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
   - Compression
     - Run-length encoding
     - Burrows-Wheeler Transform
     - Huffman Shaped Wavelet tree
4. Experiments and Results
   - Popcount
   - Pre-compute binary rank values in blocks
   - Concatenated Bitmaps and/or page-aligned blocks
   - Cumulative Sum of precomputed rank values

What is a wavelet tree?

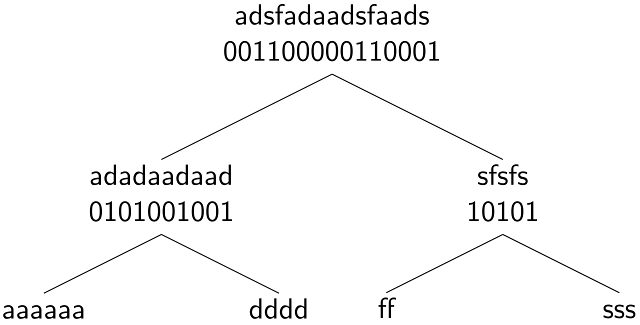
Wavelet Tree: Definitions

- In its basic form, the wavelet tree is a balanced binary tree.
- It 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.
- The tree has height $h = \lceil \log \sigma \rceil$, and $2\sigma - 1$ nodes, with σ of those as leaf nodes and $\sigma - 1$ as internal nodes.

Constructing the Wavelet Tree

- The wavelet tree is constructed recursively, starting at the root node and moving down the tree, with each node in the tree receiving a string constructed by its parent, except the root node that receives the full input string.
- 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}.

Wavelet Tree Example



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

Construction time and memory usage

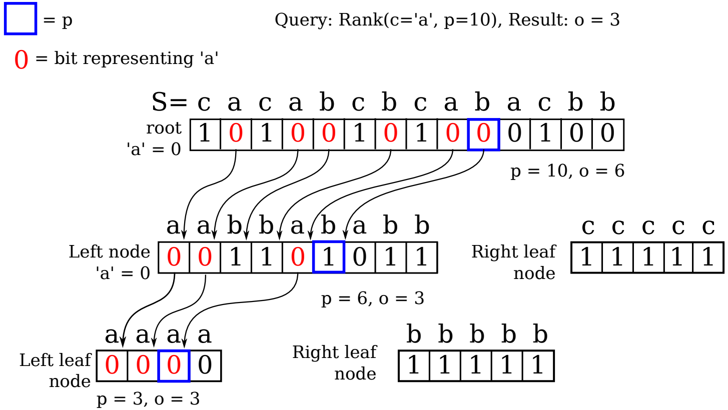
- Construction time: $O(n \cdot h) = O(n \log \sigma)$
  - The Wavelet Tree can theoretically be constructed in $O(n \cdot h) = O(n \log \sigma)$ time as the sum of the lengths of the strings being processed at any single layer of the tree is the length of the input string to the tree.
- Memory usage: $O(n \log \sigma + \sigma \cdot ws)$ bits
  - At each level in the tree at most n bits are stored in the bitmaps in total, making $n \cdot h = n \cdot \log \sigma$ an upper bound to the total number of bits that a wavelet tree stores in its bitmaps.
  - In addition to this, each node takes some constant amount of machine words of space, and there are $2\sigma - 1$ nodes in the tree. ws is the size of our machine words. This makes the total memory consumption $O(n \log \sigma + \sigma \cdot ws)$ bits.

Queries

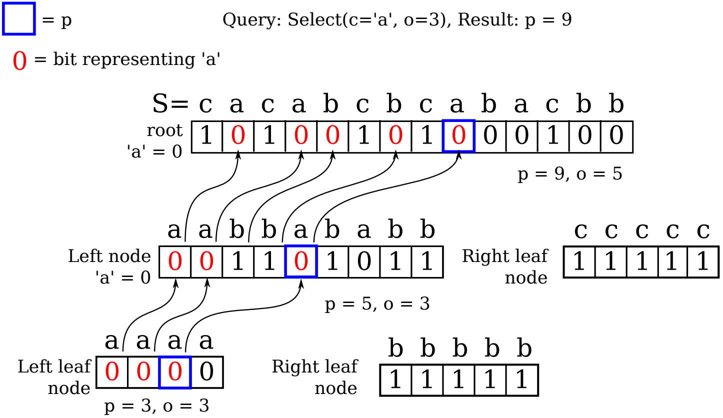
Wavelet Tree: Queries

- The wavelet tree supports three queries:
  - **Access(p)**: Return the character c at position p in sequence S.
    - Running time: $O(n \log \sigma)$.
    - We have not implemented Access because it resembles Rank.
  - **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



Select on a Wavelet Tree



Applications

Information Retrieval: Applications

- Information Retrieval
  - Positional inverted index
  - Document retrieval
  - 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.

