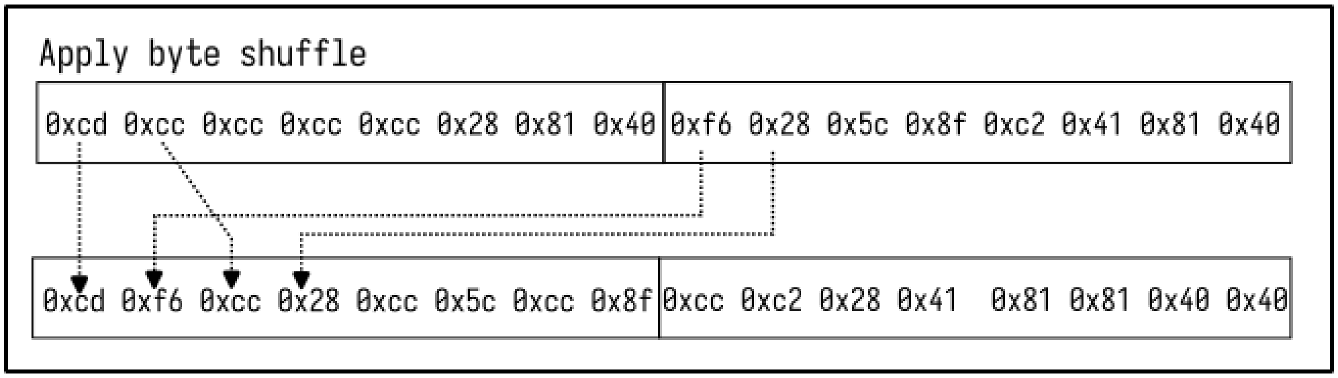
Compression with byte shuffling

For some numerical data, it is possible to achieve (much) better compression performance by re-arranging the bytes (or bits) of the elements of the data array prior to compressing them. This has to do with how numbers are laid out in memory, and mileage may vary as to whether shuffling always improves compression ratios. Also, as a note to some readers, while we use the term "shuffle", *this is not a random operation* analogous to shuffling a deck of cards is supposed to be, but closer in execution to a structured re-arrangement of bytes in multi-byte data types.

```
[549.1   552.22]
```

We can view these double-precision floats (64 bits) as contiguous blocks of 8 bytes (8 bits) each These bytes are what a compression algorithm will see when it tries to compact your data. Byte-shuffling re-arranges your data so that the ith byte of each element is stored contiguously:


Apply byte shuffle
0xcd 0xcc 0xcc 0xcc 0xcc 0x28 0x81 0x40
0xf6 0x28 0x5c 0x8f 0xc2 0x41 0x81 0x40
0xcd 0xf6 0xcc 0x28 0xcc 0x5c 0xcc 0x8f
0xcc 0xc2 0x28 0x41 0x81 0x81 0x40 0x40

Below is some idiomatic Python code to implement this concept:

```python
import numpy as np

def byte_shuffle_encode(x: np.array):
    byte_buffer = x.view(np.uint8)
    return byte_buffer.reshape((-1, x.itemsize)).T.flatten()

def byte_shuffle_decode(byte_buffer: np.array, dtype: np.dtype):
    return byte_buffer.reshape((dtype().itemsize, -1)).T.flatten().view(dtype)
```

