The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds

Paolo Ferragina

Giorgio Vinciguerra

Università di Pisa

The predecessor search problem

- Given n sorted input keys (e.g. integers), implement $predecessor(x)$ = “largest key $\leq x$”
- Range queries and joins in DBs, conjunctive queries in search engines, IP routing…
- Lookups alone are much easier; just use Cuckoo hashing for lookups at most 2 memory accesses (without sorting data!)

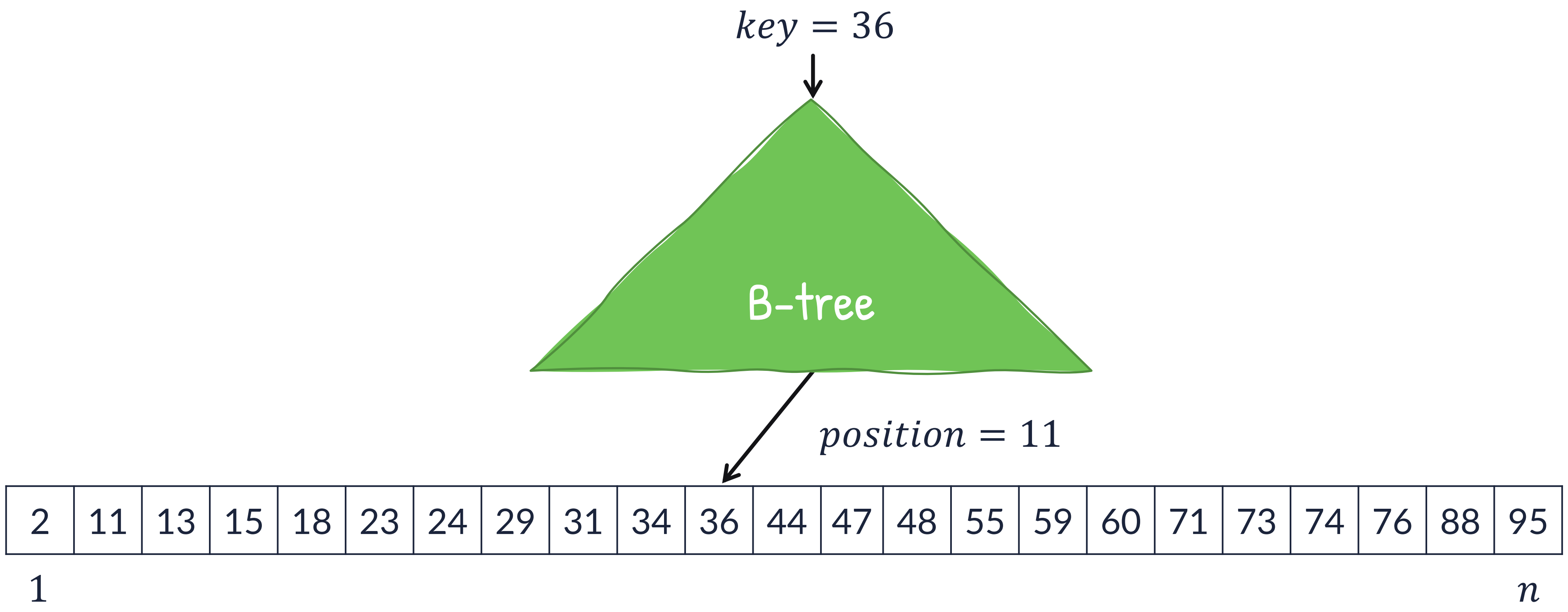
$predecessor(36) = 36$

2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95

1 $\qquad\qquad$ n

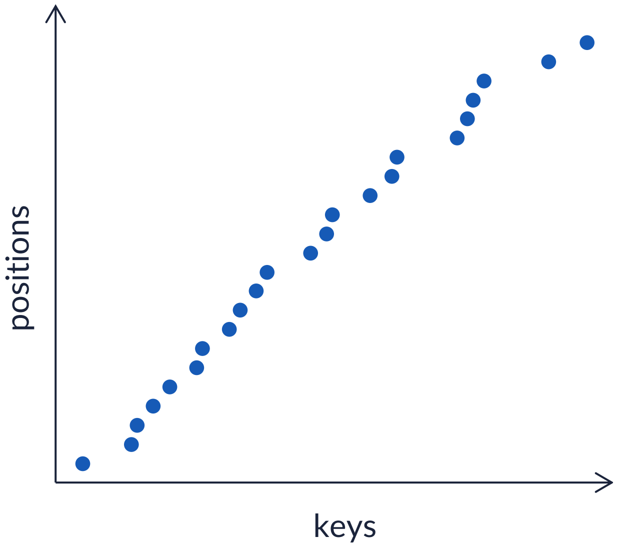
$predecessor(50) = 48$

Indexes



(values associated to keys are not shown)

Input data as pairs ($key, position$)

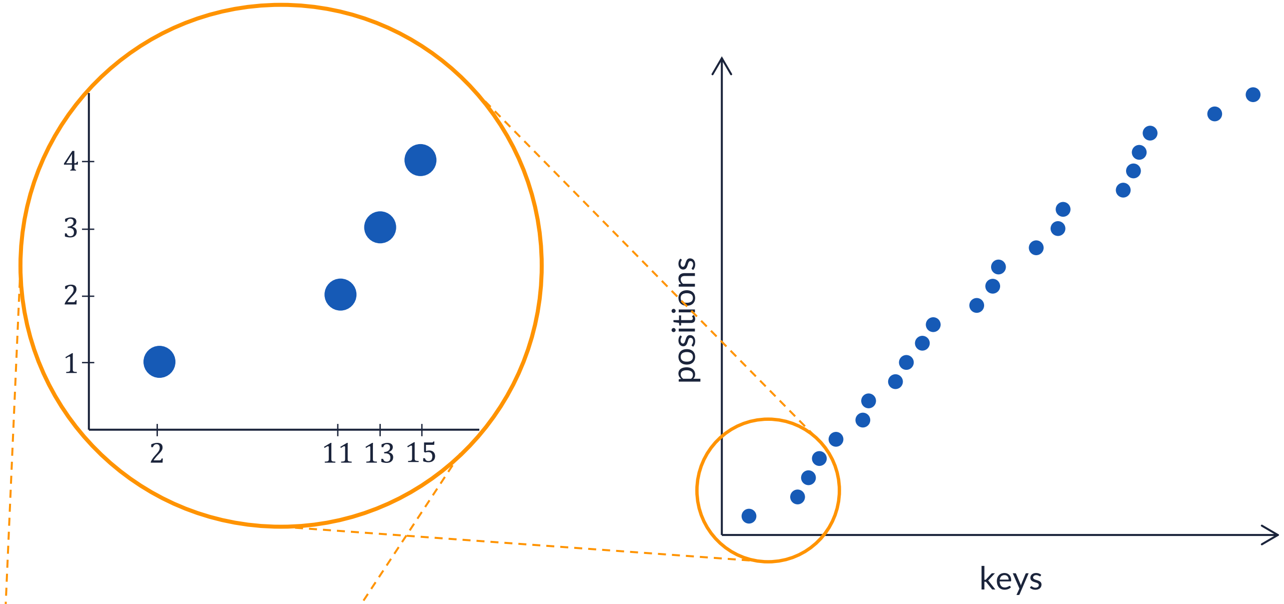


2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95

1 n

Ao et al. [VLDB 2011]

Input data as pairs $(key, position)$



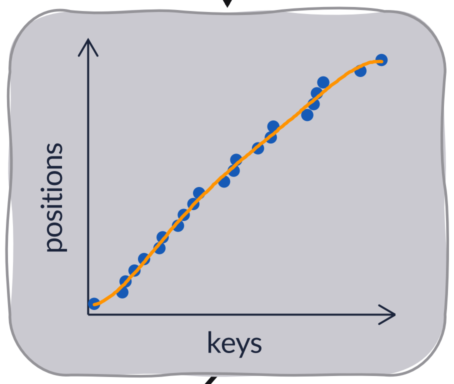
2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95
1	2	3	4																			n

Ao et al. [VLDB 2011]