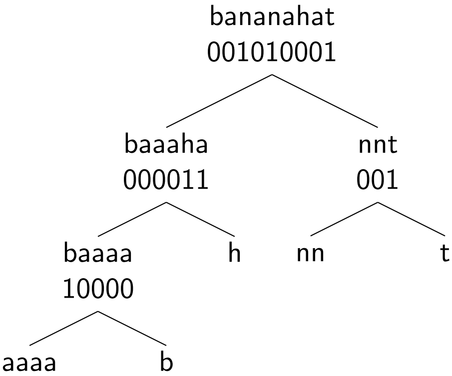
Compression: Applications

- 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.
  - $H_k <= H_0 <= \log \sigma$.

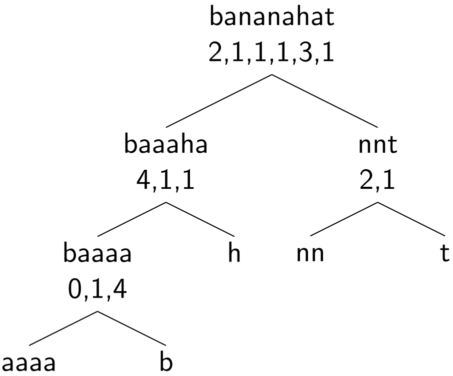
Compression: Run-length encoding

- Run-length encoding counts the number of consecutive occurrences of a symbol and substitutes the consecutive occurrences with the symbol followed by its number of occurrences.
- Example: RLE(aaaaabbbaacccccaaaaa) = a5,b3,a2,c5,a5.
- Binary example: $RLE(00000000001111100000) = 10, 5, 5$
  - We can avoid specifying the symbol by assuming that 0 is always the first symbol.
  - If the binary number begins with a 1 we just add a 0 to the beginning of the result.
- Query by reversing RLE. It takes linear time $O(n)$ to reverse. Rank and select query time becomes $O(2n\log\sigma) = O(n\log\sigma)$
- Achieves space complexity within H_0

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

$S = \text{dca}$

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

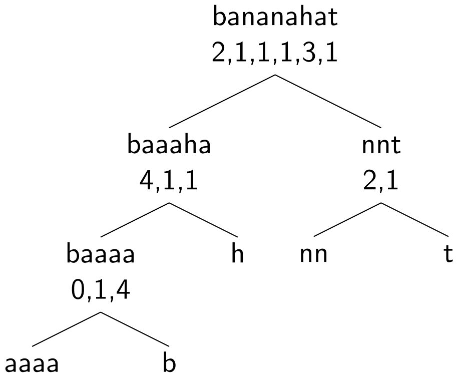
$BWT(S) = acd$

Reverse BWT:

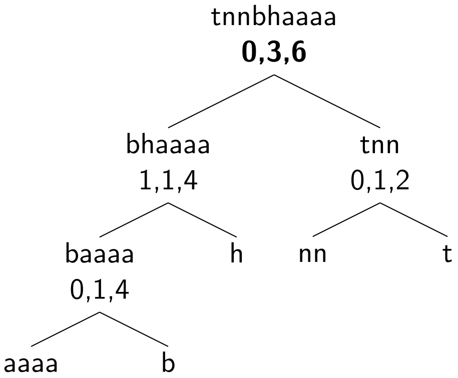
Add 1	Sort 1	Add 2	Sort 2	Add 3	Sort 3	Add 4	Sort 4
a	$\#$	$a\#$	$\#d$	$a\#d$	$\#dc$	$a\#dc$	$\#dca$
c	a	ca	$a\#$	$ca\#$	$a\#d$	$ca\#d$	$a\#dc$
d	c	dc	ca	dca	$ca\#$	$dca\#$	$ca\#d$
$\#$	d	$\#d$	dc	$\#dc$	dca	$\#dca$	$\mathbf{dca\#}$

