Ordered Set Problems

Giulio Ermanno Pibiri

giulio.pibiri@di.unipi.it
http://pages.di.unipi.it/pibiri

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

07/06/2019

The Static Ordered Set Problem

Given a set of n items and an *order relation* defined on them, we are asked to design a data structure that supports **Access**, **Contains**, **Successor**, **Predecessor** efficiently.

The Static Ordered Set Problem

Given a set of n items and an *order relation* defined on them, we are asked to design a data structure that supports **Access**, **Contains**, **Successor**, **Predecessor** efficiently.

Let us assume our items are integers drawn from some universe of size $u \geq n$.

The Static Ordered Set Problem

Given a set of n items and an *order relation* defined on them,
we are asked to design a data structure that supports
Access, **Contains**, **Successor**, **Predecessor** efficiently.

Let us assume our items are integers drawn from some universe of size $u \geq n$.

If the integers are **not to be compressed**:
use an **array**.
Operations are made efficient
by *binary search with loop unrolling*
with cut-off to SSE/AVX (SIMD) *linear search*
on small segments.

If the keys are **uniformly distributed**,
interpolation search can help:
O(log log n) time *with high probability*.

The Static Ordered Set Problem

Given a set of *n* items and an *order relation* defined on them,
we are asked to design a data structure that supports
Access, **Contains**, **Successor**, **Predecessor** efficiently.

Let us assume our items are integers drawn from some universe of size *u* ≥ *n*.

If the integers are **not to be compressed**:
use an **array**.
Operations are made efficient
by *binary search with loop unrolling*
with cut-off to SSE/AVX (SIMD) *linear search*
on small segments.

If the keys are **uniformly distributed**,
interpolation search can help:
O(log log *n*) time *with high probability*.

Let us also assume *n* is so big that we must compress the set.

Sorted integer sets are *ubiquitous*

Inverted indexes

Databases

E-Commerce

Graph compression

Semantic data

Geospatial data

The Static *Compressed* Ordered Set Problem

Large research corpora describing different **space/time** trade-offs.

- Elias' Gamma and Delta
- Elias-Fano
- Variable-Byte Family
- Binary Interpolative Coding
- Simple Family
- PForDelta
- QMX
- Quasi-Succinct
- Partitioned Elias-Fano
- Clustered Elias-Fano
- Optimal Variable-Byte
- DINT

~1970

↓

2019

+ set intersection, union and decode

Partitioning by Cardinality

The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

Partitioning by Cardinality

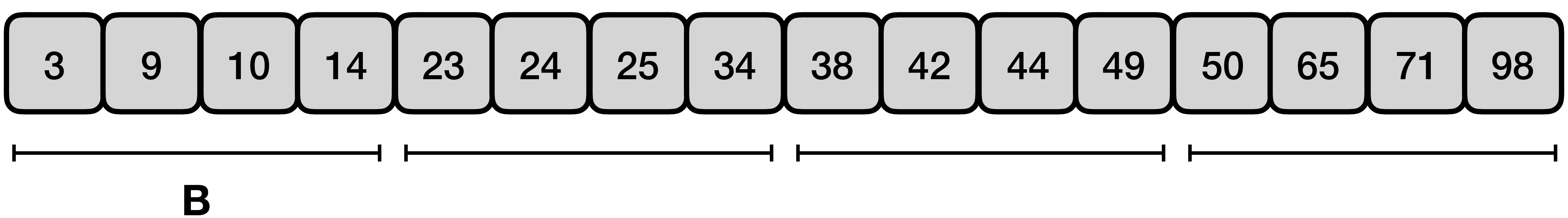
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

Solution 1
Introduce some redundancy to accelerate queries: the so-called *skip pointers.*

Partitioning by Cardinality

The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

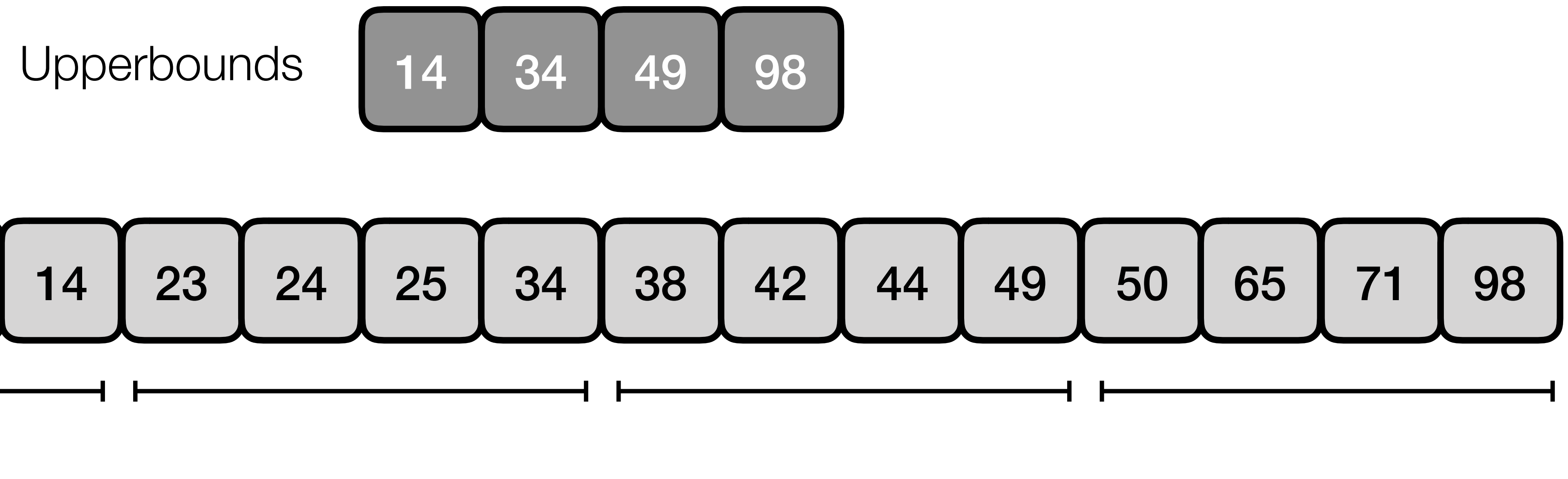
Solution 1
Introduce some redundancy to accelerate queries: the so-called *skip pointers.*



