Web Science - Data Compression

Alexander Treptau
alexander.treptau@stud.uni-hannover.de
Overview

2 Papers from the topic **data compression**

- Exploring HTTP/2 Header Compression
- Reorganizing Compressed Text
Exploring HTTP/2 Header Compression (2017)

From:

Kazuhiko Yamamoto  Internet Initiative Japan Inc.
Tatsuhiro Tsujikawa  Z Lab Corporation
Kazuho Oku  Fastly
The HTTP Problem

- HTTP/1.1 stateless protocol
  - header redundancy
    - large cookies (for state management)
    - request & response with nearly the same data
  - average request header size: 800 bytes
Solution - New Protocol

- HTTP/2
  - header compression
  - Based on SPDY
    - Goals: reduced load latency & improve security
    - Designed with DEFLATE compression

- New Problem: insecure compression method
  - CRIME
A new Problem: CRIME

- SSL/TSL supports compression based on DEFLATE
  - Adaptive dictionary, *recurring strings* get encoded (Huffman coding)

**Szenario**

- Attacker can execute JavaScript and send requests
- Normally he shouldn't be able to access the secure data
  - e.g.: `Cookie: secret=5261434616`
- **But:** he can extend the request, iterative

```plaintext
POST / HTTP/1.1
HOST: l3s.de
[...]
Cookie: secret=5261434616
[...]
POST / HTTP/1.1
HOST: l3s.de
[...]
Cookie: secret=0
[...]
```
A new Problem: CRIME

- He can now try out & measure bytes
  (only length of compressed text known)
  - Cookie: secret=0, secret=1, …, secret=5
- DEFLATE will recognise the repeated and longer sequence
  - Better compression -> 1 byte less
  - Attacker knows now that the secret starts with 5
- Continuous until he finds the secret
Solution: compress Header by Header

- HPACK (RFC7541)
  - reoccurring string matches
  - Dictionary
    - Static (:method = GET, :status = 404, cookie, referer, ...)
    - Dynamic (all other headers)
      - first-in first-out, limited size list
  - static Huffman code
Request #1:

:authority: blog.cloudflare.com
**:method:**GET
:path: /
**:scheme:**https
accept: text/html, application/xhtml+xml, application/xml;q=0.9, image/webp, */q=0.8
accept-encoding: gzip, deflate, sdch, br
accept-language: en-US, en; q=0.8
cookie: 297 byte cookie
**upgrade-insecure-requests:**1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2853.0 Safari/537.36

HPACK Example

Request #2:
:authority:blog.cloudflare.com
:method:GET
:path:/assets/images/cloudflare-sprite-small.png
:**scheme:**https
accept:image/webp,image/*;q=0.8
**accept-encoding:**gzip, deflate, sdch, br
**accept-language:**en-US,en;q=0.8
**cookie:**same 297 byte cookie
referer:https://blog.cloudflare.com/assets/css/screen.css?v=2237be22c2
**user-agent:**Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/55.0.2853.0 Safari/537.36

~300 bytes encoded as 2 bytes
~130 bytes encoded as 2 bytes

Whats New? - The Paper Contributions

- Simplification of the Specification for HPACK
  - removed reverence set

- Improved implementation techniques
  - token based reverse indices
  - length guessing
  - pre-calculated state transitions
Simplification

- decoding / decompression
  - unique & clearly described
- encoding / compression
  - not defined brief

- Considered methods:
- Validation
  - run test cases
  - calculate compression rate
    - for sample data

<table>
<thead>
<tr>
<th></th>
<th>static table</th>
<th>dynamic table</th>
<th>reference set</th>
<th>Huffman encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>NaiveH</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Static</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>StaticH</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Linear</td>
<td>used</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>LinearH</td>
<td>used</td>
<td>used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Diff</td>
<td>used</td>
<td>used</td>
<td>used</td>
<td>not used</td>
</tr>
<tr>
<td>DiffH</td>
<td>used</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
</tbody>
</table>
Average Compression Ratio

