Succinct Rank Dictionaries Revisited

Saska Dönges
Department of Computer Science, University of Helsinki, Finland

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Abstract

We study data structures for representing sets of m elements drawn from the universe $[0..n-1]$ that support access and rank queries. A classical approach to this problem, foundational to the fields of succinct and compact data structures, is to represent the set as a bitvector X of n bits, where $X[i] = 1$ iff i is a member of the set. Our particular focus in this paper is on structures taking $\log_2 \binom{n}{m} + o(n)$ bits, which stem from the so-called RRR bitvector scheme (Raman et al., ACM Trans. Alg., 2007). In RRR bitvectors, X is conceptually divided into n/b blocks of b bits each. A block containing c 1 bits is then encoded using $\log_2 b + \log_2 \binom{b}{c}$ bits, where $\log b$ bits are used to encode c, and $\log_2 \binom{b}{c}$ bits are used to say which of the $\binom{b}{c}$ possible combinations the block represents. In all existing RRR implementations the code assigned to a block is its lexicographical rank amongst the $\binom{b}{c}$ combinations of its class. In this paper we explore alternative *non-lexicographical* assignments of codes to blocks. We show these approaches can lead to faster query times and offer relevant space-time trade-offs in practice compared to state-of-the-art implementations (Gog and Petri, Software, Prac. & Exp., 2014) from the Succinct Data Structures Library.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases data structures, data compression, succinct data structures, compressed data structures, weighted de Bruijn sequence, text indexing, string algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.15

Supplementary Material *Software (Source code)*: `https://github.com/saskeli/h0_bv` [3]
archived at `swh:1:dir:383ab8d9247150886cfacd830469e5e5bdd72551`

Acknowledgements We thank Zsuzsanna Lipták, Taku Onodera, and Rajeev Raman for interesting discussions, and the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Bit vectors supporting rank and access queries are essential to the field of compact data structures [15, 11] and, as such, have seen intense research in the past two decades.

We are given a sequence $X[0..n-1]$ of n bits, m of which are set to 1 and should preprocess X so that the following queries can be answered quickly[1]:

$$\text{access}(i) = X[i]$$
$$\text{rank}(i) = \text{number of 1s among the first } i \text{ bits.}$$

Classical methods use $n + o(n)$ bits and leave the input bit string intact, building small structures on top of it to support rank queries in constant time [6, 14]. These methods are extremely fast in practice [5, 8], but incur large space overheads on sparse or otherwise compressible inputs. The Elias-Fano representation, used within web search engines [21, 17]

[1] Another related query is select(i), which returns the position of the ith 1 bit, but it is less used than the other two queries, particularly in the context of text indexing using the backward search algorithm.

and full-text indexes [10] takes $2m + m\log(n/m)$ bits and supports access in $O(1)$ time and rank in $O(\log(n/m))$ time is $o(n)$ extra bits are used[2]. There are also a number of heuristic and hybrid methods that are tuned for certain scenarios, such as full-text indexing [8].

Storing a bitvector of length n with m 1 bits set can be seen as the problem of storing m elements of a finite universe of size n so that membership and rank queries can be answered efficiently in practice. The information theoretic lower bound for storing m elements from a universe of size n is $B = \log \binom{n}{m}$. Raman, Raman, and Rao [19], building on earlier work by Pagh [18], describe a data structure that matches this bound up to an $o(n)$ term and supports membership and rank queries in $O(1)$ time. This data structure is commonly referred to as *RRR* and has since been heavily engineered by a series of authors [2, 16, 5], culminating in its embodiment in the Succinct Data Structures Library (SDSL)[3].

RRR is of particular interest in full-text indexing because when applied to the bitvectors of a wavelet tree constructed over the Burrows-Wheeler transform of a string T of n symbols the resulting structure achieves size $nH_k + o(n)$ bits, where nH_k is kth-order empirical entropy of T [13, 15], a technique called implicit compression boosting [12, 4].

All RRR implementations of which we are aware divide the bitvector X into n/b blocks of b bits each and store the encoded block containing c 1 bits using $\log b + \log \binom{b}{c}$ bits. $\log b$ of the bits are used to encode c, the number of 1 bits the block contains, which is referred to as the block's *class*. The other $\log \binom{b}{c}$ bits are used to encode the block's *offset*, its lexicographical rank amongst all possible $\binom{b}{c}$ combinations of b-length blocks containing c 1 bits (we provide a detailed description in Section 2).

Contribution. In this paper, we revisit the implementation of entropy-compressed bitvectors stemming from the basic RRR scheme. Our contributions are summarized as follows:

1. In order to later decode a block when answering an access or rank query, some RRR implementations [2, 5] use a lookup table of size 2^b that is indexed with the block's class and offset to obtain the original block in $O(1)$ time. We describe two methods for reducing the size of the lookup table: a simple sparsification technique that provides a space-time trade-off, and an approach based on weighted binary de Bruijn sequences from combinatorics on words [20] that reduces the lookup table by a factor of b. Both these approaches allow slightly larger values of b to be used, reducing overall space usage.
2. An alternative to the lookup-table approach that is taken by some RRR implementations [16, 5] is to decode the block directly from its class-offset encoding by using properties of lexicographical offset assignment. These methods decode the original block bit-by-bit in $O(b)$ time. We develop an alternative non-lexicographical scheme for assigning offset values, that proves slightly faster to decode in practice.

Both our new approaches above represent a departure from the lexicographical assignment of offsets that is common to all previous work on the problem.

Roadmap. The remainder of this paper is structured as follows. In the next section, we give a detailed overview of the basic RRR approach common to all previous implementations of it, as well as the vagaries of some specific implementations. We also describe the experimental setup used throughout. Then, in Section 3 we describe methods for reducing the size of the block-decoding table relevant to some RRR implementations. Section 4 describes our

[2] Logarithms thoughout are base 2.
[3] `https://github.com/simongog/sdsl`

new method for inverting combinations, with further engineering measures described in Section 4.1. In Section 5 we apply our new RRR implementations as an encoding option for hybrid bitvector schemes. Section 6 concludes and offers directions for further research.

2 Preliminaries

2.1 RRR bitvector structure

In this section we describe the elements of the RRR data structure that are common to existing implementations. Implementations of RRR [2, 16, 5] generally sacrifice the constant-time queries from the original paper [19] in favor or practicality. The first public implementation of RRR was due to Claude and Navarro [2], and we now describe their approach – all other implementations of which we are aware build on it.

We are given a bitvector X of n bits, m of which are set to 1. We divide X up into n/b blocks of b bits each. We assume here for ease of exposition that b divides n – the situation where this is not the case and the last block has less than b bits is trivial to deal with.

