Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors

Florian Kurpicz
Karlsruhe Institute of Technology, Germany

Abstract

Bit vectors are fundamental building blocks of succinct data structures used in compressed text indices, e.g., in the form of the wavelet trees. Here, two types of queries are of interest: rank and select queries. In practice, the smallest (uncompressed) rank and select data structure cs-poppy has a space overhead of $\approx 3.51\,\%$ [Zhou et al., SEA '13]. Using the same overhead, we present a data structure that can answer queries up to $8\,\%$ (rank) and $16.5\,\%$ (select) faster compared with cs-poppy.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases rank and select, space-efficient, bit vector, succinct data structures

Supplementary Material All implementations presented in this paper and scripts to reproduce our experimental evaluation are available under the GPLv3 license.
Rank and select data structure implementations: https://github.com/pasta-toolbox/bit_vector
Scripts for reproduction of results: https://github.com/pasta-toolbox/bit_vector_experiments

1 Introduction and Related Work

Given a bit vector B of length n and $\alpha \in \{0,1\}$, *rank* and *select* are defined as:

rank: given $i \in [0,n)$, rank returns the number of ones (or zeros) in $\mathsf{B}[0,i]$, i.e.,

$$\mathsf{B}.rank_\alpha(i) = |\{j \in [0,i] \colon \mathsf{B}[j] = \alpha\}|$$

select: given a rank i, select returns the leftmost position where the bit vector contains a one (or zero) with rank i, i.e.,

$$\mathsf{B}.select_\alpha(i) = \min\{j \in [0,n) \colon \mathsf{B}.rank_\alpha(j) = i\}$$

Bit vectors are building blocks of many important compact and succinct data structures like wavelet trees [10] that have applications in many compressed full-text indices (e.g., the FM-index [10] and r-index [11]; we point to the following surveys [6, 9, 17, 18] for more information on wavelet trees), succinct graph representations (e.g., LOUDS [14]), and can also be used as a representation of monotonic sequences of integers (e.g., Elias-Fano coding [7, 8]) that supports predecessor queries. It should be noted that all of the applications mentioned above require rank and/or select queries on bit vectors.

Given a length-n bit vector, it is known how to solve rank and select queries in constant time using only $n+o(n)$ bits of space [5, 14]. Here, the bit vector occupies n bits. The rank and select data structures require only $o(n)$ additional bits. There also exist more precise results, when considering a length-n bit vector containing k ones, focusing on applications where the number of ones is small. Currently, the best known result requires $\lg\binom{n}{k} + \frac{n}{(\lg n/t)^t} + \tilde{O}(n^{3/4})$ bits of space and can answer rank and select queries in $O(t)$ time [23] (by no explicitly storing the bit vector).

Related Work.

In this paper, we focus on practical space-efficient *uncompressed* rank and select data structures that can handle bit vectors of arbitrary size. Prominent implementations can be

© Florian Kurpicz;
licensed under Creative Commons License CC-BY 4.0

found in the popular succinct data structure library (SDSL) [12]. Furthermore, there exist highly tuned select-only data structures by Vigna [25], which currently can answer select queries the fastest while being reasonably space-efficient. The currently most space-efficient rank and select data structure by Zhou et al. [26] requires only 3.51 % additional space. There exists more work on practical space-efficient rank and select data structures for bit vectors that require more space, answer queries slower, and/or can handle only bit vectors up to size 2^{32}, e.g., [13, 15, 19], which we, therefore, do not consider in this paper. There also exists work on *compressed* rank and select data structures, e.g., [1, 2, 3, 24] and on rank and select data structures for *mutable* bit vectors, e.g., [21, 22].

2 Preliminaries

Due to the simplicity of the problem, the notations in this paper are rather simple. We have a *bit vector* of length n, where we can access each bit in constant time. In the following descriptions, we make use of the notion of *blocks*. Here, a block is an interval within the bit vector that *covers* the bits within the interval. Given (a part of) a bit vector, the *population count* or *popcount* is the number of ones within (the part of) the bit vector. Popcount instructions are supported by most modern CPUs for up to 64 bit words. Since their first introduction, rank and select data structures that require sub-linear additional space and can answer rank and/or select queries in constant time have similar structures. In this section, we briefly describe the commonly used designs for rank and select data structures.

Almost all practical rank data structures follow the same layout. First, the bit vector is partitioned into consecutive *basic blocks*, which are the smallest unit for which any information is collected in an index. The rank of bits within a basic block is determined directly on the bit vector. Then, there exists a hierarchy of different (overlaying) blocks of different sizes. For each type of block, there exists an index that stores information about the number of ones. The scope in which the number of ones is considered can differ, e.g., the number of ones in the block or the number of ones up to the beginning of the block from the beginning of either the bit vector or another overlaying block. See Figure 1 for an example. Queries are then answered using the information provided by the blocks and the popcount of the basic block up to the position. Depending on the sizes of the blocks, the pertinent information in the indices can be accessed very efficiently. It should be noted, that it suffices to store information about the number of ones in each block, as $rank_0(i) = i - rank_1(i)$.

