Kickoff meeting PRIN:
Attività di ricerca UNIPI

~

Paolo FERRAGINA, **Giorgio VINCIGUERRA**

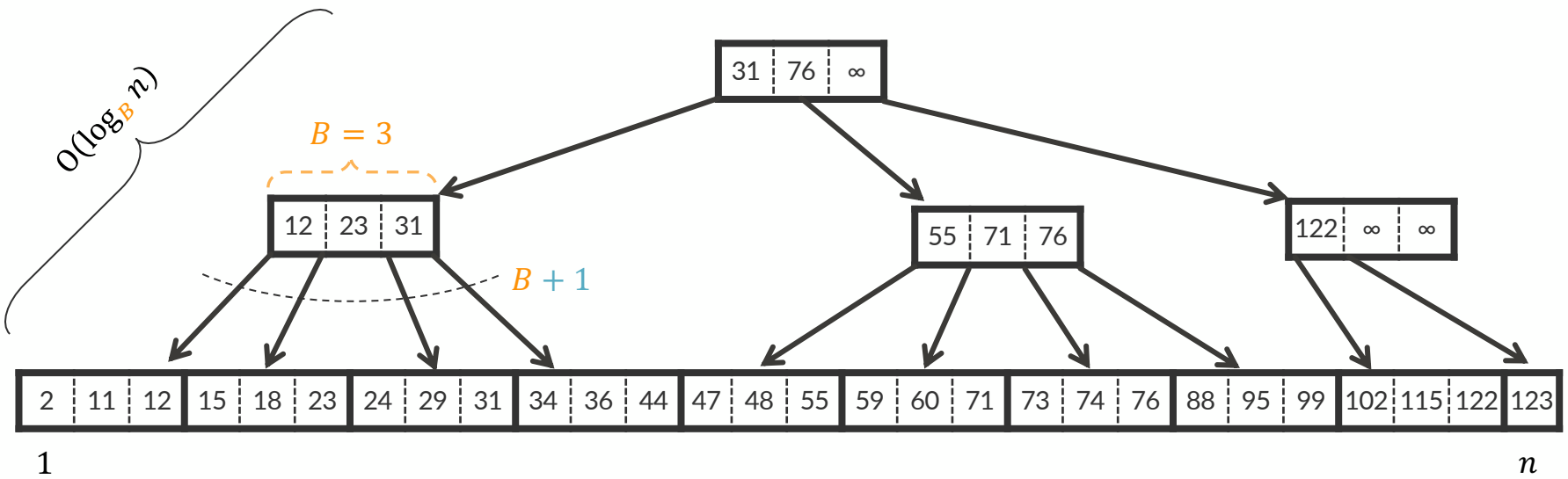
Università di Pisa

Outline

~

1. Learned indexes
2. PGM-index
   - Construction
   - Distribution-aware variant
   - Compressed variant
   - Experiments
3. Open problems

Predecessor and range search in ext. memory

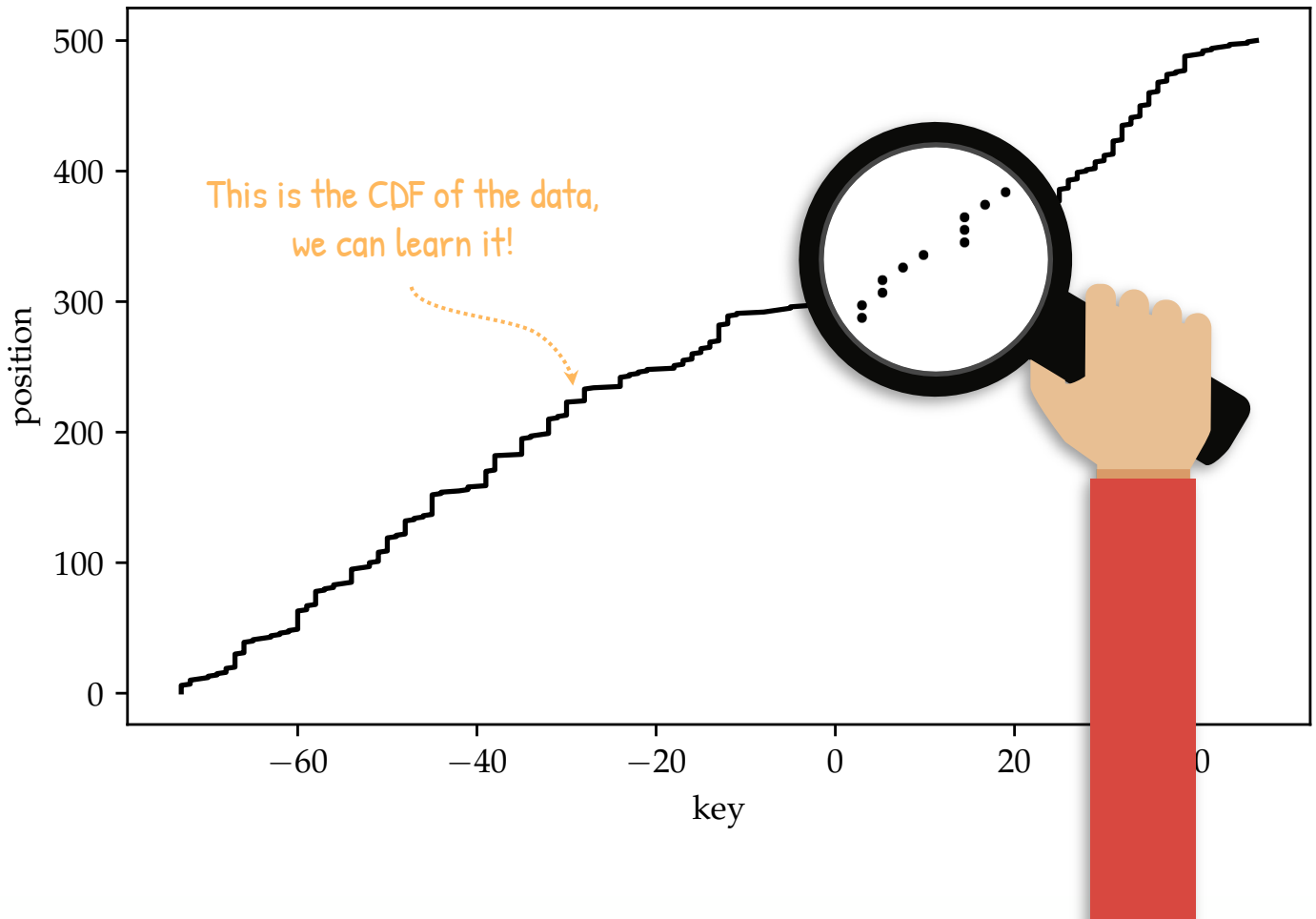


"B-trees have become, de facto, a standard for file organization" [ACM CSUR '79]