Denote the i^{th} block as x_i. Let the *class*, c_i, of block x_i be the number of 1 bits x_i contains. There are $\binom{b}{c}$ different blocks belonging to class c. Each block corresponds to one of the $\binom{b}{c}$ combinations of its class. We define the *offset*, f_i of block x_i to be an integer in the range $[0..\binom{b}{c})$ that uniquely determines which element of the class (i.e., which combination) the block corresponds to. All previous work on RRR assigns a block's offset as its lexicographical rank among all blocks of the same class. We call this the block's *lexicographical offset*.

Classes are stored in an $\lceil n/b \rceil \cdot \lceil \log(b+1) \rceil$ bit array, C, that trivially supports random access. The offsets are stored in a $\sum_{c_i} \lceil \log \binom{b}{c_i} \rceil$ bit sequence, F. Note that random access to elements of F is not trivial, because they are of variable length. To enable random access to F, pointers to the start of every k^{th} element of the offset sequences are stored[4] in a pointer table of $\lceil \frac{n}{bk} \rceil$ words. To support fast rank queries an additional $\lceil \frac{n}{bk} \rceil$ word table storing cumulative ranks up to every k^{th} block is added.

Now, to support rank(i) we proceed as follows. Position i is contained in block i/b. We scan $C[\frac{i}{kb}..\frac{i}{b})$ and sum the class values $c_{i/(kb)}, \ldots, c_{i/(b-1)}$, adding this to the cumulative rank at $\frac{i}{bk}$ which we have precomputed. This gives the rank up to the start of block $x_{i/b}$. To obtain $f_{i/b}$, the offset for the block, when scanning class values $c_{i/k}$ to $c_{i/b} - 1$ we sum values $\log(\binom{b}{c_{i/kb}})$, $\log(\binom{b}{c_{(i/(kb))+1}})$, …, $\log(\binom{b}{c_{i/b}-1})$, from which we obtain the starting position in F of $f_{i/b}$. As described, a scan of up to k elements of C is required, followed by a single access to F. The class and offset for a block are thus obtained. What remains is to *unrank* the block's offset – i.e., obtain its original bit representation – which allows us to complete the query answer by counting the bits in the relevant prefix in the case of a rank query. It is at this point that implementations begin to differ.

Lookup table decoding. The implementation of Claude and Navarro [2] uses a two-dimensional lookup table of 2^b entries that contains all the possible blocks of size b (an approach prescribed in Raman at al.'s original paper [19]). The ith row of the table contains all the blocks of class i (i.e., all the blocks containing i 1 bits). Having worked out the class c and the offset f of the block containing the query index as described above, the query is completed by accessing cell $[c, f]$ of the table. Claude and Navarro store the blocks in a row in lexicographical order, but the order is not important to the query algorithm just described.

[4] For all of the implementations k is set to 32 (blocks).

The lookup table has total size $b2^b$ bits and indeed it is this size that adds some tension to the data structure: ideally, one would like to increase b in order to decrease the $(\log b)$-bit overhead incurred per block for storing the class values. However increasing b causes a rapid blowup in the table size. In order to keep the lookup table manageable, and to fit c to four bits, Calude and Navarro fix the block size to $b = 15$, resulting in a lookup table $15 \cdot 2^{15}$ bits in size, and an overhead from the class array of $\approx 0.27n$ bits. Gog and Petri engineer this approach further in the SDSL [5] with a class called `rrr_vector_15`.

On-the-fly decoding. Arguing that 25% is a non-trivial overhead in the context of succinct data strucutres, Navarro and Providel [16] show that the lookup table can be eliminated by decoding the bits of a block from its lexicographical offset f in $O(b)$ time, by checking the range of values that offset f belongs to and extracting bits consecutively as follows.

For a block length b, consider all the blocks of class c arranged in lexicographical order. The first $\binom{b-1}{c}$ of them begin with a 0 and the remaining $\binom{b-1}{c-1}$ begin with a 1. Thus, if $f \geq \binom{b-1}{c}$, then the first bit of the block was a 1. In this case we decrement b and c, and decrease f by $\binom{b-1}{c}$. Otherwise, the first bit of the block was a 0. In this case we only decrement b. Now we continue extracting the next bit from a block of length $b-1$. The process can be stopped if f becomes zero, which implies that all the remaining bits are a single run of zeroes or ones. All the binomial coefficients are precomputed in a table in order to make decoding faster, but this is much smaller than the original lookup table. Navarro and Providel show that queries, while slower than the lookup-table approach, are still acceptably fast in practice. The approach was further engineered by Gog and Petri [5] and is included in the SDSL.

2.2 Experimental Setup

We measure and report on experiments with our new methods throughout the paper and so define our experimental machine and test data now. Additionally, our source code is available at `https://github.com/saskeli/h0_bv`.

Data sets. We used the same data sets for all of our experiments. Three of the bitvectors are derived from real-world data, and chosen to be diverse and relevant to practical compressed indexing applications. WT-WEB-1GB[5] and WT-DNA-1GB[6] are data sets by Gog and Petri [5].

bv-dump.bin[7] is the concatenated bit vectors from the "plain matrix" representation of the spectral Burrows-Wheeler transform (SBWT) proposed by Alanko, Puglisi and Vuohtoniemi [1]. The SBWT in question is built from a set of 17,336,887 Illumina HiSeq 2500 reads of length 502 sampled from the human gut (SRAidentifier ERR5035349) in a study on irritable bowel syndrome and bile acid malabsorption [7] with k-mer size 31.

In addition, we generated random bitvectors with varying probability p for a 1-bit. All of these bit vectors are one gigabyte in size, and p is i.i.d.. `RND-1.bin` has $p = 2^{-1}$, `RND-5.bin` has $p = 2^{-5}$ and `RND-10.bin` $p = 2^{-10}$.

Some summary statistics on the datasets are presented in Table 1

[5] Available at `http://people.eng.unimelb.edu.au/sgog/data/WT-WEB-1GB.gz`
[6] Available at `http://people.eng.unimelb.edu.au/sgog/data/WT-DNA-1GB.gz`
[7] Available at `https://doi.org/10.5281/zenodo.11031751`

Table 1 Details of the datasets used in performance benchmarks.

Dataset	n	H_0	mean H_0 entropy of 64-bit blocks	Fraction of uniform 64-bit blocks
`WT-WEB-1GB`	8.66×10^9	0.998	0.0997	0.844
`WT-DNA-1GB`	8.24×10^9	0.980	0.615	0.276
`bv-dump.bin`	8.26×10^9	0.811	0.756	0.007 23
`RND-1.bin`	8.59×10^9	1.00	0.989	0
`RND-5.bin`	8.59×10^9	0.201	0.188	0.131
`RND-10.bin`	8.59×10^9	0.0112	0.007 20	0.939

Experiments. The experimental run is the same for each of the following three sections, with only the data structures under test changing. In particular, data structures are built for each of the data sets, after which 10^7 random *access* and *rank* queries each are run and timed on the built index. The space usage of the built index along with mean query times are reported. We only report times for *access* queries because comparative *rank* performance between data structure implementations remains the same, just very slightly and consistently slower than *access* queries. Full results for *rank* and *select* queries as timed in our experiments, along with construction times, are available in Appendix A.

