25 Oct 2016 | 6 min. (1161 words)

How LZ4 works

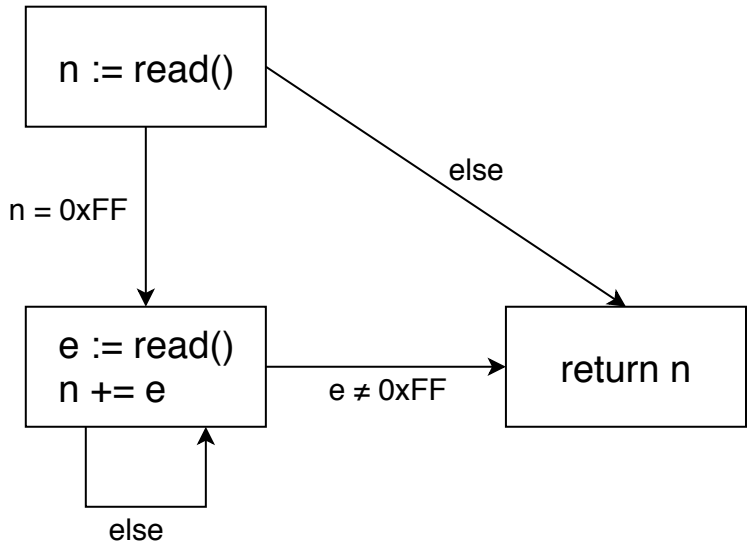
LZ4 is a really fast compression algorithm with a reasonable compression ratio, but unfortunately there is limited documentation on how it works. The only explanation (not spec, explanation) can be found on the author's blog, but I think it is less of an explanation and more of an informal specification.

This blog post tries to explain it such that anybody (even new beginners) can understand and implement it.

...r small-integer code (LSIC)

The first part of LZ4 we need to explain is a smart but simple integer encoder. It is very space efficient for 0–255, and then grows linearly, based on the assumption that the integers used with this encoding rarely exceeds this limit, as such it is only used for small integers in the standard.

It is a form of addition code, in which we read a byte. If this byte is the maximal value (255), another byte is read and added to the sum. This process is repeated until a byte below 255 is reached, which will be added to the sum, and the sequence will then end.



In short, we just keep adding bytes and stop when we hit a non-0xFF byte.

We'll use the name "LSIC" for convinience.

Block

eam is divided into segments called "blocks". Blocks contains a liter is opied directly to the output stream, and then a back reference, which t to copy some number of bytes from the already decompressed stream.

This is really were the compression is going on. Copying from the old stream allows deduplication and runs-length encoding.

Overview

A block looks like:

$$\overbrace{\underbrace{t_1}_{4\text{ bits}}\ \underbrace{t_2}_{4\text{ bits}}}^{\text{Token}}\quad \underbrace{\overbrace{e_1}^{\texttt{LISC}}}_{\text{If } t_1=15}\quad \underbrace{\overbrace{L}^{\text{Literal}}}_{t_1+e_1\text{ bytes}}\quad \underbrace{\overbrace{O}^{\text{Little endian}}}_{2\text{ bytes}}\quad \underbrace{\overbrace{e_2}^{\texttt{LISC}}}_{\text{If } t_2=15}$$

And decodes to the L segment, followed by a $t_2 + e_2 + 4$ bytes sequence copied from position $l - O$ from the output buffer (where l is the length of the output buffer).

We will explain all of these in the next sections.

Token

Any block starts with a 1 byte token, which is divided into two 4-bit fields.

