Theory and practice of learning-based compressed data structures

March 19, 2021
Links seminar @ Université de Lille and Inria

Giorgio Vinciguerra
PhD Student in CS
pages.di.unipi.it/vinciguerra

Università di Pisa

Outline

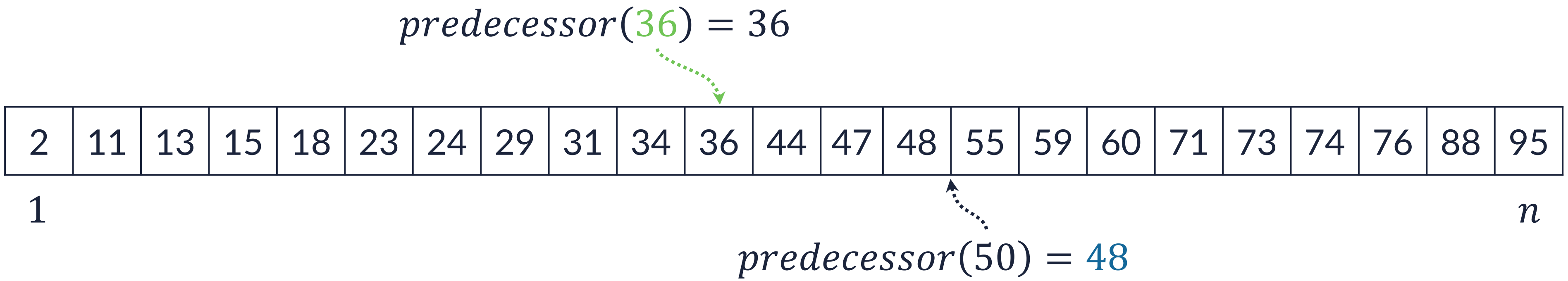
- Revisit two classical problems in data structure design:
  - Predecessor search
  - Rank/select dictionary problem
- Exploit a new kind of data regularity based on geometric considerations: *approximate linearity*
- Introduce two theoretically and practically efficient solutions for the problems above:
  - PGM-index
  - LA-vector
- Discuss the theoretical grounds on the “power” of the *approximate linearity* concept

Problem 1

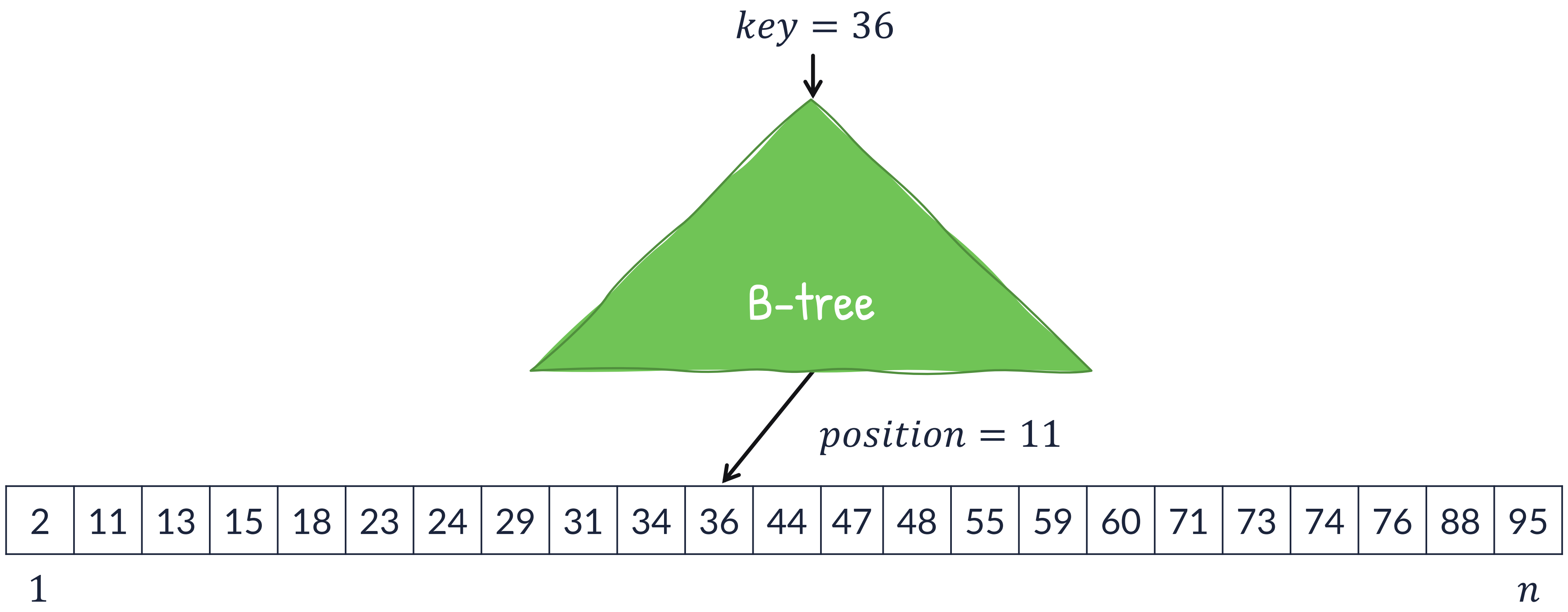
Predecessor search

The predecessor search problem

- Given n sorted input keys (e.g. integers), implement $predecessor(x) =$ "largest key $\leq x$"
- Range queries 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!)

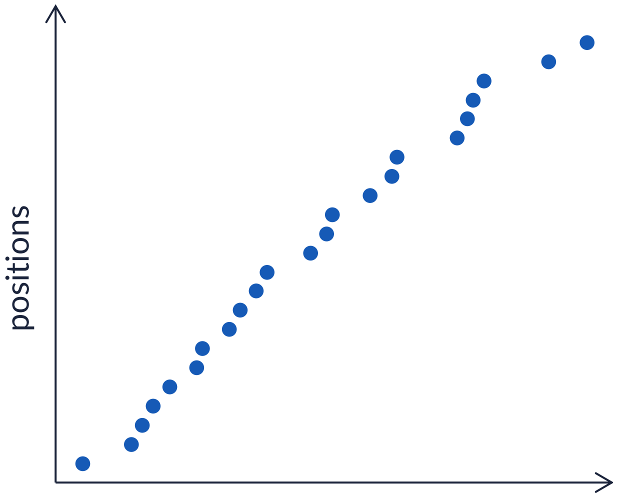


Indexes



(values associated to keys are not shown)

Input data as pairs $(key, position)$

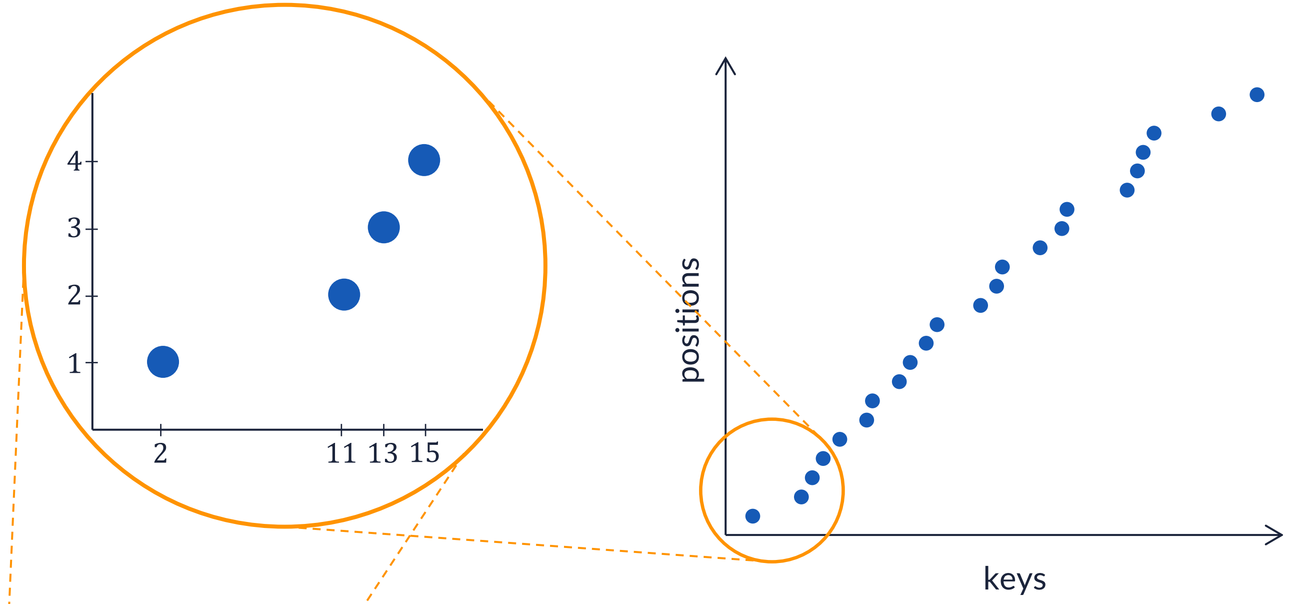


2	11	13	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95

1 n

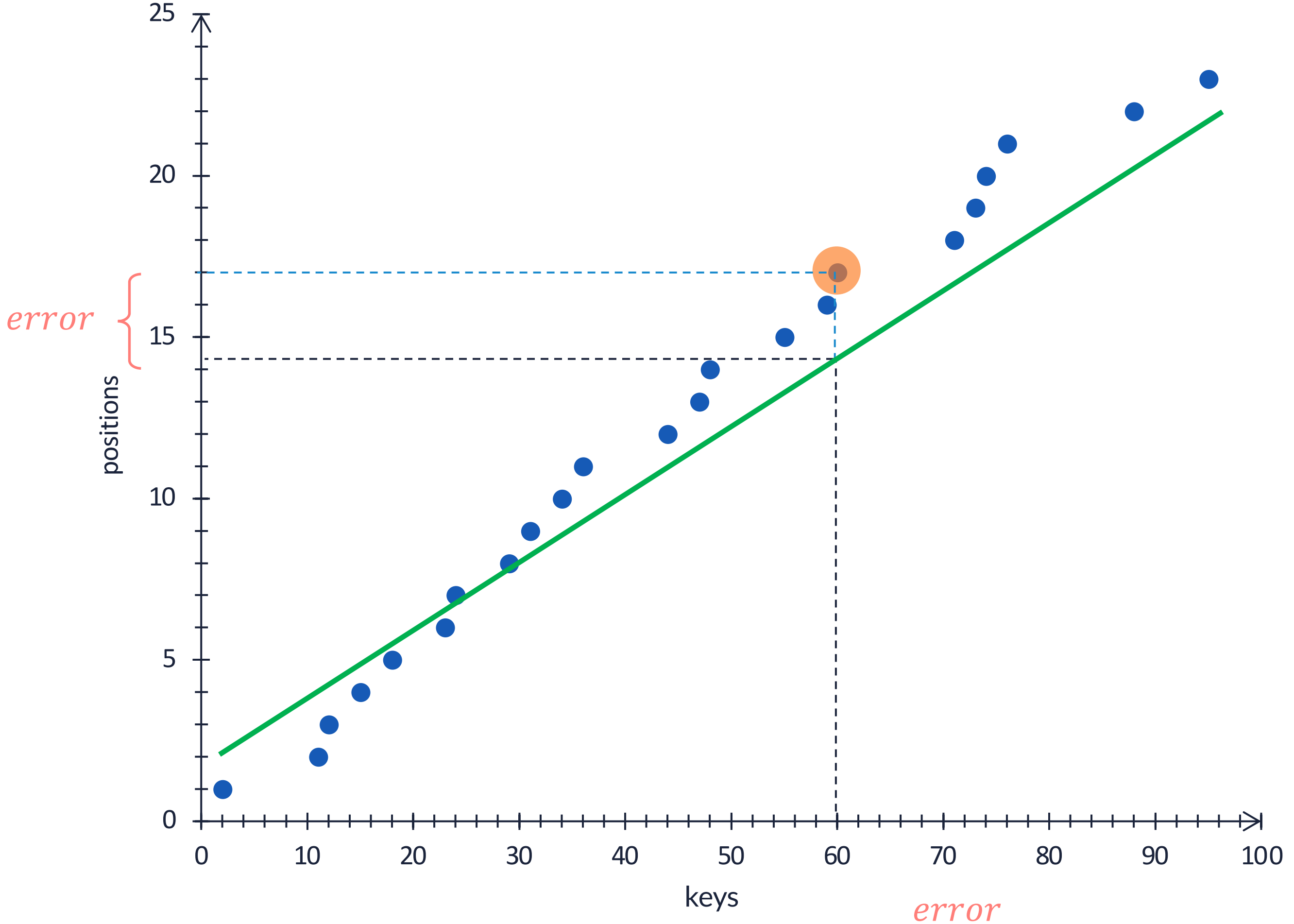
Ao et al. [VLDB 2011]

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

Learning the mapping keys ↣ positions



error

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	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

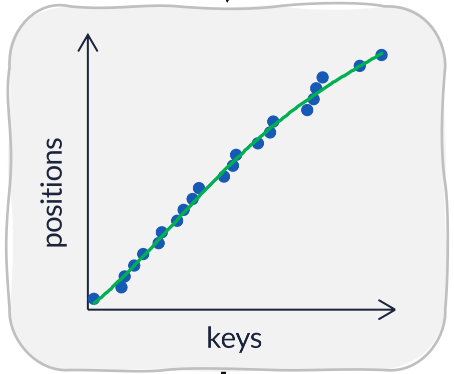
Learned indexes

Query latency = time to output a position + time to "fix the error" via binary search

How to strike a good balance between the model complexity and the query latency?

