Separation of Ztoc and compression algorithm

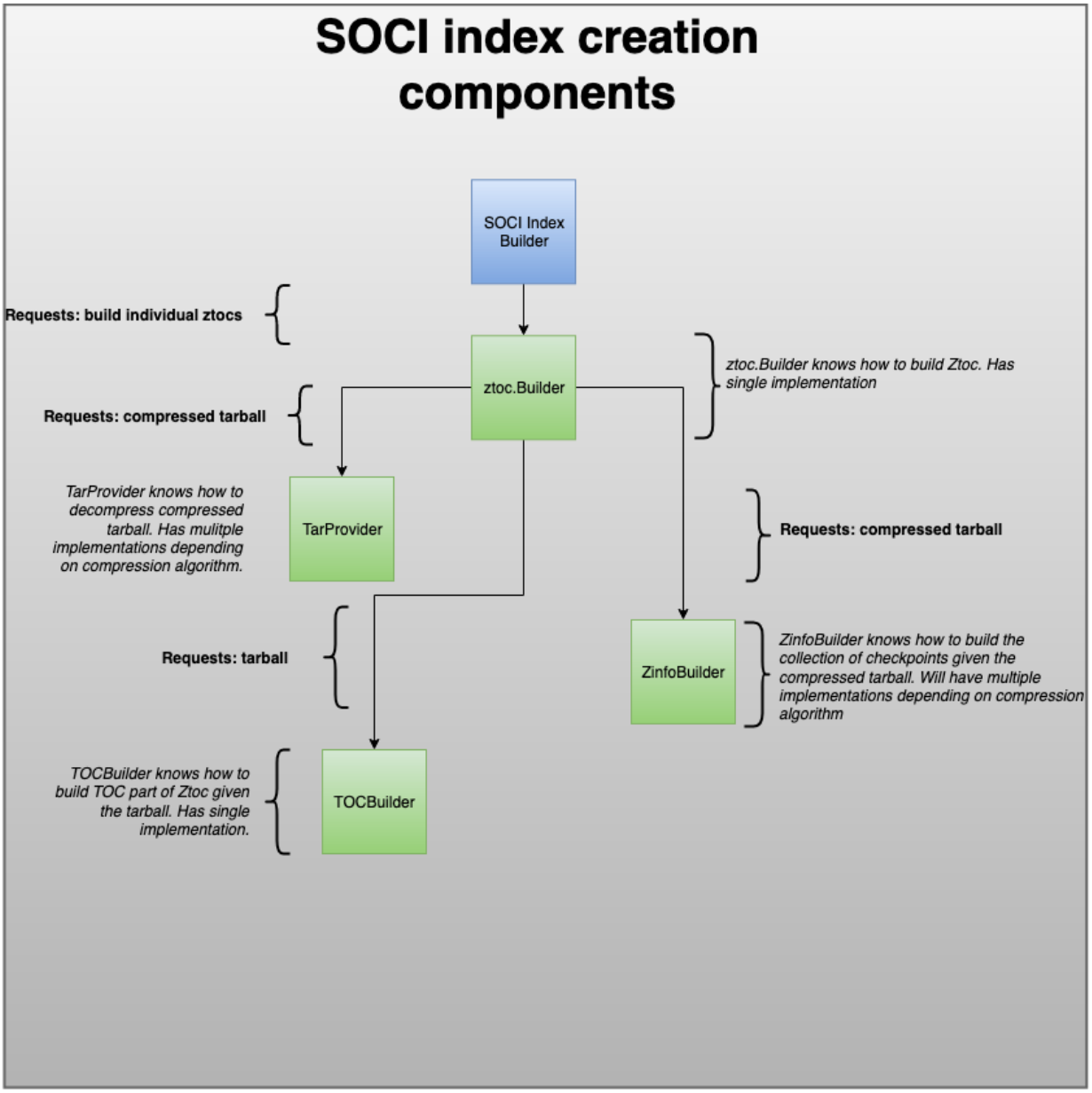
Problem statement and design goals

Soci snapshotter's Ztoc is tightly coupled with the compression algorithm. This results in inflexible implementation of Ztoc building as well as data extraction. This goes beyond the package soci and also has traces within span manager.
This design doc is trying to achieve the following:

