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

ID	Name
1	addressbook
2	address
3	city
4	code
5	country
6	name

```
<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>
```

☞ Very small; can easily be kept in main memory

XML Indexes

Value index

Contains all distinct *text/attribute values* of a database

ID	Text
1	1088477
2	80142
3	Giovanni Ciampa
4	IT
5	JP
6	Napoli
7	Tokyo

```
<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>
```

☞ Comparable to index structures of RDBMS; may get very large

XML Indexes

Inverted index

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

ID	Text	Values
1	1088477	8
2	80142	19
3	Giovanni Ciampa	17
4	IT	23,74,100
5	JP	12
6	Napoli	21
7	Tokyo	10

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

ID	Text	Values
1	1088477	8
2	80142	19
3	ciampa	17
4	giovanni	17
5	it	23,74,100
6	jp	12

```
<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>
```

☞ 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 — XQuery ⇢ xquery
- diacritics are removed — España ⇢ Espana
- stop words are ignored — One (or) two, (and) we (are) finished.
- words are stemmed — behavior ⇢ behav
- Thesaurus is applied to tokens — modify ⇢ change

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

XML Indexes

Path summary

Remembers *distinct* location paths

ID	Path	
1	1	addressbook
2	1/2	addressbook/address
3	1/2/3	addressbook/address/name
4	1/2/4	addressbook/address/code
5	1/2/5	addressbook/address/city
6	1/2/6	addressbook/address/country

```
<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>
```

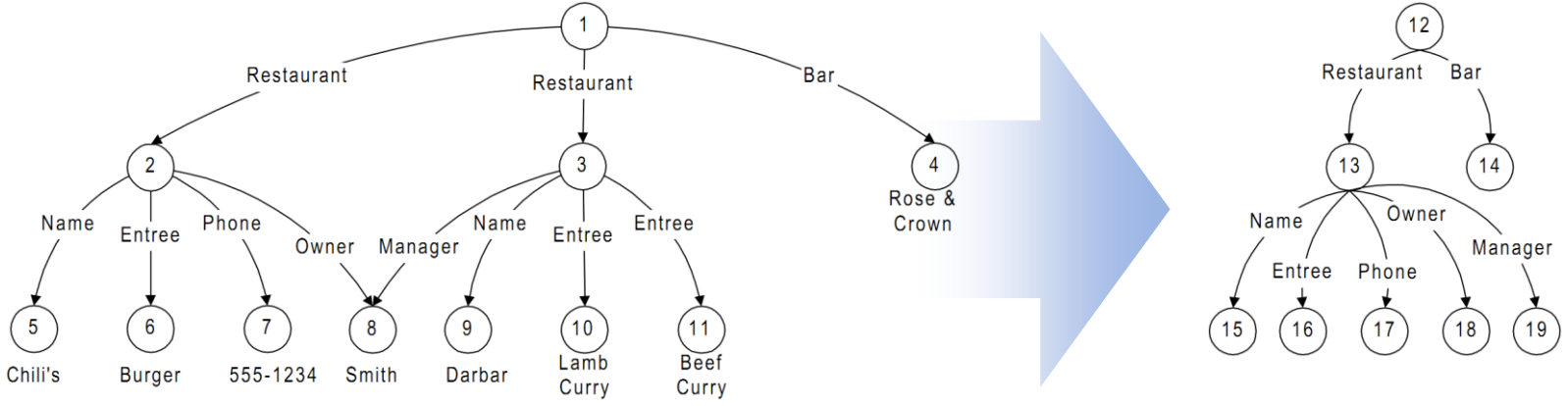
☞ Names can be replaced by their index references

