Next Generation Indexes For Big Data Engineering

Daniel Lemire and collaborators
blog: https://lemire.me
twitter: @lemire

Université du Québec (TÉLUQ)
Montreal 🇨🇦
Knuth on performance

Premature optimization is the root of all evil
Knuth on performance

Premature optimization is the root of all evil (...) After a programmer knows which parts of his routines are really important, a transformation like doubling up of loops will be worthwhile.
### Constants matter

fasta benchmark:

<table>
<thead>
<tr>
<th></th>
<th>elapsed time</th>
<th>total time (all processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-threaded</td>
<td>1.36 s</td>
<td>1.36 s</td>
</tr>
</tbody>
</table>

[https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fasta.html](https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/fasta.html)
Constants matter

fasta benchmark:

<table>
<thead>
<tr>
<th></th>
<th>elapsed time</th>
<th>total time (all processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-threaded</td>
<td>1.36 s</td>
<td>1.36 s</td>
</tr>
<tr>
<td>multicore (4 cores)</td>
<td>1.00 s</td>
<td>2.00 s</td>
</tr>
</tbody>
</table>

Daniel Lemire, Waterloo University, May 10th 2018.
### Constants matter

**fasta benchmark:**

<table>
<thead>
<tr>
<th></th>
<th>elapsed time</th>
<th>total time (all processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-threaded</td>
<td>1.36 s</td>
<td>1.36 s</td>
</tr>
<tr>
<td>multicore (4 cores)</td>
<td>1.00 s</td>
<td>2.00 s</td>
</tr>
<tr>
<td>vectorized (1 core)</td>
<td>0.31 s</td>
<td>0.31 s</td>
</tr>
</tbody>
</table>

[https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/](https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/)
“One Size Fits All”: An Idea Whose Time Has Come and Gone (Stonebraker, 2005)
Rediscover Unix

In 2018, Big Data Engineering is made of several specialized and re-usable components:

- Calcite : SQL + optimization
- Hadoop
- etc.
"Make your own database engine from parts"

We are in a Cambrian explosion, with thousands of organizations and companies building their custom high-speed systems.

- Specialized used cases
- Heterogeneous data (not everything is in your Oracle DB)
For high-speed in data engineering you need...

- Front-end (data frame, SQL, visualisation)
- High-level optimizations
- Indexes (e.g., Pilosa, Elasticsearch)
  - Great compression routines
  - Specialized data structures
- ....
Sets

A fundamental concept (sets of documents, identifiers, tuples...)

→ For performance, we often work with sets of integers (identifiers).
• tests: $x \in S$?
• intersections: $S_2 \cap S_1$, unions: $S_2 \cup S_1$, differences: $S_2 \setminus S_1$
• Similarity (Jaccard/Tanimoto): $|S_1 \cap S_1| / |S_1 \cup S_2|
• Iteration

```python
for x in S do
    print(x)
```
How to implement sets?

- sorted arrays (`std::vector<uint32_t>`)  
- hash tables (`java.util.HashSet<Integer>`, `std::unordered_set<uint32_t>`)  
- ...  
- bitmap (`java.util(BitSet)`)  
- ❤️ ❤️ ❤️ compressed bitmaps ❤️ ❤️ ❤️
Arrays are your friends

```java
while (low <= high) {
    int mI =
        (low + high) >>> 1;
    int m = array.get(mI);
    if (m < key) {
        low = mI + 1;
    } else if (m > key) {
        high = mI - 1;
    } else {
        return mI;
    }
}
return -(low + 1);
```
Hash tables

- value $x$ at index $h(x)$
- random access to a value in expected constant-time
  - much faster than arrays
in-order access is kind of terrible

- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)
- \([15, 3, 0, 6, 11, 4, 5, 9, 12, 13, 8, 2, 1, 14, 10, 7]\)

