Università di Pisa

Learned data structures

(Seminar for Algorithm Engineering A.Y. 2019/20)

~

Giorgio VINCIGUERRA
pages.di.unipi.it/vinciguerra/

Outline

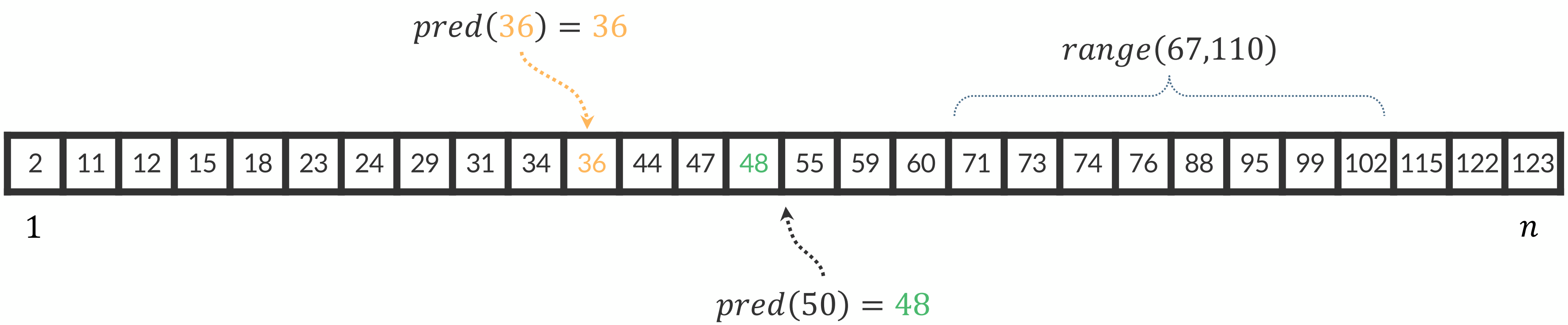
~

1. Background
   - Predecessor & range search in external memory
2. Learned structures for pred & range search
   - RMI by Google + MIT
   - PGM-index by UNIPI
3. Learned structures for approx membership
4. (Extra) tools for writing efficient code

Not part of the syllabus

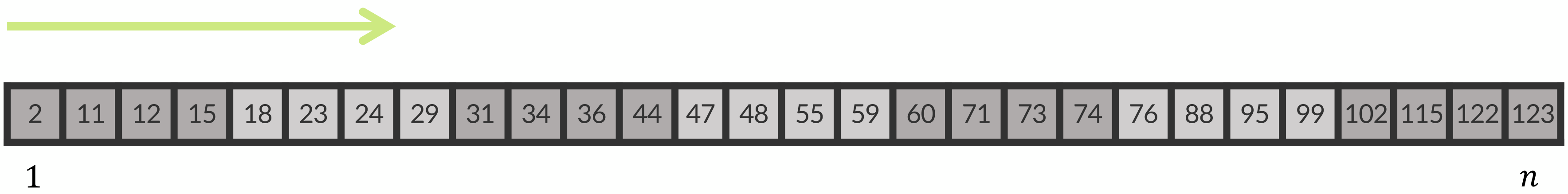
Background

Predecessor search & range queries

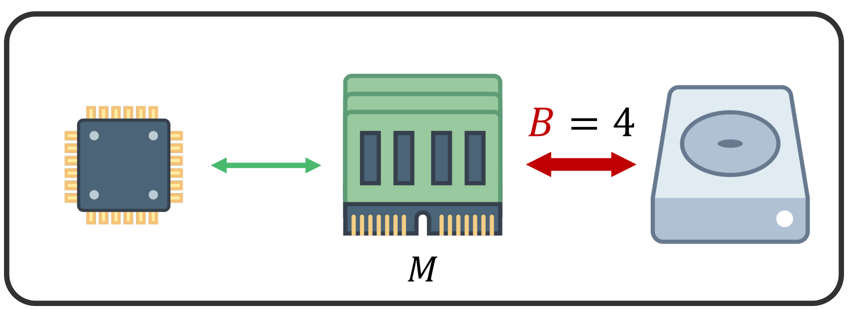


Baseline solutions for predecessor search

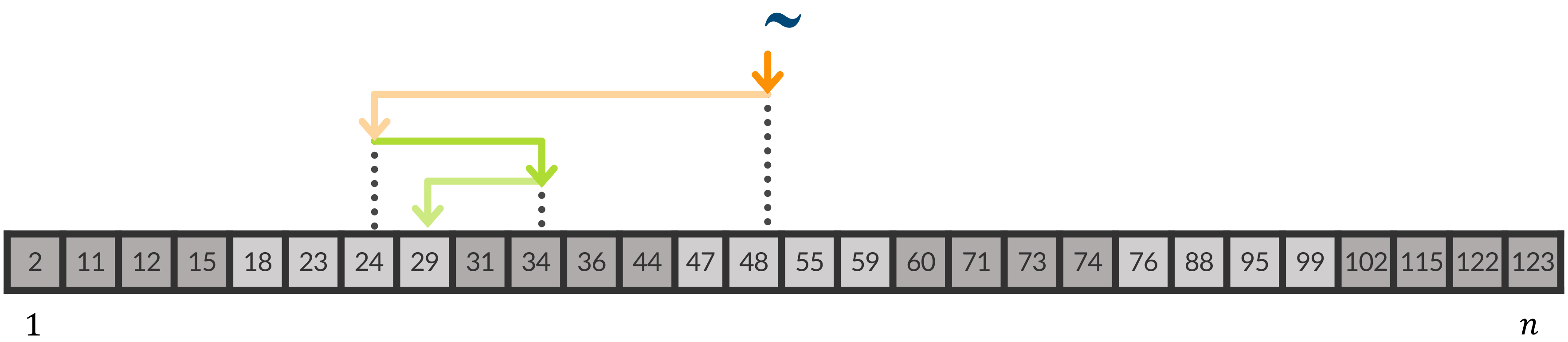
~



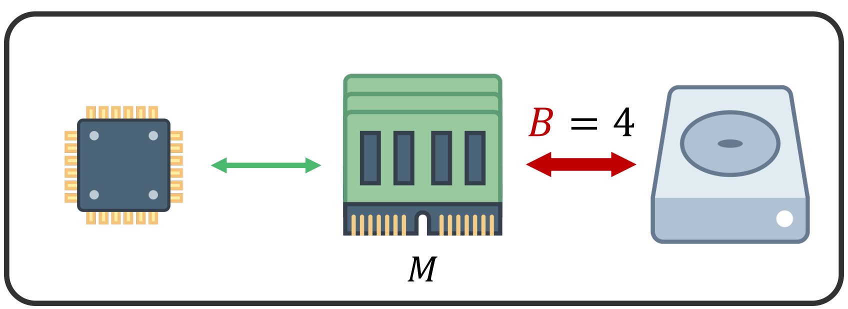
Solution	RAM model **Worst case time**	EM model **Worst case I/Os**	EM model **Best case I/Os**
Scan	$O(n)$	$O(n/B)$	$O(1)$



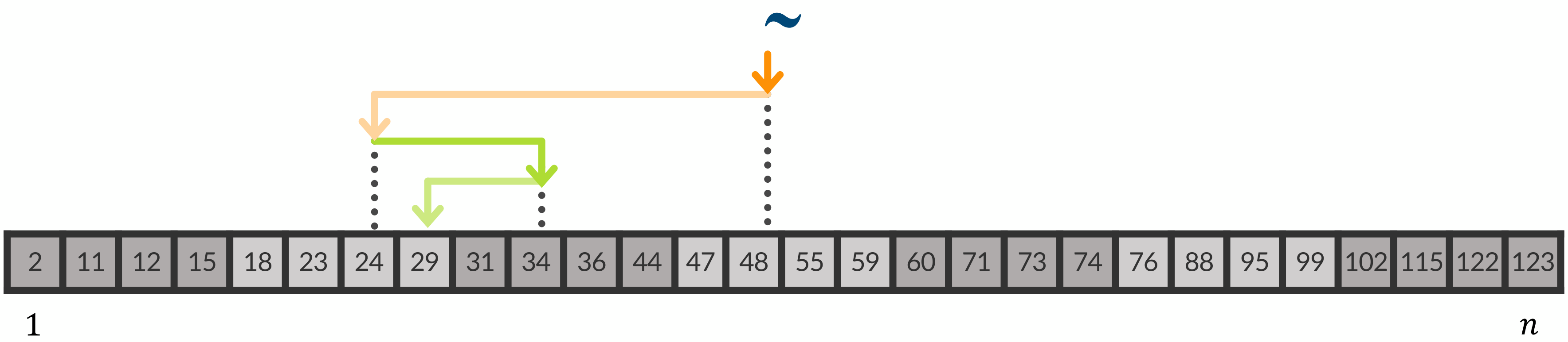
Baseline solutions for predecessor search



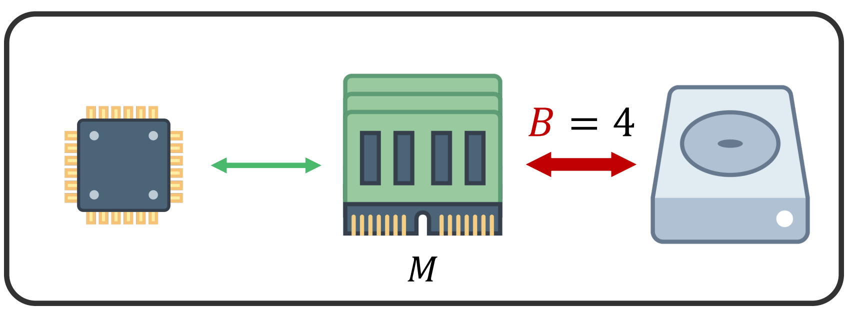
Solution	RAM model **Worst case time**	EM model **Worst case I/Os**	EM model **Best case I/Os**
Scan	$O(n)$	$O(n/B)$	$O(1)$
Binary search	$O(\log n)$		



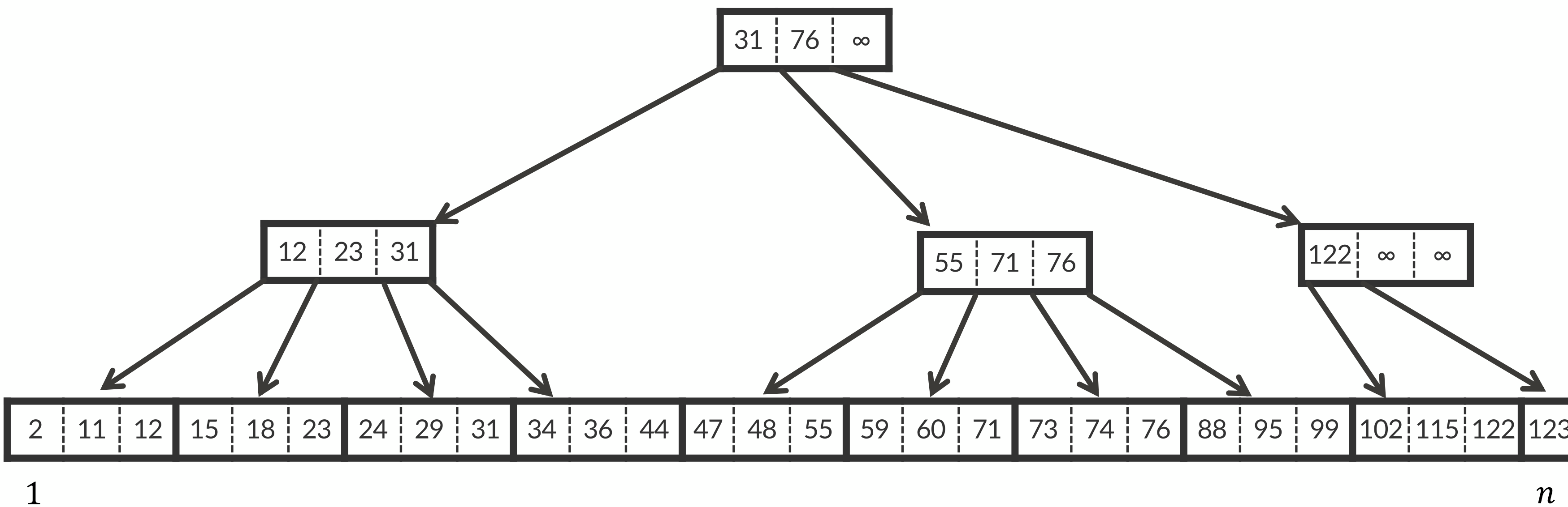
Baseline solutions for predecessor search



Solution	RAM model **Worst case time**	EM model **Worst case I/Os**	EM model **Best case I/Os**
Scan	$O(n)$	$O(n/B)$	$O(1)$
Binary search	$O(\log n)$	$O(\log(n/B))$	$O(\log(n/B))$



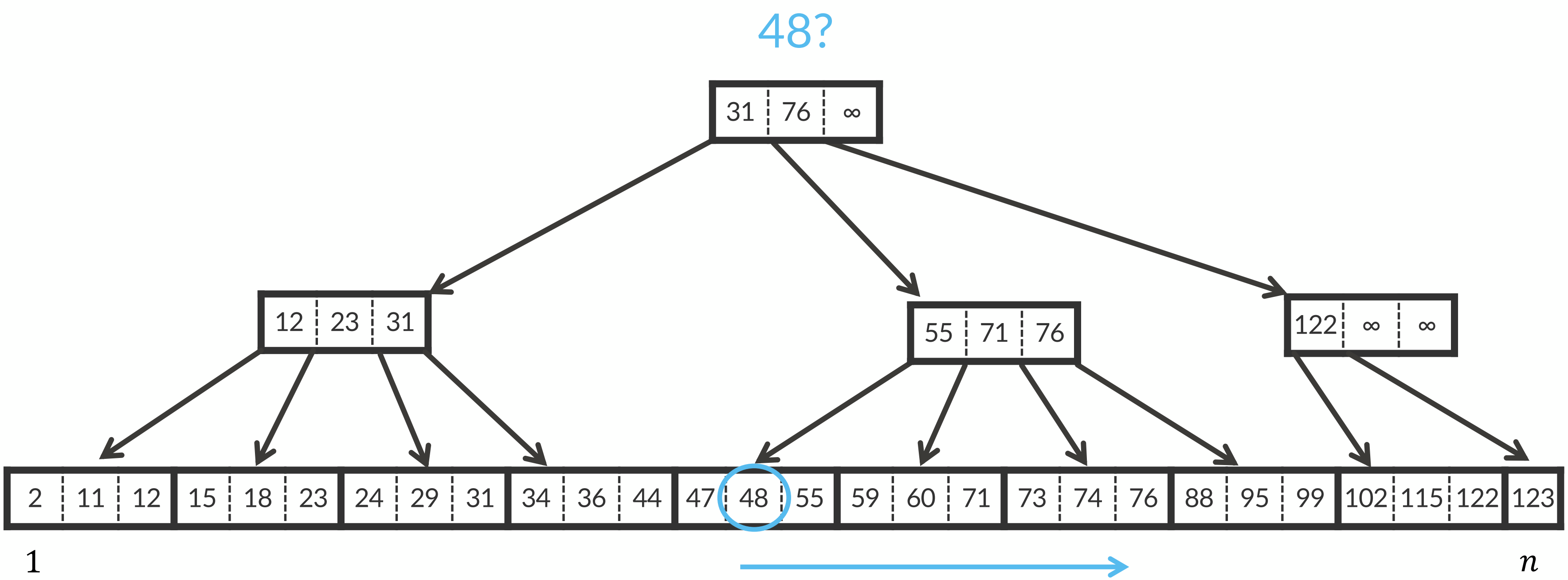
B+ trees



31
76
∞
12
23
31
55
71
76
122
∞
∞
2
11
12
15
18
23
24
29
31
34
36
44
47
48
55
59
60
71
73
74
76
88
95
99
102
115
122
123
1
n

