Karlsruhe Institute of Technology

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

29th International Symposium on String Processing and Information Retrieval

Florian Kurpicz

Bit Vectors and Applications

Karlsruhe Institute of Technology

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

- bit vector is a text over the alphabet $\{0, 1\}$
- in practice space is very important
  - 64 bits are stored in one 64-bit word
  - don't use `std::vector<bool>`

- Elias-Fano coding
  - compact representation of sorted sequences
  - predecessor and successor support
- succinct tree representations
  - represent trees with n nodes in $2n$ bits
  - navigation in trees with additional $o(n)$ bits
- wavelet trees
  - rank and select support for arbitrary alphabets
  - building block for compressed text indices
- block trees
  - wavelet tree alternative that is better compressible
- ...

Bit Vectors and Applications

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

- bit vector is a text over the alphabet $\{0, 1\}$
- in practice space is very important
  - 64 bits are stored in one 64-bit word
  - don't use std::vector<bool>

- applications require *rank* and *select* support

- Elias-Fano coding
  - compact representation of sorted sequences
  - predecessor and successor support
- succinct tree representations
  - represent trees with n nodes in $2n$ bits
  - navigation in trees with additional $o(n)$ bits
- wavelet trees
  - rank and select support for arbitrary alphabets
  - building block for compressed text indices
- block trees
  - wavelet tree alternative that is better compressible
- ...

Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

$\text{select}_\alpha(j)$ position of j-th α

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

$\text{select}_\alpha(j)$ position of j-th α

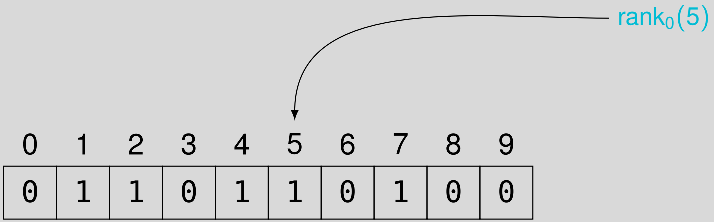
$\text{rank}_0(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

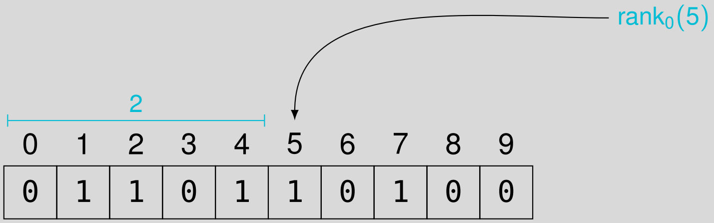
$\text{select}_\alpha(j)$ position of j-th α



Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

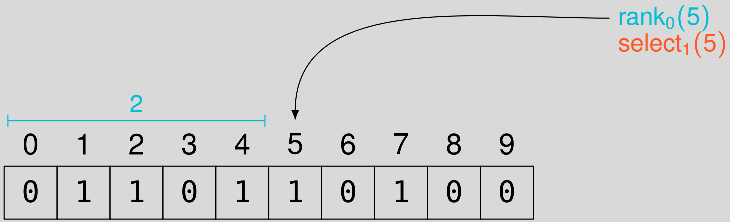
$\text{select}_\alpha(j)$ position of j-th α



Rank and Select Queries

$\mathrm{rank}_\alpha(i)$ # of αs before position i

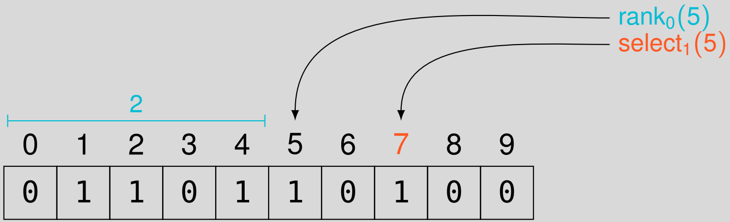
$\mathrm{select}_\alpha(j)$ position of j-th α



Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

$\text{select}_\alpha(j)$ position of j-th α

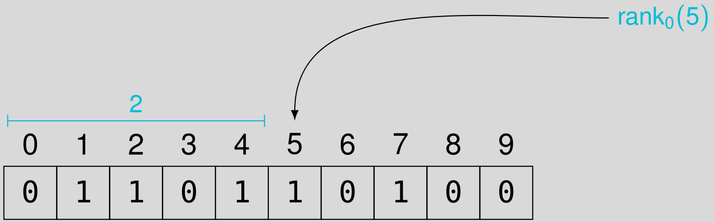


Rank and Select Queries

Karlsruhe Institute of Technology

$\text{rank}_\alpha(i)$ # of αs before position i

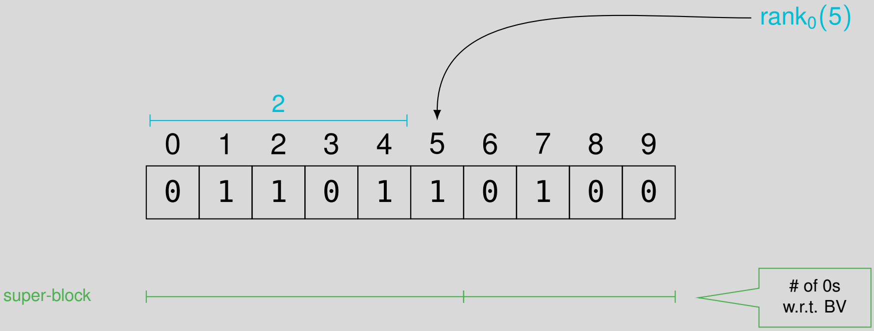
$\text{select}_\alpha(j)$ position of j-th α



Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

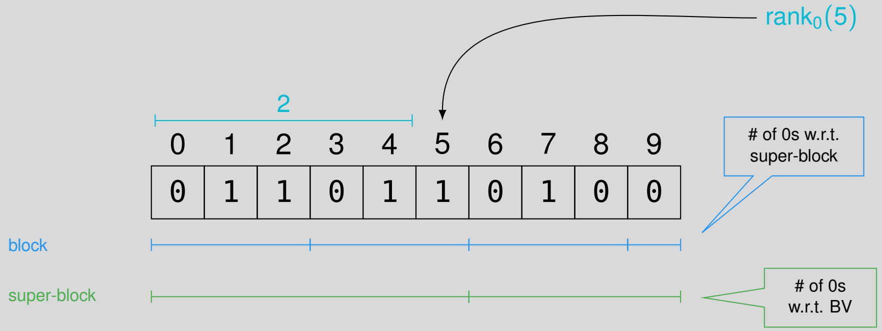
$\text{select}_\alpha(j)$ position of j-th α



Rank and Select Queries

$\text{rank}_\alpha(i)$ # of αs before position i

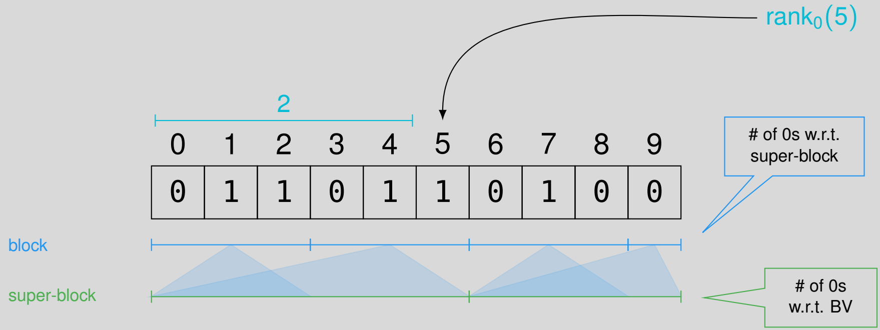
$\text{select}_\alpha(j)$ position of j-th α



