Irmin-pack layered-store/GC design document

Date: 2022-02-03 ff.

Introduction

This is a short document to describe the design of a garbage collection mechanism for irmin-pack (the variant of Irmin currently used by Tezos).

TL;DR

The irmin pack file stores objects. References from one object to another are stored using the position of the referred-to object in the pack file. Currently, objects are never deleted, so disk usage grows indefinitely; this is the problem we want to solve. In order to delete old objects and reclaim disk space, whilst still referencing objects by their "position in the pack file", we replace an initial segment of the pack file -- up to a particular "garbage collection root commit" -- with an "object store", which essentially stores the live objects from the initial segment, indexed by their original position in the pack file. Every so often, we choose a new GC root commit and reconstruct the object store from scratch. In this way, disk usage is required only for recent objects and those reachable from the last GC root commit.

Background

The Tezos blockchain uses irmin-pack as the main storage component. Roughly speaking, irmin-pack is a git-like object store, optimized for the Tezos usecase. Irmin-pack has several components (including an "index", and a "dictionary"), but we focus on the main file used to store object data: the store pack file.

Pack file: The pack file stores marshalled objects one after the other. An object can contain pointers to earlier (but not later!) objects in the file. Pointers to an earlier object are typically represented by *the offset (position) of the earlier object in the pack file.* The pack file is append-only: existing objects are never updated.

```
Pack file as a sequence of objects: [obj][obj][obj]...
```

Aside: Previously, every object was identified by hash, and there was another structure -- the index -- that translated hashes to offsets; however, this indirection was inefficient compared with directly addressing objects by their offset. Now, most objects are referenced using the offset in the pack file.

Commit objects: Some of the objects in the pack file are "commit objects"; a commit, together with the objects reachable from that commit, represents "the state of the tezos node at a point in time". (It is possible for the state to split in two temporarily, but we will ignore this for the time being.) The Tezos node only needs the "latest" commits in order to process new blocks. Thus, any objects not reachable from the latest commits can be considered "garbage".

Archive nodes and rolling nodes: There are different types of Tezos node. An "archive node" stores the complete history of the blockchain from the genesis block. Currently this is over 2 million blocks. Roughly speaking, a block corresponds to a commit. A "rolling node" stores only the last n blocks (where n is chosen to keep the total disk usage within some bound - n may be as small as 5 or even less, or as large as 40,000 or more). Another type of node is the "full node", which is somewhere between an archive node and a rolling node.

Rolling nodes, disk space usage: The purpose of the rolling node is to keep resource usage, particularly of disk space, bounded by only storing the last n blocks. However, the current implementation does not achieve this aim: as rolling nodes execute, the pack file grows larger and larger, and no old data is discarded. To get around this problem, node owners periodically "snapshot export" the current state of the blockchain from the node, delete the old data, and then "snapshot import" the state back again, in a form of manual garbage collection.

Problem summary: The main problem we want to avoid is Tezos users having to periodically export and import the blockchain state in order to keep the disk usage of the Tezos node bounded. Instead, we want to perform garbage collection of unreachable objects automatically: Periodically, a commit should be chosen as the GC root, and objects constructed before the commit that are not reachable from the commit should be considered garbage, removed from the pack store, and the disk space reclaimed. The problem is that with the current implementation of the pack file, which is just an ordinary file, it is not possible to "delete" regions corresponding to dead objects, and "reclaim" the space.

Proposed solution

Consider the following pack file, where the commit object has been selected as the GC root:

```
Pack file: [obj1][obj2]...[obj3][obj4][obj5][commit]...
```

Objects that precede the commit are either reachable from the commit (by following object references from the commit), or not. For the unreachable objects, we want to reclaim the disk space. For reachable objects, we need to be able to continue to access them via their "offset in the pack file".

