XML Technologies

XML Databases: Indexes

Christian Grün

Database & Information Systems Group
Universität Konstanz
Introduction

Why using indexes?

- *speeds up* access to data by orders of magnitudes
- conventional approach: *sequential scan*, $O(n)$
- *index* access: $O(1)$ or $O(\log n)$
- particularly important: *disk access* is minimized!
- last but not least, indexes may even *save space* (*⚠️ remember why?*)

Index structures

- sorted list: binary search
- B-Trees, B*-Trees, R-Trees
- hash sets, maps
- tries, suffix trees, ...

Data structures and indexes may *complement* each other
XML Indexes

So many types:

- name index
- value index
- full-text index
- path index
- structural index
- combined indexes

...use all of them?
XML Indexes

Name index

References all *element/attribute* names of a database

```
<addressbook>
  <address id="1" date="2010-12-12">
    <name>Yoshinari Matsuo</name>
    <code>1088477</code>
    <city>Tokyo</city>
    <country>JP</country>
  </address>
</addressbook>

<addressbook>
  <address id="2" date="2010-11-03">
    <name>Giovanni Ciampa</name>
    <code>80142</code>
    <city>Napoli</city>
    <country>IT</country>
  </address>
</addressbook>
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>addressbook</td>
</tr>
<tr>
<td>2</td>
<td>address</td>
</tr>
<tr>
<td>3</td>
<td>city</td>
</tr>
<tr>
<td>4</td>
<td>code</td>
</tr>
<tr>
<td>5</td>
<td>country</td>
</tr>
<tr>
<td>6</td>
<td>name</td>
</tr>
</tbody>
</table>

Very small; can easily be kept in main memory
XML Indexes

Value index

Contains all distinct text/attribute values of a database

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1088477</td>
</tr>
<tr>
<td>2</td>
<td>80142</td>
</tr>
<tr>
<td>3</td>
<td>Giovanni Ciampa</td>
</tr>
<tr>
<td>4</td>
<td>IT</td>
</tr>
<tr>
<td>5</td>
<td>JP</td>
</tr>
<tr>
<td>6</td>
<td>Napoli</td>
</tr>
<tr>
<td>7</td>
<td>Tokyo</td>
</tr>
</tbody>
</table>

Comparable to index structures of RDBMS; may get very large
XML Indexes

Inverted index

Contains *back references* to the main table (storage) as values

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1088477</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>80142</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Giovanni Ciampa</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>IT</td>
<td>23, 74, 100</td>
</tr>
<tr>
<td>5</td>
<td>JP</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>Napoli</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>Tokyo</td>
<td>10</td>
</tr>
</tbody>
</table>

Pre Entity

-----------------------------
0  addressbook.xml
1  <addressbook>
2  <address>
3  id="1"
4  date="2010-12-12"
5  <name>
6  Yoshinari Matsuo
7  <code>
8  1088477
9  <city>
10  Tokyo
11  <country>
12  JP
XML Indexes

Full-text index

Special type of value index, stores *tokenized* document texts

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1088477</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>80142</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>ciampa</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>giovanni</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>it</td>
<td>23,74,100</td>
</tr>
<tr>
<td>6</td>
<td>jp</td>
<td>12</td>
</tr>
</tbody>
</table>

Smaller set of index keys, more back references
XML Indexes

Full-text index: tokenization

Incoming texts are tokenized, i.e. split into tokens and normalized:

- upper/lower case is ignored
- diacritics are removed
- stop words are ignored
- words are stemmed
- Thesaurus is applied to tokens

XQuery → xquery
España → Espana
One (or) two, (and) we (are) finished.
behavior → behav
modify → change

The input and the queried terms must be tokenized the same way!
XML Indexes

Path summary

Remembers distinct location paths

<table>
<thead>
<tr>
<th>ID</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>addressbook</td>
</tr>
<tr>
<td>2</td>
<td>addressbook/address</td>
</tr>
<tr>
<td>3</td>
<td>addressbook/address/name</td>
</tr>
<tr>
<td>4</td>
<td>addressbook/address/code</td>
</tr>
<tr>
<td>5</td>
<td>addressbook/address/city</td>
</tr>
<tr>
<td>6</td>
<td>addressbook/address/country</td>
</tr>
</tbody>
</table>

Example:

```xml
<addressbook>
  <address id="1" date="2010-12-12">
    <name>Yoshinari Matsuo</name>
    <code>1088477</code>
    <city>Tokyo</city>
    <country>JP</country>
  </address>
  <address id="2" date="2010-11-03">
    <name>Giovanni Ciampa</name>
    <code>80142</code>
    <city>Napoli</city>
    <country>IT</country>
  </address>
</addressbook>
```

- Names can be replaced by their index references
XML Indexes

Path summary
Goldman & Widom [1997]: DataGuides: Enabling Query Formulation and Optimization […]

- “concise and accurate structural summary of semistructured databases”
- dynamic schema; may be enriched with statistics and meta data
XML Indexes

Path summary: using DTDs/XML Schemas

+ index can be built \textit{without accessing} input data
+ the summary does not have to be changed as long as all \textit{update operations} on the data comply with the schema
  - many documents have no schemas attached
  - the input data may be more \textit{expressive} than an existing \textit{schema}
  - real-life schemas are often \textit{bulky} and contain \textit{obsolete/redundant} structures

\textbullet Hybrid approach: build Path index from schemas (if supplied) and enrich it with information of incoming XML documents
XML Indexes

Which index structures can be utilized?

```
//book
/mondial/country/name
//animal[@name = "蟻"]
//article[. contains text "သမၼပာ"]
count(//node())
```

Choose from...

- Name Index
- Path Summary
- Value Index
- Full-Text Index

Which indexes get outdated by the following updates?