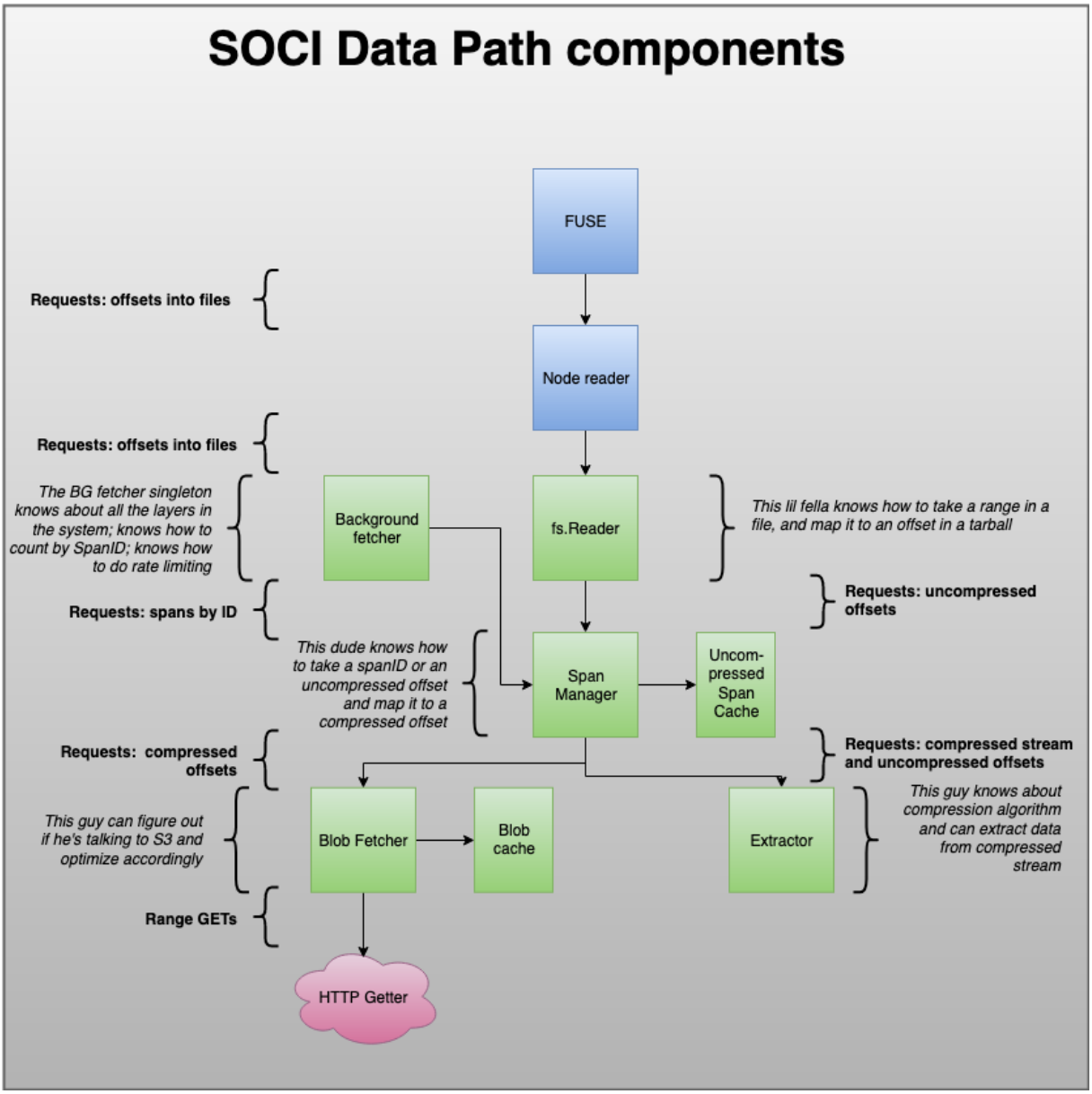
- It should be easy to support the new version of compression algorithm, by implementing the "compression" interface and plugging it in in a few places without any rewriting of the existing Ztoc building and file/data extraction logic.
- Separate the zTOC technology from anything related to containers, layers, or SOCI

To define the separable concepts and also to talk about interfaces and their usage in the new world, let's first define the big picture with the with all the components and then go through the stories on how they will stick together.

The Ztoc creation path components and their relations can be found on the diagram below:



SOCI data path components and their relations can be found on the diagram below (edited the original image from @wsm):



Stories for using Ztoc

There are several stories to use the Ztoc. Let's start defining the interfaces as the stories unfold.

ZTOC CREATION

To create the Ztoc, we need to build two separate pieces and combine them together. Those pieces are: `TOC` and `Zinfo`.