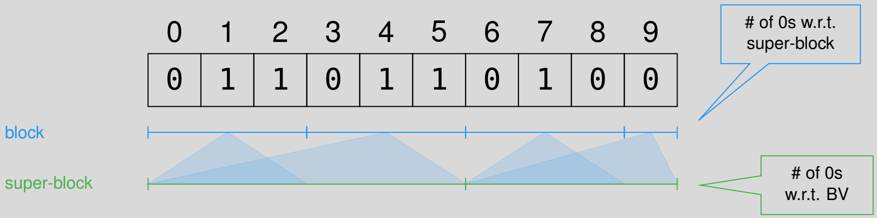
Rank and Select Queries

$\mathrm{rank}_\alpha(i)$ # of αs before position i

$\mathrm{select}_\alpha(j)$ position of j-th α



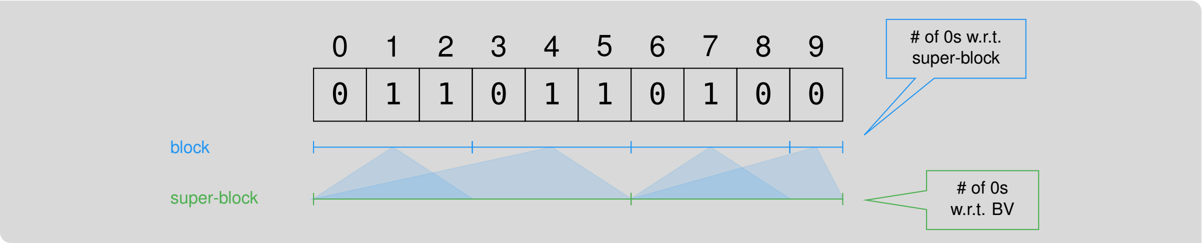
Rank and Select Data Structures



Block-Based Rank

- store number of 1s for (super-)blocks
- query: sum up values in (super-)blocks for position and scan bit vector in block

Rank and Select Data Structures



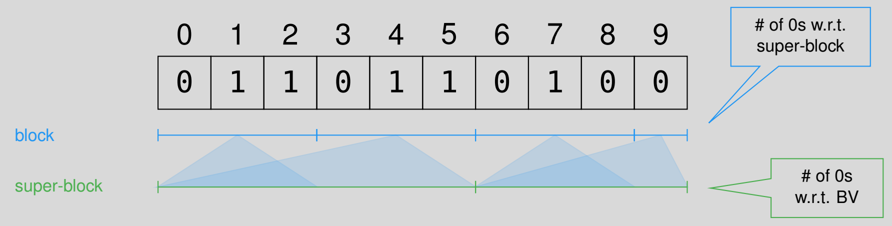
Block-Based Rank

- store number of 1s for (super-)blocks
- query: sum up values in (super-)blocks for position and scan bit vector in block

Block-Based Select

- same as block-based rank
- query: identify block and scan bit vector in block

Rank and Select Data Structures



Block-Based Rank

- store number of 1s for (super-)blocks
- query: sum up values in (super-)blocks for position and scan bit vector in block

Block-Based Select

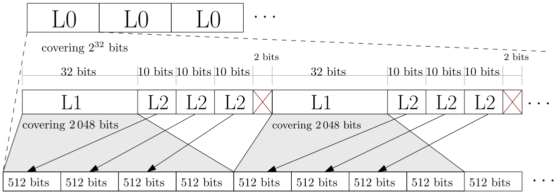
- same as block-based rank
- query: identify block and scan bit vector in block

Sample-Based Select

- store sampled positions for
- query: jump to sample and scan bit vector

Current State of the Art w.r.t. Space [ZAK13]

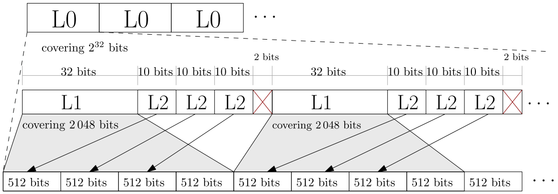
- super-block (L0): 2^{32} bits
- block (L1): 2048 bits
- sub-block (L2): 512 bits



Current State of the Art w.r.t. Space [ZAK13]