Partitioning by Cardinality

The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

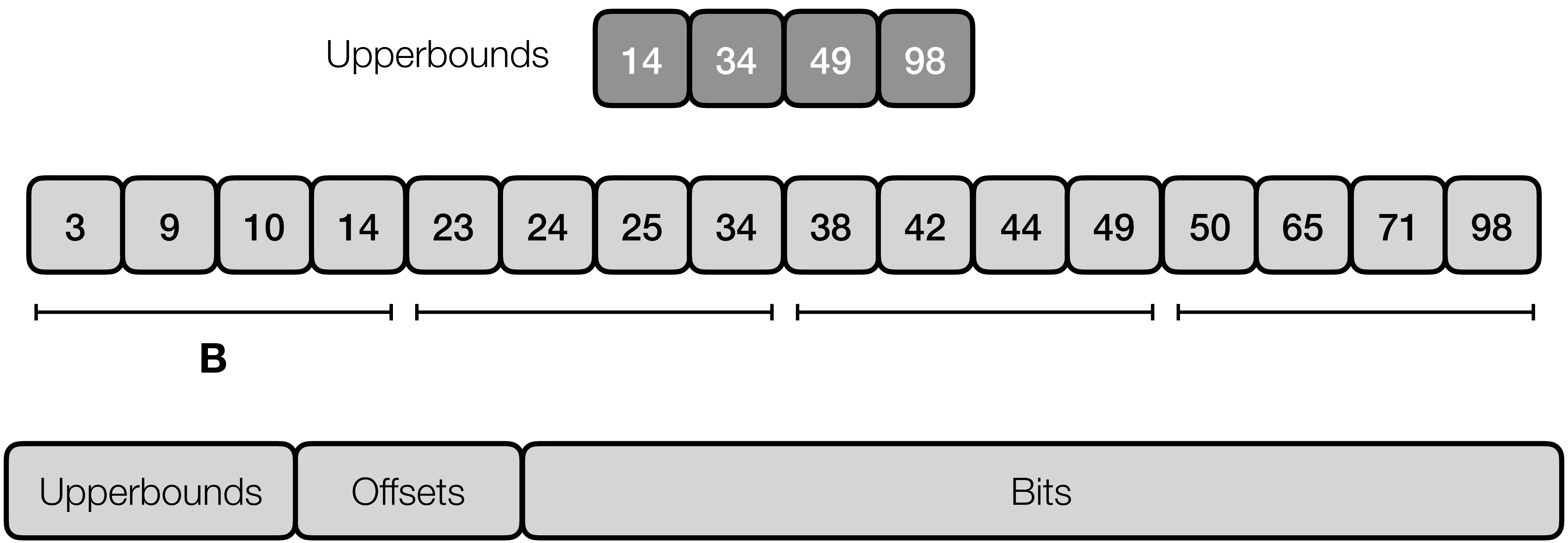
Solution 1
Introduce some redundancy to accelerate queries: the so-called *skip pointers.*



Partitioning by Cardinality

The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

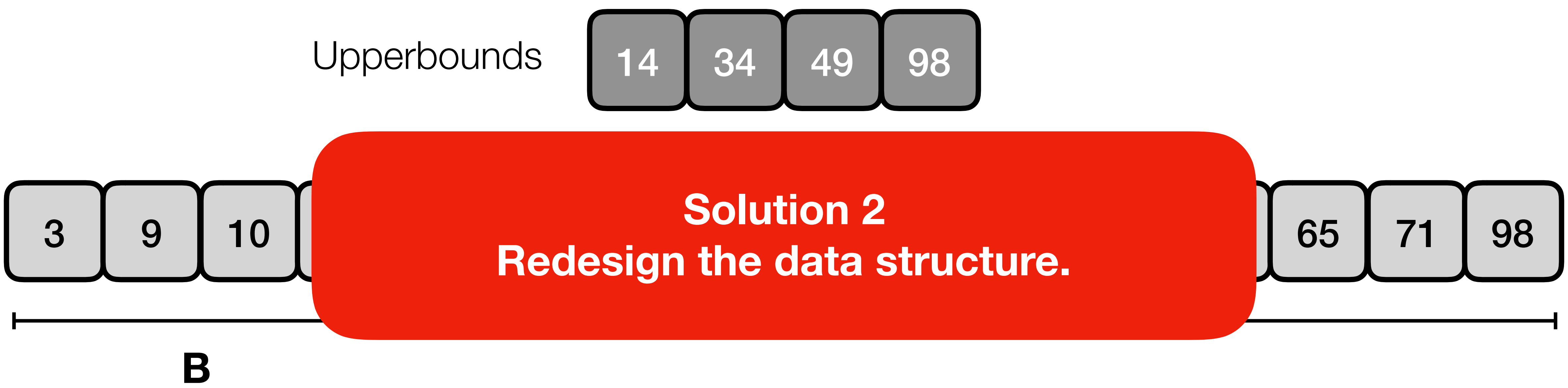
Solution 1
Introduce some redundancy to accelerate queries: the so-called *skip pointers.*



Partitioning by Cardinality

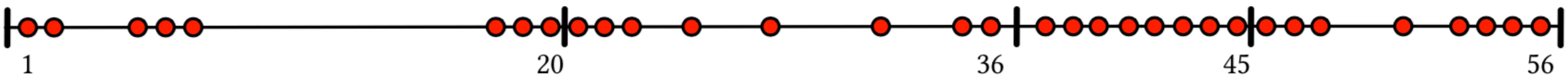
The problem of (almost all) such representations is that Access, Contains, Predecessor/Successor are **not natively supported**, but we can just decode sequentially.

Solution 1
Introduce some redundancy to accelerate queries: the so-called *skip pointers.*

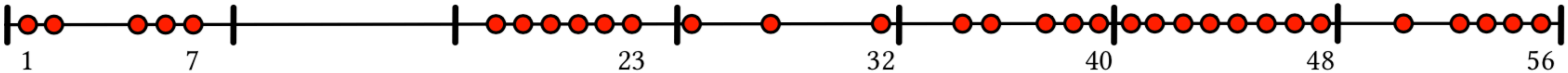


Solution 2
Redesign the data structure.

Partitioning by Universe

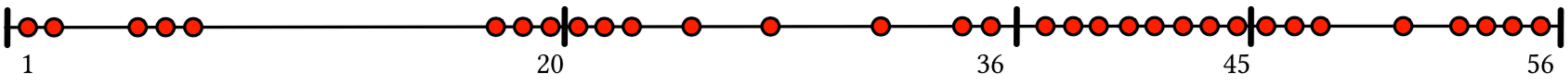


(a) partitioning by cardinality – PC

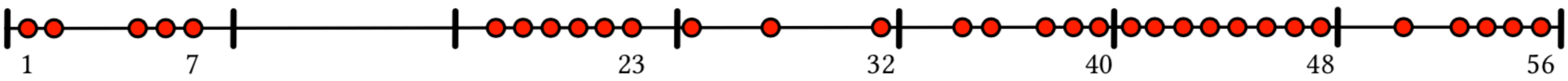


(b) partitioning by universe – PU

Partitioning by Universe



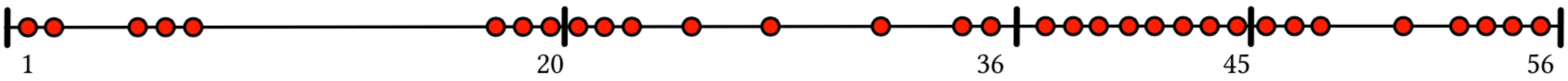
(a) partitioning by cardinality – PC



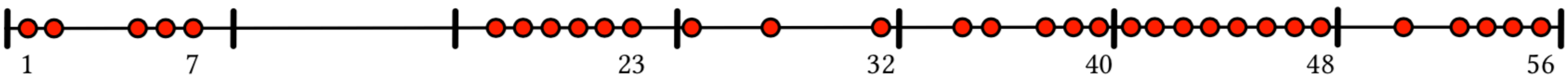
(b) partitioning by universe – PU

Does this remind you of something?

Partitioning by Universe



(a) partitioning by cardinality – PC



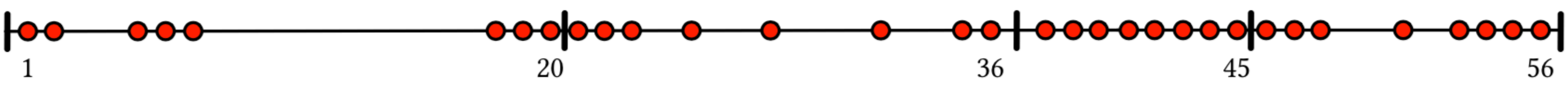
(b) partitioning by universe – PU

Does this remind you of something?