(Robin Hood, linear probing, MurmurHash3 hash function)
Set operations on hash tables

\[
\begin{align*}
h_1 & \leftarrow \text{hash set} \\
h_2 & \leftarrow \text{hash set} \\
\ldots & \\
& \text{for}(x \text{ in } h_1) \{ \\
& \quad \text{insert } x \text{ in } h_2 // \text{cache miss?} \\
& \}
\end{align*}
\]
"Crash" Swift

```swift
var S1 = Set<Int>(1...size)
var S2 = Set<Int>()
for i in d {
    S2.insert(i)
}
```
Some numbers: half an hour for 64M keys

<table>
<thead>
<tr>
<th>size</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>0.8</td>
</tr>
<tr>
<td>8M</td>
<td>22</td>
</tr>
<tr>
<td>64M</td>
<td>1400</td>
</tr>
</tbody>
</table>

- Maps and sets can have quadratic-time performance
  https://lemire.me/blog/2017/01/30/maps-and-sets-can-have-quadratic-time-performance/
- Rust hash iteration+reinsertion
  https://accidentallyquadratic.tumblr.com/post/153545455987/rust-hash-iteration-reinsertion
**Bitmaps**

Efficient way to represent sets of integers.

For example, 0, 1, 3, 4 becomes `0b11011` or "27".

- `{0} → 0b00001`
- `{0, 3} → 0b01001`
- `{0, 3, 4} → 0b11001`
- `{0, 1, 3, 4} → 0b11011`
Manipulate a bitmap

64-bit processor.

Given \( x \), word index is \( x / 64 \) and bit index \( x \mod 64 \).

```java
add(x) {
    array[x / 64] |= (1 << (x % 64))
}
```
How fast is it?

\[
\begin{align*}
\text{index} &= x / 64 \quad \rightarrow \text{a shift} \\
\text{mask} &= 1 \ll (x \% 64) \quad \rightarrow \text{a shift} \\
\text{array}[\text{index}] \mid (- \text{mask}) \quad \rightarrow \text{a OR with memory}
\end{align*}
\]

One bit every $\approx 1.65$ cycles because of superscalarity
Bit parallelism

Intersection between \{0, 1, 3\} and \{1, 3\}

a single AND operation

between \texttt{0b1011} and \texttt{0b1010}.

Result is \texttt{0b1010} or \{1, 3\}.

No branching!
Bitmaps love ❤️ wide registers

- SIMD: Single Instruction Multiple Data
  - SSE (Pentium 4), ARM NEON 128 bits
  - AVX/AVX2 (256 bits)
  - AVX-512 (512 bits)

AVX-512 is now available (e.g., from Dell!) with Skylake-X processors.
Bitsets can take too much memory

\{1, 32000, 64000\} : 1000 bytes for three values

We use compression!
Git (GitHub) utilise EWAH

Run-length encoding

Example: 000000001111111100 est
00000000 − 11111111 − 00

Code long runs of 0s or 1s efficiently.

https://github.com/git/git/blob/master/ewah/bitmap.c
Complexity

- Intersection: $O(|S_1| + |S_2|)$ or $O(\min(|S_1|, |S_2|))$
- In-place union ($S_2 \leftarrow S_1 \cup S_2$): $O(|S_1| + |S_2|)$ or $O(|S_2|)$
Roaring Bitmaps

http://roaringbitmap.org/


- Java, C, Go (interoperable)
Hybrid model

Set of containers

- sorted arrays \{1,20,144\}
- bitset \(0b10000101011\)
- runs \([0,10],[15,20]\)

Related to: O'Neil's RIDBit + BitMagic
Figure 1. Roaring bitmap containing the first 1000 multiples of 62, all integers in the intervals $[2^{16}, 2^{16} + 100)$, $[2^{16} + 101, 2^{16} + 201)$, $[2^{16} + 300, 2^{16} + 400)$ and all even integers in $[2 \times 2^{16}, 3 \times 2^{16})$.
Roaring