- super-block (L0): 2^{32} bits
- block (L1): 2048 bits
- sub-block (L2): 512 bits

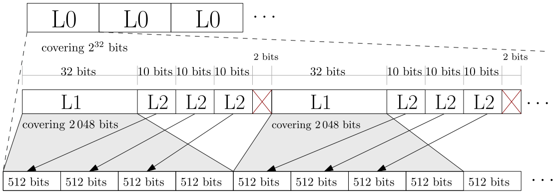
- sub-blocks store popcount
- last sub-block in block not needed
  use next block instead



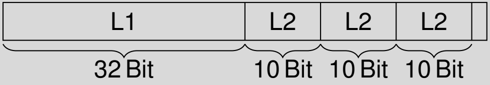
Current State of the Art w.r.t. Space [ZAK13]

- super-block (L0): 2^{32} bits
- block (L1): 2048 bits
- sub-block (L2): 512 bits

- sub-blocks store popcount
- last sub-block in block not needed
  - use next block instead



L1+L2 together 64 bits

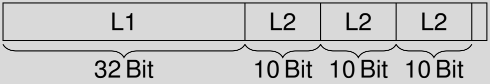


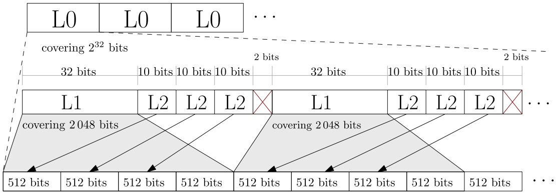
Current State of the Art w.r.t. Space [ZAK13]

- super-block (L0): 2^{32} bits
- block (L1): 2048 bits
- sub-block (L2): 512 bits

- sub-blocks store popcount
- last sub-block in block not needed
  - ℹ use next block instead

L1+L2 together 64 bits

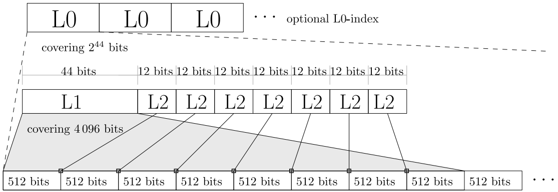




- finding sub-block by scanning
- wasting two bits for every 2048 bits in the bit vector

Stop Wasting Space: Flat Rank Support

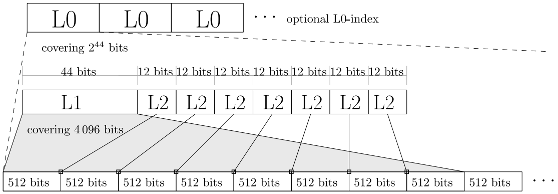
- super-block (L0): 2^{44} bits
- block (L1): 4096 bits
- sub-block (L2): 512 bits



Stop Wasting Space: Flat Rank Support

- super-block (L0): 2^{44} bits
- block (L1): 4096 bits
- sub-block (L2): 512 bits

- sub-blocks store number of 1 w.r.t. block
- last sub-block is not needed
  - use next block instead

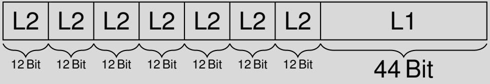


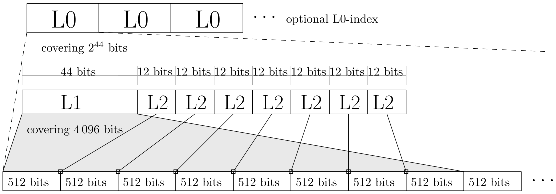
Stop Wasting Space: Flat Rank Support

- super-block (L0): 2^{44} bits
- block (L1): 4096 bits
- sub-block (L2): 512 bits

- sub-blocks store number of 1 w.r.t. block
- last sub-block is not needed
  - use next block instead