Unlike rank data structures, select data structures come in two flavors: *rank*-based and *position*-based. Rank-based select data structures utilize a rank data structure that is enhanced by a small index containing sample positions for every k-th one (or zero). To answer a query, we first determine the closest block using the sampled positions. Then, we have to look at blocks until we have found the basic block that contains the correct bit with the requested rank. The position in the basic block can then easily be computed. While fast in practice (see Section 4), this type of select data structure usually cannot guarantee a constant query time.

Position-based select data structures on the other hand only store sample positions. Usually, they differentiate between different block sizes and densities, e.g., for very sparse blocks, the answer of every select query can be stored directly. One advantage of position-based select data structures is a constant worst-case query time can be achieved, e.g., [25]. However, there is also a disadvantage of position-based select data structures. Unlike when answering rank queries, we cannot use a $select_1$ query to answer a $select_0$ query. The significant difference between rank- and position-based select data structures is that we can

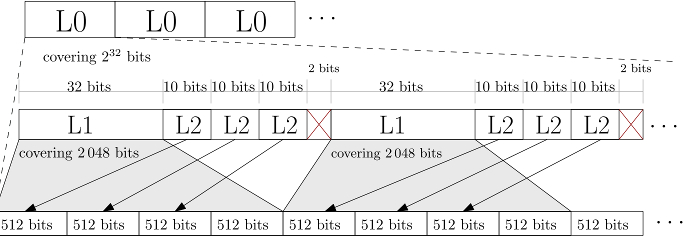

Figure 1 L0-index and interleaved L1- and L2-index of cs-poppy. Arrows indicate that the popcount of the basic block is stored. Wasted bits are marked by red crosses.

easily use the information in a rank-based data structure to answer both $select_0$ and $select_1$ queries. We can simply transform the number of ones (or zeros) up to a position to the number of zeros (or ones) up to that position.[1] The same is not possible with position-based select data structures.

3 Space Efficient Rank and Select Data Structures

First, in Section 3.1, we describe the design of a space efficient rank and select data structure by Zhou et al. [26] named *cs-poppy* that makes use of fast popcount instructions. In Section 3.2, we present our main result—a significantly faster rank and select data structure—that requires the same space as cs-poppy and makes use of SIMD. Finally, in Section 3.3, we present a new and relatively simple rank data structure that provides better rank query times (in practice) by combining some techniques from our main result with a slightly higher space usage.

3.1 CS-Poppy: Rank-Based Rank and Select Data Structure

We describe the rank and select data structure *cs-poppy* top-down, starting with the largest blocks. For an overview, see Figure 1. First, the bit vector is split up into consecutive L0-blocks of size 2^{32} bits. For each L0-block we store the number of ones occurring in the bit vector before the L0-block in an L0-index. To accommodate bit vectors of arbitrary size, each entry in the L0-index requires 64 bits. Note that the indices for the different block types are just plain arrays where the entry of the i-th block is stored at the i-th array entry. Given a position in the bit vector, we can identify all blocks that cover the position by dividing it by the block size (in bits). Next, each L0-block is split up into consecutive L1-blocks of size 2048 bits. This time, we are interested in the number of ones occurring in the L0-block before the L1-block. For each L1-block, we store this information in the L1-index. Since the number of ones in an L0-block can be at most 2^{32}, each entry in the L1-index requires only 32 bits. Finally, each L1-block is split up into four L2-blocks of size 512, i.e., the basic blocks of cs-poppy. We store the number of ones in the L2-blocks for the first three L2-blocks in each L1-block in the L2-index. The number of ones in each L1-block's last L2-block is not stored, as it does not provide any information that cannot be computed using the L1-index. (We can look at the number of one-bits occurring before the next L1-block and subtract the number of one-bits seen before the fourth L2-block.) A L2-index entry has to encode a number in $[0, 512]$ and thus requires 10 bits.

[1] The sampled positions described above have to be stored for ones and zeros.

One important technique used by the authors of cs-poppy is the *interleaved L1- and L2-index*. Here, for each L1-block and the corresponding L2-blocks, a 64-bit word is used to store the entry of the L1- and L2-index. Since the L1-index contains 32-bit words and the L2-index contains 10-bit entries (of which three are pertinent to the L1-block), everything fits into 64 bits. While this approach wastes two bits for each L2-block (0.09 % additional space), it reduces the number of cache misses, as the required part of the L2-index should be loaded whenever the L1-index is accessed. Zhou et al. [26] introduced more practical improvements that can speed up answering rank and select queries using cs-poppy. We refer to their paper for a detailed description.

Answering Rank Queries.

Now, we want to answer a rank query for position i. To this end, we first have to identify the L0- and L1-block the position is covered by. We obtain both blocks by dividing i with the bit size of an L0- and L1-block respectively. The corresponding entries in the L0- and L1-index contain the number of bits occurring from the beginning of the bit vector to the beginning of the L0-block and from the beginning of the L0-block to the beginning of the L1-block. This is the first part of the result. Next, we have to determine the number of ones within the L1-block up to the position. To this end, we scan the entries of the L2-index pertinent to the L1-block until we have reached the L2-block that covers position i. We add entries of the L2-index we have scanned to the result. Finally, we have to determine the number of ones in the final L2-block directly on the bit vector. This can be done using fast popcount instructions. Overall, rank queries can be answered in constant time using cs-poppy.

Answering Select Queries.

