Learned indexes, the PGM-index, and the coding challenge

Algorithm Engineering A.Y. 2020/21
Università di Pisa

Giorgio Vinciguerra
PhD student & Teaching assistant
The predecessor search problem

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

• Range queries in DBs, lists intersection (conjunctive queries in search engines), IP routing...

• Harder than the dictionary problem: if you need to support only exact searches just use Cuckoo hashing (§8.6 of the notes)

\[
\begin{array}{cccccccccccccccccccc}
\end{array}
\]

$\text{predecessor}(36) = 36$

$\text{predecessor}(50) = 48$
The basic solution, binary search

• Works well if the keys fits in the CPU cache
• Incurs costly cache misses if keys are in memory ($\approx 100 \text{ ns per miss}$)
• Incurs very costly disk transfers if keys are on disk ($\approx 16 \mu\text{s per random I/O on SSDs}$)
“B-trees have become, de facto, a standard for file organization”
— Comer. Ubiquitous B-tree. ACM Computing Surveys. ‘79

(values associated to keys are not shown)
A different look at the data

Map data to points \((key, position)\)

Indexes an arbitrary number of keys in \(O(1)\) space and time

\[ pos = 0.5 \times key \]
A different look at *realistic* data
Learned indexes

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

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?
An optimal solution: the PGM-index

- Opt. piecewise linear model
  Fast to construct, captures non-linearities

- Fixed model “error” $\varepsilon$
  Control the size of the search range

- Recursive design
  Adapt to the memory hierarchy
Step 1. Compute the optimal piecewise linear $\varepsilon$-approximation in $O(n)$ time.
PGM-index construction

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

**Step 2.** Store the segments as triples $s_i = (key, slope, intercept)$
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)
- (23, sl, ic)
- (31, sl, ic)
- (48, sl, ic)
- (71, sl, ic)
- (88, sl, ic)
- (122, sl, ic)
- (145, sl, ic)

Binary search in $[pos - \varepsilon, pos + \varepsilon]$
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

\[ s_i = (\text{key}, \text{slope}, \text{intercept}) \]

Step 3. Keep only \( s_i \).

2 23 31 48 71 88 122 145
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

**Step 4.** Repeat recursively
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

Step 4. Repeat recursively.
Memory layout of the PGM-index
Predecessor search with $\varepsilon = 1$

- $B = \text{disk page-size}$
- Set $\varepsilon = \Theta(B)$ for queries in $O(\log_B n)$ I/Os
- $O(n/\varepsilon)$ space

The PGM-index is never worse in time and space than a B-tree.
The PGM-index, in practice

As fast as a static cache-optimised B-tree but 80× more compressed

Up to 3× faster than a dynamic B-tree, and over 1100× more compressed (1.7 GB -> 1.5 MB)

The PGM-index is never worse in time and space than a B-tree
A stronger theoretical result on the space of a PGM

**Theorem.** Consider iid gaps between consecutive input keys with finite mean $\mu$ and variance $\sigma^2$.

If $\varepsilon$ is sufficiently large, the number of segments ($\approx$ the space of a PGM) on $n$ input keys is, with high probability,

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

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

Website: [https://pgm.di.unipi.it](https://pgm.di.unipi.it)

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

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

Documentation: [https://pgm.di.unipi.it/docs/](https://pgm.di.unipi.it/docs/)
#include <vector>
#include <iostream>
#include <algorithm>
#include "pgm_index.hpp"

int main() {
    // Generate some random data
    std::vector<uint32_t> data(1000000);
    std::generate(data.begin(), data.end(), std::rand);
    std::sort(data.begin(), data.end);

    // Construct the PGM-index
    const int eps = 128;
    const int eps_recursive = 8;
    pgm::PGMIndex<uint32_t, eps, eps_recursive> index(data);

    // Query the PGM-index
    auto q = 42;
    auto range = index.search(q);
    auto lo = data.begin() + range.lo;
    auto hi = data.begin() + range.hi;
    std::cout << *std::lower_bound(lo, hi, q) << std::endl;
    std::cout << index.size_in_bytes() << std::endl;

    return 0;
}
API of the piecewise linear model

• To construct the vector of segments
  
  ```cpp
  std::vector<Segment>
  make_pgm_segments(
      const std::vector<uint64_t> &data,
      size_t epsilon
  );
  ```

• To compute a prediction
  
  ```cpp
  size_t pos = segments[i](key)
  ```

  (this API is available only on the challenge website)
The challenge

*Beat the space-time Pareto curve of the C++ reference implementation of the PGM-index*

• Use all the tools (including PGM) you have seen in the lab lectures as Lego bricks to design your solution

• Propose new ideas and code them

• Required methods:
  • `MyIndex(const std::vector<uint64_t> &data)`
  • `size_t size_in_bytes()`
  • `uint64_t nextGEQ(uint64_t x)`  // assume: data.front() ≤ x < data.back()`
The challenge (cont.)

• Submit your `index.hpp` to `ae2020challenge.di.unipi.it` and check the real-time leaderboard

• Three datasets with possibly repeated keys (read-only plain vector)

• Time-space performance is important, but more important are originality and elegance of the solution