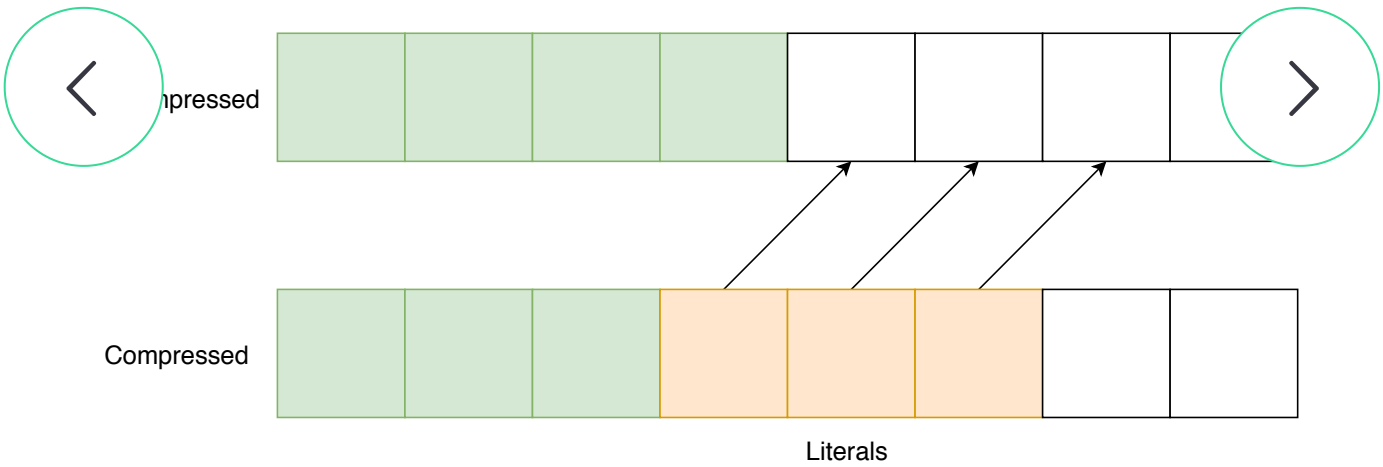
Literals

The first (highest) field in the token is used to define the literal. This obviously takes a value 0–15.

Since we might want to encode higher integer, as such we make use of LSIC encoding: If the field is 15 (the maximal value), we read an integer with LSIC and add it to the original value (15) to obtain the literals length.

Call the final value L.

Then we forward the next L bytes from the input stream to the output stream.



Deduplication

The next few bytes are used to define some segment in the already decoded buffer, which is going to be appended to the output buffer.

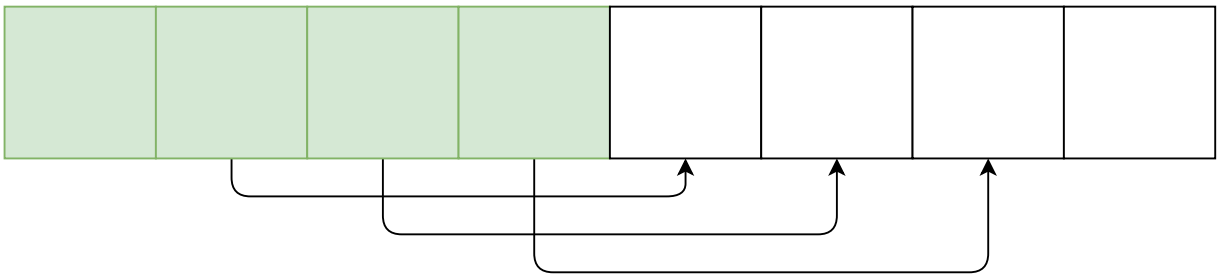
This allows us to transmit a position and a length to read from in the already decoded buffer instead of transmitting the literals themself.

To start with, we read a 16-bit little endian integer. This defines the so called offset, O. It is important to understand that the offset is not the starting position of the copied

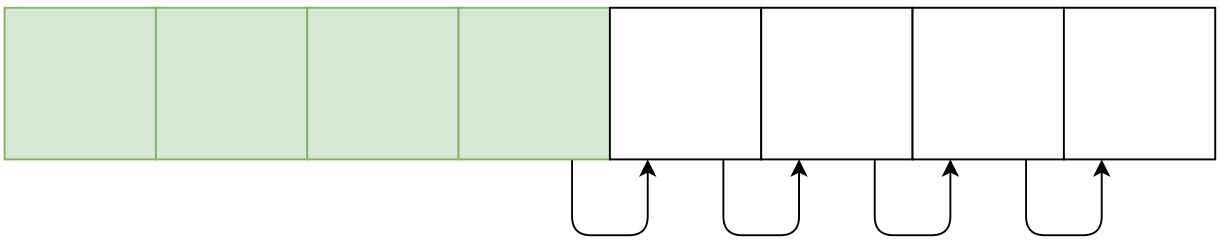
buffer. This starting point is calculated by $l - O$ with l being the number of bytes already decoded.