To answer a select query for a rank i, we first identify the L0-block where the first position with rank i can occur. This can be done by scanning the L0-index until we see an entry greater or equal to i. The position has to be in the L0-block belonging to the previous entry in the L0-index. Now, we have to scan the L1-index, until we have identified the L1-block that contains the searched position. To speed up select queries, the position of every 8 192-th one is sampled. We can use these samples to skip some parts of the L1-index. When we have identified the correct L1-block, we continue scanning the L2-index until we find the L2-block that contains the position we are looking for. During each step, we subtract the number of ones in the L0-, L1-, and L2-entries from the rank i, because otherwise, we could not identify the block containing the result in the L1- and L2-index, resp. Finally, we have to identify the position within the L2-block and return it. The search in the L2-block has been further optimized by using SIMD by Pandey et al. [20]. Note that this query algorithm requires linear time in the worst-case, but is fast in practice, see Section 4.

3.2 Flat-Popcount: Storing More Information Wasting No Bits

Now, we present the main result of this paper, a rank and select data structure that requires the same space as cs-poppy but is faster in practice, see Section 4. We call this data structure *flat-popcount*. To achieve this, we have to store additional information about blocks without using any additional space. Here, we make use of the two bits that cs-poppy wastes for every L1-block.

As mentioned before, we want to store additional information. To be more precise, we store the number of ones that occur before each L2-block within each L1-block—similar to the L0- and L1-index—see Figure 2 for an example. If we consider cs-poppy, we have to

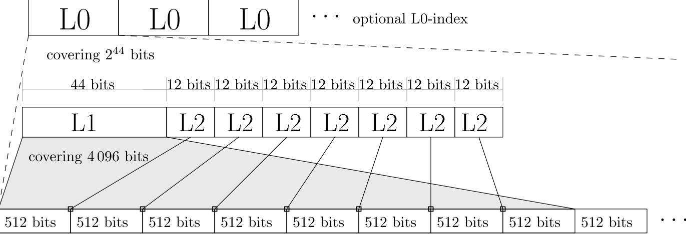

Figure 2 L0-index and interleaved L1- and L2-index of flat-popcount. Boxes indicate that the number of ones in the L1-block up to this point is stored in the L2-index.

store numbers in $[0, 1536]$, which require 11 bits each. Thus, there is not enough space to do so using only 32 bits. Our solution to this problem is to double the size of an L1-block. We still have L2-blocks that cover consecutive 512 bits of the bit vector. Therefore, we now have to consider eight L2-blocks within one L1-block, i.e., each L1-block covers 4 096 bits (compared with 2 048 bits in cs-poppy).

On the other hand, we now have 128 bits to store any information regarding the L1- and eight L2-blocks, i.e., two times 64 bits used in cs-poppy. As before, we do not have to store any information regarding the last L2-block in the L1-block, as we can compute this information using the information stored for the next L1-block. We can therefore store the number of ones up to each L2-block within the L1-block using 12 bits each, as the number of previously set bits is in $[0, 3\,584]$. This requires $7 \cdot 12 = 84$ bits, leaving us with 44 bits for the entry in the L1-index. Similar to cs-poppy, we interleave the L1- and L2-index within one 128-bit word, which is supported by almost all modern CPUs. Not only can we now store more information for each L2-block, we can also increase the size of the L0-blocks to 2^{44} bits. This allows us to make the L0-index optional, as it would be required only for bit vectors of size greater than 2^{44} bits, which is significantly larger than the 2^{32} bits that cs-poppy supports without L0-index and would require 2 TiB space for the bit vector alone.

The space requirements of both cs-poppy and flat-popcount are nearly the same.However, there is one big advantage of our approach: in flat-popcount, we have random access to the L2-index. When using cs-poppy, we have to scan the entries in the L2-index, as they are delta encoded. Using flat-popcount, we store the number of ones occurring to the left of the L2-block within the L1-block *directly*, as for the L1- and L0-index. This allows us to answer both rank queries slightly and select queries significantly more efficiently (in practice). We now take a look at the changes in the query algorithms.

3.2.1 Answering Rank Queries with Flat-Popcount.

Answering rank queries for a position i in this data structure is similar to answering rank queries using cs-poppy, which we describe in Section 3.1. At least for identifying the entries in the L0- and L1-index. Here, nothing has changed, as we only improved on the L2-index. As mentioned earlier, the main advantage of our new design is that we do not have to scan all L2-entries to compute the number of one-bits up the final L2-block. Thus, we can now determine the number of bits occurring in the L1-block before the L2-block that contains the position i accessing only one entry of the L2-index. This entry can be computed the same way we compute the entries of the L0- and L1-index. In our experiments (see Section 4), we can see that scanning up to three L2-entries and adding up their values contributes significantly to the running time of a query.

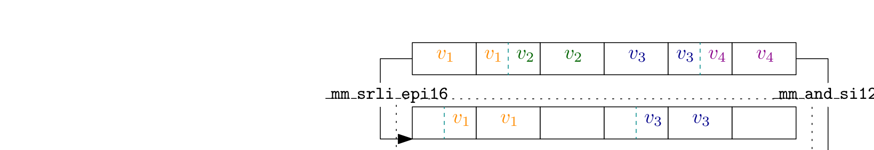

Figure 3 Simplified transformation of four packed 12-bit integers to 16-bit integers. Each block represents 8 bit. The split blocks contain 4 bit of two different entries.