- All containers are small (8 kB), fit in CPU cache
- We predict the output container type during computations
- E.g., when array gets too large, we switch to a bitset
- Union of two large arrays is materialized as a bitset...
- Dozens of heuristics... sorting networks and so on
Use Roaring for bitmap compression whenever possible. Do not use other bitmap compression methods (Wang et al., SIGMOD 2017)
Unions of 200 bitmaps

bits per stored value

<table>
<thead>
<tr>
<th></th>
<th>bitset</th>
<th>array</th>
<th>hash table</th>
<th>Roaring</th>
</tr>
</thead>
<tbody>
<tr>
<td>census1881</td>
<td>524</td>
<td>32</td>
<td>195</td>
<td>15.1</td>
</tr>
<tr>
<td>weather</td>
<td>15.3</td>
<td>32</td>
<td>195</td>
<td>5.38</td>
</tr>
</tbody>
</table>

cycles per input value:

<table>
<thead>
<tr>
<th></th>
<th>bitset</th>
<th>array</th>
<th>hash table</th>
<th>Roaring</th>
</tr>
</thead>
<tbody>
<tr>
<td>census1881</td>
<td>9.85</td>
<td>542</td>
<td>1010</td>
<td>2.6</td>
</tr>
<tr>
<td>weather</td>
<td>0.35</td>
<td>94</td>
<td>237</td>
<td>0.16</td>
</tr>
</tbody>
</table>
Sometimes you do want arrays!!!

- But you'd like to compress them up.
- *Not always:* compression can be counterproductive.
- Still, if you must compress, you want to do it *fast*
Integer compression

- "Standard" technique: VByte, VarInt, VInt
- Use 1, 2, 3, 4, ... byte per integer
- Use one bit per byte to indicate the length of the integers in bytes
- Lucene, Protocol Buffers, etc.
varint-GB from Google

- VByte: one branch per integer
- varint-GB: one branch per 4 integers
- each 4-integer block is preceded by a control byte
Vectorisation

- Stepanov (STL in C++) working for Amazon proposed varint-G8IU
- Use vectorization (SIMD)
  - *Patented*
  - Fastest byte-oriented compression technique (until recently)

SIMD-Based Decoding of Posting Lists, CIKM 2011
https://stepanovpapers.com/SIMD_Decoding_TR.pdf
Observations from Stepanov et al.

- We can vectorize Google's varint-GB, but it is not as fast as varint-G8IU
Stream VByte

- Reuse varint-GB from Google
- But instead of mixing control bytes and data bytes, ...
- We store control bytes separately and consecutively...

Daniel Lemire, Nathan Kurz, Christoph Rupp
Stream VByte: Faster Byte-Oriented Integer Compression
Information Processing Letters 130, 2018
Integer compression
Stream VByte is used by...

- Redis (within RediSearch) https://redislabs.com
- upscaledb https://upscaledb.com
- Trinity https://github.com/phaistos-networks/Trinity
Dictionary coding

Use, e.g., by Apache Arrow

Given a list of values:


Map to integers

- 0, 1, 2, 0, 2

Compress integers:

- Given $2^n$ distinct values...
- Can use $n$-bit per values (binary packing, patched coding, frame-of-reference)
## Dictionary coding + SIMD

<table>
<thead>
<tr>
<th>dict. size</th>
<th>bits per value</th>
<th>scalar</th>
<th>AVX2 (256-bit)</th>
<th>AVX-512 (512-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
<td>8</td>
<td>3.5</td>
<td>2</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
<td>12</td>
<td>5.5</td>
<td>4.5</td>
</tr>
</tbody>
</table>

(cycles per value decoded)

https://github.com/lemire/dictionary
To learn more...

- Blog (twice a week): [https://lemire.me/blog/](https://lemire.me/blog/)
- GitHub: [https://github.com/lemire](https://github.com/lemire)
- Home page: [https://lemire.me/en/](https://lemire.me/en/)
- CRSNG: *Faster Compressed Indexes On Next-Generation Hardware* (2017-2022)
- Twitter: @lemire