B⁺ trees

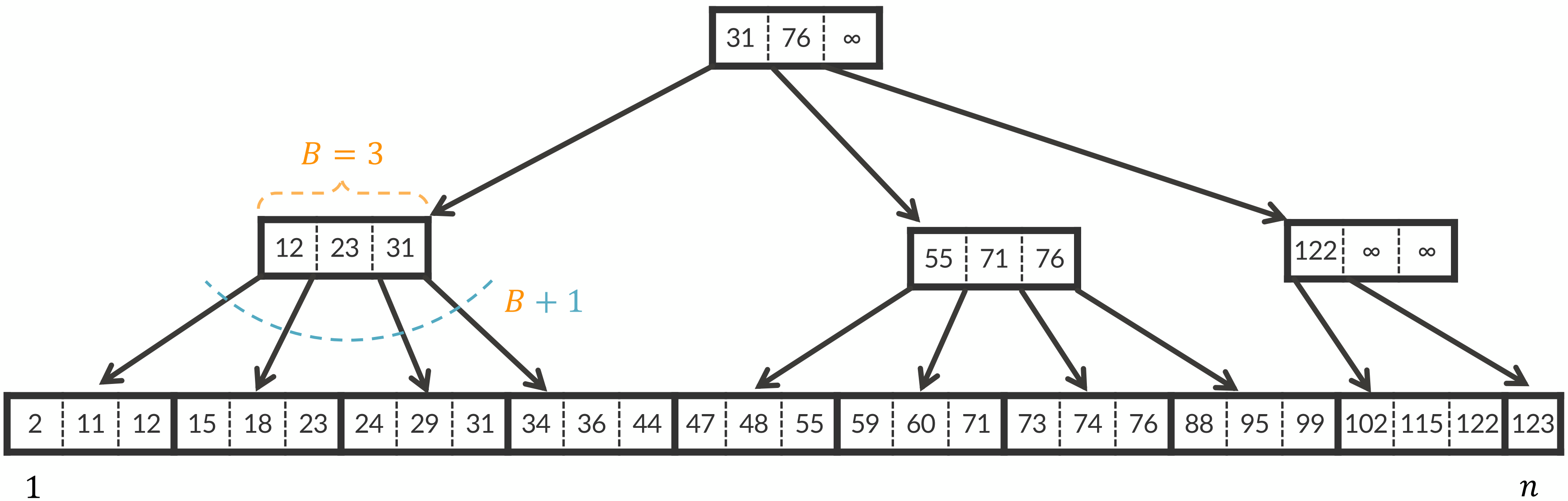
~



48?
31 76 ∞
12 23 31
55 71 76
122 ∞ ∞
2 11 12 15 18 23 24 29 31 34 36 44 47 48 55 59 60 71 73 74 76 88 95 99 102 115 122 123
1
n

B^+ trees

~



Solution	Space	RAM model **Worst case time**	EM model **Worst case I/Os**
Scan	$O(1)$	$O(n)$	$O(n/B)$
Binary search	$O(1)$	$O(\log n)$	$O(\log(n/B))$
B^+ tree	$O(n)$	$O(\log n)$	$O(\log_B n)$

B-trees are everywhere

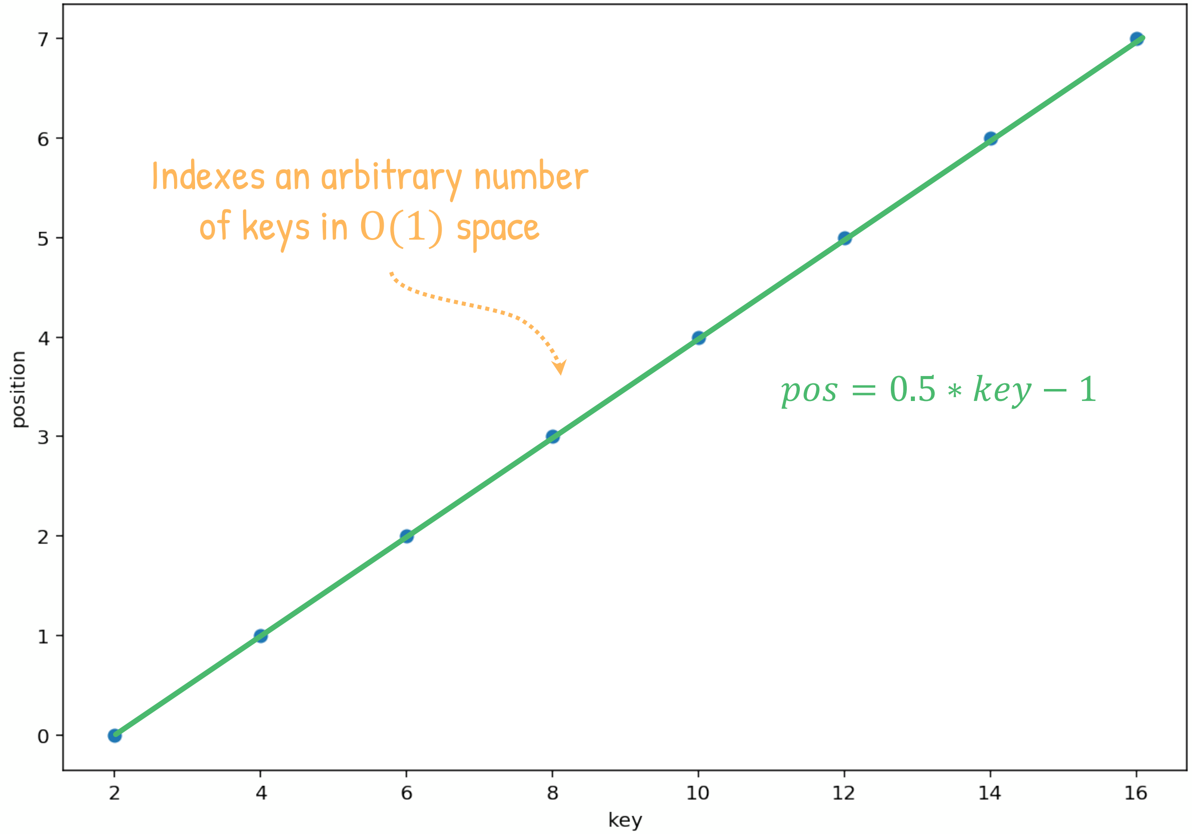
~

1. "B-trees have become, de facto, a standard for file organization" Comer. *Ubiquitous B-tree*. ACM Computing Surveys. '79
2. This is still true today

A closer look at the data

~

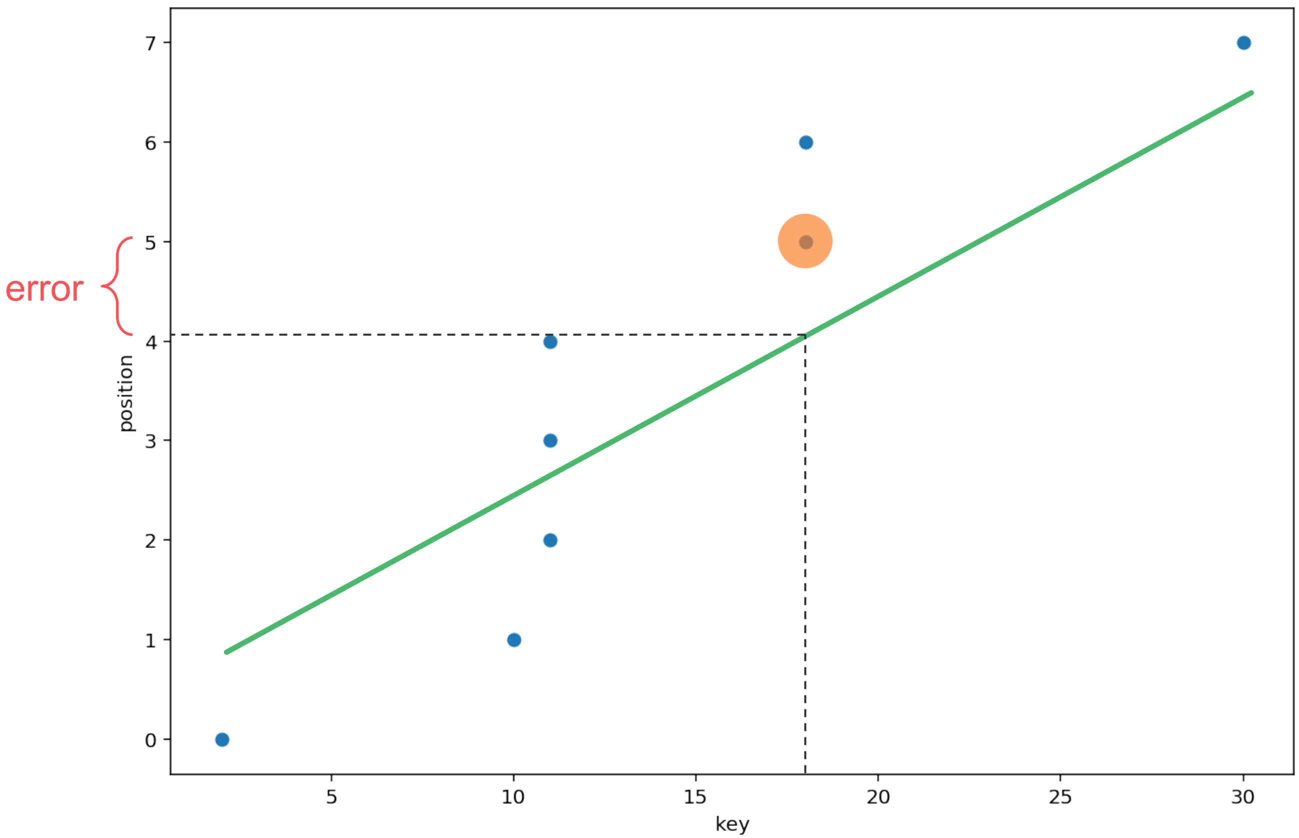
keys	2	4	6	8	10	12	14	16
positions	0	1	2	3	4	5	6	7



A closer look at realistic data

~

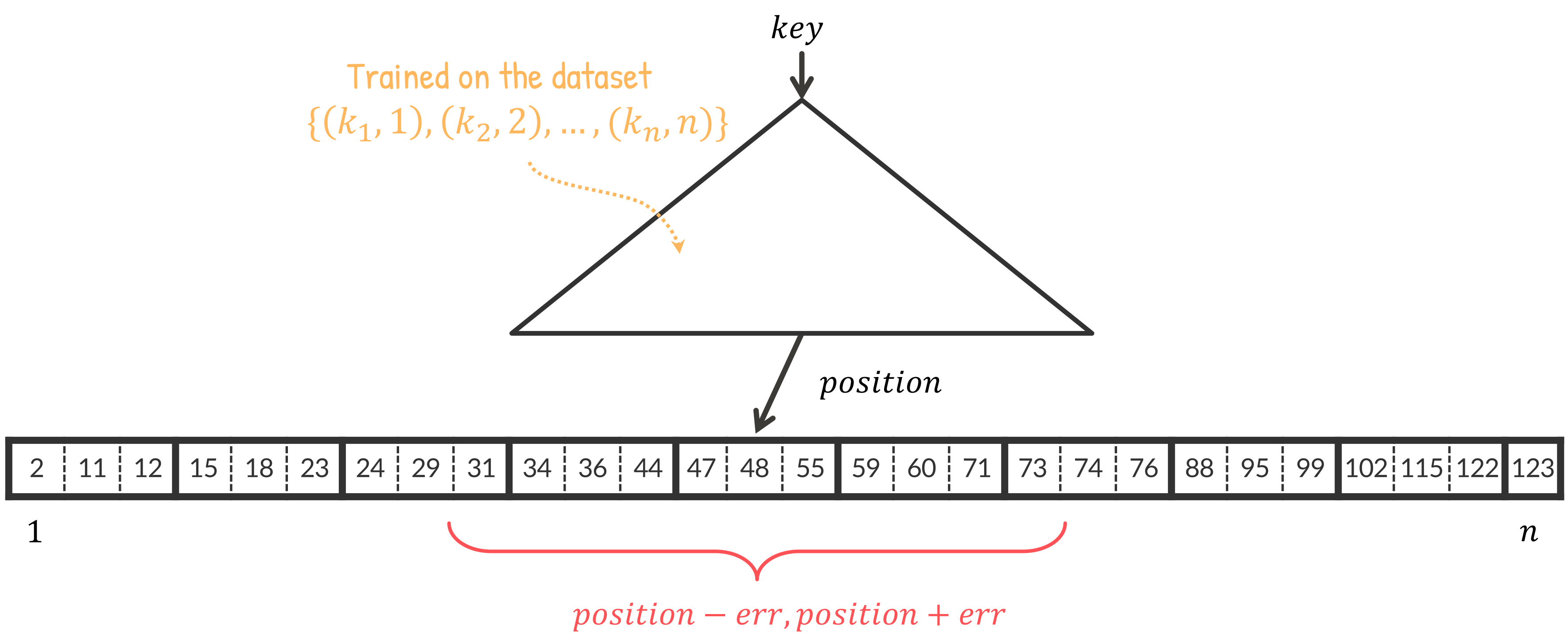
keys	2	10	11	11	11	18	18	30
positions	0	1	2	3	4	5	6	7



