Neue Datenbank-Technologien

Stefan Böttcher

Universität Paderborn

+ more than 5 years experience in  text compression
+ more than 7 years experience in  tree compression
+ more than 10 years experience in XML processing

+ worked with more than 30 companies
New Challenges for Data Management

Big Data

Streaming data

Mobile data

Versions

Main memory database management

New data formats (text, documents, multimedia, graphs, ...)

Neue Datenbank-Technologien - Prof. Dr. Stefan Böttcher (Universität Paderborn)
Big Data / Streaming Data Examples

Data from sensors, RFID readers, cameras, microphones, ...

Financial transactions

Scientific data, satellite data, weather data

Telecommunication data

Webserver-Logs

Smart metering data
Big Data Challenges

Storage & transport (compression, (lossy) aggregation, ...)

Search (Keyword search, index creation, ...)

Quality (correctness, data cleaning, modification, ...)

Presentation (transformation, aggregation, visualisation)
Focus of today‘s talk

Data formats: text & tree structured data

Data compression, i.e., text compression & tree compression

Search (Keyword search, query processing,...)

Modification

Data Transformation

Research perspective
Why text compression? Why block-sorting?

Energy & data transfer costs:
→ data compression

Search in Big Data:
search engines need fast location of keyword in documents

→ tries, suffix arrays, Burrows Wheeler Transformation (BWT) (e.g. in bzip2)
Text compression

Large texts shall be compressed – many different ideas:

→ learning and reusing patterns → e.g., LZ family
+ using grammars → e.g. sequitur
→ using entropy coding → e.g. Huffman encoding
→ encoding character repetitions → e.g. Run Length Encoding (RLE)

Some techniques, e.g. RLE, are only good, if there are many repetitions:
→ start with preparation step finding a permutation that has more repetitions
  e.g. BWT

→ idea: Burrows Wheeler Transformation (BWT) is based on locality
  e.g. (in English text) ‘...ugh...’ it is very likely that _ is ‘o’
  when sorting all letter of a text (e.g. ‘o’) according to their suffix (e.g. ‘ugh...’),
  the ‘o’ of many sequences ‘...ough...’ are sorted together
  → Run Length Encoding gives better results
BWT – how to construct it from the Input?

Input = $ a b c $ $ c b a $ $ b a c $

compute all rotations:
$ a b c $ $ c b a $ $ b a c 
$ b c $ $ c b a $ $ b a c $ $ a 
$ c $ $ c b a $ $ b a c $ $ a b 
$ b c $ $ c b a $ $ b a c $ $ a b c 
$ a b c $ $ c b a $ $ b a c $ $ a b c 
$ b a $ $ b a c $ $ a b c $ $ c 
$ a b c $ $ a b c $ $ c b $ $ c b a 
$ b a c $ $ a b c $ $ c b a $ $ b 
$ a c $ $ a b c $ $ c b a $ $ b 
$ c $ $ a b c $ $ c b a $ $ b a 

F (first) = $ a b c $ $ c b a $ $ b a c 

sort rotations alphabetically:
$ a b c $ $ c b a $ $ b a c 
$ b a c $ $ a b c $ $ c b a 
$ c b a $ $ b a c $ $ a b c 
$ a b c $ $ c b a $ $ b a c 
$ a c $ $ a b c $ $ c b a $ $ b 
$ c $ $ a b c $ $ c b a $ $ b a 

L (last) = BWT = $ c $ $ a b c $ $ a b c $ $ c b a 

E (end of Input) = store only this
BWT-Decompression using Rank on L & Select on F

F (first) \[ \text{alphabetically sorted rotations} \]

L (last) = BWT

The arrows illustrate the L \( \rightarrow \) F function

Start decompression at E (end)
$ = \text{word delimiters}

6 c characters before a word delimiter ($$) occur next to each other

7th c character not followed by $

6 c characters before a word delimiter ($$) = \text{last characters of the words in Input}

2nd last characters of words ending with “c$“

The arrows illustrate the $L \rightarrow F$ function
BWT – backwards interval search for „bec$“

$ = word delimiters

6 c characters before a word delimiter ($) = last characters of the words in Input

6 c characters before a word delimiter ($) occur next to each other

3 e characters before “c$“

e-interval of 2nd last characters of words ending with “c$“

b-interval of 3rd last characters of words ending with “ec$“
BWT pros and cons

BWT

+ good compression (when used with MTF&Huffman in bzip2)
+ excellent for keyword search (when used like suffix arrays)
- creation (sorting) takes too long (limit approx. 50 TB)
- modification impossible

→ improvement: IRT
IRT - a Queryable and Updateable Text Compression

IRT – small modification of BWT

+ texts compressed with IRT + RLE + Wavelet Tree are queryable and updateable (fast insert and delete on compressed texts) without decompression

➔ compressed trees (e.g. XML documents, JSON trees) including text and attribute values are queryable and updateable (fast insert and delete of compressed subtrees) without decompression
Difference IRT \leftrightarrow BWT: deletion and BWT do not commute

Even if the position of the first/last letter of the word to be deleted is known (e.g. by an index), deletion of a word and transformation by BWT do not commute.

