Engineering Rank and Select Queries on Wavelet Trees

Roland Larsen Pedersen

Datalogi, Aarhus Universitet

Thesis defence

June 25, 2015

Overview

1. What is a Wavelet Tree?
   - Definitions
   - Constructing the Wavelet Tree
2. Queries
   - Rank
   - Select
3. Applications
   - Information Retrieval and Compression
4. Experiments and Results
5. Conclusion

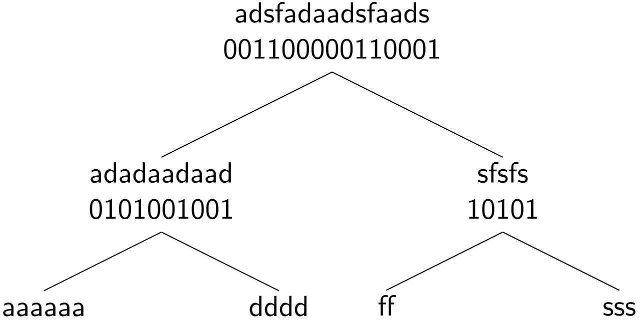
What is a wavelet tree?

Wavelet Tree: Definitions

- Balanced binary tree.
- Stores a *sequence* $S[1, n] = c_1 c_2 c_3 \ldots c_n$ of *symbols* $c_i \in \Sigma$, where $\Sigma = [1 \ldots \sigma]$ is the *alphabet* of S.
- Height $h = \lceil \log \sigma \rceil$.
- $2\sigma - 1$ nodes
- Construction time: $O(n \log \sigma)$
- Memory usage: $O(n \log \sigma + \sigma \cdot ws)$ bits.

Constructing the Wavelet Tree

- The wavelet tree is constructed recursively.
- Each node calculates the middle character of Σ and uses it to set the bits in the bitmap and split S in two substrings S_{left} and S_{right}.



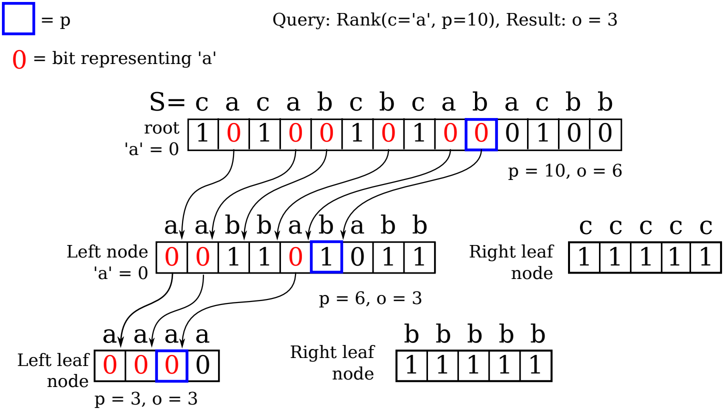
$S = \text{adsfadaadsfaads}, \Sigma = \text{adfs}$

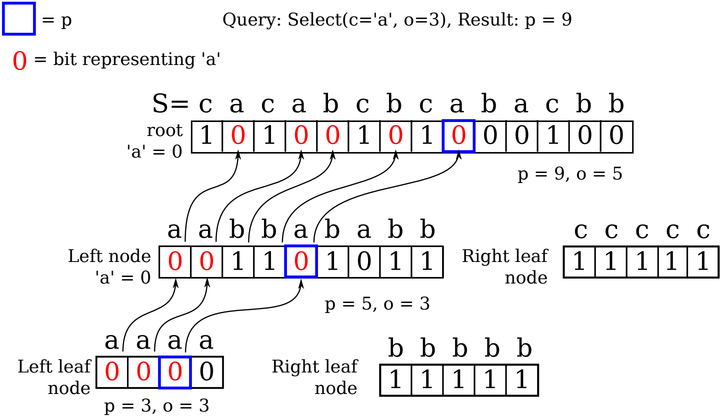
Queries

Wavelet Tree: Queries

- The wavelet tree supports three queries:
  - **Access(p)**: Return the character c at position p in sequence S.
    - Running time: $O(n \log \sigma)$.
  - **Rank(c, p)**: Return the number of occurrences of character c in S up to position p.
    - Running time: $O(n \log \sigma)$.
  - **Select(c, o)**: Return the position of the oth occurrence of character c in S.
    - Running time: $O(n \log \sigma)$