3.2.2 Answering Select Queries with Flat-Popcount.

Answering a select query for a rank i is also very similar to cs-poppy. We still sample the position of every 8 192-th one in the bit vector and use the samples to speed up identifying the L1-block that contains the first position that has rank i. Thus, the first part of the query algorithm that changes is the identification of the L2-block, where the first position with rank i occurs. Due to us storing the number of ones occurring (within the L1-block) before each L2-block, we have a monotonic increasing sequence of integers, allowing us to search for the L2-block more efficiently.

Finding the Correct L2-Entry using Linear or Binary Search.

When we use a linear search, we scan the L2-index until we have found an entry that contains the rank that we are looking for. Therefore, answering queries is not much different than before, when using cs-poppy. However, since we do store the number of ones before the L2-block, we save some additions (of L2-entries) when answering queries, which results in a small improvement. Since the number of ones in the L1-block occurring before each L2-block is a monotonic increasing sequence, we can also use a binary search on the entries of the L2-index. As we store seven entries in the L2-index for each L1-block, we can use a *uniform* binary search [16, p. 414f], which always requires three iterations to identify the correct block. Most importantly, we can pre-compute the whole decision tree and reuse entries of the L2-index. Both of these approaches are rather simple and do not make use of the fact that all entries of the L2-block that we are interested in are contained in a single 128-bit word. Modern CPUs are able to compare multiple values contained in such a word at the same time. However, there are some limitations, which we describe in the following section.

Finding the Correct L2-Entry using SIMD.

In addition to these more obvious approaches described above, we can also search for the L2-block using the *streaming SIMD extension* (SSE), which allows us to divide a 128-bit computer word into consecutive blocks and apply operations on each of the blocks at the same time. One limitation of these instructions is that all blocks must have the same size and that the sizes of the block must be either 8, 16, 32, or 64 bits. Our main goal is to use the compare operation `_mm_cmpgt_epi16` to compare all seven entries of the L2-index that are covered by the L1-block at the same time. Unfortunately, we cannot simply use the compare operation, because we have to store the L2-entries using only 12 bits and there is no compare operation working on 12-bit blocks.

Therefore, our first objective is to transform the seven entries to use 16 bits during the comparison. Of course, we cannot simply store the entries using 16 bits each, as this would

increase the memory requirements significantly. Instead, we have to unpack the 12-bit entries into consecutive 16-bit words, see Figure 3. To this end, we first consider the entries as 8-bit words. Now, three 8-bit words contain two 12-bit entries: the first 8-bit word contains the upper part of the first entry, the second 8-bit word contains the lower part of the first entry and the upper part of the second entry, and the third 8-bit word contains the lower part of the second entry. This pattern repeats for all entries.

We now can shift the words that have their upper part in a whole 8-bit word four bits to the right (`_mm_srli_epi16`) and mask the lower 12 bit of the other entries (`_mm_set1_epi16`) to obtain 16-bit words containing the entries as results. Then, we can take one 16-bit word of each result alternately to obtain our final result, where each 12-bit entry is stored in consecutive 16-bit words. Using this final result, we can compare (`_mm_cmpgt_epi16`) with the remaining rank minus one (because there does not exist a greater-or-equal comparison). As the compare operation does not return the word, where the first match occurs, we have to transform the result to the position of the block. The compare operation returns a word, where the result of the compare operation is marked by all-ones (true) or all-zeros (false). Therefore, we can take the most significant bit of each 8-bit word (`_mm_movemask_eop8`) and apply a simple popcount operation on the result, because we are only interested in the first result where the entry is greater or equal. Now, we can continue the select query as before.

3.3 Wide-Popcount: Faster Rank

As mentioned before, using 16 bits to store an entry of the L2-index would allow us to directly use the SIMD instructions on the L2-index without transforming it first. We do so in our final rank data structure that we call *wide-popcount*. Since we now have 16 bits available for each entry in the L2-index, we also have to make the L1-block bigger, because otherwise, the additional space required by the index would be too much. Therefore, we do now let each L1-block cover 128 L2-blocks, or 65 536 bits. As before, we are only interested in the first 127 L2-blocks within each L1-block. Thus, each entry in the L2-index is in $[0, 65\,024]$ and can be stored using 16 bits. For the L1-index, we use 64-bit words, because then we do not need an L0-index. This increases the required space of the data structure for rank queries to 3.198 % additional space. On the other hand, we have only two levels of indices instead of three. We also do not interleave the L1- and L2-index, as there is no advantage because not all L2-entries can be loaded directly into the cache together with the L1-entry, due to their size and number.

Answering rank and select queries works the same as with flat-popcount (without an L0-index). In Section 4, we will see that this approach works very well for rank queries, but it is not well suited for select queries. This is mostly because we have to search through a lot of L2-entries during a select query. Even when speeding up the search using a uniform binary search or SIMD instructions, answering select queries requires significantly more time than using flat-popcount.