The straightforward solution is to implement the pack file using two other data-structures:

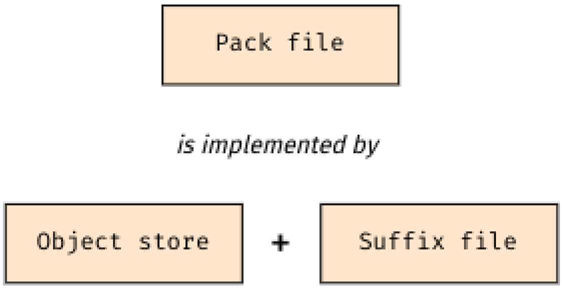
- The "suffix file" contains the commit object, and all bytes following in the pack file.
- The "object store" contains all the objects reachable from the commit, indexed by their offset. (Recall: these objects all appear earlier in the pack file than the commit object.)

```
Pack file         : [obj1][obj2]...[obj3][obj4][obj5][commit]...
Object reachable? :  Y     N        Y     N     Y

  ...is implemented by...

Object store      : a map for reachable objs: off1->obj1, ... off3->obj3, off5->obj5
                                              where off1 is the offset of obj1 etc.
  and
Suffix file       :                                  [commit]...
```

Attempting to read from the pack file is then simulated in the obvious way: if the offset is for the commit, or later, we read from the suffix file, and otherwise we lookup the offset in the object store and return the appropriate object. It is assumed that we only ever access the reachable objects in the object store, and that we do so via their offset.



We replace the irmin pack file with these two datastructures. Every time we perform garbage collection from a given commit, we create the next versions of the object store and suffix file, and switch from the current version to the next version, deleting the old suffix file and object store to reclaim disk space. Creating the next versions of the object store and suffix file is potentially expensive, so we implement these steps in a separate process with minimal impact on the running Tezos node.

Aside: Following git, a commit will typically reference its parent commit, which will then reference *its* parent, and so on. Clearly, if we used these references to calculate object reachability, all objects would remain reachable forever. This is not what we want, so when calculating the set of reachable objects for a given commit, we ignore the references from a commit to its parent commit.

Object store

The object store is a persistent datastructure that implements a map from key (offset in pack file) to data (the marshalled bytes representing an object). In our scenario, the worker creates the object store, which is then read-only for the main process: objects are never mutated, or deleted from the object store. In this setting, a very simple implementation of an object store suffices: we store live objects in a *data* file, and maintain a persistent (int → int) map from "offset in the original pack file" to "offset in the object store data file".

Terminology: We introduce the term "virtual offset" for "offset in the original pack file", and the term "real offset" for "offset in the object store data file". Thus, the object store map is from virtual offset to real offset.

Example: Consider the following, where the pack file contains reachable objects o_1, o_3, o_5, (with virtual offsets $v_1, \ldots$):

```
Pack store     : ...[o1]...[o3]...[o5]...
Virtual offset :    ^v1    ^v3    ^v5

Object store data file : [o3][o5][o1]
Real offset            : ^r3 ^r5 ^r1
```

The object store data file contains the same objects, but with different "real" offsets r_1, r_3, r_5.

The persistent map needs to contain an entry $v_1 \to r_1$ (and similarly for the other objects) to relate the virtual offset in the pack file with the real offset in the data file.

To read from "virtual offset v_3" (say), we use the map to retrieve the real offset r_3 in the object store data file, and then read the object data from that position.

Asynchronous implementation

Garbage collection is performed periodically. We want each round of garbage collection to take place asynchronously, with minimal impact on the main Tezos node. For this reason, when a commit is chosen as the GC root, we fork a worker process to construct the next object-store and next suffix-file. When the worker terminates, the main process "handles worker termination": it switches from the current object-store+suffix-file to the next, and continues operation. This switch takes place almost instantaneously. The hard work is done in the worker process.

Read-only Tezos nodes: In addition to the main Tezos read/write node that accesses the pack store, there are a number of read-only nodes, which also access the pack store (and other irmin data files) in read-only mode. It is important that these are synchronised when the switch is made from the current object-store+suffix to the next object-store+suffix. This synchronisation makes use of a "control file".