<table>
<thead>
<tr>
<th>Method</th>
<th>Static Table</th>
<th>Dynamic Table</th>
<th>Reference Set</th>
<th>Huffman Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>NaiveH</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Static</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>StaticH</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Linear</td>
<td>used</td>
<td>used</td>
<td>not used</td>
<td>not used</td>
</tr>
<tr>
<td>LinearH</td>
<td>used</td>
<td>used</td>
<td>not used</td>
<td>used</td>
</tr>
<tr>
<td>Diff</td>
<td>used</td>
<td>used</td>
<td>used</td>
<td>not used</td>
</tr>
<tr>
<td>DiffH</td>
<td>used</td>
<td>used</td>
<td>used</td>
<td>used</td>
</tr>
</tbody>
</table>

Overhead:
- Naive: not used
- NaiveH: used
- Static: used
- StaticH: used
- Linear: used
- LinearH: used
- Diff: used
- DiffH: used

Compression ratio:
- Naive: 1.1
- NaiveH: 0.86
- Static: 0.84
- StaticH: 0.66

Inset:
- Linear: 0.39
- LinearH: 0.31
- Diff: 0.39
- DiffH: 0.31
Improvements

- Improved vs naive Implementation
  - Headers to Indices
  - Huffman Encoding
  - Huffman Decoding
Improvements - Headers to Indices

- **Decoding**
  - Converting indices to header names
  - $O(1)$ with the static & dynamic tables
- **Encoding**
  - Naive implementation - search table $O(n)$
  - Better - reverse indices
    - Finite map of finite maps $O(\log n)$
    - 2 lookups
      - Header names
      - Value & header pairs
    - Hash tables $O(1)$ average case
      - **Careful**: hash-collisions $O(n)$
Improvements - Headers to Indices

- Reduce 2 lookups to 1
  - Tokens for header names
    - Inside static table
  - One string comparison

```c
const token_t *
to_token(const char *name, size_t len) {
  switch (len) {
    ...
    case 3:
      switch (name[2]) {
        case 'a':
          if (memcmp(name, "vi", 2) == 0)
            return TOKEN_VIA;
          break;
        case 'e':
          if (memcmp(name, "ag", 2) == 0)
            return TOKEN_AGE;
          break;
      }
    break;
    case 4:
      switch (name[3]) {
        case 'e':
```
Improvements - Huffman Encoding

- Naive implementations:
  - String copy to temp buffer
  - Two buffer traversals

- Better: guess byte count
  - Average compression of 20%
  - Sometimes encoded string > original
    - ASCII cookie
  - Guess encoding length with 0.8 * length of input
Improvements - Huffman Decoding

- Naive implementations:
  - Transit binary tree bit by bit
- Better: transition destination by $n$ bits basis in advance
  - Converted to $2^n$-way tree
  - Number of tree nodes: $256 \times 2^n$
  - Static data
  - Shareable for all sessions between client & server
Evaluation - Encoding

- Enc 1: original, linear search
- Enc 2: Enc 1 + reverse indices, finite map of finite maps
- Enc 3: Enc 2 + token based reverse indices
- Enc 4: Enc 3 + length guessing

- 2.10x faster
Evaluation - Decoding

Figure 4: The performance progression of HPACK decoding for LinearH with $2^n$ way Huffman tree. The smaller, the better.
Conclusion

- HPACK reduces header redundancy issue
- Simplification can help
  - Why keep something complex if it doesn't contribute to the speed?

- Naive implementations are naive ;)
  - Lookout for improvements
Reorganizing Compressed Text (2008)

From:

Nieves R. Brisaboa *
Antonio Fariña *
Susana Ladra *
Gonzalo Navarro #

* Univ. of A Coruña, A Coruña, Spain.
# Univ. of Chile, Santiago, Chile
Why Text Compression?