4 Experimental Evaluation

Our implementations are available at https://github.com/pasta-toolbox/bit_vector as open-source (GPLv3, header-only C++-library). In addition to the source code of our implementations, we also provide scripts to easily reproduce all results presented in this paper (https://github.com/pasta-toolbox/bit_vector_experiments). Our experiments were conducted on a workstation equipped with an AMD Ryzen 9 3950X (16 cores with frequencies up to 3.5 GHz, 64 KiB L1d and L1i cache and 512 KiB L2 cache per core, and 4 times 16 MiB

L3 cache) and 64 GiB DDR4 RAM running Ubuntu 20.04.2 LTS. Since our experiments are sequential, only one core was used at a time. We compiled the code using GCC 10.2 with flags `-O3`, `-march=native`, and `-DNDEBUG` and created the makefile using CMake version 3.22.1.

In our experiments, we use two types of random inputs with different densities of ones in the bit vector (10 %, 50 %, and 90 % of all bits are ones). For the first type of input, the ones are uniformly distributed. This type of input should be the easiest one of the two. The second type of input is an adversarial input similar to the one used by Vigna [25]. Assume that k % of the bits in the bit vector should be ones. Then, we set 99 % of the ones in the last k % of the bit vector and the remaining one percent in the first $100 - k$ % of the bit vector. Here, the first part of the bit vector is very sparse while the second part of the bit vector is very dense. Overall, these two types of distribution are the extreme ends of distributions that can occur. All data structures are tested on the same bit vectors and the same queries. The reported running times are the average of three runs (each with a new bit vector and queries). During each run, we constructed the data structure and then asked 100 million queries of each query type supported by the data structure. We compare the following rank and select data structures:

- *cs-poppy* is the space-efficient rank-based rank and select data structure described in Section 3.1 by Zhou et al. [26],
- *cs-poppy-fs* is the same as cs-poppy but with the supposedly faster select algorithm used for the final 64-bit word by Pandey et al. [20],
- *simple-select$_x$* is a position-based select data structure by Vigna [25] that allows for tuning parameter x that determines the size of additional space the data structure is allowed to allocate,
- *simple-select$_h$* is a version of simple-select by Vigna [25] that is highly tuned for bit vectors that contain the same amount of ones and zeros,
- *rank9select* is a rank and select data structure by Vigna [25] that stores 9-bit values to answer rank queries and positions to answer select queries,
- *sdsl-mcl* Clark's select data structure [4] contained in the SDSL [12],
- *sdsl-v* is a simple rank data structure that requires 25 % additional memory and is contained in the SDSL, and
- *sdsl-v5* is a more space-efficient variant of the rank data structure above (only 6.25 % additional space) and also contained in the SDSL

with our implementations that we describe in Sections 3.1–3.3:

- *pasta-poppy* is our implementation of cs-poppy,
- *pasta-flat$_t$* is the rank-based rank and select data structure that we describe in Section 3.2 with $t \in \{$`linear, binary, SIMD`$\}$, and
- *pasta-wide* is the rank-data structure that we describe in Section 3.3.

Unfortunately, two competitors cs-poppy [26] and cs-poppy-fs [20] were not able to compute the select queries on all inputs in a reasonable time (3 hours for all queries on a single bit vector). We were not able to find the error causing this problem, but want to highlight that all queries asked were feasible queries.

We only include *pasta-flat*$_{\texttt{SIMD}}$ in the plots as it is overall the fastest variant. For a comparison of the select query times of the three *pasta-flat* query versions, see Figure 6. The rank query times are identical, as the same data structure and query algorithm is used. Note that we did not include the rank and select data structures that have already been shown to be slower (and to require more additional memory) than the ones included in the experimental evaluation, e.g., *combined-sampling* [19], which is slower than cs-poppy and only

Table 1 Average additional space in percent used by all evaluated data structures on bit vectors of different sizes over the uniform and adversarial distribution.

Name	$n =$	$1 \cdot 10^9$	$2 \cdot 10^9$	$4 \cdot 10^9$	$8 \cdot 10^9$	$16 \cdot 10^9$	$32 \cdot 10^9$
cs-poppy		3.32	3.32	3.32			
cs-poppy-fs		3.32	3.32	3.32			
pasta-poppy		3.58	3.58	3.58	3.58	3.58	3.58
pasta-flat$_{\texttt{SIMD}}$		3.58	3.58	3.58	3.58	3.58	3.58
pasta-wide		10.16	10.17	10.16	10.16	10.16	10.16
sdsl-v		25.00	25.00	25.00	25.00	25.00	25.00
sdsl-v5		6.25	6.25	6.25	6.25	6.25	6.25
sdsl-mcl		18.51	18.52	18.53	18.54	18.55	18.56
simple-select$_0$		8.72	8.72	8.72	8.72	8.72	8.72
simple-select$_1$		9.88	9.88	9.88	9.88	9.88	9.88
simple-select$_2$		12.21	12.20	12.20	12.20	12.20	12.20
simple-select$_3$		16.85	16.85	16.84	16.84	16.84	16.84
simple-select$_h$		15.62	15.63	15.63	15.63	15.63	15.63
rank9select		56.25	56.25	56.25	56.25	56.25	56.25

works for bit vectors up to size 2^{32} bits, *BitRankF* [13], which is slower than simple-select and also requires more space, and the data structures described by Kim et al. [15].

Space Requirements and Select$_0$ & Select$_1$ Queries.

