



Partial Rust Implementation of a Compact Data Structures for Rank and Select Queries on Bit Vectors

L. Nasarek (2434739)  

Karlsruhe Institute of Technology, Karlsruhe, Germany

1 What we implemented

Succinct data structures are asymptotically at the information theoretical minimum of their possible space usage. Unlike compressed data structures this enables efficient processing of queries. The two queries in interest for a bit vector of length n are:

■ $\text{rank}_\alpha(i)$ returns the count of α s before the i -th position

■ $\text{select}_\alpha(\text{nth})$ returns the position of the α -nth in the bit vector.

where $\alpha \in \{0, 1\}$, $0 < i < n$ and $\text{nth} = \min\{i | \text{rank}(i) = \text{nth}\}$. Intuitively, rank and select are the inverse of each other. Our data structure implements the indices of [1] and supports the following lemma: Given an in-memory bit vector of length n the data structure can be computed in $O(n)$ with a size of $o(n)$ and answers rank and select queries in $O(1)$. We are especially interested in cases where $n > 2^{30}$ (e.g several GBs) as in such cases simply counting bits is unfeasable.

It should be noted, that the data structure is not optimal in a theoretical sense as it has an asymptotic space overhead of 3.125%, but this should not be a practical concern on modern computer systems.

2 How and why we implemented the data structure

Care must be taken to store the bit vector itself efficiently. Storing, for example an 8 bit number for every bit would waste the upper 7 bits. Even storing a boolean value for each bit is just as wasteful in Rust as each boolean is represented with 8 bit. It is therefore necessary to manually shift single bits into bytes and store those. This can be a source of many problems and subliminal bugs. To work around those we used the bitvec crate. This library implements functionality of the standard library for an efficiently stored bit vector and therefore allows idiomatic access while not wasting space.

2.1 Naïve approach

As computing the queries on the fly is infeasible, the bit vector is preprocessed and partitioned into blocks. For each block the count of ones occurring before each block is saved. Therefore, the maximum possible bit vector length is limited by the size of the data type used to store the amount of ones in the entire bit vector. The size of a single block is additionally constrained by the data type of each block: e.g. storing a 64 bit sum for every 64 bits of the vector would double the amount of space necessary and is hardly *succinct*. We want our blocks to cover a large amount of bits. But additional computation is then necessary: After a block lookup the bits covered by the block still need to be accounted for. If the amount of bits covered is too large, this will be slow due many additional memory lookups of the bit vector itself. These two constraints are at odds with each other.

2.2 Cache-centric design

To speed up processing of queries and minimize the space overhead we take into account today's computer architecture. In general the performance of a large data structure is mainly limited by cache misses.

To minimize cache misses, we implemented a three-level data structure. The L0 cache entries are 64 bits in size, with each L0 entry accounting for 2^{32} bits. Since each L0 covers a substantial amount of bits, the L0 data structure will always fit into the CPU cache as the amount of L0 indices will be small. Counting the bits in an area covered by an L0 index directly is infeasible, so we use two additional indices.

To further minimize cache misses, we use an interleaved L1 and L2 index. Each L1 index is a 32-bit counter, covering 2048 bits. The L1 index stores the count of ones from the beginning of the L0 block to the current L1 block. Each L2 index is 10 bits in size and counts the ones in a 512-bit block inside a L1 block. We only need to store three out of four L2 blocks with each L1 block as the last L2 block can be computed. We interleave the L1 and L2 indexes and store them in a single 64-bit number wasting only 2 bits in the process.

This design has a space overhead of 3.125% or 64 bits per L1 block. Due to the small number of L0 indices saved, they can be disregarded.

2.3 Answering rank queries

First the indices in which the queried position resides are computed by dividing the position with the appropriate block size. The L0 index and L1 index are added with the cumulative sum of the L2 indices which come before the position. The bits inside the queried position's L2 block are computed with bit vector data.

2.4 Answering select queries

To minimize space usage, we reuse the rank data structure for select queries. Initially, we perform a linear search to identify the appropriate L0 block, followed by a binary search to determine the L1 index. For the L2 index, we sequentially examine each block from the first to the last. The remaining bits are counted initially in 64-bit chunks and subsequently on a bit-by-bit basis.

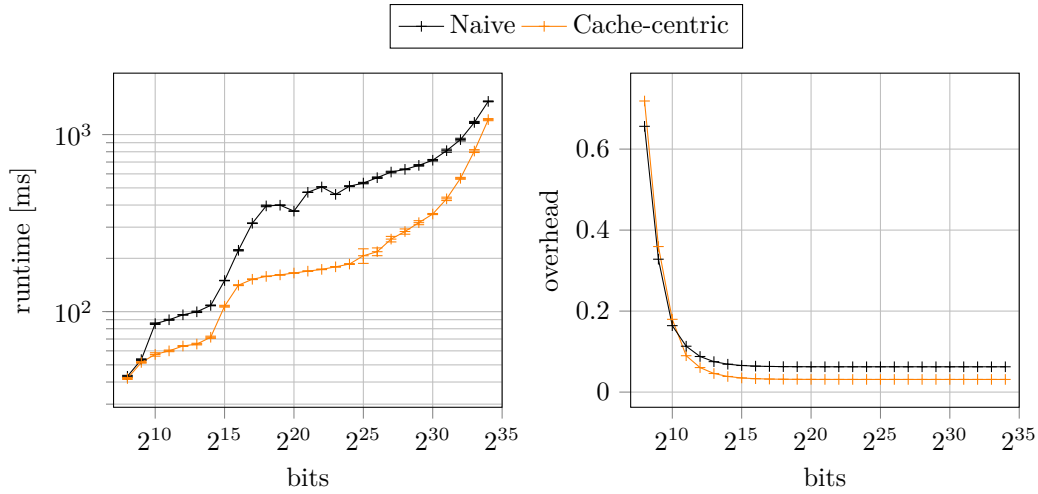
The original data structure presented in [1] employs additional sampling to speed up select queries. However, we encountered difficulties implementing sampling correctly, resulting in a significant slowdown (50% of the entire query processing time). We were unable to identify the bug causing this issue.

3 Evaluation

All benchmarks were run on a dedicated vCPU Server (Hetzner CCX33) with 8 Cores, 32 GB RAM and 240 GB SSD running on AMD Milan EPYC™ 7003 (only 16MB cache) running Ubuntu 24.04 LTS. For the naive design we used a block size of 1024 bits.

3.1 Different implementations

We compare the cache-centric with the naive implementation. Ignoring the cache effects occurring up to 2^{17} bits, the cache-centric implementation is on average 300 ms faster for all queries combined as can be seen in Figure 1. Furthermore, we can observe that the cache-centric is not punished as much as the naive implementation for exceeding the CPU's cache



■ **Figure 1** Average runtime for 1 Million random queries with the naive and cache-centric design.

size of 16MB. While the speed difference might not be overwhelming the naive implementation uses two times the space to answer the same queries with worse performance. We also observe the non-optimal space overhead in both cases, but lower with the cache-centric design. For small bit vectors, both designs result in significant overhead, as some metadata still needs to be computed and saved.

References

- 1 Dong Zhou, David G Andersen, and Michael Kaminsky. Space-efficient, high-performance rank and select structures on uncompressed bit sequences. In *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*, pages 151–163. Springer, 2013.

A Code

- The code can be found at <https://github.com/notourist/runaway-datastructures/>.
- The bit vector generator can be found at https://github.com/paulheg/kit_advanced_data_structures.