Machine and Compilation. Tests were run on a machine with an AMD EPYC 7452 Processor and 500GB of DDR4 RAM. The operating system was AlmaLinux 8.10 (with Linux 4.18.0-372.9.1.el8.x86_64 kernel). Code was compiled with GCC 13.3.0 and the performance-relevant flags: `-std=c++2a, -march=native, -Ofast` and `-DNDEBUG`. Experiments were run multiple times, on multiple machines to ensure the stability of the results.

3 Reducing the size of the lookup table

As discussed in Section 2.1, the principle hurdle to reducing the lower order terms in the original RRR representation (and its implementation by Claude and Navarro) is the lookup table that stores all possible bitstrings of length b. This lookup table has size $b2^b$ bits, making it very sensitive to increases in b. Navarro and Providel's approach is to remove this lookup table entirely and reconstruct its entries as needed through computation, at the cost of query time. In this section we keep the lookup table, but examine ways to reduce its size, with the hope of increasing b to reduce overall size and retain fast query times.

The two approaches we explore are sparsification of the original lookup table and essentially compacting the lookup table by replacing it with a series of weighted de Bruijn sequences [20].

Sparse lookup table. Instead of storing all $\binom{b}{c}$ possible length-b bitstrings with weight c in the table, we can store every gth entry ($\lceil\binom{b}{c}/g\rceil$ in blocks total) and reconstruct any desired block in at most g constant-time steps. This changes the total space required for the lookup table to $b2^b/g + b$ bits. Further space savings can be had by storing samples only for blocks where $c \leq \lceil b/2\rceil$, effectively halving the space usage. Our implementations thus use $b2^{b-1}/g + b$ bits for storing the lookup table, and decoding a block given c and f values takes $\mathcal{O}(g)$ time with an algorithm that, given a block, can compute its lexicographic successor in $O(1)$ time.

We use the "Compute the lexicographically next bit permutation"-algorithm from the "Bit Twiddling Hacks" website[8]. See Listing 1. The fast enumeration algorithm compiles to 9 instructions per iteration when optimized and compiled by the GCC compiler.

A natural selection for gap size is $g = b$, leading to the lookup table taking $2^{b-1} + b$ bits.

Listing 1 Code for fast enumeration of the $\binom{b}{c}$ blocks of lenght b with c set bits.

```
constexpr uint32_t next(uint32_t v) {
    uint32_t t = v | (v - 1);
    return (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctzl(v) + 1));
}
```

Weighted de Bruijn sequences. In [20], Ruskey, Sawada and Williams study weighted de Bruijn sequences, circular strings of length $\binom{b}{c}$ from which all possible substrings of length b having c 1 bits can be efficiently extracted. In particular, we create circular bitstrings (the weighted de Bruijn sequences) of length $\binom{b}{c}$, such that all bitstrings of length b and weight c can be created from $(b-1)$-length substrings of this weighted de Bruijn sequence with the necessary bit appended to reach the correct weight.

These remarkable constructions seem tailor made to reduce the size of the RRR lookup table. The class c remains the same as before, but the offset f is the relative position of the encoded block within the weighted de Bruijn sequence.

Storing these bit strings in practice requires $\binom{b}{c} + b - 1$ bits, with the first $b-1$ bits being appended to the end, to ensure that the "last" substring can be read without overhead due to the circularity of the string. Conveniently, if we concatenate these weighted de Bruijn sequences by increasing weight, adjacent sequences overlap by $b-2$ characters, provided each weighted de Bruijn sequence is the lexicographically smallest rotation. See Figure 1. Additionally we only require sequences of weights $0 \leq c \leq \lfloor b/2 \rfloor$, since higher weight sequences can be stored as inverted references to lower weight sequences. This leads to the entire lookup table fitting in

$$b - 1 + \sum_{c=1}^{\lfloor b/2 \rfloor} \binom{b}{c} + 1 < 2^b \text{ bits.}$$

The required binary sequence along with the start positions of each weight are computed using a Python script and output as static arrays for use with C++ code. These arrays are used for decoding blocks. Reading from the binary sequence consists of reading the desired bytes in big-endian, followed by masking and shifting to generate a candidate block. After this the last bit is flipped, if required before flipping the entire candidate if $c > \lfloor b/2 \rfloor$. The decoding process is visualized in Figure 2.

Performance of reduced size lookup tables. We benchmarked both the sparse lookup table and weighted de Bruijn approaches with $b \in \{15, 24\}$ and gaps 1, 7, 15, 24, 32, 64 for the sparse lookup tables. It would have been interesting to test $b = 31$ as well, but unfortunately, our methods for compile-time definition of the tables failed for these bigger tables (probably due to the compiler running out of memory or a compile time limitation in constant expression resolution).

[8] https://graphics.stanford.edu/~seander/bithacks.html

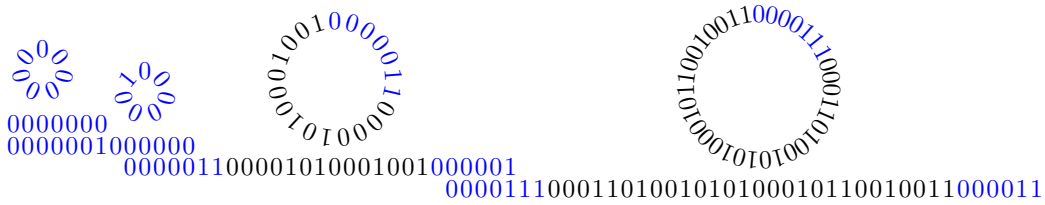

Figure 1 Figure illustrating the concatenation of weighted de Bruijn sequences with $b = 7$, to create the full sequence that replaces the traditional lookup table. The first b symbols of the lexicographically smallest rotation are highlighted, to show the overlap between subsequent sequences with the starts being repeated.

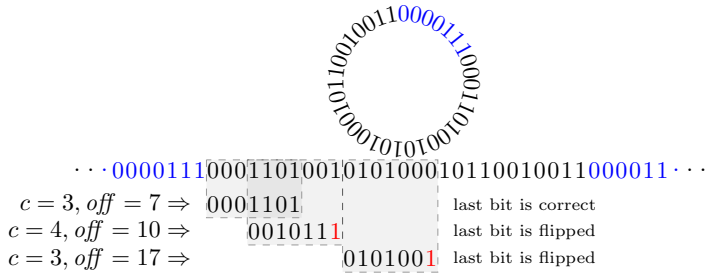

Figure 2 Figure illustrating the process of reading elements from the weighted de Bruijn sequence with $b = 7$ and $c \in \{3, 4\}$. b candidate bits are read from the offset position, the candidate bits get flipped if $c = 4$, after which the correct block is the exclusive `OR` of the candidate and **popcount**(candidate) $\neq c$.