Model trained on a dataset of pairs (key, pos)
$\mathcal{D} = \{(2,1), (11,2), \ldots, (95, n)\}$

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

Query distribution aware

Resistant to adversarial inputs and queries

Ingredients of the PGM-index

Opt. piecewise linear ε-approx.

Fast to construct, best space usage for linear learned indexes

Fixed model “error” ε

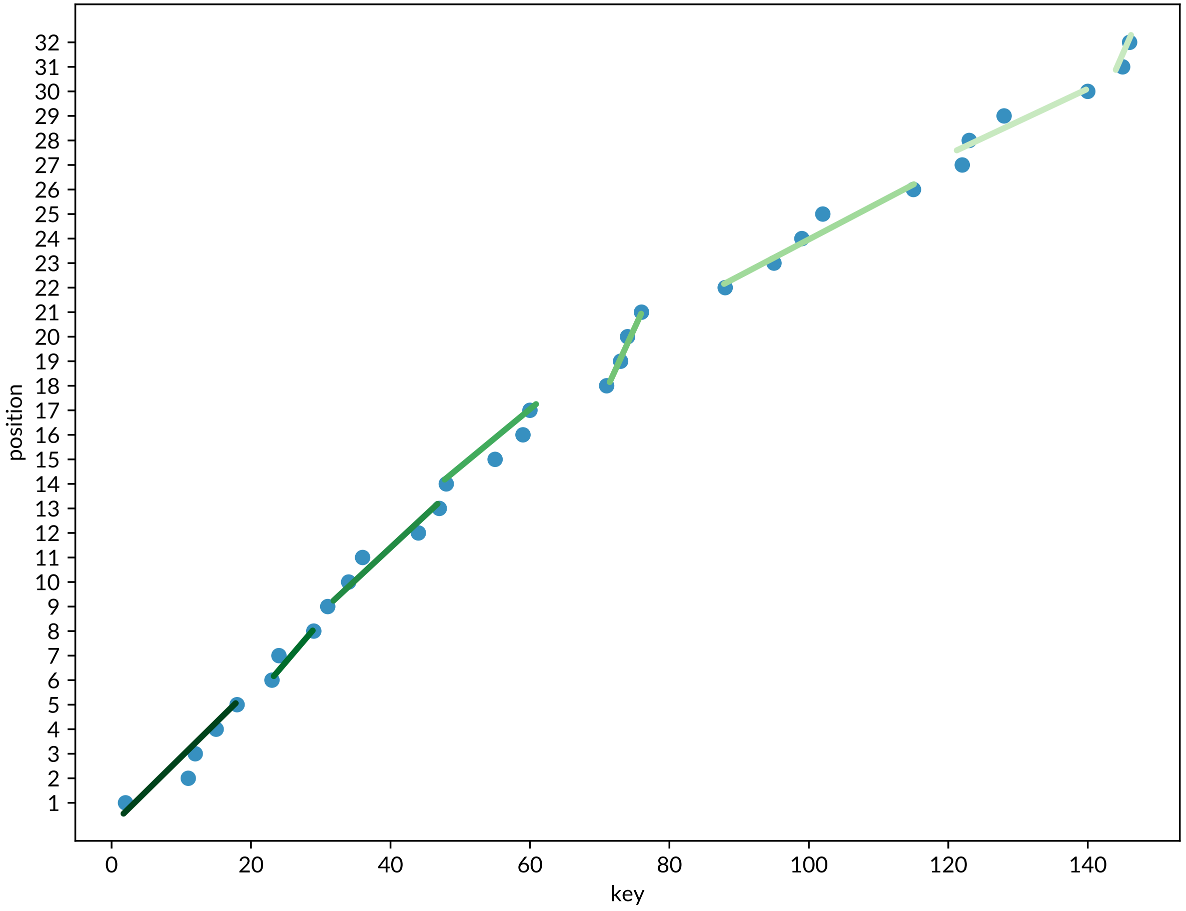
Control the size of the search range (like the page size in a B-tree)