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

☞ 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 "எறும்பு"]
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 "မြန်မာဘာသာ"]
```

XML Indexes

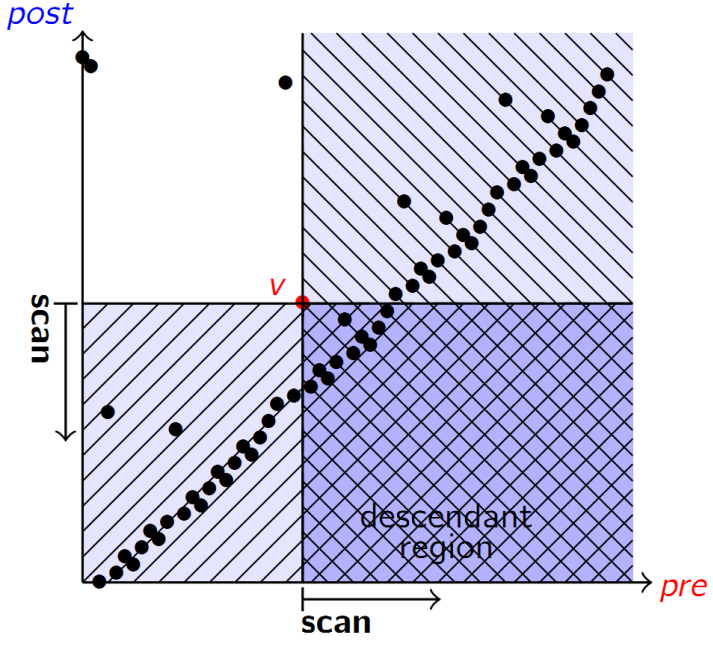
Structural index

- indexes on *XML encodings* (e.g.: pre/post)
- fast *regional queries*
- used for *axis traversals*; e.g.:
  pre(i) > pre(n) ∧ post(i) < post(n)
  (✎ which axis is evaluated here?)

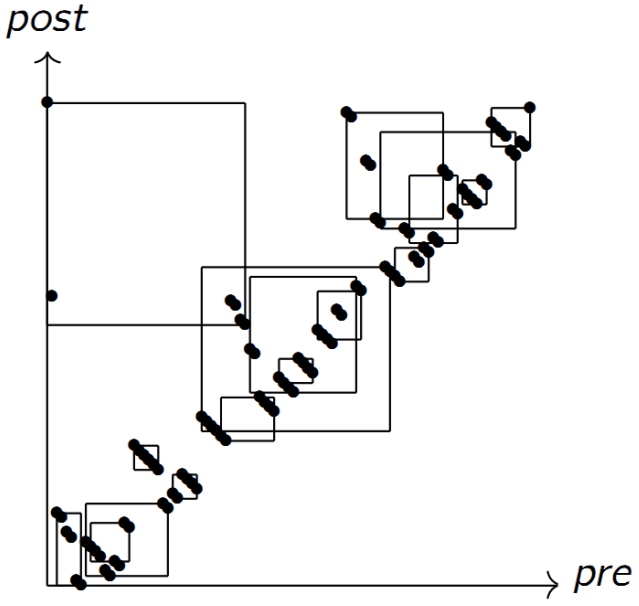
Pre	Post	Entity
0	127	addressbook.xml
1	126	<addressbook>
2	10	<address>
3	0	id="1"
4	1	date="2010-12-12"
5	3	<name>
6	2	Yoshinari Matsuo
7	5	<code>
8	4	1088477
9	7	<city>
10	6	Tokyo
11	9	<country>
12	8	JP

☞ Useful for *set-based* storage architectures, in which tuples are unordered (→ relational DBMS)

XML Indexes



B-Tree on 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

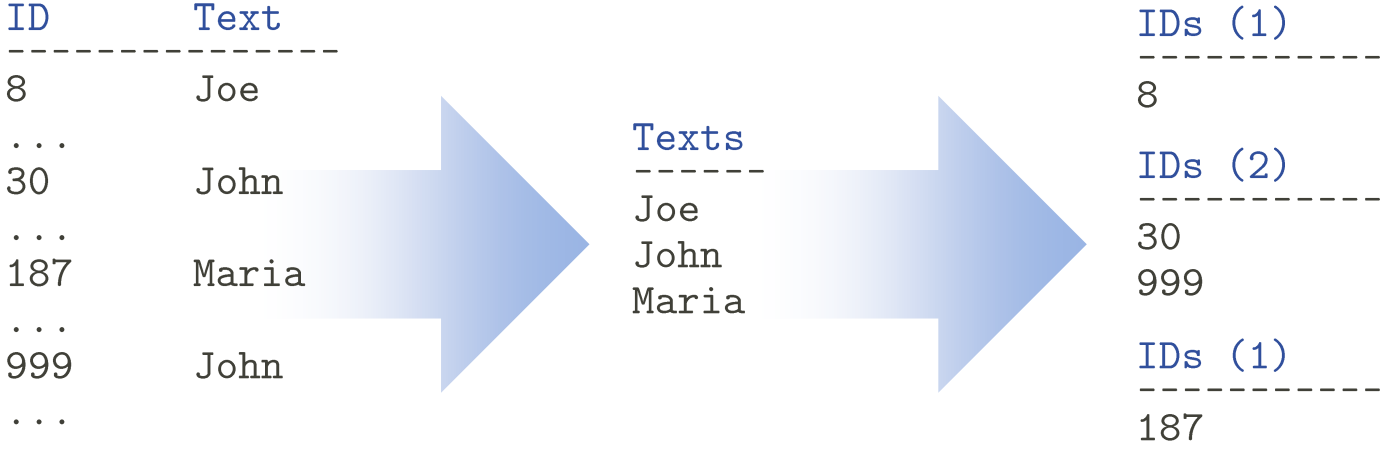
Solution

- 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