Results for benchmarking can be seen in Figure 3. Our approach allows a new space-time trade-off by increasing compression efficiency by increasing b while reducing lookup speed. The RRR15 implementation by Gog and Petri [5] remains the fastest overall, but compressing worse than our de Brujin sequence-based approach with $b = 24$ (DBS24).

4 Induced total order for b length bitstrings

The specific encoding of the offset is irrelevant as long as the mapping from the b-length bitstring x to the (c, f) pair is bijective and $0 \leq f < \binom{b}{c}$. As we saw in Section 2.1, lexicographic order allows decoding and answering queries in $\mathcal{O}(b)$ time. Selecting another induced order changes the complexity of decoding. The new method we now describe allows answering queries on blocks in $\mathcal{O}(\log c \cdot \log b)$ time.

For any bitstrings x_α and x_β of length b and class c, define the ordering such that $\text{pop}(x_\alpha \gg b/2) < \text{pop}(x_\beta \gg b/2) \Rightarrow f_a < f_b$, where $u \gg v$ denotes the binary representation of u shifted right by v bits (discarding the lower v bits of u) and pop returns the number of 1 bits in an integer (i.e., its population count). That is, the order between blocks is primarily decided by the weight of their prefixes, where a bitstring with more bits in the prefix will have a greater f value. This can then be recursively applied to the prefixes and suffixes for a full recursive definition for calculation of the f values.

Based on this, when decoding any given (c, f), the first half of the block x to decode will contain exactly

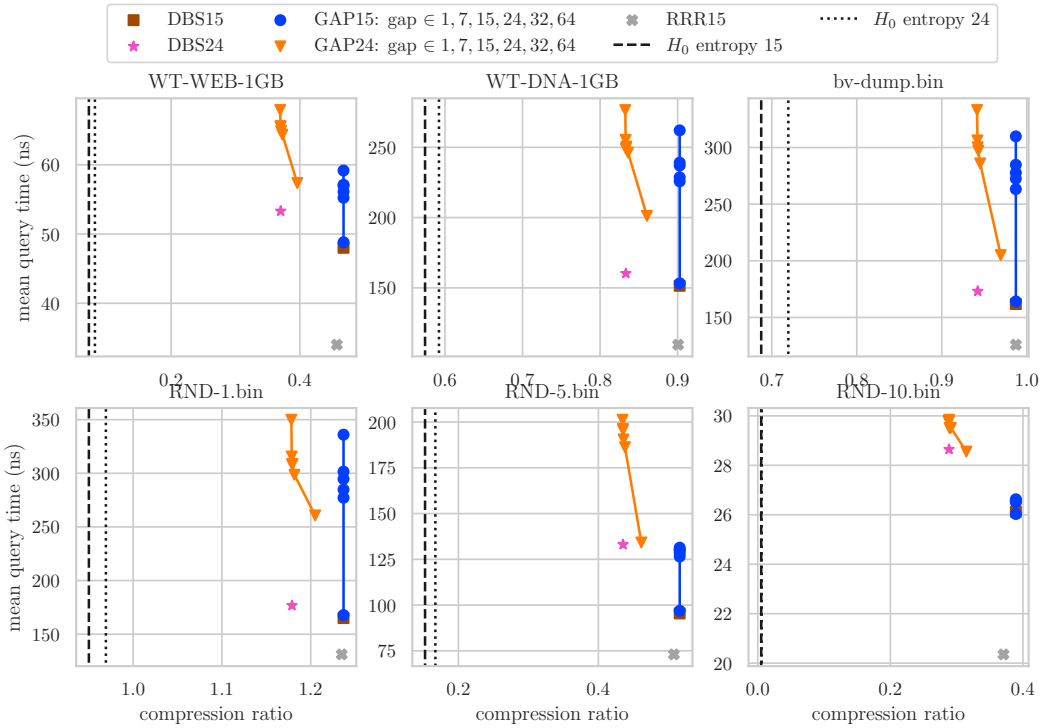

Figure 3 Performance of different lookup table -based approaches with sizes of the lookup tables considered part of the compressed data structure size. The optimized version, based on a full lookup table due to Claude and Navarro [2], and Gog and Petri [5] (RRR15 in the figure), remains the fastest for queries, and beats both our implementations with $b = 15$ in space efficiency as well, this is likely due to some specific optimizations for $b = 15$, that are missing from our implementation and would not apply for $b = 24$. For $b = 24$, our approaches improve compression, offering a space – query performance trade-off. The weighted de Bruijn sequence-based approach (DBS) outperformes the sparsified lookup tables (GAP) in all cases. Mean block wise H_0 entropies are added for reference to provide a theoretical limit for block-based H_0 compression.

$$\operatorname*{argmax}_{c_p \in \mathbb{N}} \left(\sum_{i=0}^{c_p - 1} \binom{b/2}{i} \binom{b/2}{c-i} < f \right)$$

one-bits.

Given c_p, a query targeting the prefix of the block can recurse with

$$\left(c_p, \left(f - \sum_{i=0}^{c_p - 1} \binom{b/2}{i} \binom{b/2}{c-i} \right) / \binom{b/2}{c-c_p} \right)$$

and a query targeting the suffix recurses with

$$\left(c - c_p, \left(f - \sum_{i=0}^{c_p - 1} \binom{b/2}{i} \binom{b/2}{c-i} \right) \mod \binom{b/2}{c-c_p} \right).$$

The recurrence takes $\mathcal{O}(\log b)$ steps, and binary searching for c_p takes $\mathcal{O}(\log c)$ time, given precomputed lookup tables for binomial coefficients. This yields a total run time of $\mathcal{O}(\log c \log b)$ for determining the result of any rank or access query at a block level.

The offset values with the desired properties can be computed using the following algorithm.

Encode, Algorithm for encoding blocks

Input: bitstring x

Output: class-offset pair (c, f)

$\ell \leftarrow x.\text{length}$
$c \leftarrow \sum_{i=1}^{\ell} x[i]$
If $c = \ell \vee c = 0$ **then**
  return $c, 0$
$enc_p \leftarrow \textbf{Encode}(x[1, \ell/2])$
$enc_s \leftarrow \textbf{Encode}(x[\ell/2 + 1, \ell])$
$f \leftarrow \sum_{i=\max(0,c-\ell/2)}^{enc_p.c-1} \binom{\ell/2}{i}\binom{\ell/2}{c-i}$
$f \leftarrow f + enc_p.f \cdot \binom{\ell/2}{enc_s.c}$
$f \leftarrow f + enc_s.f$
return c, f

Example. We illustrate the workings of this alternative encoding with $x = 01101000_b$, and further demonstrate querying with an $access(x, 2)$ query on the encoded block.