Rank on a Wavelet Tree




= p
Query: Select(c='a', o=3), Result: p = 9
0 = bit representing 'a'
S= c a c a b c b c a b a c b b
root
'a' = 0
1 0 1 0 0 1 0 1 0 0 0 1 0 0
p = 9, o = 5
a a b b a b a b b
Left node
'a' = 0
0 0 1 1 0 1 0 1 1
p = 5, o = 3
c c c c c
Right leaf
node
1 1 1 1 1
a a a a
Left leaf
node
0 0 0 0
p = 3, o = 3
b b b b b
Right leaf
node
1 1 1 1 1

Applications

Information Retrieval: Applications

- Information Retrieval
  - Positional inverted index.
    - For each word: Return positions of occurrences.
  - Document retrieval.
    - Return what document a word appears in.
  - Range Quantile Query.
    - Return the kth smallest number within a subsequence of a given sequence of elements.
  - FM-count.
    - Return number of occurrences of a pattern p in S.

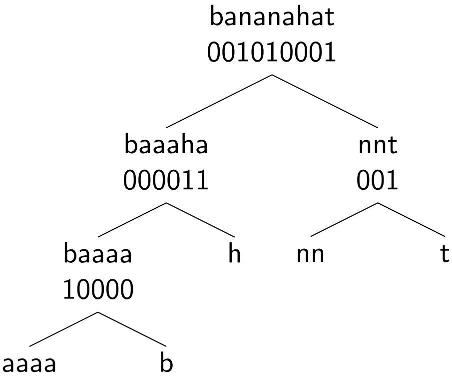
Compression: Applications

- Compression
  - Zero-order entropy compression (H_0) using a RLE Wavelet Tree or a Huffman Shaped Wavelet Tree.
  - Higher-order entropy compression (H_k) using Burrows-Wheeler transformation and a RLE wavelet tree.
  - $H_k <= H_0 <= \log \sigma$.

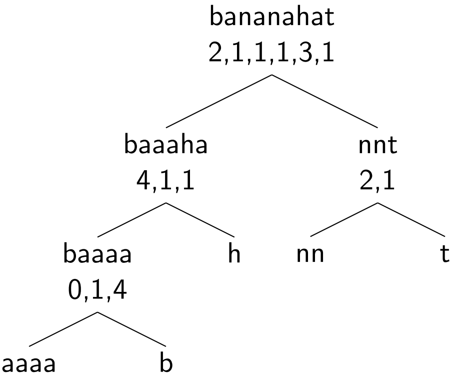
Compression: Run-length encoding

- Example: $RLE(\text{aaaaabbbaacccccaaaaa}) = \text{a5,b3,a2,c5,a5}$.
- Binary example: $RLE(00000000001111100000) = 10, 5, 5$
- Query by reversing RLE. It takes linear time $O(n)$ to reverse. Rank and select query time becomes $O(2n \log \sigma) = O(n \log \sigma)$
- Space complexity $O(nH_0(S))$

RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



(a) Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



(b) RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$

Compression: Burrows-Wheeler transform

- BWT permutes the order of the characters. If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row.
- As a result it groups symbols more which improves the effect of Run-length encoding
- BWT is reversible
- Combined with RLE Wavelet Tree it achieves H_k compression.

BWT example

$S =$ bananahat.

$$\begin{bmatrix} bananahat\# \\ ananahat\#b \\ nanahat\#ba \\ anahat\#ban \\ nahat\#bana \\ ahat\#banan \\ hat\#banana \\ at\#bananah \\ t\#bananaha \\ \#bananahat \end{bmatrix} \Rightarrow \begin{bmatrix} \#bananaha\mathbf{t} \\ ahat\#bana\mathbf{n} \\ anahat\#ba\mathbf{n} \\ ananahat\#\mathbf{b} \\ at\#banana\mathbf{h} \\ bananahat\# \\ hat\#banan\mathbf{a} \\ nahat\#ban\mathbf{a} \\ nanahat\#b\mathbf{a} \\ t\#bananah\mathbf{a} \end{bmatrix}$$

$BWT(S) =$ tnnbhaaaa.

