Implementing GC for Irmin

Date: 2022-01-07

This note describes a general scenario that might be relevant for garbage collection with irmin-pack.

Scenario

We have a file that contains (the serialized representation of) objects. Example file contents:

```
File contents: [obj_1][obj_2]...
File offsets : ^off_1 ^off_2
```

So, at the beginning of the file (offset off_1, which is actually 0) we have the bytes to represent object 1. Following, at off_2 we have object 2, and so on.

Given an offset corresponding to the start of an object, we can get the length of the object in the file (either because the length is recorded explicitly, or because the object has a tag to indicate its kind which is of fixed length, or some other way), and use the corresponding data to reconstruct the object. So, obj_i has length len_i etc.

Things are more complicated than this, because some objects may refer to (earlier) objects. For example obj_2 may involve a pointer to obj_1. In this case, the marshalled version of obj_2 contains the offset off_1 (this is how we represent pointers on disk, by using the offset of the corresponding object; we call this an "on-disk pointer" or an "offset pointer"). This usually has to be taken into account when unmarshalling: first we must reconstruct obj_1, then obj_2 making sure that the in-memory pointer from obj_2 points to the in-memory obj_1.

NOTE that objects can only refer to earlier objects... it is forbidden for an object to refer to an object whose offset is later in the file (so, generic cyclic structures cannot be represented in this way, only DAG-type structures). Moreover, on-disk pointers (i.e. offsets) are wellformed in that any such pointer is to the start of an object in the file (not, for example, an offset pointing to the middle of an object in the file).These properties combined are referred to as "(on-disk) pointer wellformedness".

The file becomes larger and larger. As time goes on some of the objects become "dead" or "unreachable" or "garbage" (to be defined shortly), and so we want to somehow reclaim the space that they take up on disk.

When does an object become garbage? There is a notion of "commit". A commit is an object itself, which somehow represents the "current state" of the system. The objects in the store belong to particular commits. In a simplified scenario, there is exactly one commit active at any point in time.

So, the file contents might look like:

```
File contents: [obj_1][obj_2]...[commit_1][obj_21][obj_22]...[commit_2][obj_31][obj_32]
```

Where $commit_1$ is an object that may have pointers to any of the preceding objects. Similarly with $commit_2$. In fact, $commit_2$ always has a pointer to the preceding $commit_1$ (this is the "parent commit"), usually has pointers to obj_{21} and obj_{22}, and may also have references to earlier objects such as obj_1. NOTE that in this example, there will be a $commit_3$ that is in the process of being constructed, presumably with pointers to obj_{31} and obj_{32}. NOTE also that we can talk about objects which occur earlier

(in the file) than a given commit (or, more generally, than some other object in the file... but this concept doesn't really make sense in general because the relative ordering of two adjacent objects is meaningless).

A particular commit object, such as $commit_2$ defines "the state of the system" (some system of interest) at a particular time. From the $commit_2$ object, we can reach potentially all of the earlier objects. In practice, only some of these objects will be reachable from $commit_2$. Moreover, if an object is reachable from $commit_3$ but occurs earlier than $commit_2$, then it MUST be reachable from $commit_2$. This is because the state of $commit_3$ is actually derived from $commit_2$, by following pointers from $commit_2$ and by adding new objects after $commit_2$. Put another way, if an object is NOT reachable from $commit_2$ it is definitely NOT reachable in any later commit. We call this property "garbage monotonicity" - the garbage (the set of unreachable objects) keeps growing as we keep adding new commits.

So, once a commit object is on disk, the earlier objects reachable from the commit are "reachable", "live", "non-garbage" and the other earlier objects are "unreachable", "dead", "garbage".

So, the file contents might look like:

```
File contents: [obj_1][obj_2][commit_1][obj_21][obj_22][commit_2][obj_31][obj_32]...
Live/dead?   :  D      L               L       D                NA      NA
```

Here the live/dead status is with respect to $commit_2$. In this example, we have two objects that are dead wrt. $commit_2$: obj_1 and obj_{22}. All the other objects are alive (with the exception of $obj_{31}, obj_{32}, \ldots$, which are neither dead nor alive wrt. $commit_2$, but will likely be live for $commit_3$).

NOTE (commit parent references should be ignored): A commit typically references its parent commit via the hash of that commit. For the sake of garbage collection these hash references do not count as proper inter-object on-disk pointers. Thus, hash references from a commit to any parents should be ignored when computing object liveness.

The problem

Finally we get to the heart of our problem: obj_1, obj_{22} are not reachable from $commit_2$ or any later commit. We know that we will never need to access these objects again, so we want to reclaim the space on disk. The question is: how to do it?