Also, a more explicit C++ implementation that handles endianess:

```cpp
#include <vector>
#include <cstdint>
#include <bit>

using byte_t = std::uint8_t;
using buffer_t = std::vector<byte_t>;

constexpr bool is_big_endian(void)
{
    return std::endian::native == std::endian::big;
}

/// @brief Shuffle the bytes of `data` into `buffer`. Also enforces little-endian ordering
/// @tparam T
/// @param data The data to transpose
/// @param buffer Where to transpose the data into
template <typename T>
void transpose(const std::vector<T> &data, buffer_t &buffer)
{
    auto nData = data.size();
    auto nBytes = nData * sizeof(T);
    buffer.clear();
    buffer.reserve(nBytes);
    for (size_t i = 0; i < sizeof(T); i++)
    {
        for (size_t j = 0; j < data.size(); j++)
        {
            auto value = data[j];
            auto byteView = reinterpret_cast<uint8_t *>(&value);
            if (is_big_endian()) {
                buffer.push_back(byteView[(sizeof(T) - 1) - i]);
            } else {
                buffer.push_back(byteView[i]);
            }
        }
    }
    return;
}

template <typename T>
void reverse_transpose(const buffer_t &buffer, std::vector<T> &data)
{
    auto nBytes = buffer.size();
    auto nData = nBytes / sizeof(T);
    data.resize(nData);
    for (size_t i = 0; i < buffer.size(); i++)
    {
        auto datum = &data[i % nData];
        auto byteView = reinterpret_cast<uint8_t *>(datum);
        if (is_big_endian()) {
            byteView[(sizeof(T) - 1) - (i / nData)] = buffer[i];
        } else {
            byteView[i / nData] = buffer[i];
        }
    }
    return;
}
```

Dictionary Encoding

Motivation

Some data arrays are hard to compress simply because their data spans a wide range of possible values but lacks structures or patterns that make compression possible despite many repetitions of the same value, simply because of how they are organized.

A good example of this is ion mobility, where the values span an instrument-specific domain with varying levels of precision, and these points are repeated out-of-order because ion mobility arrays are sorted along the m/z axis.

One thing a compression algorithm does is finds repeated patterns and encodes the pattern in a lookup table or "dictionary" and associates a unique token or placeholder that uses less space than the original pattern. How big patterns may be, what criteria they must satisfy, and other things, depend upon the algorithm, but these algorithms generally don't know much about the data they compress, which prevents them from making simplifying assumptions.

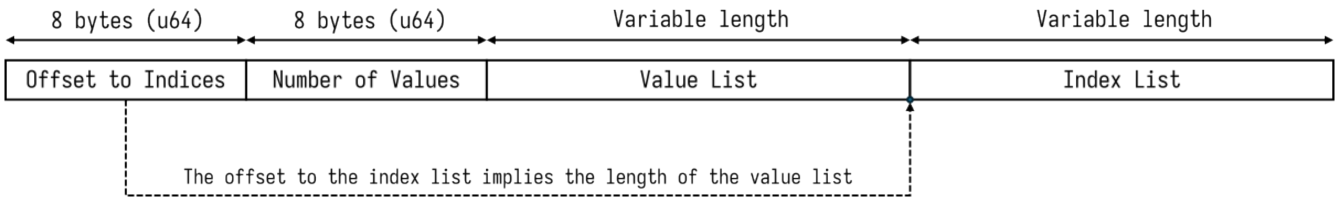
Since **we** know the context in which we are working, we can pre-construct a dictionary to encode the data with, and apply the general purpose compression algorithm on that the encoding instead, assuming it is simpler than the original. At the same time, we recognize that being *very, very clever* to shave 5% off the total size of our data but produce an algorithm so convoluted nobody but the creator could implement in multiple languages is a recipe for a waste of time so we hope to make the algorithm simple enough that other people can understand it if the portfolio of reference implementations aren't sufficient for their needs.

Method

Encoding

We construct a lookup table by first collecting all the unique values in a data array V, called V^*, and sort V^* so adjacent values are close together to hopefully make them easier to compress later. Based upon the number of unique items, we can determine the largest bit width "index" or "key" type we need to address them all. For ion mobility data at the time of this writing, there are fewer than 1,000 unique values recorded by instruments today, which fits tidily in use 216 bits. Next, we build a mapping from value to its index in V^* using the smallest standard integer type I^*, and create a new array $K = [\text{map}(v_j) : v \in V]$.

We can then write the following structure in bytes:



We prefix the dictionary encoding with two 8 byte unsigned integers in little endian order. The first is the distance, in bytes, from the begining of the buffer to the begining of the region encoding K, i_{offset}. The second is the number of unique values in V^*. Next, we encode the bytes ofV^* in the buffer in little endian byte order, followed by the bytes of K in little endian byte order.