The encoded values for the prefix $x_p = 0110_b$ and suffix $x_s = 1000_b$ are computed recursively, but the recurrences are not written out here for brevity.

- There are $\sum_{i=0}^{1} \binom{4}{i}\binom{4}{3-i} = 28$, vectors of class 3 where the prefix has less than two set bits. $f \leftarrow 28$
- $\text{enc}(x - p) = (2, 2)$, meaning there are two instances for $b = 4$ and $c = 2$ that are less than x_p, so we add the number of $b = 8$ and $c = 3$ vectors, with two set bits in the prefix where the prefix encoding is less than $(2, 2)$. I.e $2\binom{4}{1} = 8$. $f \leftarrow f + 8$
- The suffix $\text{enc}(x_s) = (1, 3)$, so we add this value to the ones we derived before. $f \leftarrow f + 3$
- Thus $\text{enc}(0110100_b) = (3, 28 + 8 + 3) = (3, 39)$.

We start with $access(i = 2)$ with $(c, f) = (3, 39)$. When we recurse i and (c, f) will be updated accordingly.

- Since $28 \leq f < 52$, we know that the class of the prefix of x is 2. As we recurse to the left we update (c, f) by subtracing the offset and dividing with the number of length 4 vectors with one set bit. $(c, f) \leftarrow (2, \lfloor \frac{39-28}{4} \rfloor) = (2, 2)$
- Since $1 \leq f < 5$, we know that the class of the new prefix is 1. Now we recurse to the right and update $i \leftarrow i - 2 = 0$ and $(c, f) \leftarrow (1, 2 - 1 \mod 2) = (1, 1)$.
- Since $1 \leq f < 2$, we know that the class of the new prefix is 1. Since we are recursing to the prefix, the only possible update value for $c = 1$ is $(c, f) \leftarrow (1, 0)$.
- We are now accessing a vector of length one with $c = 1$. I.e. for $(c, f) = (1, 0)$ $access(0) = 1$.
- Now for $(c, f) = (3, 39)$, $access(2) = 1$.

4.1 Implementation details

Recursive split. Our implementation of the method described in the previous section makes extensive use of (small) lookup tables. For our implementation with $b = 64$ we precompute $\binom{n}{m}$ for $n \in \{32, 16, 8\}$ and $0 \leq m \leq n$. The binomials are stored as static heap-ordered binary trees to enable fast branchless binary searching for binomial coefficients. In addition, the decoding for block size 8 is done using precomputed lookup tables.

A drawback with our solution is the need to compute division and modulus with binomial coefficients. We were unable to make any compiler precompute the fast integer divisions for the required binomials, and inverse modular arithmetic is impractical because implementations would require computations using 128-bit integers.

Our implementation relies on the symmetry of the recurrence, requiring a block size that is a power of two, while, for better overall data structure size, b should be $2^a - 1$ for some $a \in \mathbb{N}^+$ so that a bits are sufficient to store every possible class value. Unfortunately, asymmetric recurrences would cause additional unavoidable and unpredictable branching while decoding and have a significant negative impact on query performance.

8-bit block iterative approach. The same principle for the induced order can be applied to other splits besides the recursive even split described above. This leads to an $O(b)$ decode time, which matches the on-the-fly approach of Navarro and Providel [16] (and its later implementation by Gog and Petri [5]). The hope is that the reliance on lookup tables for 8-bit sections of the block would yield lower constant factors compared to other implementations.

This approach allows use of block sizes that are not powers of two to improve the storage of class values. If the offsets for the 8-bit sub-blocks correspond to the lexicographic order, the first partial block can be decoded with the same lookup table as the full blocks, with minimal impact from the change in b.

4.2 Results

Results from benchmarking, now including on-the-fly decoding performance, can be seen in Figure 4. The lookup table-based RRR15 remains the fastest overall, with the on-the-fly approaches lagging behind in query performance, but compressing significantly better. Of the on-the-fly approaches, our symmetric recurrence (REC) and block iterative (IT) methods offer similar compression but typically faster lookup speed than the on-the-fly implementation in the SDSL. On all data sets tested the new methods offer relevant space-time tradeoffs.

The space consumed by various components of the data structures is shown in Figure 5. Data structure size is mainly determined by the block and superblock sizes used, and lookup table sizes are irrelevant. Space for the F array increases as block size increases, but this is offset by the shrinking of the C array and superblock overhead.

5 Hybrid implementation

Regions of bitvectors that consist of alternating runs of ones and zeros have very low H_1 entropy but potentially high H_0 entropy. This leads to poor performance for the RRR-based techniques discussed so far, which achieve H_0 entropy, ignoring the $o(n)$ term.

The hybrid bitvector by Kärkkäinen, Kempa and Puglisi [8] is a widely used, practically powerful compressed bitvector scheme that, in particular, is an essential ingredient of state-of-the-art general-purpose FM indexes [4]. The approach is to divide the bitvector into blocks and then select the best encoding for each block from a number of possible encodings. The implementation of hybrid bitvectors in the `sdsl_lite` library uses 256 bit blocks, and then stores each block either as: 1) minority-bit encoded (i.e., to encode the block, we store the positions of the 1 bits using 8 bits each); 2) run-length encoded; or 3) in plain form. This scheme works extremely well for blocks with runs or blocks that are extremely dense or sparse, but can perform badly when this is not the case and many blocks get stored in plain form (resulting in no compression).