B-trees are machine learning models

~

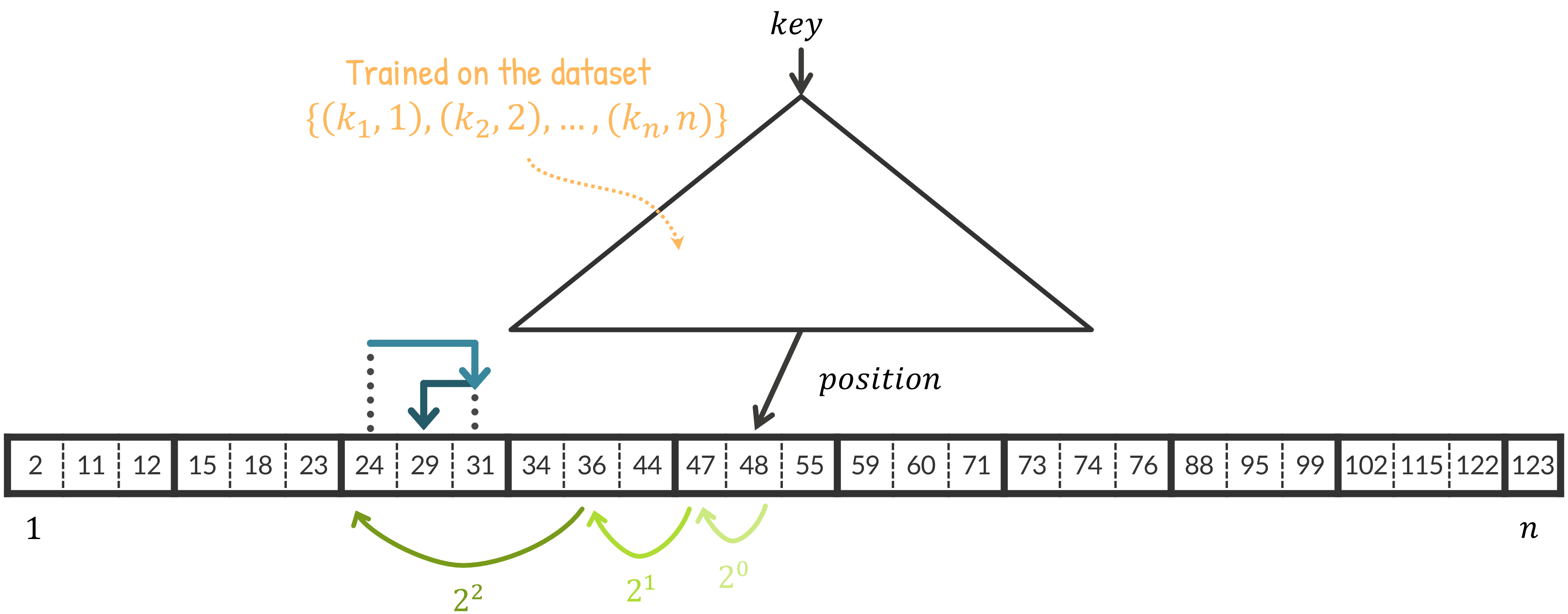
“All existing index structures can be replaced with other types of models, including deep-learning models, which we term learned indexes.” [SIGMOD ’18]



B-trees are machine learning models

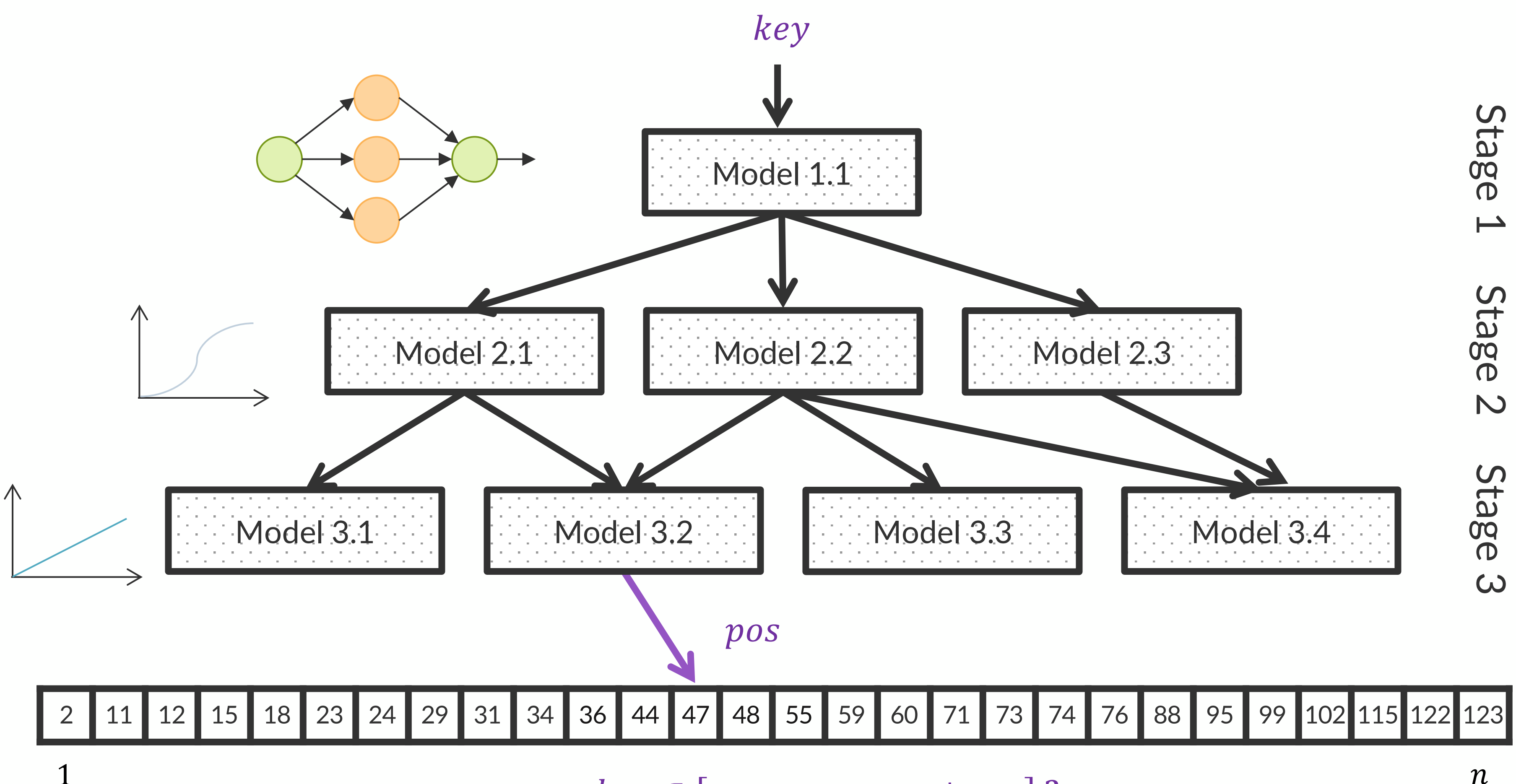
~

"All existing index structures can be replaced with other types of models, including deep-learning models, which we term learned indexes." [SIGMOD '18]



The Recursive Model Index (RMI)

~

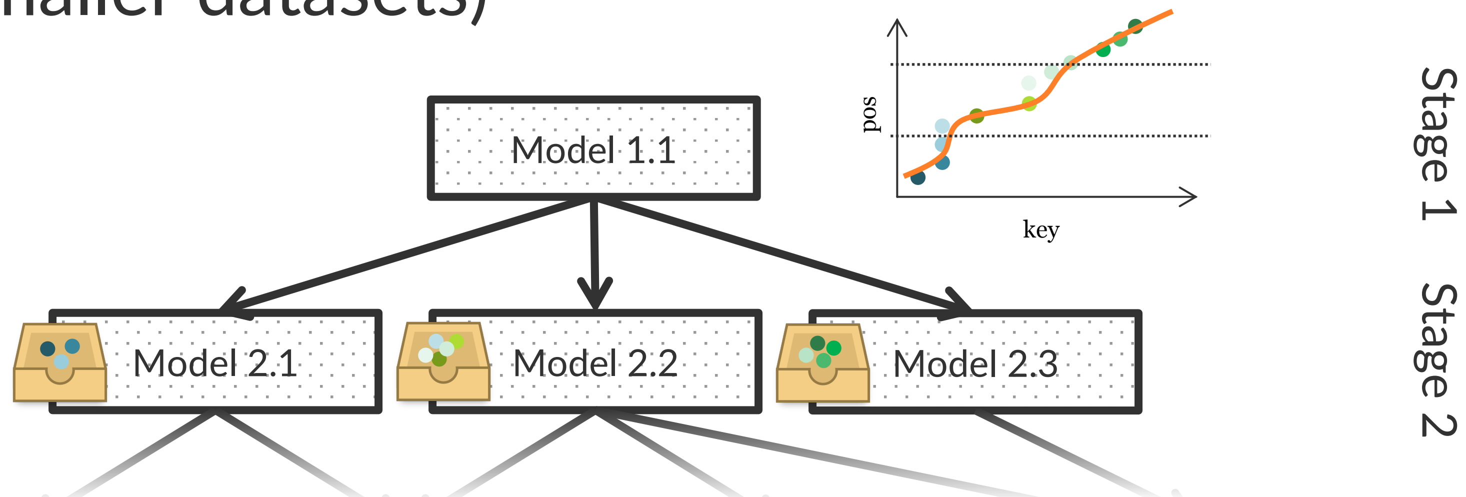


$key \in [pos - err, pos + err]$?

Construction of RMI

~

1. Train the root model on the dataset
2. Use it to distribute keys to the next stage
3. Repeat for each model in the next stage (on smaller datasets)



MIT CSAIL + Google

Performance of RMI

~

1. Up to 1.5–3x faster and two orders of magnitude smaller in space than a B^+ tree
2. Unfair? Definitely
3. GPUs/TPUs? Not really…

dawn.cs.stanford.edu/2018/01/11/index-bas

STANFORD DAWN

Home Vision Slides People Members Projects Papers Seminar Blog

Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes

by Peter Bailis, Kai Sheng Tai, Pratiksha Thaker, and Matei Zaharia

11 Jan 2018

There's recently been a lot of excitement about a new proposal from authors at Google: to replace conventional indexing data structures like B-trees and hash maps by instead fitting a neural network to the dataset. The paper compares such learned indexes against several standard data structures and reports promising results. For range searches, the authors report up 3.2x speedups over B-trees while using 9x less memory, and for point lookups, the authors report up to 80% reduction of hash table memory overhead while maintaining a similar query time.

While learned indexes are an exciting idea for many reasons (e.g., they could enable self-tuning databases), there is a long literature of other optimized data structures to consider, so naturally researchers have been trying to see whether these can do better. For example, Thomas Neumann posted about using spline interpolation in a B-tree for range search and showed that this easy-to-implement strategy can be competitive with learned indexes. In this post, we examine a second use case in the

Limitations of RMI

~

1. Fixed structure with many hyperparameters

   # stages, # models in each stage, kinds of regression models

2. Training time
3. No a priori error guarantees

   Difficult to predict latencies

4. Models are agnostic to the power of models below

   Can result in underused models (waste of space)

Our approach:
The Piecewise Geometric Model Index

Main ingredients of the PGM-index

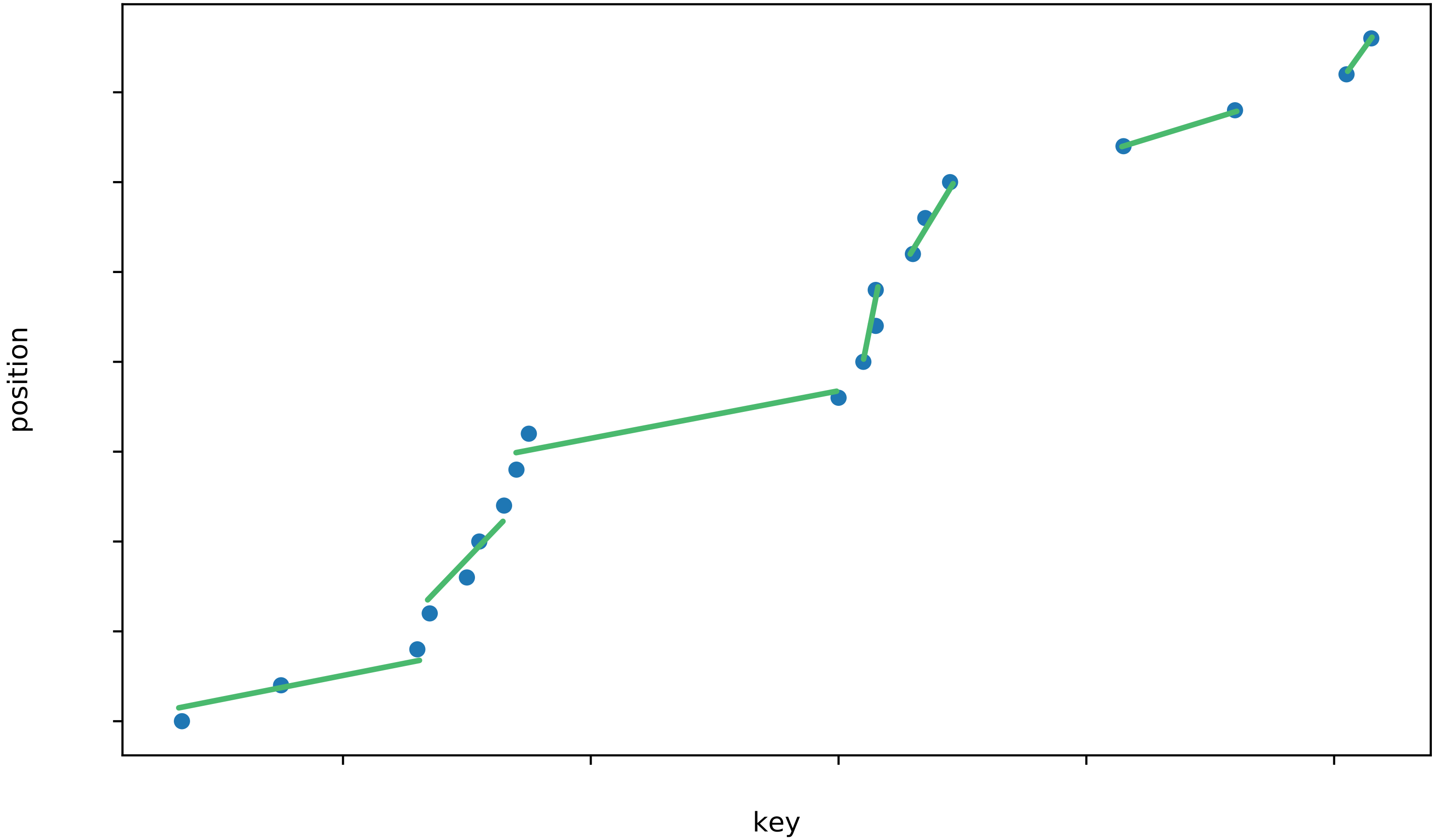
~

1. Fixed integer error $\varepsilon \geq 1$
2. Piecewise linear function: keys $\rightarrow$ positions
   a. Linear models are easy to store (2 floats)
   b. Linear models are fast (1 mul + 1 add)
3. Store models in the index nodes, not keys
4. Recursive bottom-up construction