```
Text ("John")
-------------
4A 6F 68 6E 00

IDs
-----------
00 00 00 04
00 00 00 20
00 00 03 00
00 00 40 00
00 00 40 04
```

```
ID        Text
--------------
...
0x0020    John
...
0x0300    John
...
0x4000    John
...
0x4004    John
...
```

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

```
IDs
-----------
00 00 00 04
00 00 00 20
00 00 03 00
00 00 40 00
00 00 40 04
```

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

#ids + ids: 20 bytes

Value Index: Implementation

Step 2: Bit compression

Encoding (binary)	Range (hex)	Description
00bbbbbb	00 – 3F	1 byte maps 2^6 values
01bbbbbb B	40 – 3FFF	2 bytes map 2^{14} values
10bbbbbb B B B	4000 – 3FFFFFFF	4 bytes map 2^{30} values
11000000 B B B B	40000000 – FFFFFFFF	5 bytes map 2^{32} values

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

```
IDs
-----------
00 00 00 04
00 00 00 20
00 00 03 00
00 00 40 00
00 00 40 04
```

```
ID          Text
--------------
...
0x0020      John
...
0x0300      John
...
0x4000      John
...
0x4004      John
...
```

☞ Apply bit compression to all integer values:

```
Compressed IDs
--------------------
04              (00...)
20              (00...)
43 00           (01...)
80 00 40 00     (10...)
80 00 40 04     (10...)
```

#ids + ids: 20 bytes

#ids + compressed ids: 12 bytes

Value Index: Implementation

Step 3: Compressed distances

☞ Sort values, store distances:

$d_n = id_n - id_{n-1}$, $id_{-1} = 0$

```
ID          Text
--------------
...
0x0020      John
...
0x0300      John
...
0x4000      John
...
0x4004      John
...
```

```
Compressed IDs
---------------
04
20
44 00
80 00 40 00
80 00 40 04
```

```
Compressed distances
--------------------
04
20          (  20 -    0)
42 E0       ( 300 -   20)
7D 00       (4000 -  300)
04          (4004 - 4000)
```

#ids + compressed ids: 12 bytes

#ids + compr. distances: 7 bytes

XML Indexes: Practical Results

C:/Users/user/Desktop/query.xq - BaseX 8.4 beta

Database Editor View Visualization Options Help

Editor

query.xq

```
1 db:open('1gb')//item[@id = 'item12345']
```

OK 1 : 40

Total Time: 2.11 ms

Query Info

Compiling:
- pre-evaluating db:open("1gb")
- atomic evaluation of (@*:id = "item12345")
- rewriting descendant-or-self step(s)
- applying attribute index for "item12345"

Optimized Query:
db:attribute("1gb", "item12345")/self::*:id/parent::*:item

Result

```
<item id="item12345">
  <location>United States</location>
  <quantity>1</quantity>
  <name>many watery corses</name>
  <payment>Cash</payment>
  <description>
    <text>humours respects kindness resolv
```

Performance with and without value index

atv.basex	45,7 M
atvl.basex	14,9 M
atvr.basex	3,3 M
inf.basex	18,5 k
tbl.basex	492,8 M
tbli.basex	9 b
txt.basex	472,9 M
txtl.basex	31,0 M
txtr.basex	4,1 M

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

C:/Users/user/Desktop/query.xq - BaseX 8.4 beta

Database Editor View Visualization Options Help

Editor

query.xq

```
1 db:open('1gb')//item[@id = 'item12345']
```

OK 1 : 40

Total Time: 251.46 ms

Query Info

Compiling:
- pre-evaluating db:open("1gb")
- atomic evaluation of (@*:id = "item12345")
- rewriting descendant-or-self step(s)
- converting descendant::*:item[(@*:id = "item12345")] to child steps

Optimized Query:
db:open-pre("1gb",0)/*:site/*:regions/*/*:item[(@*:id = "item12345")]

Result

```
<item id="item12345">
  <location>United States</location>
  <quantity>1</quantity>
  <name>many watery corses</name>
  <payment>Cash</payment>
  <description>
```

XML Indexes

Summary

- index structures can speed up queries by *orders of magnitudes*
- 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!