Let S be a text and W an arbitrary word of S, then

\[
\text{S} \xrightarrow{\text{deletion of W from S}} \text{S-W} \xrightarrow{\text{transformation}} \text{BWT(S-W)}
\]

\[
\text{BWT(S)} \xrightarrow{\text{deletion of W from BWT(S)}} \text{BWT(S-W)} \neq \text{BWT(S-W)}
\]

i.e. in contrast to IRT, a word cannot be deleted from BWT(S) without retransformation of BWT(S) to S.
Difference IRT ↔ BWT: but deletion and IRT commute

However, deletion of a word and transformation by IRT commute.

Let S be a text and W an arbitrary word of S, then

\[ \text{IRT}(S) - \text{IRT}(W) = \text{IRT}(S - W) \]

i.e. in contrast to BWT(S), a word in IRT(S) can be deleted from IRT(S) without retransformation of IRT(S) to S.
Difference IRT \leftrightarrow BWT: insertion and BWT do not commute

Even if the position of the first/last letter of the word to be inserted is known (e.g. by an index), insertion of a word and transformation by BWT do not commute.

Let S be a text and W an arbitrary word of S, then

\[ S \xrightarrow{\text{insertion of W as Nth word of S into S}} BWT(S) \]

\[ S+W \xrightarrow{\text{insertion of W as Nth word of S into BWT(S)}} BWT(S) + BWT(W) \]

i.e. in contrast to IRT, a word cannot be inserted into BWT(S) without retransformation of BWT(S) to S.

\[ BWT(S) + BWT(W) \neq BWT(S+W) \]
Difference IRT $\leftrightarrow$ BWT: but insertion and IRT commute

Insertion of a word and transformation by BWT commute

Let $S$ be a text and $W$ an arbitrary word of $S$, then

$$\text{IRT}(S) + \text{IRT}(W) = \text{IRT}(S+W)$$

i.e. in contrast to BWT, a word can be inserted into IRT($S$) without retransformation of IRT($S$) to $S$. 
String Compression Results (2): update time CIRT vs. bzip2

Insert / Delete boundary comparing CIRT and bzip2

Text: The bible

Here, bzip2 is faster

Here, CIRT is faster
String Compression Results (1) : Query time on CIRT

Query time on compressed IRT often faster than decompressing bzip2-compressed text

Text: The bible

<table>
<thead>
<tr>
<th>Query</th>
<th>Time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 'Moses'</td>
<td>360</td>
</tr>
<tr>
<td>== 'Moses'</td>
<td>380</td>
</tr>
<tr>
<td>&lt;= 'Moses' and != 'God'</td>
<td>500</td>
</tr>
<tr>
<td>substring 've'</td>
<td>250</td>
</tr>
<tr>
<td>Decompression bzip2</td>
<td>400</td>
</tr>
</tbody>
</table>
Summary IRT, a Queryable and Updateable Text Compression

Technique: IRT – a transformation similar to BWT, but

+ texts compressed with IRT + RLE + Wavelet Tree are queryable and updateable (fast insert and delete on compressed texts) without decompression

+ inserts of up to 18% are faster in CIRT than in bzip2

+ deletions up to 30% are faster in CIRT than in bzip2

→ good for compressing text documents
→ good for compressing text columns of relational databases
Why separate text and structure compression?

- Query processing easier
- Compression factors for texts are weaker (factors 4-10) than for structure compression (factor 12-170)
- Usually more structure than text, e.g. 4:1
  → use optimal structure compression
Why tree compression for XML (or JSON or YAML)?

Goal: Reduce verbose structure of XML (overhead of e.g. factor 3-5) by XML compression (typically a factor 12-170)

Nice to have properties and requirements to compressed XML:
• queryable → at least as fast as on uncompressed XML
• updateable
• cacheable
• streamable
• transformable by XQuery/XSLT
• directly producable from SQL/XML

without decompression
**Why tree compression?**

Different from text search, Queries on compressed trees are faster than on uncompressed trees.