PGM-index construction

~

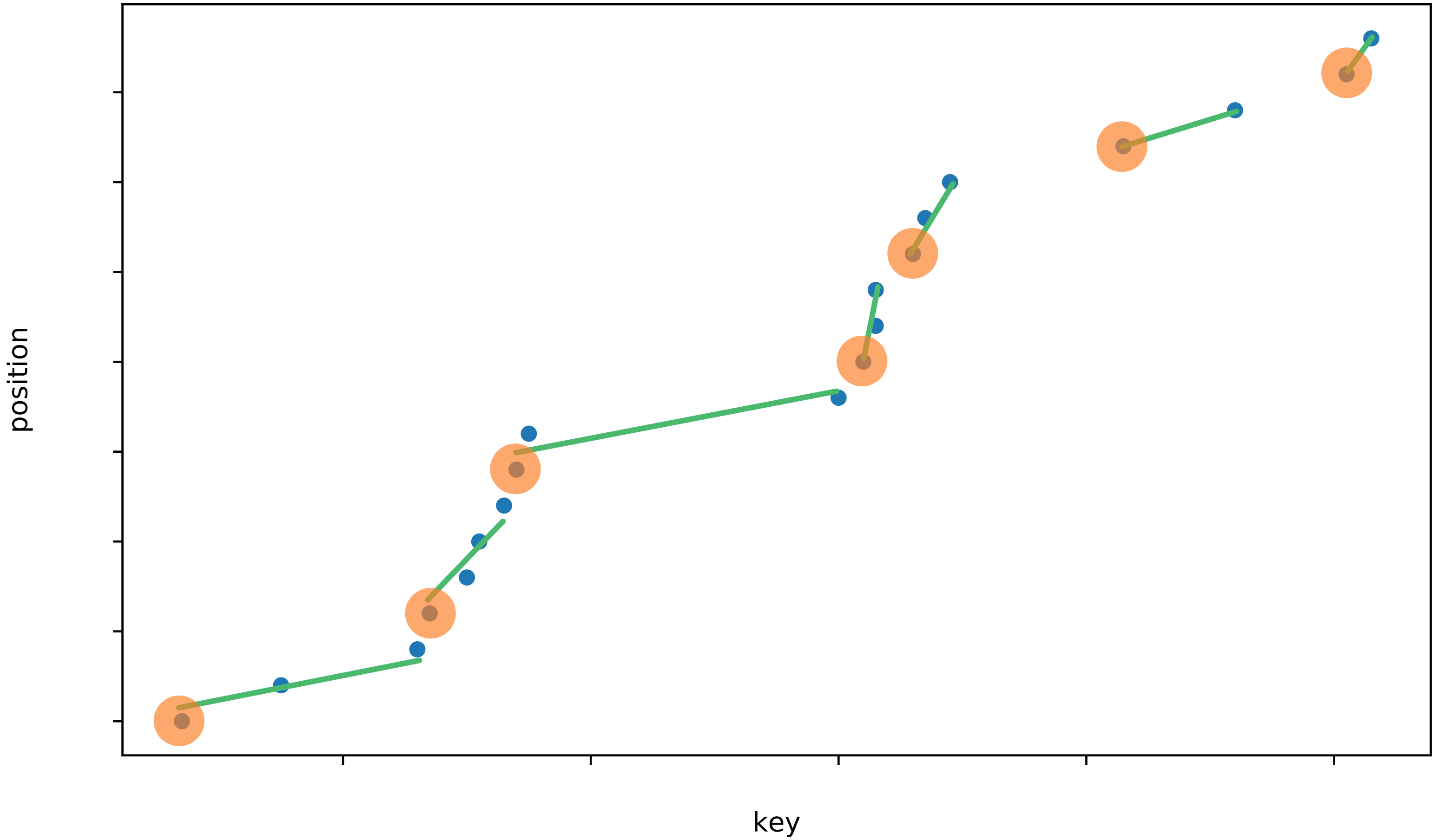
Compute the **optimal** piecewise linear approx with **guaranteed error** ε in $O(n)$



PGM-index construction

~

Save the m segments in a vector as triples $s_i = (key, slope, intercept)$



Memory layout of the PGM-index

~

Segments

(2, sl, ic)	(23, sl, ic)	(31, sl, ic)	(48, sl, ic)	(71, sl, ic)	(76, sl, ic)	(102, sl, ic)

1 m

Input keys

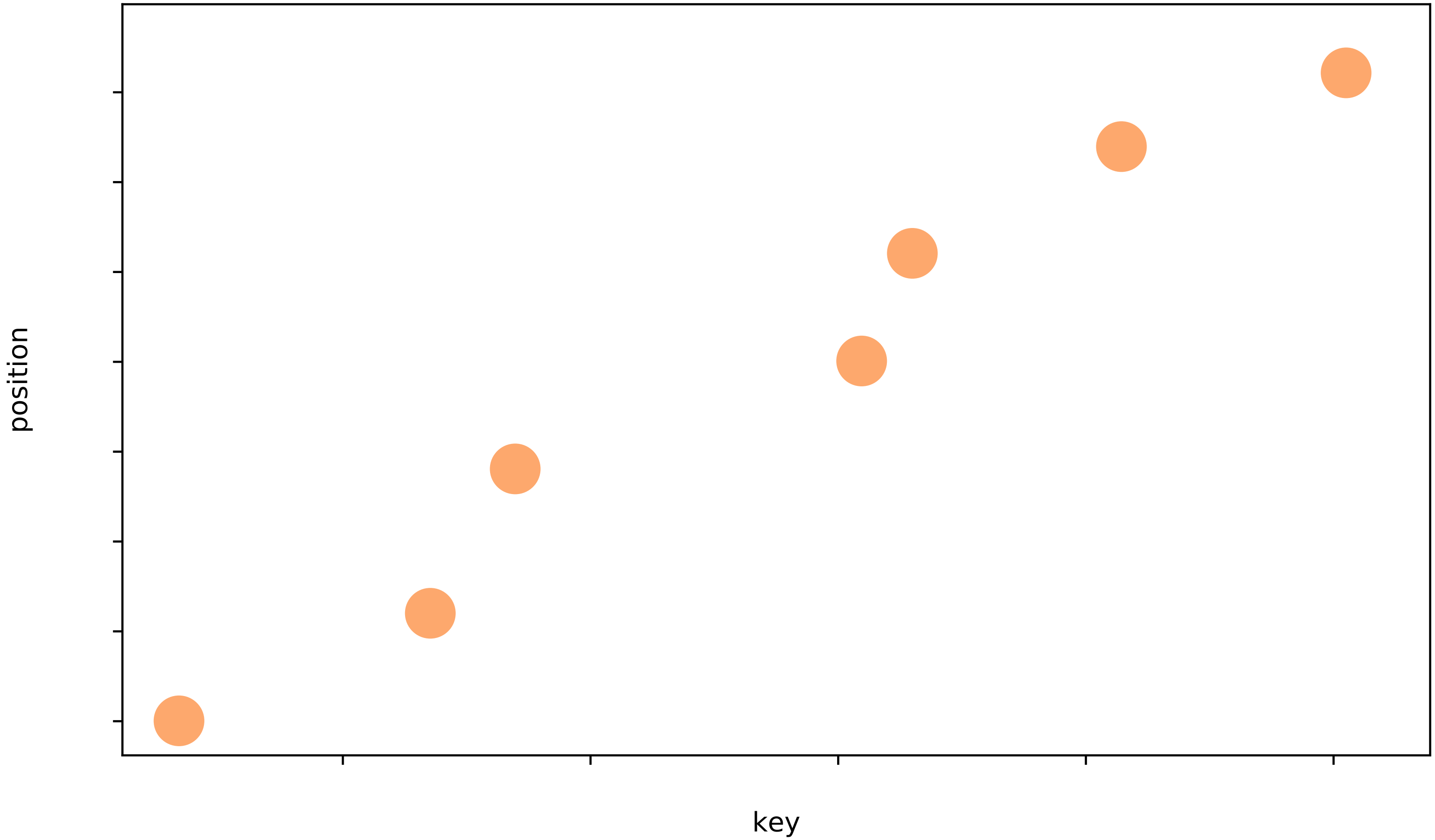
2	11	12	15	18	**23**	24	29	**31**	34	36	44	47	**48**	55	59	60	**71**	73	74	**76**	88	95	99	**102**	115	122	123

1 n

PGM-index construction

~

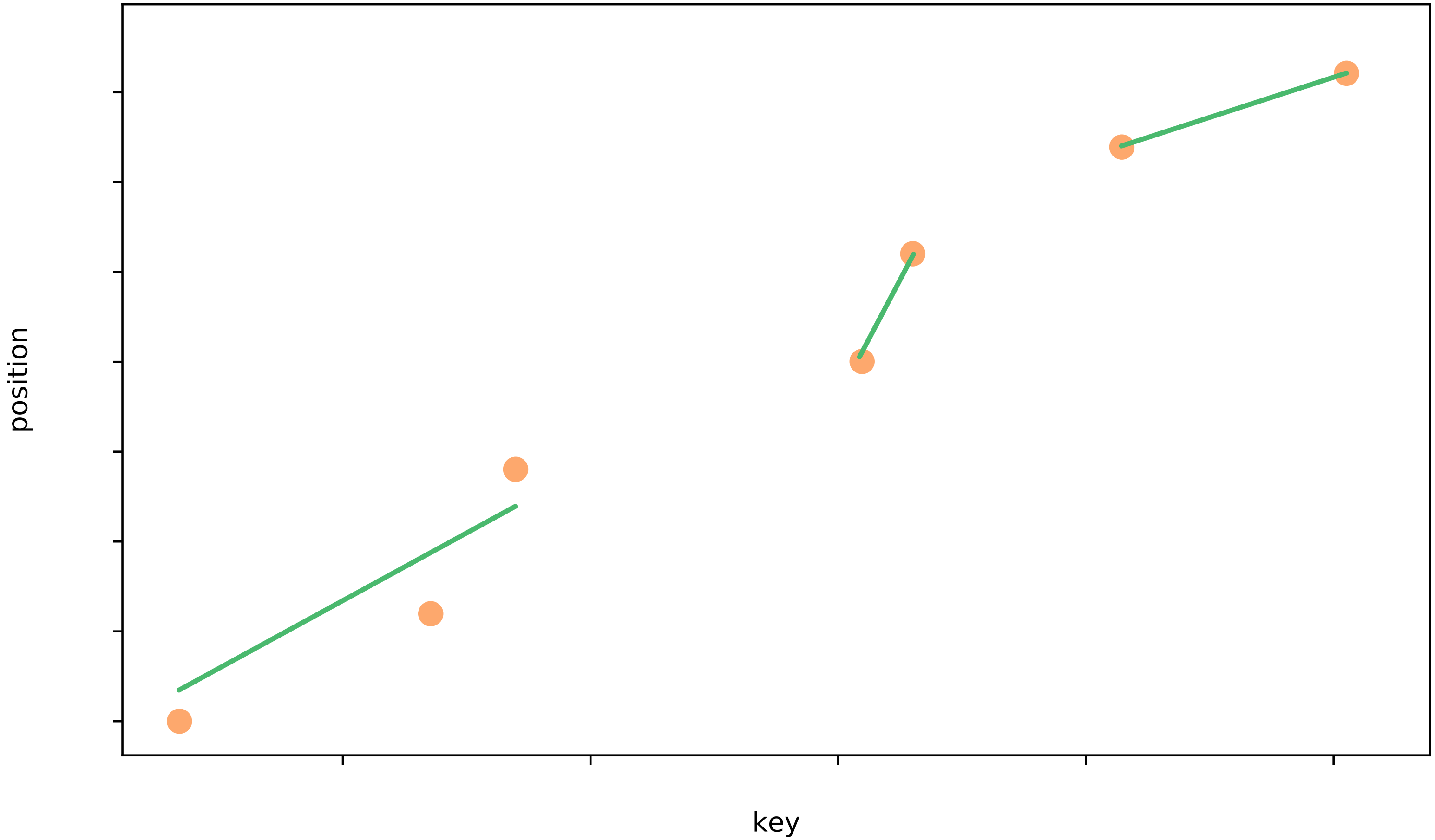
Drop all the points except $s_i.key$



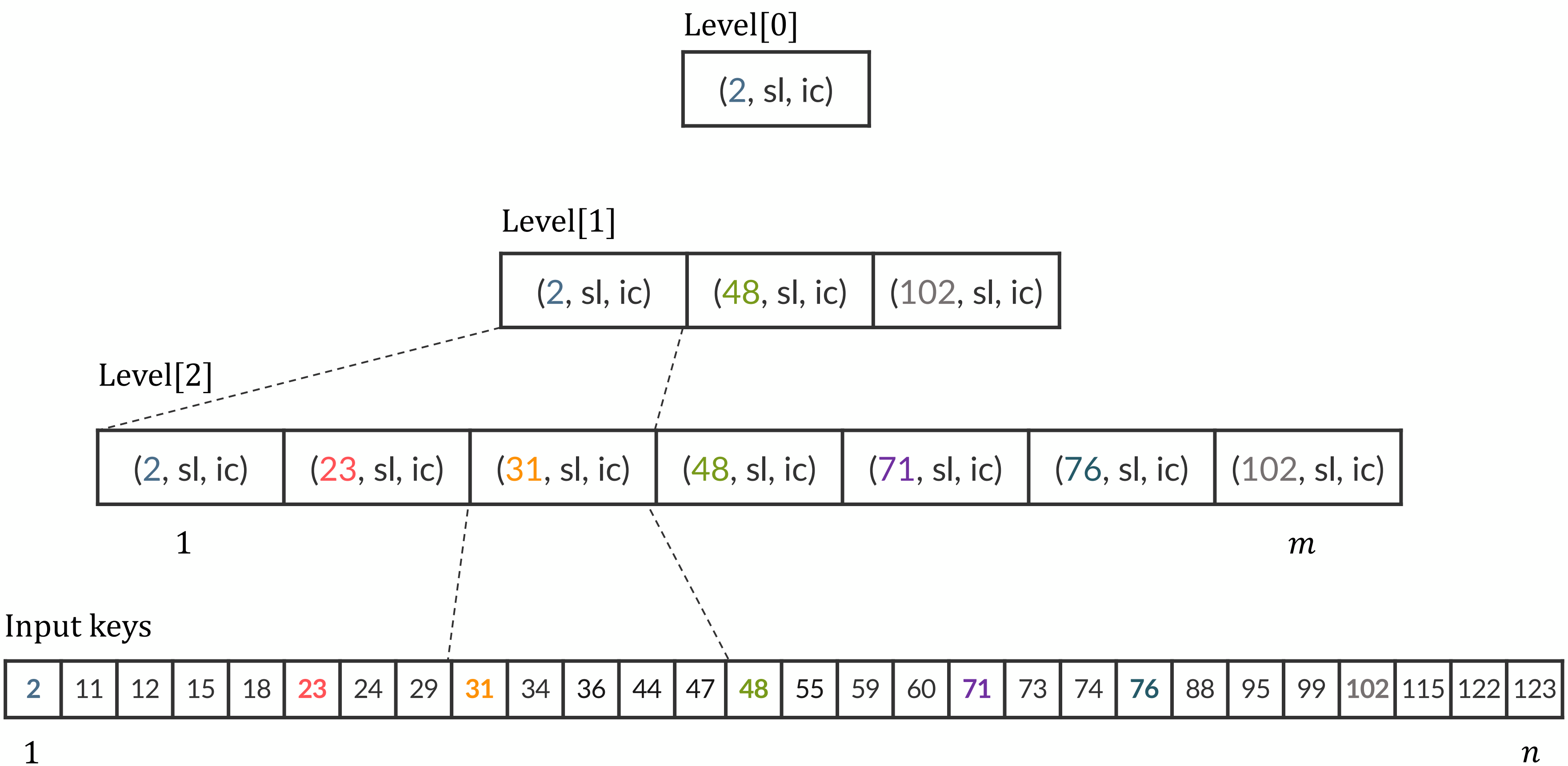
PGM-index construction

~

… and repeat!

