www.mas-ict.ch

Hes·so
Haute Ecole Spécialisée de Suisse occidentale

Module: XML et les bases de données

An Open Source Native XML Database

Houda Chabbi Drissi

houda.chabbi@hefr.ch

Source: exist.org - Meir papers

eXist

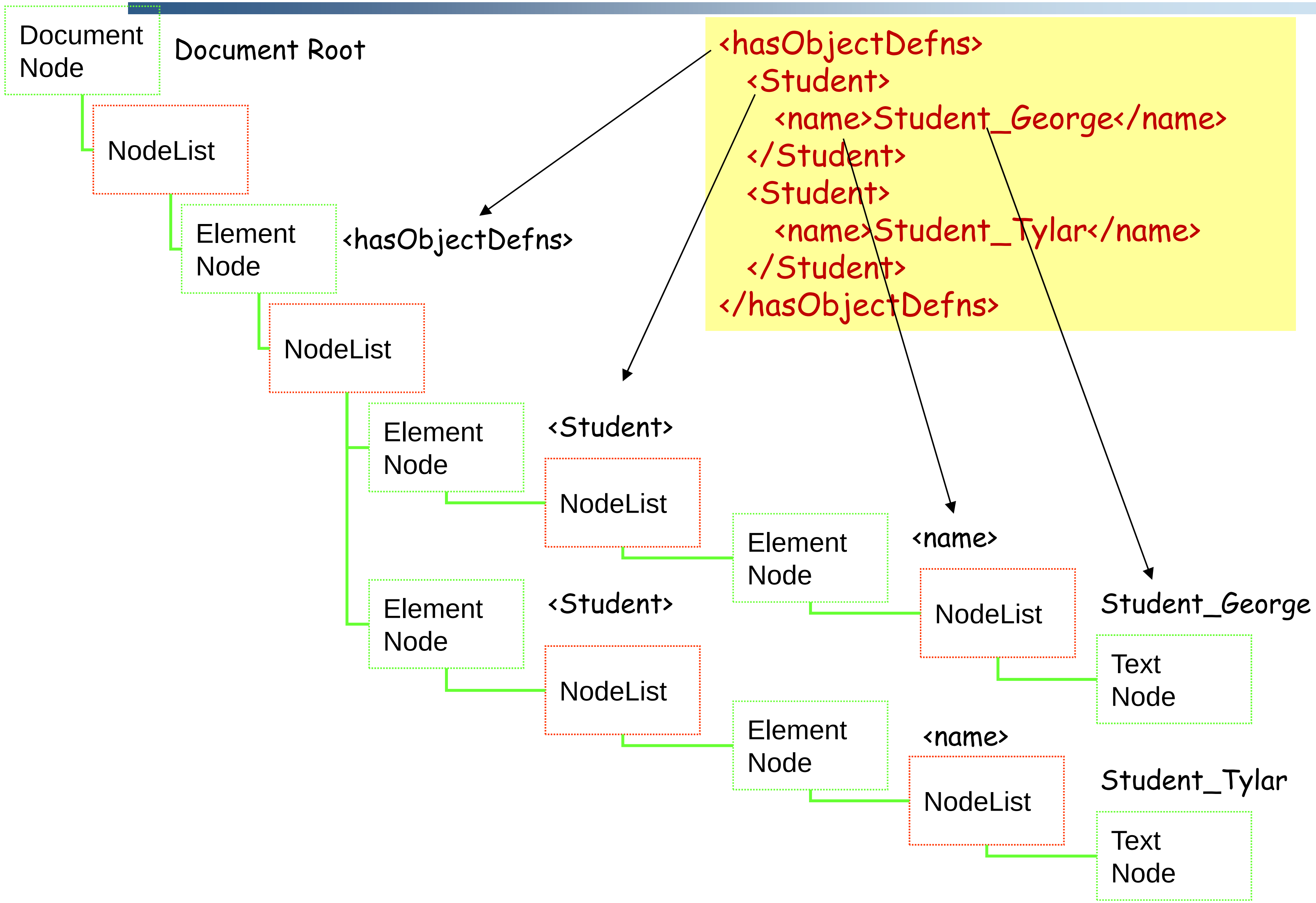
- eXist an open source native XML database, written in Java (platform independent), with an XQuery query processor.

- First version: 2000

- It stores native XML data in:
  - B+-trees and paged files,
  - Document nodes in persistent DOM trees.

- Documents can be divided into collections. The collections can be arranged in a collection hierarchy (similar to a regular file system).