*$\#$ = end of line character

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



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

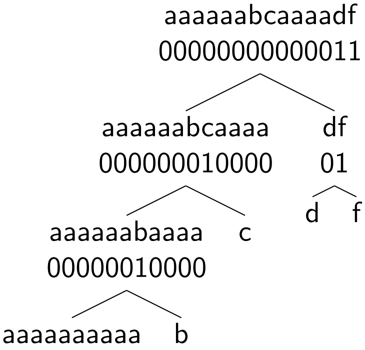


(b) BWT RLE Wavelet Tree on string *tnnbhaaaa* with alphabet $\Sigma = abhnt$

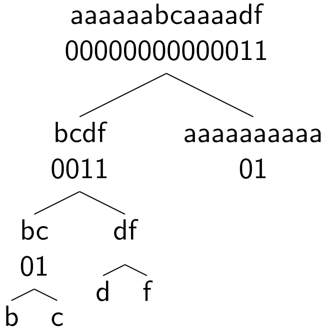
Huffman shaped wavelet tree

- Use Huffman codes of symbols to shape the tree
- A Huffman code is a binary value assigned to each symbol. The symbol with the highest frequency gets the lowest value.
- Shaping the tree based on Huffman codes places the most frequent symbols at the top of the tree and least frequent symbols at the bottom of the tree.
- Huffman shaping only makes sense on 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: $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits
- Huffman-shaped: $n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(\sigma \log n)$ bits. [Efficient Compressed Wavelet Trees over Large Alphabets by Navarro et al.]
- Huffman-shaped + Compressed Bitmap (RLE): $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \log n)$ bits.

Experiments and Results

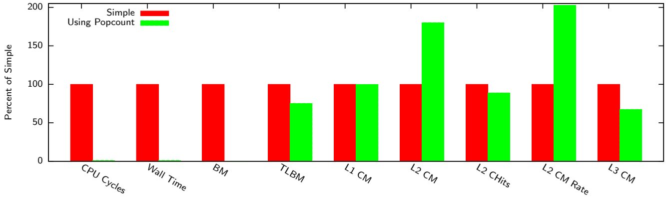
Focus of experiments

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

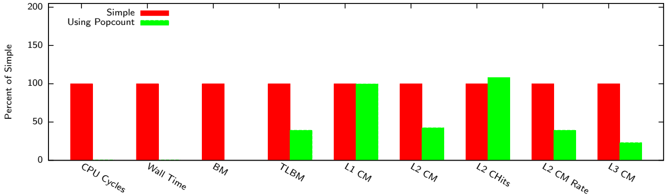
Experiments

- 1. Calculate binary rank and select using popcount
- 2. Pre-compute binary rank values in blocks
- 3. Block size dependence on input n
- 4. Pre-compute cumulative sums of rank values
- 5. Branchless select query
- 6. Queries on skewed cumulative sum wavelet tree

Calculate binary rank and select using popcount



(a) Rank

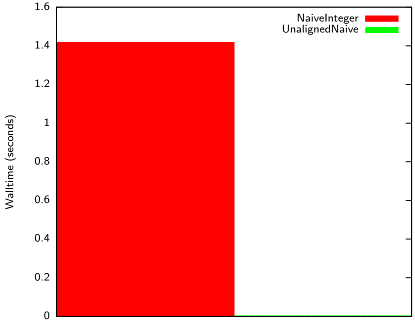


(b) Select

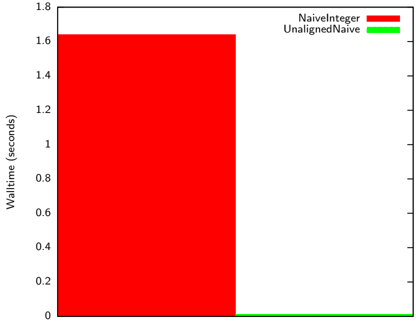
Figure : Rank and select queries using simple binary rank and select vs. rank and select queries using binary rank and select using the popcount instruction. Y-Axis is index 100 of the simple queries, that is, every value is percent of the value for the simple query.

Pre-compute binary rank values in blocks

- Pre-compute binary rank values in blocks to reduce amount of popcount calls.



(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.

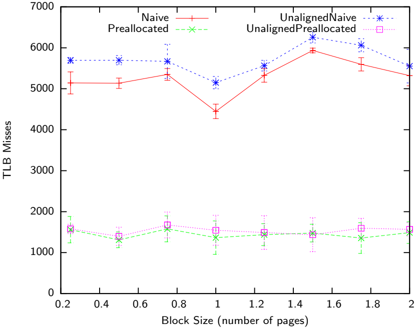
Concatenated Bitmaps and/or page-aligned blocks

- Concatenated bitmaps
  - Save all bitmaps as one large bitmap to reduce memory usage by removing pointers to individual bitmaps.
  - Access using an offset (ulong) and a size for the bitmap (uint).
  - This uses $64 + 32 = 96$ bits per node vs. individual bitmaps with pointers taking up $3 \times 64 = 192$ bits per node.
  - Individual bitmaps for each node are word-aligned, and the bits between the last used bit and the end of the last used word are wasted.
- Page-aligned blocks
  - To reduce TLB misses
  - If the blocks are not aligned with the memory pages a block can be partly in one page and partly in another.
  - Page-aligning makes sure that blocks does not span page boarders.