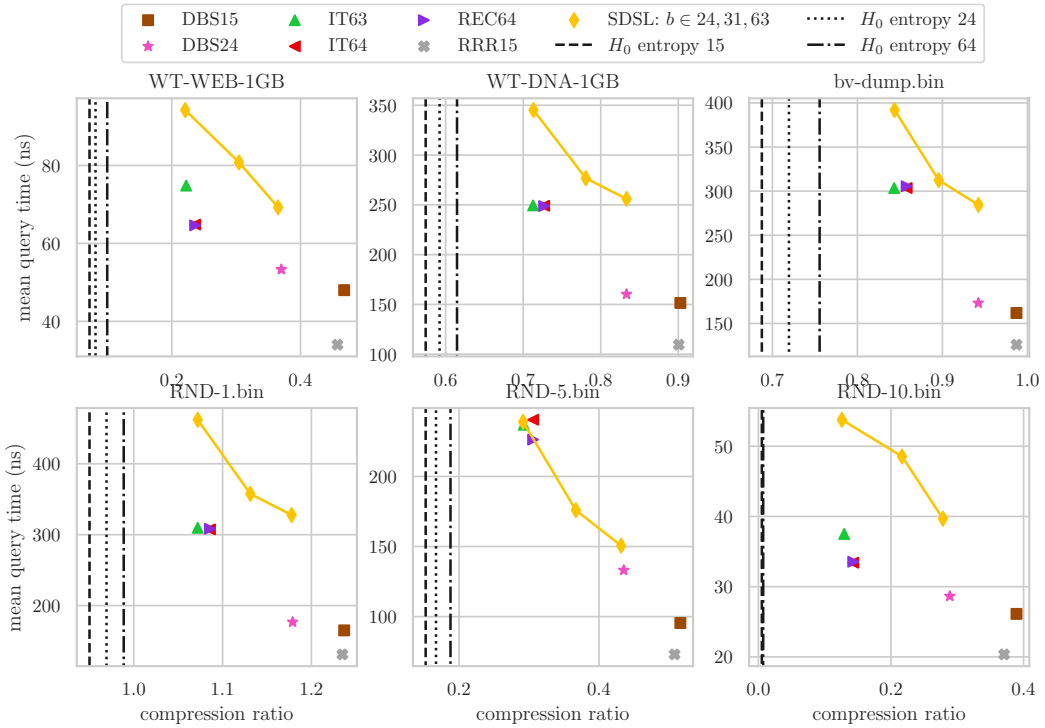

Figure 4 Performance of our H_0-based implementations compared to implementations in the SDSL. Pictured are our de Bruijn sequence-based approach (DBS), our balanced recurrence-based approach (REC) and our block iteration-based implementation (IT), along with the lookup table (RRR) and non-lookup table (SDSL)-based approaches from SDSL, with the number denoting the block size b. The highly optimized RRR15 outperformed all other implementations in query performance, while only beating our weighted de Bruijn implementation with $b = 15$ (DBS15) in compression, due to the small block size. For larger block sizes our implementations compare favorably, being comparable in compression ratio while typically improving query time. Mean block wise H_0 entropies are added for reference to provide a theoretical limit for block-based H_0 compression.

This hybrid scheme and the RRR-based scheme complement each other, and adding a H_0 option for block encoding to the hybrid bitvector should improve compression performance.

With this in mind, we added on our 64-bit block iterative encoding as an option in the 256-bit hybrid bitvector implementation included in the SDSL. As a comparison we had another version that used Gog and Petri's RRR implementation with $b = 256$ as an encoding option instead of our iterative encoding. This was done in a way that did not cause any additional space overhead for the hybrid structure, but should induce some significant overhead to each query, compared to the implementation without support for H_0 encoding.

We ran the same tests on these structures, the results of which are shown in Figure 6. We observe that by including RRR as a block-encoding options, compression levels of hybrid bitvectors can be significantly improved, though at some penalty to query performance.

6 Concluding Remarks

We have described two main improvements to the RRR bitvector data structure, the common theme of which is the non-lexicographical assignment of codes to offset values (in contrast the lexicographical assignment, which has been done in all previous work on the problem).

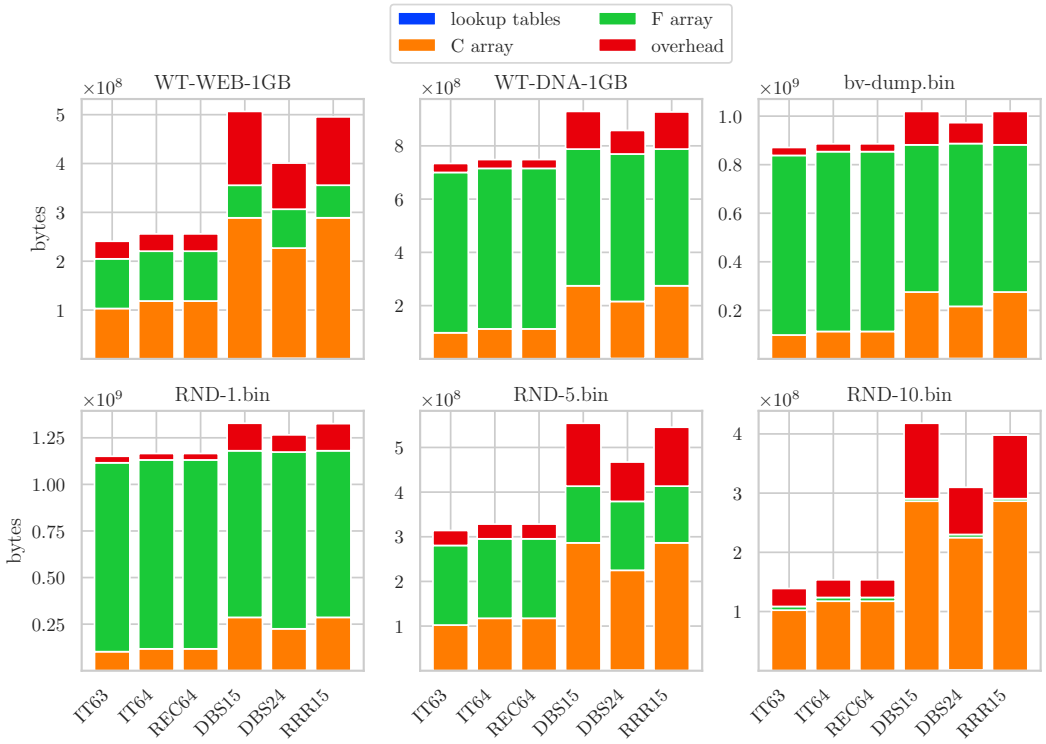

Figure 5 Visualization of the space usage for some of the tested data structures. Lookup table sizes are insignificant. The size of the F array increases with block size, but this increase is offset by the decrease in superblock overhead and C array size.

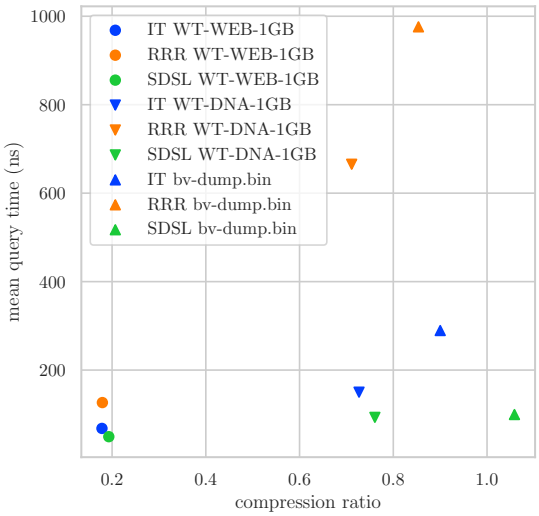

Figure 6 Performance of the SDSL hybrid bitvector, compared to versions with added option for compressing blocks with either our 64-bit block iterative implementation (IT), or the RRR 256-bit compression also from the SDSL. We find that compression can be improved as expected (c.f. blue points and green points), but with a penalty to query performance.