● Save:
  ○ Disk space
  ○ Processing time
  ○ Transmission time
  ○ Disk transfer time
● Faster searching (up to 8x)
● Effectiveness (ratios ~ 25-35%)
Why Text Compression? - Effectiveness

- Word-based model
  - Words encoded instead of characters
  - More biased distribution frequencies than for characters
- Highly compressible

- Huffman code
  - Approach 25% ratio
- Plain Huffman or Restricted Prefix Byte Codes
  - Source symbols as sequences of bytes instead of bits
  - Degrades to 30% ratio
  - Faster decompression!
Searching - Self-Synchronized

- Other encoding methods
  - Tagged Huffman codes
  - End-Tagged Dense Codes
  - (s,c)-Dense Codes

- Worse compression ratios of up to 35%

- Self-Synchronized
  - Word boundaries distinguishable from anywhere
  - Random access
  - Boyer-Moore-like direct search
Idea / Proposal

- Reordering of the bytes (codewords of compressed text)
  - Wavelet-tree-like strategy
- Always self-synchronized
  - Even with encodings which are not accessed at any point
- Accessed at any point
- Use most efficient bytewise encodings with no penalty
Faster Searching - Implicit Indexing Properties

- Very little extra space
- Not searched in proportional time to text length
- Logarithmic (typical indexed techniques)

Compare:

- Original techniques with explicit inverted index
  - Not as efficient @ same amount of space
  - Inverted index more efficient with significant extra space
Basis Work, Inspiration & Examples

- Example
- Bytewise encoders
  - Plain Huffman
  - End-Tagged Dense Code
  - Restricted Prefix Byte Codes
- Wavelet trees
- Reorganization example
Example

- “LONG TIME AGO IN A GALAXY FAR FAR AWAY”
- \(b_3 b_5 | b_1 b_2 | b_3 b_4 | b_2 b_5 | b_1 b_3 | b_2 b_4 | b_5 | b_1 | b_1 | b_2 b_3 b_4\) (normal)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FREQ</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR</td>
<td>2</td>
<td>(b_1)</td>
</tr>
<tr>
<td>IN</td>
<td>1</td>
<td>(b_2 b_5)</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>(b_1 b_3)</td>
</tr>
<tr>
<td>LONG</td>
<td>1</td>
<td>(b_3 b_5)</td>
</tr>
<tr>
<td>AGO</td>
<td>1</td>
<td>(b_3 b_4)</td>
</tr>
<tr>
<td>TIME</td>
<td>1</td>
<td>(b_1 b_2)</td>
</tr>
<tr>
<td>AWAY</td>
<td>1</td>
<td>(b_2 b_3 b_4)</td>
</tr>
<tr>
<td>GALAXY</td>
<td>1</td>
<td>(b_2 b_4 b_5)</td>
</tr>
</tbody>
</table>
Bytewise Encoders

- Plain Huffman (25% -> 30%)
  - Basic byte-oriented variant (bytes as symbols of target alphabet)
  - Faster decompression & searching (no bit manipulations)
- End-Tagged Dense Code
  - Flag: first bit of each byte indicates if its the last byte of each codewords -> find easily codeword boundaries
    - Byte skipping possible
    - Prefix code
    - No need to use Huffman
      - Dense encoding possible over last 7 bits
  - Easier to build & faster in de- & compression
    - Patter compressing + string matching
- Restricted Prefix Byte Codes
End-Tagged Dense Code

- Byte-oriented
- Decreasing probabilities \( \{p_i\} \ 0 \leq i < n \)
- Sequence of symbols of \( b \) bits
  - Digits in base \( 2^{b-1} \ [0, 2^{b-1} - 1] \)
  - Last \( [2^{b-1}, 2^{b-1} - 1] \)
    - Sequentially assignment
- Code depends on rank
  - Not actual frequency
  - Store sorted vocabulary with compressed text
- Better than Huffman, where shape of tree needed
End-Tagged Dense Code