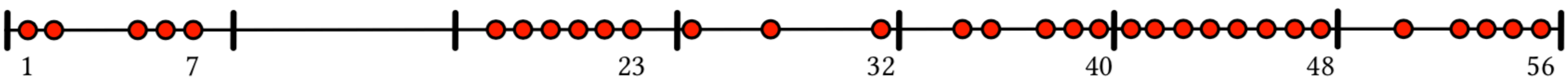
input	3	4	7	13	14	15	21	25	36	38		54	62
	0	0	0	0	0	0	0	0	1	1	**1**	1	1
high	0	0	0	0	0	0	1	1	0	0	**0**	1	1
	0	0	0	1	1	1	0	1	0	0	**1**	0	1
	0	1	1	1	1	1	1	0	1	1		1	1
low	1	0	1	0	1	1	0	0	0	1		1	1
	1	0	1	1	0	1	1	1	0	0		0	0
H	1110			1110			10	10	110		**0**	10	10
L	001-100-111			101-110-111			101	001	100-110			110	110

[Elias-Fano 1971-1975]

Partitioning by Universe



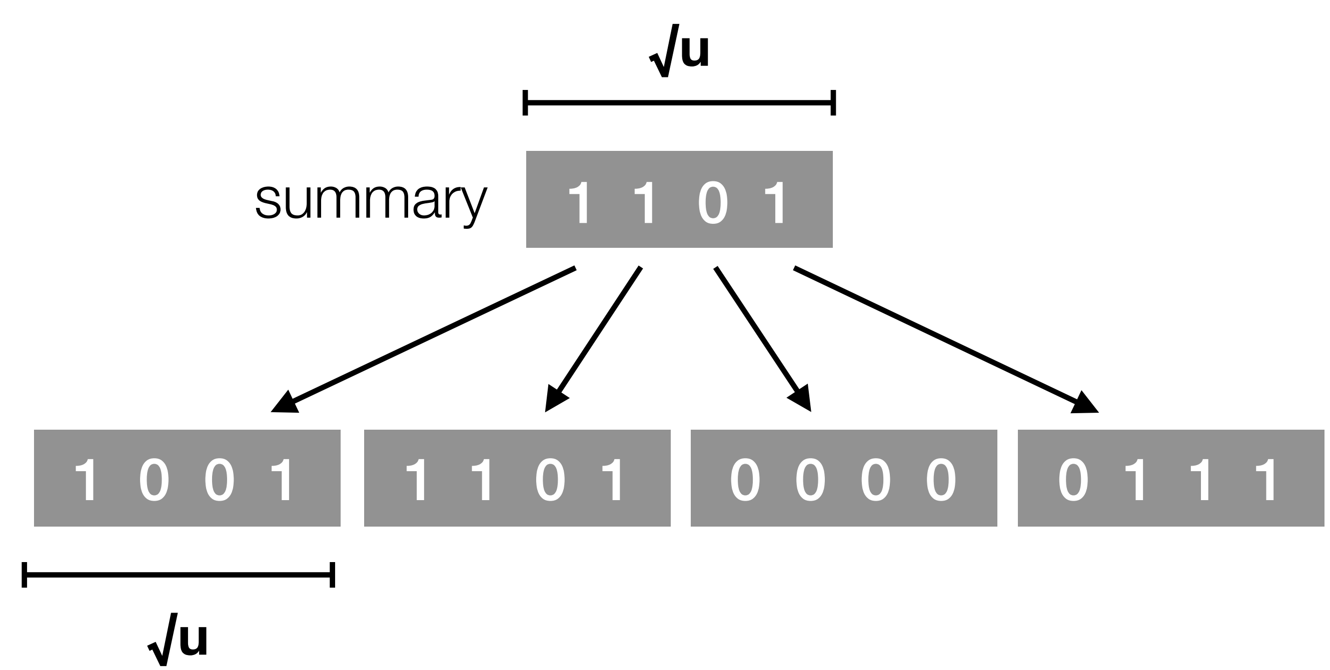
(a) partitioning by cardinality – PC



(b) partitioning by universe – PU

Does this remind you of something?

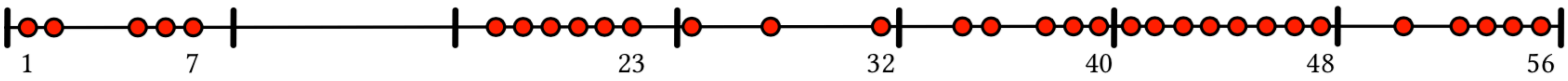
input	3	4	7	13	14	15	21	25	36	38		54	62
high	0	0	0	0	0	0	0	0	1	1	**1**	1	1
	0	0	0	0	0	0	1	1	0	0	**0**	1	1
	0	0	0	1	1	1	0	1	0	0	**1**	0	1
low	0	1	1	1	1	1	1	0	1	1		1	1
	1	0	1	0	1	1	0	0	0	1		1	1
	1	0	1	1	0	1	1	1	0	0		0	0
H	1110			1110			10	10	110		**0**	10	10
L	001-100-111			101-110-111			101	001	100-110			110	110



[Elias-Fano 1971-1975] **[van Emde Boas 1974-1975]**

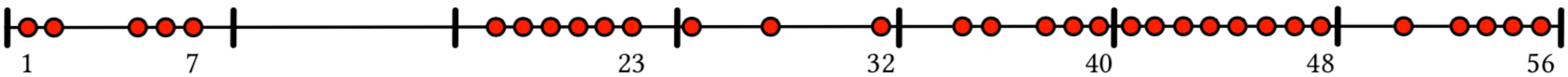
Partitioning by Universe

Assume a slice size of $\mathbf{2^3}$



Partitioning by Universe

Assume a slice size of $\mathbf{2^3}$



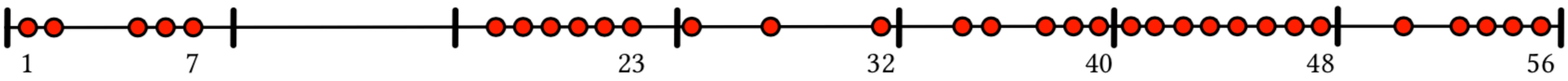
Contains(x):

i = x >> 3

search for x - (i << 3) in the i-th slice

Partitioning by Universe

Assume a slice size of $\mathbf{2^3}$



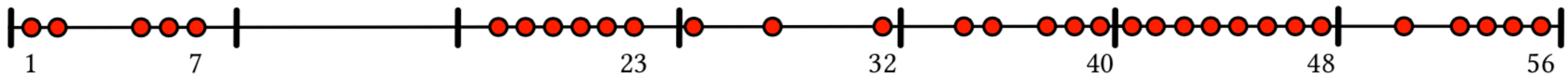
Contains(x): **x = 010101**

i = x >> 3

search for x - (i << 3) in the i-th slice

Partitioning by Universe

Assume a slice size of $\mathbf{2^3}$



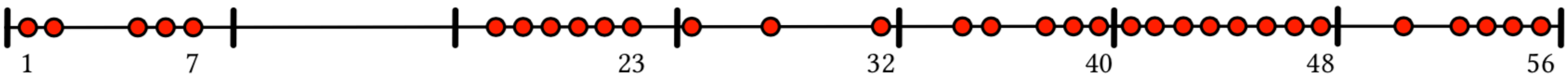
Contains(x): **x = 010101**

i = x >> 3 **010101**

search for x - (i << 3) in the i-th slice

Partitioning by Universe

Assume a slice size of 2^3



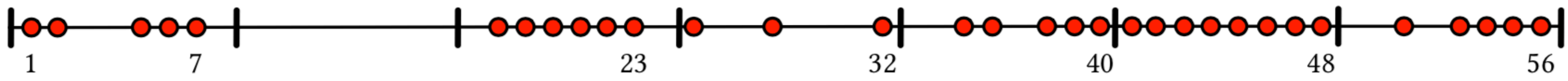
Contains(x): $\quad$ **x = 010101**

i = x >> 3 $\quad$ **010101**

search for x - (i << 3) in the i-th slice $\quad$ **x - 16 = 5**

Partitioning by Universe

Assume a slice size of **2^3**



Contains(x): **x = 010101**

i = x >> 3 **010101**

search for x - (i << 3) in the i-th slice **x - 16 = 5**

Successor(x):

i = x >> 3

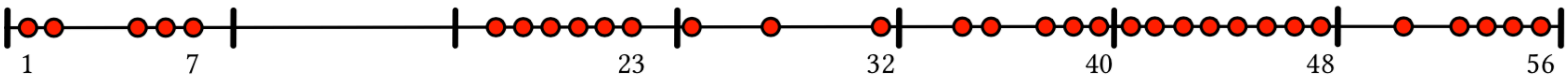
search for successor of x - (i << 3) in the i-th slice