Document Object Model: DOM

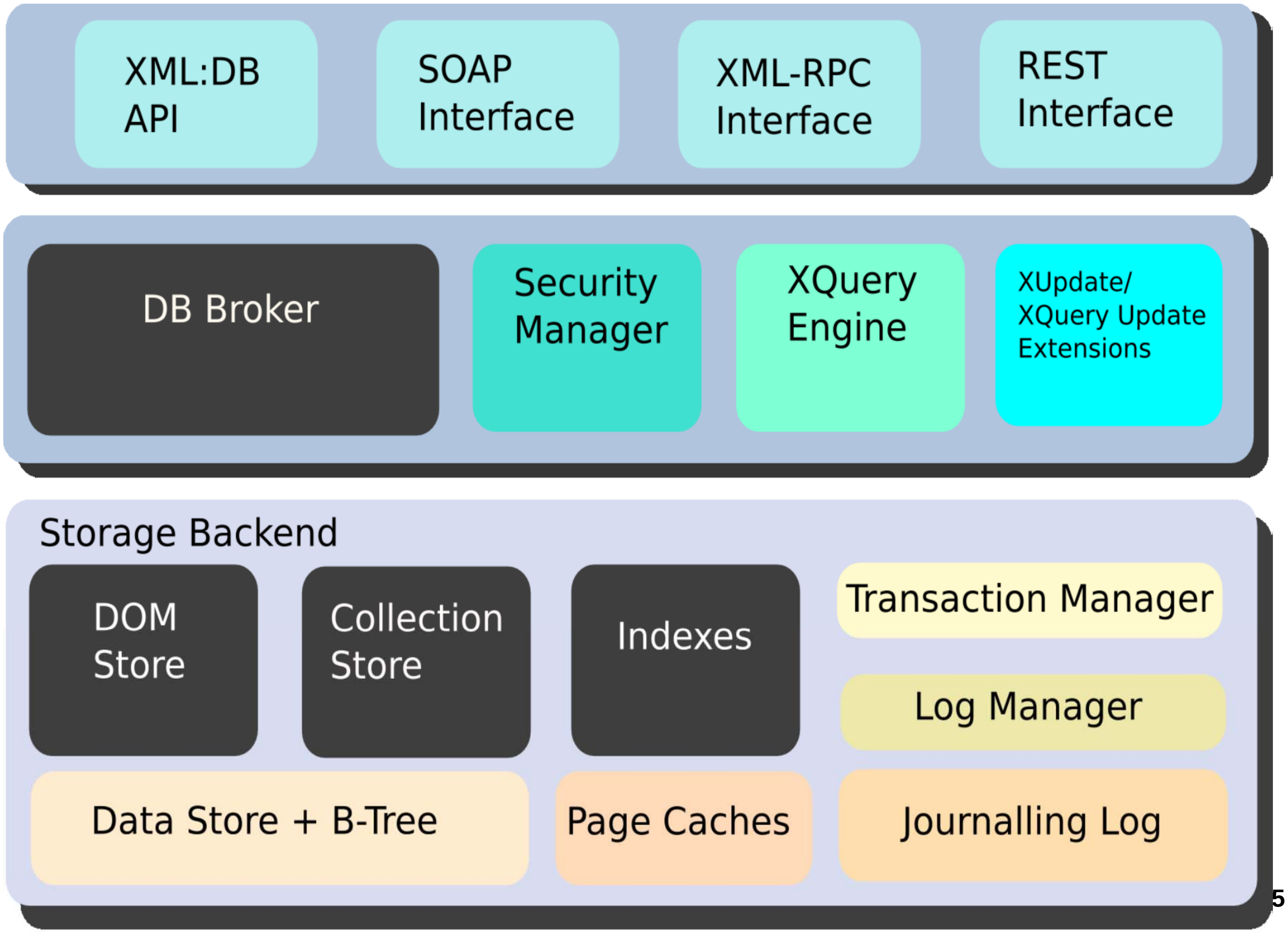


eXist

- Transaction support: No
- Authorization: Unix like, permissions at collection and document level
- XML Standards that are supported:
  - ✓XPath
  - ✓Xquery
  - ✓XUpdate
- Comes with great client GUI interface
- Types of indexes: Structural, Fulltext

Architecture

Source: http://www.xmlprague.cz/2006/slides06/meier.pdf



Index and Data Organization

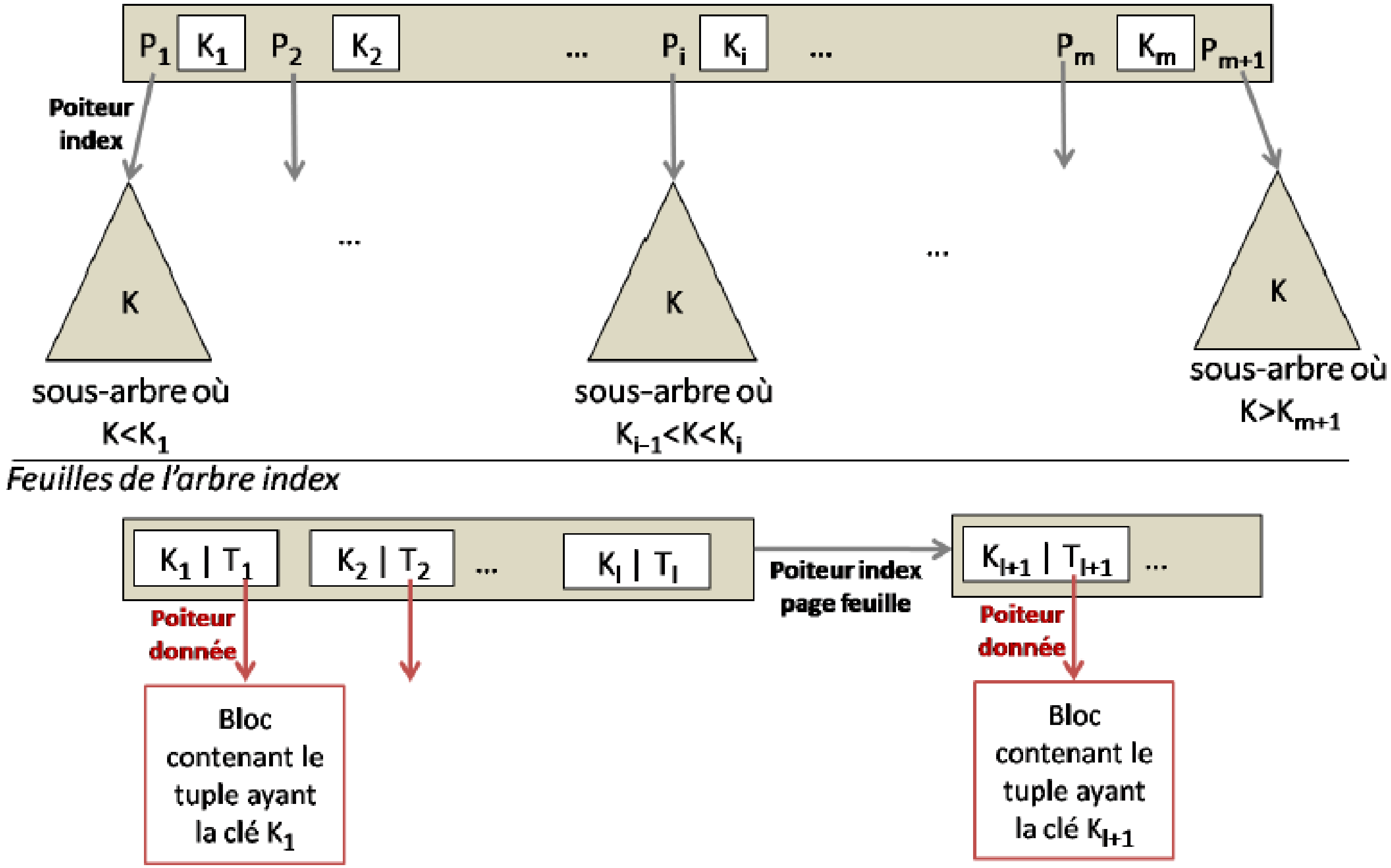
- eXist uses four index files at the XML storage backend:

  - 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 → old version replaced by a new one based on lucene.