- Simple:
  - Sort source symbols
    - Decreasing frequency
  - Sequentially assign codewords

- Get codeword $x$ for symbol $i$:
  - Add $2^{b-1}$ to the last digit

$$x = i - \frac{2^{(b-1)k} - 2^b - 1}{2^{b-1} - 1}$$
Restricted Prefix Byte Codes

- Length of codeword saved in first byte
- Encoding schema
  - \((v_1, v_2, v_3, v_4)\) 4-tuple
  - \(v_1 + v_2 + v_3 + v_4 \leq R\)
    - 1-byte code words: \(v_1\)
    - 2-byte code words: \(R v_2\)
    - 3-byte code words: \(R^2 v_2\)
    - 4-byte code words: \(R^3 v_3\)
  - \(v_1 + Rv_2 + R^2 v_3 + R^3 v_4 \geq n\)

- More flexible codeword length than ETDC
  - Improved compression ratio
  - Self-synchronization

\(R :=\) radix, typically 256
Wavelet Trees

- Succinct data structure
  - Solving rank & select queries, large alphabets
- Sequence of symbols $B$
  - $\text{rank}_b (B,i) = y$ if symbol $b$ appears $y$ times in prefix $B_{1,i}$
  - $\text{select}_b (B,j) = x$ if symbol $b$ appears at pos $x$ in sequence $B$, for the $j^{th}$ occurrence of $b$
- Balanced binary tree
  - Divides into 2 halves at each node
  - Bitmaps to mark which side was chosen
  - Recursive sequence handling @ childs
- Constant time
  - rank: top down
  - select: bottom up

https://en.wikipedia.org/wiki/Wavelet_Tree#/media/File:Wavelet_tree.png
Algorithm

Compression

● 2 Passes
  ○ 1st
    ■ Obtain vocabulary & model/frequencies
    ■ Assign codewords
  ○ 2nd
    ■ Translate words into codewords
● Spread codewords bytes in wavelet tree
● Precalculation possible (#nodes and size of each)
● Array of markers allow sequential filling
● Concatenation
Example

- “LONG TIME AGO IN A GALAXY FAR FAR AWAY”
- $b_3 b_5 | b_1 b_2 | b_3 b_4 | b_2 b_5 | b_1 b_3 | b_2 b_4 b_5 | b_1 | b_1 | b_2 b_3 b_4$ (normal)
- $b_3 b_5 | b_2 b_1 | b_4 b_3 | b_2 b_5 | b_3 b_1 | b_4 b_5 b_2 | b_1 | b_1 | b_2 b_4 b_3$ (reorganized)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>FREQ</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR</td>
<td>2</td>
<td>$b_1$</td>
</tr>
<tr>
<td>IN</td>
<td>1</td>
<td>$b_2 b_5$</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>$b_3 b_1$</td>
</tr>
<tr>
<td>LONG</td>
<td>1</td>
<td>$b_3 b_5$</td>
</tr>
<tr>
<td>AGO</td>
<td>1</td>
<td>$b_4 b_3$</td>
</tr>
<tr>
<td>TIME</td>
<td>1</td>
<td>$b_2 b_1$</td>
</tr>
<tr>
<td>AWAY</td>
<td>1</td>
<td>$b_2 b_4 b_3$</td>
</tr>
<tr>
<td>GALAXY</td>
<td>1</td>
<td>$b_4 b_5 b_2$</td>
</tr>
</tbody>
</table>
Example