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

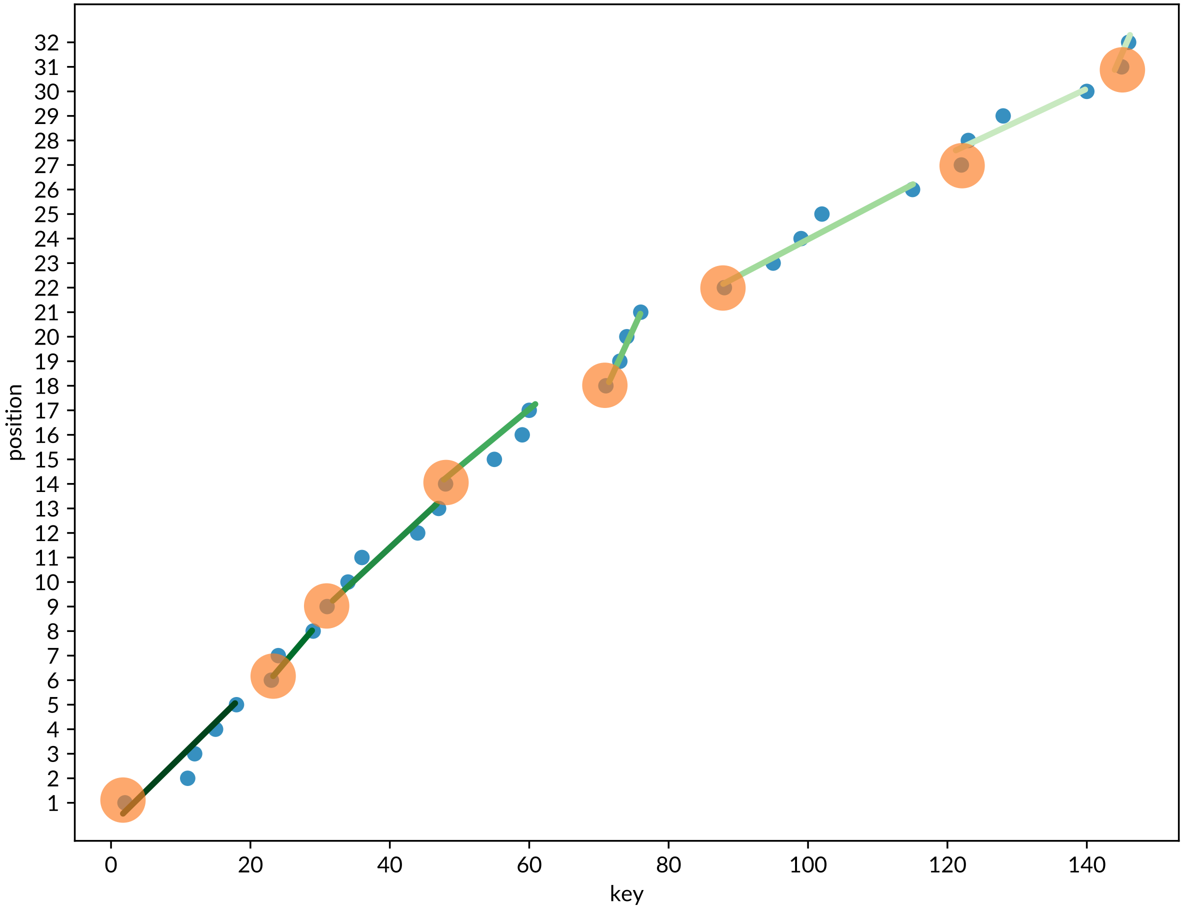


2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123	128	140	145	146

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

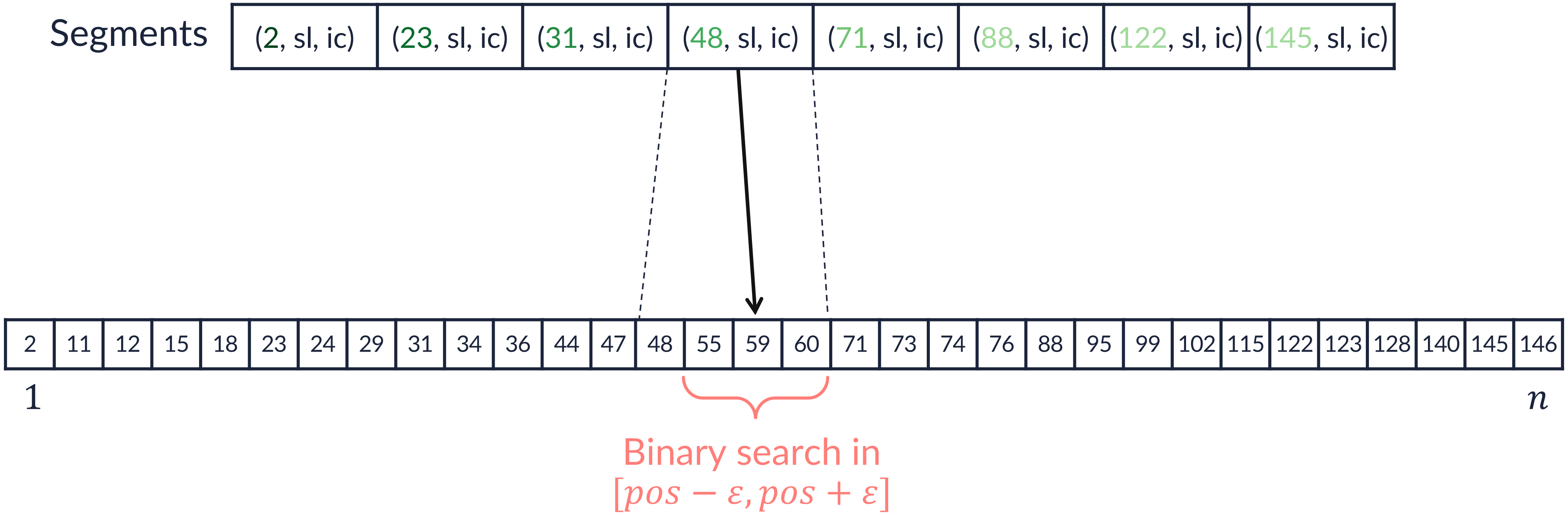
$s_i = (key, slope, intercept)$

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

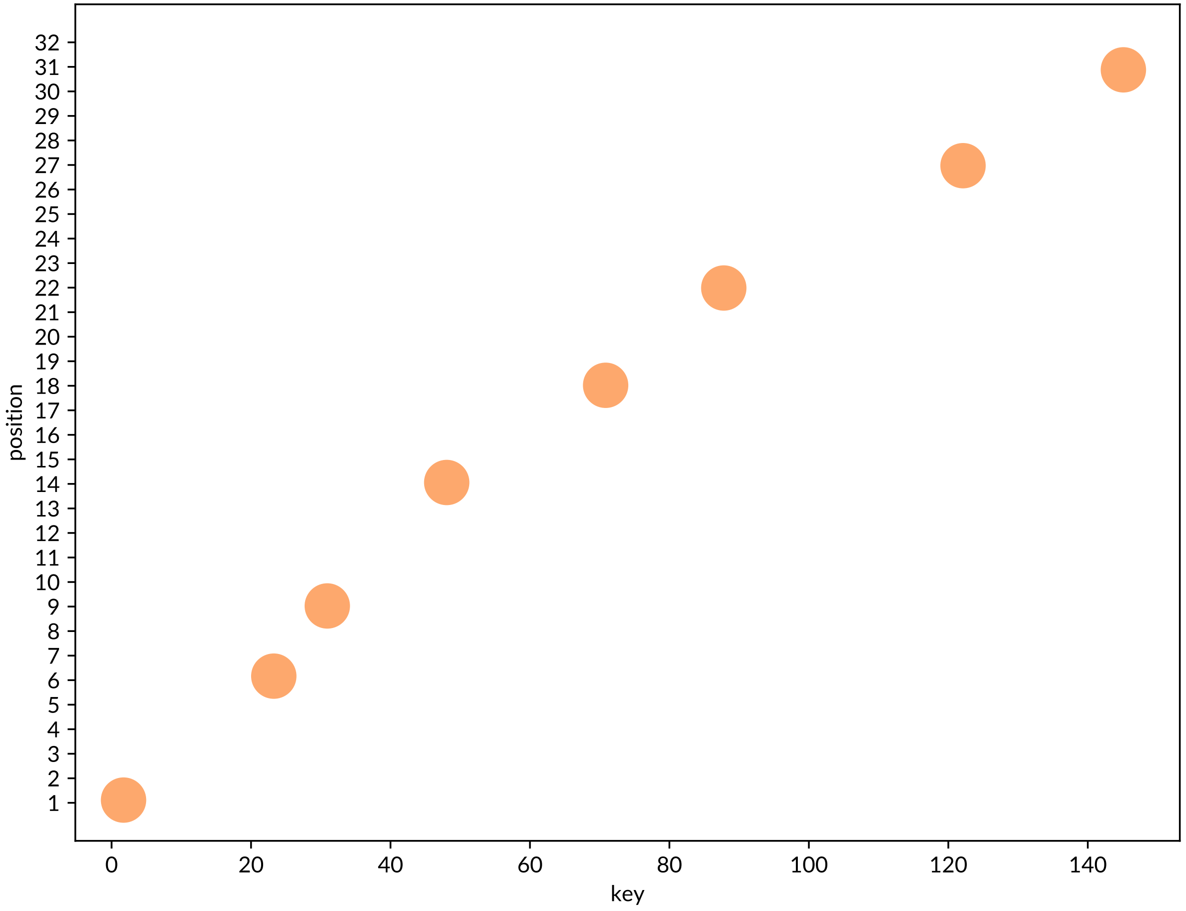
Partial memory layout of the PGM-index

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

Step 3. Keep only $s_i.key$

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

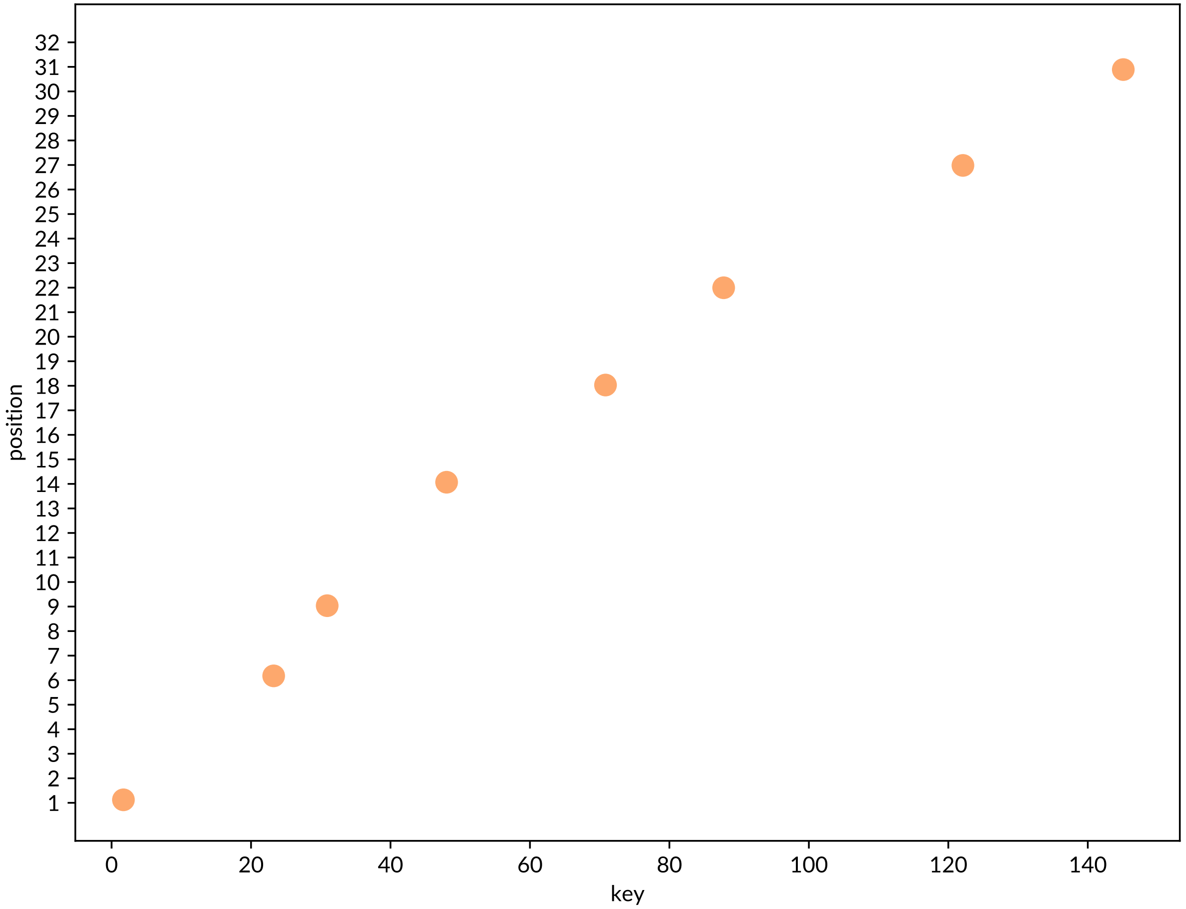
1 $\qquad$ n

PGM-index construction

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

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