TOC contains the array of `FileMetadata` (https://github.com/awslabs/soci-snapshotter/blob/main/soci/ztoc.go#L71) and contains data, which can be obtained solely from the uncompressed tarball.
To be able to build TOC, we suggest using `TOCBuilder`, which will look as follows:

```
// TocBuilder knows how to deal with .tar and its purpose is to walk through the tarba
// metadata.
type TocBuilder struct {
}

func (tb *TocBuilder) Build(uncompressedTarball io.SectionReader) (*TOC, error) {
    panic("not implemented")
}
```

To be able to separate the TOC from compression, we suggest to remove `SpanStart` and `SpanEnd` fields from `ztoc.FileMetadata` (https://github.com/awslabs/soci-snapshotter/blob/main/soci/ztoc.go#L40-L41). Those fields are redundant and we can use the existing `UncompressedOffset` and `UncompressedSize` to compute span boundaries.

Zinfo contains span data along with the collection of checkpoints. This part of Ztoc is very dependent on compression algorithm. To build Zinfo, we will introduce the interface within the ztoc package, which will have compression specific implementation.

```
// ZinfoBuilder is responsible for consuming the compressed stream and producing the
// artifacts for building Ztoc
// It will have multiple implementations, based on compression algorithm,
// e.g. GzipZinfoBuilder, ZstdZinfoBuilder, etc
// And will be consuming and using the low-level compression algorithm specific
// implementations like GzipIndex, which directly call the C code.
type ZinfoBuilder interface {
    Build(compressedTarball io.SectionReader, options *ZinfoBuildOptions) (*Zinfo, err
    GetTarball(compressedTarball io.SectionReader) (tarball io.SectionReader, err erro
}

type ZinfoBuildOptions struct {
    SpanSize int64
}
```

For building Ztoc we suggest to introduce ztoc.Builder, which will have the state from ztoc.TOCBuilder and ztoc.ZinfoBuilder:

```
// ztoc.Builder knows how to build Ztoc.
type Builder struct {
    tocBuilder   TocBuilder
    zinfoBuilder ZinfoBuilder
    tarProvider  TarProvider
}
func NewBuilder(tb TocBuilder, zb ZinfoBuilder, tp TarProvider) (*Builder, error) {
    return &Builder{tocBuilder: tb, zinfoBuilder: zb, tarProvider: tp}, nil
}
```

The Ztoc building process will follow the following path:

```
func (zb *Builder) Build(compressedTarball io.SectionReader, options *ZtocBuildOptions
                                                                    (*Ztoc, error) {
    trb := compression.TarReaderBuilder{}
    tarball, err := zb.tarProvider.Decompress(compressedTarball)
    if err != nil {
        return nil, err
```

```
    }
    toc, err := tocBuilder.Build(tarball)
    if err != nil {
        return nil, err
    }
    zinfo, err := zinfoBuilder.Build(compressedTarball,
        &ZinfoBuildOptions {SpanSize: options.SpanSize})
    if err != nil {
        return nil, err
    }
    compressedArchiveSize := compressedTarball.Size()
    uncompressedArchiveSize := tarball.Size()
    return &Ztoc{
        TOC:                    toc,
        CompressedArchiveSize:   compressedArchiveSize,
        UncompressedArchiveSize: uncompressedArchiveSize,
        Zinfo:                  zinfo,
        // also version and buildtoolid
    }, nil
}
```

Since we require the tarball, we will define the `compression.TarProvider`

```
// TarProvider knows how to decompress the compressed tarball.
// Will have multiple implementations for different compression algorithms.
type TarProvider interface {
    Decompress(compressedTarball io.SectionReader) (tarball io.SectionReader, err err
}
```

Within `soci.buildSociLayer` we create the implementation of ZinfoBuilder, TarProvider based on the compression algorithm and TocBuilder. We pass them to create the ZtocBuilder and call Build on it, passing the compressed tarball along with options.

ZTOC SERIALIZATION AND DESERIALIZATION

When Ztoc is built, it can be serialized. To do so, we introduce the the `ztoc.ZtocSerializer`. The reason for that is that Ztoc is serialized with flatbuffers, which require to manually define what will be serialized and how. It will look as follows:

```
// ZtocSerialier incapsulates the flatbuffers implementation to
// serialize/deserialize Ztoc
type ZtocSerializer struct {}

func (zs *ZtocSerializer) Serialize(ztoc *Ztoc) (io.Reader, digest.Digest, error) {
    flatbuf := ztocToFlatbuffer(ztoc)
    buf := bytes.NewReader(flatbuf)
    dgst := digest.FromBytes(flatbuf)
    size := len(flatbuf)
    return buf, ocispec.Descriptor{
        Digest: dgst,
```

```
        Size:   int64(size),
    }, nil
}

func (zs *ZtocSerializer) Deserialize(serializedZtoc io.Reader) (*Ztoc, error) {
    flatbuf, err := io.ReadAll(reader)
    if err != nil {
        return nil, err
    }

    return flatbufToZtoc(flatbuf), nil
}
```

The above implementation references the existing `ztocToFlatbuffer` and `flatbufToZtoc` functions, which will be reorganized to fit in ZtocSerializer.