Learned indexes

Black-box trained on a dataset of pairs (key, pos)
$\mathcal{D} = \{(2,1), (11,2), \dots, (95, n)\}$

key



$position$ (approximate)

2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95

1 n

Binary search in
$[position - error, position + error]$

Ao et al. [VLDB 2011], Kraska et al. [SIGMOD 2018]

The problem with learned indexes

Unpredictable latency

Too much I/O when data is on disk

Very slow to train

Fast query time and excellent space usage in practice, but no worst-case guarantees

Unscalable to big data

Must be tuned for each new dataset

Blind to the query distribution

Vulnerable to adversarial inputs and queries

Introducing the PGM-index

Predictable latency

Constant I/O when data is on disk

Very fast to build

Fast query time and excellent space usage in practice, and guaranteed worst-case bounds

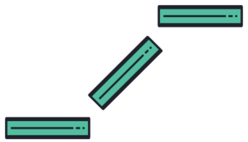
Scalable to big data

No additional tuning needed

Resistant to adversarial inputs and queries

Query distribution aware

Ingredients of the PGM-index

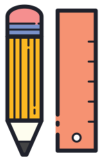


Opt. piecewise linear model

Fast to construct, best space usage
for linear learned indexes

Fixed model “error” ε

Control the size of the search range
(like the page size in a B-tree)