(if i-th slice is empty or x - (i << 3) > max_value in i-th slice,

then return first value on the right)

Partitioning by Universe

Assume a slice size of $\mathbf{2^3}$



Contains(x): **x = 010101**

i = x >> 3 **010101**

search for x - (i << 3) in the i-th slice **x - 16 = 5**

Successor(x):

i = x >> 3

search for successor of x - (i << 3) in the i-th slice

(if i-th slice is empty or x - (i << 3) > max_value in i-th slice,

then return first value on the right)

Intersection between lists has to intersect **only the slices in common** between the lists.

Bitmaps

Good old data structure for storing **dense sets**:
x-th bit is set if integer x is in the set.

Bitmaps

Good old data structure for storing **dense sets**:
x-th bit is set if integer x is in the set.

$S = \{0,1,5,7,8,10,11,14,18,21,22,28,29,30\}$

↕

1	1	0	0	0	1	0	1	1	0	1	1	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bitmaps

Good old data structure for storing **dense sets**:
x-th bit is set if integer x is in the set.

S = {0,1,5,7,8,10,11,14,18,21,22,28,29,30}

↕

1	1	0	0	0	1	0	1	1	0	1	1	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Contains: testing a bit
Successor/Predecessor: `__builtin_ctzll`
Select: `__builtin_ctzll`
Max: `__builtin_clzll`
Min: `__builtin_ctzll`
Decode: `__builtin_ctzll`
Insertion: setting a bit
Deletion: clearing a bit

Bitmaps

Good old data structure for storing **dense sets**:
x-th bit is set if integer x is in the set.

$S = \{0,1,5,7,8,10,11,14,18,21,22,28,29,30\}$

↕

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	1	0	0	0	1	0	1	1	0	1	1	0	0	1	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1	0

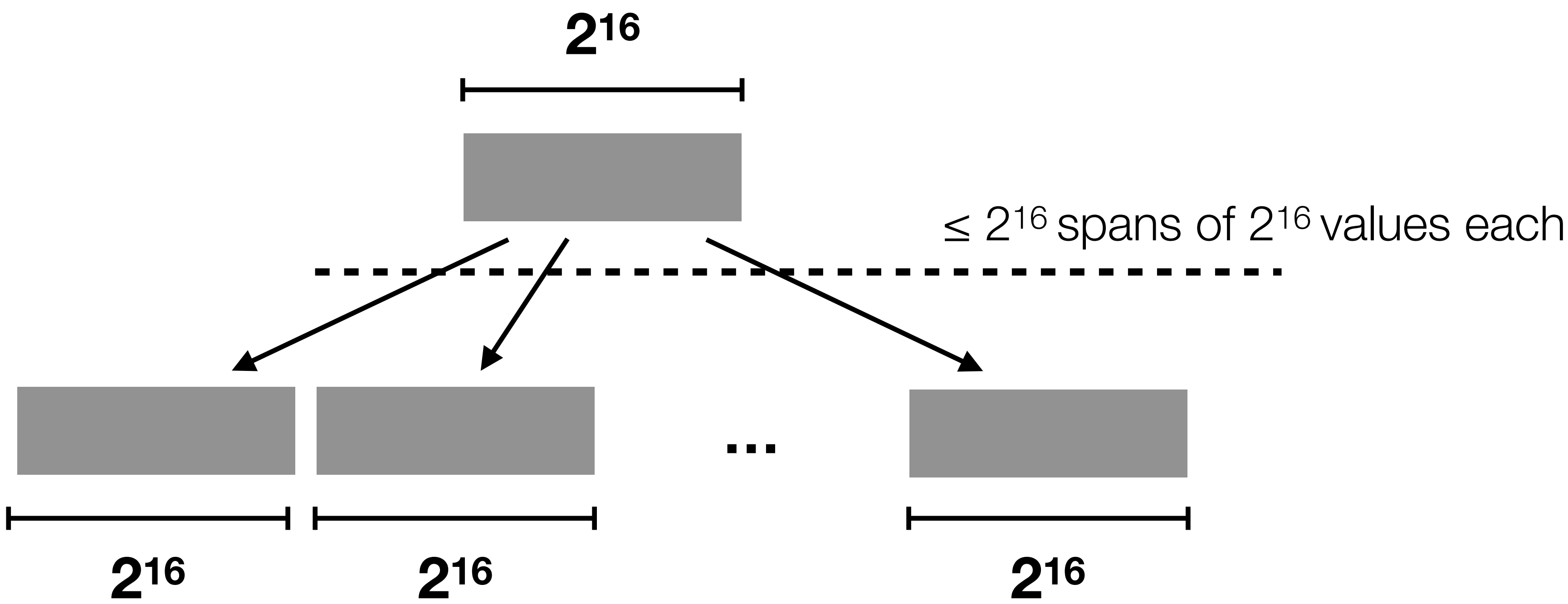
Contains: testing a bit
Successor/Predecessor: `__builtin_ctzll`
Select: `__builtin_ctzll`
Max: `__builtin_clzll`
Min: `__builtin_ctzll`
Decode: `__builtin_ctzll`
Insertion: setting a bit
Deletion: clearing a bit

Nothing is better than a bitmap for dense sets.

Roaring

[Lemire et al. 2013]