This is still true today

A closer look at the data

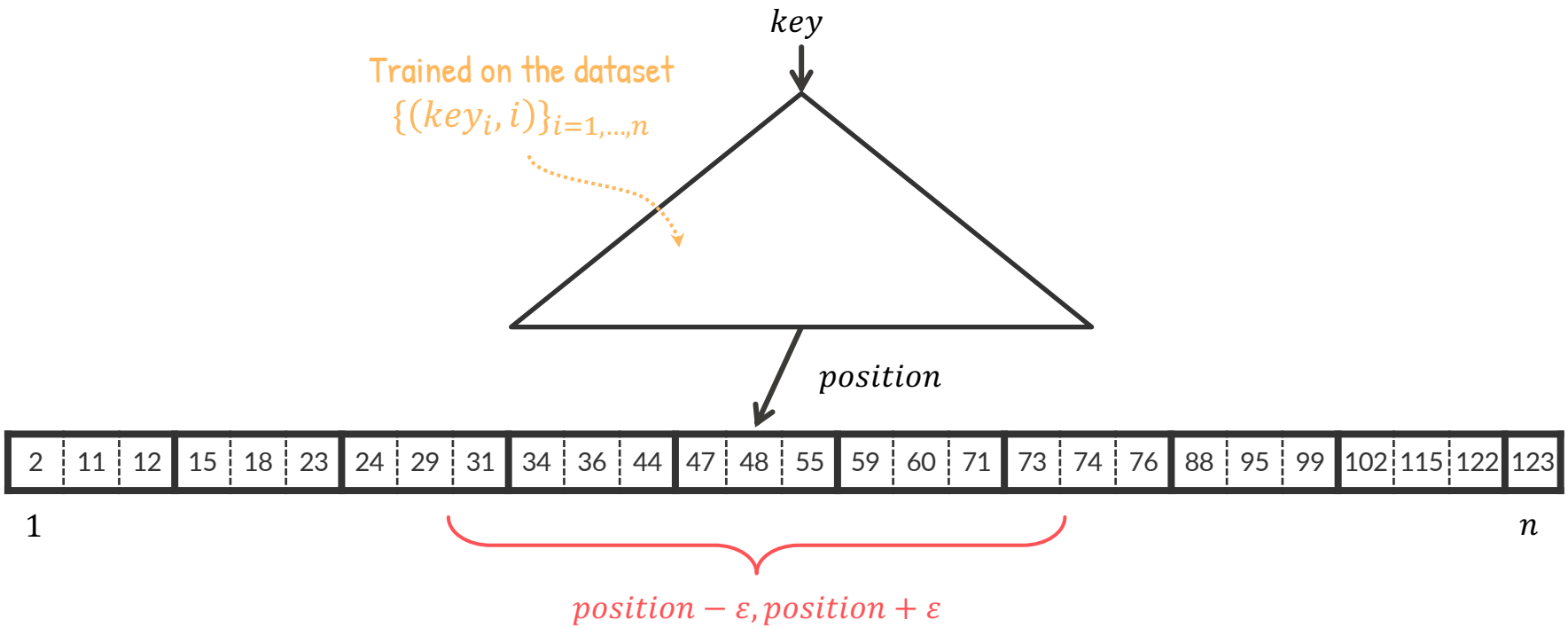
keys:	-84	-83	.....	57	59
positions:	1	2			$n = 500$



B-trees are machine learning models

~

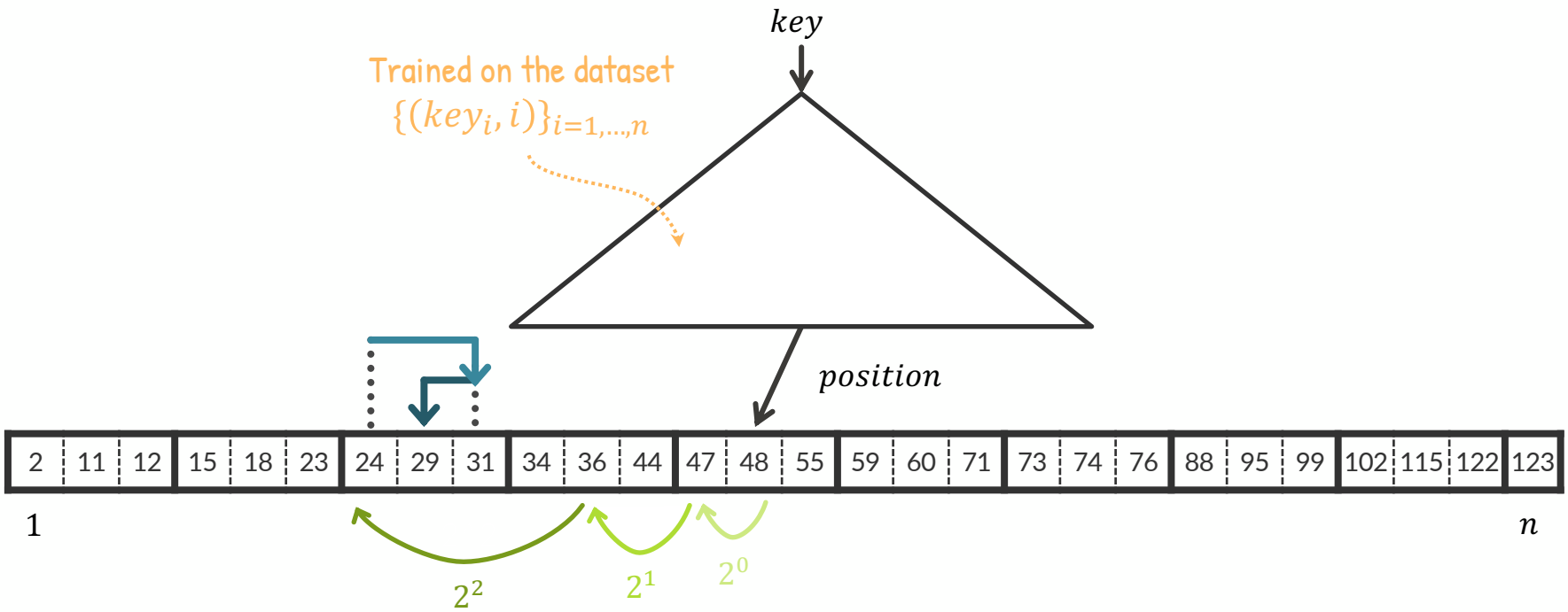
"All existing index structures can be replaced with other types of models, including deep-learning models, which we term learned indexes." [SIGMOD '18]



