

Compression formats with good support for random access within archives? [closed]

Asked 15 years, 11 months ago Modified 4 years ago Viewed 30k times



74



Closed. This question is seeking recommendations for software libraries, tutorials, tools, books, or other off-site resources. It does not meet [Stack Overflow guidelines](#). It is not currently accepting answers.

💡 We don't allow questions seeking recommendations for software libraries, tutorials, tools, books, or other off-site resources. You can edit the question so it can be answered with facts and citations.

Closed 2 years ago.

[Improve this question](#)

This is similar to a [previous question](#), but the answers there don't satisfy my needs and my question is slightly different:

I currently use gzip compression for some very large files which contain sorted data. When the files are not

compressed, binary search is a handy and efficient way to support seeking to a location in the sorted data.

But when the files are compressed, things get tricky. I recently found out about [zlib](#)'s `Z_FULL_FLUSH` option, which can be used during compression to insert "sync points" in the compressed output (`inflateSync()` can then begin reading from various points in the file). This is OK, though files I already have would have to be recompressed to add this feature (and strangely `gzip` doesn't have an option for this, but I'm willing to write my own compression program if I must).

It seems from [one source](#) that even `Z_FULL_FLUSH` is not a perfect solution...not only is it not supported by all gzip archives, but the very idea of detecting sync points in archives may produce false positives (either by coincidence with the magic number for sync points, or due to the fact that `Z_SYNC_FLUSH` also produces sync points but they are not usable for random access).

Is there a better solution? I'd like to avoid having auxiliary files for indexing if possible, and explicit, default support for quasi-random access would be helpful (even if it's large-grained--like being able to start reading at each 10 MB interval). Is there another compression format with better support for random reads than gzip?

Edit: As I mentioned, I wish to do binary search in the compressed data. I don't need to seek to a specific (uncompressed) position--only to seek with some coarse granularity within the compressed file. I just want support

for something like "Decompress the data starting roughly 50% (25%, 12.5%, etc.) of the way into this compressed file."

compression

gzip

archive

zlib

random-access

Share

Improve this question

Follow

edited Feb 16, 2020 at 19:13



Vaelus

1,065 ● 12 ● 28

asked Jan 9, 2009 at 22:29



John Zwinck

248k ● 43 ● 337 ● 451

10 Answers

Sorted by:

Highest score (default)



Take a look at **dictzip**. It is compatible with gzip and allows coarse random access.

37

An excerpt from its man page:



dictzip compresses files using the **gzip(1)** algorithm (LZ77) in a manner which is completely compatible with the gzip file format. An extension to the gzip file format (Extra Field, described in 2.3.1.1 of RFC 1952) allows extra data to be stored in the header of a compressed file. Programs like gzip and zcat will ignore this extra data. However, [dictzcat --start] will make use of

this data to perform pseudo-random access on the file.

I have the package dictzip in Ubuntu. Or its source code is in a [dictd-*.tar.gz](#). Its license is GPL. You are free to study it.

Update:

I improved dictzip to have no file size limit. [My implementation](#) is under MIT license.

Share Improve this answer

edited Mar 21, 2015 at 22:11

Follow

answered Oct 24, 2010 at 19:48



Ivo Danihelka

3,412 ● 3 ● 32 ● 27

2 I solved my problem through the use of gzip sync/flush points, which allow me to scan through the file (doing binary search) just fine. I had to write my own gzip-like program on top of libz, because the standard gzip for whatever reason doesn't include a facility to write sync points. Anyway, this works great in my case, because I do not care about being able to "read starting at byte 10000", only to "read starting about 50% of the way through the file." The dictzip approach does look very interesting, and solves a perhaps more general problem than mine. – [John Zwinck](#) Nov 6, 2010 at 16:52

1 @TroyJ: if you control the writing of the files, false positives are not going to happen often, and when they do you may

know it because decompression from those points will fail (and you can try again). If you do not control the writing, things are trickier: standard gzip-writing programs will emit lots of false positives and no true positives. You could retry N times before giving up; in my experience N will only need to be a small number (less than 10) for the system to be reasonably accurate. – [John Zwinck](#) Sep 5, 2014 at 15:00

2 I wrote stdio-like library and multithreaded compression utility. Sources are available on github: github.com/hoxnox/csio – [hoxnox](#) Jun 18, 2015 at 4:34

1 @AdamKatz: I cannot share the code, partly because it's tightly integrated with a proprietary data format, so nobody would have use for it directly. However, the idea is to write "full sync points" every so often when compressing (say once per MB), then make your reader scan for these points and verify that the messages make sense when you decompress. The difficulties are mostly (1) the standard gzip tool doesn't have an option to insert full sync points at all, (2) you need to write your own heuristic to verify valid messages when resuming. – [John Zwinck](#) Dec 3, 2015 at 4:03

1 @AdamKatz - gzipped data created by csio or dictzip – [hoxnox](#) Dec 3, 2015 at 16:36



19



I don't know of any compressed file format which would support random access to a specific location in the uncompressed data (well, except for multimedia formats), but you can brew your own.