Burrows-Wheeler reverse transform example

$S = \text{dca}$

$$M = \begin{bmatrix} dca\# \\ ca\#d \\ a\#dc \\ \#dca \end{bmatrix} \Rightarrow M' = \begin{bmatrix} \#dc\mathbf{a} \\ a\#d\mathbf{c} \\ ca\#\mathbf{d} \\ dca\boldsymbol{\#} \end{bmatrix}$$

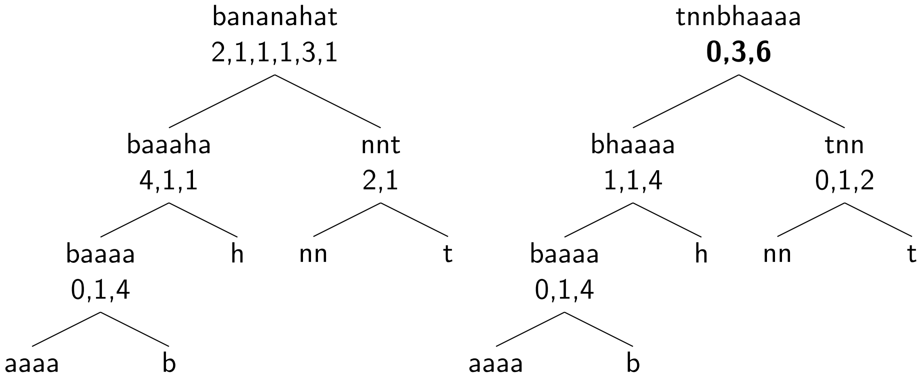
$BWT(S) = acd$

Reverse BWT:

Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4
a	$\#$	$a\#$	$\#d$	$a\#d$	$\#dc$	$a\#dc$	$\#dca$
c	a	ca	$a\#$	$ca\#$	$a\#d$	$ca\#d$	$a\#dc$
d	c	dc	ca	dca	$ca\#$	$dca\#$	$ca\#d$
$\#$	d	$\#d$	dc	$\#dc$	dca	$\#dca$	**dca#**

*$\#$ = end of line character

RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$



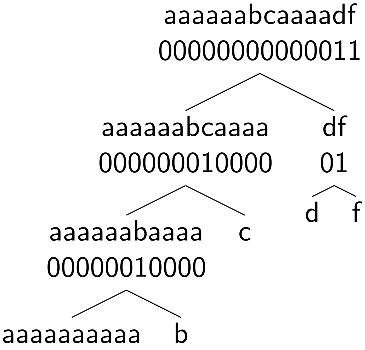
(a) RLE Wavelet Tree on string *bananahat* with alphabet $\Sigma = abhnt$

(b) BWT RLE Wavelet Tree on string *tnnbhaaaa* with alphabet $\Sigma = abhnt$

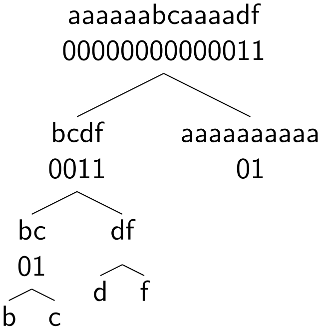
Huffman shaped wavelet tree

- Use Huffman codes of symbols to shape the tree.
- Most frequent symbols at the top of the tree.
- Least frequent symbols at the bottom of the tree.
- Best using non-uniformly distributed data like a natural language text.

Huffman Shaped Wavelet Tree: Example



(a) Balanced Wavelet tree: 39 bits



(b) Huffman-shaped wavelet tree: 22 bits

Huffman Shaped WT: Space complexity

- Balanced version: $O(n \log \sigma + \sigma \cdot \text{ws})$ bits
- Huffman-shaped: $O(n(H_0(S) + 1) + \sigma \cdot \text{ws})$ bits. [Efficient Compressed Wavelet Trees over Large Alphabets by Navarro et al.]

Experiments and Results

Focus of experiments

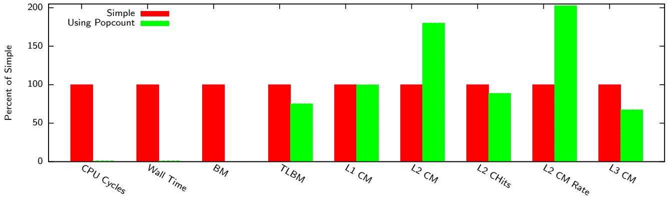
- Focus on optimizing and observing the effect of hardware penalties.
  - Cache Misses.
  - Branch Mispredictions.
  - Translation Lookaside Buffer (TLB) Misses.

Experiments

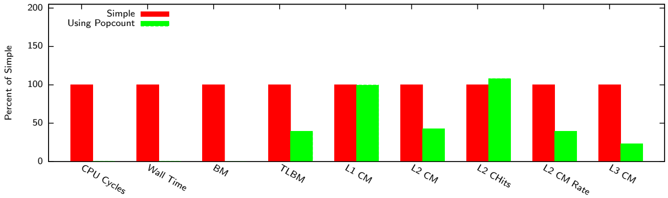
- Calculate binary rank and select using popcount.
- Pre-compute binary rank values in blocks.
- Concatenate bitmaps and Page-align blocks.
- Block size dependence on input n.
- Pre-compute cumulative sums of rank values.