B-trees are machine learning models

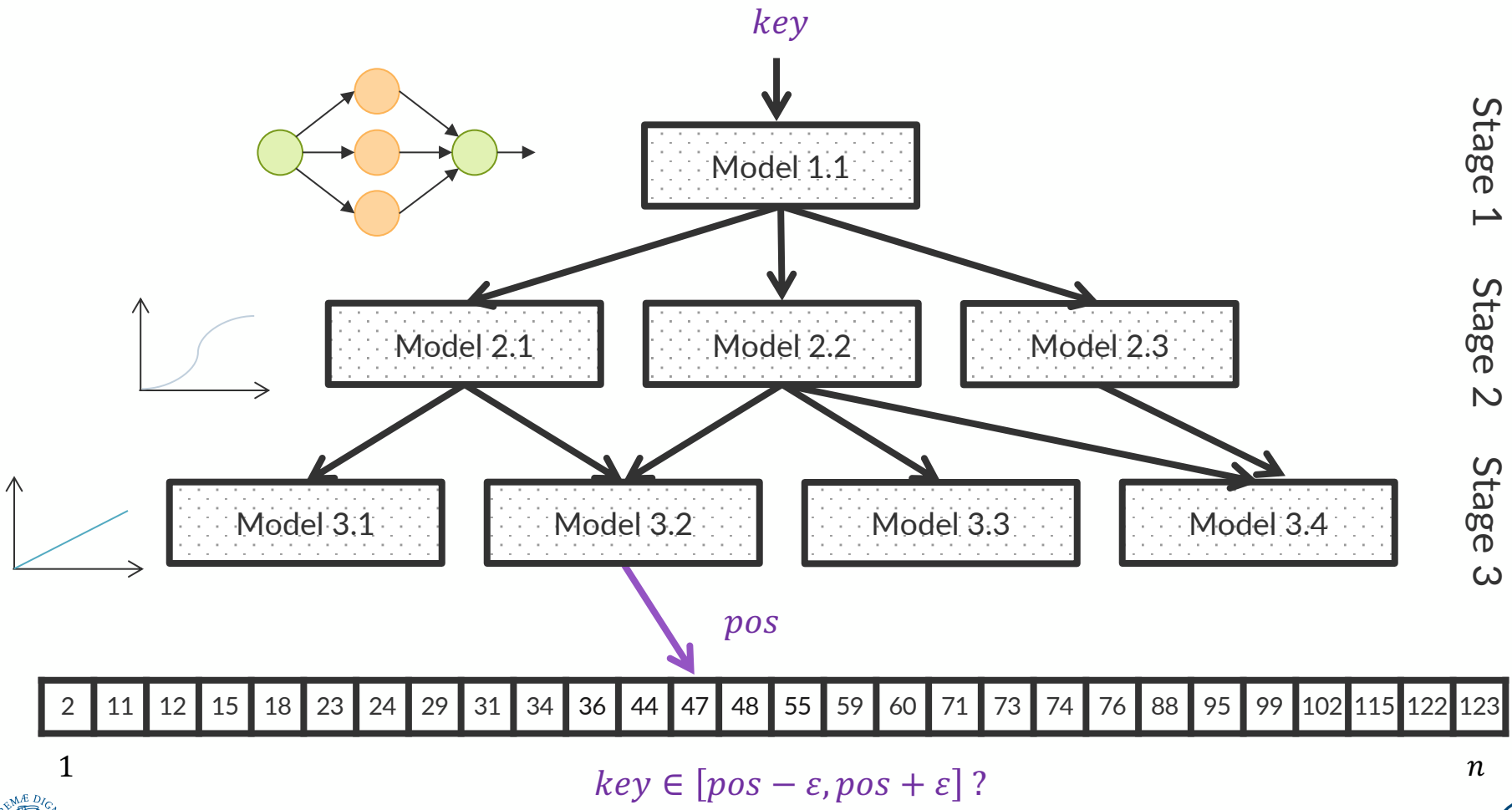
~

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



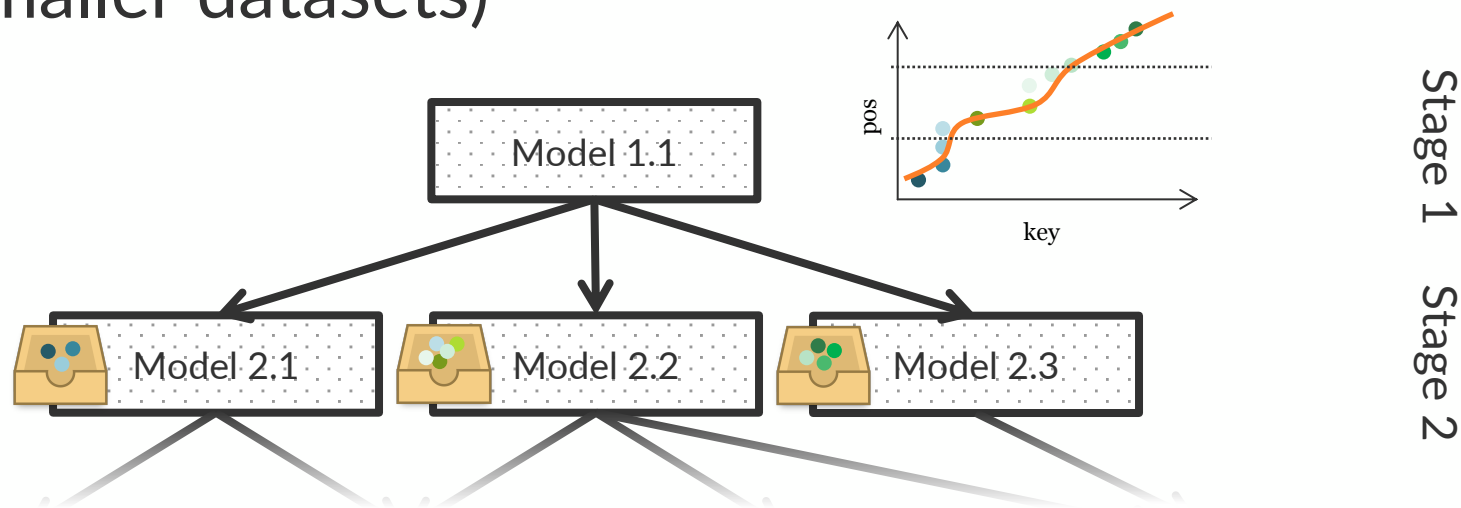
The Recursive Model Index (RMI)

~



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)



Performance of RMI

~

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

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)