Secondly, similarly to the literals length, if t_2 is 15 (the maximal value), we use LSIC to "extend" this value and we add the result. This plus 4 yields the number of bytes we will copy from the output buffer. The reason we add 4 is because copying less than 4 bytes would result in a negative expansion of the compressed buffer.

Now that we know the start position and the length, we can append the segment to the buffer itself:



It is important to understand that the end of the segment might not be initialized when the rest of the segment is appended, because overlaps are allowed. This is useful, namely "runs-length encoding", where you repeat some sequence a number of times:



Note that the duplicate section is not required if you're in the end of the stream, i.e. if there's no more compressed bytes to read.

Compression

Until now, we have only considered decoding, not the reverse process.

A dozen of approaches to compression exists. They have the aspects that they need to be able to find duplicates in the already input buffer.

In general, there are two classes of such compression algorithms:

1. HC: High-compression ratio algorithms, these are often very complex, and might include steps like backtracking, removing repeatation, non-greediy.
2. FC: Fast compression, these are simpler and faster, but provides a slightly worse compression ratio.

We will focus on the FC-class algorithms.

Binary Search Trees (often B-trees) are often used for searching for duplicates. In particular, every byte iterated over will add a pointer to the rest of the buffer to a B-tree, we call the "duplicate tree". Now, B-trees allows us to retrieve the largest element smaller than or equal to some key. In lexiographic ordering, this is equivalent to asking the element sharing the largest number of bytes as prefix.

For example, consider the table:

```
1 abcdddd => 0
  bcdddd  => 1
  cdd     => 2
  dd      => 3
```

If we search for `cddda`, we'll get a partial match, namely `cdddd => 2`. So we can quickly find out how many bytes they have in common as prefix. In this case, it is 4 bytes.

What if we found no match or a bad match (a match that shares less than some threshold)? Well, then we write it as literal until a good match is found.

As you may notice, the dictionary grows linearly. As such, it is important that you reduce memory once in a while, by trimming it. Note that just trimming the first (or last) N entries is inefficient, because some might be used often. Instead, a **cache replacement policy** should be used. If the dictionary is filled, the cache replacement policy should determine which match should be replaced. I've found PLRU a good choice of CRP for LZ4 compression.

Note that you should add additional rules like being addressible (within $2^{16} + 4$ bytes of the cursor, which is required because O is 16-bit) and being above some length (smaller keys have worse block-level compression ratio).

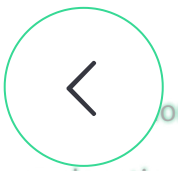
Another faster but worse (compression-wise) approach is hashing every four bytes and placing them in a table. This means that you can only look up the latest sequence given some 4-byte prefix. Looking up allows you to progress and see how long the duplicate sequence match. When you can't go any longer, you encode the literals section until another duplicate 4-byte is found.

Conclusion

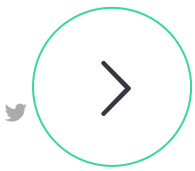
LZ4 is a reasonably simple algorithm with reasonably good compression ratio. It is the type of algorithm that you can implement on an afternoon without much complication.

If you need a portable and efficient compression algorithm which can be implement in only a few hundreds of lines, LZ4 would be my go-to.

Follow me on Twitter or Github.



on lz4 algorithms notes explanation data



ALSO ON **TICKI.GITHUB.IO**

Why I'm leaving Open Source · Ticki's blog

2 years ago • 5 comments

This blog contains my thoughts, ideas, and humble discoveries in …

Skip Lists: Done Right · Ticki's blog

4 years ago • 13 comments

Skip lists are a wonderful data structure, but it is hard to get\ right. This blog …

SeaHash: E Ticki's blog

4 years ago •

I explain how works.

1 Comment **ticki.github.io** **Sandeep**

Recommend 5 Tweet Share **Sort by Best**

Join the discussion…

Jordan Danford • 3 years ago

hanks so much for putting this explanation together! Found a o in the "Block" section: "This is really were the compression is ing on."

1 • Reply • Share ›