Aside: Sparse files

Many Linux filesystems support the notion of a "sparse" file. This is a file which may be huge in size, but where many bytes have not yet been written. In this case, parts of the file which have not been written do not take up space on disk. Instead, the filesystem simply records that those regions have not been written, and when attempting to read from such a region, the filesystem will simply return zero-byte sequences. In this way, a huge file (say,100GB) might take up almost no actual disk space if it is completely empty.

Now, normal Linux filesystems will implement sparse files at the block level, that is, only unused blocks within the file will not use disk space. However, it is not hard to implement general sparse files at the application level. We discuss implementations of sparse files below.

Back to the problem: A solution

One obvious way to deal with this problem is to simulate the original file using two other files: a "sparse" file and a "suffix" file. The sparse file will contain $commit_2$ and all earlier live objects (live wrt. $commit_2$). The suffix file will contain obj_{31}, obj_{32} and any later objects. [NOTE it does not really matter exactly how the objects are split into the sparse and suffix files; this particular choice makes a sparse file self-contained for the given commit it corresponds to, which might be useful.]

So, the file contents might now look like:

```
Sparse file: [000][obj_2][commit_1][obj_21][000][commit_2]
Suffix file: [obj_31][obj_32]...
```

Here we have used `[000]` to represent that the region of the file is "sparse" - that is, it has not been written and it takes up no disk space. The key point is that the sparse file only uses disk space to record the live objects - the rest are simply gaps in the file which are notionally filled with 0s.

For the suffix file, we have to remember that obj_{31}, which is supposed to be stored at off_{31} is now actually stored at offset 0 in the suffix file. So, when accessing offsets less than off_{31} we go to the sparse file, otherwise we subtract off_{31} to get the offset to use in the suffix file. The result is that the combination of sparse file and suffix file mimic the original, but without using disk space for the dead objects.

What happens when the suffix file gets large?

Clearly we can save some space by splitting the file at a given commit. But what happens when the suffix file gets large? Well, the combination of sparse-file and suffix-file are simply a simulation of the original file. If we want to "sparsify" (i.e. collect and remove garbage) with respect to a later commit, we can follow exactly the same process. This involves creating a completely new sparse file, and suffix file, and just repeating the steps from before.

An alternative requires that the sparse file support a notion of "deleting" data from within it. For example, suppose that we commit $commit_3$ and our files look as follows:

```
Sparse file: [000][obj_2][commit_1][obj_21][000][commit_2]
Suffix file: [obj_31][obj_32][commit_3]
```

Suppose we want to get rid of objects that are dead at $commit_3$. Now, with $commit_3$ we may find that obj_2 becomes dead. In this case, if our sparse file supports a delete operation, we can simply delete obj_2 at off_2, to get the new sparse file:

```
Sparse file: [000][000][commit_1][obj_21][000][commit_2]
```

Which we then extend with the additional live objects from $commit_3$. The advantage of this, compared to naively recreating the sparse file from scratch, is that we only have to delete all the **newly-dead** objects, rather than copy all the live objects at $commit_3$. It is expected that the newly dead objects are much less in number, compared to all live objects.

Performance and concurrency considerations

Suppose we write $commit_2$ to the main file, then decide we want to GC wrt. that commit. At this point, we can start writing to a new suffix file immediately - all new objects get written to the suffix file. We keep the main file around to provide the functionality of the sparse file, while we **asynchronously** create the sparse file (in a different process, say). When the sparse file is complete, we notify the application that it should switch from the main file to the sparse file, and delete the old main file which is now no longer required. Deleting the old main file is the step that frees up disk space.

Because the sparse file is created asynchronously in a different process, the main process is relatively unaffected (there will be extra disk IO of course, in order to create the sparse file).

Applicability to current irmin-pack codebase

This is where the story becomes less clear. To what extent does the description above match with what Irmin currently does? Is the Irmin pack file structured as a sequence of objects? Can we extract the length of an object at a given position? How does this work with the Index and the Index log files? What about the Dictionary?

Two or more active head commits: My understanding is that for short periods of time the blockchain can fork, so that there are effectively two (or more) notions of the "current" state. One simple approach to this situation is to forbid "sparsifying" in such a situation - instead, we can sparsify on the most recent ancestor commit which represented a single state of the system, or just wait for the situation to resolve into a single head commit.

(Section to be completed)

- Index, index logs
- Object encoding/decoding
- Dictionary

Appendix

Implementing sparse files

Implementation without delete: One implementation is just to store the live objects adjacent in a file.

```
Sparse file "specification": [000][obj_2][commit_1][obj_21][000][commit_2]
Sparse file implementation : [obj_2][commit_1][obj_21][commit_2]
```

But now obj_2 is no longer stored at off_2, and in general we have to take into account the on-disk pointers!