Main ingredients of our PGM-index

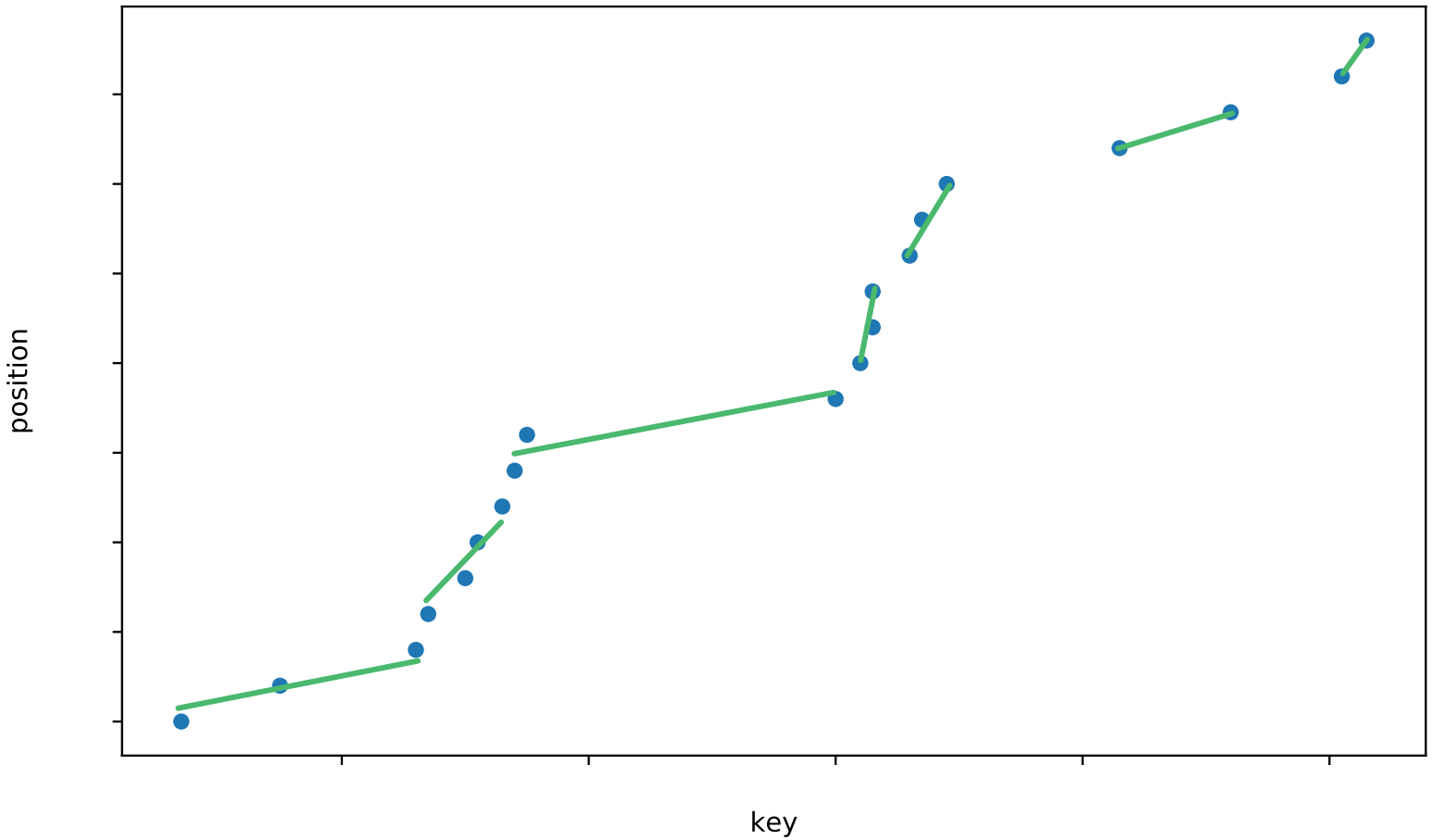
Piecewise Geometric Model

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

~

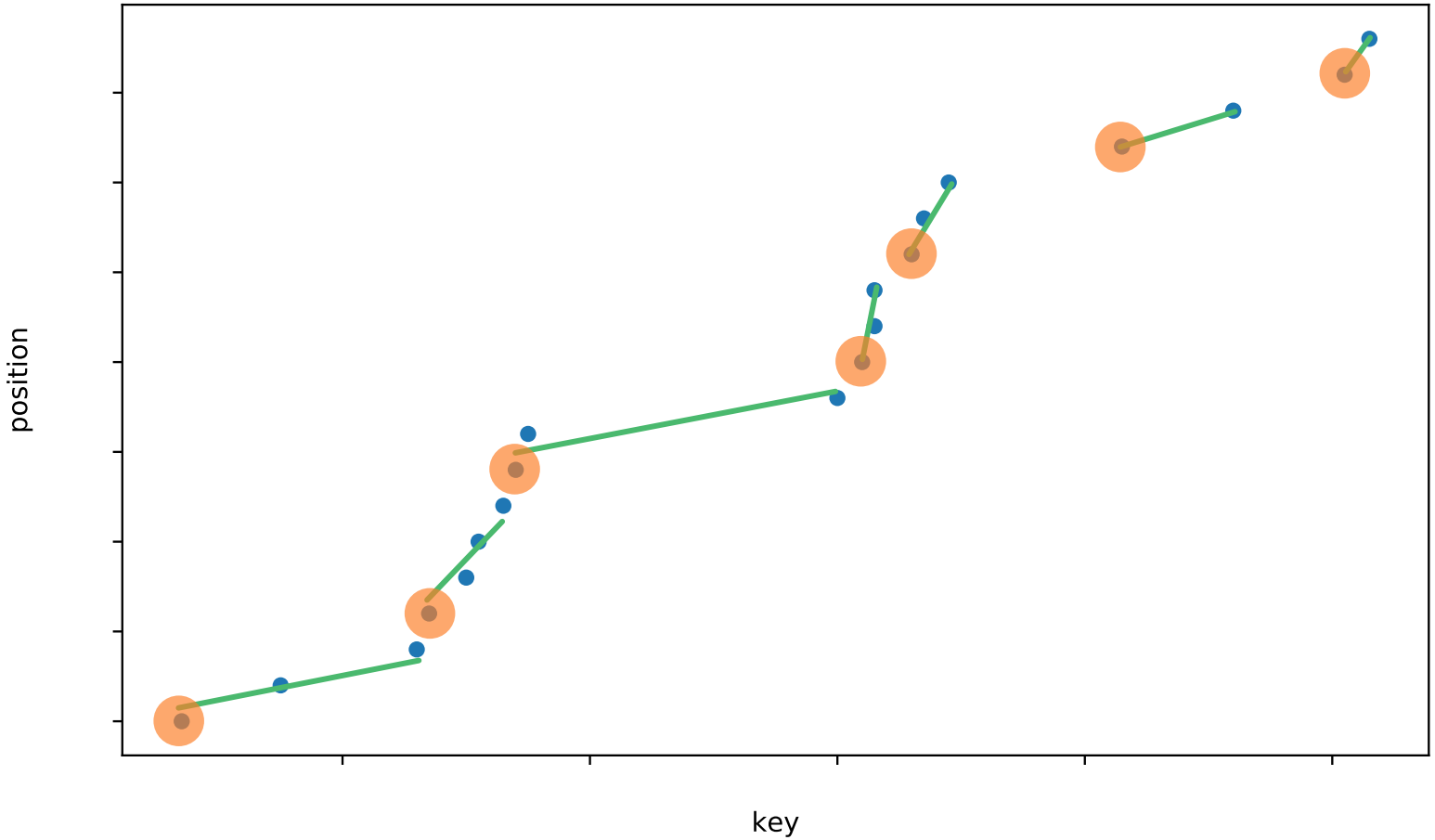
Compute the **optimal** piecewise linear approx with **guaranteed error** ε in $O(n)$



