Improved variable length raw forward index format

Goals

- Introduce new format for variable length raw forward indexes which achieves the following:
    - Balanced chunk size
    - Small number of chunks whenever a segment is sized appropriately
    - Low memory usage during ingestion
    - Permits usage in realtime segments

Non-Goals

- Replace current format by default. This format will require opt in configuration until or unless it is accepted by the community.
- Change chunk compression defaults, this is already configurable.

Problem

Raw forward indexes consist of compressed chunks of variable length data, as well as a header consisting of chunk offsets. This format requires that the number of documents per chunk is fixed[1]. While this works well for fixed width data or variable length data with very low variance (e.g. `VARCHAR(N)` for small `N`) this poses problems when the variance in length is high because of numerous constraints.

- Each value needs to fit entirely within a chunk, so the chunk size must be at least the size of the largest value.
- Every time a raw value is accessed, the chunk it resides in needs to be decompressed. The cost of decompressing a chunk, and therefore the overhead per random access, increases with the chunk size. ~1MB has been determined to be a good size (but this should be investigated without prejudice).
- Compression tends to be more effective for larger data sets as there tends to be commonality within a corpus of documents: in order to get good compression levels in the average case, we need to pack lots of documents into a single chunk.
- Compression libraries work on chunks, and many cannot produce a decompressible sequence of bytes unless the entire input was available at compression time, though many support streaming[2].
- The larger the number of documents per chunk, the larger the buffer for storing uncompressed values needs to be. If the number of documents is greater than 1, this can only safely be achieved by using a multiple of the length of the longest value in the segment, which leads to high memory usage.
- The more chunks, the more offsets are required, which increases the size of the header and decreases the efficiency of storage.

The effective outcome is a difficult choice about what to do when there is even a single large value:

- Enforce a minimum number of docs per chunk `N`, choose a chunk size equal to the largest value length times the number of docs: this needs lots of RAM, risks OOM.
- Enforce a maximum chunk `size = max(1MB, sizeof(longest value))`. This will never OOM[3], but the number of documents per chunk may be as low as 1. This

[1] Note that the chunk size is not consistent across columns or across segments, it is determined from the column statistics for each segment.
[2] Implementations of streaming compression are usually slower than block oriented compression APIs and tend to produce inferior compression ratios.
[3] Unless enormous values are encountered but this is impossible for other reasons like limits in Kafka payload sizes.

explodes the metadata, decreases chunk level compression ratios for small documents and means decompression cannot be amortised over a set of documents when scanning the index.

Current Format

The class `BaseChunkSVForwardIndexWriter` defines the format of the raw variable length forward index best. There have been numerous versions of this format but this section describes version 3; the current version.

The format consists of a 28 byte header, followed by a list of offsets to chunk starts, followed by the chunks.

The header consists of the following fields:

Offset	Name	Purpose	Size
0	`version`	Allow evolution of format	4
4	`numChunks`	Allows reader to resolve docId to chunk number	4
8	`numDocsPerChunk`	Allows resolution of docId to value within a decompressed chunk	4
12	`sizeOfEntry`	Used by reader for sizing of read buffer	4
16	`totalDocs`	Unused	4
20	`compressionType`	Compression algorithm metadata	4
24	`dataHeaderStart`	Where the chunks start in the file	4

The offsets which follow on from the header are fixed width; they were 32 bit values in versions 1 and 2, but currently have 64 bits in version 3. The offsets section of the file is not very large: a 1GB column split into 1MB chunks would require 1024 offsets or 8KB. Columns will usually be

much smaller than this. However, if there is one value in a segment column of, say, 10 million rows which is large enough to push the number of documents per chunk down to 1, there would be 10 million chunks, or 76MB of metadata.

Each chunk is a sequence of bytes delimited only by the offsets earlier on in the file. Individual use cases can be catered for by storing metadata within the chunk prior to compression, with the caveat that this can't be accessed without decompression of the chunk.

Advantages of the current format

- Offsets with a fixed number of documents allow constant time docId to chunk offset resolution by dividing the docId by the number of documents per chunk.
- Given a decompressed chunk, intrachunk metadata can be used to support random access, so that a document can be resolved by documentId in constant time.
- The header was carefully designed to be evolved!