L1+L2 together in 128 bits



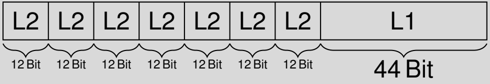


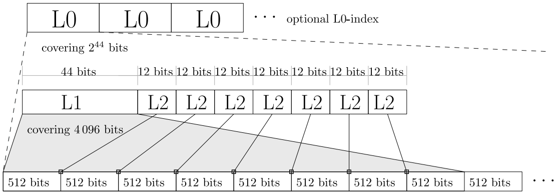
Stop Wasting Space: Flat Rank Support

- super-block (L0): 2^{44} bits
- block (L1): 4096 bits
- sub-block (L2): 512 bits

- sub-blocks store number of 1 w.r.t. block
- last sub-block is not needed
  - use next block instead

L1+L2 together in 128 bits

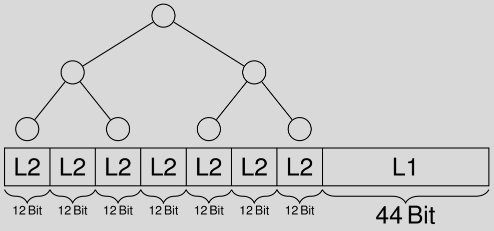




- finding sub-block by scanning,
- uniform binary search, or
- SIMD

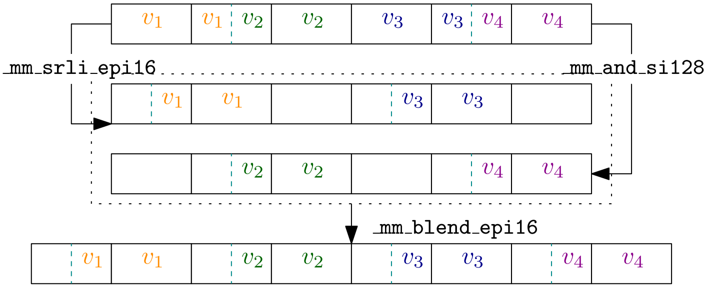
Faster Queries on L1/L2 Blocks: Uniform Binary Search

L1+L2 together in 128 bits



- all searches on array of same length
- same behavior for every search
- same number of comparisons for every sub-block

Faster Queries on L1/L2 Blocks: SIMD



- 12 bits per sub-block
- two sub-blocks share a byte
- either four MSBs or LSBs in shared byte

- finally every sub-block is contained in 16 bits
- fits in 128 bits
- identify sub-block using SIMD
  - _mm_cmpgt_epi16

Wide Rank (and Select) Support

- use 16 bits for each sub-block
- even faster access to sub-blocks
- more sub-blocks per block

- faster rank queries
- very slow select queries

Experimental Evaluation

- AMD Ryzen 9 3950X (3.5 GHz)
- Ubuntu 20.04.2 LTS
- GCC 10.2 (`-O3`, `-march=native`, and `-DNDEBUG`)

- bit vectors filled uniformly at random
- adversarial distribution (99 % of the k % set bits are set in the last k % of the bit vector)

Experimental Evaluation

Karlsruhe Institute of Technology

- AMD Ryzen 9 3950X (3.5 GHz)
- Ubuntu 20.04.2 LTS
- GCC 10.2 (`-O3`, `-march=native`, and `-DNDEBUG`)

- bit vectors filled uniformly at random
- adversarial distribution (99 % of the k % set bits are set in the last k % of the bit vector)

- same bit vector for all data structures
- different bit vector for each run
- random queries are precomputed for each run
- 100 million queries
- average of three runs

Experimental Evaluation

Karlsruhe Institute of Technology

- AMD Ryzen 9 3950X (3.5 GHz)
- Ubuntu 20.04.2 LTS
- GCC 10.2 (`-O3`, `-march=native`, and `-DNDEBUG`)

- bit vectors filled uniformly at random
- adversarial distribution (99 % of the *k* % set bits are set in the last *k* % of the bit vector)

- same bit vector for all data structures
- different bit vector for each run
- random queries are precomputed for each run
- 100 million queries
- average of three runs