Word: LONG TIME AGO IN A GALAXY FAR FAR AWAY
Position: 1 2 3 4 5 6 7 8 9

Symbol Code

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR</td>
<td>b₁</td>
</tr>
<tr>
<td>IN</td>
<td>b₂ b₅</td>
</tr>
<tr>
<td>A</td>
<td>b₃ b₁</td>
</tr>
<tr>
<td>LONG</td>
<td>b₃ b₅</td>
</tr>
<tr>
<td>AGO</td>
<td>b₄ b₃</td>
</tr>
<tr>
<td>TIME</td>
<td>b₂ b₁</td>
</tr>
<tr>
<td>AWAY</td>
<td>b₂ b₄ b₃</td>
</tr>
<tr>
<td>GALAXY</td>
<td>b₄ b₅ b₂</td>
</tr>
</tbody>
</table>

Diagram:

- **B2**
  - TIME
  - IN
  - AWAY
    - 1 b₁
    - 2 b₅
    - 3 b₄

- **B3**
  - LONG
  - A
    - 1 b₅
    - 2 b₁

- **B4**
  - AGO
  - GALAXY
    - 1 b₃
    - 2 b₅

- **B2 B4**
  - AWAY
    - 1 b₃
Example: 5th word?

Position: 1 2 3 4 5 6 7 8 9

\[ \text{rank}_{b_3}(\text{Root}, 5) = 2 \]
Example

Word: LONG TIME AGO IN A GALAXY FAR FAR AWAY
Position: 1 2 3 4 5 6 7 8 9

\[
\begin{array}{cccccccc}
& b_3 & b_2 & b_4 & b_2 & b_3 & b_4 & b_1 & b_1 & b_2 \\
\end{array}
\]

1) select_{b_2}(B4B5,1) = 1
2) select_{b_5}(B4,1) = 2
3) select_{b_4}(Root,2) = 6

SYMBOL CODE

FAR \quad b_1
IN \quad b_2 \quad b_5
A \quad b_3 \quad b_1
LONG \quad b_3 \quad b_5
AGO \quad b_4 \quad b_3
TIME \quad b_2 \quad b_1
AWAY \quad b_2 \quad b_4 \quad b_3
GALAXY \quad b_4 \quad b_5 \quad b_2
Bottleneck rank & select

● Brute force: sequentially counting all occurrence for each byte
  ○ No extra space/structure needed
● Improvement: Sequences with superblocks
  ○ Superblocks sb into b blocks with size: \( n / (sb \times b) \)
● 1st level: #occurrences of each byte
  ○ From the sequence beginning
  ○ To start of superblock
● 2nd level (can be represented with lower bits)
  ○ From the beginning of each superblock
  ○ To start of block
● Improvement from \( O(n) \) to \( O(n/(sb \times b)) \)
● Binary search for select \( \text{rank}_{bi}(B,x) = j \)
Results

- Dataset: large text collections
  - Congressional Record 1993 (CR)
  - Ziff Data 1989-1990 (ZIFF)
  - Aggregated collection (ALL)
    - CR & ZIFF
    - AP Newswire 1988
    - Financial Times 1991 to 1994

Table 1: Description of the corpora used.

<table>
<thead>
<tr>
<th>CORPUS</th>
<th>size (bytes)</th>
<th>num words</th>
<th>voc. size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>51,085,545</td>
<td>10,113,143</td>
<td>117,713</td>
</tr>
<tr>
<td>ZIFF</td>
<td>185,220,211</td>
<td>40,627,131</td>
<td>237,622</td>
</tr>
<tr>
<td>ALL</td>
<td>1,080,720,303</td>
<td>228,707,250</td>
<td>885,630</td>
</tr>
</tbody>
</table>
Small Reminder

Old variants:
- PH = Plain Huffman
- ETDC = End-Tagged Dense Code
- RPBC = Restricted Prefix Byte Codes

