Hello everyone at Ethereum Research, a while back I spoke to you all about random data compression and how I believe it may be possible. I've since made considerable progress from that point, including getting to the point where I believe I can demonstrate my approach. Though my code is designed to run on a Spark cluster – I use 240~ cores for testing 1 / 61 completion of the algorithm which takes a while. In practice a full run of the algorithm would benefit from a larger cluster.

The link to the paper I wrote for my approach is here:16 Apr, 2021 - An algorithm for random data compression

My attempt to give a simple el5:

1. Store data in 17 bit words in a golomb-coded set (GCS) – this has the same properties of a bloom filter but it's smaller.
2. Entire algorithm is then based on recreating words stored in the set.
3. Make a list for each word filled by brute forcing every value in a 17 bit word at each offset. These are the candidates.
4. Write down the offsets in these lists to the correct word. These are the nodes.
5. Break down the nodes into lists of 8 elements. Inside these 8 element lists, break them into pairs.
6. Hash each pair, to make a hash list of four hashes.
7. Do proof-of-work on each hash, look for patterns that are highly unlikely. Find a nonce that makes the most unlikely pattern.
8. Store information about the nonce patterns in meta data with the nonce.
9. You can now use the meta-data to build a list of possible 8 element offset lists.
10. You write down the correct offset of the 8 element set you want.

I've defined a new algorithm for representing information in a probabilistic fashion that can be used to compress random data. It relies on using the unlikely prefixes in nonce patterns as heuristic filters to recreate indexes that serve as data pointers into a list of tables. Collisions are an expected property of the GCS – and the algorithm is all about reducing result sets in two main stages – the GCS candidates and the q set candidates.

People are going to ask me how this addresses the pigeon hole principle and the answer is that I don't know. In my mind the GCS structure is a super-position that allows for multiple over-lapping pieces of data to be re-mapped to smaller keys. That's an inherent properly of what a GCS allows. The collisions are resolved by addressing the indexes to the right words, and then the right list of index lists at the end. This bypasses the address-size restrictions with the counting argument. That's how I visualize it… maybe I'm wrong though.

At the very least this heuristic algorithm can do things within space restrictions that I've never before seen with any other algorithm and I think this makes this a new approach that could be potentially useful. I don't claim to have all the answers and parts of the paper might make no sense. I'd appreciate it if some smart people could give this a look though! P.S. will be writing basic how-tos for my code experiments on Github but you will need a small cluster to run them!