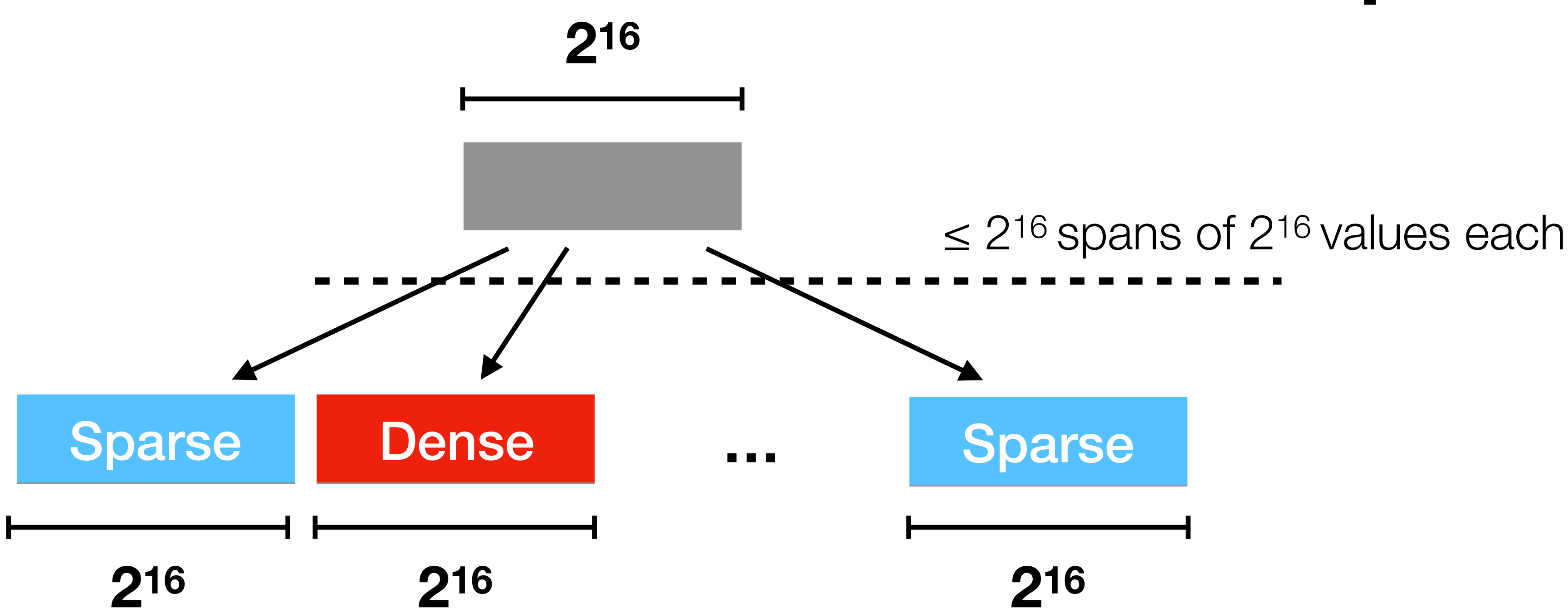
Assume $u = 2^{32}$



Roaring

[Lemire et al. 2013]

Assume $u = 2^{32}$



Dense: cardinality > 4096
Sparse: otherwise

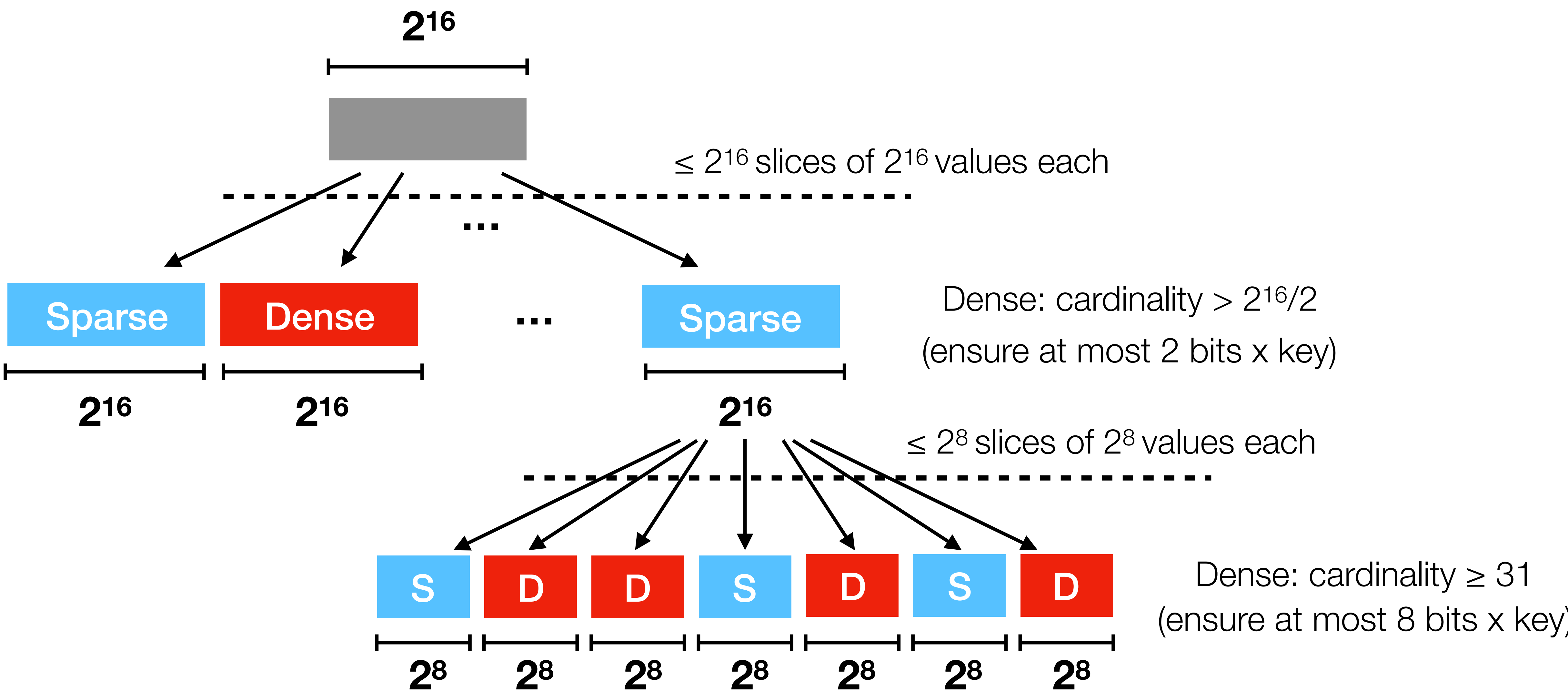
Ensure at most 16 bits x key
(excluding overhead)

Dense spans are represented with **bitmaps** of 2^{16} bits.

Sparse spans are represented with **sorted-arrays** of 16-bit integers.

Slicing

Assume $u = 2^{32}$



Dense slices are represented with bitmaps of 2^{16} or 2^{8} bits.

Sparse slices are represented with sorted-arrays of 8-bit integers.

Intersection

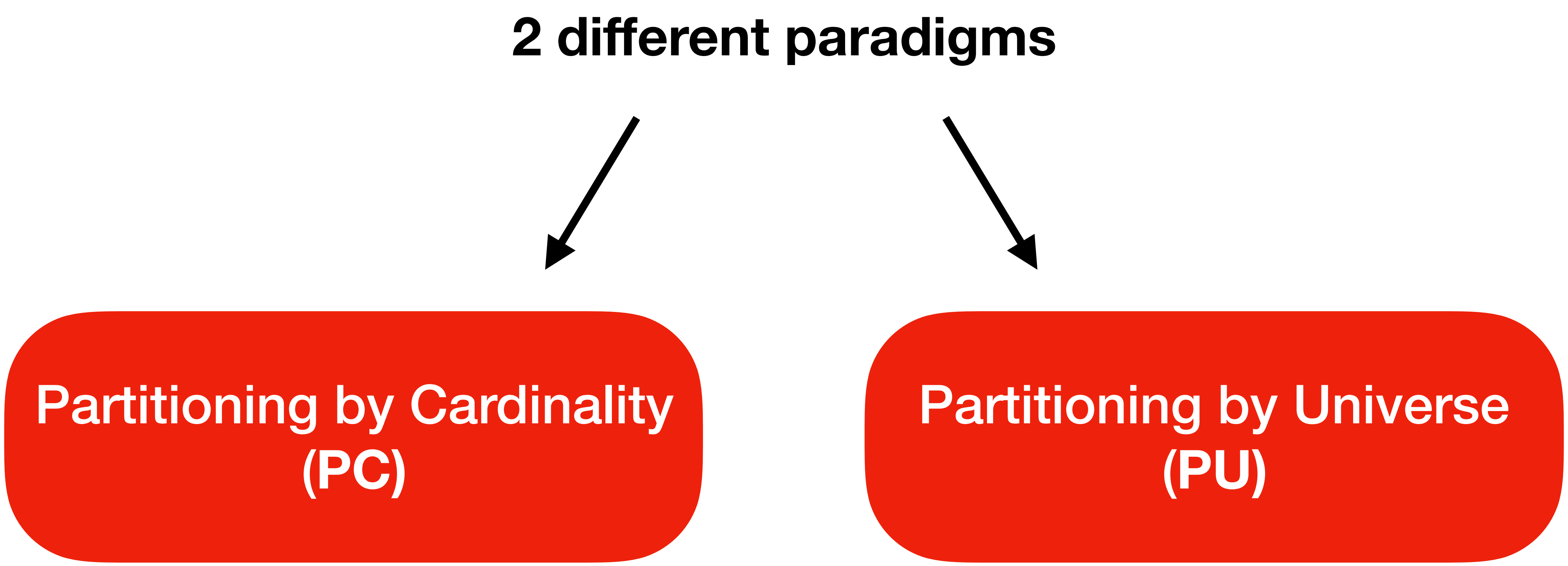
Intersection between lists has to intersect **only the slices in common** between the lists.