Disadvantages of the current format

- A fixed number of documents per chunk creates a tradeoff between metadata size, compression ratio, and decoding efficiency on one side and buffer size in RAM on the other.
- The format requires knowledge of the maximum value length, which means it can't be used for realtime columns, as it depends on segment level statistics.
- The maximum size of a decompressed chunk is not recorded, so the reader does not have the option to size a fixed buffer to decompress into.

Observations

- **Observation 1:** if a target of 1MB per compressed chunk is achieved, sensible segment sizing keeps the number of chunks small. This means that:
    1. Random access to chunks can be traded for achieving a 1MB target.
    2. Some level of metadata bloat can be accepted in exchange for achieving a 1MB chunk size target.
- **Observation 2:** ascending sets of integers (i.e. `RoaringBitmap`) are used pervasively throughout Pinot for skipping over rows in columns. It might feel like there is a lot of

random access (`ForwardIndexReader` accepts docId as input and does not impose that they are sequential) but most access is really sequential. This means that:

1. Loss of random access to chunks can be ameliorated regardless of the number of chunks if the metadata allows for skipping over chunks to *advance until* the chunk containing a docId very quickly.

- **Observation 3:** whilst access to a given chunk is constant time if the number of documents per chunk is constant, compared to the cost of decompression, the advantage over slower search approaches is unclear. Assuming a decompression speed of 2GB/s for Snappy[4] - decompression of a 512KB compressed chunk will take 250us, which is over 1000x slower than the time to locate a chunk for a sensible number of chunks for a sensibly sized segment, even if linear search is adopted. With binary search and simple optimisations to exploit sequential access patterns, we can expect this ratio to be more like 25000-50000x; locating chunks cannot be a bottleneck.

Consider the following benchmark, which does not even consider more sophisticated approaches than binary and linear search:

```
@State(Scope.Benchmark)
public class Partitioning {

 @Param("1000")
 int size;
 @Param({"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"})
 int decile;

 private int[] sizes;

 private int target;
 int avgSize;

 @Setup(Level.Trial)
 public void setup() {
   sizes = new int[size];
   int sum = 0;
   for (int i = 1; i < size; i++) {
     int next = 100;
     sizes[i] = sizes[i - 1] + next;
     sum += next;
   }
   target = sizes[decile * (size / 10)];
   avgSize = sum / sizes.length;
```

[4] https://github.com/lz4/lz4#benchmarks

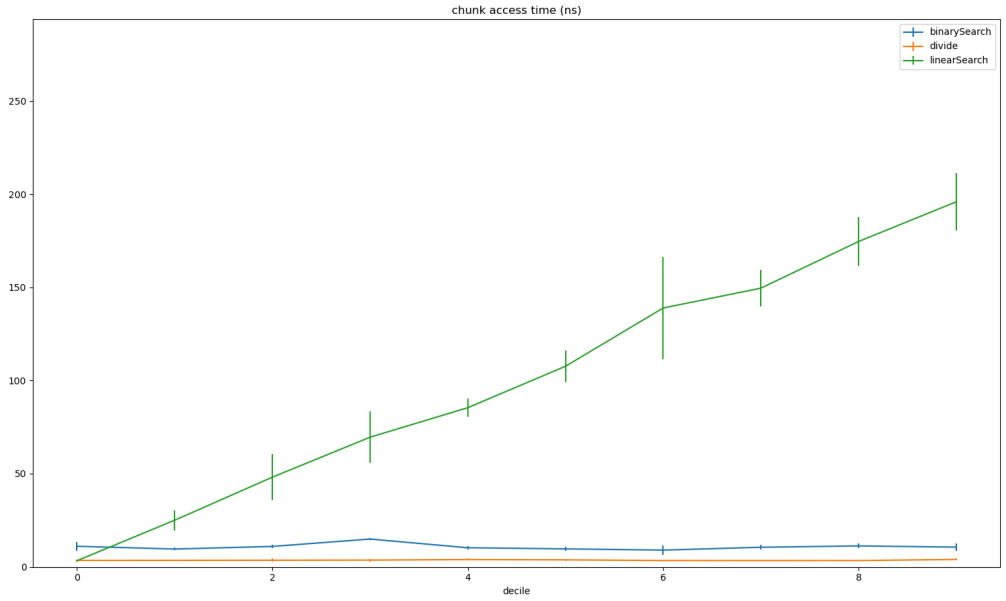
```
  }

  @Benchmark
  public int divide() {
    return target / avgSize;
  }

  @Benchmark
  public int linearSearch() {
    for (int i = 0; i < sizes.length; i++) {
      int size = sizes[i];
      if (size > target) {
        return i - 1;
      }
    }
    return sizes.length;
  }

  @Benchmark
  public int binarySearch() {
    int pos = Arrays.binarySearch(sizes, target);
    return pos >= 0 ? pos : -pos - 1;
  }
}
```