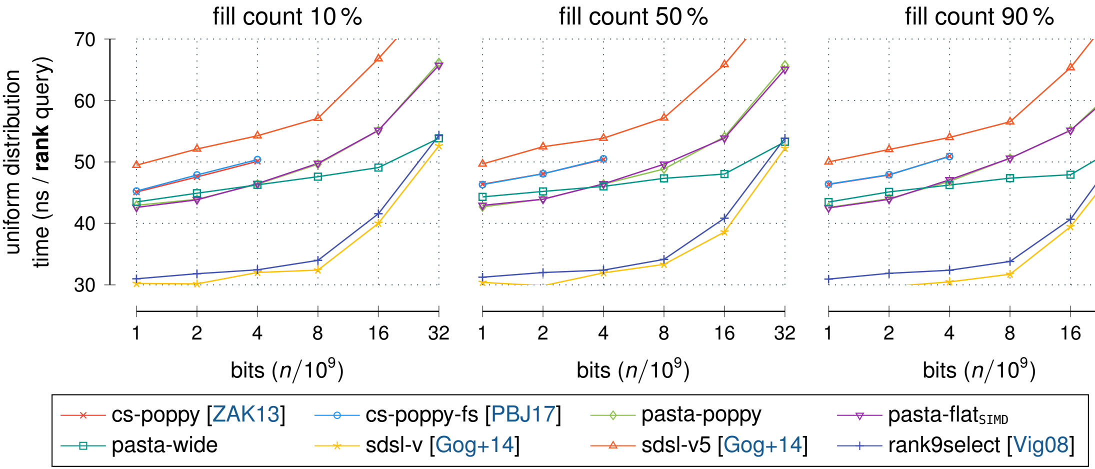
- reproducibility artifacts available
- https://github.com/pasta-toolbox/bit_vector_experiments

Space Overhead in Percent

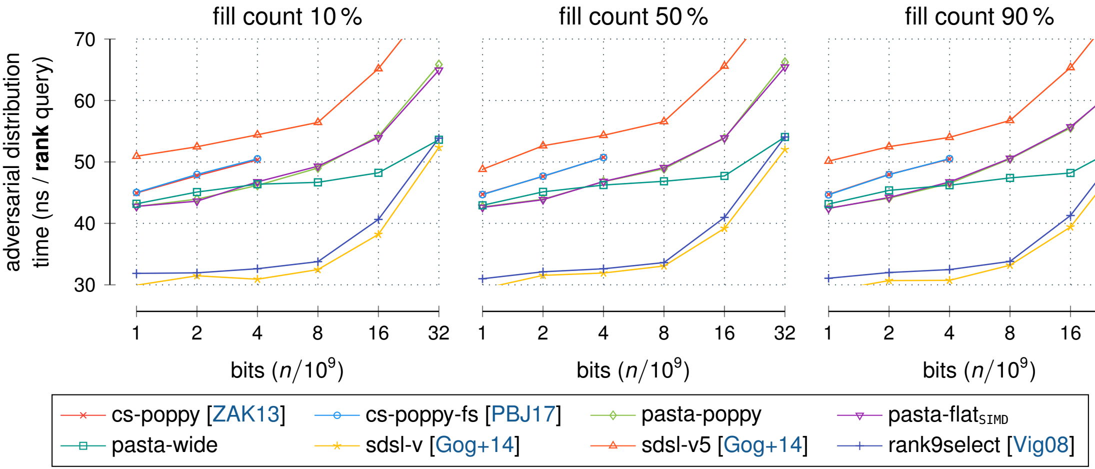
Karlsruhe Institute of Technology

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 [ZAK13]	3.32	3.32	3.32			
cs-poppy-fs [PBJ17]	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
rank9select [Vig08]	56.25	56.25	56.25	56.25	56.25	56.25
sdsl-v [Gog+14]	25.00	25.00	25.00	25.00	25.00	25.00
sdsl-v5 [Gog+14]	6.25	6.25	6.25	6.25	6.25	6.25
sdsl-mcl [Gog+14]	18.51	18.52	18.53	18.54	18.55	18.56
simple-select$_0$ [Vig08]	8.72	8.72	8.72	8.72	8.72	8.72
simple-select$_1$ [Vig08]	9.88	9.88	9.88	9.88	9.88	9.88
simple-select$_2$ [Vig08]	12.21	12.20	12.20	12.20	12.20	12.20
simple-select$_3$ [Vig08]	16.85	16.85	16.84	16.84	16.84	16.84
simple-select$_h$ [Vig08]	15.62	15.63	15.63	15.63	15.63	15.63