First, we discuss the space requirements of all data structures evaluated in this paper. For an overview, see Table 1. We measured the additional space by overwriting *malloc* to see all allocations on the heap of the different data structures. This is also the reason why it looks like cs-poppy and cs-poppy-fs require less space than pasta-poppy and pasta-flat, because the former allocates memory on the stack to store the samples for the select queries (which we did not modify to not change the results of the running time experiments). This makes our new data structures, cs-poppy, and cs-poppy-fs the most space-efficient rank and select data structures, requiring roughly half the space that the smallest rank- and select-only data structures (sdsl-v5 and simple-select$_0$) require.

Rank Queries.

Let us take a look at all data structures that support rank queries. In Figure 4, we report the average query time on all tested inputs. Here, we can see that sdsl-v and rank9select provide the fastest query times. For large inputs, pasta-wide has query times similar to sdsl-v and rank9select. All these data structures can only answer rank queries and require more space than our new data structures (1.75–6.98 times as much). Both pasta-poppy and pasta-flat have similar query times and get slower for larger inputs. Nevertheless, they are roughly 8 % faster than cs-poppy and cs-poppy-fs.

Select Queries.

We report select query times in Figure 5. Here, we can see that sdsl-mcl, rank9select, and simple-select$_0$ are among the slowest approaches. Depending on the input, either simple-

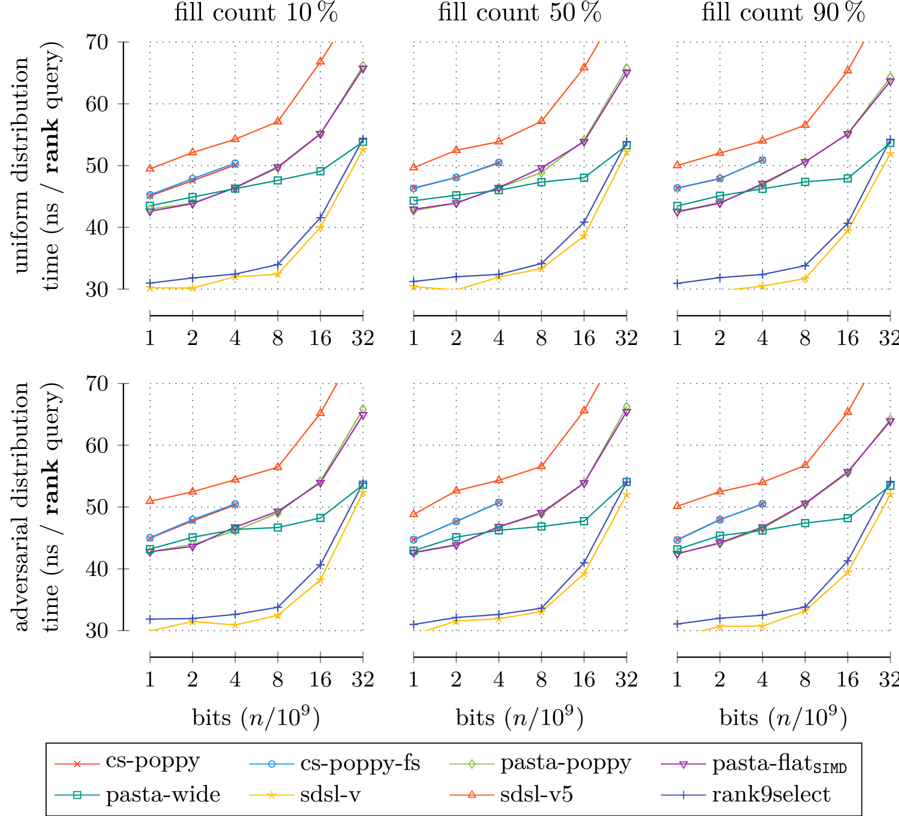

Figure 4 Average rank query time in nanoseconds on all tested inputs.

select$_1$ or simple-select$_2$ is always faster than our new data structures. All other evaluated select data structures are somewhere between pasta-flat and simple-select$_2$ when it comes to select query times. This comes without surprise, simple-select are highly tuned select-only data structures that also use at least 2.43 times as much memory as pasta-flat. However, pasta-flat$_{\texttt{SIMD}}$ is 16.5 % faster than cs-poppy and cs-poppy-fs, making it the fastest and most space-efficient uncompressed rank *and* select data structure.

5 Conclusion

With pasta-flat, we present a space-efficient rank and select data structure that is fast in practice. It requires the same space as the previously most space-efficient rank and select data structure cs-poppy and is between 8 % (rank) and 16.5 % (select) faster than cs-poppy. While there exist faster rank- and select-only data structures, they require significantly more memory and cannot easily answer both $select_0$ and $select_1$ queries, a necessity for many applications, e.g., wavelet trees [10] or succinct tree representations [14]. Pasta-flat can answer both (with a slowdown of up to 1.5 for one of the queries) without requiring additional memory.