```
replace node //animal/@name[. = "蟻"] with "蚂蚁"
rename node /mondial as "factbook"
delete node //wildlife[. contains text "မြန်မြဘာဘာဘ"]
```
XML Indexes

Structural index

• indexes on *XML encodings* (e.g.: pre/post)
• fast *regional queries*
• used for *axis traversals*; e.g.:
  \[ \text{pre}(i) > \text{pre}(n) \land \text{post}(i) < \text{post}(n) \] (\(\triangleright\) which axis is evaluated here?)

\[\begin{array}{cccc}
\text{Pre} & \text{Post} & \text{Entity} \\
0 & 127 & \text{addressbook.xml} \\
1 & 126 & \text{<addressbook>} \\
2 & 10 & \text{<address>} \\
3 & 0 & \text{id="1"} \\
4 & 1 & \text{date="2010-12-12"} \\
5 & 3 & \text{<name>} \\
6 & 2 & \text{Yoshinari Matsuo} \\
7 & 5 & \text{<code>} \\
8 & 4 & \text{1088477} \\
9 & 7 & \text{<city>} \\
10 & 6 & \text{Tokyo} \\
11 & 9 & \text{<country>} \\
12 & 8 & \text{JP}
\end{array}\]

Useful for *set-based* storage architectures, in which tuples are unordered (\(\rightarrow\) relational DBMS)
XML Indexes

B-Tree on $\text{pre/post}$

Two-dimensional R-Tree
Value Index: Implementation

Requirements

- creation should be *as fast as possible*
- today’s main memory is huge: let’s *benefit* from it!
- still: main memory should *not impose limit* for large XML instances
- out of scope here: *incremental* index updates

Solution

- balanced main-memory tree: prevents degeneration of index structure
- *partial indexes* are written to disk if memory is full
- finally, all indexes are *merged* (divide and conquer)
Value Index: Implementation

Introduction

- **texts** are organized as *sorted list*
- *inverted index*: constant node ids are stored as back references
Value Index: Implementation

Optimizations

In order to save space, various steps are applied on the index structure:

1. texts are not stored in the index at all, as they can directly be looked up in the main table
2. all integers are compressed bitwise
3. ids are sorted, and distances between values are stored
4. all keys exceeding a maximum length are ignored

Very compact representation, speeding up all I/O operations
Value Index: Implementation

Step 1: Store texts and ids

<table>
<thead>
<tr>
<th>Text (&quot;John&quot;)</th>
<th>ID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>4A 6F 68 6E 00</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x0020</td>
<td>John</td>
<td></td>
</tr>
<tr>
<td>0x0300</td>
<td>John</td>
<td></td>
</tr>
<tr>
<td>0x4000</td>
<td>John</td>
<td></td>
</tr>
<tr>
<td>0x4004</td>
<td>John</td>
<td></td>
</tr>
</tbody>
</table>

key (+ 00) + #ids + ids: 25 bytes

Discard text; use first id to look up text in table:

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 04</td>
<td></td>
</tr>
<tr>
<td>00 00 00 20</td>
<td></td>
</tr>
<tr>
<td>00 00 03 00</td>
<td></td>
</tr>
<tr>
<td>00 00 40 00</td>
<td></td>
</tr>
<tr>
<td>00 00 40 04</td>
<td></td>
</tr>
</tbody>
</table>

#ids + ids: 20 bytes
Value Index: Implementation

Step 2: Bit compression

<table>
<thead>
<tr>
<th>Encoding (binary)</th>
<th>Range (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00bbbbbb</td>
<td>00 – 3F</td>
<td>1 byte maps $2^6$ values</td>
</tr>
<tr>
<td>01bbbbbb B</td>
<td>40 – 3FFF</td>
<td>2 bytes map $2^{14}$ values</td>
</tr>
<tr>
<td>10bbbbbb B B B</td>
<td>4000 – 3FFFFFFFF</td>
<td>4 bytes map $2^{30}$ values</td>
</tr>
<tr>
<td>11000000 B B B B</td>
<td>40000000 – FFFFFFFF</td>
<td>5 bytes map $2^{32}$ values</td>
</tr>
</tbody>
</table>

b represents a bit, B represents a byte

- values < 64 will only occupy 1 byte; large integers will take 5 bytes
- similar to UTF8 encoding (ASCII: 1 byte, German umlauts: 2 bytes)
Value Index: Implementation

Step 2: Bit compression

Apply bit compression to all integer values:

<table>
<thead>
<tr>
<th>IDs</th>
<th>Compressed IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00 00 04</td>
<td>04 (00...)</td>
</tr>
<tr>
<td>00 00 00 20</td>
<td>20 (00...)</td>
</tr>
<tr>
<td>00 00 03 00</td>
<td>43 00 (01...)</td>
</tr>
<tr>
<td>00 00 40 00</td>
<td>80 00 40 00 (10...)</td>
</tr>
<tr>
<td>00 00 40 04</td>
<td>80 00 40 04 (10...)</td>
</tr>
</tbody>
</table>

#ids + ids: 20 bytes  #ids + compressed ids: 12 bytes
Value Index: Implementation

Step 3: Compressed distances

Sort values, store distances:
\[ d_n = \text{id}_n - \text{id}_{n-1}, \quad \text{id}_1 = 0 \]

Compressed IDs

<table>
<thead>
<tr>
<th>ID</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x0020</td>
<td>John</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x0300</td>
<td>John</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x4000</td>
<td>John</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x4004</td>
<td>John</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Compressed distances

<table>
<thead>
<tr>
<th>ID</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>04</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>( 20 - 0)</td>
</tr>
<tr>
<td>44 00</td>
<td>( 300 - 20)</td>
</tr>
<tr>
<td>80 00 40 00</td>
<td>(4000 - 300)</td>
</tr>
<tr>
<td>80 00 40 04</td>
<td>(4004 - 4000)</td>
</tr>
</tbody>
</table>

\#ids + compressed ids: 12 bytes
\#ids + compr. distances: 7 bytes
XML Indexes: Practical Results

Performance with and without value index

txt,atv: texts, attributes (48.7%)
tbl,tbli,inf: table (46.3%)
txtl,txtr: text index (3.3%)
atvl,atvr: attribute index (1.7%)
XML Indexes

Summary

- index structures can speed up queries by orders of magnitude
- indexes can be more important than database itself (example: web search)
- USP of XML databases: search and navigation are combined

Out of scope...

- index structures get out-of-dated when updates are performed
- incremental index updates: after successful database update, refresh data structures in index
- extreme case: updates may get slower than rebuilding full index!
All there is to say: We are done!