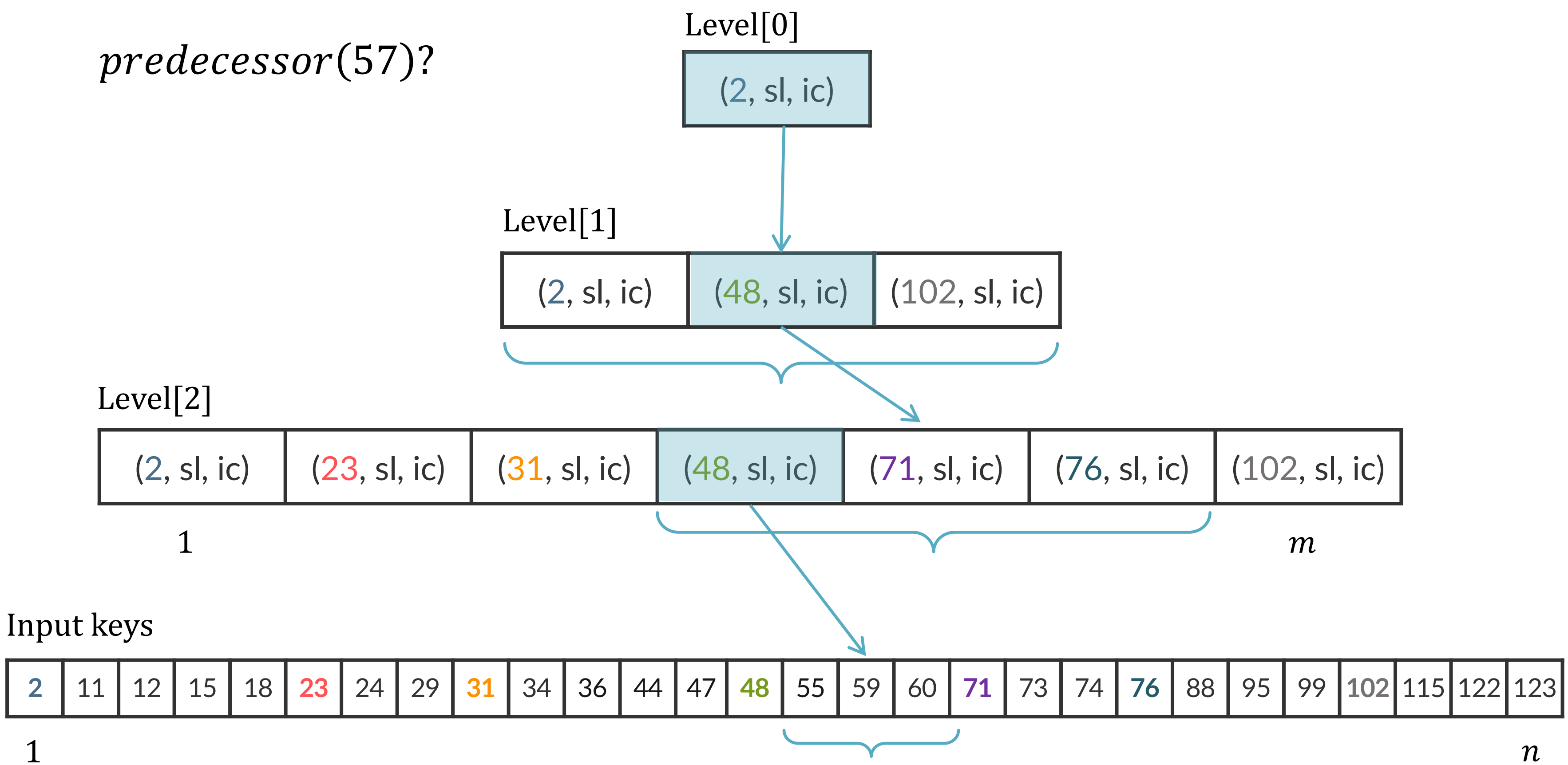


Memory layout of the PGM-index



Predecessor search in PGM-index w. $\varepsilon = 1$

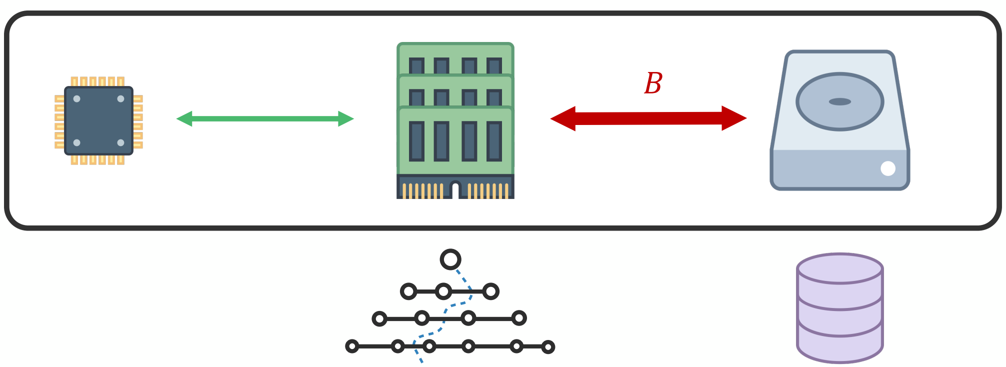
$predecessor(57)?$



Some asymptotic bounds

~

Data Structure	Space of index	RAM model Worst case time	EM model Worst case I/Os
Multiway tree	$\Theta(n)$	$O(\log n)$	$O(\log_B n)$
RMI	Fixed	$O(?)$	$O(?)$
PGM-index	$\Theta(m)$	$O(\log m)$ $m \leq n/(2\varepsilon)$	$O(\log_c m)$ $c \geq 2\varepsilon = \Omega(B)$



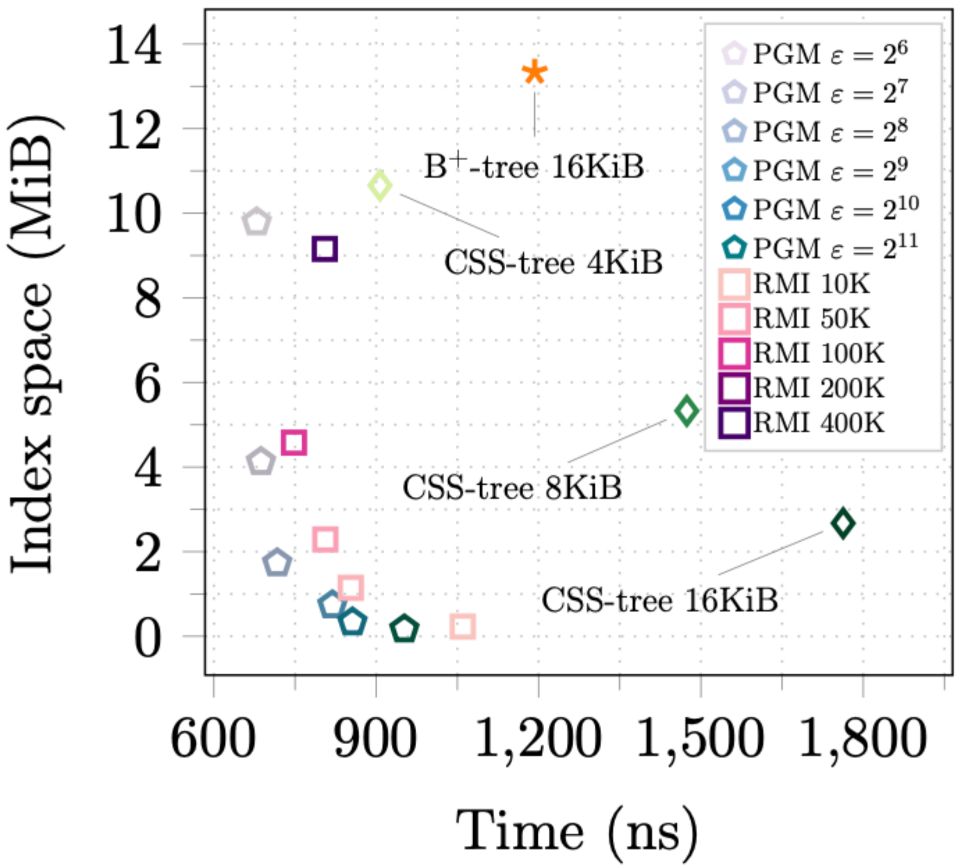
m segments, ε error

n keys

Space-time performance

~

2.3 GHz Intel Xeon Gold and 192 GiB memory



In a nutshell: indexing 95 GiB of data

~

Fastest CSS-tree

128 B pages (2× cache line)

341 MiB

1.2 s to construct

PGM-index with same performance

$\varepsilon = 128$

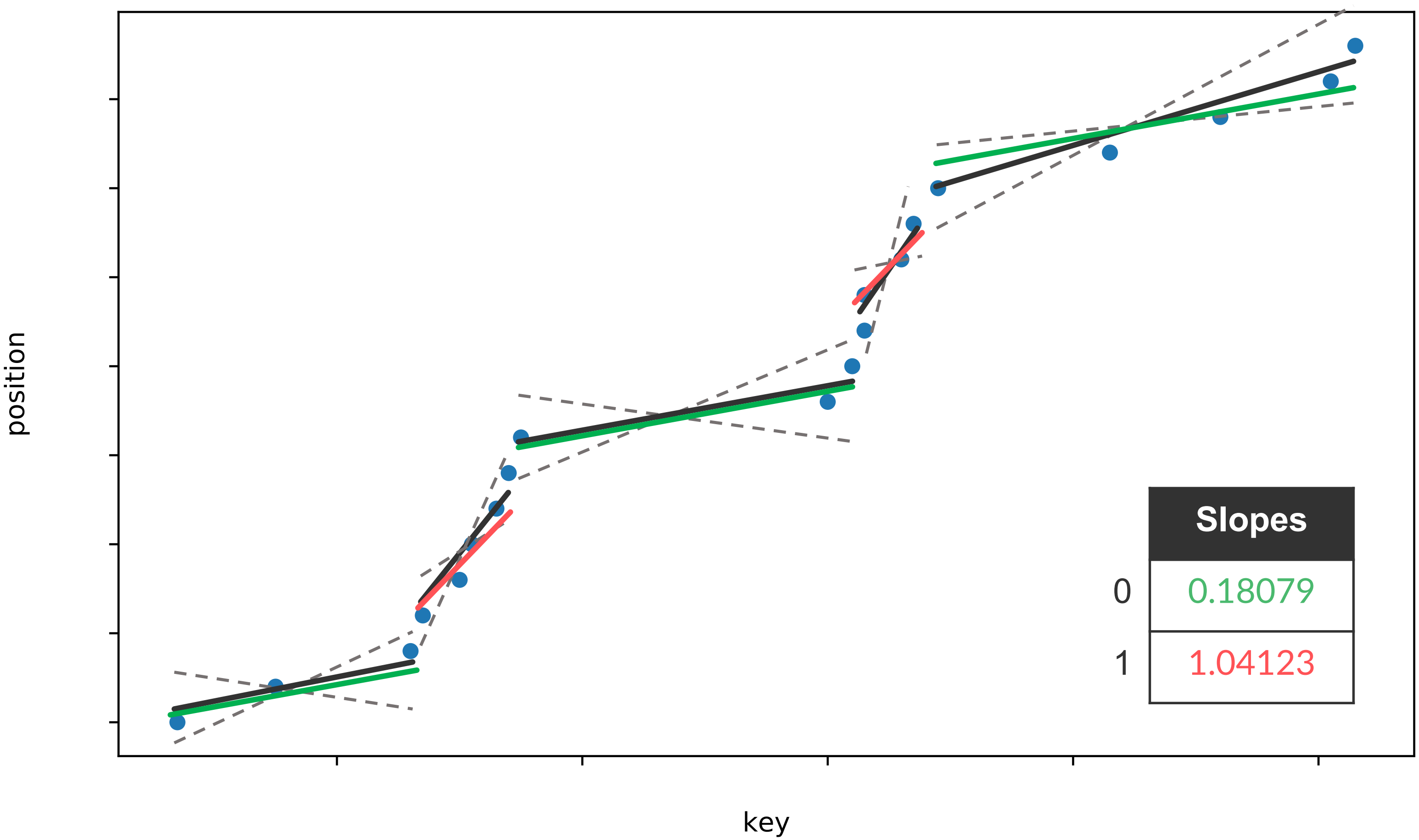
4 MiB (−85×)