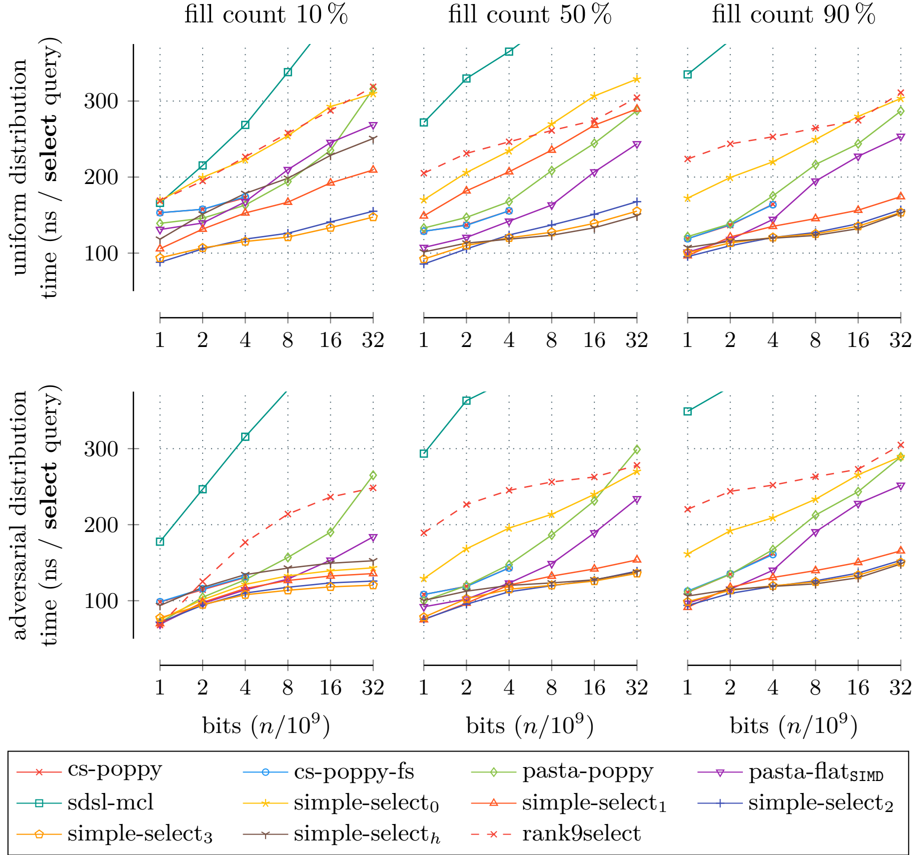

Figure 5 Average select query time in nanoseconds on all tested inputs.

Acknowledgements.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500).

References

1 Diego Arroyuelo and Manuel Weitzman. A hybrid compressed data structure supporting rank and select on bit sequences. In *SCCC*, pages 1–8. IEEE, 2020. doi:10.1109/SCCC51225.2020.9281244.

2 Kai Beskers and Johannes Fischer. High-order entropy compressed bit vectors with rank/select. *Algorithms*, 7(4):608–620, 2014. doi:10.3390/a7040608.

3 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Trans. Algorithms*, 2022. doi:10.1145/3524060.

4 David R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1997.

5 David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *SODA*, pages 383–391. ACM/SIAM, 1996.
6 Patrick Dinklage, Jonas Ellert, Johannes Fischer, Florian Kurpicz, and Marvin Löbel. Practical wavelet tree construction. *ACM J. Exp. Algorithmics*, 26:1.8:1–1.8:67, 2021. doi:10.1145/3457197.
7 Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974. doi:10.1145/321812.321820.
8 Robert Mario Fano. On the number of bits required to implement an associative memory, 1971.
9 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009. doi:10.1016/j.ic.2008.12.010.
10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.
11 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.
12 Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *SEA*, volume 8504 of *Lecture Notes in Computer Science*, pages 326–337. Springer, 2014. doi:10.1007/978-3-319-07959-2_28.
13 Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *WEA*, pages 27–38, 2005.
14 Guy Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.
15 Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2005. doi:10.1007/11427186_28.
16 Donald Ervin Knuth. *The Art of Computer Programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998.
17 Christos Makris. Wavelet trees: A survey. *Comput. Sci. Inf. Syst.*, 9(2):585–625, 2012. doi:10.2298/CSIS110606004M.
18 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.
19 Gonzalo Navarro and Eliana Providel. Fast, small, simple rank/select on bitmaps. In *SEA*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2012. doi:10.1007/978-3-642-30850-5_26.
20 Prashant Pandey, Michael A. Bender, and Rob Johnson. A fast x86 implementation of select. *CoRR*, abs/1706.00990, 2017.
21 Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Inf. Syst.*, 99:101756, 2021. doi:10.1016/j.is.2021.101756.
22 Nicola Prezza. A framework of dynamic data structures for string processing. In *SEA*, volume 75 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.11.
23 Mihai Pătraşçu. Succincter. In *FOCS*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.
24 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, 2007. doi:10.1145/1290672.1290680.
25 Sebastiano Vigna. Broadword implementation of rank/select queries. In *WEA*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008. doi:10.1007/978-3-540-68552-4_12.
26 Dong Zhou, David G. Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *SEA*, volume 7933 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 2013. doi:10.1007/978-3-642-38527-8_15.

A Additional Experimental Results

A.1 Comparison of Our Implementations Only

For better readability, we only show our fastest select algorithm in the main part of this paper. Here, in Figure 6, we compare our implementations with each other.