Step 3. Keep only $s_i.key$

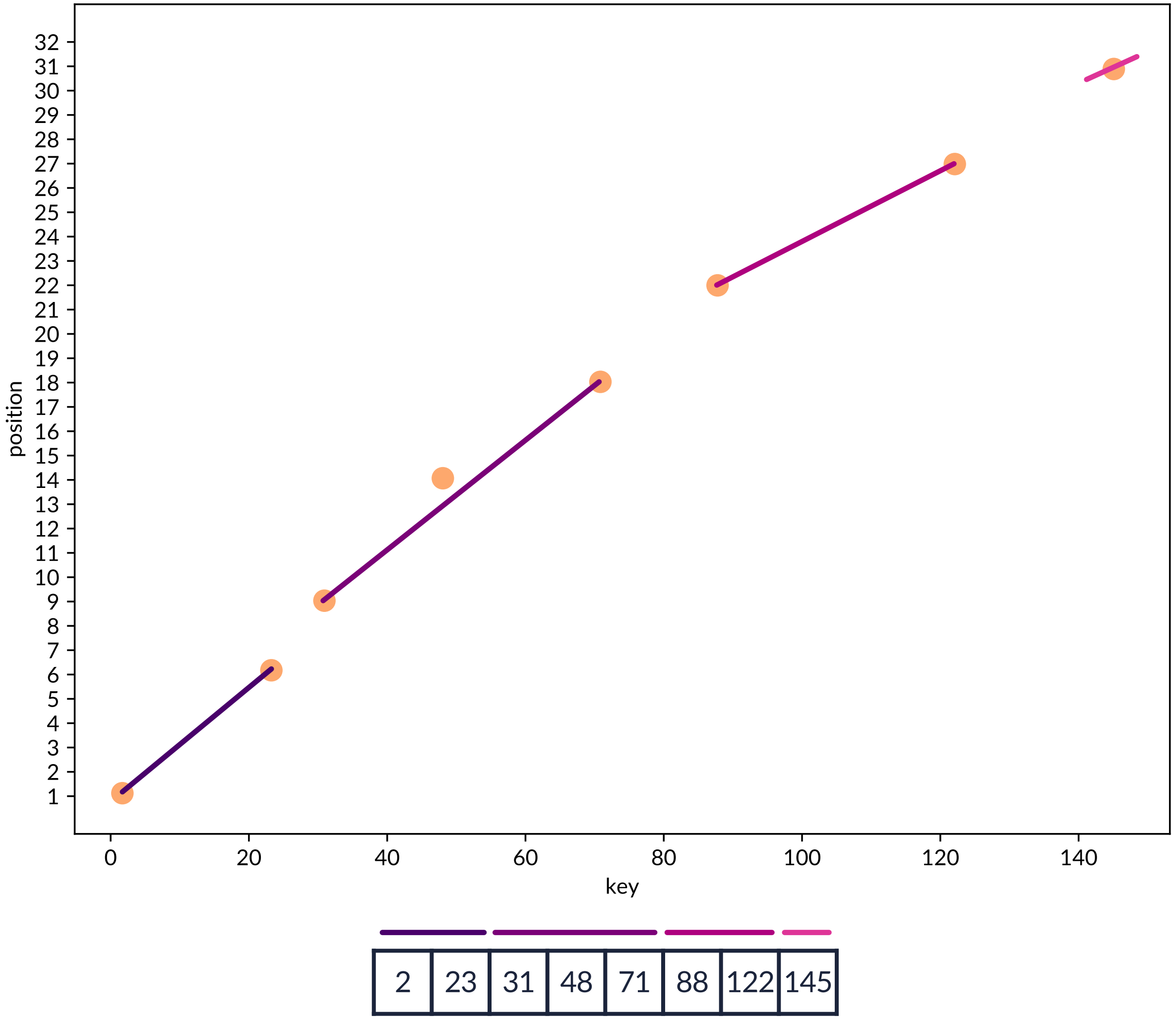


2	23	31	48	71	88	122	145

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

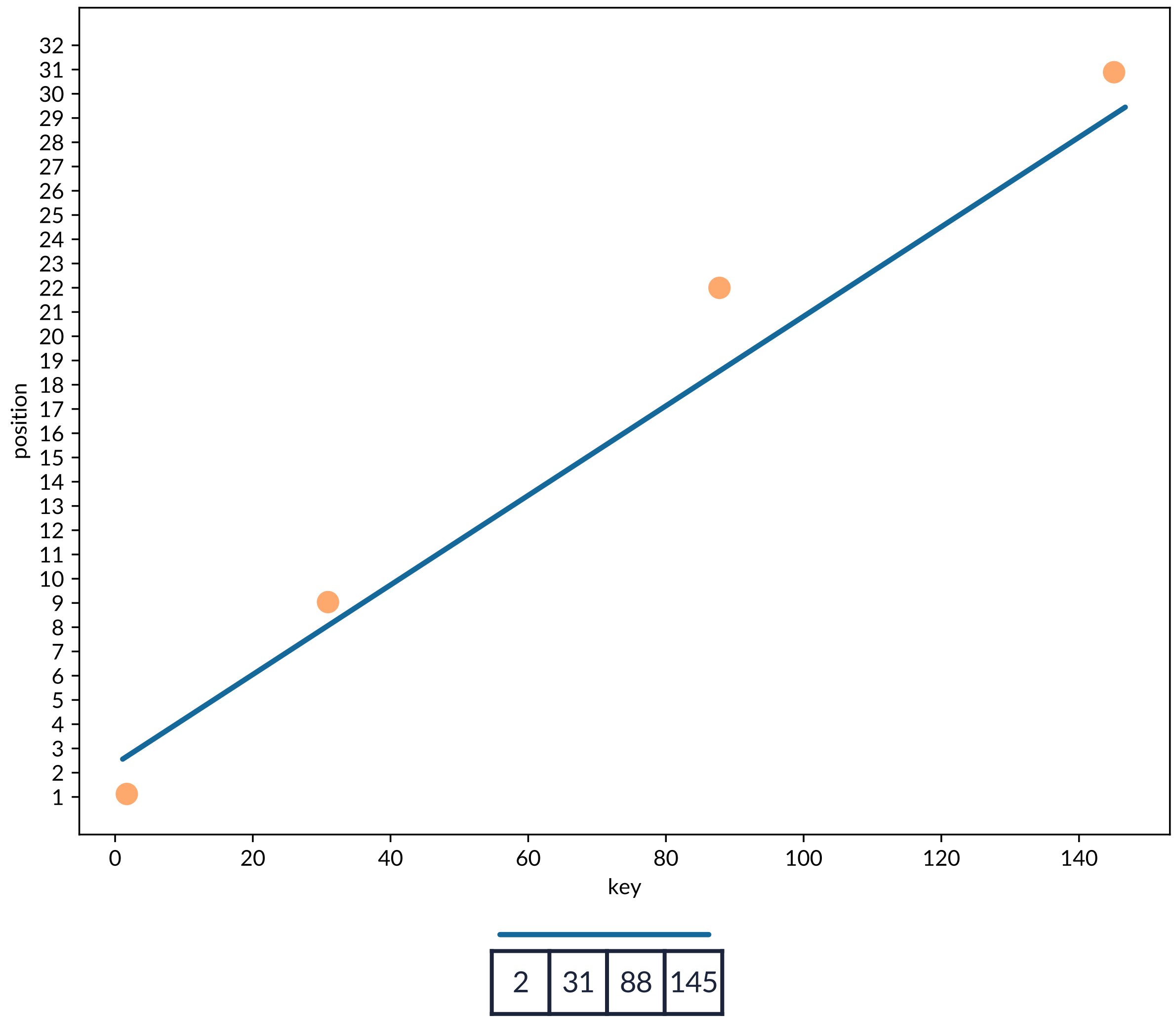
PGM-index construction

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

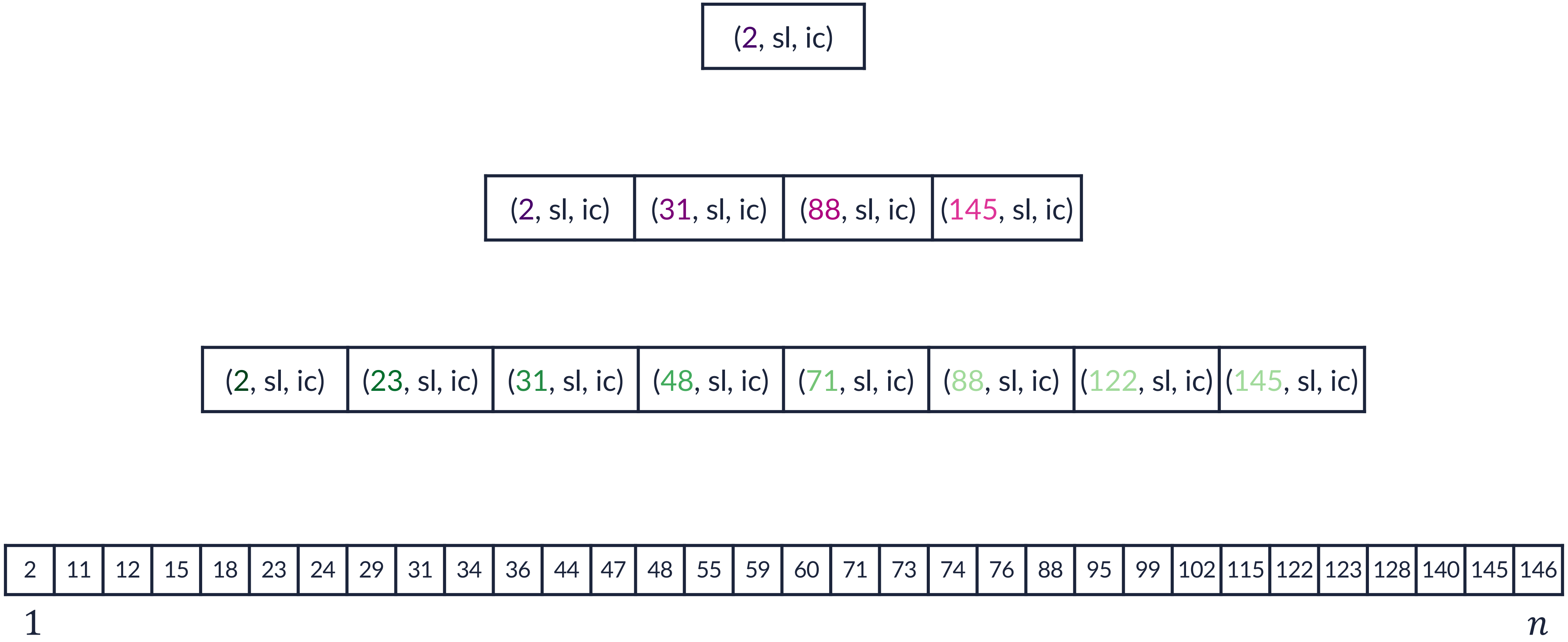
Step 2. Store the segments as triples $s_i = (key, slope, intercept)$

Step 3. Keep only $s_i.key$

Step 4. Repeat recursively



Memory layout of the PGM-index

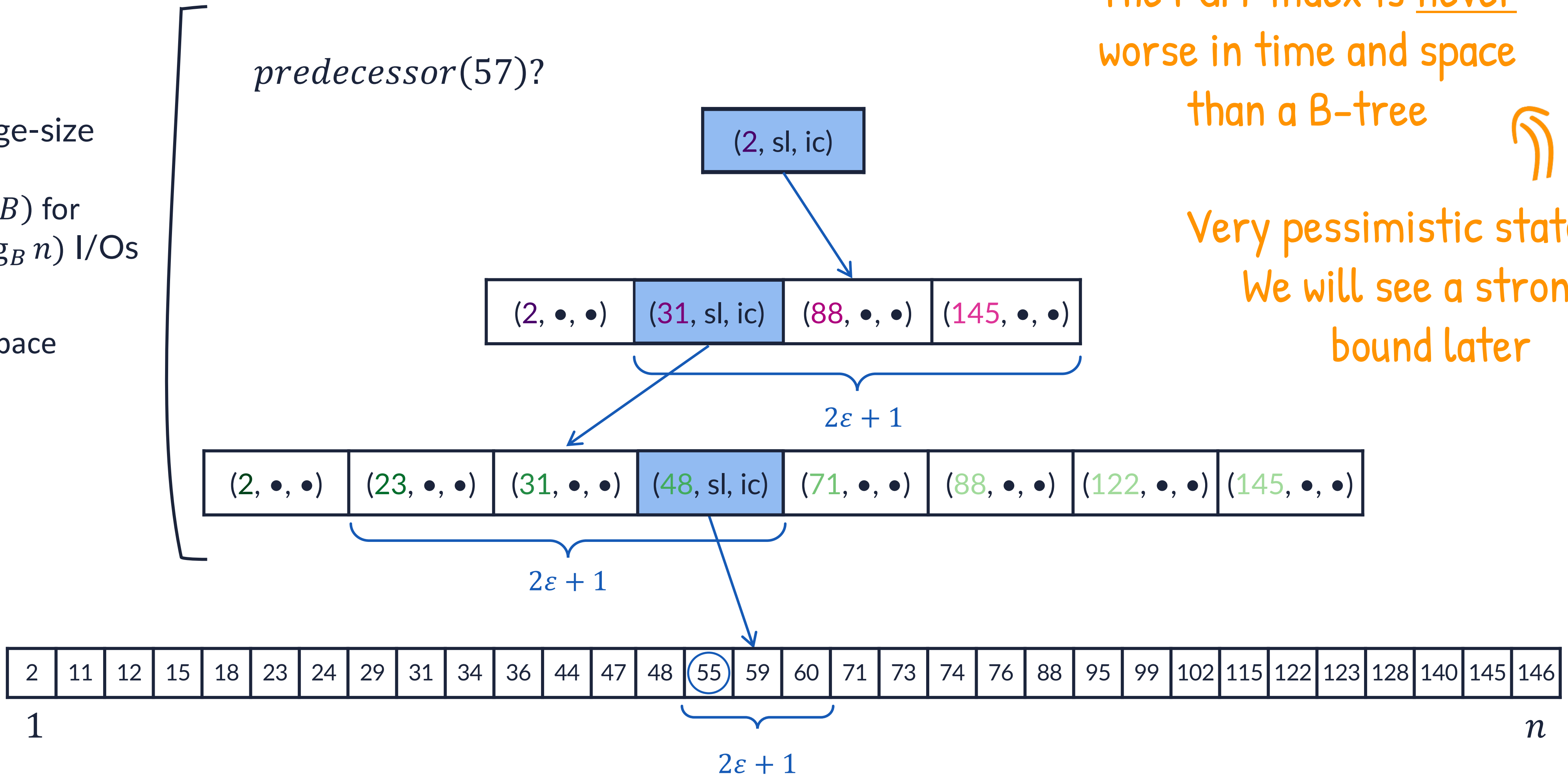


Predecessor search with $\varepsilon = 1$

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

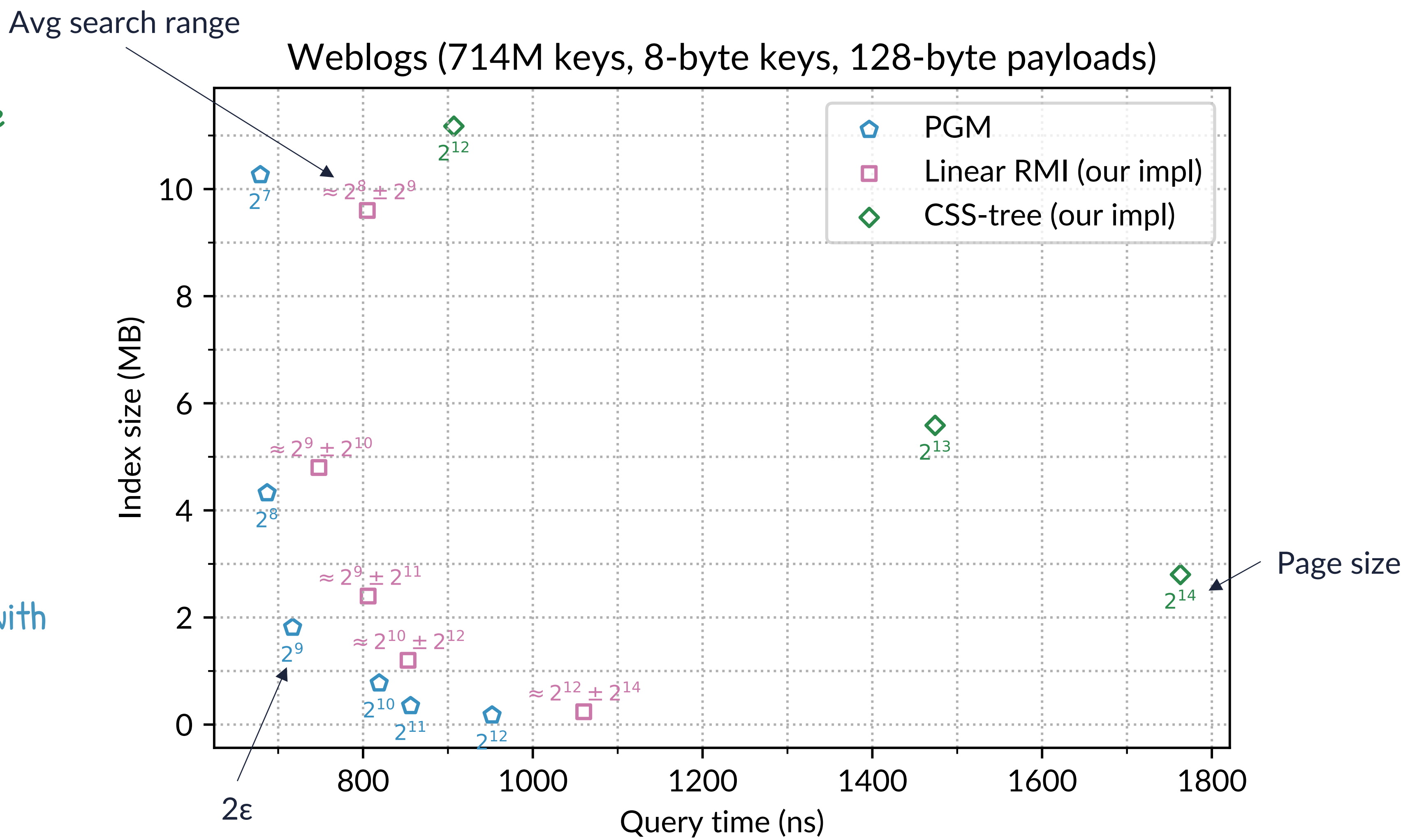
Very pessimistic statement
We will see a stronger bound later