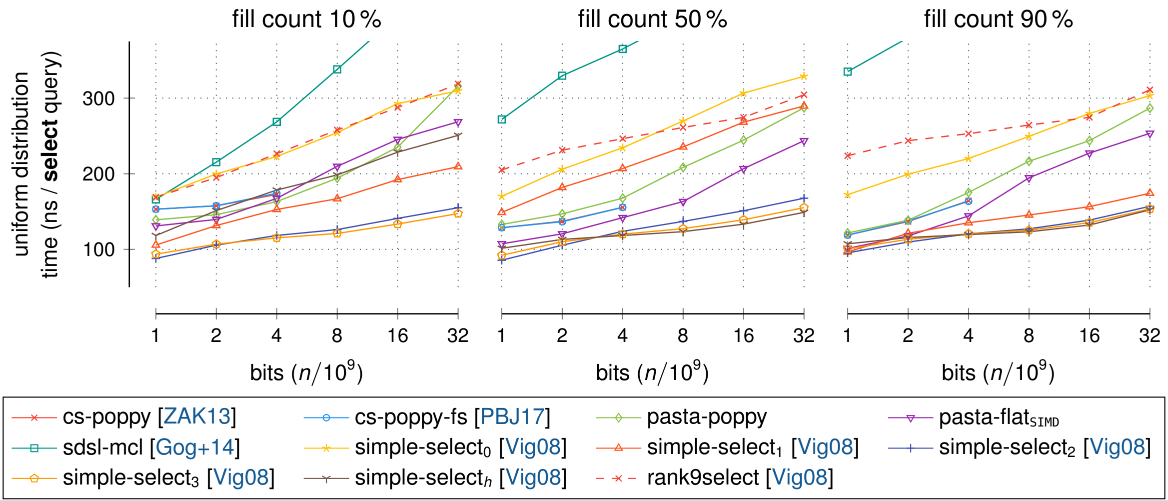
Rank Queries (Uniform Distribution)



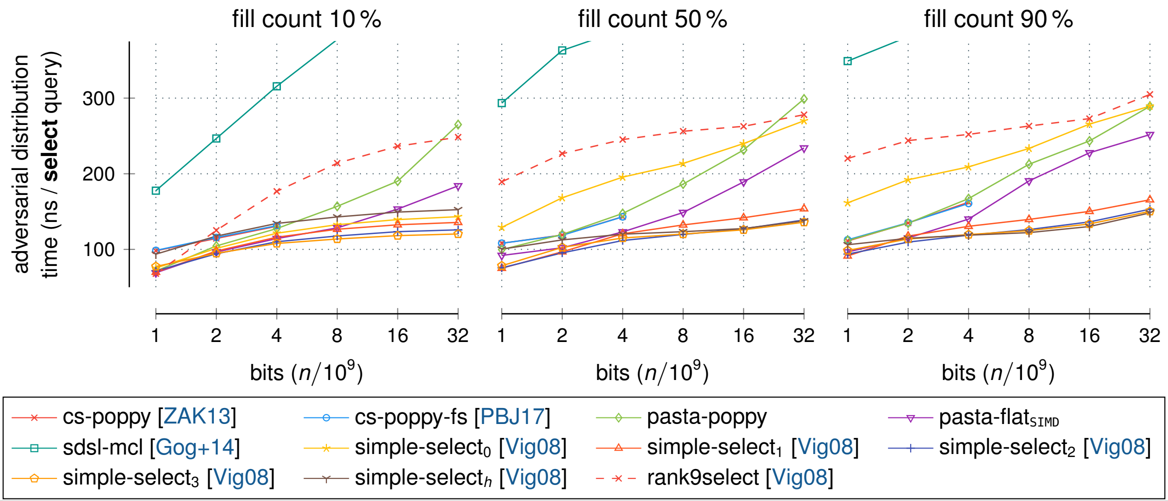
Rank Queries (Adversarial Distribution)