2.1 s to construct

Compression
&
Distribution-awareness

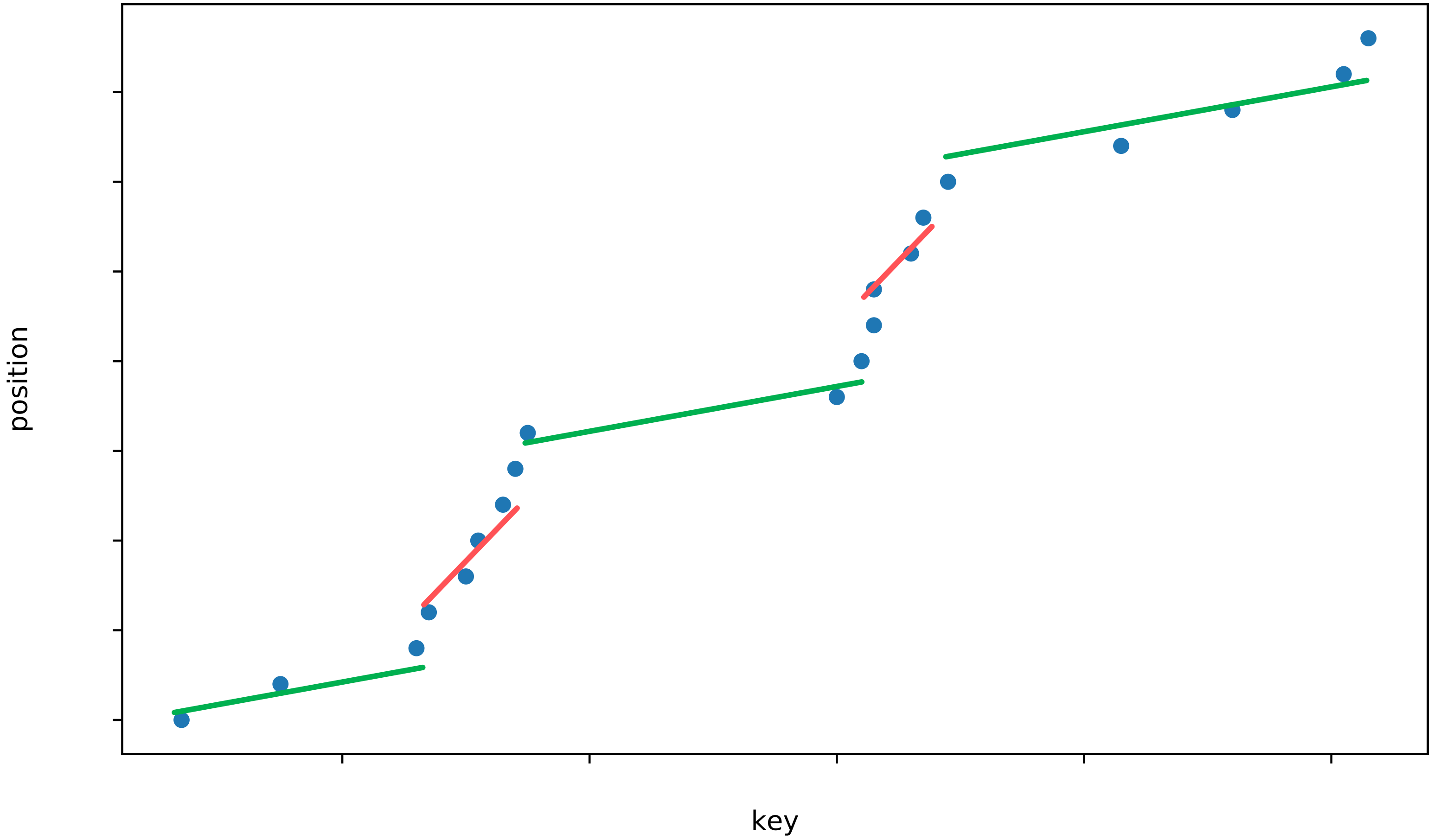
Compressed PGM-index: the slopes

~



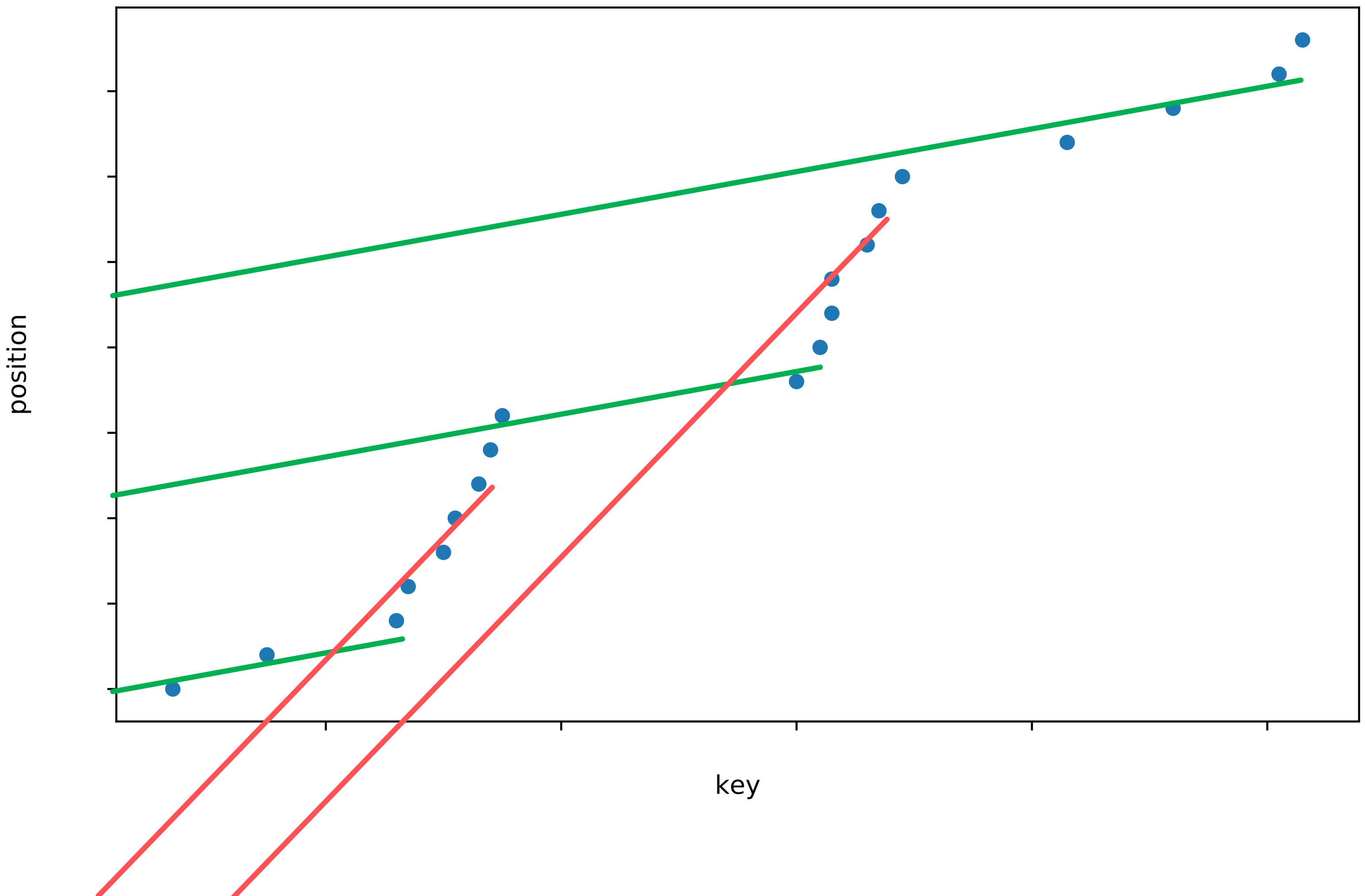
Compressed PGM-index: the intercepts

~

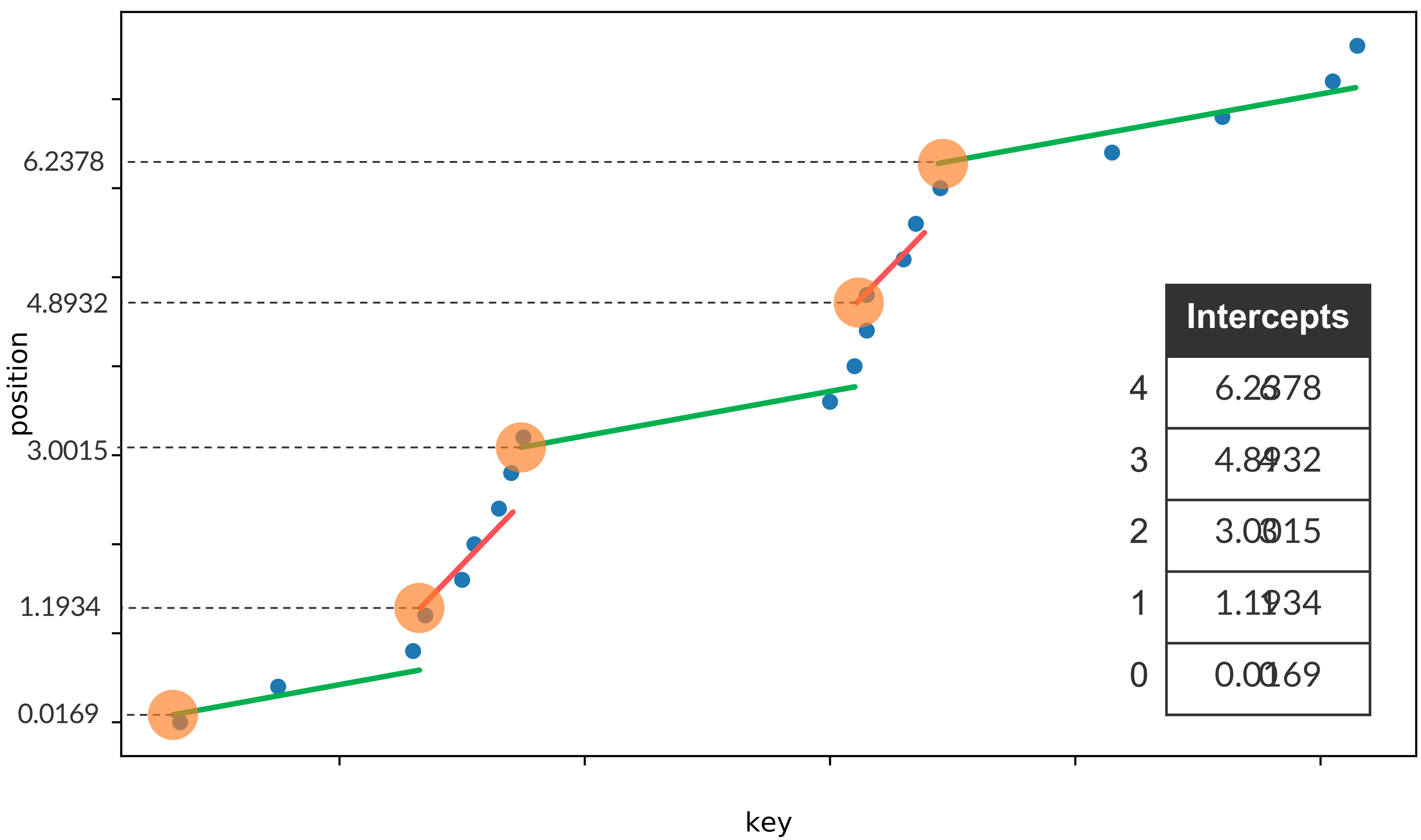


Compressed PGM-index: the intercepts

~



Compressed PGM-index: the intercepts



	Intercepts
4	6.2378
3	4.8932
2	3.0015
1	1.1934
0	0.0169

Compressed PGM-index

~

Theorem. Let m be the number of segments of a PGM-index indexing n keys

1. There exists a lossless compressor which computes the minimum number of distinct slopes $t \leq m$ and stores them in $64t + m\lceil \log t \rceil$ bits of space
2. The intercepts can be stored using $m \log(n/m) + 2m + o(m)$ bits and be randomly accessed in $O(1)$ time

Elias-Fano compressed index (§11.6 of the notes)

In practice queries are 14% slower but the space footprint is reduced by 52%

Distribution-aware predecessor problem

~

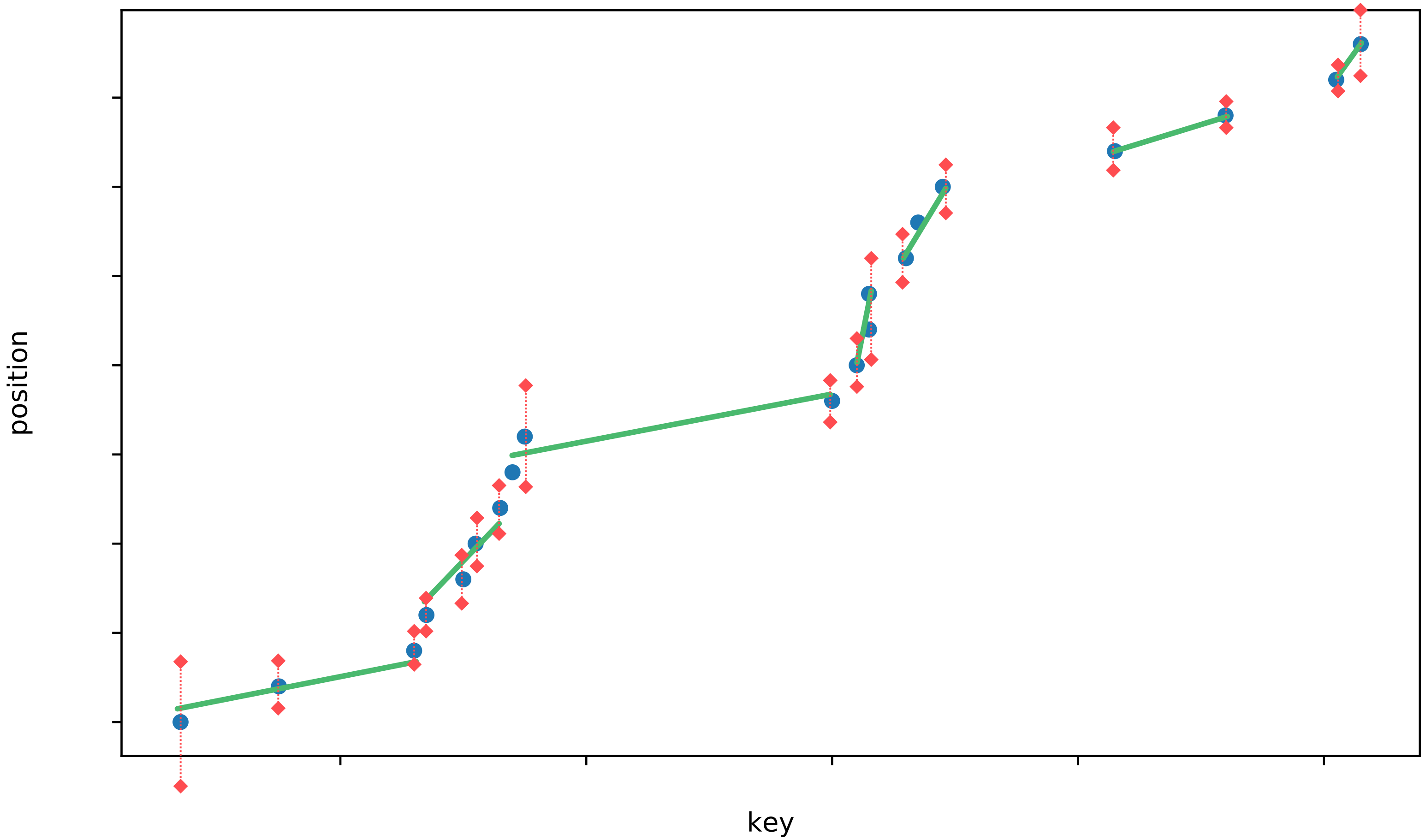
Given n pairs (k_i, p_i) where p_i is the probability of querying k_i, build a data structure that answer predecessor queries in $O(\log 1/p_i)$

Theorem. The Distribution-Aware PGM-index achieves that query time in $O(m)$ space, where m is the number of segments in the PGM-index

Distribution-aware PGM-index

~

Proof idea: define for the key k_i a range of size $\min\{1/p_i\,, \varepsilon\}$

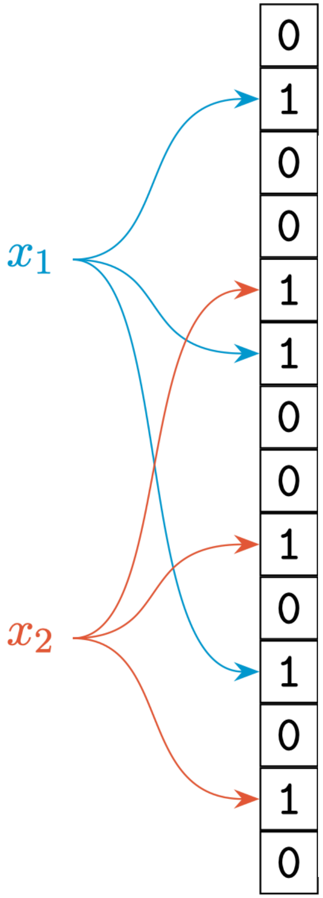


Learned approximate membership structures

Recap: the Bloom filter

~

- Bit vector of length m representing a set
  $$S = \{x_1, x_2, \dots, x_n\}$$
- r random independent hash functions $h_1, \dots, h_r$
- Initialise by setting $B[h_i(x)] = 1$ for any $x \in S, i \in [1, r]$
- Return that q is not in S when $B[h_i(q)] = 0$ for at least one i, otherwise return "maybe yes"



Ubiquitous Bloom filters

~

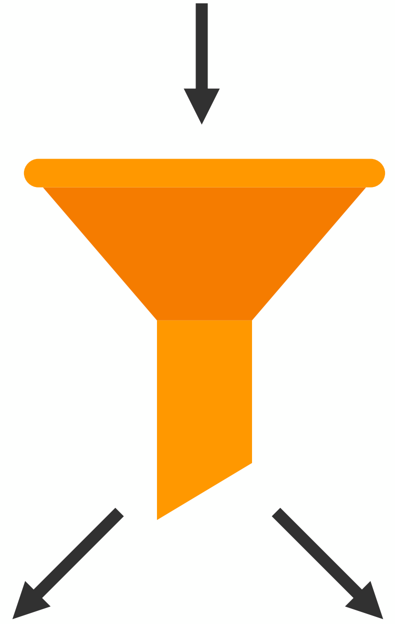
1. *Google Chrome*: identify malicious URLs.
2. *Akamai Technologies*: prevent "one-hit-wonders" from being stored in its disk caches.
3. *Google Bigtable, Apache HBase and Apache Cassandra and PostgreSQL*: reduce the disk lookups for non-existent rows or cols
4. *Medium*: avoid recommending articles a user has previously read

https://en.wikipedia.org/wiki/Bloom_filter#Examples

Bloom filter at a high level

~

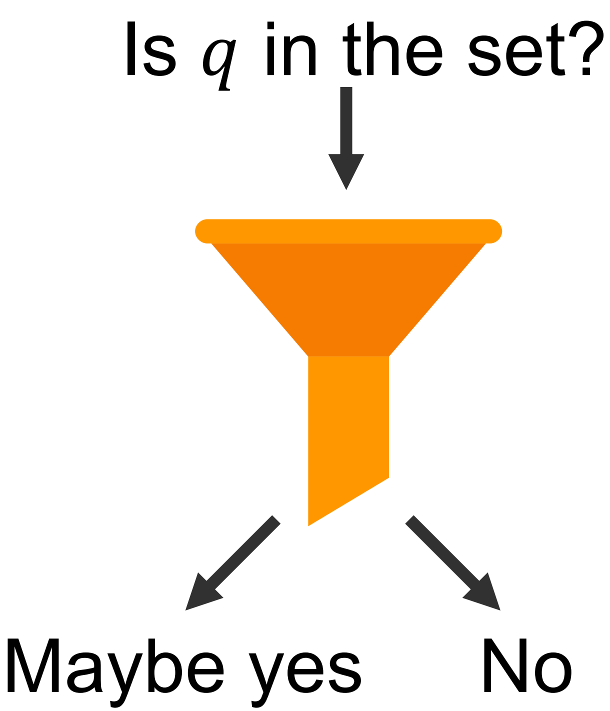
Is q in the set?