Experiments

Experiments

Fastest CSS-tree
128-byte pages
≈350 MB

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

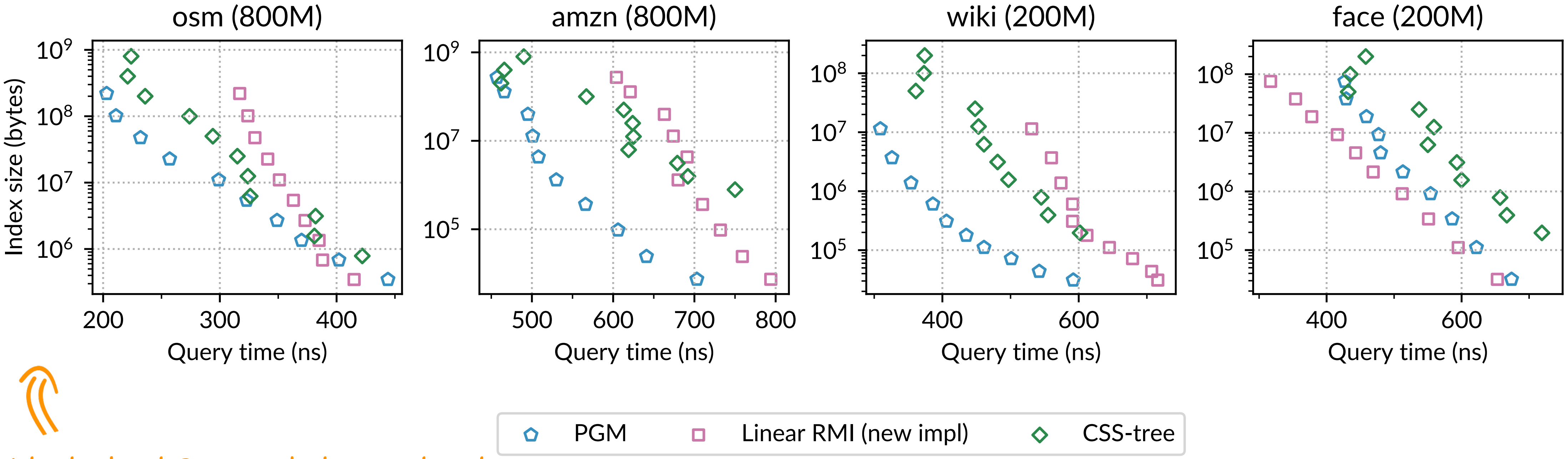


Intel Xeon Gold 5118 CPU @ 2.30GHz, data held in main memory

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

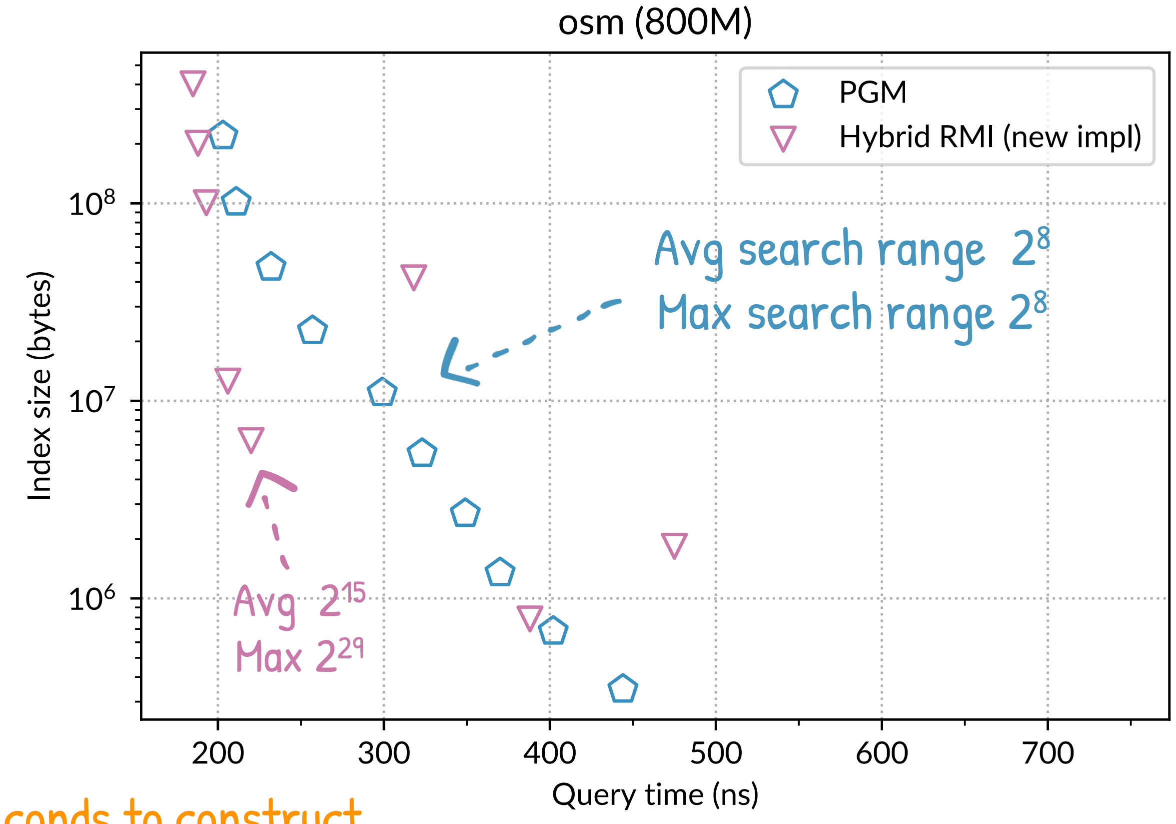


Each PGM took about 2 seconds to construct
RMI took 30× more!

New tuned Linear RMI implementation and datasets from Marcus et al. [VLDB 2021]

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



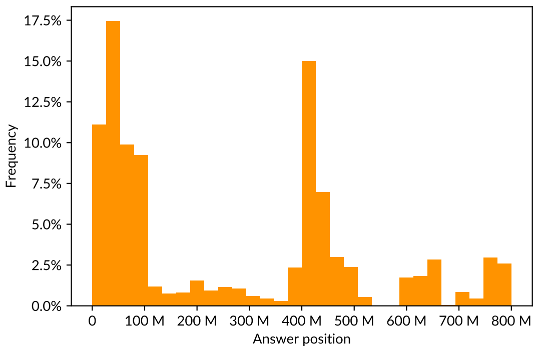
Each PGM took about 2 seconds to construct
Hybrid RMI took 40× (90× with tuning) more!

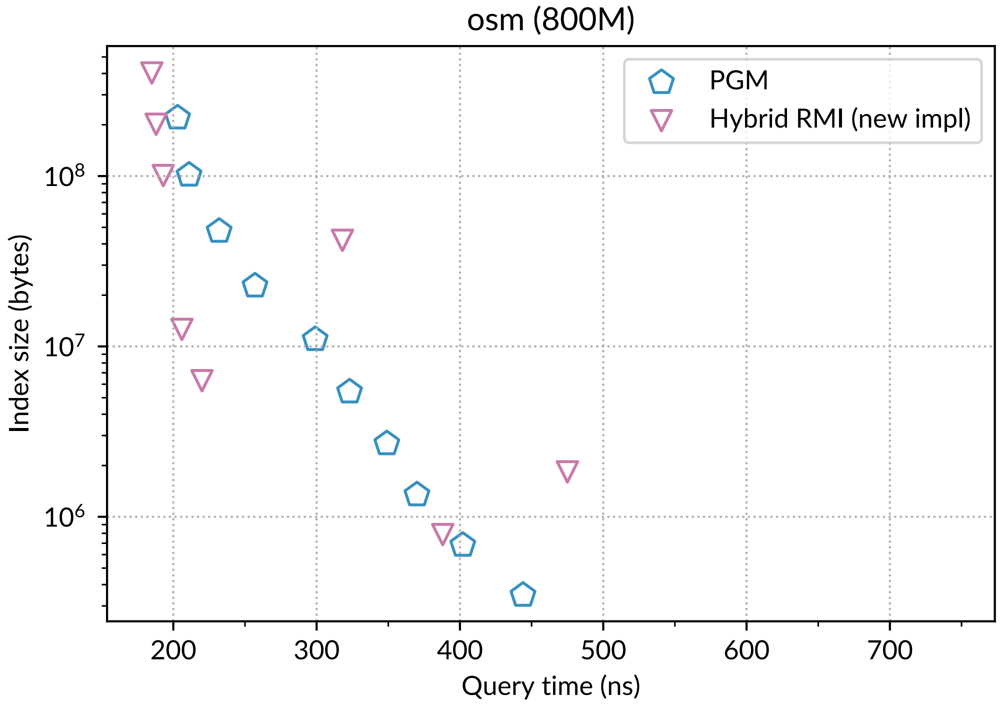
New tuned Linear RMI implementation and datasets from Marcus et al. [VLDB 2021]

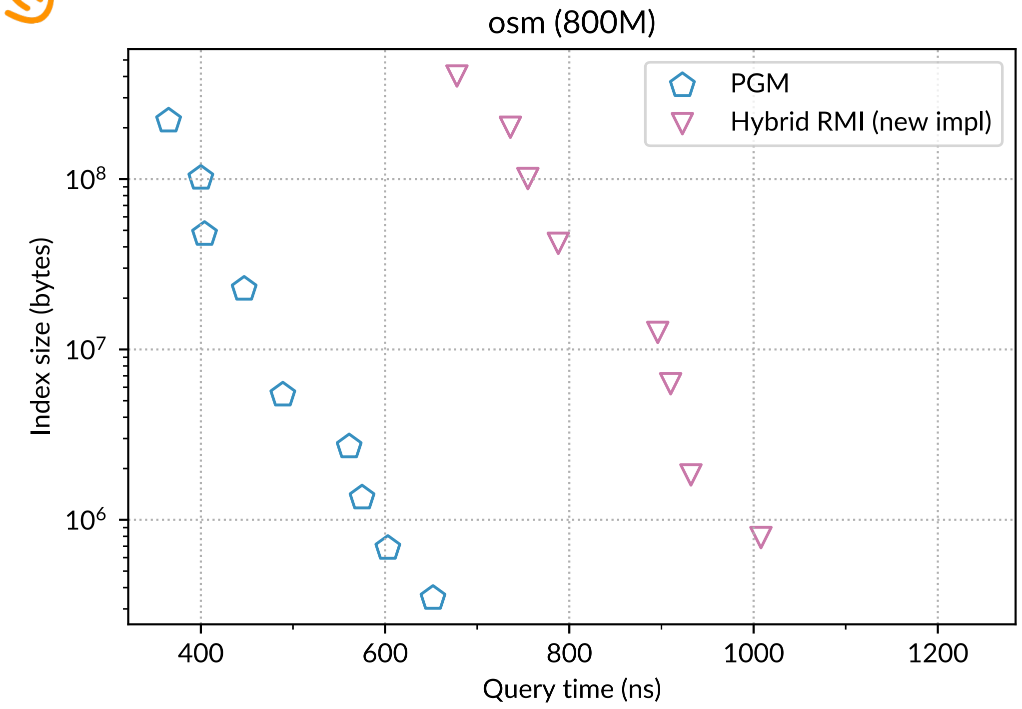
New experiments

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

Adversarial query workload



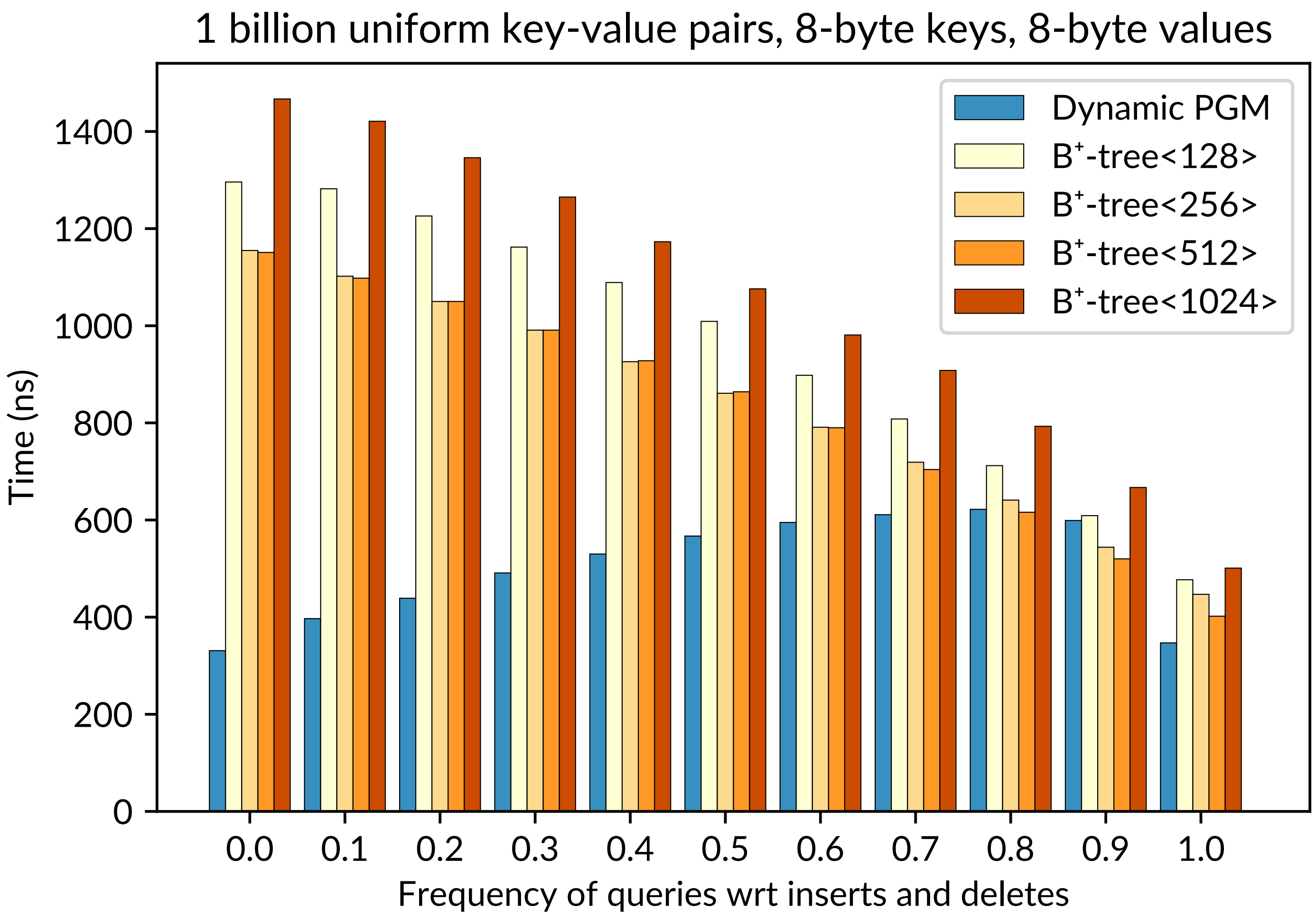




About adversarial data inputs, see Kornaropoulos et al., 2020 [arXiv:2008.00297]