The `Serialize` method will be called from `buildSociLayer` (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/layer/layer.go#L293):

```
...
zs := ZtocSerializer{}
ztocReader, ztocDesc, err := zs.Serialize(ztoc)
...
```

The `Deserialize` method will be called from current fs.layer.Resolve after obtaining the `ztocReader` (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/layer/layer.go#L293):

```
...
zs := ZtocSerializer{}
ztoc, err := zs.Deserialize(ztocReader)
...
```

FILE ACCESS - SNAPSHOTTER'S STORY

FUSE calls fs.node.Open (https://github.com/awslabs/soci-snapshotter/blob/main/fs/layer/node.go#L330), which will return the file handler as the `file` implementation in node.go, containing node and ReaderAt (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/layer/node.go#L413).
fs.node.Open calls fs.reader.OpenFile (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/reader/reader.go#L170), calling metadata reader's OpenFile (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/metadata/db/reader.go#L506), collecting the uncompressed offset along with the file size (from metadata db).
Once the read operation comes in, it lands at node.Read, which calls node.reader.ReadAt (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/reader/reader.go#L213).
Within the `reader` type (https://github.com/awslabs/soci-snapshotter/blob/cef680e7168aed00f629bcced76397d233f35e92/fs/reader/reader.go#L213) we'll keep the SpanManager as it is right now. SpanManager will know only about spans and will keep the cache for the uncompressed span data. The

BlobReader (which implements io.ReaderAt) will be providing with the compressed span content as well as serve as cache for compressed spans.

To separate compression from SpanManager, we propose to introduce the Extractor interface:

```
// Extractor extracts using the low level implementation for dealing with
// compression checkpoints.
// It implements the compression algorithm, so there will be GzipExtractor,
// ZstdExtractor, etc.
type Extractor interface {
    // Extract returns io.Reader containing uncompressed data
    // given compressed stream, uncompressedSize and uncompressedOffset.
    Extract(compressedStream io.Reader, uncompressedSize, uncompressedOffset Offset)
}
```

compression.GzipZinfo, compression.ZstdZinfo, etc will be implementing compression.Extractor, which will provide with low-level implementation to extract data, depending on the compression algorithm:

```
func (zi *Zinfo) Extractor() (Extractor, err) {
    return zi.gzipZinfo, nil
}
```

The adjusted SpanManager struct will look as follows:

```
// SpanManager does caching of uncompressed spans only.
// SpanManager does not know about compression algorithm, since this is handled by
// Extractor.
type SpanManager struct {
    extractor   compression.Extractor
    blobReader  io.ReaderAt
    // other members
}
```

With the above implementation, `fetchAndCacheSpan` on span manager will no longer cache compressed spans. It will be caching uncompressed spans only.
Within `fetchAncCacheSpan` there will be a call to Extract:

```
func (m *SpanManager) fetchAndCacheSpan(spanID soci.SpanID) (io.SectionReader, error)
    s := m.spans[spanID]
    ...
    compressedReader, err := m.fetchSpan(spanID)
    ...

    // get uncompressedOffset and uncompressed size from span
    uncompSize := s.endUncompOffset - s.startUncompOffset
    uncompOffset := s.startUncompOffet

    uncompSpan, err := m.extractor.Extract(compressedBuf, uncompSize, uncompOffset)
    if err != nil {
        return nil, err
    }
```

