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

Paolo Ferragina

Giorgio Vinciguerra
The predecessor search problem

• Given \( n \) sorted input keys (e.g. integers), implement \( \text{predecessor}(x) = \text{“largest key } \leq x\text{”} \)

• 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!)

\[
\text{predecessor}(36) = 36
\]

\[
\text{predecessor}(50) = 48
\]
Indexes

key = 36

position = 11

(values associated to keys are not shown)
Input data as pairs \((key, position)\)

Ao et al. [VLDB 2011]
Input data as pairs \((key, position)\)

Ao et al. [VLDB 2011]
Learned indexes

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

Binary search in 
\([\text{position} - \text{error}, \text{position} + \text{error}]\)

Ao et al. [VLDB 2011], Kraska et al. [SIGMOD 2018]
The problem with learned indexes

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

- Unpredictable latency
- Too much I/O when data is on disk
- Very slow to train
- Unscalable to big data
- Blind to the query distribution
- Vulnerable to adversarial inputs and queries
- Must be tuned for each new dataset
Introducing the PGM-index

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

- Predictable latency
- Constant I/O when data is on disk
- Very fast to build
- Scalable to big data
- Query distribution aware
- Resistant to adversarial inputs and queries
- No additional tuning needed
Ingredients of the PGM-index

Opt. piecewise linear model
Fast to construct, best space usage for linear learned indexes

Fixed model “error” $\varepsilon$
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
Step 1. Compute the optimal piecewise linear $\varepsilon$-approximation in $O(n)$ time
**PGM-index construction**

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

**Step 2.** Store the segments as triples $s_i = (key, slope, intercept)$
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 $2\varepsilon + 1$

Segments

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

Binary search in

$$[pos - \varepsilon, pos + \varepsilon]$$
PGM-index construction

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

**Step 2.** Store the segments as triples $s_i = (\text{key}, \text{slope}, \text{intercept})$

**Step 3.** Keep only $s_i$. key
PGM-index construction

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

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

**Step 3.** Keep only $s_i$. $key$
PGM-index construction

**Step 1.** Compute the optimal piecewise linear $\varepsilon$-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

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

It can also be constructed in a single pass

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

It can also be constructed in a single pass
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

$\varepsilon = 1$

The PGM-index is never 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×)

Weblogs (714M keys, 8-byte keys, 128-byte payloads)

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

1 billion uniform key-value pairs, 8-byte keys, 8-byte values

- Dynamic PGM
- B'-tree<128>
- B'-tree<256>
- B'-tree<512>
- B'-tree<1024>

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

1 billion uniform key-value pairs, 8-byte keys, 8-byte values

<table>
<thead>
<tr>
<th>B⁺-tree page size</th>
<th>Index size</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-byte</td>
<td>5.65 GB</td>
</tr>
<tr>
<td>256-byte</td>
<td>2.98 GB</td>
</tr>
<tr>
<td>512-byte</td>
<td>1.66 GB</td>
</tr>
<tr>
<td>1024-byte</td>
<td>0.89 GB</td>
</tr>
</tbody>
</table>

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 $B$

$\begin{array}{cccc}
  k_1 & k_2 & \cdots & k_B \\
\end{array}$

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 $1/B$

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 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
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.