All based on B^+-trees

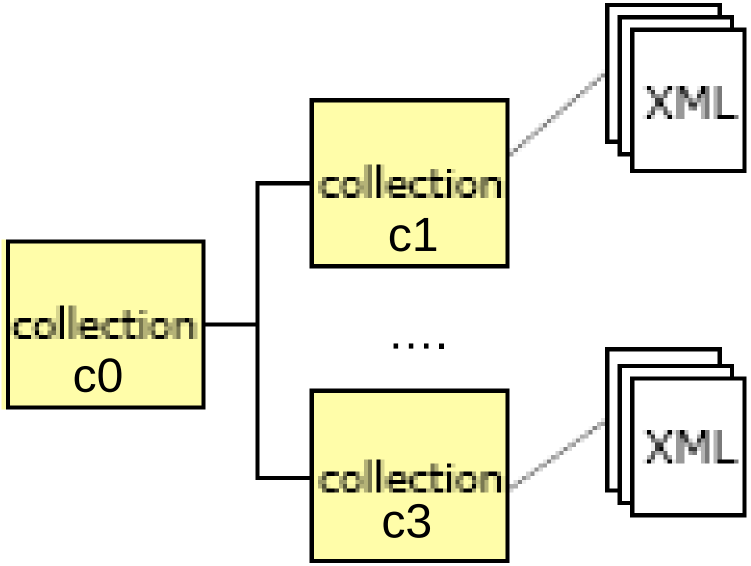
B+ arbre

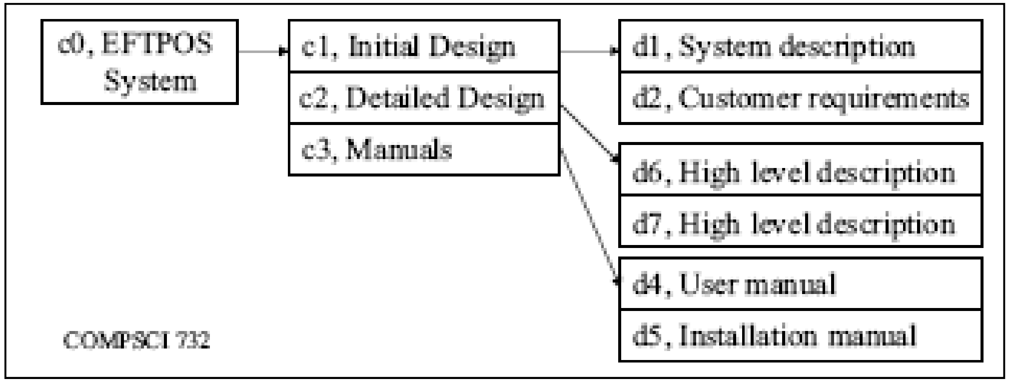
- **Un B+-arbre d'ordre d = un B-arbre où**
  - Les nœuds internes ne contiennent que les clés et les pointeurs d'index
  - Les feuilles de l'arbre contiennent toutes les clés et les pointeurs de données



collection.dbx

- Manages the collection hierarchy and maps collection names to collection objects.





Indexes for elements, attributes and keywords are organized by collection and no by document

dom.dbx

- All document nodes are stored according to the W3C's DOM (Document Object Model).
- In eXist DOM nodes are associated to unique node identifiers.
- Associate the unique node identifiers of top-level elements in a given document to the node’s storage address in the data pages.

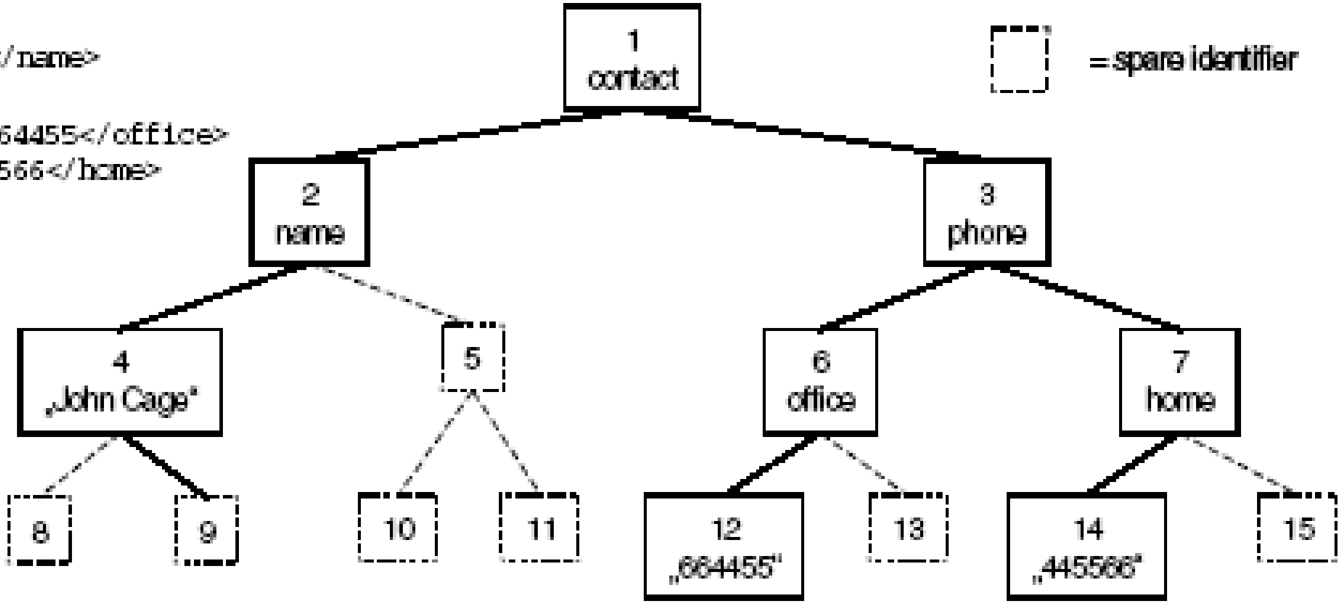
XML Index by Numbering

- A numbering scheme assigns a unique identifier to each node in an XML document
  - Old approach: Level-Order Numbering (or Virtual Node Numbering)
  - New approach: Dynamic Level Numbering (DLN) (2006)
- Certain relationships can be determined immediately

Level-Order Numbering

- XML document is modeled as a complete k-ary tree (each node-root has K children)
- Unique identifier assigned to each node by traversing tree document in level-order

```
<contact>
        <name>John Cage</name>
        <phone>
                <office>664455</office>
                <home>445566</home>
        </phone>
</contact>
```



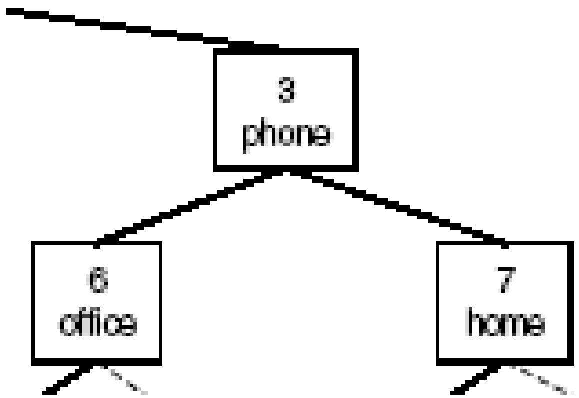
[4]