Control file: In order to coordinate between the main process and the worker process (and also any read-only processes), we introduce a control file. The control file is a single file which includes the following fields:

- generation: an integer which is bumped every time a switch occurs; processes can detect that a switch has occurred by observing changes in the generation number
- object-store: a pointer to the object store (actually, the name of the directory containing the object store)
- suffix-file: a pointer to the suffix file (actually, the name of the directory containing the suffix file)
- other metadata (not relevant for this document)

The current control file has filename `control`. In order to switch state from one object-store+suffix to the next, we can create a new control file and atomically rename it over the old control file.

Steps taken by the worker: Given a GC root commit, the worker proceeds as follows:

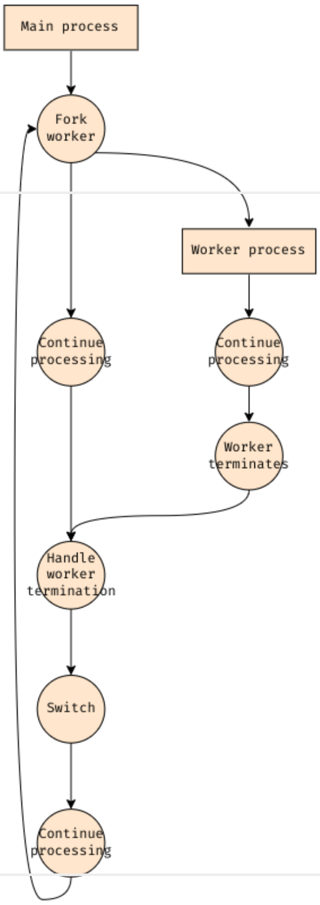
- Calculate the *reachable* objects from the commit, recording their offset and length information.
- Construct the new object store, a map from offset to object (really, the marshalled bytes representing the object).
- Construct the new suffix file, by copying the contents of the current suffix file from the GC root commit onwards.
- Finally, construct the next control file `control.nnnn` where `nnnn` is the *next* generation number, and terminate.

Aside: For efficiency, some of these steps may be combined. For example, the worker could construct the object store at the same time that it calculates the reachable objects, thereby avoiding having to visit locations in the file twice.

Steps taken by the main process on worker termination: When the main process detects the worker has terminated, it does the following:

- Read the next control file `control.nnnn` to find the next object-store and next suffix file.
- Copy any further data that has been added to the current suffix to the end of next suffix.

- Rename `control.nnnn` over `control`; read-only processes detect that the generation number has changed, and synchronise with the new state.
- Delete the old object-store and suffix file to reclaim disk space.



Worker efficiency concerns

We want the worker process to execute with minimal impact on the main Tezos node. For this reason, we take the following steps:

- The worker runs as a separate process. If needed, the process can be "nice"-ed to reduce its impact, for example, when running on a single core.
- The worker uses `fadvise` to inform the operating system that it should preferentially **not** devote resources to caching **worker** disk IO. The OS caches should instead remain allocated to the main Tezos node. In particular, the worker should not disturb the caches for the main process, when the worker calculates the reachable objects for the given commit.
- With the exception of the calculation of the reachable objects from a commit, the worker is **simply copying bytes between files.** This makes the worker very simple, and hopefully makes it easy to optimize performance and ensure correctness of operation.

Crash correctness

We now describe what happens in the event of a system crash. The interesting cases are when the worker is active, or has just terminated.

If the worker process is active, then it may have created part or all of the next object-store and next suffix. However, the control file will still point to the current object store and suffix. On restart, the main process opens the control file, at which point it knows which files are the "real" object-store and suffix (those listed in the control file), and can delete any other files it finds (which will be the partial files the worker was creating).

If the worker process has just terminated and a crash occurs, then the critical point is whether the rename of the next `control.nnnn` over the current `control` was recorded persistently on disk or not. If it was, then the main process restarts, reads `control` (which was successfully renamed from `control.nnnn`), and starts operating with the next object-store and suffix file (and deletes the old versions). If not, the main process reads the old `control` file (the rename of `control.nnnn` was not successful), and starts with the old object-store and suffix file (and deletes the new ones that the worker created). In either case, the system restarts in a correct state.