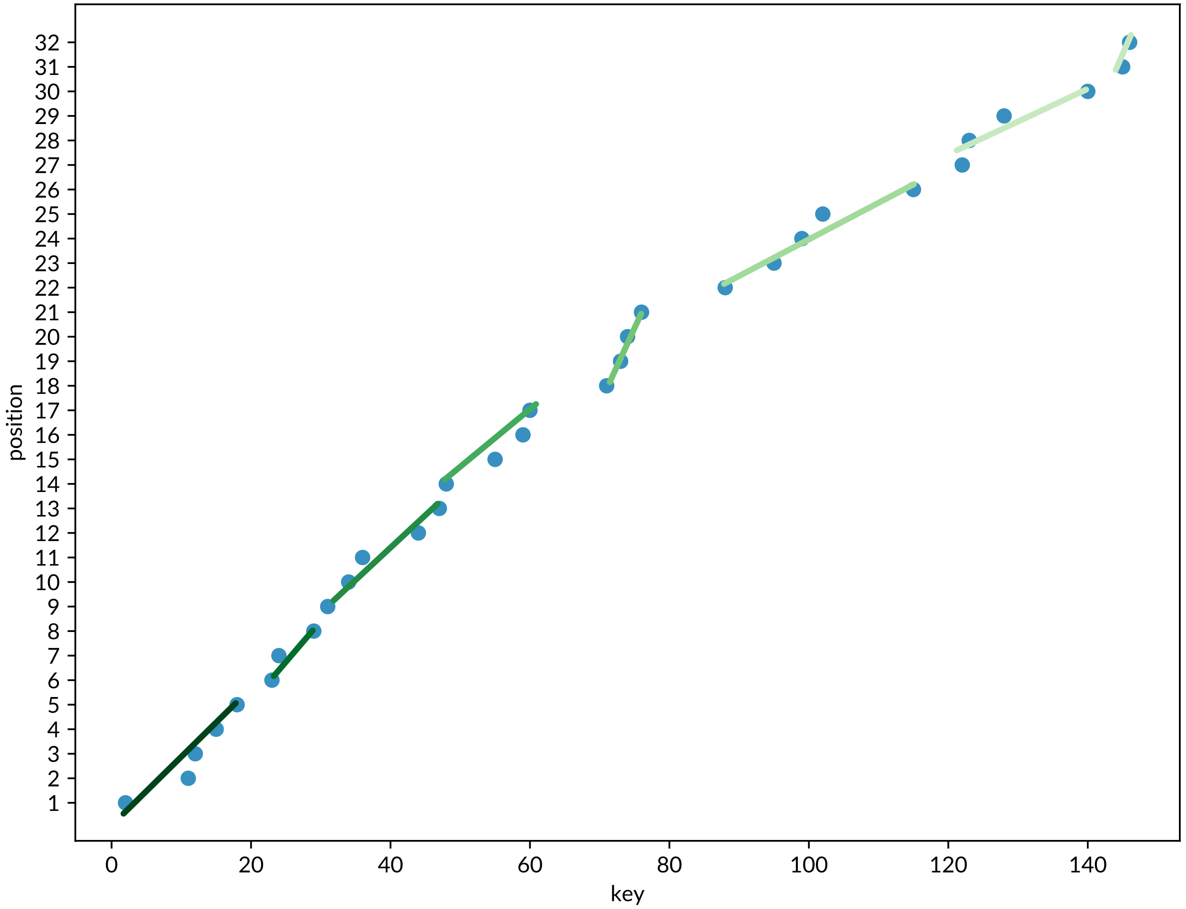


Recursive design

Adapt to the memory hierarchy
and enable query-time guarantees

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

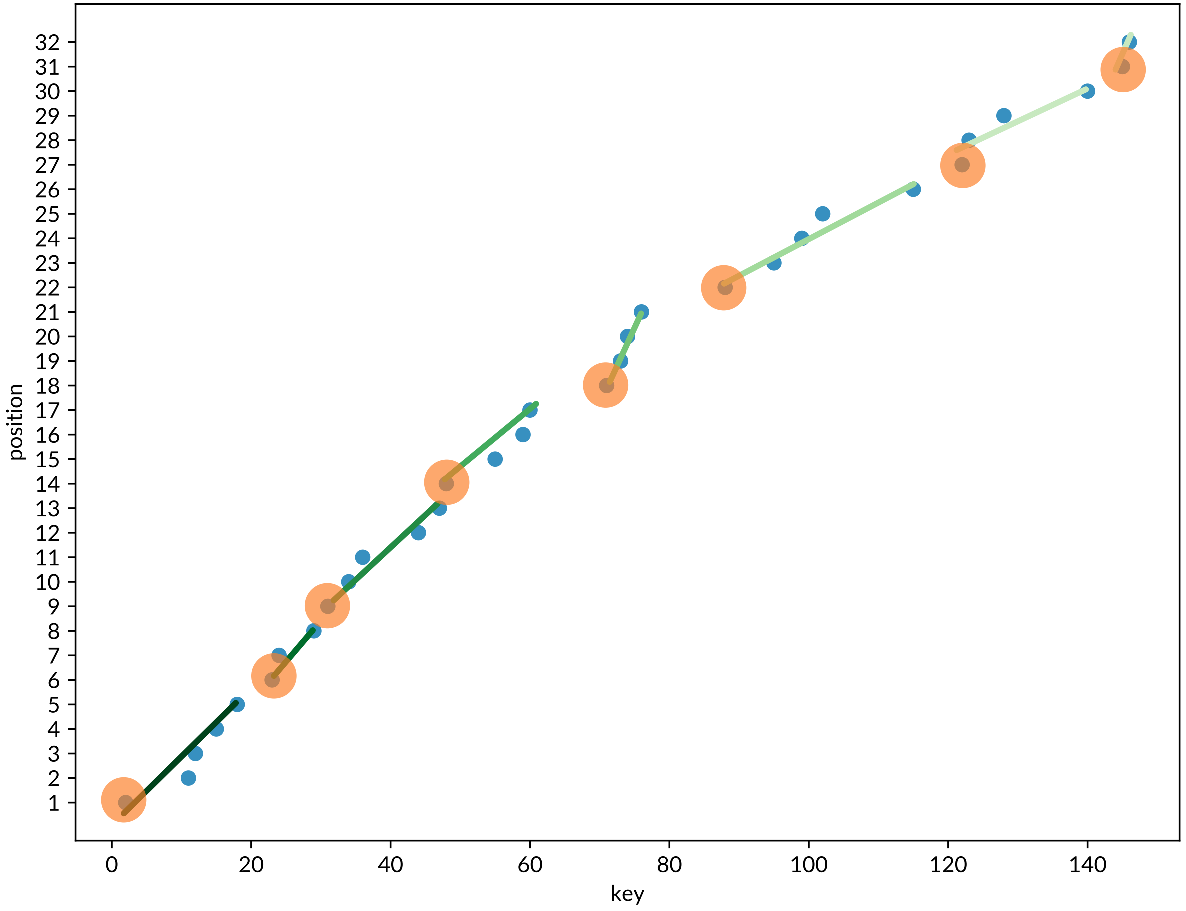


2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

1 n

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time



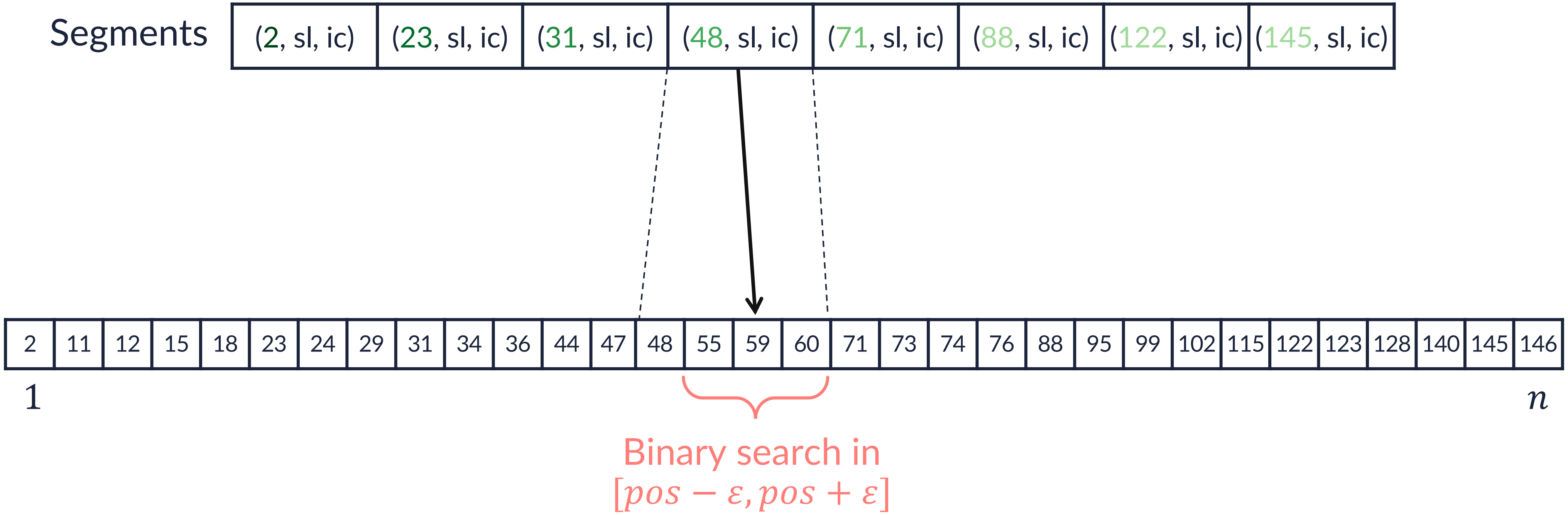
Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

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	128	140	145	146

1 n

Partial memory layout of the PGM-index

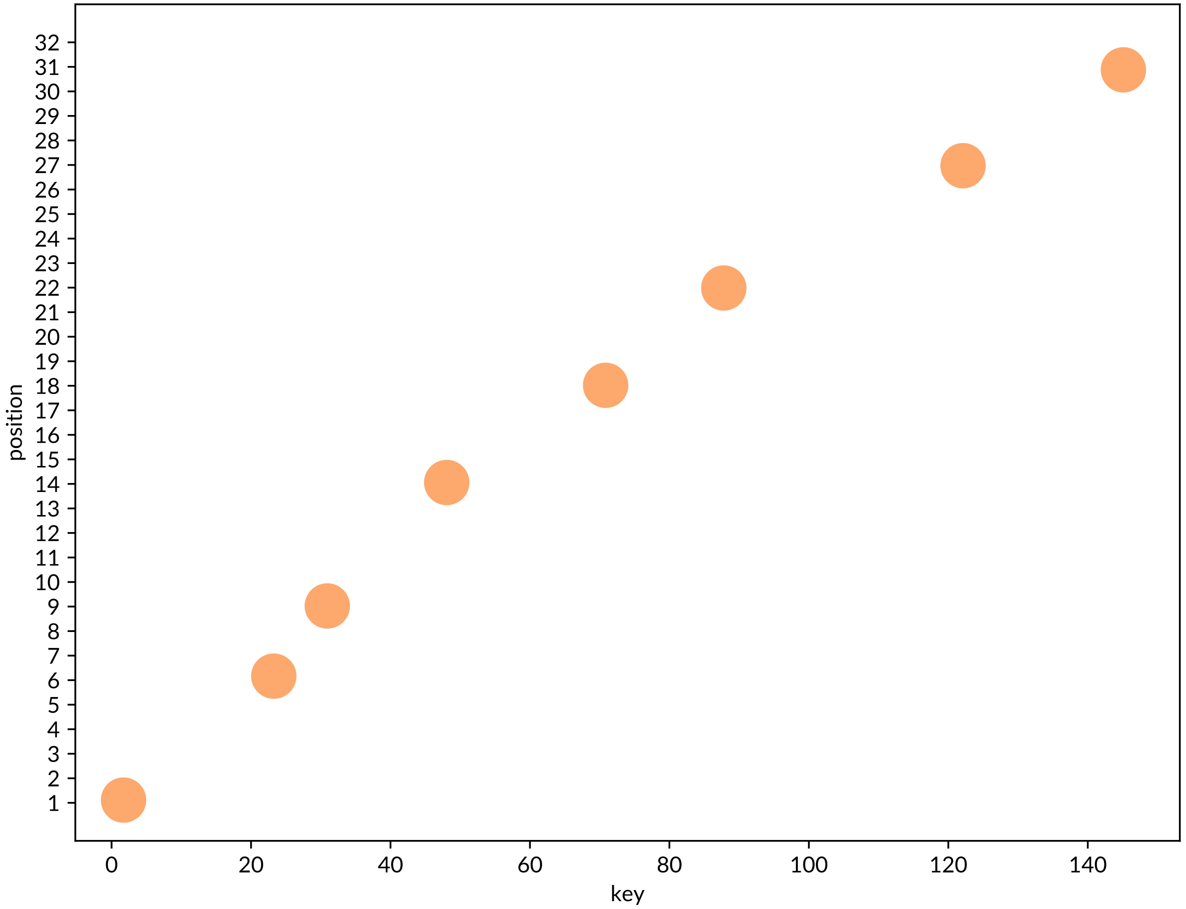
Each segment indexes a variable and potentially large sequence of keys
while guaranteeing a search range size of $2\varepsilon + 1$



PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 3. Keep only $s_i.key$



Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

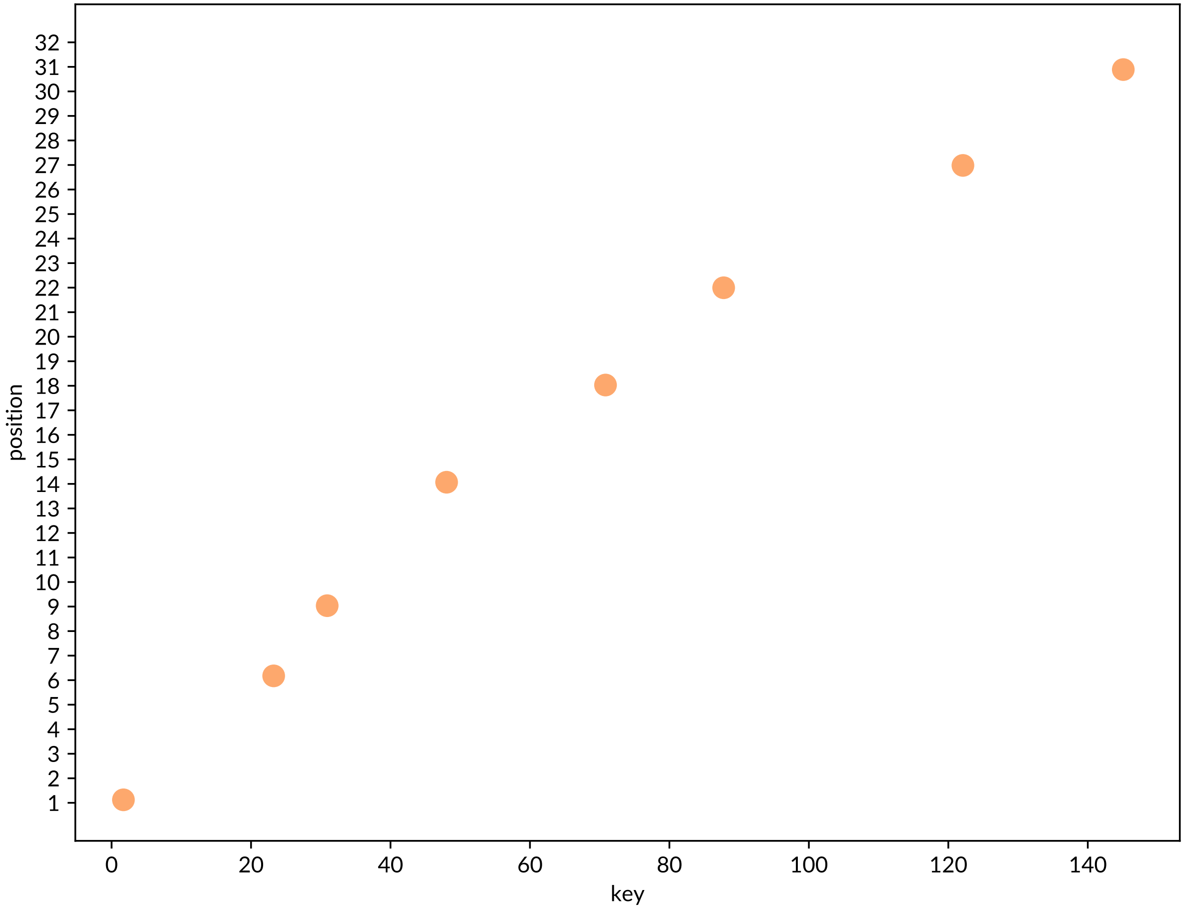
2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

1 n

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 3. Keep only $s_i.key$



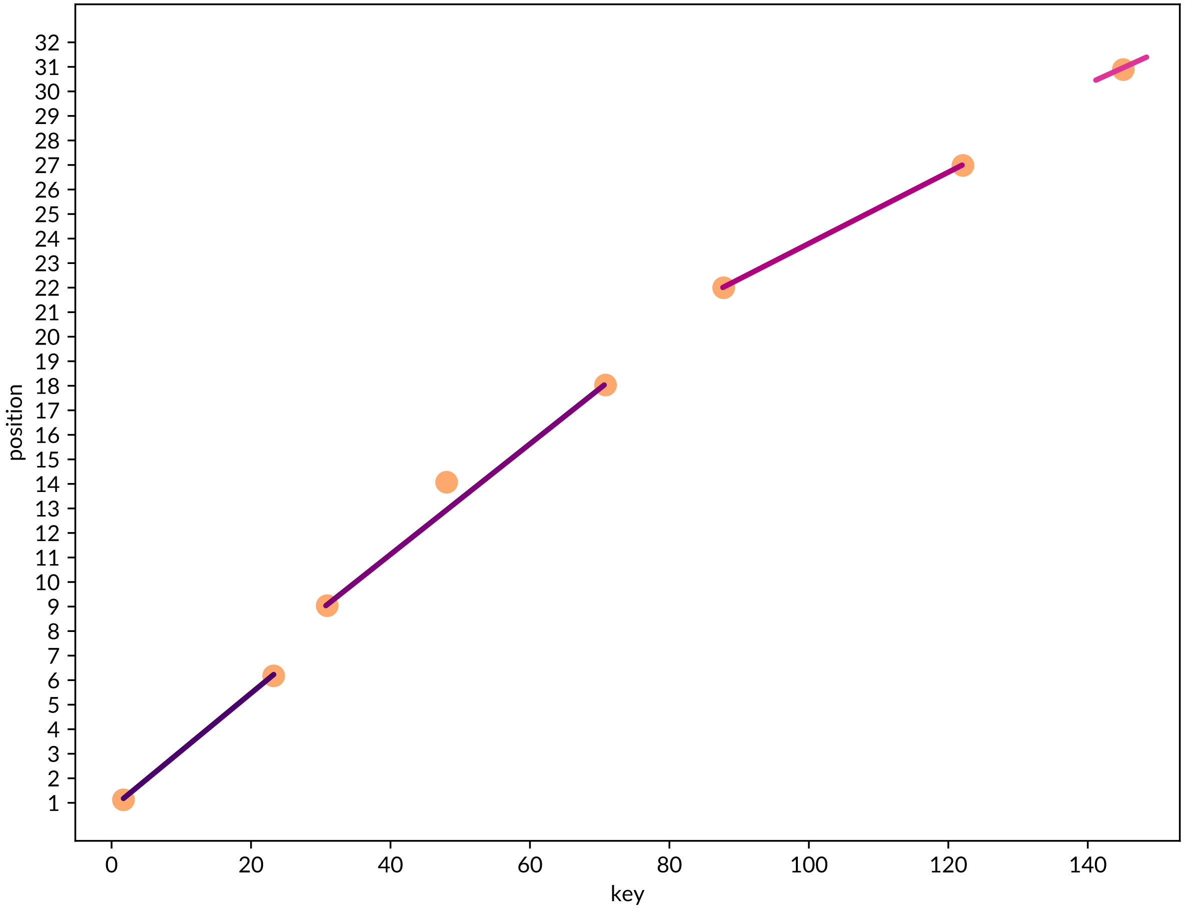
2	23	31	48	71	88	122	145

Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

PGM-index construction

Step 1. Compute the optimal piecewise linear ε-approximation in $O(n)$ time

Step 2. Store the segments as triples $s_i = (key, slope, intercept)$



Step 3. Keep only $s_i.key$

Step 4. Repeat recursively

2	23	31	48	71	88	122	145

Memory layout of the PGM-index

Very fast construction, a couple of seconds for 1 billion keys

It can also be constructed in a single pass

(2, sl, ic)

(2, sl, ic)	(31, sl, ic)	(88, sl, ic)	(145, sl, ic)

(2, sl, ic)	(23, sl, ic)	(31, sl, ic)	(48, sl, ic)	(71, sl, ic)	(88, sl, ic)	(122, sl, ic)	(145, sl, ic)

