ENGINEERING FAST INDEXES

Daniel Lemire 🍁

https://lemire.me

Joint work with lots of super smart people

Our recent work: Roaring Bitmaps

http://roaringbitmap.org/

Used by

- Apache Spark,
- Netflix Atlas,
- LinkedIn Pinot,
- Apache Lucene,
- Whoosh,
- Metamarket's Druid
- eBay's Apache Kylin

Further reading:

- Frame of Reference and Roaring Bitmaps (at Elastic, the company behind Elasticsearch)

Set data structures

We focus on sets of integers: $S = \{1, 2, 3, 1000\}$. Ubiquitous in database or search engines.

- tests: $x \in S$?
- intersections: $S_2 \cap S_1$
- unions: $S_2 \cup S_1$
- differences: $S_2 \setminus S_1$
- Jaccard Index (Tanimoto similarity) $|S_1 \cap S_1|/|S_1 \cup S_2|$

"Ordered" Set

- iterate
  - in sorted order,
  - in reverse order,
  - skippable iterators (jump to first value $\geq x$)
- Rank: how many elements of the set are smaller than k?
- Select: find the kth smallest value
- Min/max: find the maximal and minimal value

Let us make some assumptions...

- Many sets containing more than a few integers
- Integers span a wide range (e.g., $[0, 100000)$)
- Mostly immutable (read often, write rarely)

How do we implement integer sets?

Assume sets are *mostly* imutable.

- sorted arrays (`std::vector<uint32_t>`)
- hash sets (`java.util.HashSet<Integer>` , `std::unordered_set<uint32_t>`)
- `...`
- bitsets (`java.util.BitSet`)
- ❤ ❤ ❤ compressed bitsets ❤ ❤ ❤

What is a bitset???

Efficient way to represent a set of integers.

E.g., 0, 1, 3, 4 becomes `0b11011` or "27".

Also called a "bitmap" or a "bit array".

Add and contains on bitset

Most of the processors work on 64-bit words.

Given index `x`, the corresponding word index is `x/64` and within-word bit index is `x % 64`.

```
add(x) {
  array[x / 64] |= (1 << (x % 64))
}

contains(x) {
  return array[x / 64] & (1 << (x % 64))
}
```

How fast can you set bits in a bitset?

Very fast! Roughly three instructions (on x64)...

```
index = x / 64          -> a single shift
mask = 1 << ( x % 64)   -> a single shift
array[ index ] |- mask -> a logical OR to memory
```

(Or can use BMI's `bts`.)

On recent x64 can set one bit every ≈ 1.65 cycles (in cache)

Recall : Modern processors are superscalar (more than one instruction per cycle)

Bit-level parallelism

Bitsets are efficient: intersections

Intersection between {0, 1, 3} and {1, 3}
can be computed as AND operation between
`0b1011` and `0b1010`.

Result is `0b1010` or {1, 3}.

Enables *Branchless* processing.

Bitsets are efficient: in practice

```
for i in [0...n]
  out[i] = A[i] & B[i]
```

Recent x64 processors can do this at a speed of ≈ 0.5 cycles per pair of input 64-bit words (in cache) for `n = 1024`.

0.5

`memcpy` runs at ≈ 0.3 cycles.

0.3

Bitsets can be inefficient

Relatively wasteful to represent {1, 32000, 64000} with a bitset.
Would use 1000 bytes to store 3 numbers.

So we use compression…

Memory usage example

dataset : census1881_srt

format	bits per value
hash sets	200
arrays	32
bitsets	900
compressed bitsets (Roaring)	2

https://github.com/RoaringBitmap/CBitmapCompetition

Performance example (unions)

dataset : census1881_srt

format	CPU cycles per value
hash sets	200
arrays	6
bitsets	30
compressed bitsets (Roaring)	1

https://github.com/RoaringBitmap/CBitmapCompetition

What is happening? (Bitsets)

Bitsets are often best… except if data is very sparse (lots of 0s). Then you spend a lot of time scanning zeros.

- Large memory usage
- Bad performance

Threshold? ~1:100

Hash sets are not always fast

Hash sets have great one-value look-up. But
they have poor data locality and non-trivial overhead...

```
h1 <- some hash set
h2 <- some hash set
...
for(x in h1) {
   insert x in h2 // "sure" to hit a new cache line!!!!
}
```

Want to kill Swift?

Swift is Apple's new language. Try this:

```
var d = Set<Int>()
for i in 1...size {
  d.insert(i)
}
//
var z = Set<Int>()
for i in d {
    z.insert(i)
}
```

This blows up! Quadratic-time.

Same problem with Rust.

What is happening? (Arrays)

Arrays are your friends. Reliable. Simple. Economical.

But... binary search is *branchy* and has *bad locality*...

```
while (low <= high) {
  int middleIndex = (low + high) >>> 1;
  int middleValue = array.get(middleIndex);

  if (middleValue < ikey) {
    low = middleIndex + 1;
  } else if (middleValue > ikey) {
    high = middleIndex - 1;
  } else {
    return middleIndex;
  }
}
return -(low + 1);
```

Performance: value lookups ($x \in S$)

dataset : weather_sept_85

format	CPU cycles per query
hash sets (`std::unordered_set`)	50
arrays	900
bitsets	4
compressed bitsets (Roaring)	80

How do you compress bitsets?

- We have long runs of 0s or 1s.
- Use run-length encoding (RLE)

Example: 000000001111111100 can be coded as

$$00000000 - 11111111 - 00$$

or

<5><1>

using the format < number of repetitions >< value being repeated >

RLE-compressed bitsets

- Oracle's BBC
- WAH (FastBit)
- EWAH (Git + Apache Hive)
- Concise (Druid)
- . . .

Further reading:

http://githubengineering.com/counting-objects/

Hybrid Model

Decompose 32-bit space into
16-bit spaces (chunk).

Given value x, its chunk index is $x \div 2^{16}$ (16 most significant bits).

For each chunk, use best container to store least 16 significant bits:

- a sorted array ({1,20,144})
- a bitset (0b10000101011)
- a sequences of sorted runs ([0,10],[15,20])

That's Roaring!

Prior work: O'Neil's RIDBit + BitMagic

Roaring

- All containers fit in 8 kB (several fit in L1 cache)
- Attempts to select the best container as you build the bitmaps
- Calling `runOptimize` will scan (quickly!) non-run containers and try to convert them to run containers

Performance: union (weather_sept_85)

format	CPU cycles per value
bitsets	0.6
WAH	4
EWAH	2
Concise	5
Roaring	0.6

SPARK SUMMIT EAST 2017

What helps us...

- All modern processors have fast population-count functions (`popcnt`) to count the number of 1s in a word.
- Cheap to keep track of the number of values stored in a bitset!
- Choice between array, run and bitset covers many use cases!

Go try it out!

- Java, Go, C, C++, C#, Rust, Python... (soon: Swift)
- http://roaringbitmap.org
- Documented interoperable serialized format.
- Free. Well-tested. Benchmarked.
- Peer reviewed
  - Consistently faster and smaller compressed bitmaps with Roaring. Softw., Pract. Exper. (2016)
  - Better bitmap performance with Roaring bitmaps. Softw., Pract. Exper. (2016)
  - Optimizing Druid with Roaring bitmaps, IDEAS 2016, 2016
- Wide community (dozens of contributors).