- **Dense vs. Dense (Bitmap vs. Bitmap):**
  bitwise AND operations + (usually) automatic compiler vectorization

- **Dense vs. Sparse (Bitmap vs. Array):**
  Given the array `A`: check if bit `A[i]` is set in the bitmap

- **Sparse vs. Sparse (Array vs. Array):**
  Vectorized processing using `_mm_cmpestrm` and `_mm_shuffle_epi8` SIMD instructions

Summing up


2 different paradigms
Partitioning by Cardinality (PC)
Partitioning by Universe (PU)

Experimental Comparison — Setting

Datasets

Statistic	Gov2	CW09	CCNews
Lists	35,636,425	92,094,694	43,844,574
Universe	24,622,347	50,131,015	43,530,315
Integers	5,742,630,292	15,857,983,641	20,150,335,440

Machine

Intel i7-4790K CPU @4GHz, 32 GiB RAM, Linux 4.13.0

Compiler

`gcc` 7.2.0 (with all optimizations: `-march=native` and `-O3`)

C++ sources

https://github.com/jermp/s_indexes
https://github.com/jermp/dint
https://github.com/ot/ds2i
https://github.com/RoaringBitmap/CRoaring

Experimental Comparison — Setting

Datasets

Density	Statistic	Gov2	CW09	CCNews
10^{-2}	Lists	3513	5802	5930
	Integers	4,347,653,438	11,676,154,022	16,677,342,102
	%	76	74	83
10^{-3}	Lists	13,276	21,924	23,085
	Integers	5,066,748,826	13,864,451,283	18,969,946,075
	%	88	87	94
10^{-4}	Lists	85,893	99,227	79,954
	Integers	5,390,038,277	14,805,194,135	19,681,352,639
	%	94	93	98

Configurations

Method	Shorthand	Strategy
Variable-Byte	V	PC; fixed-sized partitions of 128 integers; byte-aligned
Elias-Fano	EF	PC; fixed-sized partitions of 128 integers; bit-aligned
Interpolative	BIC	PC; fixed-sized partitions of 128 integers; bit-aligned
Elias-Fano ϵ-opt.	PEF	PC; variable-sized partitions; bit-aligned
Roaring without run opt.	R2	PU; single-span; 2 container types; byte-aligned
Roaring with run opt.	R3	PU; single-span; 3 container types; byte-aligned
Slicing	S	PU; multi-span; byte-aligned

Experimental Comparison — Compression Effectiveness

bits per integer

Method	$d = 10^{-2}$			$d = 10^{-3}$			$d = 10^{-4}$		
	Gov2	CW09	CCNews	Gov2	CW09	CCNews	Gov2	CW09	CCNews
V	8.60	8.72	8.66	8.72	9.00	9.08	8.85	9.19	9.28
EF	2.72	4.44	4.72	3.25	5.14	5.37	3.65	5.56	5.66
BIC	2.33	3.59	4.37	2.72	4.11	4.97	3.02	4.41	5.24
PEF	2.37	4.01	4.52	2.85	4.62	5.16	3.20	4.96	5.45
R2	6.00	8.88	8.25	7.03	9.99	9.21	7.60	10.47	9.53
R3	5.33	8.49	8.22	6.25	9.40	9.17	6.75	9.75	9.48
S	3.23	5.44	5.98	3.91	6.39	7.18	4.46	7.00	7.77

PC-based methods, such as BIC and PEF, are best for space usage.
Slicing (PU-based) stands in trade-off position.

Experimental Comparison — Sequential Decoding Time

ns per integer

Method	$d = 10^{-2}$			$d = 10^{-3}$			$d = 10^{-4}$		
	Gov2	CW09	CCNews	Gov2	CW09	CCNews	Gov2	CW09	CCNews
V	0.51	0.61	0.53	0.55	0.66	0.59	0.58	0.71	0.62
EF	0.87	1.29	1.36	0.94	1.34	1.41	0.98	1.36	1.42
BIC	5.26	6.73	7.71	5.54	6.95	7.86	5.70	7.01	7.90
PEF	0.78	1.15	1.34	0.86	1.22	1.48	0.91	1.25	1.53
R2	0.53	0.72	0.68	0.53	0.70	0.69	0.54	0.71	0.69
R3	0.55	0.76	0.70	0.55	0.76	0.69	0.57	0.78	0.70
S	0.56	0.67	0.65	0.57	0.69	0.67	0.60	0.73	0.71

PU-based methods, are as fast as the fastest (vectorized) PC-based methods.

Experimental Comparison — Intersection Time

musec per intersection

Method	$d = 10^{-2}$			$d = 10^{-3}$			$d = 10^{-4}$		
	Gov2	CW09	CCNews	Gov2	CW09	CCNews	Gov2	CW09	CCNews
V	3648	6671	16954	710	1591	3732	40	214	523
EF	4652	8356	22818	856	1700	4455	40	192	530
BIC	12169	23608	58349	2649	6377	14765	160	905	2323
PEF	4380	7920	21710	826	1640	4185	40	190	490
R2	377	598	1138	99	232	353	10	57	98
R3	503	962	1338	128	331	395	13	75	115
S	507	1080	2370	135	378	820	11	60	159

PU-based methods outperform PC-based methods.