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

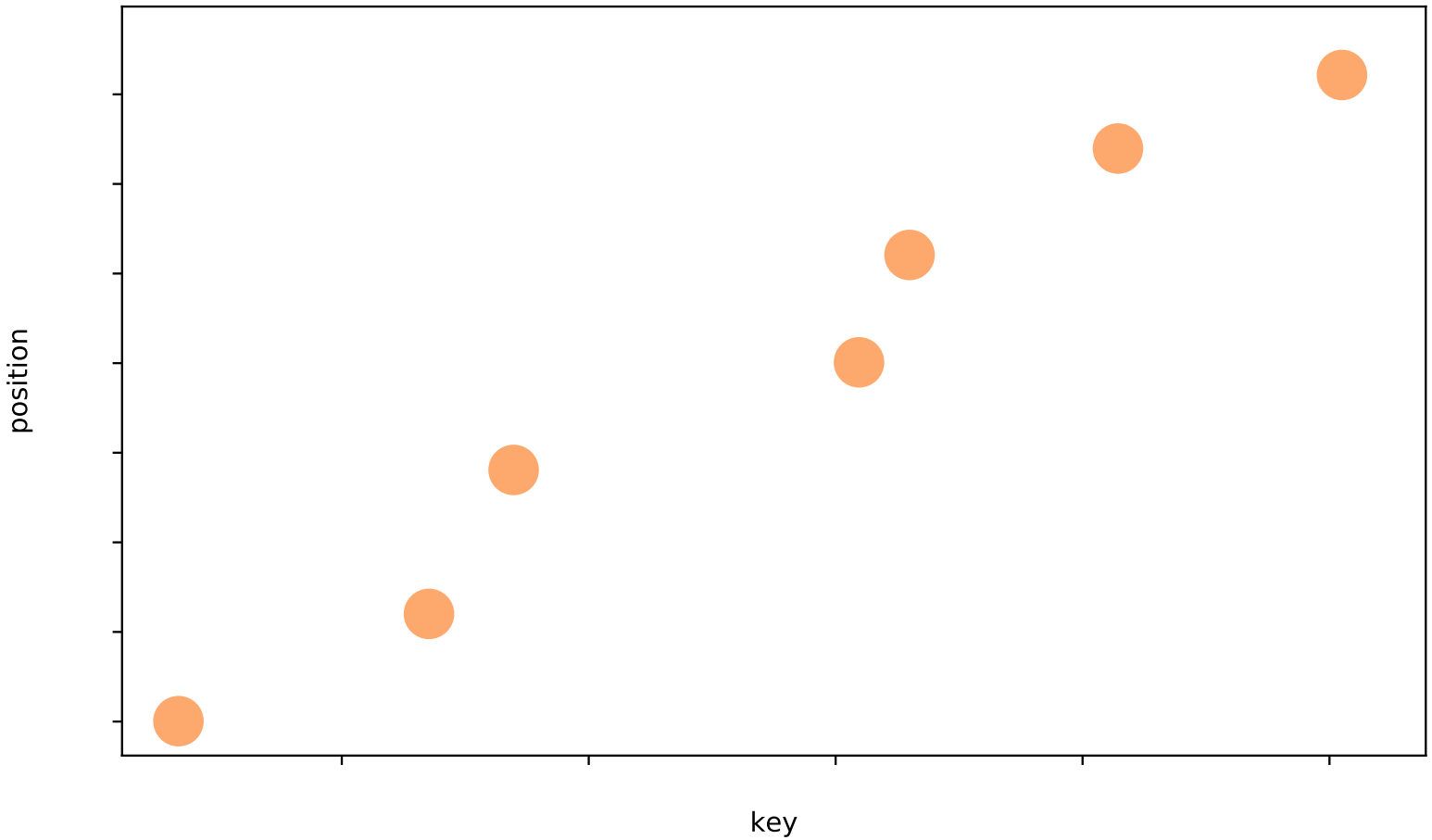
2	11	12	15	18	23	24	29	31	34	36	44	47	48	55	59	60	71	73	74	76	88	95	99	102	115	122	123