![Graph showing run-time vs data size for uncompressed and compressed trees.](image-url)
Why tree compression?

Different from text compression, Queries on compressed trees are faster than on uncompressed trees
Summary – Technologies for XML structure compression

**XML aware**

- **Encoding based**
  - FI
  - Succinct
  - …

- **Schema based**
  - DTD-Sub

- **Structure based**
  - DAG
  - RePair
  - …

- …

**Text compression**

- gzip
- bzip2
- …

IRT
Succinct Encoding - Summary

Technique: encoding based (roughly similar to FI, but ...) , uses bit-stream, text compression, inverted element lists

Compression strength:
  • comparable to gzip, bzip2
  • much stronger than FI, ...

Query Evaluation Performance:
  • faster than queries in JAXP
  • similar to queries on SAX (in our framework)

Further features:
  • streaming possible
  • updates without decompression possible
  • caching possible
  • XQuery (XSLT) transformation without decompression
XSDS - Compression Strength

+ significant compression improvement if XML Schema exists

e.g. MS-Word       (60% of the size of Microsofts compressed format)
e.g. SEPA          (11% of original size)
e.g. OpenStreetMap (10% of original size)
e.g. SOAP, OTA, ...

and wherever we have an XML standard for XML data
XSDS - Summary

Technique: remove tags given by (DTD or XML-) Schema

Compression strength:
• better than gzip, bzip2, FI, …

Query evaluation on XSDS compressed XML:
• 20 to 40 times faster than complete XML decompression
• 10 to 20 times faster than on uncompressed XML using our XML framework
• 3 to 7 times faster than JAXP

Further features: streamable, updateable, …
RePair – Compression Strength

Compression strength:
• better than Succint, DAG, gzip, bzip2, …

![Graph showing compression ratio for different files and compression methods]

- **SuccinctCompression**
- **DAGCompression**
- **RePAIRCompression**
- **FICompression**
- **GZIPCompression**
RePair Compression - Summary

Technique: sharing similar sub-trees (more general than DAG):

Compression strength:
  • better than gzip, bzip2, FI, succinct encoding…

Query evaluation:
  • usually faster than queries on uncompressed XML and faster than on succinct encoding

Nice to have properties:
  • streaming possible
  • (fast parallel) updates possible
  • caching (one of two strategies) possible
  • queryable archives of multiple versions (data deduplication)
A selection of further results

Storage and search on multiple versions

Processing and filtering huge data streams

Combining caching and compression

Querying transformed data through XQuery/XSLT views

Improvement and Generalization of SQL/XML query processing

Lessons learned about what to avoid

Overview of implemented modules
Grammar-based storage of multiple versions

V1: message from a customer
V1 \rightarrow M F C

V2: message from a **very important** customer
V2 \rightarrow M F I C

Grammar describes text patterns / data structures, i.e. grammar rules for common text phrases / data structures
\rightarrow common text / common structures are encoded only once
\rightarrow saves memory, energy, ...
\rightarrow faster search on multiple versions possible
works for text and for structured data
Search on multiple versions

V1: message from a customer
V1 \rightarrow M F C

V2: message from a very important customer
V2 \rightarrow M F I C

... F \rightarrow from a
C \rightarrow customer

Which version contains “important customer“?
Search through all versions possible at the same time
Combining XML Compression and Caching – Strategy 1

Client

XPath query

- which queries are stored in cache?

- send XPath query \( XQ \) and IDs \( I_1, \ldots, I_n \)

Server

- evaluate \( XQ \)

- calculate difference \( XQ - (I_1, \ldots, I_n) \)

- send difference \( XQ - (I_1, \ldots, I_n) \) to client

- integration of received data

- evaluate \( XQ \) on cache

- query results
Evaluation – Transferred Data Volume

- XMark documents
- 22 queries based on XPath-A of XPathMark
Caching Compressed XML – Strategy 1 - Summary

☐ Combination of compression and caching
  ■ avoids unnecessary decompression
  ■ less data transfer than with caching alone
  ■ less data transfer than with compression alone

☐ Generic approach
  ■ supports different compression techniques (e.g. DAG, Succinct)
  ■ supports different ID/Numbering schemes (e.g. OrdPath)
Caching – Strategy 2 – Performance evaluation