For example, bzip2 compressed files are composed of independent compressed blocks of size <1MB uncompressed, which are delimited by sequences of magic bytes, so you could parse the bzip2 file, get the



block boundaries and then just uncompress the right block. This would need some indexing to remember where do the blocks start.

Still, I think the best solution would be to split your file into chunks of your choice, and then compressing it with some archiver, like zip or rar, which support random access to individual files in the archive.

Share Improve this answer

edited Jan 9, 2009 at 23:33

Follow

answered Jan 9, 2009 at 23:19



[jpalecek](#)

47.7k ● 7 ● 103 ● 146

I don't need to seek to a specific uncompressed position-- only to seek somewhat randomly with some coarse granularity within the compressed file. I don't mind at all if all I can do is say "uncompress the data starting here, about 700MB into this file." – [John Zwinck](#) Jan 9, 2009 at 23:53

@John Zwinck: Add your comment to your question as an update. Note that given the variable compression of data (some stuff I compress shrinks by 94% or so - usually, except when it only shrinks by about 50% or so), your estimate of where to start decompressing might be very hit and miss. – [Jonathan Leffler](#) Jan 10, 2009 at 6:54

Just a note that is complicated by bzip2 block boundaries being within a byte, so it is doable, but there is more bookkeeping required. – [Alex Reynolds](#) Jan 29, 2015 at 21:23



The [.xz file format](#) (which uses LZMA compression) seems to support this:

11



Random-access reading: The data can be split into independently compressed blocks. Every .xz file contains an index of the blocks, which makes limited random-access reading possible when the block size is small enough.

This should be sufficient for your purpose. A drawback is that the API of liblzma (for interacting with these containers) does not seem that well-documented, so it may take some effort figuring out how to randomly access blocks.

Share Improve this answer

answered May 3, 2014 at 11:53

Follow



AardvarkSoup

1,081 ● 8 ● 18

-
- 4 Yes, that's used for instance by `pixz` for random access of members of tar archives, or `nbdkit` for accessing xz compressed files as nbd devices (to be able to mount compressed disk images for instance). `qcow2` (native format for qemu disk images) is another format that allows compression and random access. – [Stephane Chazelas](#) Jun 2, 2016 at 9:16
-



The gzip format can be randomly accessed provided an index has been previously created, as it is demonstrated on [zlib's zran.c source code](#).

8

I've developed a command line tool upon zlib's `zran.c` which creates indexes for gzip files:

<https://github.com/circulosmeos/gztool>



It can even create an index for a still-growing gzip file




(for example a log created by rsyslog directly in gzip format) thus reducing in the practice to zero the time of index creation. See the `-s` (*Supervise*) option.

Share Improve this answer

edited Sep 15, 2019 at 18:00

Follow

answered Jul 24, 2019 at 21:10

 [circulosmeos](#)
454 ● 1 ● 6 ● 20



7

`bgzip` can compress files in a `gzip` variant which is indexable (and can be decompressed by `gzip`). This is used in some bioinformatics applications, together with the `tabix` indexer.



See explanations here:

<http://blastedbio.blogspot.fr/2011/11/bgzf-blocked-bigger-better-gzip.html>, and here:

<http://www.htslib.org/doc/tabix.html>.

I don't know to what extent it is adaptable to other applications.

Share Improve this answer

edited Nov 1, 2017 at 18:29

Follow

answered Feb 4, 2016 at 13:11



bli

8,174 ● 8 ● 55 ● 101

-
- 1 FYI: **tabix** does indexing and access by biological coordinates (e.g. chromosome + nucleotide position) while [grabix](#) does indexing and access by file coordinates (e.g. line numbers). They both work great for tabular data, other data may be tricky. – [jena](#) Mar 30, 2022 at 19:07
-
- 1 BTW Heng Li has a nice backstory to `bgzip` and the BGZF format on [his blog](#) and even a [piece of timeline](#) of the ideas. It's kind of mindblowing how it all happened within the span of two months. – [jena](#) Mar 30, 2022 at 20:10 ✎
-



4



Because lossless compression works better on some areas than others, if you store compressed data into blocks of convenient length BLOCKSIZE, even though each block has exactly the same number of compressed bytes, some compressed blocks will expand to a much longer piece of plaintext than others.