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	128	140	145	146

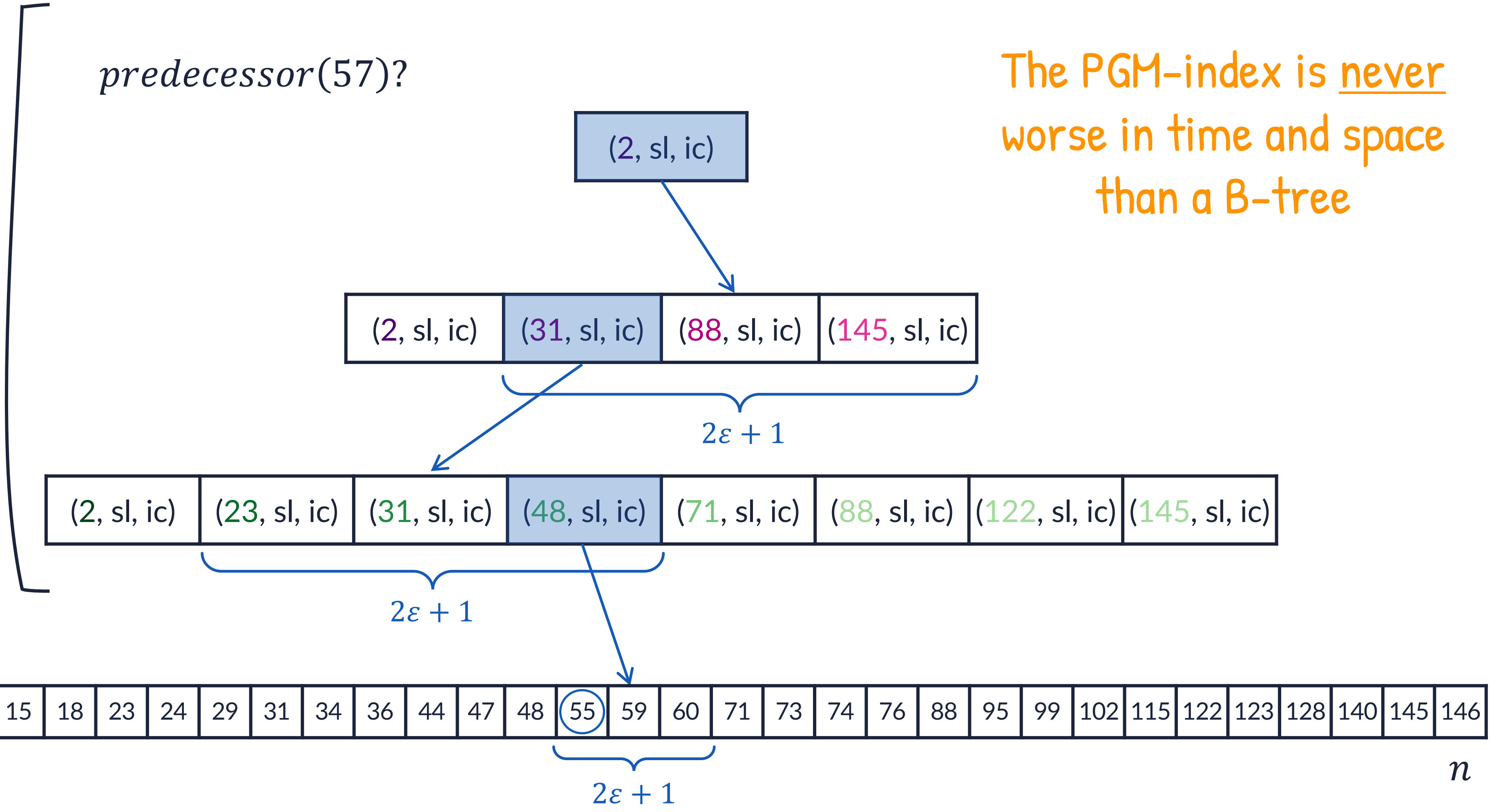
1 n

Predecessor search with $\varepsilon = 1$

B = disk page-size

Set $\varepsilon = \Theta(B)$ for queries in $O(\log_B n)$ I/Os

$O(n/\varepsilon)$ space



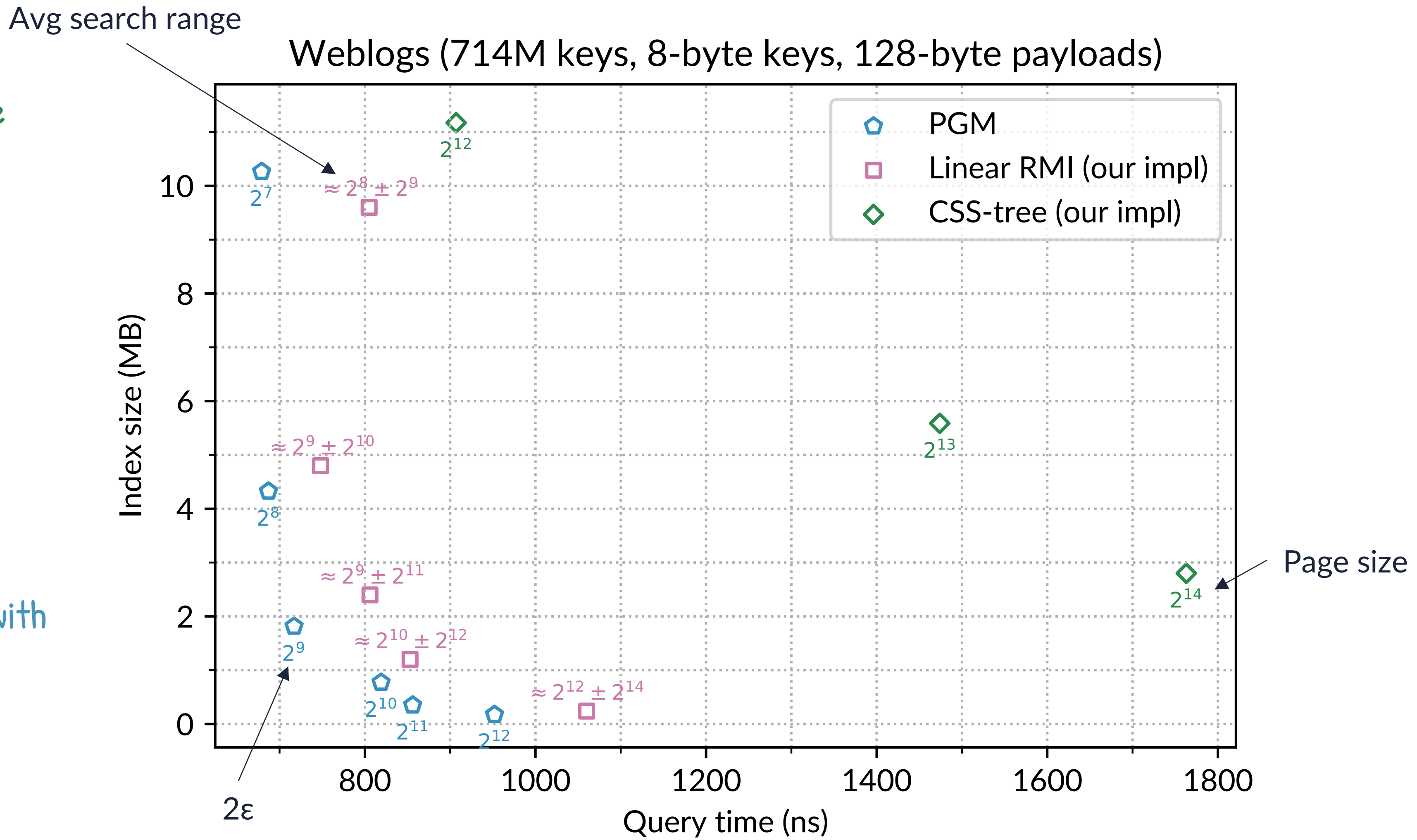
The PGM-index is <u>never</u> worse in time and space than a B-tree

