The design of learning-based compressed data structures

LADSIOS @ VLDB 2021
August 16, 2021

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

Joint work with Antonio Boffa, Paolo Ferragina, Fabrizio Lillo
Outline

• Two classical problems in data structure & DBMS design:
  • Data indexing
  • Data compression & access

• Reframe them as a problem of approximating the distribution of the input data

• Show solutions that learn the input data regularities and guarantee efficient space-time complexity bounds
Problem 1

Data indexing
The predecessor search problem

• Given \( n \) sorted input keys, implement:
  \[ \text{predecessor}(x) = \text{"largest key } \leq x\text{"} \]

• Range queries in DBs, conjunctive queries in search engines, IP routing...

• Traditionally solved by tree- or trie-based data structures

\[
\begin{align*}
\text{predecessor}(36) &= 36 \\
\text{predecessor}(50) &= 48
\end{align*}
\]
The idea: input data as pairs \((key, position)\)
The idea: input data as pairs \((key, position)\)
The idea: learning a mapping $keys \rightarrow positions$
Learned indexes

Often we need more complex models than linear ones.

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?

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

Ao et al. [VLDB 2011], Kraska et al. [SIGMOD 2018]
Introducing the PGM-index

- **Piecewise linear $\varepsilon$-approx.**
  Fast to construct and space-optimal (number of segments is minimised)

- **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

Ferragina and Vinciguerra [VLDB 2020]
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 $\mathcal{O}(n)$ time

**Step 2.** Store the segments as triples $s_i = (\text{key}, \text{slope}, \text{intercept})$
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 $[pos - \varepsilon, pos + \varepsilon]$
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 $s_i = (\text{key}, \text{slope}, \text{intercept})$

**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 $s_i = (\text{key}, \text{slope}, \text{intercept})$

**Step 3.** Keep only $s_i.\text{key}$

**Step 4.** Repeat recursively
Memory layout of the PGM-index

(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 a PGM-index

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

In ICML 20, we showed that the space is $\mathcal{O}(n/B^2)$ w.h.p.

The PGM-index is never worse in time and space than a B-tree.
Experiments
Static-scenario: experimental settings

• The majority of papers test learned indexes on positive lookups

• An index must answer correctly and efficiently also when the query keys are not in the input (training) data

• Here we test each index on 10M random predecessor queries

• In-memory data with 8-byte keys, 8-byte payload
Static-scenario: experiments

Linear RMI and PGM have the same size

PGM is often better than Linear RMI and CSS-tree

Linear RMI implementation and datasets from Marcus et al. [VLDB 2021]
Static-scenario: experiments with Hybrid RMI

(RMI with non-linear models, tuned via grid search)

Hybrid RMI is often better than PGM, but...

Tuned Hybrid RMI implementation and datasets from Marcus et al. [VLDB 2021]
Why worst-case bounds are important?

Adversarial query workload

Tuned Hybrid RMI implementation and datasets from Marcus et al. [VLDB 2021]
Dynamic-scenario: experimental settings

• Many papers show only the index/model size and disregard other design choices, e.g. half-empty nodes
  Here we show the overall data structure size

• We test PGM, B-tree, B+tree, Y-fast trie, ALEX, ART
  For the first four, we vary the page-size and show the fastest configuration

• Extract batch of 100M query+insert ops from dataset with 800M keys

• Use the remaining keys to bulk load each structure
  8-byte keys, 8-byte payload
Bulk loading $+700M$ records

For osm_cellids and books, the bulk load seconds are compared across different data structures. The Y-fast trie shows the fastest bulk load times, followed by ART, B-tree, and POM, with ALEX having the slowest times.
Dynamic-scenario: latency over 100M ops

- PGM is faster for write-heavy workloads (0% to 25%)
- PGM and ART are faster for balanced workloads (50%)
- ART is faster for read-heavy workloads (75%)
- PGM, ART and ALEX are faster for read-only workloads (100%)
Dynamic-scenario: overall memory usage

• PGM is the most memory-efficient (12.9 GB)
• B-tree is the second-best (16.5 GB)
• ART is the most memory-hungry (34.6 GB)
• Some learned indexes can be larger than traditional ones (ALEX is +15% than B-tree, +47% than PGM)
Dynamic-scenario: range queries

• Small range sizes (10 results)
  • All data structures take from 2 to 4 μs
  • ALEX is the fastest

• Medium range sizes (1K results)
  • Y-fast trie is the fastest, 7 μs
  • PGM is the second-fastest, +51%
  • ALEX is the third-fastest, +7x wrt Y-fast trie

• Large range sizes (100K results)
  • Y-fast trie is still the fastest, 605 μs
  • PGM is still the second-fastest, +1% wrt Y-fast trie
  • ALEX is still the third-fastest, +7x wrt Y-fast trie
More structures in the PGM library

- Variants of the PGM
  - CompressedPGM
  - EliasFanoPGM
  - BucketingPGM

- MultidimensionalPGM
  - Orthogonal range queries
  - k-NN queries (thanks DBlab @ Nagoya Univ.)
Problem 2

Data compression & access
The idea
The idea: data = segments

Represent integers with an information loss of $\varepsilon$
The idea: data = segments + corrections

Represent integers with an information loss of ε

Complement the approximations to recover the original set

LA-vector: a compressed array supporting efficient random access and other query operations

\( x_i = f(i) + C[i] \)
Experiments on LA-vector, in brief

- Tested on DNA, 5Gram, URLs, and inverted lists
- Fast random access and competitive queries wrt well-engineered compressed data structures
- Space occupancy close to the most compressed (but less query-efficient) approaches

Boffa, Ferragina, Vinciguerra: A “learned” approach to quicken and compress rank/select dictionaries.

Code available at github.com/gvinciguerra/la_vector
Conclusions
Wrap up

• Introduced data structures that learn the input data regularities, without giving up worst-case bounds:
  • The PGM-index for data indexing
  • The LA-vector for compressing and indexing data

• Practical performance on par with or orders of magnitude better than traditional data structures

• Libraries are open-source, we invite users and contributors
  We are extending the PGM to big integers (up to 256 bytes)
Open questions

1. Can we further improve the performance of Dynamic PGM over read-heavy workloads?

2. Can we learn $\epsilon$-approximate nonlinear models efficiently?

3. Do these models improve the $O(n/\epsilon^2)$ space bound of the piecewise-linear model adopted by PGM?
References