You might look at "Compression: A Key for Next-Generation Text Retrieval Systems" by Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates in *Computer* magazine November 2000 <http://doi.ieeecomputersociety.org/10.1109/2.881693>

Their decompressor takes 1, 2, or 3 whole bytes of compressed data and decompresses (using a vocabulary

list) into a whole word. One can directly search the compressed text for words or phrases, which turns out to be even faster than searching uncompressed text.

Their decompressor lets you point to any word in the text with a normal (byte) pointer and start decompressing immediately from that point.

You can give every word a unique 2 byte code, since you probably have less than 65,000 unique words in your text. (There are almost 13,000 unique words in the KJV Bible). Even if there are more than 65,000 words, it's pretty simple to assign the first 256 two-byte code "words" to all possible bytes, so you can spell out words that aren't in the lexicon of the 65,000 or so "most frequent words and phrases". (The compression gained by packing frequent words and phrases into two bytes is usually worth the "expansion" of occasionally spelling out a word using two bytes per letter). There are a variety of ways to pick a lexicon of "frequent words and phrases" that will give adequate compression. For example, you could tweak a LZW compressor to dump "phrases" it uses more than once to a lexicon file, one line per phrase, and run it over all your data. Or you could arbitrarily chop up your uncompressed data into 5 byte phrases in a lexicon file, one line per phrase. Or you could chop up your uncompressed data into actual English words, and put each word -- including the space at the beginning of the word -- into the lexicon file. Then use "sort --unique" to eliminate duplicate words in that lexicon file. (Is picking

the perfect "optimum" lexicon wordlist still considered NP-hard?)

Store the lexicon at the beginning of your huge compressed file, pad it out to some convenient BLOCKSIZE, and then store the compressed text -- a series of two byte "words" -- from there to the end of the file. Presumably the searcher will read this lexicon once and keep it in some quick-to-decode format in RAM during decompression, to speed up decompressing "two byte code" to "variable-length phrase". My first draft would start with a simple one line per phrase list, but you might later switch to storing the lexicon in a more compressed form using some sort of incremental coding or zlib.

You can pick any random even byte offset into the compressed text, and start decompressing from there. I don't think it's possible to make a finer-grained random access compressed file format.

Share Improve this answer

answered [Aug 7, 2010 at 20:52](#)

Follow

community wiki
[David Cary](#)



4

Two possible solutions:

1. Let the OS deal with compression, create and mount a compressed file system (SquashFS, clicfs, cloop,



cramfs, e2compr or whatever) containing all your text files and don't do anything about compression in your application program.

2. Use clicfs directly on each text file (one clicfs per text file) instead of compressing a filesystem image. Think of "mkclicfs mytextfile mycompressedfile" being "gzip <mytextfile >mycompressedfile" and "clicfs mycompressedfile directory" as a way of getting random access to the data via the file "directory/mytextfile".

Share Improve this answer

Follow

edited Feb 10, 2012 at 18:32



[animuson](#) ♦

54.7k ● 28 ● 141 ● 150

answered Feb 10, 2012 at 16:52



[Joachim Wagner](#)

41 ● 1

Wow, interesting thoughts on an old question of mine. Your first suggestion (squashfs) is not entirely what I would want, because it has implications for remote storage: using a compressed filesystem and compressed SSH connections, you would manage to decompress the data and recompress it to send it over the network. What would be amazing would be something like a compressed filesystem that you could share via NFS. Which I guess is what your clicfs suggestion might yield. Documentation on clicfs seems quite hard to come by (at least by my quick search), but it's promising.

Thank you. – [John Zwinck](#) Feb 11, 2012 at 0:21

From the information in the original question, SquashFS is exactly what you're asking for. It would of course be ideal if you didn't have to decompress and recompress over a

network, but if your SquashFS is set up with a fast decompression algorithm, then the total cost of the decompress + compress is presumably negligible. – [malthe](#)

Mar 7, 2019 at 7:47



4



I don't know if its been mentioned yet, but the [Kiwix project](#) had done great work in this regard. Through their program Kiwix, they offer random access to [ZIM file archives](#). Good compression, too. The project originated when there was a demand for offline copies of the Wikipedia (which has reached above 100 GB in uncompressed form, with all media included). They have successfully taken a 25 GB file (a single-file embodiment of the Wikipedia without most of the media) and compressed it to a measly 8 GB zim file archive. And through the Kiwix program, you can call up any page of the Wikipedia, with all associated data, faster than you can surfing the net.