Level-Order Numbering: benefits

- A node's parent identifier can be computed with

$$parent_i = \left\lfloor \frac{(i-2)}{k} + 1 \right\rfloor$$ [4]

- Exemple: parent$_7$ = [(7-2) / 2 + 1] = [3.5] = 3



[4]

Level-Order Numbering: Disadvantage

- Typical documents have a small number of top level elements (e.g. chapters of a book), whereas the deeper elements contain many more children (e.g. paragraphs in a chapter)

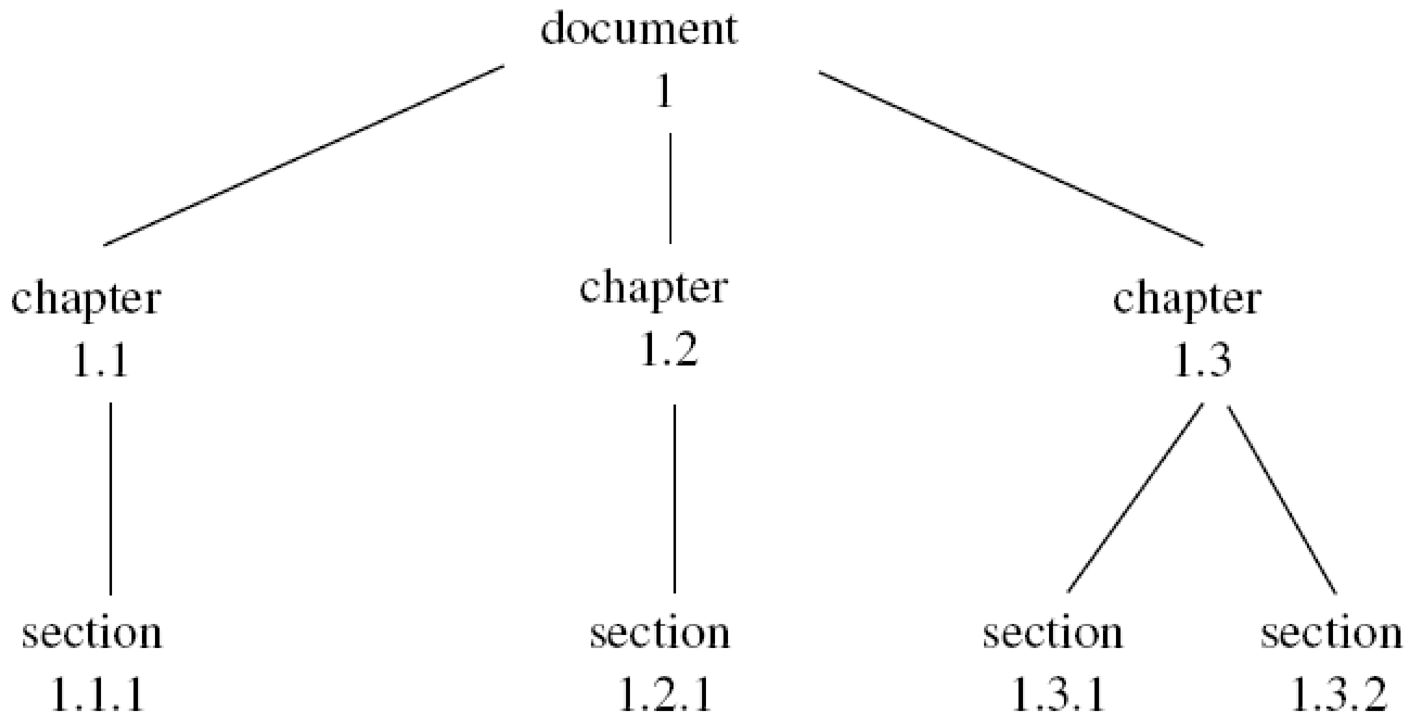
→ Spare identifiers need to be inserted at the top level elements (to be complete)

- Not update friendly: node insertions may trigger a complete renumbering of the tree

Dynamic Level Numbering

- Path-based identification scheme
- Node IDs consist of the ID of the parent node as prefix and a level value: 1, 1.1, 1.2, 1.2.1 . . .
- Between two nodes 1.1 and 1.2, a new node can be inserted as 1.1/1

