Information Systems 73 (2018) 25–34

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Rank and select: Another lesson learned

Szymon Grabowski[*], Marcin Raniszewski

Lodz University of Technology, Institute of Applied Computer Science, Al. Politechniki 11, Łódź 90–924, Poland

ARTICLE INFO

Article history:
Received 15 January 2017
Revised 10 July 2017
Accepted 2 December 2017
Available online 5 December 2017

2010 MSC:
68W32

Keywords:
Binary sequences
Rank
Select
Compressed data structures

ABSTRACT

Rank and select queries on bitmaps are essential building bricks of many compressed data structures, including text indexes, membership and range supporting spatial data structures, compressed graphs, and more. Theoretically considered yet in 1980s, these primitives have also been a subject of vivid research concerning their practical incarnations in the last decade. We present a few novel rank/select variants, focusing mostly on speed, obtaining competitive space-time results in the compressed setting. Our findings can be summarized as follows: (*i*) no single rank/select solution works best on any kind of data (ours are optimized for bit arrays obtained from wavelet trees for real text datasets, but also work well for LOUDS-encoded XML tree layouts), (*ii*) it pays to efficiently handle blocks consisting of all 0 or all 1 bits, (*iii*) compressed select does not have to be significantly slower than compressed rank at a comparable memory use.

1. Introduction

Rank and select are essential building bricks of many compressed data structures, and text indexes in particular. In their most frequently used binary incarnation, they can be defined as follows: given a bit-vector $B[0\dots n-1]$, $rank_b(B,\ i)$ returns the number of occurrences of symbol b in the prefix $B[0\dots i]$ and $select_b(B,\ i)$ returns the position of the i-th occurrence of symbol b in B, where $b \in \{0,\ 1\}$.

Note that $rank_0(B, i) = i + 1 - rank_1(B, i)$, hence it is enough to directly support the rank only for one binary symbol (e.g., 1). There is no similar relation binding the values of $select_0(B,\ i)$ and $select_1(B,\ i)$.

It is known for at least two decades [1–3] that these operations can be performed in constant time, using the extra space of $o(n)$ bits. Raman et al. [4] showed how to compress the vector B to $nH_0(B) + o(n)$ bits, where $H_0(B)$ is the order-0 entropy of B, and still support rank and select in constant time.

Much research has been dedicated to construct rank and select solutions, both compressed and non-compressed, to answer the queries as fast as possible in practice. Especially in the compressed setting also the lower-order terms of the space matter, hence the practical questions involve two aspects: the query time and the space used by the data structure.

In this work we propose several novel rank/select variants, focusing mostly on speed, obtaining competitive space-time results in the compressed setting. By being competitive we mean good performance on real-world data (taken from FM-indexes and XML tree layouts in LOUDS representation). The performance of our variants on random data with uneven ratio of set to unset bits is not so competitive, as most of them cannot attain significant (or any) compression in such a scenario. Yet, on the real-world test datasets our rank solutions belong to the fastest, and can be beaten in compression only for the price of much longer times. Also for select queries our solutions are typically Pareto-optimal, although here more differences can be seen from dataset to dataset.

The roadmap of the paper is as follows. The next section briefly recalls the history of practical solutions for rank and select on binary sequences, starting from the non-compressed ones. Section 3 introduces our algorithms. Section 4 presents the experimental rank/select results on multiple datasets, also including count query times for the FM-index with several ranks plugged in. The last section concludes.

2. Related work

The original rank solution, by Jacobson [1], which needs $O(n \log\log n / \log n)$ bits of space[1] in addition to the original bit-vector, performs three memory accesses (to one superblock counter and one block counter, plus a lookup into a table with precomputed popcount answers), which often translates into three cache misses, a significant penalty. González et al. [5] proposed a scheme with one level of blocks followed by sequential scan.

* Corresponding author.
E-mail address: sgrabow@kis.p.lodz.pl (S. Grabowski).

[1] Logarithms of base 2 are used throughout the paper.

https://doi.org/10.1016/j.is.2017.12.001
0306-4379/© 2017 Elsevier Ltd. All rights reserved.

In theoretical terms, this solution no longer obtains both constant time and sublinear extra space, yet it fares well practically and is very simple.

Vigna [6] interleaved data from different levels to improve access locality. Gog and Petri [7] carried this idea even further, interleaving the precomputed rank values and the bit-vector data. The sequential scans over small blocks of data are performed with an efficient 64-bit hardware popcount instruction (*popcnt*), available in Intel and AMD processors since 2008. In the manner of the González et al. solution, Gog and Petri store only one level of precomputed ranks, yet data from two successive cache lines can be sometimes read in their scheme. More or less at the same time, Zhou et al. [8] presented similar ideas of a cache-friendly layout, interleaving data and using the hardware popcount instruction, in a three-level logical solution handling bit-vectors with maximum size of 2^{64}. In a recent work, Grabowski et al. [9] showed a solution with one cache miss in the worst case. They achieve it with interleaving 64-bit precomputed rank fields with $512 - 64 = 448$ bits of data. As 64 bits per rank is more than needed, part of this field stores popcount values for some prefixes of the following block, thus saving on the *popcnt* invocations.