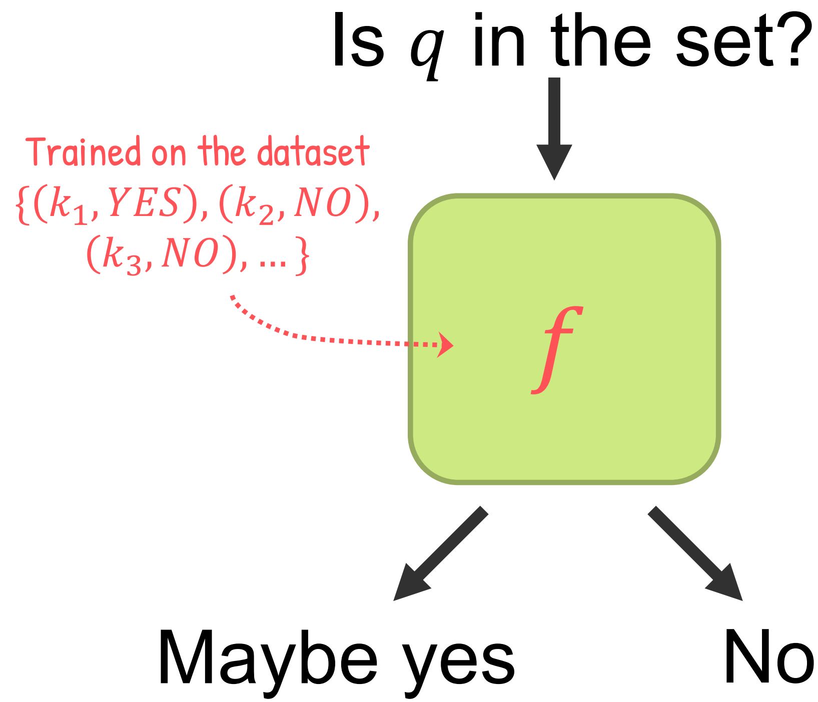


Maybe yes

No

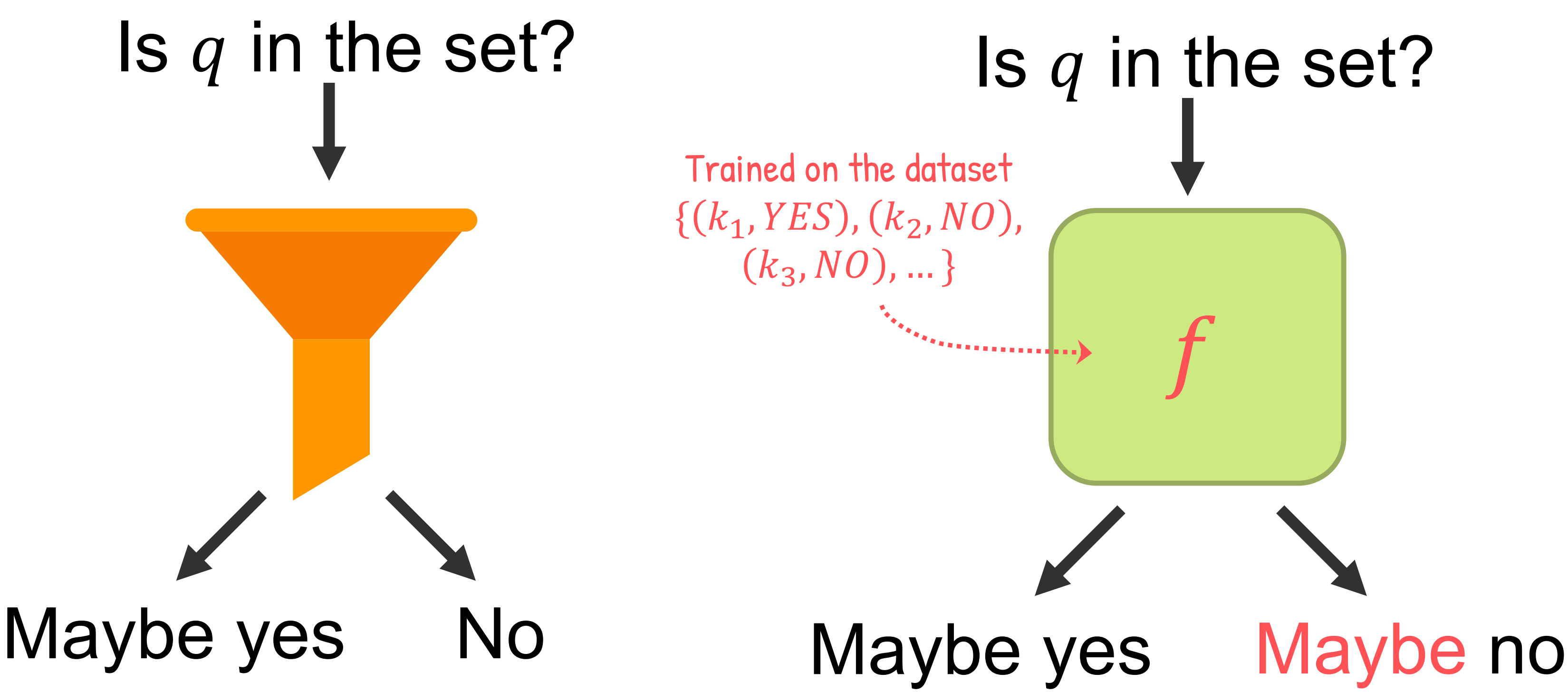
Bloom filter at a high level



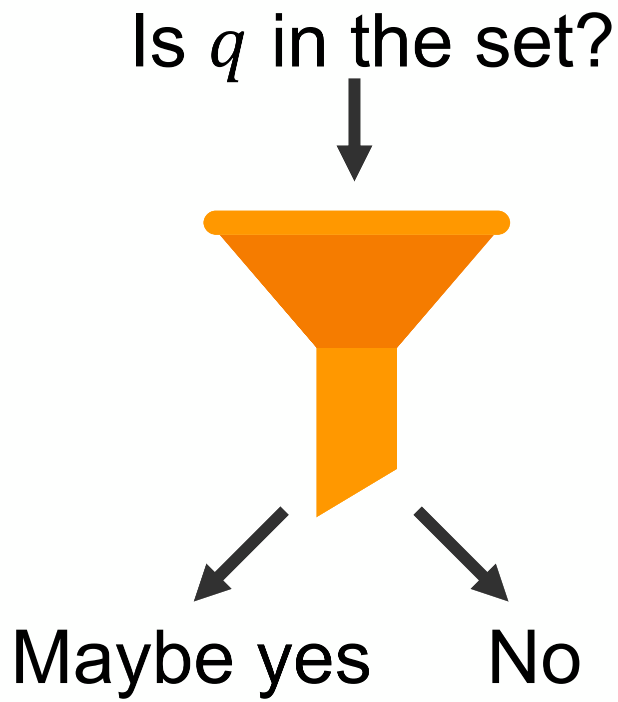


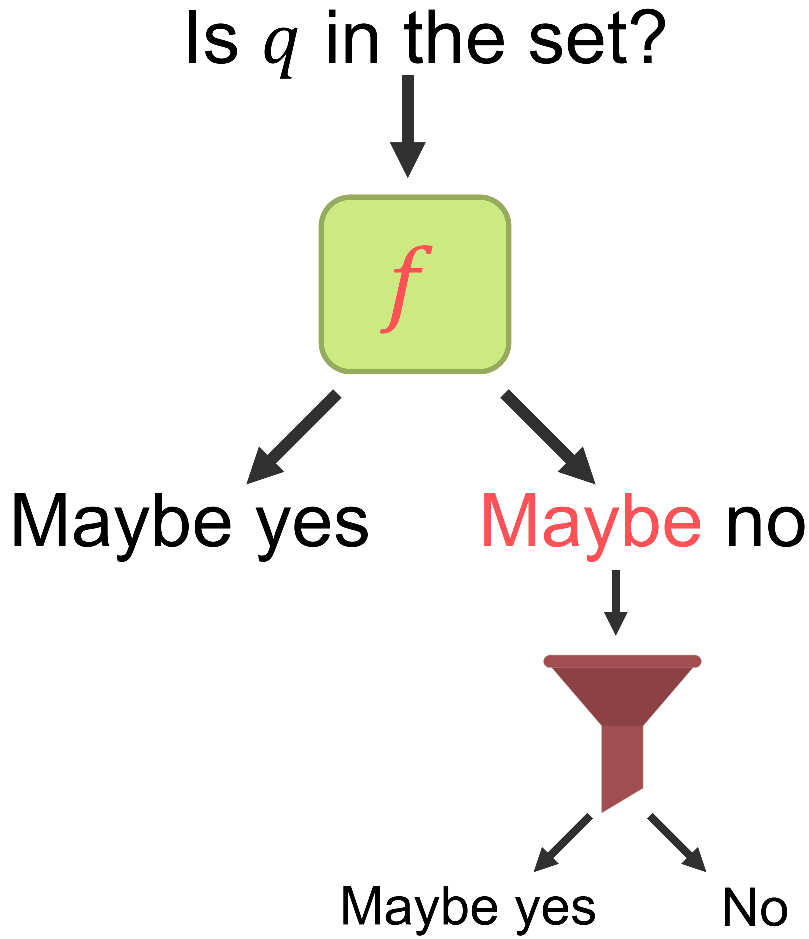
Bloom filter at a high level

~



Bloom filter at a high level





Learned vs classic Bloom filter

~

+ Reduces memory (~30%)
– Memory = MBs

+ No training
+ Fast evaluation
+ FPR on any non-set item

False positives != false positives

1. **Bloom filter**: the probability that *any* non-set item yields a false positive

$$\Pr(\text{all } r \text{ bits checked for a key not in S are 1}) \approx \left(1 - e^{rn/m}\right)^r$$

2. **Learned Bloom filter**: *measure* the false positive rate (FPR) over a test set and hope that future data looks like training data
   - Universe of elements is $\mathcal{U} = [0, 1\,000\,000)$
   - Store 500 elements *randomly* chosen from $R = [1000, 2000]$
   - Learned bloom filter might accept all keys from R and reject $\mathcal{U} \setminus R$
   - If test set (or future queries) is uniform over $\mathcal{U}$, then FPR will be low
   - If test set consists of elements in [1, 10 000], then FPR will be high

Many more learned algorithms and data structures

~

1. Learned hash tables
2. Learned sorting
3. Learned scheduler
4. Cardinality estimation of SQL queries
5. Frequency estimation for streams
6. …

Conclusions

To sum up

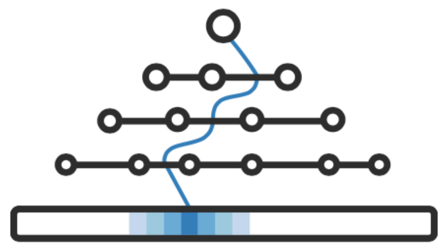
~

1. Thinking of classic data structure as ML models can bring several advantages
    a. Adaptation to the data distribution
    b. Better space-time performance
2. But those learned structures lose all the worst-case guarantees
    a. Unless you can prove they don't (PGM-index)
    b. Unless you just use them to augment, and not replace, classic structures

pgm.di.unipi.it

The PGM-index

Home Docs



The PGM-index

The ***Piecewise Geometric Model index*** (PGM-index) is a data structure that enables fast point and range searches in arrays of billions of items using orders of magnitude less space than traditional indexes.

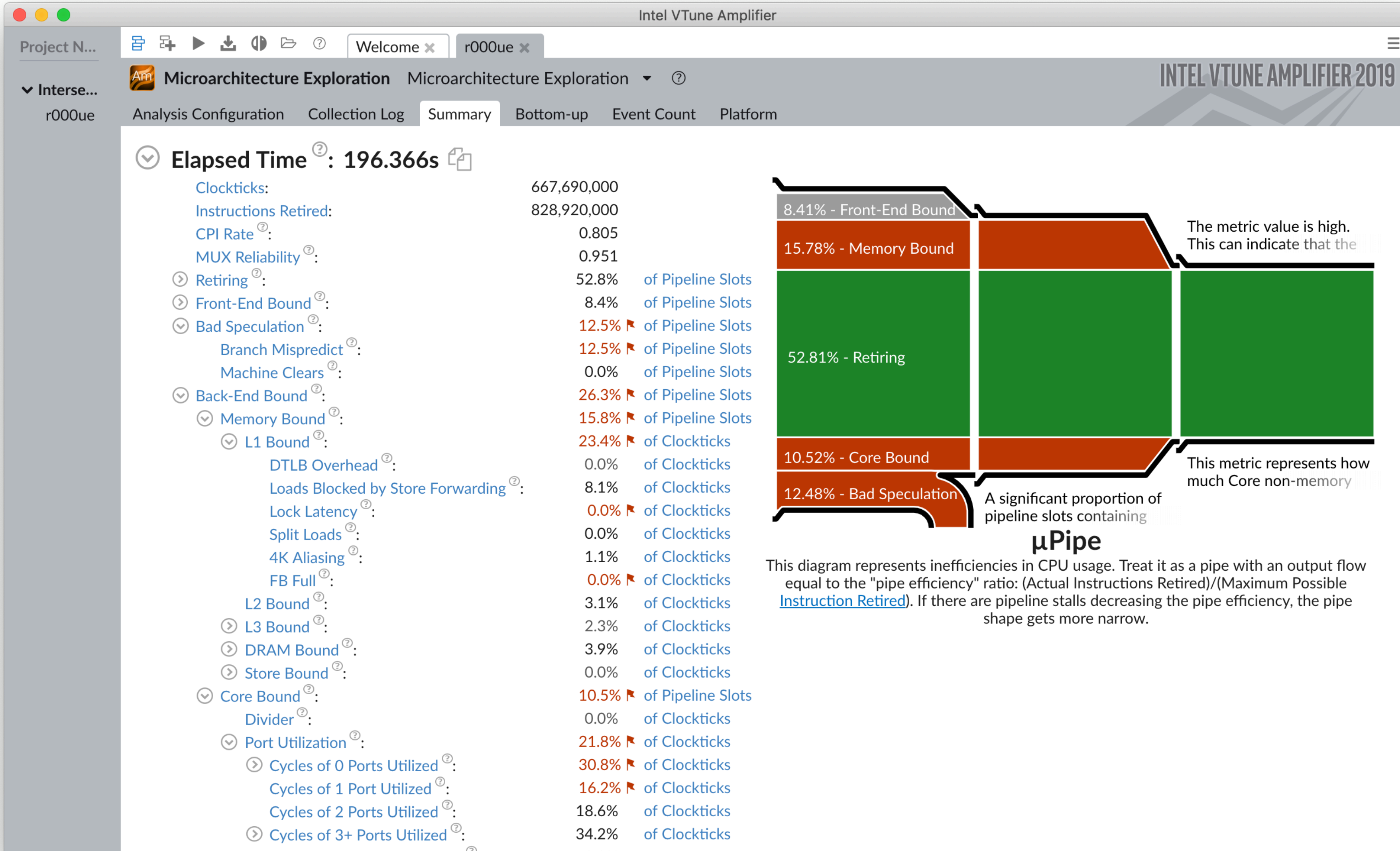
READ THE DOCS | VIEW ON GITHUB | GRAB THE PAPER