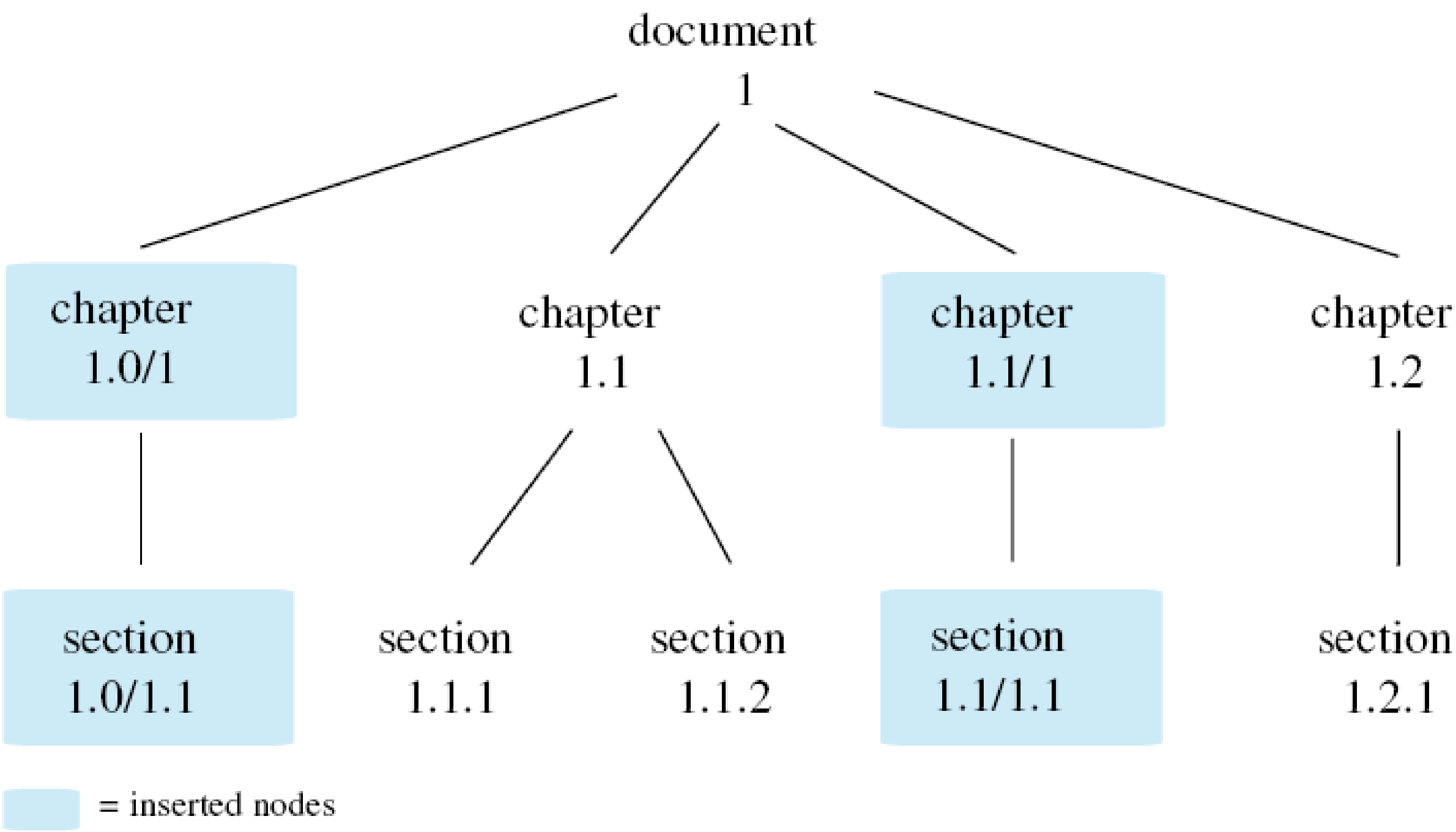
Figure: Node IDs assigned by DLN



Dynamic Level Numbering

- 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

Figure: Node tree after updates



Dynamic Level Numbering: Disadvantage

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

Example: IDs Picked From a Real TEI Document

```
<div6 exist:id="1.46.9.7.8" xml:id="JG10229">
  <head exist:id="1.46.9.7.8.7">
    <name exist:id="1.46.9.7.8.7.3" type="Dramenfigur">FAUST</name>
    <stage exist:id="1.46.9.7.8.7.4">unruhig</stage>
    <stage exist:id="1.46.9.7.8.7.6"> auf seinem Sessel am Pulten</stage>
  </head>
  <sp exist:id="1.46.9.7.8.8" xml:id="JG10230">
    <lg exist:id="1.46.9.7.8.8.4">
      <l exist:id="1.46.9.7.8.8.4.7" n="1" part="N">Hab nun ach die Philosophey</l>
      <l exist:id="1.46.9.7.8.8.4.8" part="N">Medizin und Juristerey,</l>
      <l exist:id="1.46.9.7.8.8.4.9" part="N">Und leider auch die Theologie</l>
      <l exist:id="1.46.9.7.8.8.4.10" part="N">
        Durchaus<note exist:id="1.46.9.7.8.8.4.10.4" place="unspecified" anchored="yes">
        <hi exist:id="1.46.9.7.8.8.4.10.4.4">Durchaus]</hi> Vollständig.</note>
        studirt mit heisser Müh.
      </l>
    </lg>
  </sp>
</div6>
```

Annexe: Binary Encoding of a Level Value

Variable-length encoding using fixed-size units (currently: 4 bits)

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

Hes·so Haute Ecole Spécialisée de Suisse occidentale

Annexe: Binary Encoding of a Level Value

Variable-length encoding using fixed-size units (currently: 4 bits)

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

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 1111000101001100100 1 0 0001	30

Annexe: Binary Encoding of a Level Value

Variable-length encoding using fixed-size units (currently: 4 bits)

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

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 1111000101001100100 1 0 0001	30

Hes·so Haute Ecole Spécialisée de Suisse occidentale

Annexe: Binary Encoding of a Level Value

Variable-length encoding using fixed-size units (currently: 4 bits)

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

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

Annexe: Binary Encoding of a Level Value

Variable-length encoding using fixed-size units (currently: 4 bits)

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

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

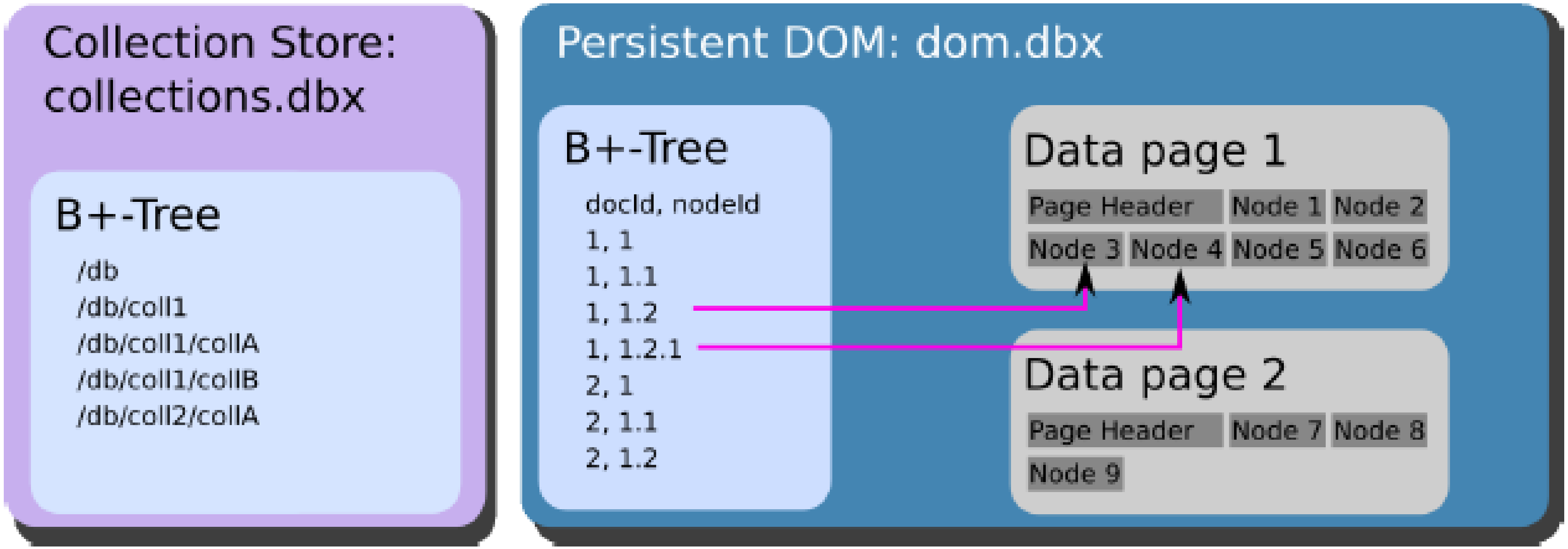
- To store a DLN with subvalue, we use a 1-bit to separate subvalues
- Level values are separated with a 0-bit as before
- Example: 1.1/7 is encoded as 0001 0 0001 1 1000