1 n

PGM-index construction

~

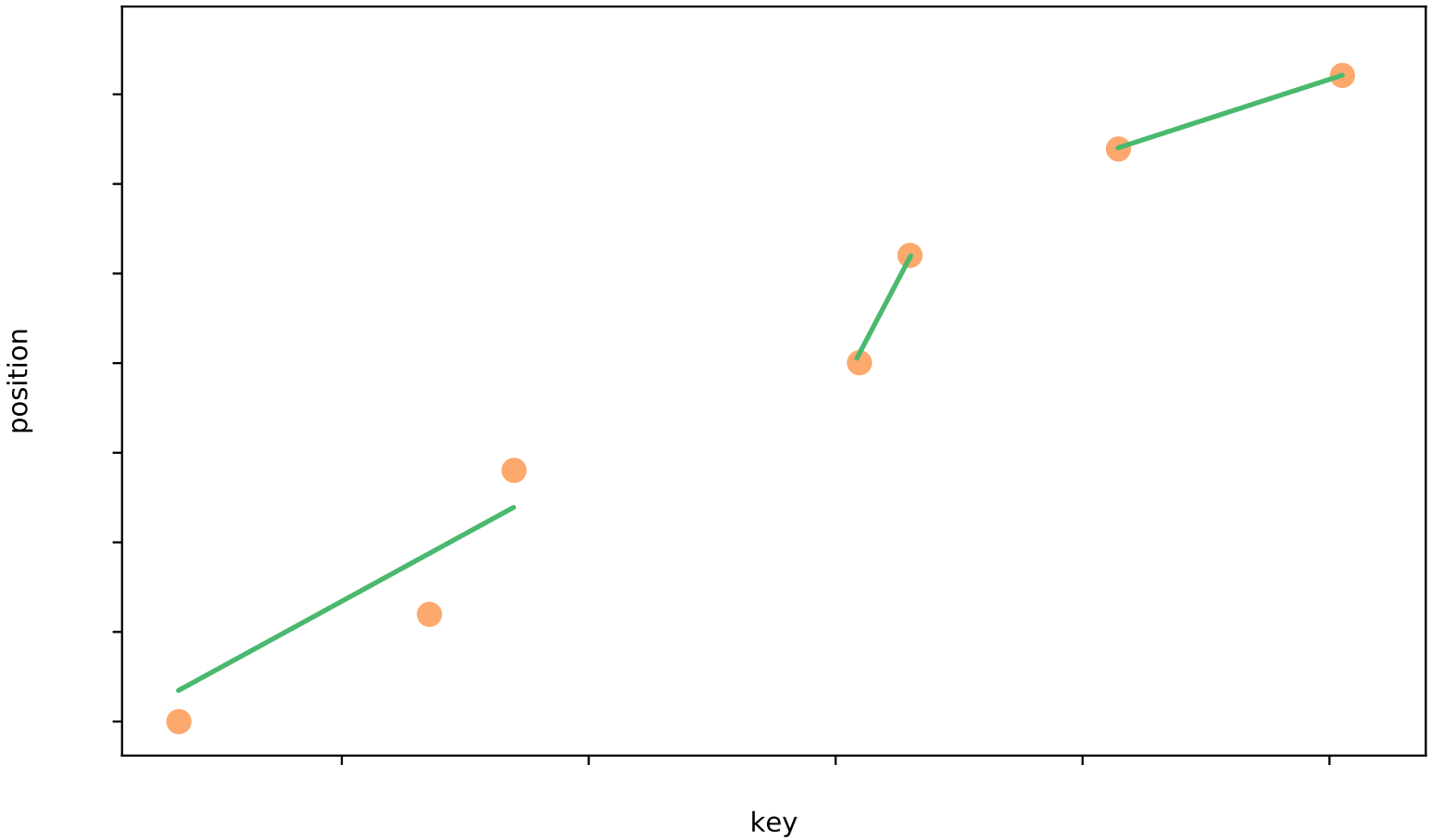
Drop all the points except $s_i.key$



PGM-index construction

~

... and repeat!



Memory layout of the PGM-index

~

Level[0]

(2, sl, ic)

Level[1]

(2, sl, ic)	(48, sl, ic)	(102, sl, ic)

Level[2]

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

1 m

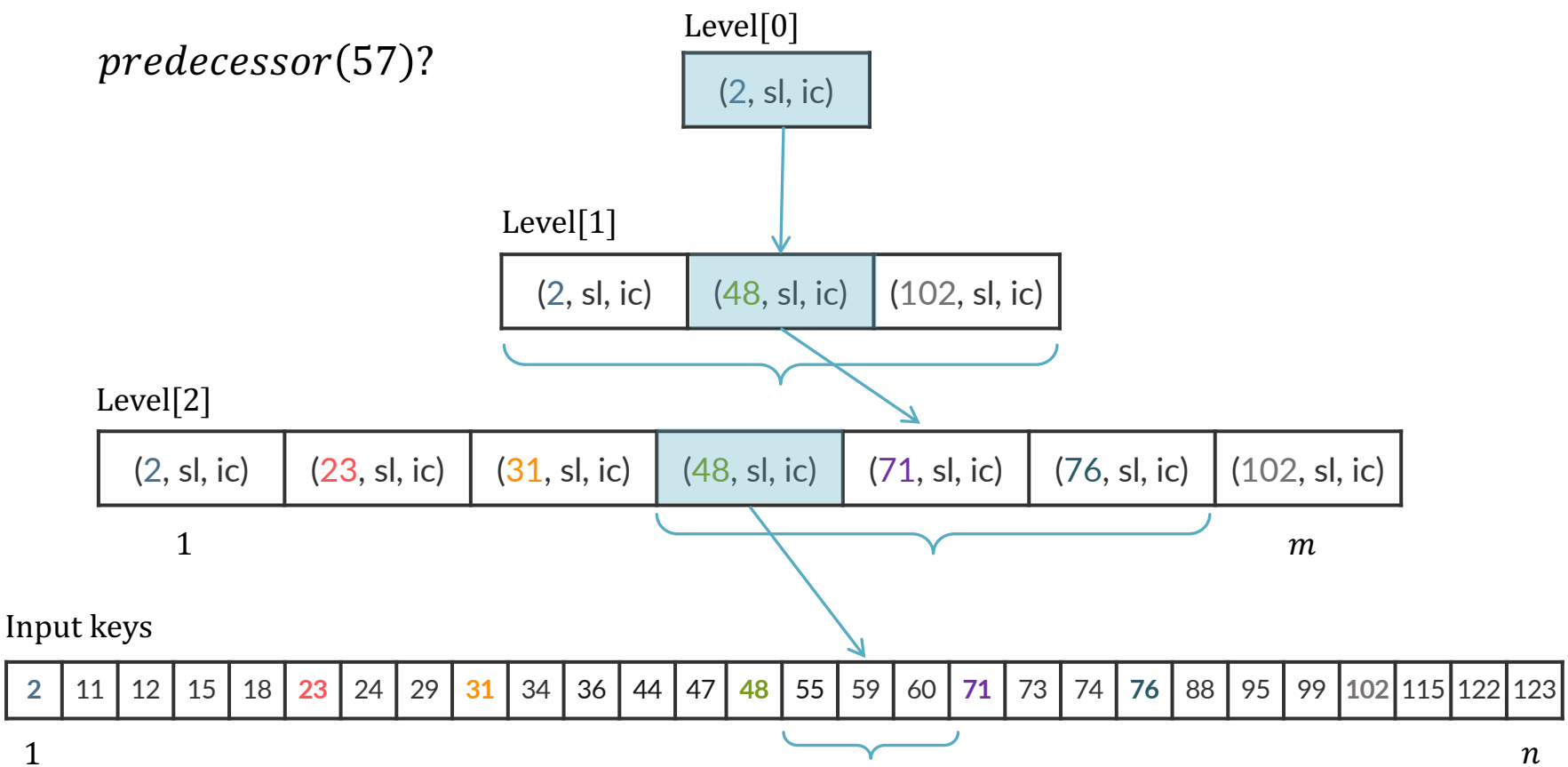
Input keys

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

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

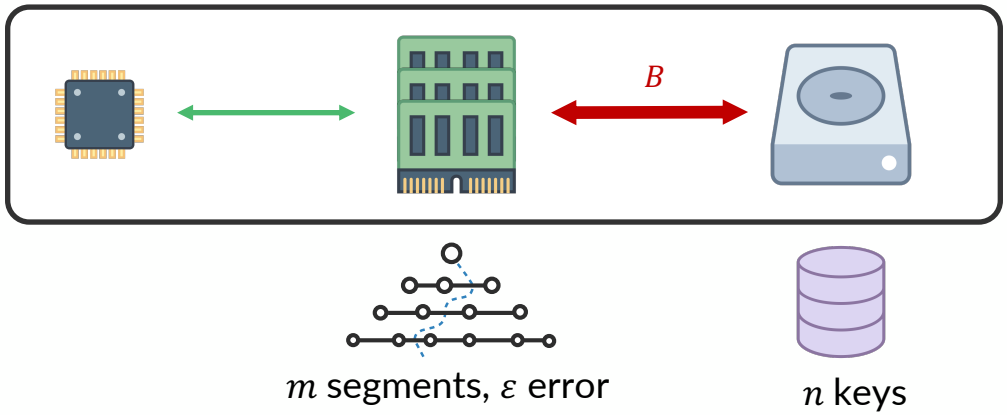
$predecessor(57)$?