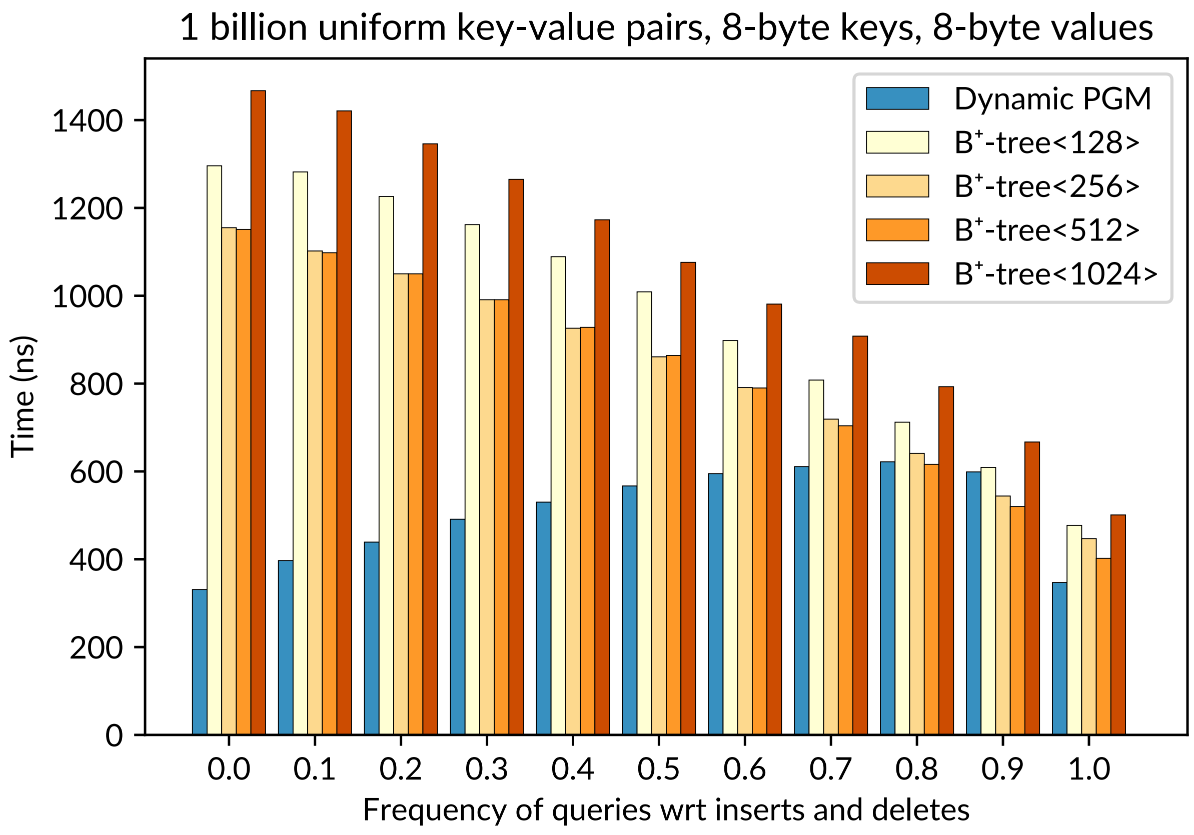
New tuned Linear RMI implementation and datasets from Marcus et al. [VLDB 2021]

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

Also in the paper:

- Compression of segments
- Query distribution awareness

VLDB2020 TOKYO Aug. 31 Sep. 04

46th International Conference on Very Large Data Bases

Paolo Ferragina and Giorgio Vinciguerra. *The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds*. PVLDB, 13(8): 1162-1175, 2020.

Website and libraries

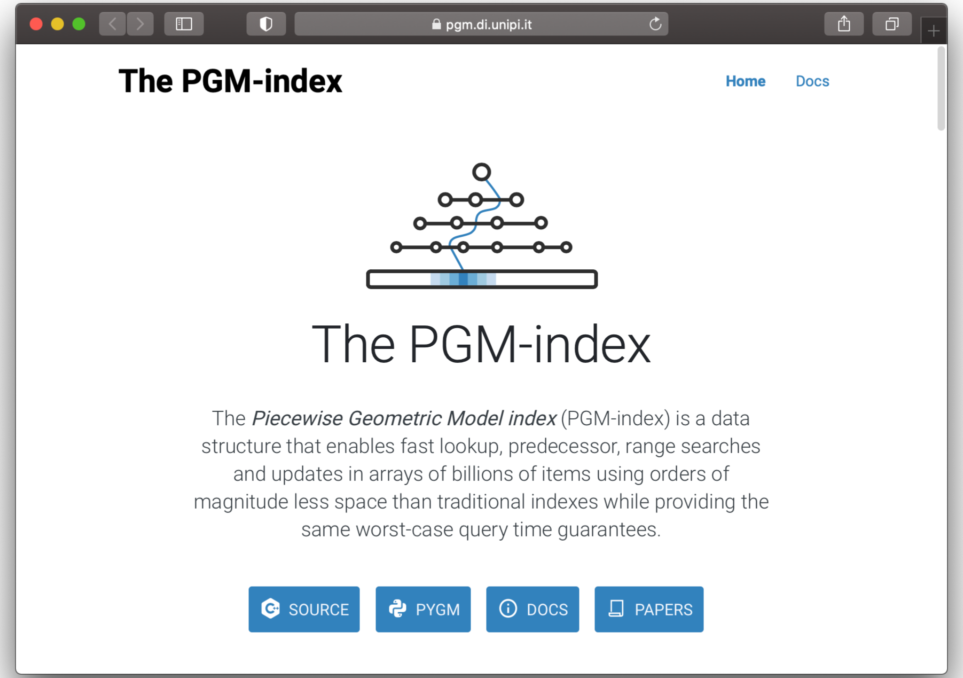
Website: https://pgm.di.unipi.it

Library (C++17): https://github.com/gvinciguerra/PGM-index

Library (Python): https://github.com/gvinciguerra/PyGM

New features

- ✓ Indexing data on disk
- ✓ Multidimensional data
- ✓ C interface



Intermezzo

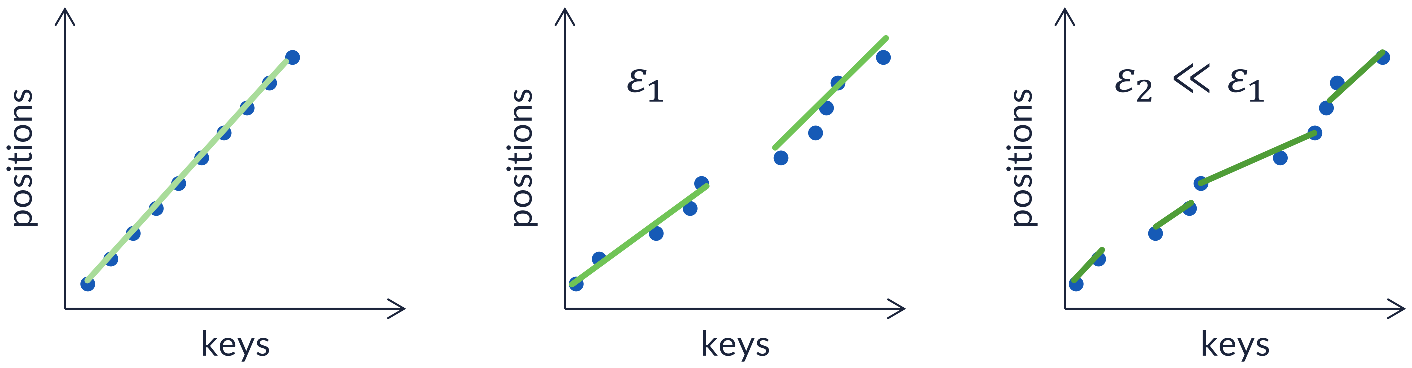
Theoretical grounds of learning-based data structures

The knowledge gap

Practice		Theory	
Same query time of traditional tree-based indexes	vs →	Same asymptotic query time of traditional tree-based indexes	👍
Space improvements of orders of magnitude, from GBs to few MBs	vs →	Same asymptotic space occupancy of traditional tree-based indexes	👎

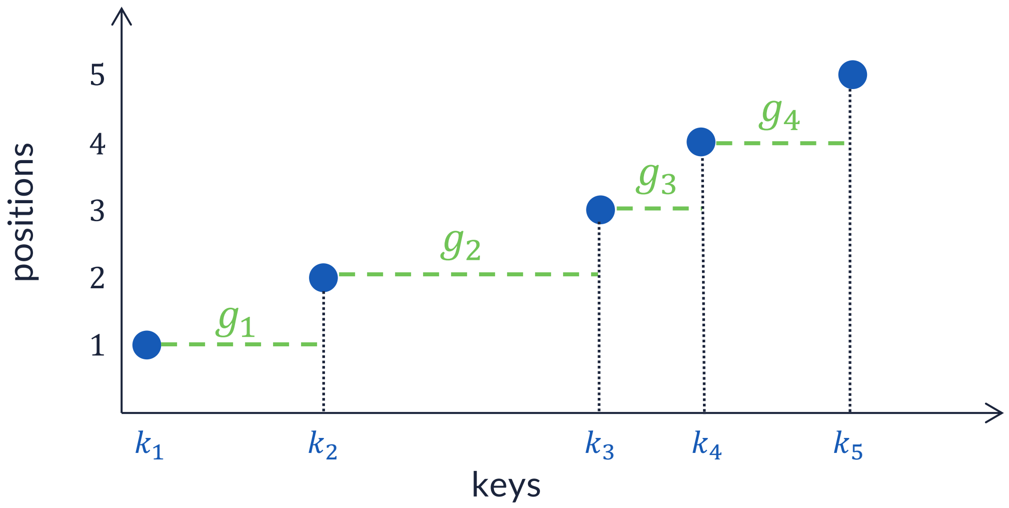
What is the space of learned indexes?

- Space occupancy ∝ Number segments
- The number of segments depends on
  - The size of the input dataset
  - How the points (key, pos) map to the plane
  - The value ε, i.e. how much the approximation is precise



Model and assumptions

- Consider gaps $g_i = k_{i+1} - k_i$ between consecutive input keys
- Model the gaps as positive iid rvs that follow a distribution with finite mean μ and variance σ^2



The result

Theorem. Consider iid gaps between consecutive input keys with finite mean μ and variance σ^2.