Dynamic Level Numbering: notes

- 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

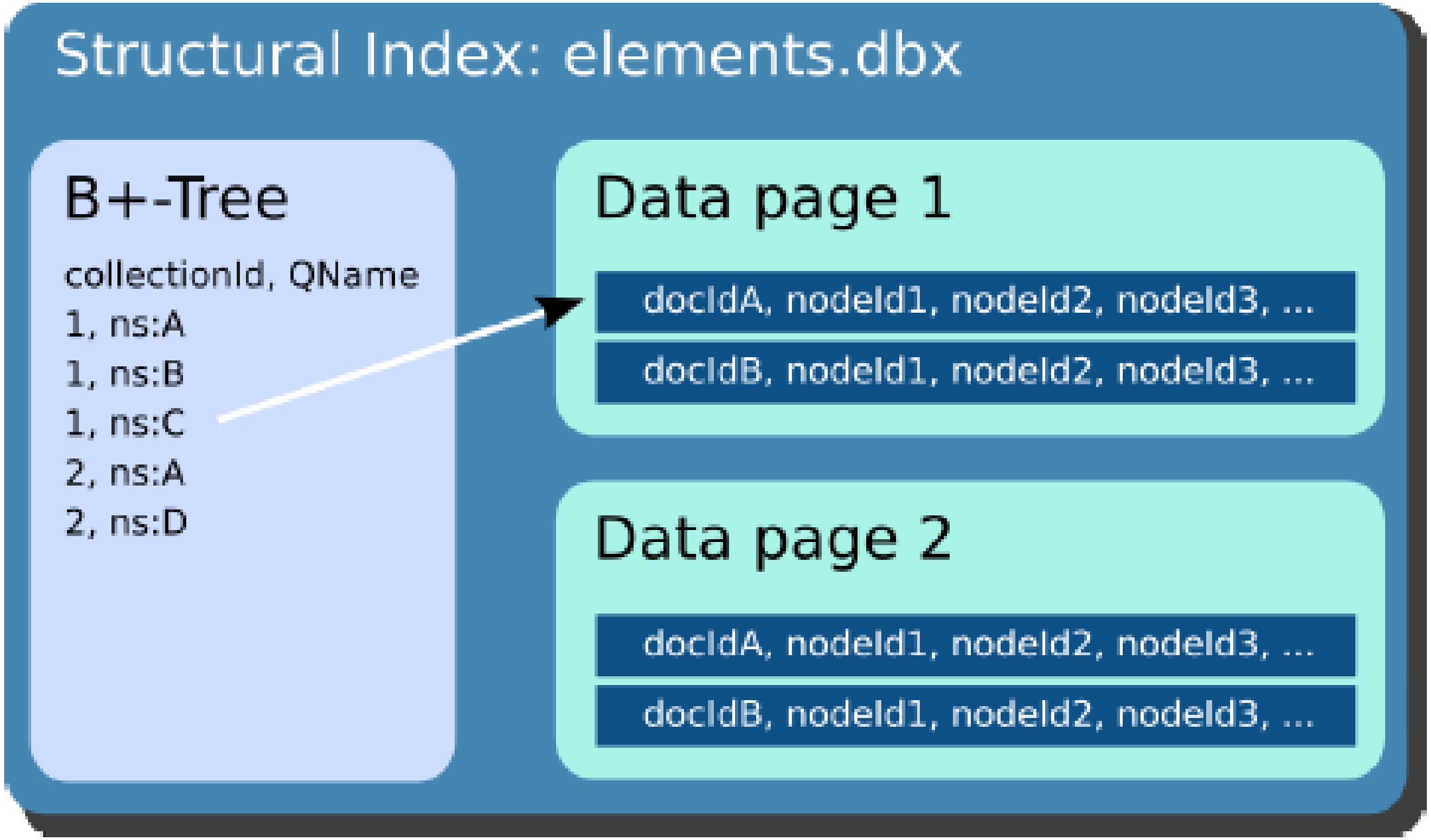
dom.dbx

Every node in eXist is identified by a tuple <docId, nodeId>

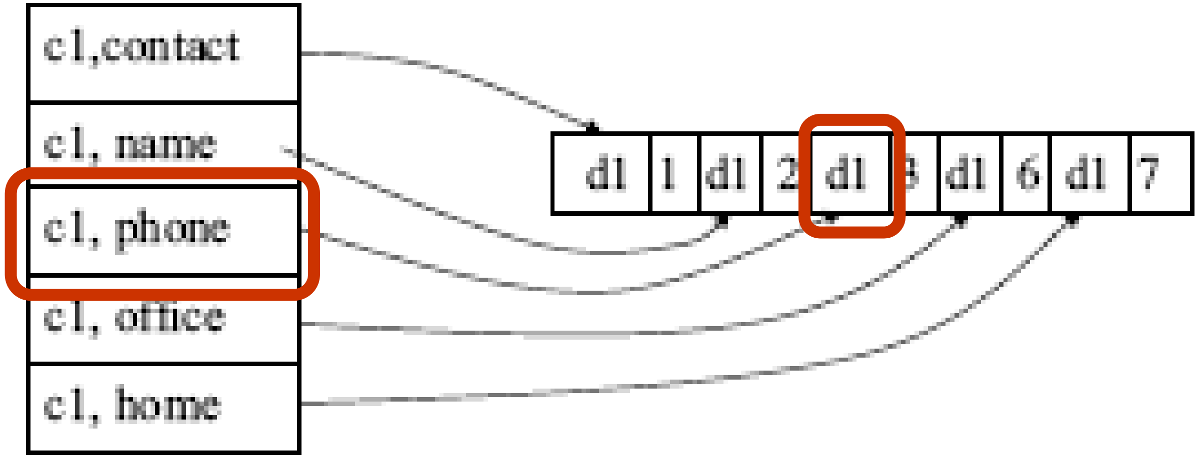


elements.dbx: Structural Index

- Indexed by <collection-id, name-id> pairs
- Value is ordered list of <document-id, node-id> pairs, which are the nodes whose name matches the key entry



elements.dbx: Structural Index



Find all the documents in collection c that have an element "phone".

