What are some alternatives to a bit array?

Asked 16 years, 3 months ago Modified 10 years, 2 months ago

Viewed 3k times

8

I have an information retrieval application that creates bit arrays on the order of 10s of million bits. The number of "set" bits in the array varies widely, from all clear to all set. Currently, I'm using a straight-forward bit array (`java.util.BitSet`), so each of my bit arrays takes several megabytes.

My plan is to look at the cardinality of the first *N* bits, then make a decision about what data structure to use for the remainder. Clearly some data structures are better for very sparse bit arrays, and others when roughly half the bits are set (when most bits are set, I can use negation to treat it as a sparse set of zeroes).

- What structures might be good at each extreme?
- Are there any in the middle?

Here are a few constraints or hints:

1. The bits are set only once, and in index order.
2. I need 100% accuracy, so something like a Bloom filter isn't good enough.

3. After the set is built, I need to be able to efficiently iterate over the "set" bits.
4. The bits are randomly distributed, so run-length–encoding algorithms aren't likely to be much better than a simple list of bit indexes.
5. I'm trying to optimize memory utilization, but speed still carries *some* weight.

Something with an open source Java implementation is helpful, but not strictly necessary. I'm more interested in the fundamentals.

data-structures **information-retrieval**

Share

Improve this question

Follow

edited Aug 30, 2008 at 18:35

asked Aug 30, 2008 at 16:39

erickson

269k ● 59 ● 401 ● 497

7 Answers

Sorted by: Highest score (default)

16

Unless the data is truly random ***and*** has a symmetric 1/0 distribution, then this simply becomes a lossless data compression problem and is very analogous to CCITT Group 3 compression used for black and white (i.e.:

Binary) FAX images. CCITT Group 3 uses a Huffman Coding scheme. In the case of FAX they are using a fixed set of Huffman codes, but for a given data set, you can generate a specific set of codes for each data set to improve the compression ratio achieved. As long as you only need to access the bits sequentially, as you implied, this will be a pretty efficient approach. Random access would create some additional challenges, but you could probably generate a binary search tree index to various offset points in the array that would allow you to get close to the desired location and then walk in from there.

Note: The Huffman scheme still works well even if the data is random, as long as the 1/0 distribution is not perfectly even. That is, the less even the distribution, the better the compression ratio.

Finally, if the bits are truly random with an even distribution, then, well, according to *Mr. Claude Shannon*, you are not going to be able to compress it any significant amount using any scheme.

Share Improve this answer

Follow

edited Oct 2, 2014 at 23:06

Jason Sundram

12.5k ● 20 ● 72 ● 86

answered Aug 30, 2008 at 20:05

Tall Jeff

9,984 ● 7 ● 46 ● 61

Beautiful solution. It might even be fast as well since memory loads are so costly today. – I GIVE CRAP ANSWERS Oct 5,

2008 at 15:16

4

I would strongly consider using range encoding in place of Huffman coding. In general, range encoding can exploit asymmetry more effectively than Huffman coding, but this is especially so when the alphabet size is so small. In fact, when the "native alphabet" is simply 0s and 1s, the only way Huffman can get any compression at all is by combining those symbols -- which is exactly what range encoding will do, more effectively.

Share Improve this answer

Follow

answered Sep 18, 2008 at 1:06

Antaeus Feldspar

Thanks Antaeus, I had actually looked into range coding already, as the accepted answer cited Huffman coding as just one example of lossless compression. However, Huffman is easy to implement and works well on moderately asymmetric input. For highly asymmetric input, run-length methods are good. – erickson Sep 18, 2008 at 5:02

2

Maybe too late for you, but there is a very fast and memory efficient library for sparse bit arrays (lossless) and other data types based on tries. Look at Judy arrays

Share Improve this answer

Follow

answered Jun 17, 2009 at 17:16

bill

1,361 ● 11 ● 9

Thanks Bill. I remember hearing about Judy arrays before but I had completely forgotten about them. I will take another look at them. – erickson Jun 17, 2009 at 18:03

1

Thanks for the answers. This is what I'm going to try for dynamically choosing the right method:

I'll collect all of the first *N* hits in a conventional bit array, and choose one of three methods, based on the symmetry of this sample.

- If the sample is highly asymmetric, I'll simply store the indexes to the set bits (or maybe the distance to the next bit) in a list.
- If the sample is highly symmetric, I'll keep using a conventional bit array.
- If the sample is moderately symmetric, I'll use a lossless compression method like Huffman coding suggested by InSciTekJeff.

The boundaries between the asymmetric, moderate, and symmetric regions will depend on the time required by the various algorithms balanced against the space they need, where the relative value of time versus space would be an adjustable parameter. The space needed for Huffman coding is a function of the symmetry, and I'll profile that with testing. Also, I'll test all three methods to determine the time requirements of my implementation.

It's possible (and actually I'm hoping) that the middle compression method will always be better than the list or the bit array or both. Maybe I can encourage this by choosing a set of Huffman codes adapted for higher or lower symmetry. Then I can simplify the system and just use two methods.

Share Improve this answer

Follow

edited May 23, 2017 at 12:19

Community Bot

1 • 1

answered Aug 31, 2008 at 16:23

erickson

269k • 59 • 401 • 497

1

One more compression thought:

If the bit array is not crazy long, you could try applying the Burrows-Wheeler transform before using any repetition encoding, such as Huffman. A naive implementation would take O(n^2) memory during (de)compression and O(n^2 log n) time to decompress - there are almost certainly shortcuts to be had, as well. But if there's any sequential structure to your data at all, this should really help the Huffman encoding out.

You could also apply that idea to one block at a time to keep the time/memory usage more practical. Using one block at time could allow you to always keep most of the data structure compressed if you're reading/writing sequentially.

Share Improve this answer

Follow

answered Aug 31, 2008 at 21:23

Tyler

28.9k ● 12 ● 93 ● 108

0

Straight forward lossless compression is the way to go. To make it searchable you will have to compress relatively small blocks and create an index into an array of the blocks. This index can contain the bit offset of the starting bit in each block.

Share Improve this answer

Follow

answered Aug 31, 2008 at 9:58

Tim Ring

1,833 ● 1 ● 20 ● 27

0

Quick combinatoric proof that you can't really save much space:

Suppose you have an arbitrary subset of n/2 bits set to 1 out of n total bits. You have (n choose n/2) possibilities. Using Stirling's formula, this is roughly 2^n / sqrt(n) * sqrt(2/pi). If every possibility is equally likely, then there's no way to give more likely choices shorter representations. So we need log_2 (n choose n/2) bits, which is about n - (1/2)log(n) bits.

That's not a very good savings of memory. For example, if you're working with n=2^20 (1 meg), then you can only save about 10 bits. It's just not worth it.

Having said all that, it also seems very unlikely that any really useful data is truly random. In case there's any more structure to your data, there's probably a more optimistic answer.

Share Improve this answer

Follow

answered Aug 31, 2008 at 11:16

Tyler

28.9k ● 12 ● 93 ● 108