Experiments

Experiments

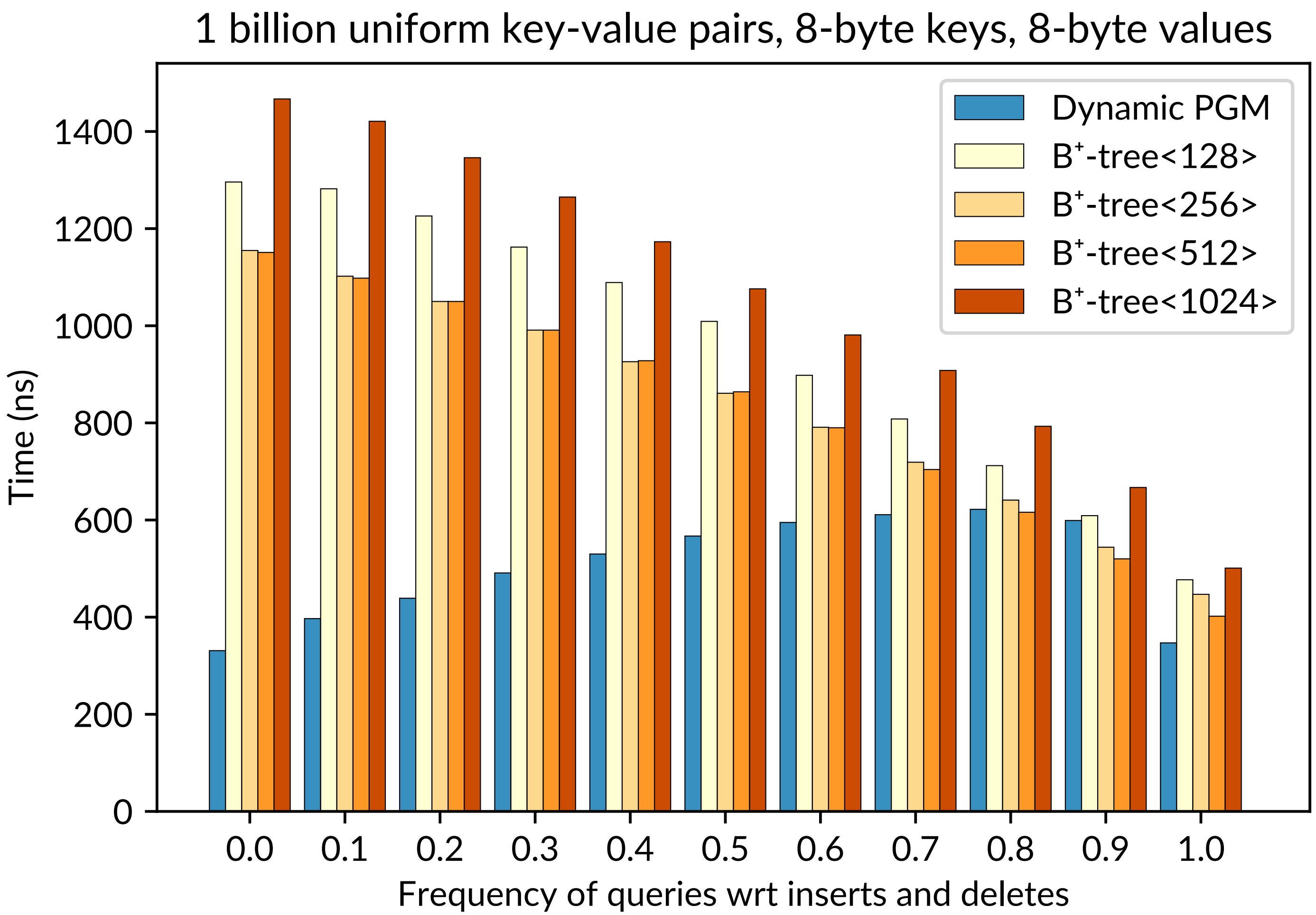
Fastest CSS-tree
128-byte pages
≈350 MB

Matched by PGM with
2ε set to 256
≈4 MB (−83×)



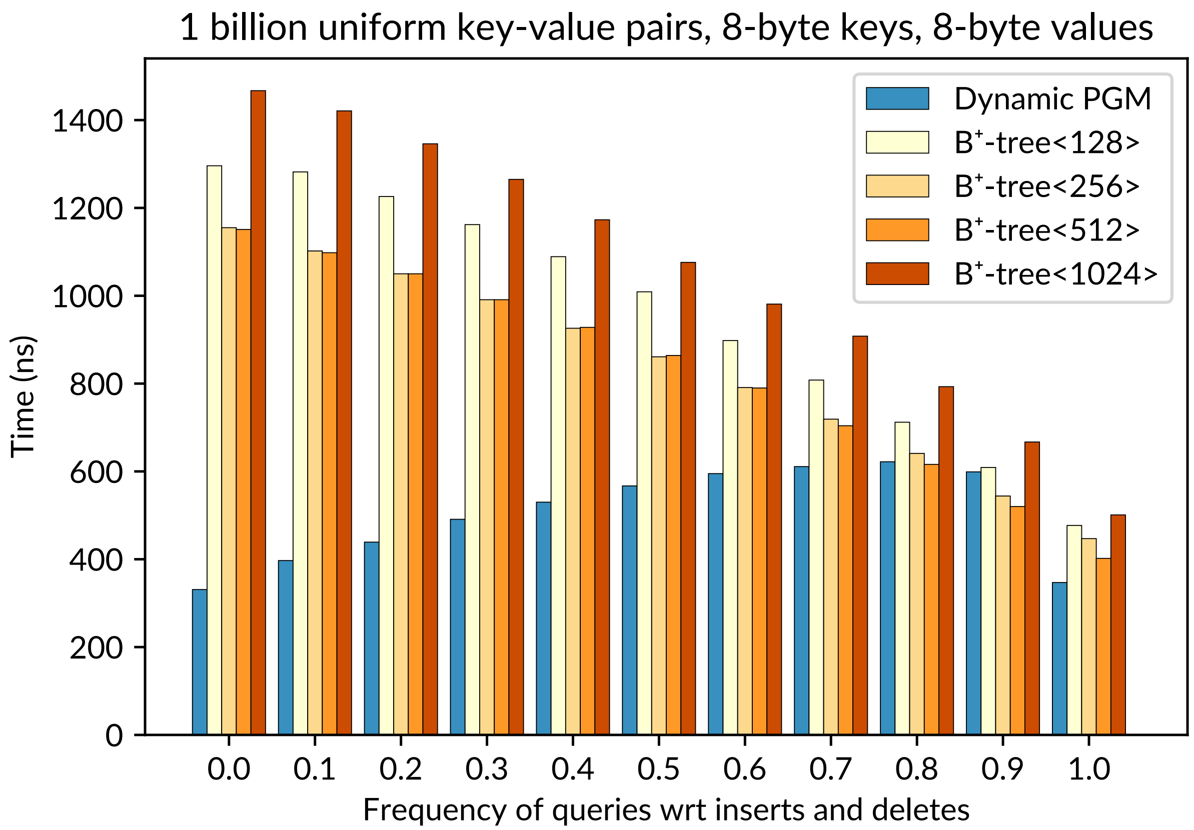
Intel Xeon Gold 5118 CPU @ 2.30GHz, data held in main memory

Experiments on updates



Intel Xeon Gold 5118 CPU @ 2.30GHz, data held in main memory

Experiments on updates



B+-tree page size	Index size	
128-byte	5.65 GB	3891×
256-byte	2.98 GB	2051×
512-byte	1.66 GB	1140×
1024-byte	0.89 GB	611×

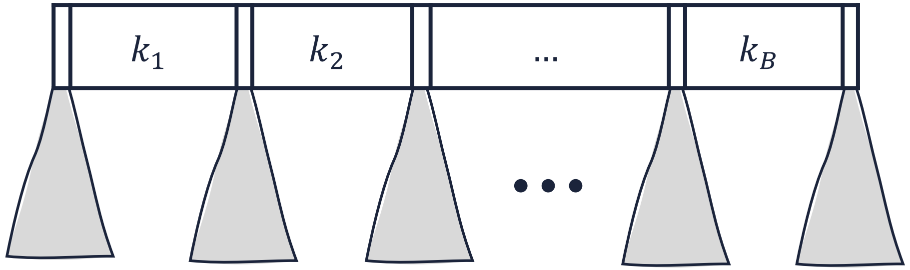
Dynamic PGM-index: 1.45 MB

Intel Xeon Gold 5118 CPU @ 2.30GHz, data held in main memory

Why the PGM is so effective?

