Theory and practice of learning-based compressed data structures

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

March 19, 2021
Links seminar @ Université de Lille and Inria
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 $\text{predecessor}(x) = \text{"largest key } \leq x\text{"}$

• 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!)

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

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

\[ \text{position} = 11 \]

\[ \text{key} = 36 \]

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

Ao et al. [VLDB 2011]
Input data as pairs \((key, position)\)
Learning the mapping keys $\mapsto$ positions

![Graph showing the mapping of keys to positions with error markers.]
Learned indexes

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

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?

Binary search in \([\text{position} - \text{error}, \text{position} + \text{error}]\)
The problem with learned indexes

- Fast query time and excellent space usage in practice, but no worst-case guarantees
- Too much I/O when data is on disk
- Unpredictable latency
- 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 $\varepsilon$-approx.
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
PGM-index construction

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

$$s_i = (\text{key}, \text{slope}, \text{intercept})$$
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 $2\varepsilon + 1$
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 $\epsilon$-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

![Graph showing key vs. position with data points at 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

---

![Graph showing key vs. position with data points: 2, 23, 31, 48, 71, 88, 122, 145.](attachment:image.png)
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

(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 $\varepsilon = 1$

$B =$ disk page-size

Set $\varepsilon = \Theta(B)$ for queries in $O(\log_B n)$ I/Os

$O(n/\varepsilon)$ space

$\varepsilon = 1$

predecessor(57)?

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

Very pessimistic statement
We will see a stronger bound later
Experiments
Experiments

Experiments

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

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)

Avg search range

Page size

Query time (ns)

Index size (MB)

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

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>

Frequency of queries wrt inserts and deletes

Time (ns)

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
Website and libraries

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)

New features

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

Theoretical grounds of learning-based data structures
The knowledge gap

Practice
Same query time of traditional tree-based indexes

Space improvements of orders of magnitude, from GBs to few MBs

Theory
Same asymptotic query time of traditional tree-based indexes

Same asymptotic space occupancy of traditional tree-based indexes

👎 vs 👍 vs 👍
What is the space of learned indexes?

• Space occupancy $\propto$ 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 $\varepsilon$, 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 $\mu$ and variance $\sigma^2$
The result

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

If $\epsilon$ is sufficiently large, the number of segments ($\approx$ the space of a PGM) on $n$ input keys converges to

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

**Corollary.** Under the assumption above, the PGM-index with $\epsilon = \Theta(B)$ 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 $\varepsilon$

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 $\mu$

4. Compute the expectation of
   \[ i^* = \min\{i \in \mathbb{N} \mid (k_t, i) \text{ is outside the red strip} \} \]
   i.e. the Mean Exit Time (MET) of the random walk

5. Show that the slope $m = 1/\mu$ maximises $E[i^*]$, giving $E[i^*] = (\mu^2/\sigma^2) \varepsilon^2$

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 $\text{rank}(x)$: number of elements in $S$ which are $\leq x$
  3. Implement $\text{select}(i)$: return the $i$th smallest element in $S$

• Building block of succinct data structures for texts, genomes, graphs, etc. Very mature field

$\text{rank}(12) = 3$

$\text{select}(7) = 40$
The idea
The idea: data = segments

Represent integers with an information loss of $\varepsilon$

$$f_1(x) = 5x - 5$$

$$f_2(x) = 6x - 5$$

<table>
<thead>
<tr>
<th>position</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td>53</td>
</tr>
</tbody>
</table>
The idea: data = segments + corrections

Represent integers with an information loss of $\varepsilon$

Complement the approximations to recover the original set

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

\[
\begin{align*}
\text{Value} & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\
\text{Position} & \quad 3 \quad 1 \quad 0 \quad 0 \quad -2 \quad -3 \quad 3 \quad 0 \quad -2 \quad -2 \\
\end{align*}
\]

\( f_1(x) = 5x - 5 \)

\( f_2(x) = 6x - 5 \)
How to minimise the space?

• Space $O(\ell) + nc$ bits. How to choose $c$ without increasing $\ell$?
• 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

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 sophisticated 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 $2^0, 2^1, \ldots, 2^b$
- An insert merges the first $j - 1$ full levels into the first free level $j$
Sketch of the fully-dynamic PGM

- Define $b = \Theta(\log n)$ 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 $j$

```plaintext
\{ \} \cup \{ \text{merged levels} \} = \text{Merge in } O(2^j) \text{ time}
```
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 $j$