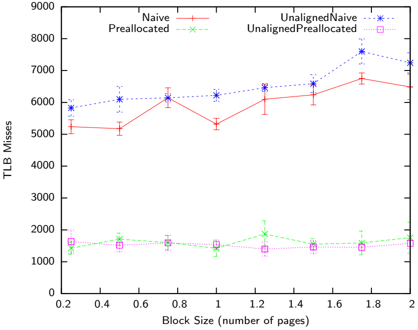
The various precomputed versions

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

Rank and select TLB misses



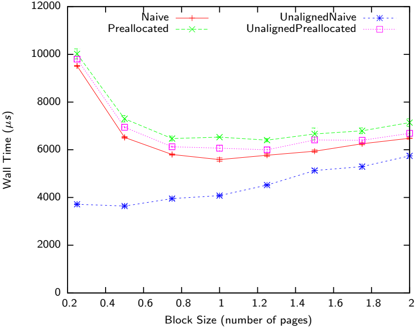
(a) Rank: TLB Misses



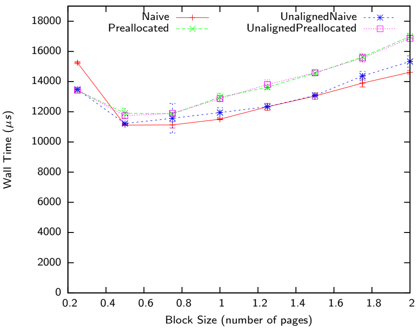
(b) Select: TLB Misses

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

Running time: Pre-compute binary rank values in blocks



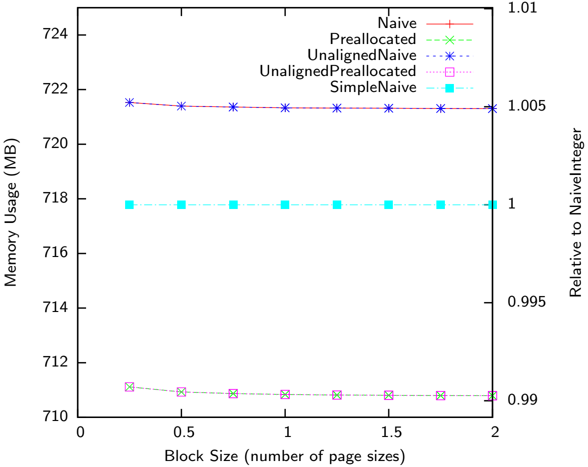
(a) Rank: Running Time



(b) Select: Running Time

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

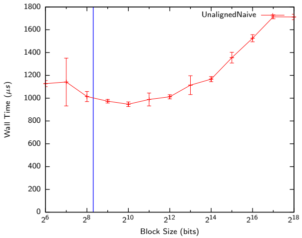
Memory usage: Pre-compute binary rank values in blocks



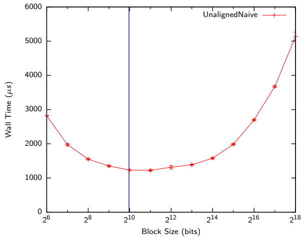
Block size dependence on input size n

- When using lookups of precomputed values It costs $O(\frac{n}{b} + b)$ to calculate the binary rank.
- It costs $O(\frac{n}{b})$ to scan the blocks, and $O(b)$ to calculate the rank within a single block using popcount. The optimal block size should be one that minimizes this.
- The derivative of $\frac{n}{b} + b$ is $1 - \frac{n}{b^2}$ and its root is $n = b^2$ making the optimal block size $b = \sqrt{n}$.
- This is only the optimal block size for a single bitmap, and a wavelet tree has many bitmaps of varying sizes n that are lower near the leaves.

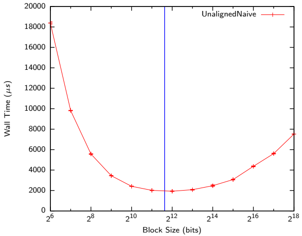
Experiment: Block size dependence on input size n for Rank



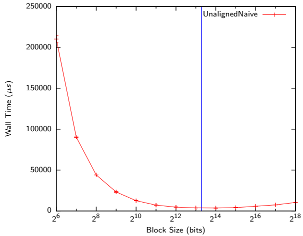
(a) $n = 10^5$



(b) $n = 10^6$



(c) $n = 10^7$



(d) $n = 10^8$

Cumulative sum

-

The End