This correctness argument uses the "atomic rename" property of filesystems: when renaming a file within a directory, and a crash occurs, on restart the file is found either under the old name or under the new name.

Summary

The above gives an overview of the design. Further detailed comments on particular aspects are included in the appendix below.

Appendix

The following sections provide further detail on particular aspects of the design. They can be skipped on first reading.

Scaling

It is worth noting that several computations (finding the set of reachable objects; constructing the object store) scale as "the number of reachable objects per commit". However, we might want to scale as "the number of actively changed objects per commit". In this case, a completely different design would be required (see the Alternative design section below).

Object store overhead

Object store overhead and the number of reachable objects: The object store consists of a data file, and a persistent map. The use of a persistent map introduces overhead compared to the space needed to store the raw objects. For each object, we need at least two integers (or 16 bytes) for each map entry. Thus, 1M reachable objects requires at least 16MB of overhead for the map, 10M requires 160MB and 100M requires 1.6GB.

A snapshot of the current Tezos state contains 30M objects, which corresponds to an overhead of 480MB for the map. This is significant, so we would like to reduce the map size if possible.

Extents: Consider adjacent reachable objects in the pack file between two offsets:

```
Pack file: ...[obj1][obj2][obj3]...
              ^off1 ^off2 ^off3
              \________________/
              A single "extent"
```

Instead of storing (virtual offset → real offset) entries for each object, we can consider adjacent objects in the pack file as an "extent" (an extent is just a region in a file between two offsets), and instead store (virtual offset → real offset) entries for each *extent*. It is still possible to locate individual objects via their virtual offset, but hopefully a map based on extents would have significantly fewer entries.

Thus, the proposal would be to work with a map from "virtual extent offset" to "real extent offset".

Summary: The prototype implementation of the object store will use a map with an entry for each object. For production, we will replace this implementation with one based on extents. The sequence of operations for the worker, when calculating reachablility using extents, is as follows:

Traverse reachable objects and store unsorted (off,len) information in a file (30M objects, approx 1GB)

↓

Sort on-disk object (off,len) data by offset (another 1GB file)

↓

Calculate (off,len) extents corresponding to adjacent objects and store on disk (size unknown)

Whether extents reduce the size of the map is currently unknown. The worst case is if each reachable object is separated from all others, so the number of extents is the same as the number of objects, and the map remains the same size. If there are many adjacent reachable objects (the expected case), then using extents should result in a much reduced size for the persistent map.

We hope the use of extents will make the persistent map sufficently small. If the extent-based map file is still "too large", we will then consider further ways to reduce its size.

Possible optimisations

We expect the above design will perform well. However, it is possible to make further efficiency improvements, at the cost of increased complexity in the implementation. We now describe some potential optimisations.

Reduce persistent map size further: The persistent map, from virtual extent offset to real offset, is far from arbitrary (it is monotonic, for example). This should make it easy to compress, if the size of the map is considered too large.

Reuse object ancestor information: When we select a commit as the GC root, we calculate the objects reachable from that commit. This requires that for each object, we (implicitly or explicitly) calculate the reachable ancestors of that object. When it comes to the next GC iteration, we select a later commit. However, if one of the objects reachable before, is still reachable, we know that all that object's ancestors are also reachable. Rather than repeatedly traverse those ancestors on each GC, we could remember the (offsets of) ancestors of each object, and reuse this information for the next GC cycle.

Prefer sequential object traversal: Traversing the object graph during GC involves accessing earlier objects, typically in "random access" fashion. We might want to be smarter about this, so that objects are visited (as far as possible) in order of increasing offset on disk. This takes advantage of the fact that sequential file access is often much quicker than accessing the file randomly.

Prefer sequential object traversal for new objects: Between two GC commits, the pack file contains new objects:

```
Pack file: ...[commit][obj][obj]...[obj][commit]...
                       \                /
                        This region may also contain non-GC commits
```

When calculating the reachable objects for the later commit, we already noted that we can reuse ancestor information from the earlier commit. In addition, when calculating reachability information, we can process the objects between the two commits in the order they appear in the pack file. With these two optimisations, we only ever access the pack file sequentially, from the previous GC commit to the next.