Some asymptotic bounds

~

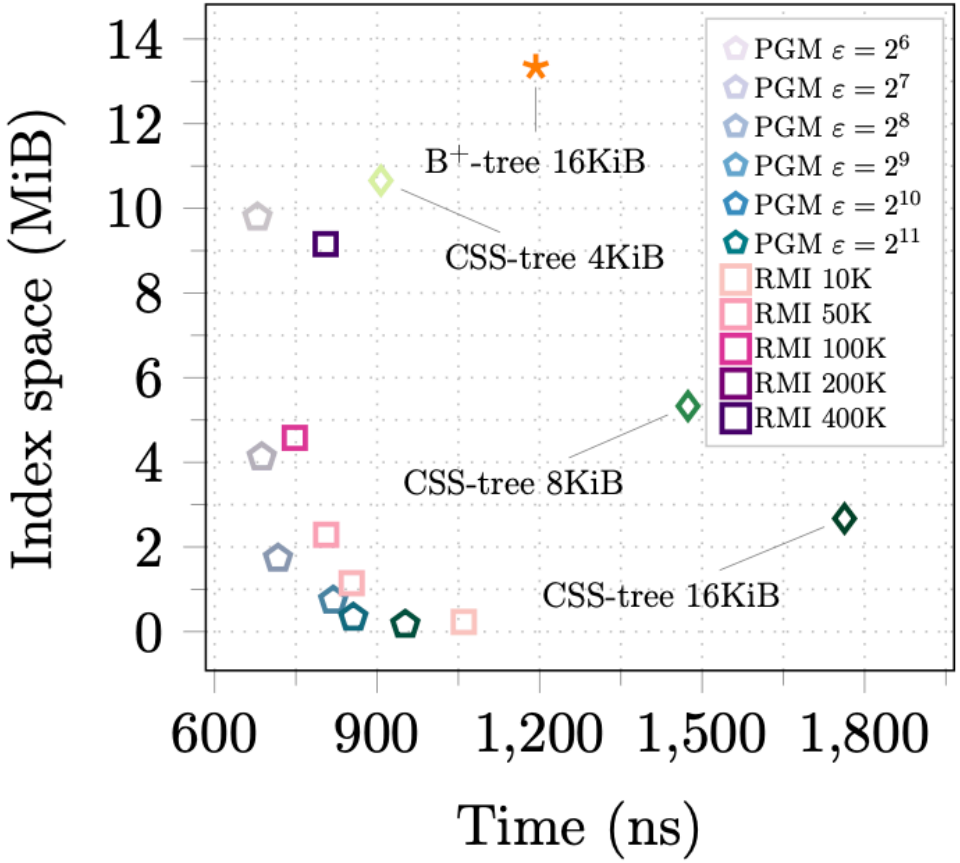
Data Structure	Space of index	RAM model Worst case time	EM model Worst case I/Os	EM model Best case I/Os
Multiway tree	$\Theta(n)$	$O(\log n)$	$O(\log_B n)$	$O(\log_B n)$
RMI	Fixed	$O(?)$	$O(?)$	$O(1)$
PGM-index	$\Theta(m)$	$O(\log m)$ $m \leq n/(2\varepsilon)$	$O(\log_c m)$ $c \geq 2\varepsilon = \Omega(B)$	$O(1)$



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

ε = 128

4 MiB (−85×)

3.1 s to construct (single-threaded)

Distribution-aware predecessor problem

~

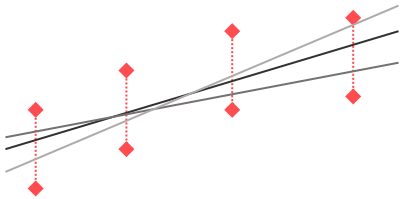
Given n pairs (k_i, p_i) where p_i is the probability of querying k_i, build a data structure that answer predecessor queries in $O(\log 1/p_i)$

Theorem. The Distribution-Aware PGM-index solves the distribution-aware predecessor problem in $O(m)$ space and $O(\mathcal{H})$ average time, where $\mathcal{H}$ is the entropy of the query distribution and m is the number of segments in the PGM-index

Compressed PGM-index

~

1. Slopes of the m segments can be chosen with a certain amount of freedom



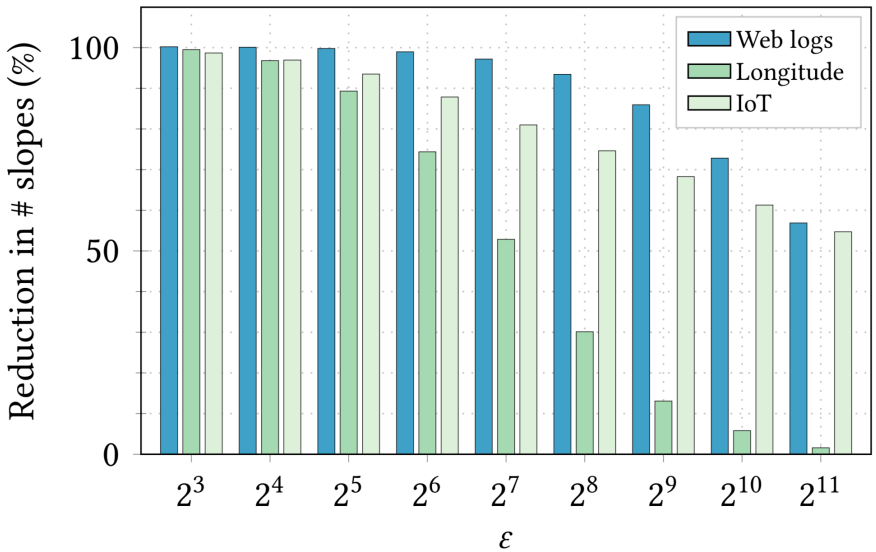
2. Minimise the number of distinct slopes from m to t (a "clustering of slopes")
3. Store t slopes in a table. Add a pointer for each segment to an element of this table