A B-tree node

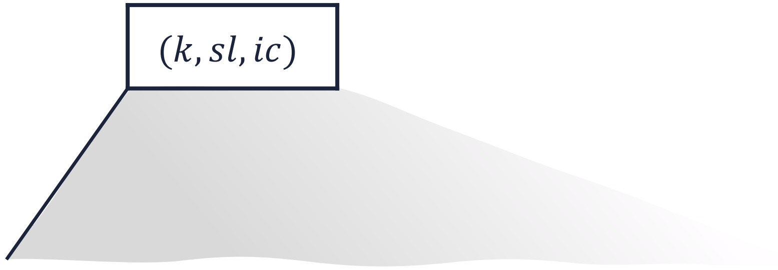
Page size B



In one I/O and $O(\log_2 B)$ steps the search range is reduced by $1/B$

A PGM-index node

$2\varepsilon = B$



Here the search range is reduced by **at least** $\cancel{1/B}$

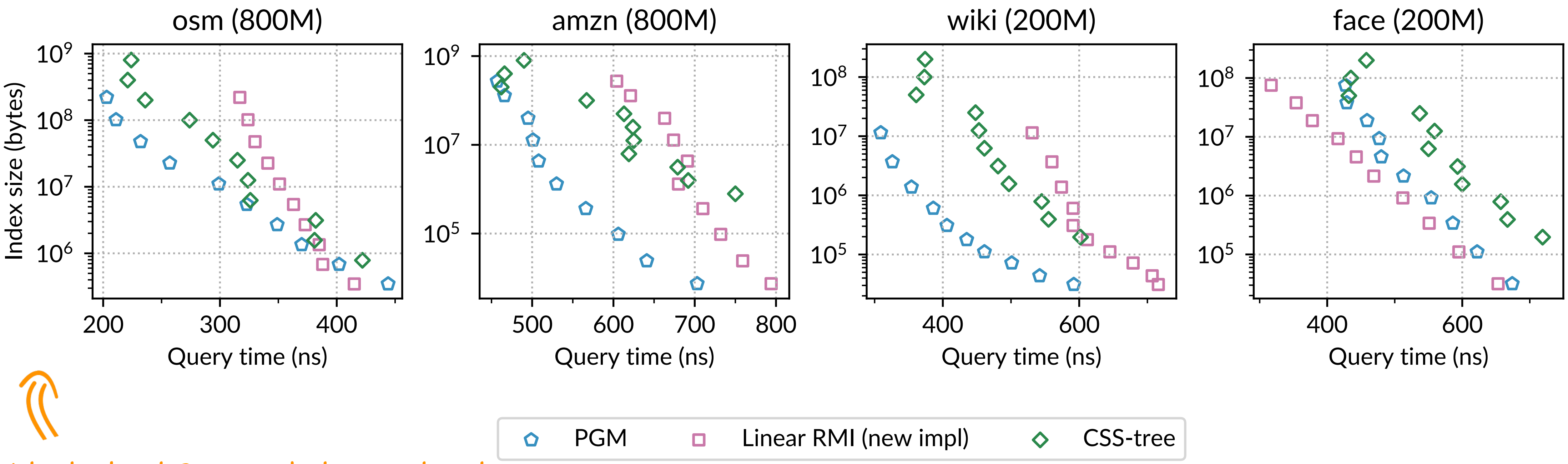
w.h.p. $1/B^2$

Ferragina et al. [ICML 2020]

New experiments with tuned Linear RMI

- 8-byte keys, 8-byte payload
- Tuned Linear RMI and PGM have the same size
- 10M predecessor searches, uniform query workload

PGM improved the empirical performance of a tuned Linear RMI



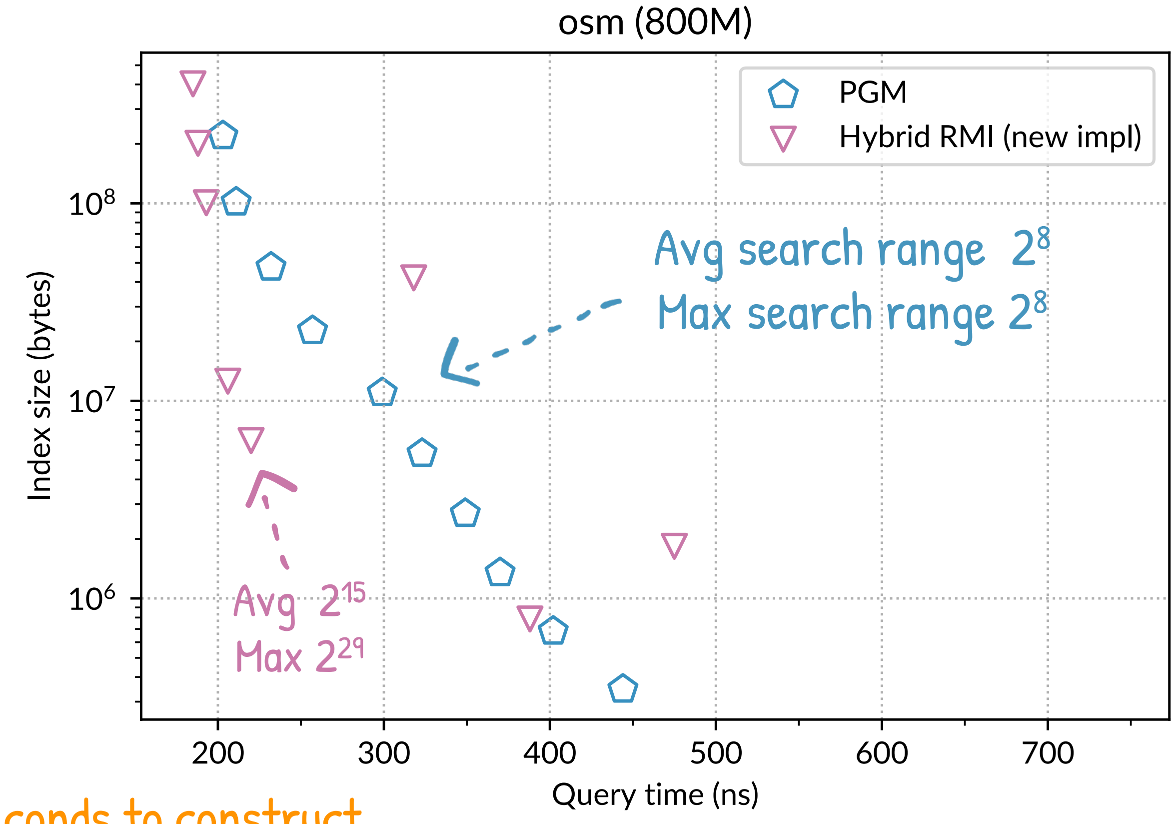
Each PGM took about 2 seconds to construct
RMI took 30× more!

They tested positive lookups. Here we test predecessor queries

New tuned Linear RMI implementation and datasets from Marcus et al., 2020 [arXiv:2006.12804]

New experiments with tuned Hybrid RMI

- 8-byte keys, 8-byte payload
- RMI with non-linear models, tuned via grid search
- 10M predecessor searches, uniform query workload



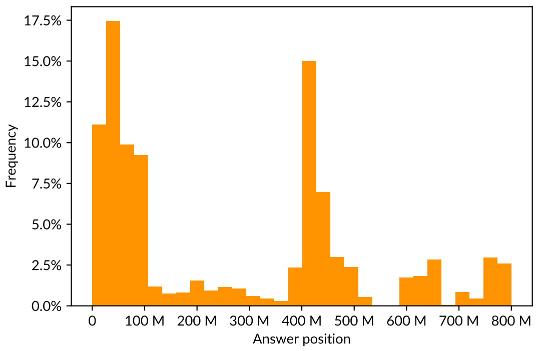
Each PGM took about 2 seconds to construct
Hybrid RMI took 40× (90× with tuning) more!

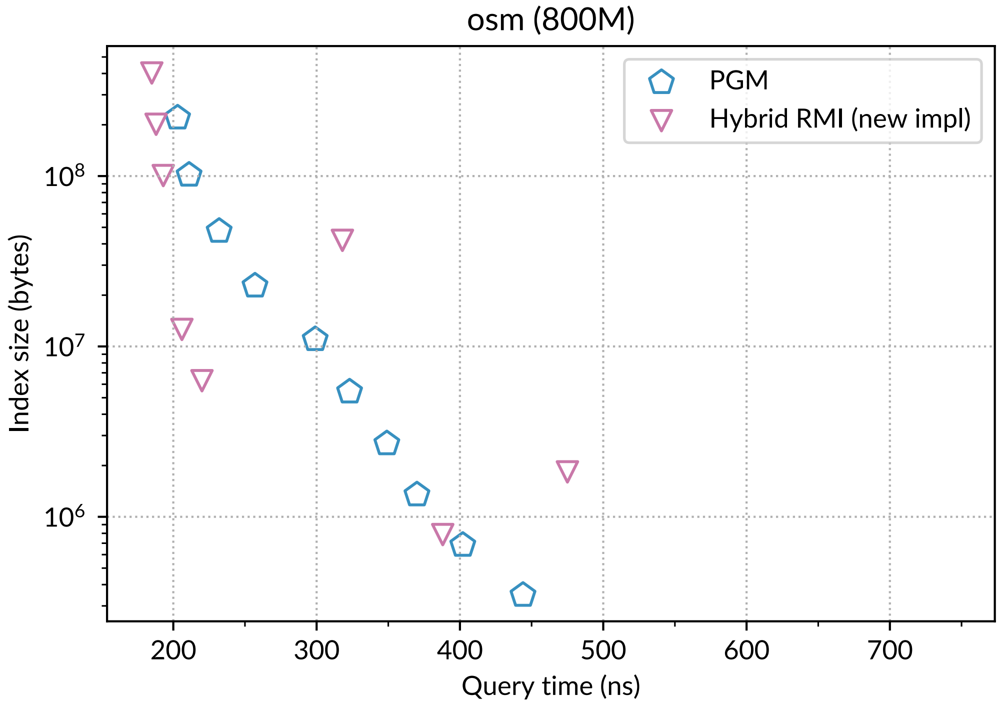
New tuned Hybrid RMI implementation and datasets from Marcus et al., 2020 [arXiv:2006.12804]

