The Wavelet Trie: Maintaining an Indexed Sequence of Strings in Compressed Space

Roberto Grossi and Giuseppe Ottaviano

Università di Pisa

Abstract

We study the problem of maintaining *sequences of strings* under insertion/deletion and indexed queries in compressed space.
We introduce a new data structure, the *Wavelet Trie*, that supports efficient operations in space close to the information-theoretic lower bound.

Rank/Select Sequences

Let $S = \langle s_0, \dots, s_{n-1} \rangle$ sequence of symbols from alphabet S_{set}.

- $\mathbf{Access}(i)$: access the i-th symbol s_i
- $\mathbf{Rank}(s, \mathrm{pos})$: count the number of occurrences of s before position pos
- $\mathbf{Select}(s, \mathrm{idx})$: find the position of the idx-th occurrence of s

Can use **Rank** to count the number of occurrences of a symbol in an interval of the sequence, **Select** to iterate all the occurrences of a symbol.

Example: $S = \underset{0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10}{\texttt{abracadabra}}$, $S_{\text{set}} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{r}\}$.

- $\mathbf{Access}(0) = \texttt{a}$
- $\mathbf{Rank}(\texttt{a}, 3) = 1$
- $\mathbf{Rank}(\texttt{a}, 4) = 2$
- $\mathbf{Select}(\texttt{a}, 2) = 5$

"Easy" for the binary case, $S_{\text{set}} = \{0, 1\}$. Sequences from a binary alphabet are called *bitvectors*.

Dynamic sequences

A data structure for storing sequences is *dynamic* if it supports the following operations:

- $\mathbf{Insert}(s, \mathrm{pos})$: insert the symbol s immediately before s_{pos}
- $\mathbf{Delete}(\mathrm{pos})$: delete the symbol at position pos

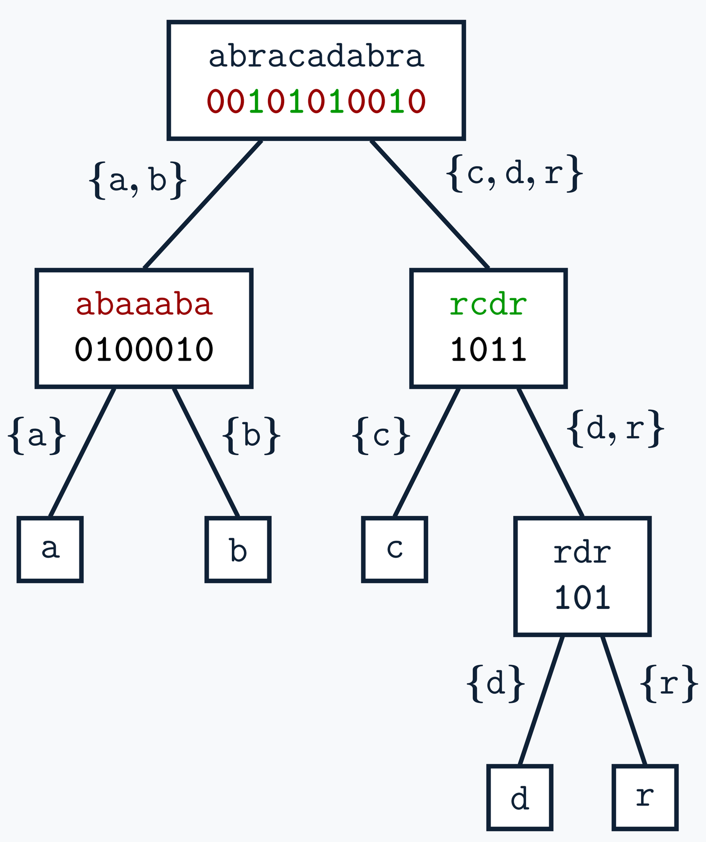
We define **Append** as a special case of **Insert**:

- $\mathbf{Append}(s)$: append the symbol s at the end of the sequence

We call a data structure that only supports ***Append*** *append-only*.
Example scenarios: logging, time series, column-oriented databases, ...

