - **Index Usage and Structural Joins**
 - Consequences for Query Processing
- 4 Outlook

General Principles

Access to the persistent DOM is always expensive!

- Try to process expressions without loading the actual DOM node
- Avoid traversals of the DOM tree

Most optimizations in eXist try to execute the query in such a way that a structural joins can be applied to node sets

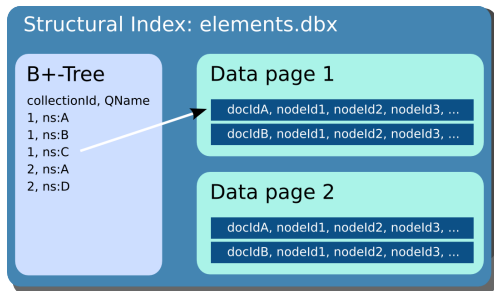
Node References

Every node in eXist is identified by a tuple

`<docId, nodeId>`

- `docId` unique identifier for the document
- `nodeId` ID assigned by a level-order traversal of the document tree

Structural Index



- Maps element and attribute QNames to a list of docId, nodeId
- Created by default for every element or attribute in a document

Structural Joins

Example: `a/descendant::b`

If all IDs of `a` and `b` nodes in the DB are known, a **structural join** operation can be applied on the two node sets to evaluate the path expression.

- Based on **decision** as well as **reconstruction**
- Join only compares the node IDs
- No access to the persistent DOM
- Structural joins replace tree traversals

Structural Join

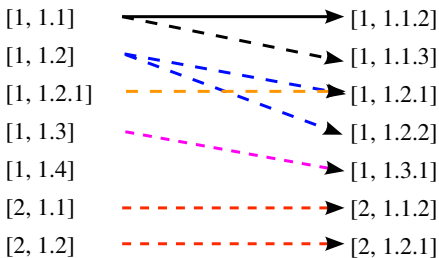
A/descendant-or-self::B

Ancestors: A

[docId, nodeId]

Descendants: B

[docId, nodeId]



Strategies

Depending on the context of the query, we can use different strategies to process a ancestor-descendant join:

- 1 Iterate through the descendant list, recursively reconstruct the parent ID for every descendant and check if it is contained in the ancestor list
- 2 Iterate through the ancestor list and check if it has any descendants in the descendant list

For the DLN scheme, strategy 2 can be more efficient than 1.

Consequences for Query Processing

Axes

For an explicit node selection by QName, `A//B` should be as fast as `A/B` or even `A/ancestor::B`

- `A/B/C/D` will usually be slower than `A//D`
- Node join algorithm itself consumes less time than the index lookups

Consequences for Query Processing and Optimization

Optimization

Performance is best if the query engine can process a path expression on two given node sets in one, single operation!

- A join can typically handle the entire context sequence at once
- Context sequence may contain nodes from different documents in different db collections

Some Optimization Examples

Predicates and comparisons

```
//A[B = 'C']
```

The predicate is processed as a set operation, filtering out all A nodes for which the general comparison did not return a match. If a range index is defined on A, the general comparison is evaluated in a single index lookup.

Some Optimization Examples

Wildcard Tests

```
//A/*[B = 'C']
```

eXist defers the evaluation of the wildcard test until the predicate has been evaluated. Nodes not matching the wildcard are filtered out. No access to the persistent DOM needed!

Further Consequences

eXist prefers XPath predicate expressions over an equivalent FLWOR construct using a “where” clause!

- Many users tend to formulate SQL-style queries
- “for” expression forces the query engine into a step-by-step iteration over the input sequence: possible optimizations are lost
- Whenever possible, the query engine will process a “where” clause like an equivalent XPath with predicate

Example

```
for $i in //entry where $i/@type = 'subject'  
or $i/@type = 'definition' or $i/@type =  
'title' return $i
```

could be rewritten as:

```
//entry[@type = ('subject', 'definition',  
'title')]
```


Example

```
1 for $section in collection("/db/articles")//section
2 for $match in $section//p[contains(., 'XML')]
3 return
4   <match>
5     <section>{$section/title/text()}</section>
6     {$match}
7   </match>
```

Faster formulation

```
1 for $match in collection("/db/articles")//section//p[contains(., 'XML')]
2 return
3   <match>
4     <section>{$section/ancestor::title/text()}</section>
5     {$match}
6   </match>
```

Outlook: Indexing

- Indexing system needs to be more modularized
- Decouple index create and maintenance from the db core
- Plug in new index types (spatial indexes, n-gram. . .)
- Better design, so indexes can provide relevant information to query optimizer

Outlook: Query Optimization

- Currently, most optimizations are implicit
- “Query plan” is hard-coded into the query engine
- Hard to maintain/debug/profile

eXist needs a better, query-rewriting optimization engine!

Query Rewriting

- Exploit the fact that eXist supports fast evaluation along the ancestor or parent axes
- Move higher selective subexpressions to the front
- Reduce size of node sets to be processed
- Reduce I/O operations

There's still a huge potential for query optimization!

Thank you!

Website:

<http://exist-db.org>

Contact:

Email: wolfgang@exist-db.org

IRC <http://irc.exist-db.org>