The first of these approaches represents all possible blocks of size b as a series of weighted de Bruijn sequences. The second provides a new on-the-fly block-decoding mechanism for answering queries in $\mathcal{O}(\log(c)\log(b))$ time based on a new induced total order for offset encoding. These methods offer improved practical tradeoffs for H_0-compressed bitvectors. We have also descibed a sparsification technique for reducing the size of the lookup table in the case that lexicographic assignment is used.

There are numerous avenues to further optimize the RRR scheme. Firstly, profiling reveals that our non-lookup table-based implementations execute between three and eight `div` instructions per query. Eliminating these divisions efficiently would improve the query performance of our data structures and possibly be of independent interest. Efficiently optimizing divisions for 64 bit (and larger) integers is an open problem, but perhaps existing schemes for shorter integers[9] could be applied to larger integers as well as hardware keeps improving.

Larger block sizes have the potential to improve the compression ratio further, but implementations require computations using large integers (128, 256 or 512-bit integers), with significantly slower arithmetic operations than 64-bit integers.

Finally, the `rrr_vector_15` class in the SDSL uses bit-parallelism to great effect. Since its publication [5] some 10 years ago, architecture support for bit-parallelism has improved significantly, and we expect it is possible to improve the existing optimizations of `rrr_vector_15` or apply novel optimizations to implementations with $b > 15$ using new SIMD instructions.

We leave these open problems to be the focus of future research.

References

1 Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral Burrows-Wheeler transform. In *SIAM Conference on Applied and Computational Discrete Algorithms, ACDA 2023*, pages 225–236. SIAM, 2023. doi:10.1137/1.9781611977714.20.

2 Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *String Processing and Information Retrieval*, pages 176–187, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-540-89097-3_18.

3 Saska Dönges. saskeli/h0_bv. Software, swhId: swh:1:dir:383ab8d9247150886cfacd830469e5e5bdd72551 (visited on 2025-07-02). URL: https://github.com/saskeli/h0_bv, doi:10.4230/artifacts.23794.

4 Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2019. doi:10.1007/S00453-018-0475-9.

5 Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software: Practice and Experience*, 44(11):1287–1314, 2014. doi:10.1002/spe.2198.

6 Guy Jacobson. Space-efficient static trees and graphs. In *Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

7 Ian B Jeffery, Anubhav Das, Eileen O'Herlihy, Simone Coughlan, Katryna Cisek, Michael Moore, Fintan Bradley, Tom Carty, Meenakshi Pradhan, Chinmay Dwibedi, et al. Differences in fecal microbiomes and metabolomes of people with vs without irritable bowel syndrome and bile acid malabsorption. *Gastroenterology*, 158(4):1016–1028, 2020.

8 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. Data Compression Conference*, pages 302–311. IEEE, 2014. doi:10.1109/DCC.2014.87.

9 Daniel Lemire, Owen Kaser, and Nathan Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *Software: Practice and Experience*, 49(6):953–970, 2019. doi:10.1002/spe.2689.

10 Danyang Ma, Simon J. Puglisi, Rajeev Raman, and Bella Zhukova. On Elias-Fano for rank queries in FM-indexes. In *Proc. 31st Data Compression Conference*, pages 223–232. IEEE, 2021. doi:10.1109/DCC50243.2021.00030.

11 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design: Bioinformatics in the Era of High-Throughput Sequencing (2nd edition)*. Cambridge University Press, 2023. URL: http://www.genome-scale.info/.

12 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. String Processing and Information Retrieval, 14th International Symposium*, LNCS 4726, pages 229–241. Springer, 2007. doi:10.1007/978-3-540-75530-2_21.

13 Giovanni Manzini. An analysis of the burrows-wheeler transform. *J. ACM*, 48(3):407–430, 2001. doi:10.1145/382780.382782.

14 J. Ian Munro. Tables. In *Proc. 16th Foundations of Software Technology and Theoretical Computer Science*, LNCS 1180, pages 37–42. Springer, 1996. doi:10.1007/3-540-62034-6_35.

15 Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016. ISBN 978-1-107-15238-0. 570 pages.

16 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, pages 295–306. Springer, 2012. doi:10.1007/978-3-642-30850-5_26.

17 Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *Proc. 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR*, pages 273–282. ACM, 2014. doi:10.1145/2600428.2609615.

18 Rasmus Pagh. Low redundancy in static dictionaries with O(1) worst case lookup time. In *Proc. Automata, Languages and Programming, 26th International Colloquium, ICALP'99*, LNCS 1644, pages 595–604. Springer, 1999. doi:10.1007/3-540-48523-6_56.

19 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43–es, November 2007. doi:10.1145/1290672.1290680.

20 Frank Ruskey, Joe Sawada, and Aaron Williams. De Bruijn sequences for fixed-weight binary strings. *SIAM J. Discret. Math.*, 26(2):605–617, 2012. doi:10.1137/100808782.

21 Sebastiano Vigna. Quasi-succinct indices. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, pages 83–92. ACM, 2013. doi:10.1145/2433396.2433409.

A Tabulated experiment results

	WT-WEB-1GB	WT-DNA-1GB	bv-dump.bin	RND-1.bin	RND-5.bin	RND-10.bin
GAP	block size: 15, gap: 1					
build time (s)	3.73	4.06	4.56	4.13	3.83	3.58
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	48.8	153	164	168	96.9	26.0
rank (ns)	163	172	172	172	167	154
select (ns)	914	932	972	1290	964	896
GAP	block size: 15, gap: 7					
build time (s)	4.14	4.46	4.78	4.55	4.23	3.99
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	55.2	226	263	277	127	26.5
rank (ns)	175	250	277	288	209	158
select (ns)	917	1010	1070	1410	1010	902
GAP	block size: 15, gap: 15					
build time (s)	3.80	4.13	4.48	4.15	3.87	3.64
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	56.1	229	272	285	128	26.6
rank (ns)	176	257	286	295	212	158
select (ns)	926	1020	1100	1420	1020	919
GAP	block size: 15, gap: 24					
build time (s)	3.79	4.11	4.44	4.15	3.87	3.64
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	57.0	237	278	295	130	26.6
rank (ns)	177	260	292	305	213	158
select (ns)	920	1020	1100	1450	1020	909
GAP	block size: 15, gap: 32					
build time (s)	3.78	4.15	4.48	4.20	3.92	3.63
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	57.1	239	285	302	130	26.6
rank (ns)	174	260	297	311	211	155
select (ns)	918	1020	1110	1450	1020	904
GAP	block size: 15, gap: 64					
build time (s)	3.95	4.19	4.54	4.26	3.89	3.64
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	59.2	262	310	336	131	26.5
rank (ns)	176	283	322	346	212	156
select (ns)	921	1070	1170	1520	1030	906