Kärkkäinen et al. [10] proposed a hybrid scheme for the compressed rank, where the bit-vector is divided into blocks and each block is encoded separately, choosing one of a few different methods, depending on its "local" performance. This general approach was implemented in a version with three encodings: no compression (i.e., the block kept verbatim), storing the positions of the minority bits (zeros or ones, whichever have fewer occurrences in the block), and run-length encoding for runs of zeros and ones. To make the data structure even more compact, blocks are grouped into superblocks. Thanks to it, the blocks' header data store ranks and offsets to the beginnings of the encoded block bodies with respect to the beginning of the superblock rather than the beginning of the whole structure. Only the rank operation is supported, yet the authors mention briefly a possibility to extend their scheme in order to support selects too.

As it can be implied from the literature, an efficient select is harder to design than an efficient rank, even in the non-compressed variant. Clark [2] was the first to show a constant-time select with $3n/\lceil \log\log n \rceil + O(n^{1/2} \log n \log\log n)$ bits of extra space. The solution is relatively complicated and needs at least 60% space overhead. González et al. [5] noticed that implementing select with binary search over a rank structure is often superior (in spite of having $O(\log n)$ time complexity), both in execution times and the space overhead. Yet, for large inputs or for low densities of set bits (assuming that we focus on the $select_1$ query), Clark's solution dominates. More recently, Gog and Petri [7] presented a practical implementation of the Clark select idea, reducing its worst-case space overhead to less than 29%. Very recently, Pandey et al. [11,12] proposed an ingenious fast select implementation, making use of two non-standard x86 CPU instructions, PDEP and TZCNT. Significant speedups compared to select variants from [8] and the SDSL library are reported, yet the non-standard instructions require an Intel's Haswell (or newer) CPU.

Okanohara and Sadakane [13] were the first to consider practical *compressed* implementations of rank/select structures and they introduced four novel r/s dictionaries (each of which was based on different ideas), reaching different space/time tradeoffs in theory and in practice. For example, they offer to answer rank or select queries in a few tenths of a microsecond (on a 3.4 GHz Intel Xeon) spending about 25% of extra space for densely (50%) populated bit-vectors. The ideas used in their work include (among others) the enumerative code (related to [4]), compacting the bit-vector by removing (and flagging) small blocks consisting of only zeros and recursively applying the same technique to the remaining blocks, and gap encoding. The Okanohara and Sadakane paper is important also because they were the first to distinguish between dense and sparse bit-vectors and offering different solutions depending on the case.

Table 1
The impact of the fraction of mono-blocks f on the average number of memory accesses per query and the total query times in the rank variants *basic* and *cf*. The block size is 64 bytes. The numbers of memory accesses are calculated from the formulas on f given in Section 3.1, in the paragraph on the *cf* variant. The times are expressed in nanoseconds.

Dataset	Fraction of mono-bl. [%]	Basic, accesses	cf, accesses	Basic, time	cf, time
dna200-bal	73.60	1.2640	1.1943	23.02	25.82
english200-bal	52.95	1.4705	1.2491	32.61	34.88
proteins200-bal	45.66	1.5434	1.2481	37.00	37.27
sources200-bal	53.82	1.4618	1.2485	32.16	34.48
xml200-bal	78.50	1.2150	1.1688	21.39	23.81
dna200-huff	9.03	1.9097	1.0822	36.68	30.96
english200-huff	23.74	1.7626	1.1811	40.87	36.02
proteins200-huff	3.84	1.9616	1.0370	44.40	31.81
sources200-huff	36.78	1.6322	1.2325	39.39	36.88
xml200-huff	68.35	1.3165	1.2163	23.63	27.02

Vigna [6] obtained times similar to [13], yet with about twice smaller space overhead. It was the first work on rank/select capable of handling vectors of size up to 2^{64} bits. Focusing on 64-bit architectures also allowed to successfully apply bit-parallel (also called broadword programming) tricks, e.g., for an Elias–Fano representation of monotone sequences, useful for sparsely populated bit-vectors.

Navarro and Providel [14] raised the bar even higher (or, should we say, lower?), reducing the space overhead to about 10% of the original bit-vector on top of the entropy, solving in this space both rank and select. Their rank queries are handled within about 0.4 μsec and select queries within 1 μsec (on a 3.0 GHz Intel Core 2 Duo). They also show how to reuse sampling data between the rank and the select in a non-compressed scenario, with a benefit in space, which allows to answer these queries within around 0.2 μsec, using only 3% of extra space on top of the plain bit-vector. Another noteworthy idea from this work is computing entries of the universal table needed in the Raman et al. [4] solution on the fly, which requires more time to rebuild a desired block, but greatly reduces the space overhead.

A unique approach was taken by Beskers and Fischer [15], who focus on sequences with low *higher-order* entropies. Their solutions are likely to be competitive, e.g., for representing wavelet trees for repetitive collections of strings, like individual genomes of the same species.

3. Our algorithms

In the two following subsections we present our compressed rank and select variants. We describe 32-bit implementations of our solutions (where the bit-vector size is limited to $2^{32} - 1$ bits, i.e., almost 512 MB). We also implemented corresponding 64-bit variants, where the size limit is increased to $2^{64} - 1$ bits, but applying our 64-bit ranks or selects to relatively small bit-vectors is inefficient. For this reason, in the experiments our 64-bit variants are used only for the large datasets.

3.1. Compressed rank

The input bit-vector B is conceptually divided into blocks of k bytes, and each sequence of h successive blocks is grouped in a superblock. For each superblock a fixed-size header is stored, having h ranks of the prefix of B up to the current block and h offsets to the areas storing the successive blocks. Popcounting over a block (or its prefix), with up to $\lceil k/8 \rceil$ 64-bit built-in instructions, is used