Calculate binary rank and select using popcount

- Rank: Running time $O(n \log \sigma)$.
- Select: Running time $O(n \log \sigma)$.



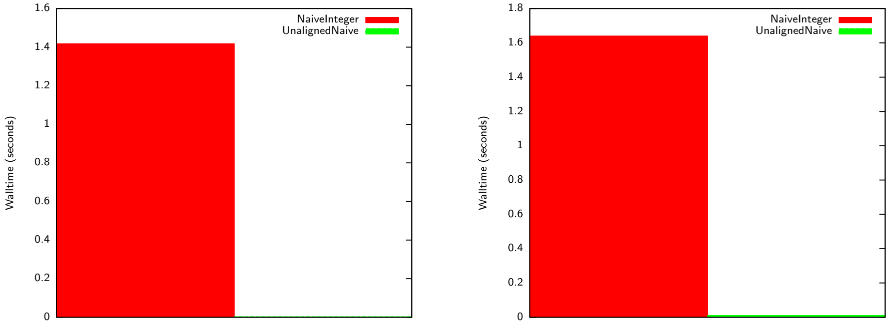
(a) Rank



(b) Select

Pre-compute binary rank values in blocks

- Rank: Running time $O((\frac{n}{b} + b)\log\sigma)$.
- Select: Running time $O((\frac{n}{b} + b)\log\sigma)$.



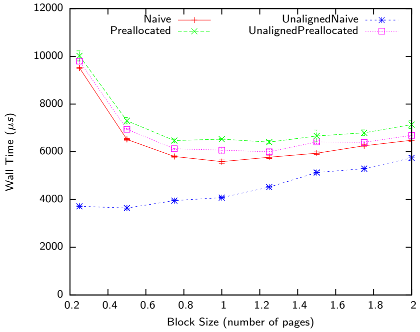
(a) Rank (b) Select

Figure : Comparison of wall time of rank and select queries between SimpleNaive not using precomputed values and UnalignedNaive using precomputed values.

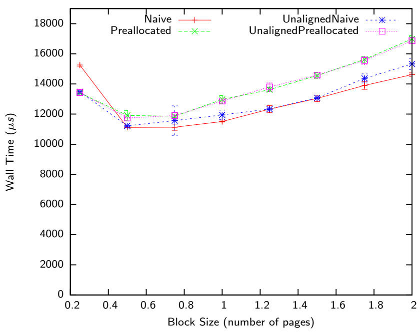
The various precomputed versions

Name	Concatenated Bitmaps	Page-aligned Blocks
Preallocated	yes	yes
UnalignedPreallocated	yes	no
Naive	no	yes
UnalignedNaive	no	no

Running time: Pre-compute binary rank values in blocks



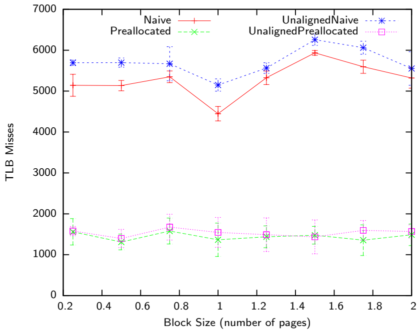
(a) Rank: Running Time



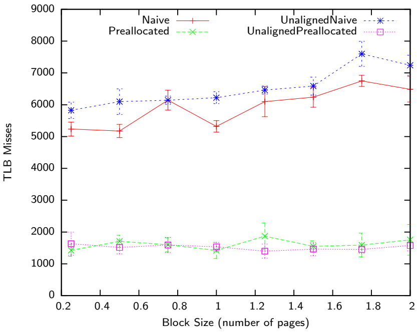
(b) Select: Running Time

Best Block size: $\frac{1}{2}$ page size $= \frac{1}{2} * 4096$ bytes $= 2048$ bytes.