One possible downside is that by processing each of the objects between the two commits, we potentially do extra work when very few of these intervening objects are reachable (and so very few of these objects need to be touched when computing reachability from the later commit). However, we expect that the benefit to processing these objects in the order they appear in the file outweighs this possible downside.

Prototype

A prototype of the proposed design can be found at: https://github.com/tomjridge/sparse-file (the object store is referred to as a "sparse file" in that repository). The code for the worker, for example, looks like the following (modulo some renaming of functions):

```
let run_worker ~dir ~commit_offset =
  (* load the control, obj-store, suffix; calculate reachability from commit_offset and
     store objs in new obj-store; create new suffix file; create new control file; terminate *)
  let io = Io.open_ dir in
  let dreach = Dr.disk_calc_reachable ~io ~off:commit_offset in
  create_object_store ~io ~dreach;
  create_suffix_file ~io ~off:commit_offset;
  create_control_file ~io;
  log "worker: terminating";
  ()
```

This code matches exactly the informal description of the steps taken by the worker, described earlier.

Retaining older objects for archive nodes

Archive nodes contain the complete blockchain history, starting from the genesis block. This results in a huge store pack file, many times larger than main memory. Live objects are distributed throughout this huge file, which makes it difficult for OS caching to work effectively, and as a result, as the store becomes larger, the archive node becomes slower.

In previous versions of the layered store, the design also included a "lower" layer. For archive nodes, the lower layer contained all the objects before the most recent GC commit, whether they were reachable or not - the lower layer was effectively the full prefix of the pack file before the GC commit.

One possibility with the new design proposed above is to retain the lower layer, whilst still sticking with the object store + sparse file approach, and preferentially reading from the object store where possible. The advantage (compared with just keeping the full pack file) is that the object store is a dense store of live objects (much denser than the full pack file), which improves OS disk caching, thereby improving performance of snapshot export for recent commits. In addition, the OS can preferentially cache the object store, which improves general archive node performance compared with trying to cache the huge pack file, *providing* we mostly access objects in the object store+suffix rather than the lower layer. However, some uses of the archive node, such as responding to RPC requests concerning arbitrary blocks, would still use the lower layer, and no performance improvement can be expected in these cases.

The goal of improving archive node performance for arbitrary blocks (not just those corresponding to recent commits) still remains, and is not addressed by this proposal.

Alternative design

The design proposed above scales with "the number of objects reachable from a commit". If this grows increasingly large over time, the proposed design may become unsuitable. An alternative is to use a fully-fledged object store as the storage layer, and implement GC in that layer. This would avoid having to recreate the object-store+suffix file each time. This approach would scale as "the number of objects mutated on each commit".

The pack file stores objects, which reference each other via their position in the pack file. Clearly there is nothing particularly special about the position of an object in a pack file - it is just a way of uniquely identifying the objects.

In the design proposed above, when we build the object store, we construct a map from key (offset in pack file) to object, and we construct a new object store each time we perform garbage collection.

One alternative design would be to replace the pack file completely, and instead use a "real" object store (with no suffix file) to store objects. This "real" object store would be far more sophisticated than the object store outlined above. For example, the real object store should also support a "delete" operation, with asynchronous garbage collection, so we can delete unreachable objects and reclaim the disk space. This would provide a clean interface for the bottom-most storage layer, based on the notions of object and object id, rather than file offsets and byte strings we currently manipulate. Garbage collection could be implemented cleanly in this layer. With such an approach, and using the "ancestor optimisation" above, GC could scale as "the number of objects mutated on each commit" (although we do not spell out the details here).

The problem is that existing databases and object stores do not provide the performance needed for the Tezos usecase. This is partly the reason Tezos uses Irmin (the other main reason is that Irmin provides efficient tree-like operations). However, we have developed a prototype object store "kv-hash", as part of work on optimising another part of Irmin, that could potentially be used as an object store if this alternative design were pursued.