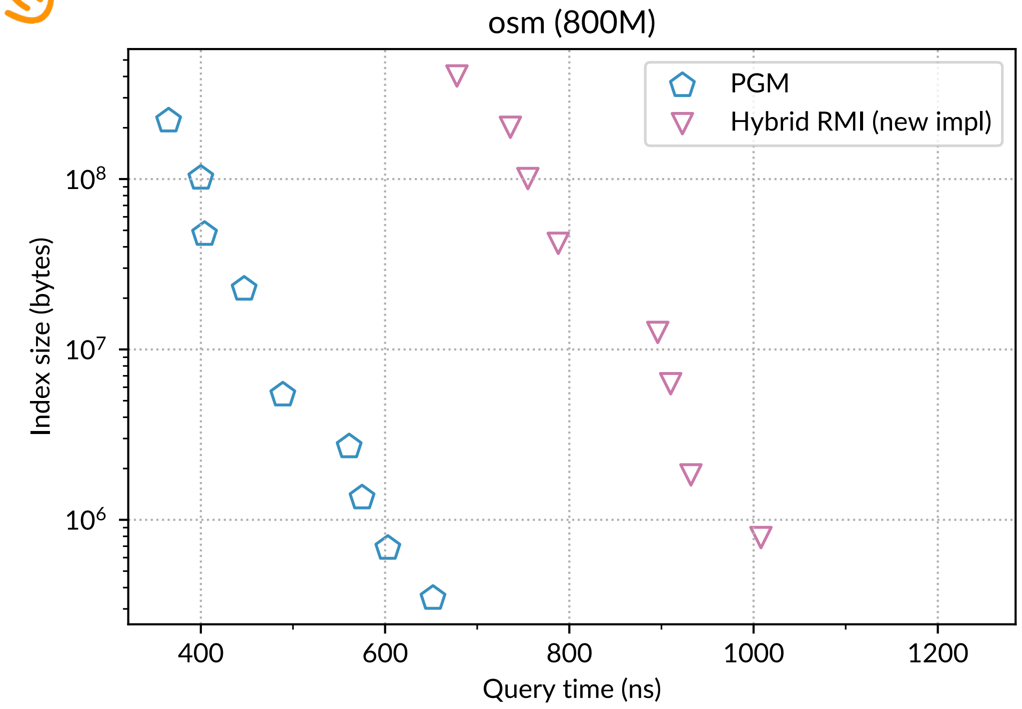
New experiments

- 8-byte keys, 8-byte payload
- RMI with non-linear models, tuned via grid search
- 10M predecessor searches

Adversarial query workload



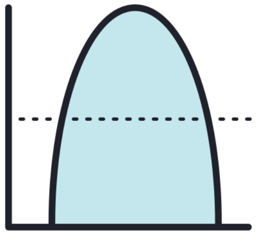




About adversarial data inputs, see Kornaropoulos et al., 2020 [arXiv:2008.00297]

New tuned Hybrid RMI implementation and datasets from Marcus et al., 2020 [arXiv:2006.12804]

More results in the paper

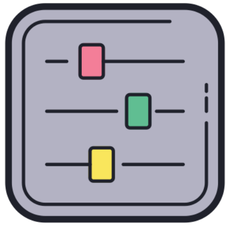


Query-distribution aware

Minimise average query time wrt
a given query workload

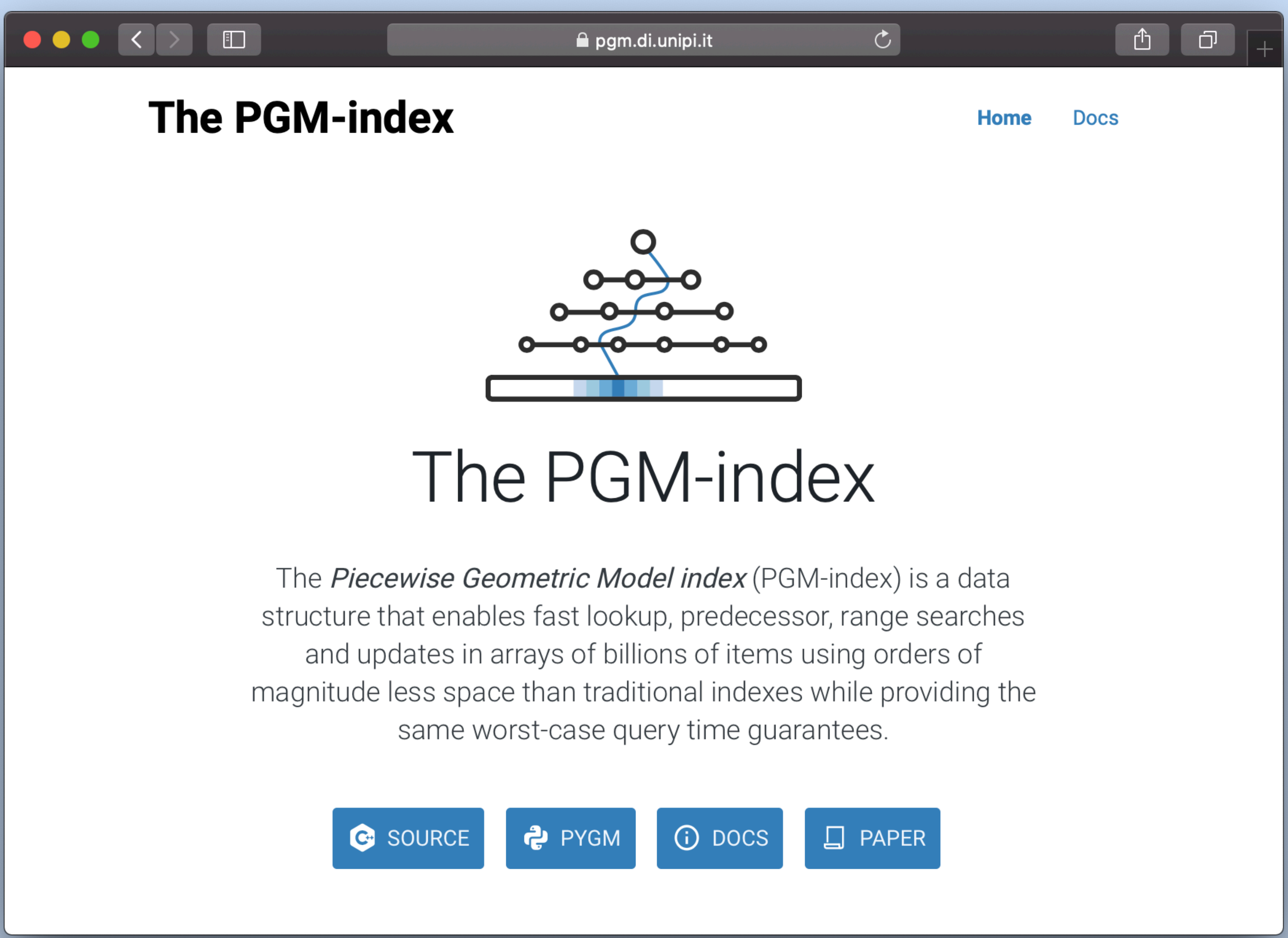
Index compression

Reduce the space of the index by a
further 52% via the compression of
slopes and intercepts



Multicriteria tuner

Minimise query time under a
given space constraint and vice versa
in a few dozens of seconds


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 lookup, predecessor, range searches and updates in arrays of billions of items using orders of magnitude less space than traditional indexes while providing the same worst-case query time guarantees.
SOURCE
PYGM
DOCS
PAPER