Rank and select TLB misses



(a) Rank: TLB Misses

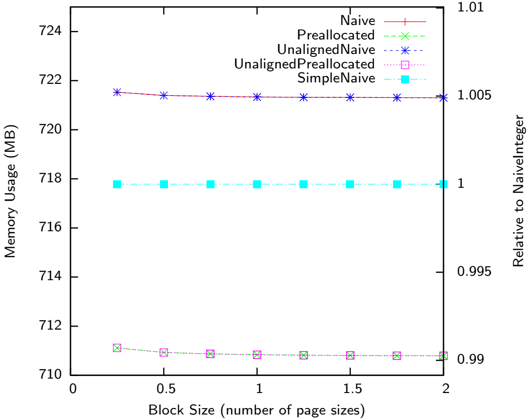


(b) Select: TLB Misses

- *Naive* does reduce TLB misses because of page alignment.
- Concatenated bitmaps reduces TLB misses, but page-aligning does not have much effect.

Memory usage: Pre-compute binary rank values in blocks

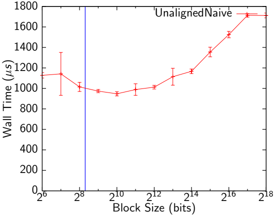
- There are $O(\frac{n}{b})$ blocks per level of the tree, and so an extra memory consumption of $O(\frac{n}{b}\log\sigma)$ words making the total memory consumption $O(n\log\sigma + (\sigma + \frac{n}{b}\log\sigma)\cdot ws)$ bits.



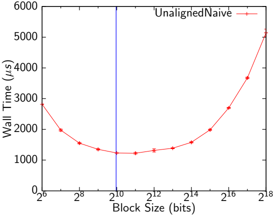
Block size dependence on input size n

- Costs $O(\frac{n}{b} + b)$ to calculate the binary rank.
- Costs $O(\frac{n}{b})$ to scan the blocks, and $O(b)$ to calculate the rank within a single block using popcount.
- Optimal block size $b = \sqrt{n}$.
- A wavelet tree has many bitmaps of varying sizes n.

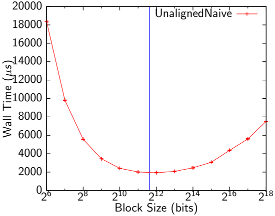
Experiment: Block size dependence on input size n for Rank



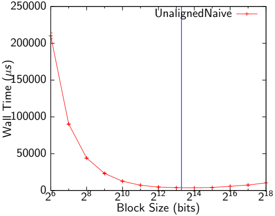
(a) $n = 10^5$



(b) $n = 10^6$



(c) $n = 10^7$

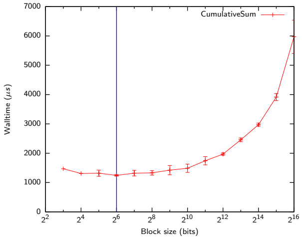


(d) $n = 10^8$

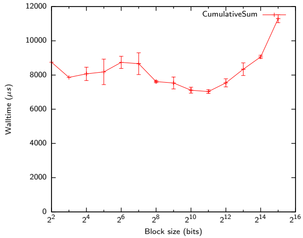
Cumulative sum

- Each block contain sum of previous blocks.
- Binary rank in $O(b)$ time in stead of $O(\frac{n}{b} + b)$ time.
- Binary search in select.
- Work per level change from $O(\frac{n}{b} + b)$ to $O(\log \frac{n}{b} + b)$. Select query total work $O((\log \frac{n}{b} + b) \log \sigma)$.
- Best block size does not depend on n for Rank.
- A block size below 64 bits should not be an improvement because popcount works on words of size 64 bits.

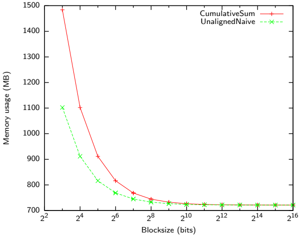
Cumulative sum: Rank and Select running time



(a) Rank.



(b) Select.



(c) Memory Usage.

Conclusion: What did we learn?

- What effect hardware can have on running time and memory.
- How to do tests and how to show their results in an understandable way.
- The wavelet tree has many applications.
- The wavelet tree is great for compression of natural language texts.
- Choosing the right test parameters and what data to use can be difficult.
- How to do literature search and how important it is.
- In general, improvements that reduced the raw amount of computations and memory accesses needed were a big improvement.
- That a simple concept can be very difficult to implement.
- Gained experience with profilers and hardware measurement tools (cachegrind, PAPI, Massif)

Conclusion: Problems and questions we faced

- Should we use uniform or non-uniform data?
- How should non-uniform data be distributed?
- How large alphabet and input size should we use?
- Debugging implementation errors in c++
- Making the implementations work
- Should we have focused on compression in stead?
- PAPI produced weird memory measurements. Figuring out what was wrong took some time.
- How to avoid introducing bias in tests.

The End