Experimental Comparison — Point Queries

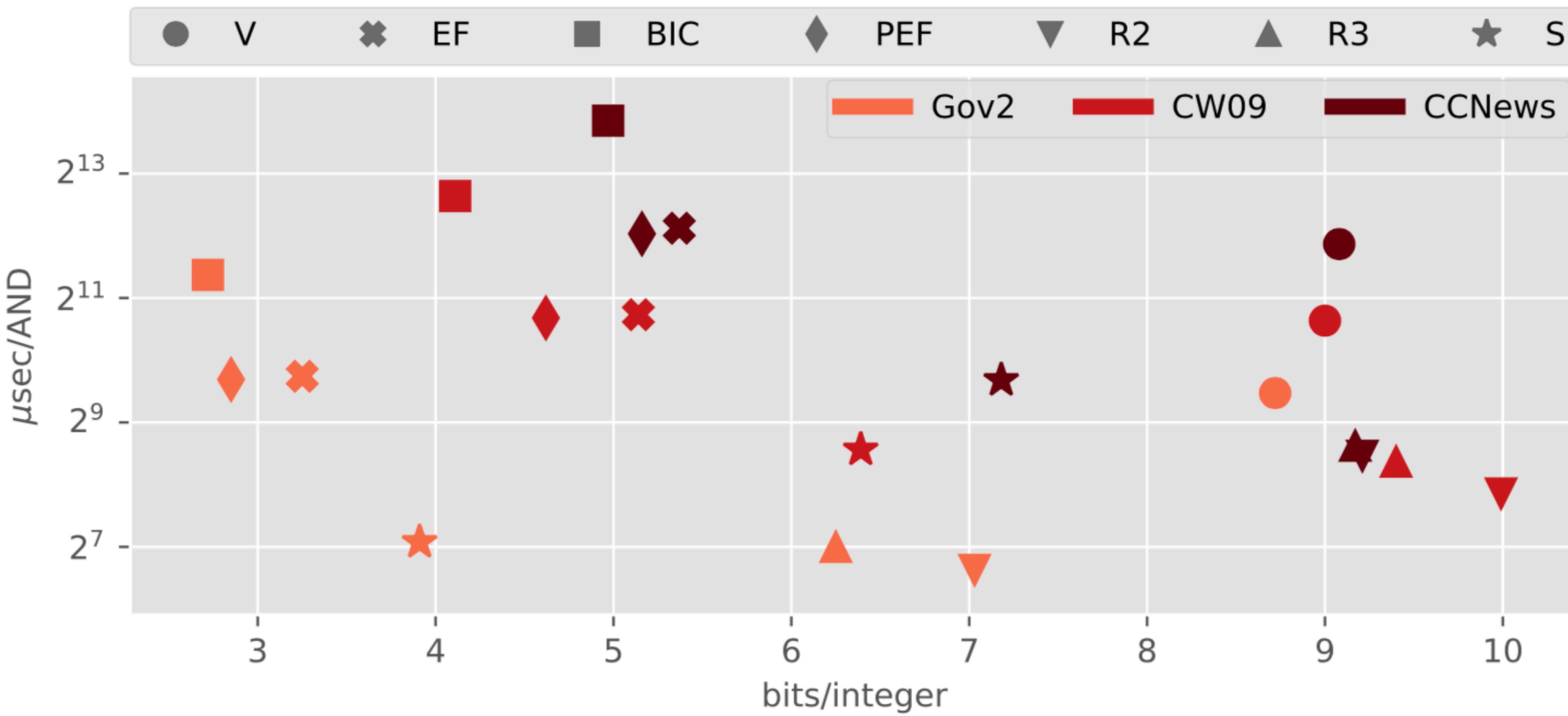
Access: ns per query

Method	$d = 10^{-2}$			$d = 10^{-3}$			$d = 10^{-4}$		
	Gov2	CW09	CCNews	Gov2	CW09	CCNews	Gov2	CW09	CCNews
V	195	174	240	155	184	222	105	151	189
EF	118	122	173	88	103	123	58	75	86
BIC	890	835	1295	904	960	1230	685	876	1062
PEF	154	171	210	118	145	126	77	100	72
R2	475	545	610	294	453	402	111	365	310
R3	5604	18710	2852	2151	7681	1221	443	2254	612
S	153	170	244	105	116	152	55	61	78

Successor: ns per query

Method	$d = 10^{-2}$			$d = 10^{-3}$			$d = 10^{-4}$		
	Gov2	CW09	CCNews	Gov2	CW09	CCNews	Gov2	CW09	CCNews
V	252	226	308	255	226	279	197	181	243
EF	187	122	250	146	155	175	91	113	120
BIC	955	897	1385	951	1012	1290	710	878	1100
PEF	167	182	229	138	157	144	94	118	89
R2	115	137	185	90	119	133	55	80	82
R3	105	138	188	80	115	136	50	72	85
S	145	174	225	90	110	134	48	57	69

Experimental Comparison — The Trade-Off Curve



Density = 1/1000

Future Research Directions

The Static Ordered Set Problem

↓

The *Dynamic* Ordered Set Problem

+ insertions / deletions

Future Research Directions

The Static Ordered Set Problem

↓

The *Dynamic* Ordered Set Problem

+ insertions / deletions

Theory

Fusion Trees
van Emde Boas Trees
Exponential Search Trees
Y-Fast Tries
Dynamic Elias-Fano

Practice

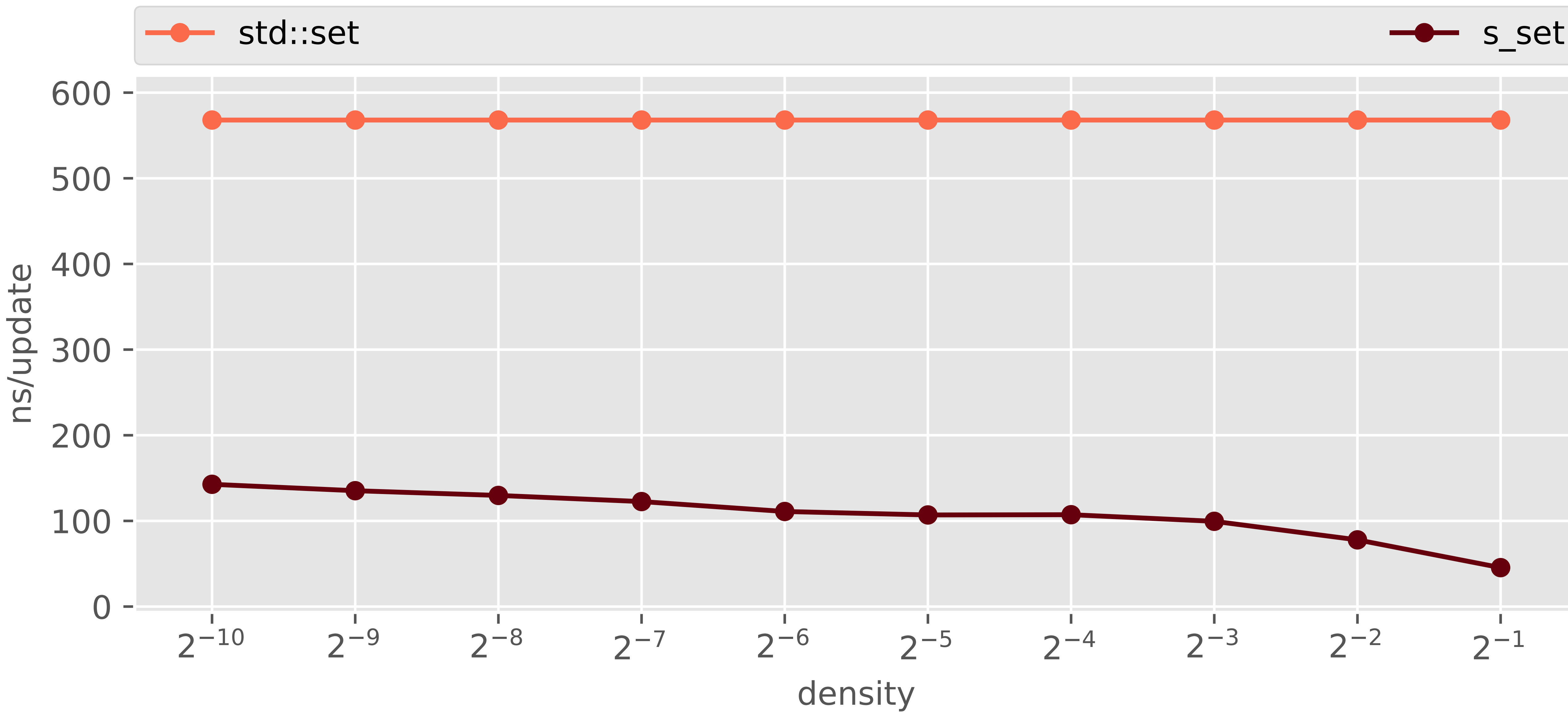
Red-Black Trees
B-Trees

Memory management is the challenge.