Query engine

- eXist prefers XPath predicate expressions over an equivalent FLWOR construct using a "where" clause!

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

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

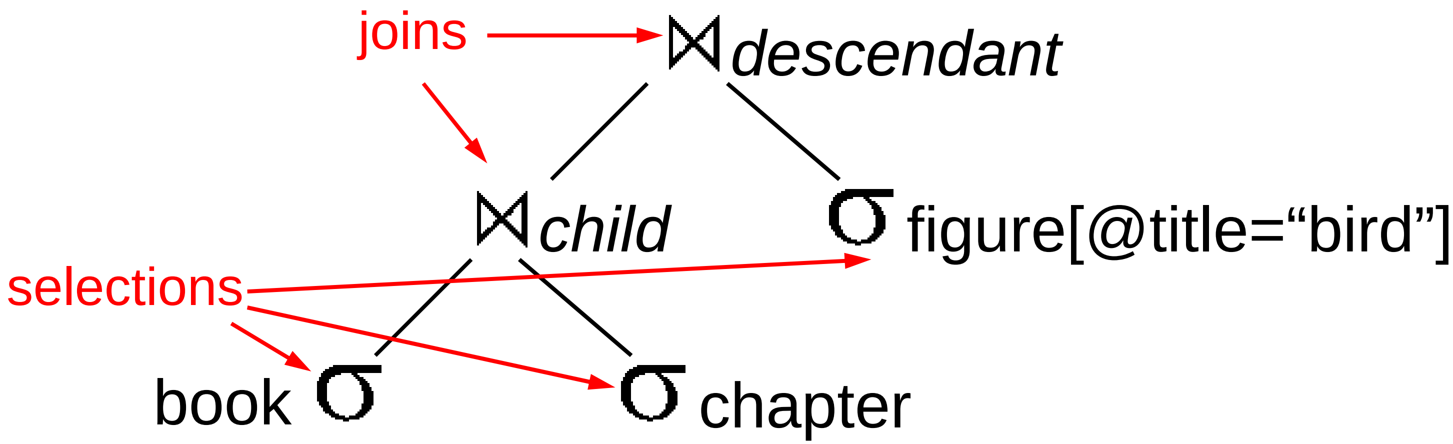
- Based on these provided index types, the eXist XQuery engine relies on path join algorithms for efficient computation of node relationships instead of traditional tree traversals.

Limitations

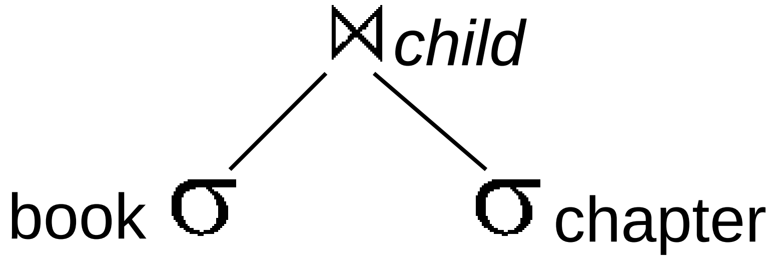
- Currently, most optimizations are implicit
- “Query plan” is hard-coded into the query engine
- Hard to maintain/debug/profile
- eXist needs a better, query-rewriting optimization engine!

Evaluating path expressions

- //book/chapter//figure[@title="bird"]



Join algorithms



//book/chapter

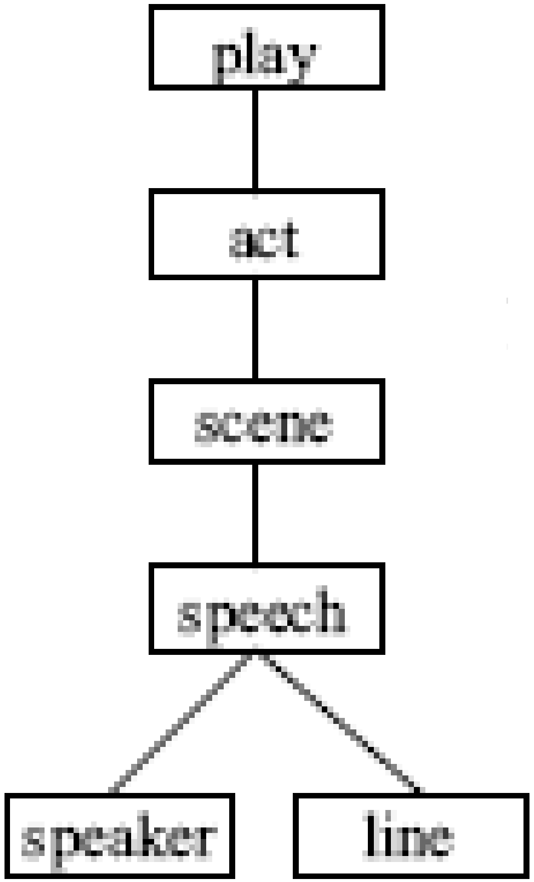
- **Nested loop:**
  1. Find "book" elements by enumerating all elements
  2. For each "book" element, enumerate all children and output the ones named "chapter"

- **"Path join":** efficient join operations utilizing indexes
  1. A={IDs of all elements named "book"}, obtained through the elements.dbx
  2. B={IDs of all elements named "chapter"}
  3. *child axis join:* Result= $\{x \mid x \in B, \mathrm{parent}(x) \in A\}$

Example of how indexes used

/play//speech[speaker = 'Hamlet']

/play//speech → speech[speaker] ← speaker = 'Hamlet'