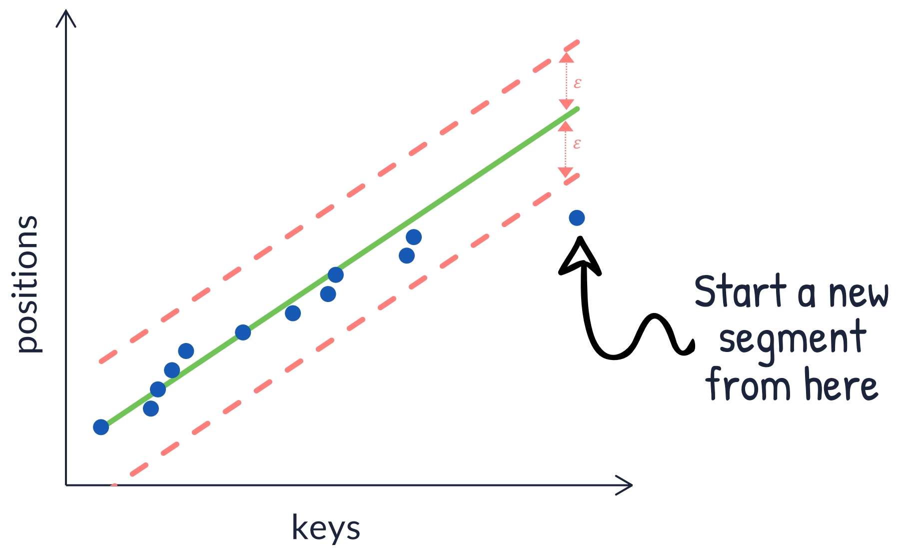
If ε is sufficiently large, the number of segments (≈ the space of a PGM) on n input keys converges to

$$\frac{\sigma^2}{\mu^2}\,\frac{n}{\varepsilon^2}$$

Corollary. Under the assumption above, the PGM-index with $\varepsilon = \Theta(B)$ improves the space of a B-tree from $\Theta(n/B)$ to $O(n/B^2)$

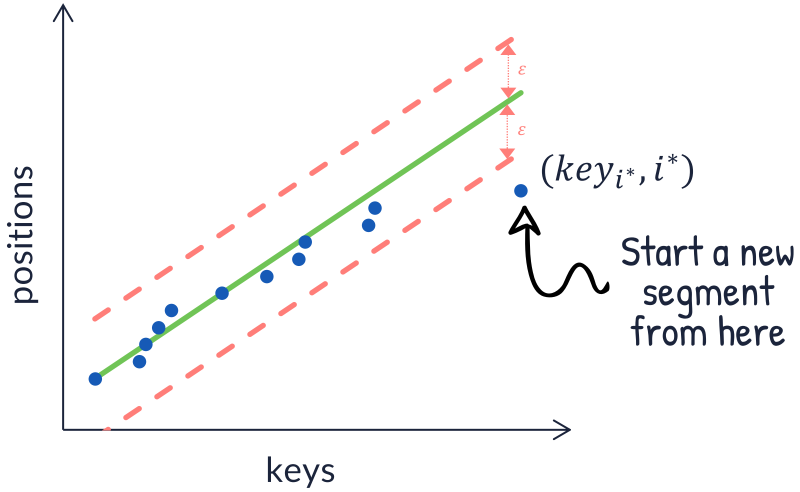
Sketch of the proof

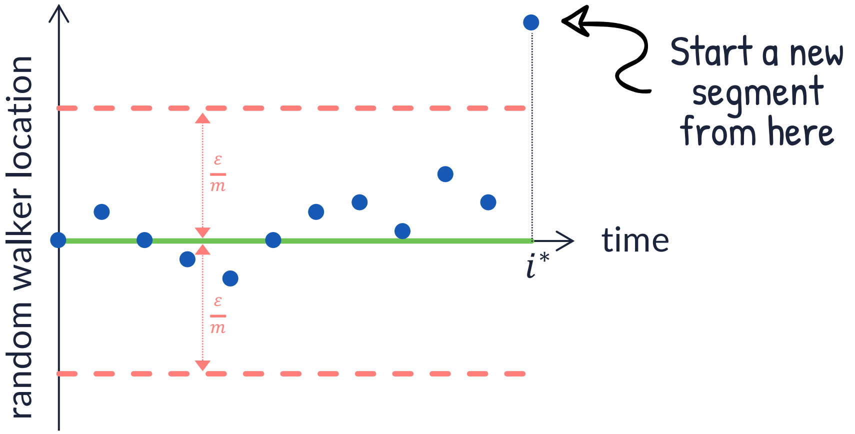
1. Consider a segment on the stream of random gaps and the two parallel lines at distance ε
2. How many steps before a new segment is needed?



Sketch of the proof (2)

3. A discrete-time random walk, iid increments with mean μ
4. Compute the expectation of
$$i^* = \min\{i \in \mathbb{N} \mid (k_i, i) \text{ is outside the red strip}\}$$
i.e. the Mean Exit Time (MET) of the random walk
5. Show that the slope $m = 1/\mu$ maximises $E[i^*]$, giving $E[i^*] = (\mu^2/\sigma^2)\,\varepsilon^2$





ICML
International Conference
On Machine Learning

Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. *Why are learned indexes so effective?* In: Proc. 37th Intl. Conference on Machine Learning (ICML), 2020.

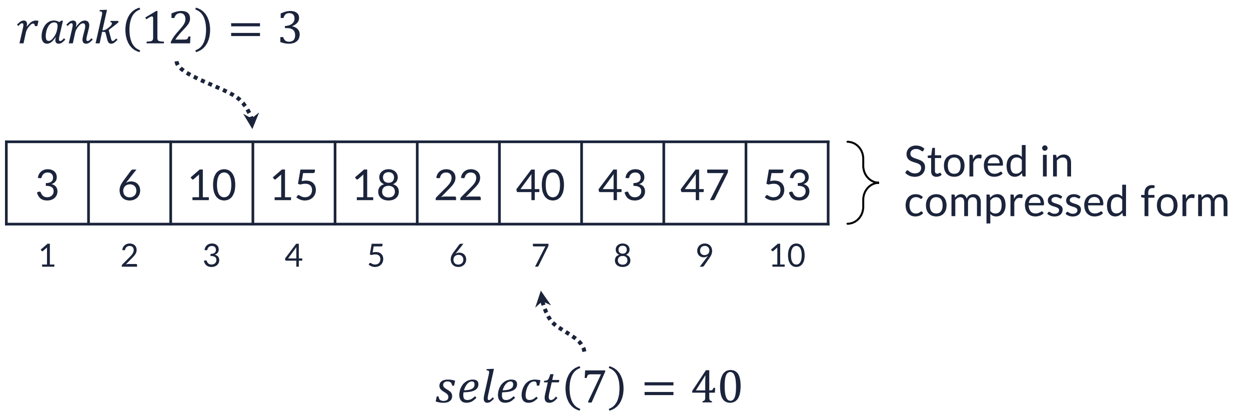
Code available at github.com/gvinciguerra/Learned-indexes-effectiveness

Problem 2

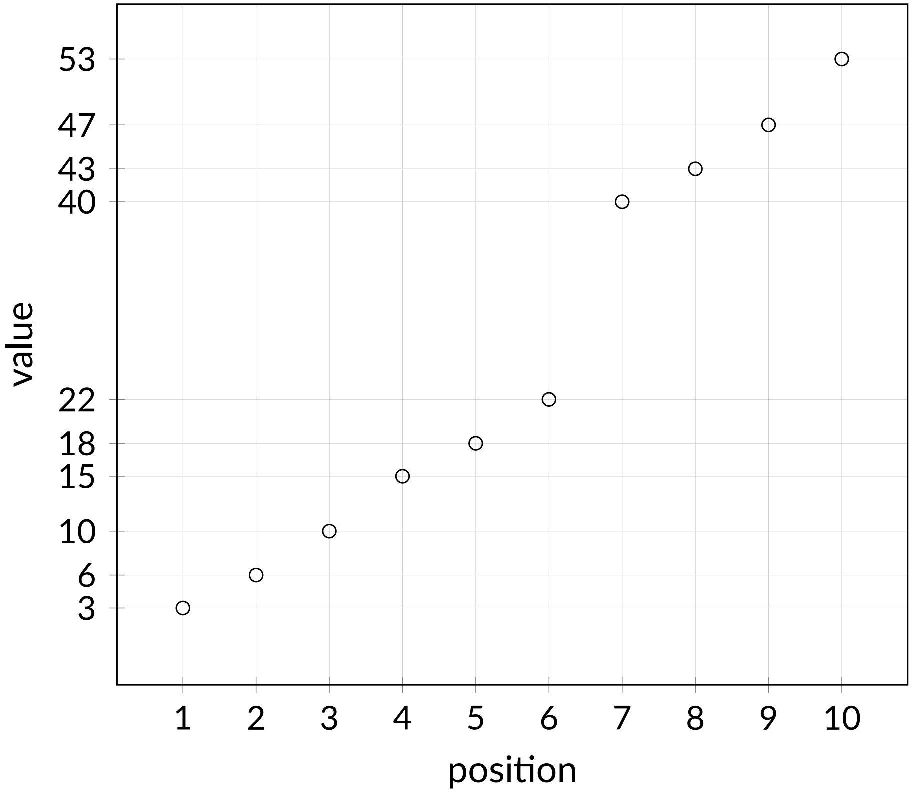
Rank/select dictionaries

Rank/select dictionaries

- Given a set S of n elements over an integer universe $0,1,\ldots,u$
  1. Store them in compressed form
  2. Implement $rank(x)$: number of elements in S which are $\leq x$
  3. Implement $select(i)$: return the ith smallest element in S
- Building block of succinct data structures for texts, genomes, graphs, etc. Very mature field