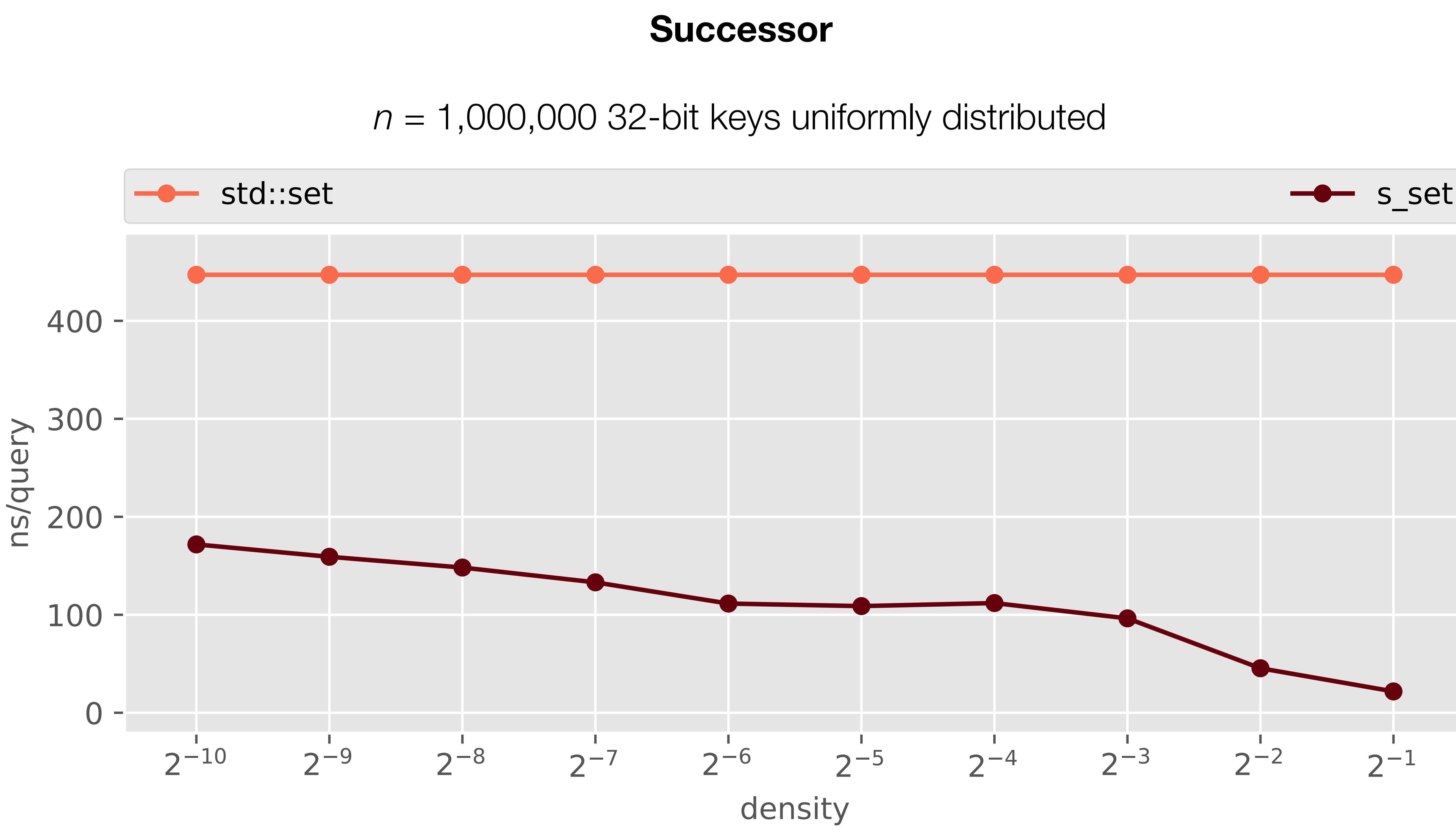
The Dynamic Ordered Set Problem — On-going Work

Insert

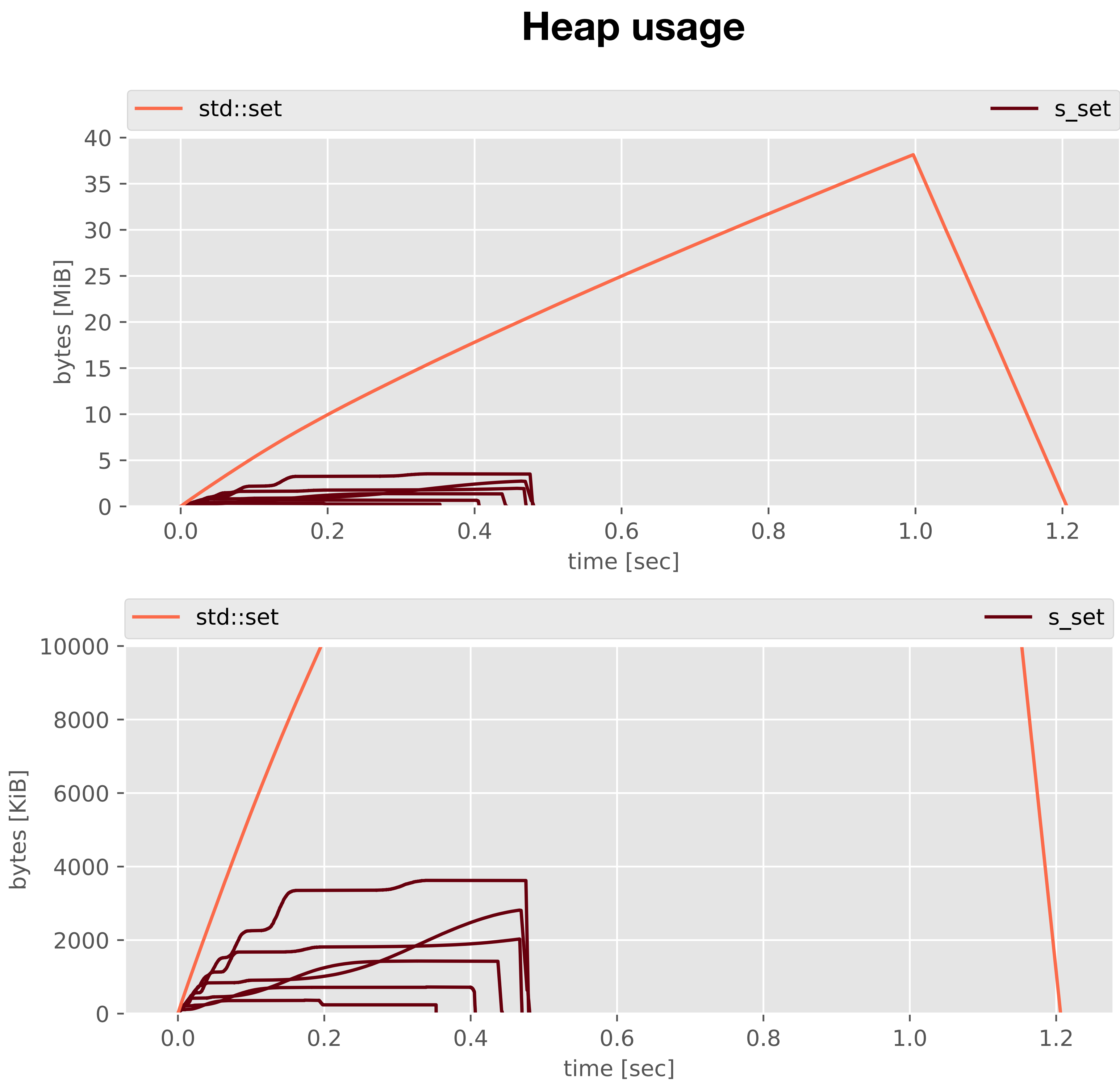
n = 1,000,000 32-bit keys uniformly distributed



The Dynamic Ordered Set Problem — On-going Work



The Dynamic Ordered Set Problem — On-going Work



Thanks for your attention, time, patience!

Any questions?