Wavelet Trees

Wavelet Trees reduce queries on alphabet S_{set} to queries on bitvectors.
Example for $S_{\text{set}} = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{r}\}$, $S = \texttt{abracadabra}$:



- Balanced tree built on S_{set}
- Each node splits the alphabet into two subsets
- At each node, sequence split into two subsequences
- At each node, 0s in correspondence with left subsequence, 1s with right subsequence
- Support **Access** and **Rank** by performing **Rank** operations top-down on bitvectors, **Select** by bottom-up **Select**

By using *dynamic bitvectors* on the nodes, **Insert** and **Delete** can be supported, but *the alphabet S_{set} must be set a priori.*

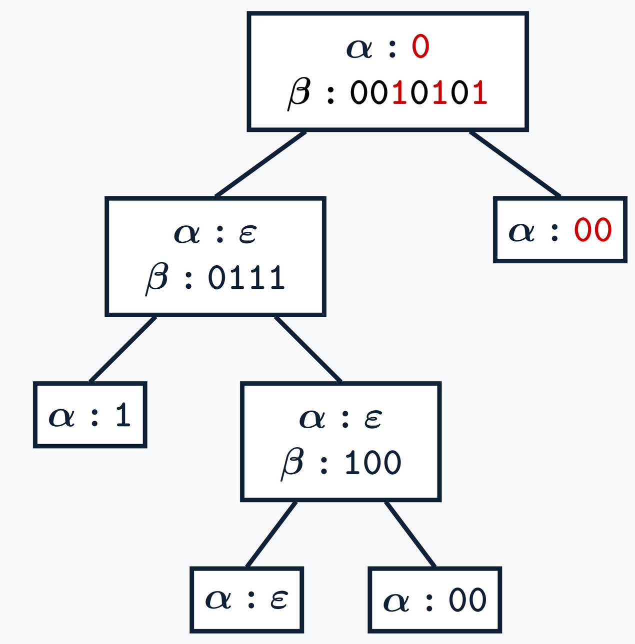
This limitation prevents the use of dynamic Wavelet Trees for large alphabets and database applications.

The Wavelet Trie

We consider the problem of *sequences of binary strings*, i.e. $S_{\text{set}} \subset \{0, 1\}^*$.
No loss of generality: non-binary strings, integers, ... can be binarized.
Example: $S = \langle \underset{0}{0001}, \underset{1}{0011}, \underset{2}{0100}, \underset{3}{00100}, \underset{4}{0100}, \underset{5}{00100}, \underset{6}{0100} \rangle$.

- $\mathbf{Rank}(0100, 2) = 0$
- $\mathbf{Rank}(0100, 3) = 1$
- $\mathbf{Select}(0100, 2) = 6$

We introduce the *Wavelet Trie* on S:



- Tree structure is the Patricia Trie on S_{set}, each node corresponds to a subsequence with a common prefix
- α is the longest common prefix of the subsequence
- Each subsequence is partitioned based on the first bit after α
- Bitvector β discriminates between left and right subsequence
- Same operations as Wavelet Tree

New prefix operations

The Wavelet Trie enables two new operations:

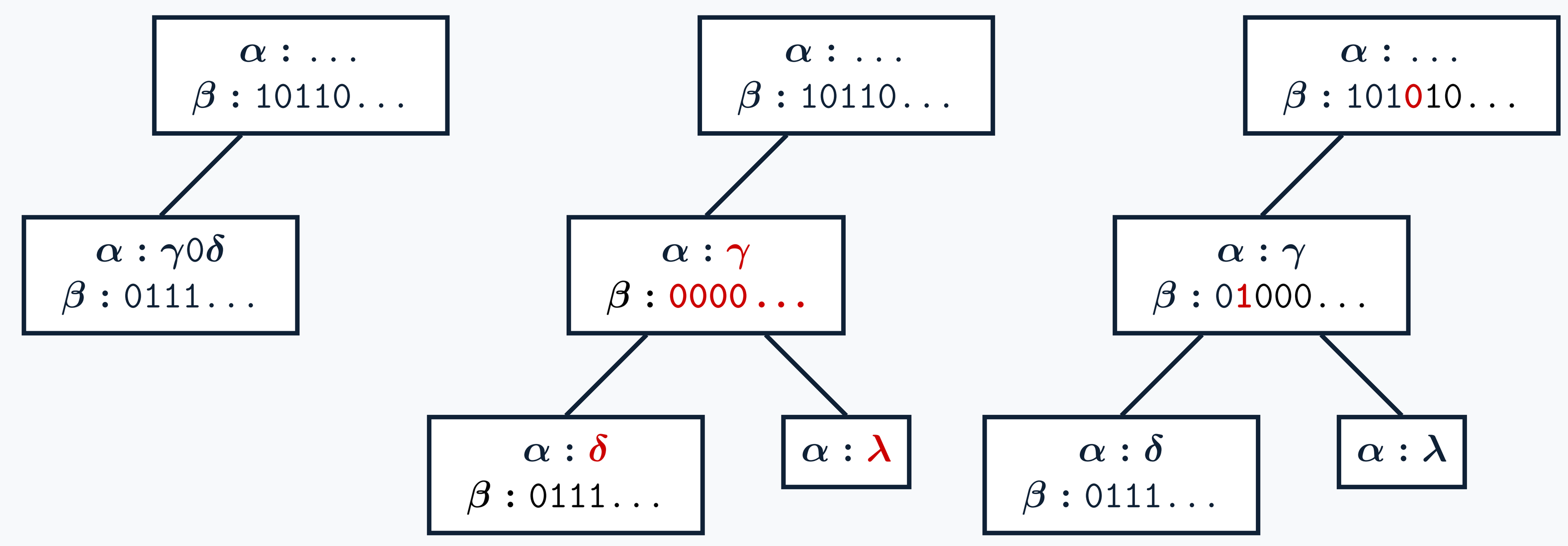
- $\mathbf{RankPrefix}(p, \mathrm{pos})$: count the strings prefixed by p before position pos
- $\mathbf{SelectPrefix}(p, \mathrm{idx})$: find the position of idx-th string prefixed by p

Example application: S is a sequence of URLs, find the number of URLs from a given hostname in a given range, or enumerate them.

String set updating

The Patricia trie structure enables updates to the alphabet S_{set}. When an unseen string is inserted, an existing node is split. The new node is given a constant bitvector.

For example, $\mathbf{Insert}(\dots\gamma 1 \lambda, \mathrm{pos})$ performs the following operations:



Delete is symmetric. To support efficient bitvector initialization with a constant sequence, we introduce new dynamic compressed bitvector data structures.

This yields the first *dynamic compressed sequence* data structure that efficiently supports a *dynamic alphabet*.

Time and space

	Query	Append	Insert	Delete	Space (in bits)
Static	$O(\lvert s\rvert + h_s)$	–	–	–	$\mathrm{LB} + o(\tilde{h}n)$
Append-only	$O(\lvert s\rvert + h_s)$	$O(\lvert s\rvert + h_s)$	–	–	$\mathrm{LB} + \mathrm{PT} + o(\tilde{h}n)$
Fully-dynamic	$O(\lvert s\rvert + h_s \log n)$	$O(\lvert s\rvert + h_s \log n)$	$O(\lvert s\rvert + h_s \log n)$	$O(\lvert s\rvert + h_s \log n)^\dagger$	$\mathrm{LB} + \mathrm{PT} + O(nH_0)$

- Sequence of n strings $\langle s_0, \dots, s_{n-1} \rangle$, h_s: number of nodes traversed in the trie for string s, $\tilde{h}$: average height
- LB: information theoretic lower bound $\mathrm{LT} + nH_0$, where LT is the lower bound for the *set* of strings S_{set}
- PT: space for dynamic Patricia trie on the *set* of strings S_{set}

Symposium on Principles of Database Systems (PODS) 2012