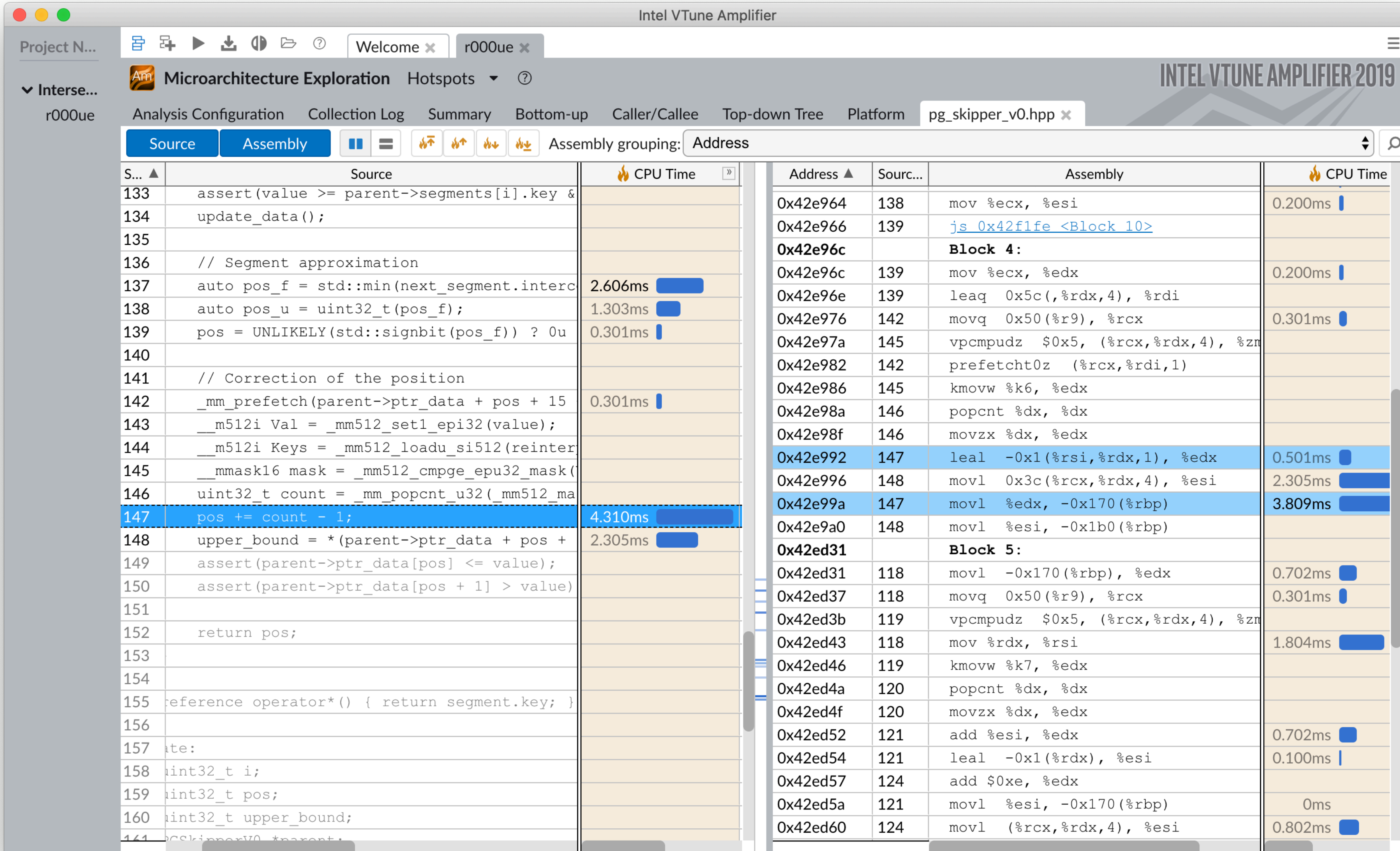
Reference

~

1. T. Kraska, A. Beutel, E.H. Chi, J. Dean, and N. Polyzotis. *The Case for Learned Index Structures*. In SIGMOD 2018.
2. M. Mitzenmacher. *A Model for Learned Bloom Filters, and Optimizing by Sandwiching*. In NeurIPS 2018.
3. P. Ferragina and G. Vinciguerra. *The PGM-index: a multicriteria, compressed and learned approach to data indexing*. Oct. 2019. arXiv: 1910.06169.

Extra slides


Intel VTune Amplifier
Project N...
Interse...
r000ue
Welcome
r000ue
Microarchitecture Exploration
Microarchitecture Exploration
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Event Count
Platform
Elapsed Time: 196.366s
Clockticks: 667,690,000
Instructions Retired: 828,920,000
CPI Rate: 0.805
MUX Reliability: 0.951
Retiring: 52.8% of Pipeline Slots
Front-End Bound: 8.4% of Pipeline Slots
Bad Speculation: 12.5% of Pipeline Slots
Branch Mispredict: 12.5% of Pipeline Slots
Machine Clears: 0.0% of Pipeline Slots
Back-End Bound: 26.3% of Pipeline Slots
Memory Bound: 15.8% of Pipeline Slots
L1 Bound: 23.4% of Clockticks
DTLB Overhead: 0.0% of Clockticks
Loads Blocked by Store Forwarding: 8.1% of Clockticks
Lock Latency: 0.0% of Clockticks
Split Loads: 0.0% of Clockticks
4K Aliasing: 1.1% of Clockticks
FB Full: 0.0% of Clockticks
L2 Bound: 3.1% of Clockticks
L3 Bound: 2.3% of Clockticks
DRAM Bound: 3.9% of Clockticks
Store Bound: 0.0% of Clockticks
Core Bound: 10.5% of Pipeline Slots
Divider: 0.0% of Clockticks
Port Utilization: 21.8% of Clockticks
Cycles of 0 Ports Utilized: 30.8% of Clockticks
Cycles of 1 Port Utilized: 16.2% of Clockticks
Cycles of 2 Ports Utilized: 18.6% of Clockticks
Cycles of 3+ Ports Utilized: 34.2% of Clockticks
8.41% - Front-End Bound
15.78% - Memory Bound
52.81% - Retiring
10.52% - Core Bound
12.48% - Bad Speculation
The metric value is high. This can indicate that the
This metric represents how much Core non-memory
A significant proportion of pipeline slots containing
µPipe
This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.


Intel VTune Amplifier
Project N...
Interse...
r000ue
Welcome
r000ue
Microarchitecture Exploration
Hotspots
INTEL VTUNE AMPLIFIER 2019
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
pg_skipper_v0.hpp
Source
Assembly
Assembly grouping:
Address
S...
Source
CPU Time
133 assert(value >= parent->segments[i].key &
134 update_data();
135
136 // Segment approximation
137 auto pos_f = std::min(next_segment.interc 2.606ms
138 auto pos_u = uint32_t(pos_f); 1.303ms
139 pos = UNLIKELY(std::signbit(pos_f)) ? 0u 0.301ms
140
141 // Correction of the position
142 _mm_prefetch(parent->ptr_data + pos + 15 0.301ms
143 __m512i Val = _mm512_set1_epi32(value);
144 __m512i Keys = _mm512_loadu_si512(reinter
145 __mmask16 mask = _mm512_cmpge_epu32_mask(
146 uint32_t count = _mm_popcnt_u32(_mm512_ma
147 pos += count - 1; 4.310ms
148 upper_bound = *(parent->ptr_data + pos + 2.305ms
149 assert(parent->ptr_data[pos] <= value);
150 assert(parent->ptr_data[pos + 1] > value)
151
152 return pos;
153
154
155 eference operator*() { return segment.key; }
156
157 te:
158 int32_t i;
159 int32_t pos;
160 int32_t upper_bound;
Address
Sourc...
Assembly
CPU Time
0x42e964 138 mov %ecx, %esi 0.200ms
0x42e966 139 js 0x42f1fe <Block 10>
0x42e96c Block 4:
0x42e96c 139 mov %ecx, %edx 0.200ms
0x42e96e 139 leaq 0x5c(,%rdx,4), %rdi
0x42e976 142 movq 0x50(%r9), %rcx 0.301ms
0x42e97a 145 vpcmpudz $0x5, (%rcx,%rdx,4), %zm
0x42e982 142 prefetcht0z (%rcx,%rdi,1)
0x42e986 145 kmovw %k6, %edx
0x42e98a 146 popcnt %dx, %dx
0x42e98f 146 movzx %dx, %edx
0x42e992 147 leal -0x1(%rsi,%rdx,1), %edx 0.501ms
0x42e996 148 movl 0x3c(%rcx,%rdx,4), %esi 2.305ms
0x42e99a 147 movl %edx, -0x170(%rbp) 3.809ms
0x42e9a0 148 movl %esi, -0x1b0(%rbp)
0x42ed31 Block 5:
0x42ed31 118 movl -0x170(%rbp), %edx 0.702ms
0x42ed37 118 movq 0x50(%r9), %rcx 0.301ms
0x42ed3b 119 vpcmpudz $0x5, (%rcx,%rdx,4), %zm
0x42ed43 118 mov %rdx, %rsi 1.804ms
0x42ed46 119 kmovw %k7, %edx
0x42ed4a 120 popcnt %dx, %dx
0x42ed4f 120 movzx %dx, %edx
0x42ed52 121 add %esi, %edx 0.702ms
0x42ed54 121 leal -0x1(%rdx), %esi 0.100ms
0x42ed57 124 add $0xe, %edx
0x42ed5a 121 movl %esi, -0x170(%rbp) 0ms
0x42ed60 124 movl (%rcx,%rdx,4), %esi 0.802ms

The `%timeit` built-in line magic

In [1]:

```
from random import uniform
from itertools import cycle

gen_point = lambda: (uniform(0, 100), uniform(0, 100))
points_pairs = [(gen_point(), gen_point()) for _ in range(100000)]
iter_points_pairs = cycle(points_pairs)
```

Cython

In [ ]:
```
!pip install cython
%load_ext Cython
```

In [5]:
```
%%cython -a

cimport libc.math

def cython_distance((double, double) p1, (double, double) p2):
    cdef double dx = p2[0] - p1[0]
    cdef double dy = p2[1] - p1[1]
    cdef double res = libc.math.sqrt(dx * dx + dy * dy)
    return res
```

The `%lprun` magic (from the `line_profiler` module)

In [ ]:

```
!pip install line_profiler
%load_ext line_profiler
```

In [9]:

```
%lprun -f distance.euclidean [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

```
Total time: 8.01555 s
File: /usr/local/miniconda3/lib/python3.6/site-packages/scipy/spatial/distance.py
Function: euclidean at line 566

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   566                                           def euclidean(u, v, w=None):
   567                                               """
   568                                               Computes the Euclidean distance between two 1-D arrays.
   569
   570                                               The Euclidean distance between 1-D arrays `u` and `v`, is defined as
   571
   572                                               .. math::
   573
   574                                                  {||u-v||}_2
   575
   576                                                  \\left(\\sum{(w_i |(u_i - v_i)|^2)}\\right)^{1/2}
   577
   578                                               Parameters
   579                                               ----------
   580                                               u : (N,) array_like
   581                                                   Input array.
   582                                               v : (N,) array_like
   583                                                   Input array.
   584                                               w : (N,) array_like, optional
   585                                                   The weights for each value in `u` and `v`. Default is None,
   586                                                   which gives each value a weight of 1.0
   587
   588                                               Returns
   589                                               -------
   590                                               euclidean : double
   591                                                   The Euclidean distance between vectors `u` and `v`.
   592
   593                                               Examples
   594                                               --------
   595                                               >>> from scipy.spatial import distance
   596                                               >>> distance.euclidean([1, 0, 0], [0, 1, 0])
   597                                               1.4142135623730951
   598                                               >>> distance.euclidean([1, 1, 0], [0, 1, 0])
   599                                               1.0
   600
   601                                               """
   602    100000    8015546.0     80.2    100.0      return minkowski(u, v, p=2, w=w)
```

The `%lprun` magic (from the `line_profiler` module)

In [ ]:
```
!pip install line_profiler
%load_ext line_profiler
```

In [9]:
```
%lprun -f distance.euclidean [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

In [10]:
```
%lprun -f distance.minkowski [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

```
   470                                                   minkowski : double
   471                                                       The Minkowski distance between vectors `u` and `v`.
   472
   473                                                   Examples
   474                                                   --------
   475                                                   >>> from scipy.spatial import distance
   476                                                   >>> distance.minkowski([1, 0, 0], [0, 1, 0], 1)
   477                                                   2.0
   478                                                   >>> distance.minkowski([1, 0, 0], [0, 1, 0], 2)
   479                                                   1.4142135623730951
   480                                                   >>> distance.minkowski([1, 0, 0], [0, 1, 0], 3)
   481                                                   1.2599210498948732
   482                                                   >>> distance.minkowski([1, 1, 0], [0, 1, 0], 1)
   483                                                   1.0
   484                                                   >>> distance.minkowski([1, 1, 0], [0, 1, 0], 2)
   485                                                   1.0
   486                                                   >>> distance.minkowski([1, 1, 0], [0, 1, 0], 3)
   487                                                   1.0
   488
   489                                                   """
   490    100000    1692341.0     16.9     18.8          u = _validate_vector(u)
   491    100000    1292432.0     12.9     14.4          v = _validate_vector(v)
   492    100000      93284.0      0.9      1.0          if p < 1:
   493                                                       raise ValueError("p must be at least 1")
   494    100000     403287.0      4.0      4.5          u_v = u - v
   495    100000      85169.0      0.9      0.9          if w is not None:
   496                                                       w = _validate_weights(w)
   497                                                       if p == 1:
   498                                                           root_w = w
   499                                                       if p == 2:
   500                                                           # better precision and speed
   501                                                           root_w = np.sqrt(w)
   502                                                       else:
   503                                                           root_w = np.power(w, 1/p)
   504                                                       u_v = root_w * u_v
   505    100000    5320230.0     53.2     59.2          dist = norm(u_v, ord=p)
   506    100000      99468.0      1.0      1.1          return dist
```