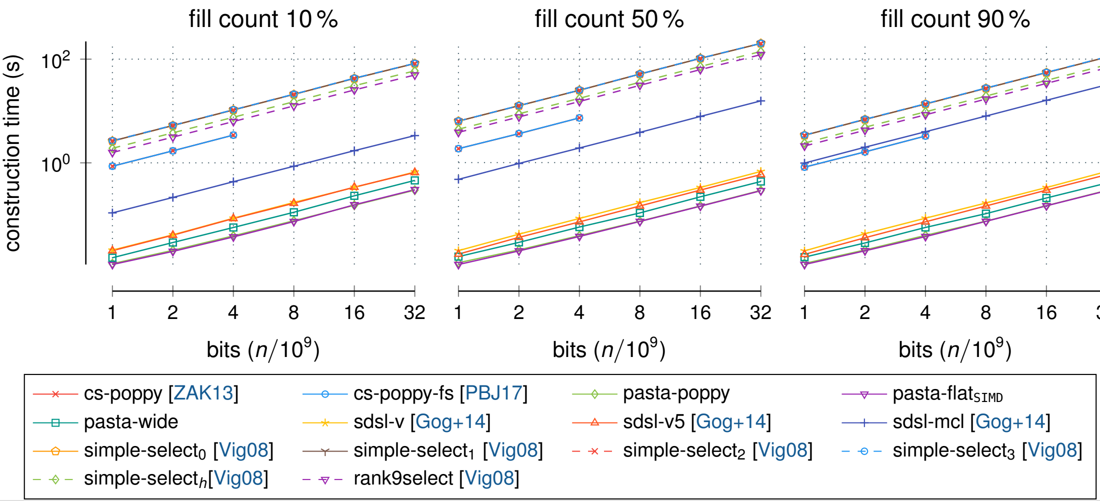
Select Queries (Uniform Distribution)



Select Queries (Adversarial Distribution)



Construction Time



Easy to Use Interface

```
pasta::BitVector bv(1000, 0);
for (size_t i = 0; i < bv.size(); ++i) {
 if (i % 2 == 0) { bv[i] = 1; }
}
for (auto it = bv.begin(); it != bv.end(); ++it) {
 std::cout << ((*it == true) ? '1' : '0');
}
std::cout << std::endl;

pasta::RankSelect rs(bv);
std::cout << rs.rank0(250) << ", " << rs.rank1(250) << std::endl;
std::cout << rs.select0(250) << ", " << rs.rank1(250) << std::endl;
```

Conclusion

Karlsruhe Institute of Technology

- compact rank and select data structure
- SIMD useful for encoding small integers in computer words
- very fast construction
- select_0 and select_1 queries w/o doubling space

- code available under GPLv3 license
- `https://github.com/pasta-toolbox/bit_vector`
- easy to use header-only library

- future work: compress bit vector

European Research Council
Established by the European Commission

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).

Bibliography

[Gog+14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *SEA*. Volume 8504. Lecture Notes in Computer Science. Springer, 2014, pages 326–337. DOI: 10.1007/978-3-319-07959-2_28.

[PBJ17] Prashant Pandey, Michael A. Bender, and Rob Johnson. "A Fast x86 Implementation of Select". In: *CoRR* abs/1706.00990 (2017).

[Vig08] Sebastiano Vigna. "Broadword Implementation of Rank/Select Queries". In: *WEA*. Volume 5038. Lecture Notes in Computer Science. Springer, 2008, pages 154–168. DOI: 10.1007/978-3-540-68552-4_12.

[ZAK13] Dong Zhou, David G. Andersen, and Michael Kaminsky. "Space-Efficient, High-Performance Rank and Select Structures on Uncompressed Bit Sequences". In: *SEA*. Volume 7933. Lecture Notes in Computer Science. Springer, 2013, pages 151–163. DOI: 10.1007/978-3-642-38527-8_15.