Even though Kiwix program is a technology based around the Wikipedia database structure, it proves that you can have excellent compression ratios and random access simultaneously.

Share Improve this answer

Follow

edited Sep 6, 2020 at 14:48



agc

8,366 ● 2 ● 31 ● 51

answered Apr 8, 2013 at 3:14



CogitoErgoCogitoSum

490 ● 5 ● 7



3

I'm not sure if this would be practical in your exact situation, but couldn't you just gzip each large file into smaller files, say 10 MB each? You would end up with a



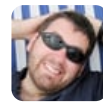
bunch of files: file0.gz, file1.gz, file2.gz, etc. Based on a given offset within the original large, you could search in the file named `"file" + (offset / 10485760) + ".gz"`. The offset within the uncompressed archive would be `offset % 10485760`.

Share Improve this answer

edited Jan 10, 2009 at 6:43

Follow

answered Jan 9, 2009 at 22:41



William Brendel

32.2k ● 14 ● 73 ● 77

-
- 3 Or you could TAR them all and end up with a .GZ.TAR. :)
– [Vilx-](#) Jan 9, 2009 at 22:48

That would definitely make things cleaner. I was just trying to go for simplicity here, but your suggestion is well taken :-)
– [William Brendel](#) Jan 9, 2009 at 22:49

-
- 2 .gz.tar is not really random access, since you must jump through all the headers to get to one file – [jpalecek](#) Jan 9, 2009 at 23:03

Well, yes and no. With fixed size chunks (10 MB in this case), you would not have to walk through a list of headers. This relies on the assumption that the tar will order the files alphabetically (which happens to be the case in GNU-land).
– [William Brendel](#) Jan 9, 2009 at 23:13

Yes, but the files would not be compressed then (10 MB uncompressed for your indexing expression to work, 10 MB compressed for direct access in tar to work). It's hard to compress anything to a fixed size, although you could make that size sufficiently large and handle excess space with sparse files – [jpalecek](#) Jan 9, 2009 at 23:30



1



I am the author of an open-source tool for compressing a particular type of biological data. This tool, called `starch`, splits the data by chromosome and uses those divisions as indices for fast access to compressed data units within the larger archive.



Per-chromosome data are transformed to remove redundancy in genomic coordinates, and the transformed data are compressed with either `bzip2` or `gzip` algorithms. The offsets, metadata and compressed genomic data are concatenated into one file.

Source code is available from our [GitHub](#) site. We have compiled it under Linux and Mac OS X.

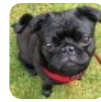
For your case, you could store (10 MB, or whatever) offsets in a header to a custom archive format. You parse the header, retrieve the offsets, and incrementally `fseek` through the file by `current_offset_sum + header_size`.

Share Improve this answer

[edited Jan 29, 2015 at 21:04](#)

Follow

answered Oct 26, 2011 at 21:02



Alex Reynolds

96.8k ● 59 ● 250 ● 351

Updated link to Github site. – [Alex Reynolds](#) Jan 29, 2015 at 21:04

"BEDOPS also introduces a novel and lossless compression format called Starch that reduces whole-genome BED datasets to ~5% of their original size (and BAM datasets to roughly 35% of their original size)" <-- This is amazing. You should advertise your tool. – [tommy.carstensen](#) Jan 29, 2015 at 21:14

We wrote a paper:

bioinformatics.oxfordjournals.org/content/28/14/1919.abstract

– [Alex Reynolds](#) Jan 29, 2015 at 21:17

Samtools faidx doesn't compress near as well as Starch, and it requires keeping a second file with the genomic data, but it offers finer indexing and so is more popular. Starch works really well if you need to squeeze out space or you're doing whole-genome work and want to parallelize tasks by chromosome. I'm working on "Starch 2", which will offer base-level interval queries, but that may be a few months out.

– [Alex Reynolds](#) Jan 29, 2015 at 21:20 ✎

Compression of bam to 35% is even better than the cram format. I must read the paper when home. I can't believe this is not widely used. – [tommy.carstensen](#) Jan 29, 2015 at 21:22