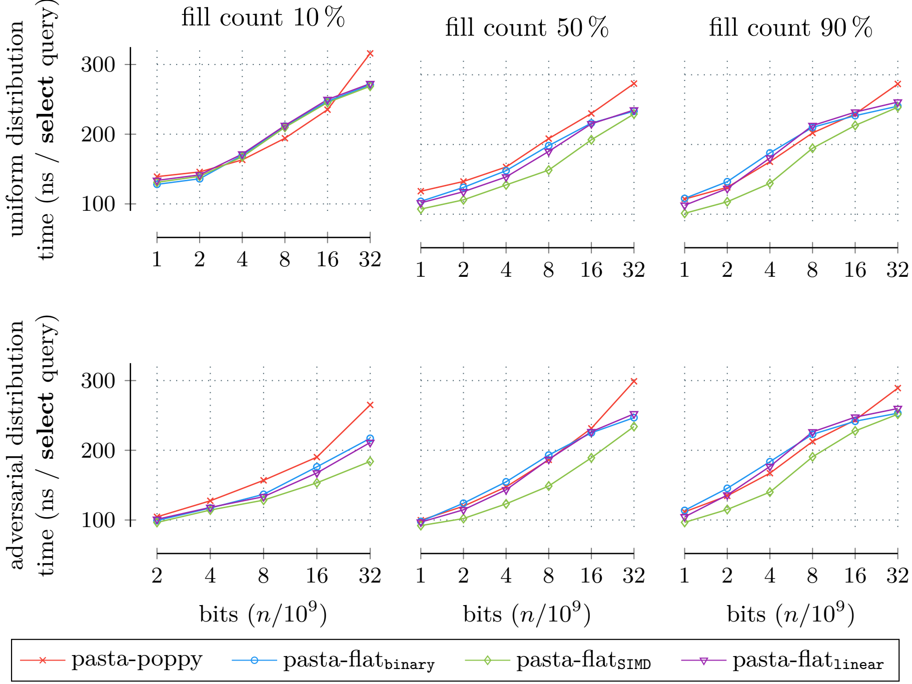

Figure 6 Average select query time on all tested inputs of our implementations.

A.2 Construction Times

Finally, we want to report the construction times for the different data structures, see Figure 7. While the query times of the data structures are definitely more important, as we usually ask multiple queries but construct the data structure only once, we discovered that there are huge differences. Overall, pasta-poppy, pasta-flat$_{\texttt{SIMD}}$, and pasta-wide are the fastest to construct, followed by the data structures contained in the SDSL. All other data structures (all variants of simple-select and cs-poppy) are orders of magnitude slower to construct. The difference in construction time is so big, that we can answer more than 2 000 000 select queries using pasta-flat$_{\texttt{SIMD}}$ and simple-select$_2$ requires the same amount of time when we include the construction time.

A.3 Answering Both Queries

Note that our implementations are the only ones that can answer $select_0$ and $select_1$ without requiring twice the memory.[2] Since all other select data structures in this evaluation are

[2] Per design, every rank-based select data structure can do so.

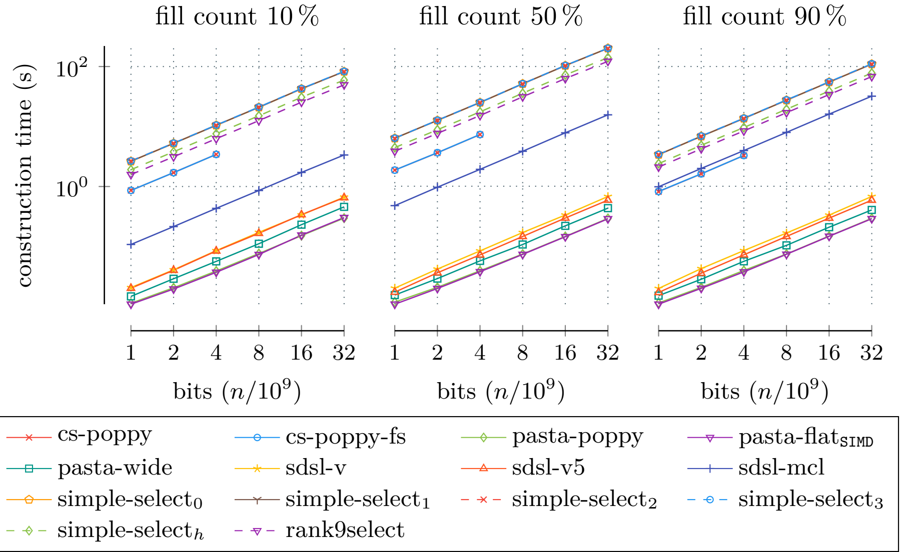

Figure 7 Average construction time in seconds over all inputs.

Table 2 Slowdown of $select_0$ queries compared with $select_1$ queries, when the ranks of ones are stored for each block. Note that our implementations also support optimized $select_0$ queries.

Name	slowdown (uniform)	slowdown (adversarial)
pasta-poppy	1.632	1.559
pasta-flat$_{\texttt{binary}}$	**1.526**	**1.402**
pasta-flat$_{\texttt{SIMD}}$	1.817	1.813
pasta-flat$_{\texttt{linear}}$	1.651	1.604

position-based, this is also the only data structure that can do so without increasing the space required by the data structure. In Table 2, we can see the slowdown of $select_0$ queries compared with $select_1$ queries, when the data structure is optimized for $select_1$ queries. Surprisingly, the binary search approach has the least slowdown. Here, we have a trade-off: we can either double the required memory to answer both types of select queries or we can use a data structure that answers one of the queries 1.5 times slower.