New wavelet-tree reorganization variants:
- WPH, WTDC, WRPBC

And with superblocks:
- WPH+, WTDC+, WRPBC+
### Results

#### Table 2: Compression ratio (in %).

<table>
<thead>
<tr>
<th></th>
<th>PH</th>
<th>ETDC</th>
<th>RPBC</th>
<th>WPH</th>
<th>WTDC</th>
<th>WRPBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>31.06</td>
<td>31.94</td>
<td>31.06</td>
<td>31.06</td>
<td>31.95</td>
<td>31.07</td>
</tr>
<tr>
<td>ZIFF</td>
<td>32.88</td>
<td>33.77</td>
<td>32.88</td>
<td>32.88</td>
<td>33.77</td>
<td>32.89</td>
</tr>
<tr>
<td>ALL</td>
<td>32.83</td>
<td>33.66</td>
<td>32.85</td>
<td>32.83</td>
<td>33.66</td>
<td>32.85</td>
</tr>
</tbody>
</table>

#### Table 3: Compression time.

<table>
<thead>
<tr>
<th></th>
<th>PH</th>
<th>ETDC</th>
<th>RPBC</th>
<th>WPH</th>
<th>WTDC</th>
<th>WRPBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>-2%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>-4%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>2.886</td>
<td>2.870</td>
<td>2.905</td>
<td>3.025</td>
<td>2.954</td>
<td>2.985</td>
</tr>
<tr>
<td>ALL</td>
<td>71.317</td>
<td>71.452</td>
<td>71.614</td>
<td>74.631</td>
<td>73.392</td>
<td>74.811</td>
</tr>
</tbody>
</table>

#### Table 4: Decompression time.

<table>
<thead>
<tr>
<th></th>
<th>PH</th>
<th>ETDC</th>
<th>RPBC</th>
<th>WPH</th>
<th>WTDC</th>
<th>WRPBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>-20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>-25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>0.574</td>
<td>0.582</td>
<td>0.583</td>
<td>0.692</td>
<td>0.697</td>
<td>0.702</td>
</tr>
<tr>
<td>ZIFF</td>
<td>2.309</td>
<td>2.254</td>
<td>2.289</td>
<td>2.661</td>
<td>2.692</td>
<td>2.840</td>
</tr>
</tbody>
</table>
Results

- Corpus: ALL
- 100 random distinct words
  - freq. <= 50k
- count pattern
- locate first
- locate all
- extract all snippets of 10 words around pattern

Table 6: Searching capabilities

<table>
<thead>
<tr>
<th></th>
<th>Count (ms)</th>
<th>First (ms)</th>
<th>Locate (s)</th>
<th>Snippet (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PH</td>
<td>2605.600</td>
<td>48.861</td>
<td>2.648</td>
<td>7.955</td>
</tr>
<tr>
<td>ETDC</td>
<td>1027.400</td>
<td>22.933</td>
<td>0.940</td>
<td>1.144</td>
</tr>
<tr>
<td>RPBC</td>
<td>1996.300</td>
<td>41.660</td>
<td>2.009</td>
<td>7.283</td>
</tr>
<tr>
<td>WPH</td>
<td>238.500</td>
<td>17.173</td>
<td>0.754</td>
<td>72.068</td>
</tr>
<tr>
<td>WTDC</td>
<td>221.900</td>
<td>17.882</td>
<td>0.762</td>
<td>77.845</td>
</tr>
<tr>
<td>WRPBC</td>
<td>238.700</td>
<td>17.143</td>
<td>0.773</td>
<td>75.435</td>
</tr>
<tr>
<td>WPH+</td>
<td>0.015</td>
<td>0.017</td>
<td>0.123</td>
<td>5.339</td>
</tr>
<tr>
<td>WTDC+</td>
<td>0.015</td>
<td>0.014</td>
<td>0.129</td>
<td>6.130</td>
</tr>
<tr>
<td>WRPBC+</td>
<td>0.015</td>
<td>0.018</td>
<td>0.125</td>
<td>5.036</td>
</tr>
</tbody>
</table>

(better = lower)
Implicit vs Classical Indexes

- $WT_1$ uses 1 block per 2,000 bytes
  1 superblock per each 8 blocks
- $WT_2$ uses 1 block per 7,000 bytes
  1 superblock per each 20 blocks

Table 7: Sizes of the compared WTDC+ and II structures.