	WT-WEB-1GB	WT-DNA-1GB	bv-dump.bin	RND-1.bin	RND-5.bin	RND-10.bin
GAP	block size: 24, gap: 1					
build time (s)	3.41	10.8	11.7	17.0	2.97	2.40
bits/bit	0.396	0.860	0.969	1.20	0.461	0.315
access (ns)	57.4	201	205	261	134	28.6
rank (ns)	170	237	214	269	173	154
select (ns)	866	1010	1040	1530	899	837
GAP	block size: 24, gap: 7					
build time (s)	3.65	10.5	13.8	16.3	3.13	2.42
bits/bit	0.373	0.836	0.945	1.18	0.438	0.291
access (ns)	64.3	246	286	299	186	29.5
rank (ns)	179	268	291	308	234	159
select (ns)	879	1030	1090	1540	969	846
GAP	block size: 24, gap: 15					
build time (s)	3.53	10.5	11.5	16.3	3.27	2.55
bits/bit	0.371	0.834	0.943	1.18	0.436	0.289
access (ns)	64.9	251	297	309	191	29.5
rank (ns)	181	272	304	315	237	159
select (ns)	882	1030	1100	1570	984	854
GAP	block size: 24, gap: 24					
build time (s)	3.83	10.7	11.7	17.1	3.31	2.71
bits/bit	0.370	0.833	0.942	1.18	0.435	0.289
access (ns)	65.5	249	300	308	196	29.8
rank (ns)	182	272	308	314	240	158
select (ns)	886	1020	1110	1530	993	865
GAP	block size: 24, gap: 32					
build time (s)	3.82	10.9	11.8	16.7	3.58	2.84
bits/bit	0.370	0.833	0.942	1.18	0.435	0.288
access (ns)	65.6	255	306	316	196	29.8
rank (ns)	181	276	316	323	240	158
select (ns)	889	1040	1120	1550	999	869
GAP	block size: 24, gap: 64					
build time (s)	4.40	11.4	12.3	17.8	3.97	3.42
bits/bit	0.369	0.833	0.942	1.18	0.435	0.288
access (ns)	67.9	276	333	350	201	29.8
rank (ns)	184	296	342	356	245	158
select (ns)	895	1060	1170	1600	1020	865

	WT-WEB-1GB	WT-DNA-1GB	bv-dump.bin	RND-1.bin	RND-5.bin	RND-10.bin
IT	block size: 63					
build time (s)	2.29	3.24	4.00	4.03	3.98	1.90
bits/bit	0.222	0.713	0.843	1.07	0.292	0.129
access (ns)	74.8	249	304	310	237	37.5
rank (ns)	179	262	317	320	271	160
select (ns)	760	876	954	970	878	660
IT	block size: 64					
build time (s)	1.14	2.23	2.94	2.83	2.54	0.785
bits/bit	0.236	0.727	0.858	1.09	0.306	0.143
access (ns)	64.8	249	304	307	240	33.4
rank (ns)	179	265	323	321	272	160
select (ns)	764	888	971	1150	860	647
REC	block size: 64					
build time (s)	2.25	8.69	11.5	13.4	3.62	0.809
bits/bit	0.236	0.727	0.858	1.09	0.306	0.143
access (ns)	64.6	249	306	308	226	33.5
rank (ns)	186	275	320	320	252	161
select (ns)	763	871	957	1110	851	703
DBS	block size: 15					
build time (s)	3.76	4.10	4.45	4.13	3.91	3.62
bits/bit	0.468	0.903	0.987	1.24	0.516	0.389
access (ns)	48.0	151	162	165	95.4	26.1
rank (ns)	167	169	170	170	170	166
select (ns)	909	918	958	1280	960	905
DBS	block size: 24					
build time (s)	3.32	10.2	11.0	15.8	3.03	2.35
bits/bit	0.370	0.833	0.942	1.18	0.435	0.289
access (ns)	53.3	160	173	177	133	28.6
rank (ns)	171	182	181	183	173	167
select (ns)	863	916	941	1390	893	843
RRR	block size: 15					
build time (s)	3.54	4.77	5.77	5.16	6.34	3.11
bits/bit	0.457	0.901	0.987	1.23	0.508	0.371
access (ns)	34.0	110	126	131	73.1	20.4
rank (ns)	105	149	157	158	135	93.3
select (ns)	649	831	951	1280	925	822

	WT-WEB-1GB	WT-DNA-1GB	bv-dump.bin	RND-1.bin	RND-5.bin	RND-10.bin
SDSL	block size: 24					
build time (s)	6.47	24.7	26.2	40.3	10.1	3.15
bits/bit	0.365	0.833	0.942	1.18	0.432	0.278
access (ns)	69.2	256	284	328	151	39.7
rank (ns)	132	269	304	338	201	145
select (ns)	613	921	1080	1520	932	777
SDSL	block size: 31					
build time (s)	5.69	23.6	24.9	39.6	9.27	2.57
bits/bit	0.304	0.781	0.896	1.13	0.367	0.217
access (ns)	80.8	277	312	358	176	48.5
rank (ns)	136	284	326	360	216	154
select (ns)	582	909	1080	1310	916	741
SDSL	block size: 63					
build time (s)	4.32	21.0	22.1	36.4	8.00	1.65
bits/bit	0.221	0.713	0.844	1.07	0.292	0.126
access (ns)	94.2	345	392	462	239	53.8
rank (ns)	148	345	396	456	273	170
select (ns)	507	914	1100	1220	895	648
HYB-IT	block size: 256					
build time (s)	2.51	2.47	3.69	1.46	9.76	2.89
bits/bit	0.178	0.727	0.900	1.08	0.322	0.0859
access (ns)	68.1	150	289	118	172	80.6
rank (ns)	75.9	165	291	134	174	96.8
select (ns)	nan	nan	nan	nan	nan	nan
HYB-RRR	block size: 256					
build time (s)	3.34	25.4	42.8	2.65	22.9	2.66
bits/bit	0.179	0.711	0.854	1.08	0.282	0.0859
access (ns)	127	665	976	143	400	81.7
rank (ns)	134	669	947	161	408	98.2
select (ns)	nan	nan	nan	nan	nan	nan
HYB	block size: 256					
build time (s)	2.26	2.03	1.81	1.28	9.86	2.55
bits/bit	0.193	0.761	1.06	1.08	0.328	0.0859
access (ns)	49.4	93.1	99.5	94.7	146	72.4
rank (ns)	63.7	119	132	129	149	89.5
select (ns)	nan	nan	nan	nan	nan	nan