For example, offset off_2 for obj_2 should be interpreted as offset 0 in the sparse file implementation (because that is where obj_2 is in the sparse file implementation). In general, we need to map the offsets of live objects in the original file to offsets of the same objects in the sparse file. We call this the "sparse offset map", and we can store it in another file.

```
Sparse file           : [obj_2][commit_1][obj_21][commit_2]
Offset in sparse file : ^0     ^n1       ^n2     ^n3
Sparse offset map     : (off_2->0),(off_commit_1->n1),(off_21->n2),(off_commit_2->n3)
```

In this way we can associate on-disk pointers in the original file to offsets in the sparse file (but only for the live objects in the sparse file of course).

PENDING How to store the sparse offset map? It probably depends on how large the map is, which is proportional to the number of objects in the sparse file, which is the number of live objects wrt. a given commit. Hopefully it can be stored easily in memory and simply written to disk as a sequence of int pairs. Otherwise there are various other options available.

This implementation does not naturally support the "delete" operation discussed above, which would allow to delete newly-dead objects from an existing sparse file.

Implementation with delete: A simple implementation that supports delete is to implement a sparse file as a directory (say), where an object obj stored at position off could be represented by a file in the directory, with name off, and with contents the bytes representing the object. This efficiently supports the delete operation for an object at a given offset. Alternative implementations might use a database such as LMDB.

Craig's queries

- *Under "What happens when the suffix file gets large?" there is some discussion of a deletion strategy for newly-dead objects. Do you know of an efficient way to detect such objects? To me this seems like the main complication with a "reclaim" GC strategy.*

This is a good question. Let's say that the last GC (sparsification!) took place for $commit_i$ and now we are at $commit_j$ and want to perform another GC. We could traverse all objects from $commit_j$ to determine the live objects. We already know the live objects for $commit_i$ (let's call these $Live(i)$; the sparse file knows the offsets of these objects, or alternatively we can explicitly record them when performing GC for $commit_i$). Then the newly dead objects are just $Live(j) - Live(i)$, where the $(-)$ operation is set difference. This scheme requires a traversal of all live objects from a given commit whenever we do a GC. Is it possible to do better? FIXME think about this

- *The description of a sparse offset map strikes me as very similar to our index, but with virtual file offsets as virtual addresses rather than hashes. For live objects it may be fine to keep this page table in memory for now (we can probably play clever hashtable tricks to get it down to 1 or 2 words per live entry), but this may not last for long. Last time I checked, there were around 30 million objects in a snapshot (& I think this probably means a hashtable of size ~1G).*

The query concerns the implementation of the sparse offset map. For 30 million live objects, we need at least 2 ints per object (the key -- which is the position in the original file -- and the value which is the offset in the sparse file). Thus, $30M * 2 * 8B = 480MB$ lower bound just to store the offset map. Not to mention that recreating this sparse file every time will be extremely costly. NOTE that $30M$ objects, and an on-disk snapshot size of $6GB$ gives a per-object size of 200 bytes, which sounds plausible.

So, if we were to implement a sparse file using the sparse-offset-map, we might want to keep it mostly on-disk.

This is the problem with sparse files: the offset map may be huge, whereas if we simply rewrote all the old offsets with the new offsets in the sparse file, we wouldn't need any extra space. FIXME more thinking

- *Can this framing of the problem be extended to support the "performant archive stores" aspect discussed on Friday? I guess we'd need some sort of non-sparse file to sit beneath the sparse one as a fallback? (the equivalent of a "lower" store in the layered terminology)*

Probably not immediately, since this is really about GC, not necessarily about trying to make recently accessed objects really quick to access. Craig proposes to make a sparse store of all the live objects at a given commit, and then also have another store of all the non-live objects at that commit. Then, the live objects will likely be quickly accessable, and otherwise we have to go back to the huge archive of non-live objects. So, we have two sparse files, one for the live objects and one for the dead. And every time we perform a GC we might push objects from live to dead. This is most efficient when the "sparse-file-for-live-objects" contains a "delete" operation as discussed above.

- *I believe "garbage monotonicity" doesn't hold in the presence of an index. We can drop a reference to an earlier indexed object in one commit, then later recover it via the index when exporting. I don't see an invocation of this property later in the document, but this caveat seems fine under the proposed sparse file scheme: we'd just end up assigning a second virtual address to this object after collecting the first.*

I guess I was ignoring the index for the purposes of liveness, since (a bit like a commit having a reference to the parent commit) the index potentially allows every object to be "live" (that is, reachable...via the index). If we only index commits, then I think we can ignore the index. FIXME not sure of the last bit "we'd just end up..." - I interpret this to mean that we try to look up an object in the index; the index has an offset for the object; BUT the sparse file does not know about that offset (because the object has been GC'ed at that offset); in this case the index should return that the object is not indexed, rather than returning the (no longer valid) old offset. Probably this causes the application to write a new copy of the object in question.