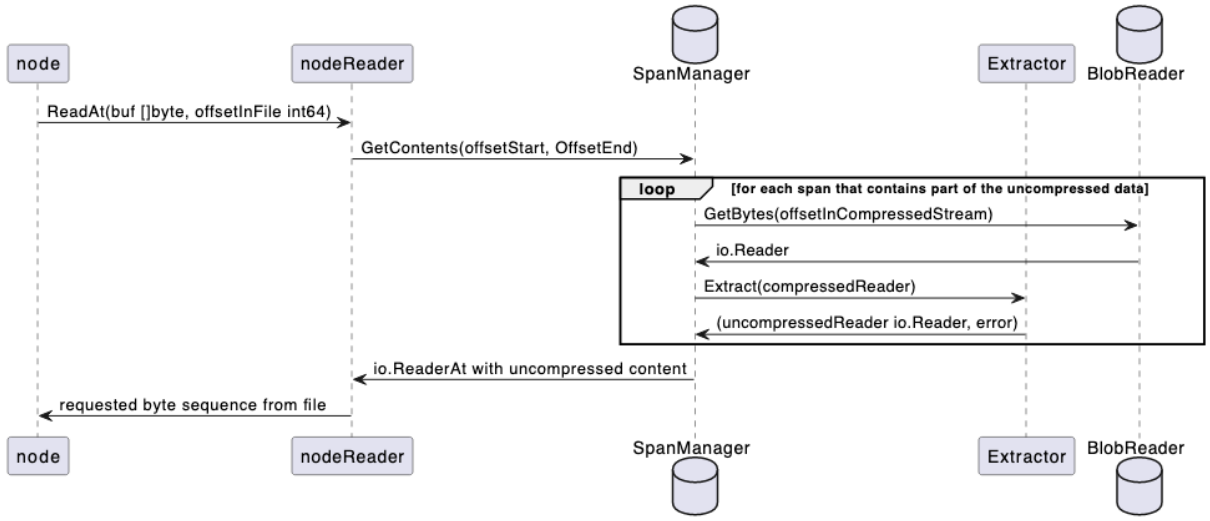
```
    ...
    return uncompSpan, nil
}
```

Span manager's fetchAndCacheSpan is taking care of two things:

- checking the cache for uncompressed span content
- calling fetchSpan, which returns the compressed reader
- decompressing and caching the uncompressed span

SpanManager's fetchSpan will be using the underlying BlobReader to fetch the span.
Instead of doing decompression using gzip_zinfo, SpanManager will be using Extractor.Extract to do the job. This will allow to decouple compression algorithm implementation from span manager.

The diagram below summarizes the data flow during file read (the database sign refers to the caching layers):



It's worth mentioning that the SpanManager→BlobReader→Extractor flow is a loop for each span that contains part of the uncompressed data.

FILE ACCESS - ZTOC.EXTRACTFILE STORY

The interface for Ztoc.ExtractFile remains the same. Accessing the file, will require to use the SpanManager to extract the data (no caching on SpanManager in this case):

```
func (z* Ztoc) ExtractFile(r *io.SectionReader, config *FileExtractConfig) ([]byte, err
    sm, err := spanmanager.New(z, r)
    if err != nil {
        return nil, err
    }
    start := config.UncompressedOffset
    end := config.UncompressedOffset + config.UncompressedSize
    contentReader, err := sm.GetContents(start, end)
    if err != nil {
```

```
        return nil, err
    }
    _, err := contentReader.Read(bytes)
    if err != nil {
        return nil, err
    }
    return bytes, nil
}
```

The SpanManager's behavior is exactly the same as in snapshotter's story.

Action items

1. **[Done]** Extractor.Extract how will it know which checkpoint to use?
2. In theory a single image may have gzip & zstd layers intermixed. How does this design solve it?
3. **[Done]** Add a block diagram showing the components and their dependencies at the Problem Statement section (don't need a full UML diagram, just something to give us a roadmap for the rest of the document, before we set out on the journey. Or maybe a one line description of the major interfaces)
4. **[Done]** Put the stories upfront and describe the data flow at the function call
5. **[Done]** Try to write it as a paragraph, not as numbered list
6. Example of Ztoc and other structs we are calling
    a. Put in the appendix
7. Separate Ztoc from anything related to containers, layers or soci related
8. **[Done - only span manager deals with spans]** In the ztoc spans are not modeled, in the extraction we think in terms of spans
9. **[Done]** Documentation
    a. Provide more context on the state stack in the entities I'm referring to
10. **[Done]** Try to find some language between checkpoint and span to make it easier to understand
    a. Maybe call it SpanDecompressor?
    b. Maybe call BuildCompressionContext
11. **[Done - deferred]** Implement fs interface. FS.Open
12. [**Done - see snapshotter's file access story for more details]** Layers:
    a. Files - FileExtractor
    b. Spans - SpanExtractor
    c. Checkpoints - Decompressor extract arbitrary data from checkpoints
13. **[Done - see snapshotter's file access story for more details]** SpanManager will be the middle layer for 7?
    a. Move download data - separate layer from SpanManager into section reader
14. **[Done - see snapshotter's file access story for more details]** FS asks open file ztoc. Ztoc is created from the section reader from the compressed stream. It can create a span mgr w/ the appropriate uncompressor and then span mgr can do the work of combining those
15. **[Done]** Or extractor is given the io stream. It takes offsets you are going to extract. Span mgr has knowledge of extractor
16. Section readers everywhere. Do they need to be abstracted? Maybe we can be constructing