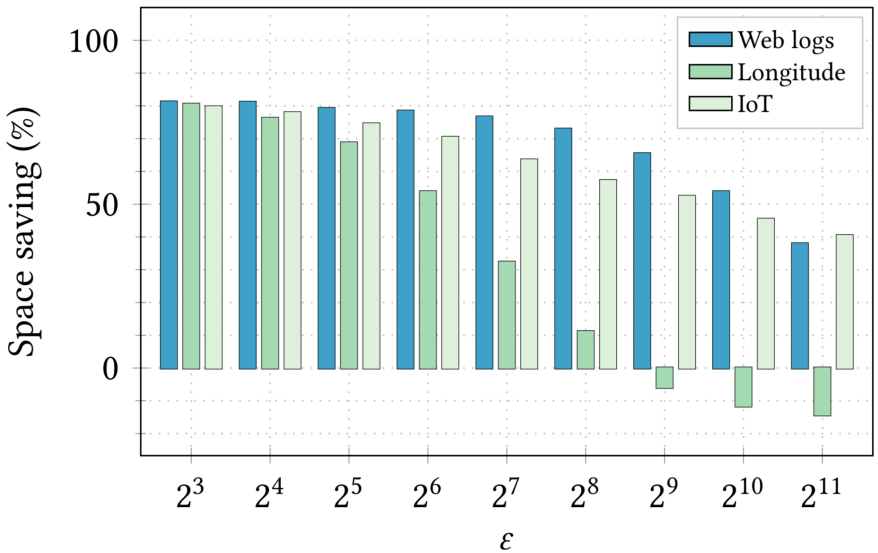
Compressed PGM-index (cont)

~

1. The m intercepts are increasing
2. Round each intercept ($\varepsilon \to \varepsilon + 1$)
3. Universe size is n
4. Store in $m \log \frac{n}{m} + 2m$ bits with e.g. [OkanoharaSadakane07], add $o(m)$ bits for constant time access

Slope compression in practice



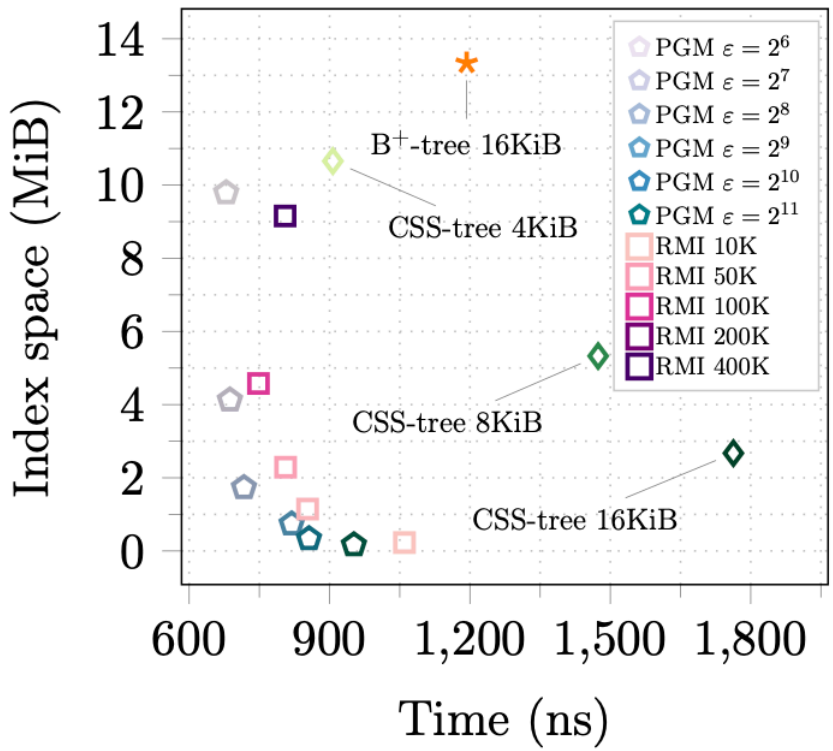


e.g. when $\varepsilon = 64$, queries are 14% slower on the biggest dataset (Weblogs) but the space footprint is reduced by 52%

How to explore this space of trade-offs?

~

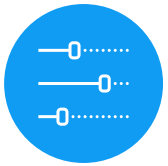
Given a space bound S, find efficiently the index that minimizes the query time within space S and vice versa



Multicriteria data structures

~

A multicriteria data structure is defined by a family of data structures and an optimisation algorithm that selects the best data structure in the family within some computational constraints



FAMILY
PGM-indexes ∀ε

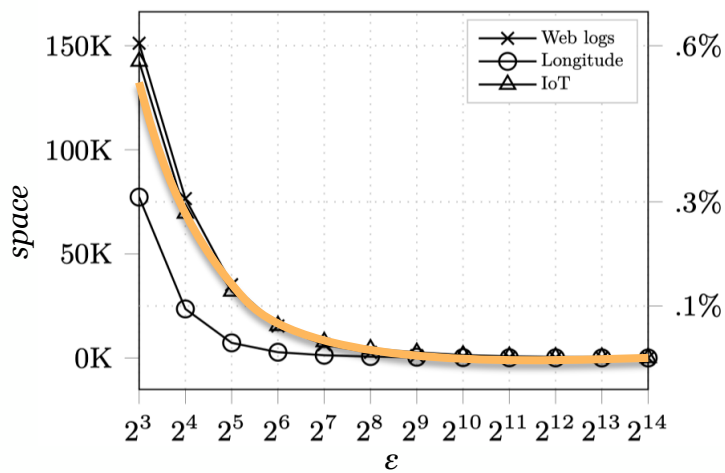
CONSTRAINTS
Space & Time

OPTIMISATION
???

The Multicriteria PGM-index

~

1. We designed a cost model for the space $s(\varepsilon)$ and the time $t(\varepsilon)$
2. … but we don't have a closed formula for $s(\varepsilon)$, it depends on the input array
3. We fit $s(\varepsilon)$ with a power law of the form $a\varepsilon^{-b}$
4. In practice, given a space (time) bound, it finds the fastest (most compact) index for 715M keys in < 1 min



Open problem 1

~

Is there any benefit in using more powerful models (e.g. NN)?

a. Running an index on GPU is too slow because of the cost of transferring data
b. Batching queries could be a possible solution
c. Trading off the increased latency with throughput should be studied, possibly within the multicriteria framework

Open problem 2

~

How can we handle insertion and deletions in the PGM-index?

a. Use a buffer for new items to amortise index reconstructions (we're experimenting this in Redis)
b. Use data structure dynamisation [Overmars]
c. Use gaps in the data array to accommodate insertions and reconstruct single segments, not the whole index

Open problem 3

~

What is the theoretical justification for the improvements of learned indexes?

a. In practice the space decreases as a power law $a\varepsilon^{-b}$ with b ≈ 1.2
b. Assuming a distribution of the input keys, can we bound the space (# segments) of a PGM-index?

Open problem 4

~

Can we generalise the PGM-index to string keys of arbitrary length?

Open problem 5

~

Can we efficiently synthesise hybrid indexes that combine NN, linear models, compressed data structures (...)?