Decoding

To decode, we first read the first 16 bytes and interpret it as two 64 bit unsigned integers which tells us the offset i_{offset} to the index list K and the number of values in V^*, $|V^*|$. We assume we know the type of value we want to decode V^* into, so we can validate that $\frac{i_{\text{offset}}-16}{|V^*|}$ matches the size of that type in bytes. It is then straight-forwards to reconstruct V^* from byte 16 to the index offset, reinterpreting the bytes as needed to build a native representation of V^* in the correct type. Next, we infer the I^* from $|V^*|$ using the same logic as during encoding. Finally, we reconstruct the original data array V using $V = [V^*[k_i] : k_i \in K]$.

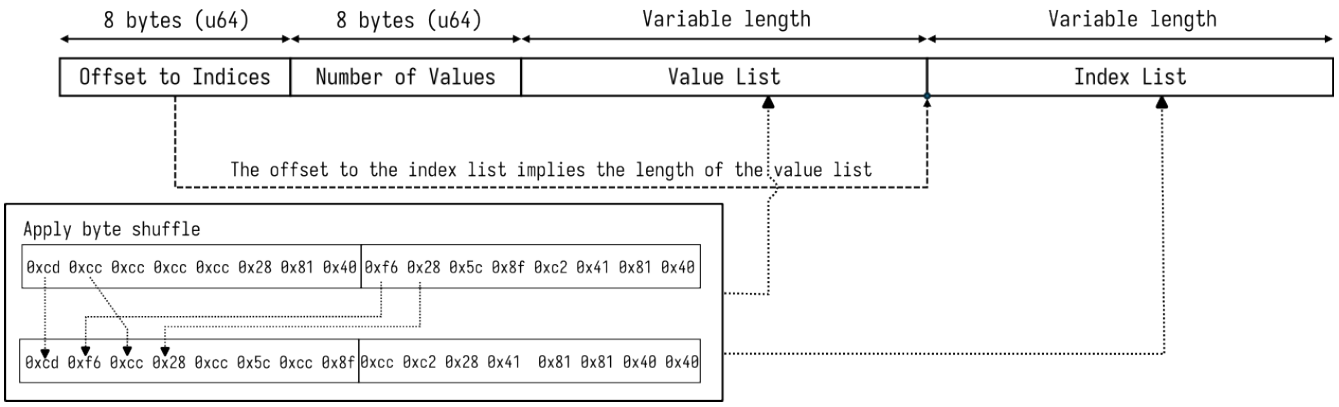
`zstd.compress(dict_encode(V))` is reversed by `dict_decode(zstd.decompress(buffer), type_of_V)`

Below is Python code implementing this method:

```python
import numpy as np
import struct

bytes_views = {
    8: np.uint64,
    4: np.uint32,
    2: np.uint16,
    1: np.uint8,
}
index_widths = [
    (2**8, np.uint8),
    (2**16, np.uint16),
    (2**32, np.uint32),
    (2**64, np.uint64),
]

def dict_encode(data: np.typing.ArrayLike) -> bytes:
    byte_view_tp = bytes_views[data.itemsize]
    view = data.view(byte_view_tp)
    values = np.sort(np.unique(view))

    index_tp = None
    for size, idx_type in index_widths:
        if len(values) < size:
            index_tp = idx_type
            break
    else:
        raise ValueError(f"Cannot find index dtype for {len(values)} values")

    indices = np.zeros(len(view), dtype=index_tp)
    for i, k in enumerate(values):
        indices[view == k] = i

    n_values = np.uint64(len(values))
    total_size = 16 + values.nbytes + indices.nbytes
    buffer = np.empty(total_size, dtype=np.uint8)
    buffer[:8] = memoryview(struct.pack("<Q", np.uint64(16 + values.nbytes)))
    buffer[8:16] = memoryview(struct.pack("<Q", np.uint64(len(values))))
    buffer[16 : values.nbytes + 16] = values.view(np.uint8)
    buffer[(16 + values.nbytes) :] = indices.view(np.uint8)
    return buffer


def dict_decode(buffer: np.typing.ArrayLike, dtype: np.dtype) -> bytes:
    index_tp = None
    byte_view_tp = bytes_views[dtype().itemsize]

    (offset_to_data,) = struct.unpack("<Q", buffer[:8])
    (n_values,) = struct.unpack("<Q", buffer[8:16])
    for size, idx_type in index_widths:
        if size > n_values:
            index_tp = idx_type
            break
    else:
        raise ValueError(f"Cannot find index dtype for {n_values} indices")
    values = buffer[16:offset_to_data].view(byte_view_tp)
    indices = buffer[offset_to_data:].view(index_tp)
    return values[indices].view(dtype)
```