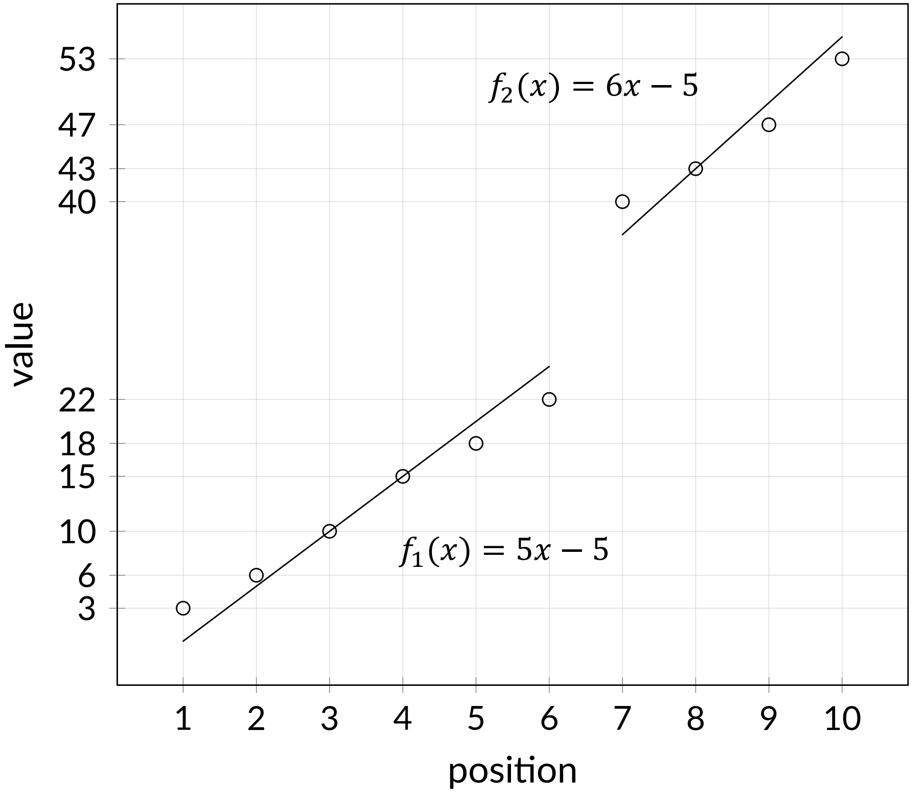
The idea



3	6	10	15	18	22	40	43	47	53
1	2	3	4	5	6	7	8	9	10

The idea: data = segments

Represent integers with
an information loss of ε

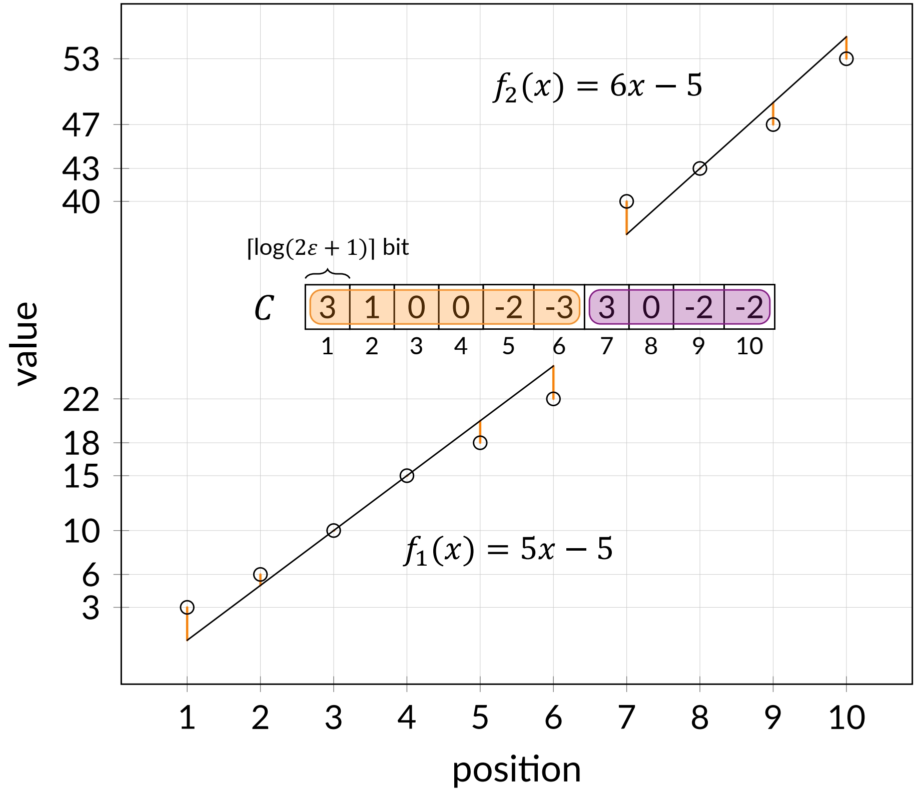


3	6	10	15	18	22	40	43	47	53
1	2	3	4	5	6	7	8	9	10

The idea: data = segments + corrections

Represent integers with an information loss of ε

Complement the approximations to recover the original set



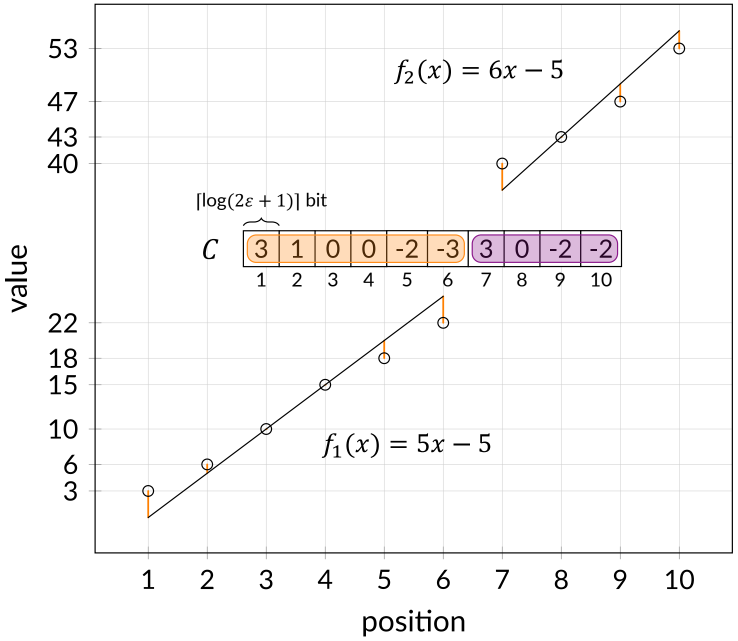
3	6	10	15	18	22	40	43	47	53
1	2	3	4	5	6	7	8	9	10

$$x_i = \alpha \cdot i + \beta + C[i]$$

The LA-vector

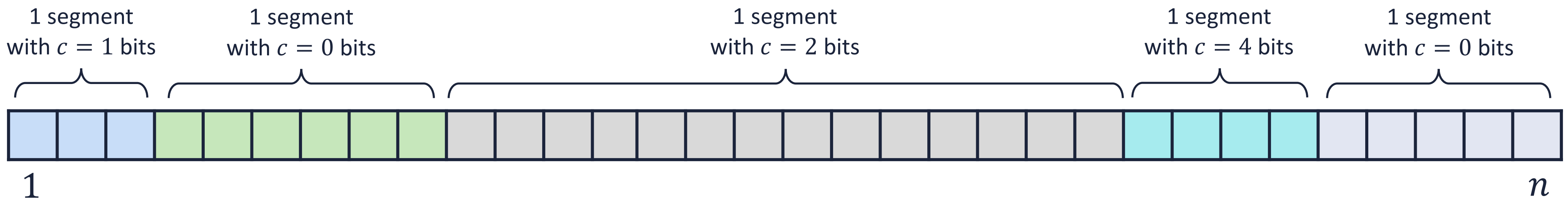
Given c bits for the corrections (i.e. allow an error of $\varepsilon = 2^{c-1} - 1$) :

- Space $O(\ell) + nc$ bits, where ℓ is the number of segments
- Select in $O(1)$ time, in additional $n + o(n)$ bits
- Rank in $O(\log\log(u/\ell) + c)$ time, in additional $O((\ell/2^c)\log u)$ bits



How to minimise the space?

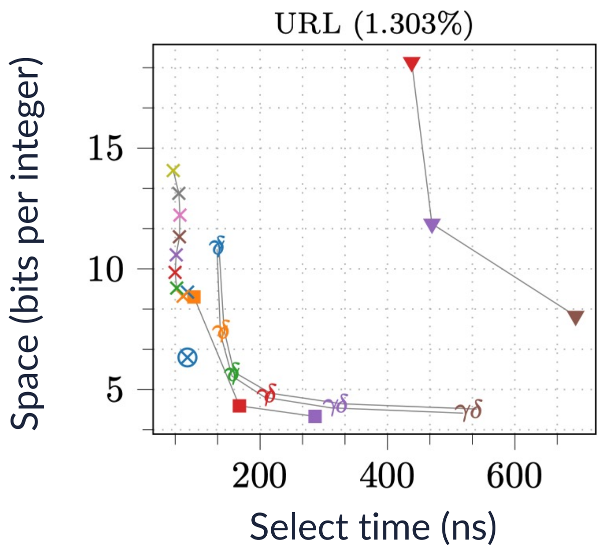
- Space $O(\ell) + nc$ bits. How to choose c without increasing ℓ?
- Partition the input according to its linearities, choose a different c for each chunk, minimise the overall space

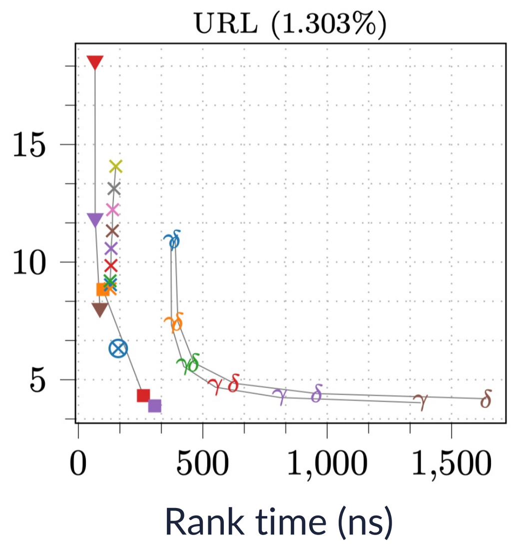


- Reduction to the shortest path problem on ad hoc graphs
  - Optimal takes $O(n^2 \log u)$ time and $O(n \log u)$ space
  - Greedy takes $O(n \log u)$ time and $O(n)$ space
  - We prove that the greedy adds a constant factor more space wrt the optimal

Experiments on LA-vector

- Tested on DNA, 5Gram, URLs, and inverted lists
- Compared to well-engineered rank-select structures (Elias-Fano, RRR-vector, Gap-encoded vector) implemented in Gog's SDSL
- Faster select and competitive rank





Symposium on Algorithm Engineering and Experiments

Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra.
A “learned” approach to quicken and compress rank/select dictionaries.
To appear in: Proc. SIAM Symposium on Algorithm Engineering and Experiments (ALENEX), 2021.

Code available at github.com/gvinciguerra/la_vector

Conclusions

Wrap up

- New way to look at the data based on geometric considerations
- Introduced two theoretically and practically efficient structures that exploit the approximate linearity of the data
  - The *PGM-index* for the predecessor search problem
  - The *LA-vector* for the rank/select dictionary problem
- Studied the theoretical grounds of the structures that use approximate linearity

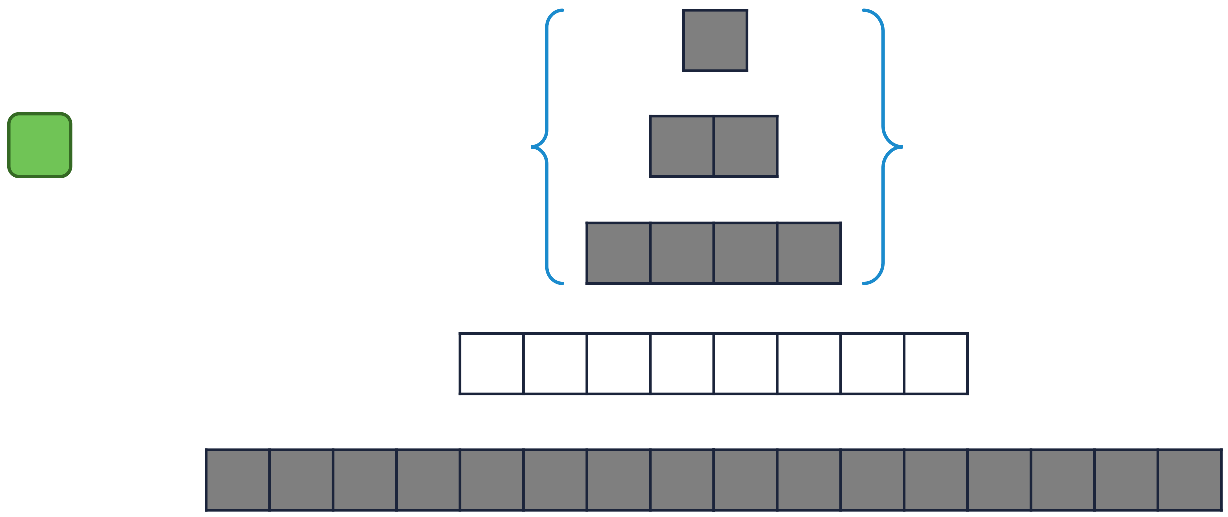
Ongoing and future research work

- Apply these ideas to string indexing and compression, and possibly other problems
- Study and experiment more sophisticate models (i.e. nonlinear) while retaining the same theoretical guarantees on the error
- Integrate these structures into a real data system
- Study the relation between approximate linearity and existing compressibility measures such as Shannon’s entropy

Extra slides

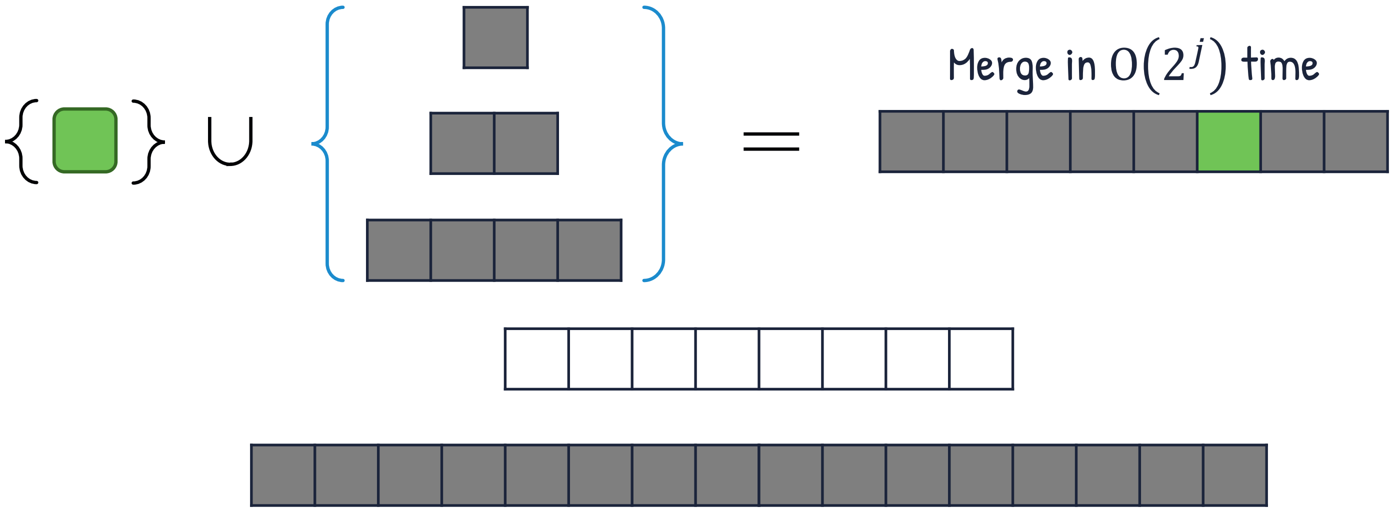
Sketch of the fully-dynamic PGM

- Define $b = O(\log n)$ PGM-indexes either empty or of sizes $2^0, 2^1, \dots, 2^b$
- An insert merges the first $j - 1$ full levels into the first free level j



Sketch of the fully-dynamic PGM

- Define $b = O(\log n)$ PGM-indexes either empty or of sizes $2^0, 2^1, \ldots, 2^b$
- An insert merges the first $j-1$ full levels into the first free level j



Sketch of the fully-dynamic PGM

- Define $b = O(\log n)$ PGM-indexes either empty or of sizes $2^0, 2^1, \dots, 2^b$
- An insert merges the first $j - 1$ full levels into the first free level j