Random access has very little advantage over binary search for the numbers of chunks we would aim to have.



Benchmark	Mode	Threads	Samples	Score	Score Error (99.9%)	Unit	Param: decile	Param: size
binary	avgt	1	5	11.017476	2.431537	ns/op	0	1000
binary	avgt	1	5	9.53242	0.782707	ns/op	1	1000
binary	avgt	1	5	10.930717	0.944045	ns/op	2	1000
binary	avgt	1	5	14.898981	0.696605	ns/op	3	1000
binary	avgt	1	5	10.222341	0.833502	ns/op	4	1000
binary	avgt	1	5	9.616832	1.24699	ns/op	5	1000
binary	avgt	1	5	8.950074	2.473554	ns/op	6	1000
binary	avgt	1	5	10.484628	1.312675	ns/op	7	1000
binary	avgt	1	5	11.214457	1.277363	ns/op	8	1000
binary	avgt	1	5	10.565306	1.937458	ns/op	9	1000
divide	avgt	1	5	3.428586	0.168904	ns/op	0	1000
divide	avgt	1	5	3.448357	0.109429	ns/op	1	1000
divide	avgt	1	5	3.505568	0.896609	ns/op	2	1000

divide	avgt	1	5	3.585745	0.761142	ns/op	3	1000
divide	avgt	1	5	3.879105	0.485956	ns/op	4	1000
divide	avgt	1	5	3.733576	0.575513	ns/op	5	1000
divide	avgt	1	5	3.336567	0.245159	ns/op	6	1000
divide	avgt	1	5	3.323152	0.057803	ns/op	7	1000
divide	avgt	1	5	3.337873	0.232488	ns/op	8	1000
divide	avgt	1	5	3.95304	0.568197	ns/op	9	1000
linear	avgt	1	5	3.26149	0.570315	ns/op	0	1000
linear	avgt	1	5	25.006058	5.533474	ns/op	1	1000
linear	avgt	1	5	48.116374	12.453734	ns/op	2	1000
linear	avgt	1	5	69.589489	13.688593	ns/op	3	1000
linear	avgt	1	5	85.424912	4.8607	ns/op	4	1000
linear	avgt	1	5	107.66478	8.685049	ns/op	5	1000
linear	avgt	1	5	138.97430	27.409321	ns/op	6	1000
linear	avgt	1	5	149.62112	9.669953	ns/op	7	1000
linear	avgt	1	5	174.68570	13.030853	ns/op	8	1000
linear	avgt	1	5	195.99762	15.541902	ns/op	9	1000

Proposal

Make the following alterations to the file format

1. Increment the version in the header to 4.
2. Remove `numChunks, numDocsPerChunk, totalDocs` from the header.
3. Add the target chunk size to the header. Readers should use this to inform decompression buffer sizing policies. Chunks which require larger buffers than the target chunk size after decompression should be treated as exceptional and allocated for *just in time*. Mitigations for large data are discussed under “rejected alternatives” but in short, users with very large data should be able to configure a buffer size for their (rare) use

case, enough to prevent this from being frequent. Compression libraries can read compression metadata about the decompressed size of compressed data, so this does not need to be recorded for each chunk in the header.