Combining dictionaries and byte shuffling

When the dictionary is large, and especially when the values are large, it can be advantageous to apply byte shuffling to the values list and the index list separately. This works in two ways. The first is that the sorted value array V^* is more likely to compress better when shuffled for the same reason byte shuffling works well on an m/z array. The second is that the larger, repetitive index array K, being an integer type, is more compressable than a floating point type when shuffled, even unsorted.



8 bytes (u64)
8 bytes (u64)
Variable length
Variable length
Offset to Indices
Number of Values
Value List
Index List
The offset to the index list implies the length of the value list
Apply byte shuffle
0xcd 0xcc 0xcc 0xcc 0xcc 0x28 0x81 0x40
0xf6 0x28 0x5c 0x8f 0xc2 0x41 0x81 0x40
0xcd 0xf6 0xcc 0x28 0xcc 0x5c 0xcc 0x8f
0xcc 0xc2 0x28 0x41 0x81 0x81 0x40 0x40

This is relatively straight-forwards to do by injecting the shuffling step immediately prior to encoding the dictionary buffers and decoding them prior to using them as an index map.

```
def dict_encode_with_shuffle(data: np.typing.ArrayLike) -> bytes:
    byte_view_tp = bytes_views[data.itemsize]
    view = data.view(byte_view_tp)
    values = np.sort(np.unique(view))

    index_tp = None
    for size, idx_type in index_widths:
        if len(values) < size:
            index_tp = idx_type
            break
    else:
        raise ValueError(f"Cannot find index dtype for {len(values)} values")

    indices = np.zeros(len(view), dtype=index_tp)
    for i, k in enumerate(values):
        indices[view == k] = i

    values = byte_shuffle_encode(values)
    indices = byte_shuffle_encode(indices)

    n_values = np.uint64(len(values))
    total_size = 16 + values.nbytes + indices.nbytes
    buffer = np.empty(total_size, dtype=np.uint8)
    buffer[:8] = memoryview(struct.pack("<Q", np.uint64(16 + values.nbytes)))
    buffer[8:16] = memoryview(struct.pack("<Q", np.uint64(len(values))))
    buffer[16 : values.nbytes + 16] = values.view(np.uint8)
    buffer[(16 + values.nbytes) :] = indices.view(np.uint8)
    return buffer


def dict_decode_with_shuffle(buffer: np.typing.ArrayLike, dtype: np.dtype) -> bytes:
    index_tp = None
    byte_view_tp = bytes_views[dtype().itemsize]

    (offset_to_data,) = struct.unpack("<Q", buffer[:8])
    (n_values,) = struct.unpack("<Q", buffer[8:16])
    for size, idx_type in index_widths:
        if size > n_values:
            index_tp = idx_type
            break
    else:
        raise ValueError(f"Cannot find index dtype for {n_values} indices")
    values = buffer[16:offset_to_data].view(byte_view_tp)
    indices = buffer[offset_to_data:].view(index_tp)

    indices = byte_shuffle_decode(indices.view(np.uint8), index_tp)
    values = byte_shuffle_decode(values.view(np.uint8), byte_view_tp)
    return values[indices].view(dtype)
```