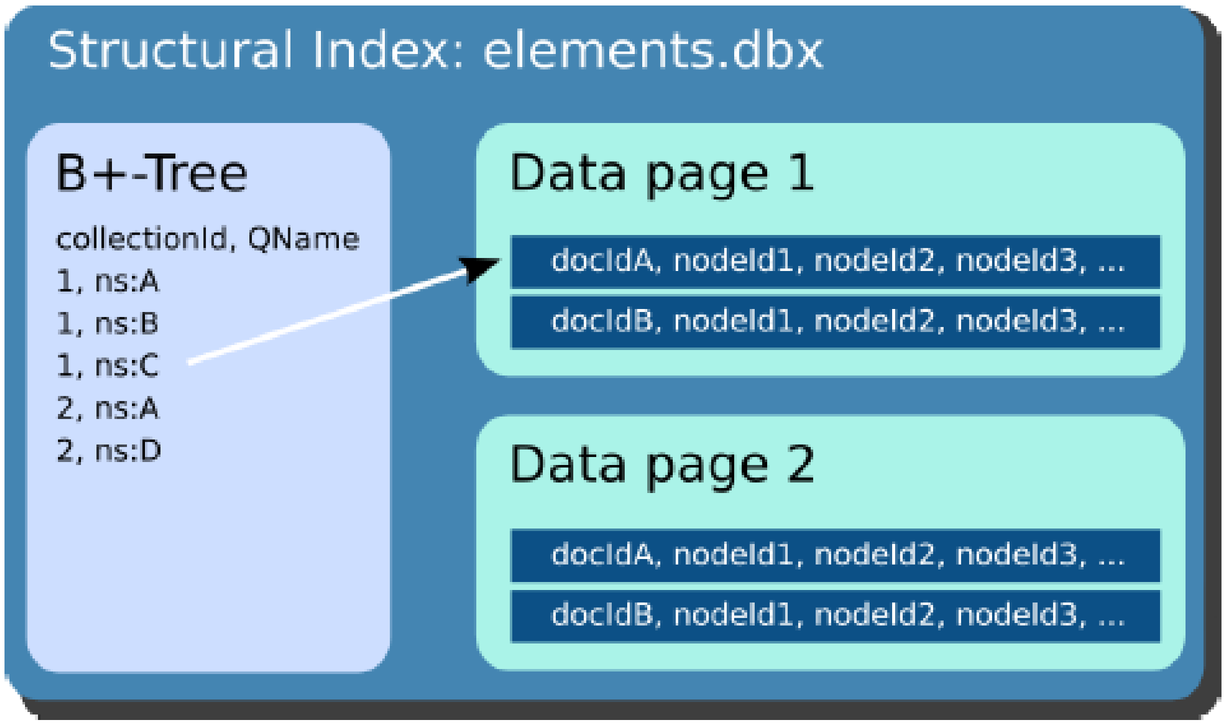


Example of how indexes used: First step

/play//speech[speaker = 'Hamlet']

/play//speech

- find play element for all documents using elements.dbx → set$_1$ <doc_id, node_id>
- find speech elements using elements.dbx→ set$_2$ <doc_id, node_id>



Example of how indexes used: First step

/play//speech[speaker = 'Hamlet']

/play//speech → speech[speaker]

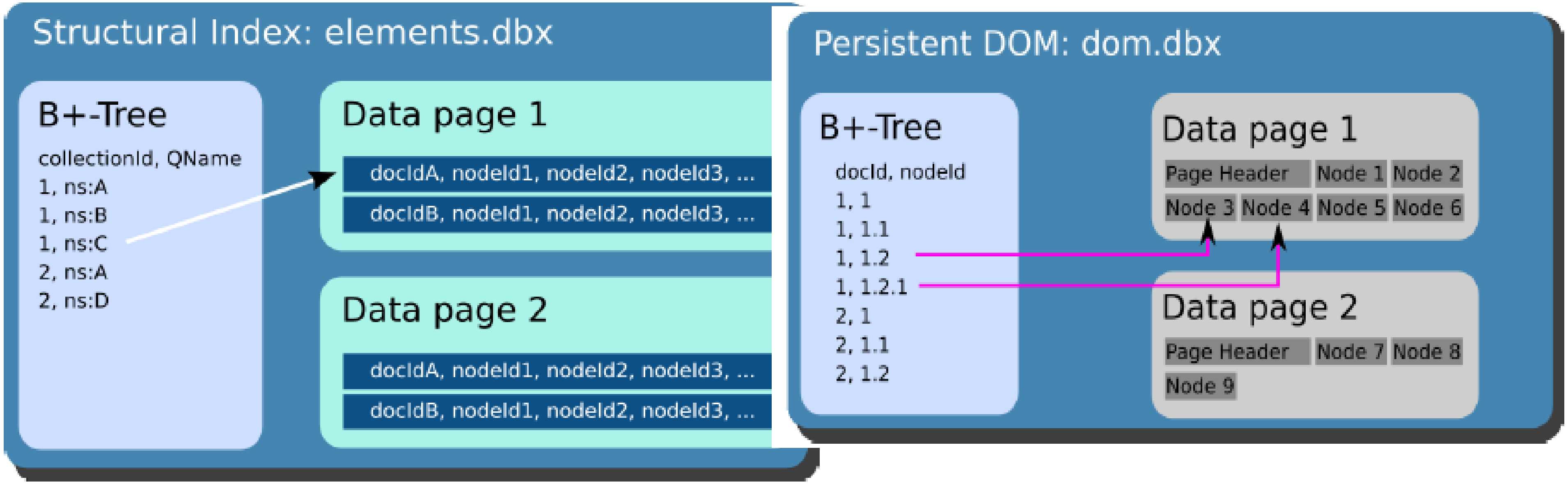
- find play element for all documents using elements.dbx → set$_1$ <doc_id, node_id>
- find speech elements using elements.dbx→ set$_2$ <doc_id, node_id>
- Ancestor-descendant join algorithm is applied to set$_1$ and set$_2$ (<document-id, node-id>): keep the speech which are descendant of play → we also get the ancestor node set for the expression speech[speaker]

```
<div6 exist:id="1.46.9.7.8" xml:id="JG10229">
  <head exist:id="1.46.9.7.8.7">
    <name exist:id="1.46.9.7.8.7.3" type="Dramenfigur">FA
    <stage exist:id="1.46.9.7.8.7.4">unruhig</stage>
    <stage exist:id="1.46.9.7.8.7.6"> auf seinem Sessel a
  </head>
  <sp exist:id="1.46.9.7.8.8" xml:id="JG10230">
```

Annexe Example of how indexes used: Second step

/play//speech[speaker = 'Hamlet']

speaker = 'Hamlet'



- find speaker element for all documents using elements.dbx and use dom.dbx to access the actual DOM nodes in the XML store and keep → set_4 <doc_id, node_id> the speaker nodes having value "Hamlet"

Example of how indexes used: Last step

/play//speech[speaker = 'Hamlet']

/play//speech → speech[speaker] ← speaker = 'Hamlet'

- find play element for all documents → set_1 <doc_id, node_id>
- find speech elements → set_2 <doc_id, node_id>
- *Ancestor-descendant join algorithm* is applied to keep the speech which are descendant of play → set_3 <doc_id, node_id>
- find speaker element having value "Hamlet" → set_4 <doc_id, node_id>
- Finally, the resulting set **set_4** is joined with the context node **set_3** by applying an *ancestor-descendant path join* and output the resulting nodes speech.