Dataset
XMark [Schmidt at al. 2002] (sizes 34kB to 11MB)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>XML</th>
<th>tree</th>
<th>compr. tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMark 0.0001</td>
<td>34kB</td>
<td>8kB</td>
<td>3.4kB</td>
</tr>
<tr>
<td>XMark 0.001</td>
<td>116kB</td>
<td>37kB</td>
<td>8kB</td>
</tr>
<tr>
<td>XMark 0.01</td>
<td>1.1MB</td>
<td>374kB</td>
<td>48kB</td>
</tr>
<tr>
<td>XMark 0.1</td>
<td>11MB</td>
<td>3.6MB</td>
<td>364kB</td>
</tr>
</tbody>
</table>

Transferred data volume
send/receive for 10 consecutive queries
Caching Compressed XML – Strategy 2 - Summary

Less transferred data than (compressed) query shipping through

→ only sending the **difference** in data values that are needed by Client for evaluating the given query.

→ the **difference** is determined by Server through running/simulating the given query

Data values are sent compressed, and in the order as needed by evaluator.

→ performance gain depends on the choice of (a) **query evaluator** and (b) **compressor**
Transformation of Compressed XML - Summary

Goal: fast transformation of compressed data

Techniques: compute paths to required data, copy compressed data if possible

+ faster than decompression + transformation + compression

+ sometimes faster than transforming uncompressed XML

Compression techniques supported: Succinct, DAG, RePair
Generating XML from SQL/XML Views

SQL-Query

```sql
select
    xmlelement( name "Kunde",
                xmlelement(name "Kundennr", K.Knr),
                xmlelement(name "Name", K.Name),
                xmlelement(name "Ort", K.Wohnort) )
    from Kunde K
    where ...
```

Kunde

<table>
<thead>
<tr>
<th>Kundennr</th>
<th>Name</th>
<th>Ort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Meier</td>
<td>Paderborn</td>
</tr>
<tr>
<td>5</td>
<td>Peters</td>
<td>Essen</td>
</tr>
</tbody>
</table>

Result

```xml
<Kunde>
    <Kundennr>1</Kundennr>
    <Name>Meier</Name>
    <Ort>Paderborn</Ort>
</Kunde>

<Kunde>
    <Kundennr>5</Kundennr>
    <Name>Peters</Name>
    <Ort>Essen</Ort>
</Kunde>
```
Generating compressed XML from SQL/XML Views

SQL-Query

```sql
select
    xmlelement( name "Kunde",
        xmlelement(name "Kundennr", K.Knr),
        xmlelement(name "Name", K.Name),
        xmlelement(name "Ort", K.Wohnort)
    )
from Kunde K
where ...
```

Schema

```xml
<! Element( Kunde ( Kundennr, Name, Ort ) ) >
<! Ergebnis ( kunde * ) >
```

Kunde

<table>
<thead>
<tr>
<th>Kundennr</th>
<th>Name</th>
<th>Ort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Meier</td>
<td>Paderborn</td>
</tr>
<tr>
<td>5</td>
<td>Peters</td>
<td>Essen</td>
</tr>
</tbody>
</table>

Result

```xml
&lt;Kunde&gt;
    &lt;Kundennr&gt;1&lt;/Kundennr&gt;
    &lt;Name&gt;Meier&lt;/Name&gt;
    &lt;Ort&gt;Paderborn&lt;/Ort&gt;
&lt;/Kunde&gt;
&lt;Kunde&gt;
    &lt;Kundennr&gt;5&lt;/Kundennr&gt;
    &lt;Name&gt;Peters&lt;/Name&gt;
    &lt;Ort&gt;Essen&lt;/Ort&gt;
&lt;/Kunde&gt;
```

Compressed data

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Meier Paderborn</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>5 Peters Essen</td>
</tr>
</tbody>
</table>

Generating compressed XML from SQL/XML Views
Many more improvements … but complex algorithms

Technology transfer includes: Help to avoid complex approaches!

Examples:

We have implemented combined compression strategies (succinct+DAG compression, …, RePair+DTD-Subtraction, etc.)

→ small improvement in compression in comparison to DAG/RePair alone,
   but algorithms get more complex (too complex for industry)

We have used functional dependencies to compress text data

→ little improvement in comparison to bzip2,
   but algorithms get more complex (too complex for industry)
Topics Around (Compressed) Structured Data

- Keyword search in compressed trees
- Querying and Updating compressed texts
- Caching compressed data and energy efficient transfer
- Compressing & archiving multiple data versions
- Top-down, bottom-up, and mixed query processing
- Query optimization on (compressed) XQuery, XSLT, and SQL/XML views
- Queries on XML streams
- Updating compressed trees

(Compressed) text and structured data