4. Allocate a fixed capacity buffer of 1MB to buffer uncompressed documents into. If a document does not fit into the remainder of the buffer, compress the contents of the buffer and flush the chunk to disk. If after flushing, the value still does not fit in the buffer, compress the value and write it to disk as a single value chunk. **This breaks the dependency on segment level statistics and allows for usage for realtime segments.**
5. Record the smallest docId in the header alongside the offset to the chunk start. Since values are written in ascending docId order, this is effectively the cumulative count of documents in chunks up to but not including this chunk; the number of documents in the chunk *i* is $minDocId_{i+1} - minDocId_i$. Since the documents are stored in ascending docId order, the minimum docIds for each segment can be binary searched to locate the chunk in time similar to the existing random access.
    a. Knowing that document ids are accessed sequentially can be used to prune the space for the binary search by storing the last document id in the reader context.
    b. The cumulative counts correspond to the doc id range for each chunk, which can be stored in the reader context to avoid doing extra lookups (check id the doc id is in the current range)

The new header format would be

Offset	Name	Purpose	Size
0	`version`	Allow evolution of format	4
4	`targetDecompressedChunkSize`	Allows readers to size buffers for decompression purposes	4
8	`sizeOfEntry`	Used by reader for sizing of read buffer, max value tracked during building the segment	4
12	`compressionType`	Compression algorithm metadata	4
16	`dataHeaderStart`	Where the chunks start in the file	4

The header would be followed by repeated chunk metadata:

Name	Purpose	Size
Byte Offset	The start of the chunk in bytes	8
DocId Offset	The first doc id in the chunk. The MSB is used to mark huge chunks.	4

In mixed version clusters, reader support should be ubiquitous before segments using the version 4 format start getting generated, otherwise older pinot servers will not be able to read segments containing version 4 columns. Therefore, reader support should be released along with writer support behind a feature flag in version 0.9.0. The feature flag will be removed in the subsequent release, with a mandatory upgrade via 0.9.0 to 0.10.0. The current format is also used for mission critical applications by numerous Pinot users, and these users must be given the option to evaluate and accept the new format before making it the default.

Rejected Variations

Dynamic buffer allocation

The buffer could be dynamically allocated to create larger chunks than was targeted if larger data is encountered. Instead, a buffer of the target chunk size will be allocated. When a record doesn't fit in it, a flush happens - the buffer is compressed and written to disk. If after a flush happens the value *still* doesn't fit, the value itself is compressed and written to disk as a chunk. This avoids excessively sized lingering buffers because a large value was seen once. However, this leads to a failure mode where *all* values are larger than the target uncompressed chunk size, and we create as many chunks as there are documents. This case should be very rare, but should be catered for by a configurable buffer size, but a good value should be chosen as the default.

Store maximum size of uncompressed chunk in header to allow readers to size decompression buffers

The size of the target uncompressed chunk size is stored in the header, but not the size of the largest chunk size. When values larger than the target uncompressed size are encountered,

they are compressed directly, so the target chunk size is not guaranteed to be large enough for use within a pool of buffers decompression purposes.

If there are few enough chunks and access is sequential, chunk decompression should be rare for readers. Optimising memory allocation for decompression is likely a symptom of doing too much decompression because there are too many chunks or dealing with chunks that are too large. If most chunks are 1MB uncompressed and a *N*MB chunk is encountered just once, the size of the buffer pool would be oversized by a factor of *N*, so buffer sizing based on the maximum value leads to bloat; inefficient usage of RAM increases operating costs.

Instead, the reader should take the target chunk size from the header and preallocate buffers of that size if deemed necessary. On the *exceptional event* that this buffer is not large enough because a huge raw value was encountered, a large buffer can be allocated just in time for decompression. This could be detected by recovering the decompressed size from the compressed chunk by extending the `Decompressor` abstraction to expose the capabilities of the compression libraries.

This leads to an identical failure mode to above, where *most* chunks exceed the target uncompressed chunk size. In this case, the target uncompressed buffer size should be increased to match the needs of the use case, with a good default.