<table>
<thead>
<tr>
<th>Index type</th>
<th>Wavelet trees</th>
<th>Block-add Inv Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$WT_1$</td>
<td>$WT_2$</td>
</tr>
<tr>
<td>size (MB)</td>
<td>457.32</td>
<td>397.97</td>
</tr>
<tr>
<td>C. ratio (%)</td>
<td>44.37</td>
<td>38.61</td>
</tr>
</tbody>
</table>
Implict vs Classical Indexes

Table 8: Searching for words: WTDC+ Vs Block-addressing Inverted Index. Times in seconds.

<table>
<thead>
<tr>
<th></th>
<th>Freq.</th>
<th>$WT_1$</th>
<th>$II_{s8,b16}$</th>
<th>$WT_2$</th>
<th>$II_{s32,b256}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locate</td>
<td>1-100</td>
<td>0.005</td>
<td>0.020</td>
<td>0.008</td>
<td>0.270</td>
</tr>
<tr>
<td></td>
<td>101-1,000</td>
<td>0.134</td>
<td>0.580</td>
<td>1.343</td>
<td>7.260</td>
</tr>
<tr>
<td></td>
<td>1,001-10,000</td>
<td>0.478</td>
<td>5.820</td>
<td>1.715</td>
<td>42.130</td>
</tr>
<tr>
<td></td>
<td>&gt;10,000</td>
<td>3.702</td>
<td>31.240</td>
<td>6.748</td>
<td>66.450</td>
</tr>
<tr>
<td>Snippet</td>
<td>1-100</td>
<td>0.028</td>
<td>0.030</td>
<td>0.064</td>
<td>0.280</td>
</tr>
<tr>
<td></td>
<td>101-1,000</td>
<td>0.771</td>
<td>0.640</td>
<td>2.845</td>
<td>7.300</td>
</tr>
<tr>
<td></td>
<td>1,001-10,000</td>
<td>5.456</td>
<td>6.130</td>
<td>13.251</td>
<td>42.440</td>
</tr>
<tr>
<td></td>
<td>&gt;10,000</td>
<td>44.115</td>
<td>33.700</td>
<td>102.722</td>
<td>68.870</td>
</tr>
</tbody>
</table>
Implict vs Classical Indexes

Table 9: Searching for phrases: WTDC+ Vs Block-addressing Inverted Index. Times in seconds.

<table>
<thead>
<tr>
<th></th>
<th>#words</th>
<th>WT1</th>
<th>II_{s8,b16}</th>
<th>WT2</th>
<th>II_{s32,b256}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locate</td>
<td>2</td>
<td>1.920</td>
<td>5.570</td>
<td>4.250</td>
<td>22.880</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.020</td>
<td>4.980</td>
<td>4.090</td>
<td>16.500</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1.300</td>
<td>2.020</td>
<td>3.050</td>
<td>10.990</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.940</td>
<td>1.250</td>
<td>2.630</td>
<td>8.470</td>
</tr>
<tr>
<td>Snippet</td>
<td>2</td>
<td>5.070</td>
<td>5.750</td>
<td>11.730</td>
<td>23.050</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2.100</td>
<td>4.990</td>
<td>4.230</td>
<td>16.500</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1.300</td>
<td>2.020</td>
<td>3.060</td>
<td>11.000</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.950</td>
<td>1.250</td>
<td>2.630</td>
<td>8.460</td>
</tr>
</tbody>
</table>
Conclusion

● Extend known algorithms: PH & RPBC
  ○ efficient compressors
  ○ not self-synchronizing
● Simple step: reorganization
  ○ efficient search, compression & decompression
    ■ implicit indexing
    ● alternative to inverted indexes
    ■ tradeoff space for speed
References


Discussion