Verneuil:[1] incremental snapshot replication for S3

Backtrace wrote the Verneuil VFS to physically replicate *snapshots* of sqlite databases to blob stores like S3. We hope that it will let us easily improve the scale and availability of our servers without sacrificing the operational efficiency we enjoy with sqlite, especially for on-premises installations. Our design goals were as follows, in decreasing order of importance.

1. Preserve correct behaviour for the single-node case: regardless of what may happen to our replication logic or to the network, replacing the default Unix VFS with Verneuil should never introduce failures in sqlite operations. This requirement implies that replication must be asynchronous, and that even a long-lasting outage of the target blob store must not result in unbounded growth of replication data.
2. Snapshots should be valid: when we find a snapshot in the blob store, it must represent the state of the database file at some point in the past… even after unexpected operations like a direct update to the DB file with the default Unix VFS, or a differently-configured Verneuil VFS.
3. Readers should be able to query frequently updated snapshots. (Also the answer to "Why not litestream?" https://github.com/benbjohnson/litestream/issues/8)
4. Crash-safe multiprocessing should be possible. Sqlite supports multiple processes connecting to the same file, and processes are allowed to crash without causing corruption or losing committed transactions. Verneuil provides the same crash-safety guarantees, and should also behave well as long as all the processes are configured identically.
5. Hard bound on replication costs (for writers), reasonable costs for readers: we want to enable replication without thinking about it, even if we don't currently have readers.

It is not a goal to provide an audit log, nor to guarantee the ability to go back in time at regular intervals. In fact, there isn't even a hard bound on replication lag: remote storage should be updated within a few seconds of sqlite committing writes when everything is in a healthy state, but local write transactions can always complete normally, even during a network outage.

We do however guarantee something stronger than eventual consistency. When there is no write, the remote snapshot will eventually catch up to the local copy (no deadlock). More importantly, even under constant writes, the remote snapshots are constantly updated: while they may be late, they never get stuck (no livelock). This is the least important property (more like a quality issue than correctness), and, although replication lag should be on the order of seconds when everything is in a healthy state… snapshots *could* go back in time.[2]

Here's how it works.

[1] The Verneuil process was the first commercial method of manufacturing synthetic gemstones… and DRH insists on pronouncing sqlite like a mineral, surely a precious one (:

[2] Hard to avoid in the absence of conditional PUT. However, for a single host that uses only one spooling prefix, Verneuil uses file locks to avoid time-travelling snapshots.

The shape of replicated data in blob stores

We represent snapshots with protobuf-encoded “directory” blobs. The blobs describe metadata about the snapshot, and refer to the file’s contents with content-addressed chunks, where each chunk spans 64 KB, except for the last one, which may be short. For maximum efficiency, the sqlite database should be configured with 64 KB pages, but that is not mandatory.

The name of the directory blob for a given database file is deterministically derived from the hostname and the database’s path; 4-hex-character hash prefixes help spread out the load and make it easy to paginate over the data in parallel. Directory blobs can be configured to hit a dedicated bucket; we expect that bucket to be versioned.

The chunks are described with a 128-bit umash fingerprint of their contents, and their name is deterministically constructed from that fingerprint. Chunk blobs can be configured to hit a dedicated bucket; we expect that bucket to *not* be versioned, and subject to TTL for garbage collection (a background process will “touch” live chunks). Content-addressing avoids coordination when uploading chunk data to blobs (any collision is benign). Random names would offer the same property, but content-addressing also naturally exposes redundancy across snapshots. This lets us bound the chunk data footprint with the actual rate of writes to the replicated database, independently of the frequency at which we might have to rebuild a snapshot from scratch (e.g., after application or OS crashes). It also lets us share storage for data that is identical across multiple databases: configuration databases often start as copies of the same initial state.

The chunk size is hardcoded at 64 KB, but can be changed for testing. It’s fixed because we would like larger chunks (blob stores are usually more efficient for object sizes on the order of a few megabytes), but 64 KB is the maximum page size that sqlite supports. Larger chunk sizes would thus result in write amplification, and that’s not ideal given our goal of minimising cost for log writers (letting read replicas pay-as-they-go makes sense). Later, we might want to make this configurable.

In order to reconstruct a snapshot, a reader will thus fetch a directory proto for the file they want to copy (e.g., by reading the most recent version of the directory blob), read the array of fingerprints in the protobuf, fetch the corresponding blobs from the chunks bucket, and concatenate the result.

This representation is reasonably efficient in the common case because B-trees have a high fanout factor (especially with 64 KB pages), so a few point writes will only affect a couple pages. Readers should thus be able to cache most chunks when constructing snapshots separated by a few small write transactions. Moreover, the worst case is bounded by the database size. There is no need to tune a “snapshot frequency:” we only produce snapshots, and their format that naturally exposes redundancy.

Writers could naively compute snapshots from scratch, and rely on content addressing and a cache of recently uploaded chunks to minimise API costs. However, in the common case, writers can start from a local snapshot file that’s synchronised with the database file before the write transaction, remember the set of dirty chunks (where sqlite wrote in the database file), and update the snapshot after committing to the local file.

Writers always release the write lock before updating the snapshot, so concurrent sqlite readers aren’t blocked by the additional work introduced by replication (obviously, writers are slowed down, but that’s hopefully not unexpected). The protocol to update snapshots allows any number of threads or processes to make progress together, without any synchronisation except for a read lock on the sqlite database file. The wait-free nature of the protocol maintains sqlite’s crash-safety properties.

Writers do not directly update the blob store. They instead maintain a local spooling directory, and a separate copier asynchronously copies spooled data to remote blob stores.

The spooling directory

Each sqlite database file has a dedicated snapshot spooling subdirectory, within the configured spooling path for the Verneuil VFS. We do not leave the spooling subdirectory next to the database file because writes to the spooling subdirectory are merely crash-atomic with respect to userspace processes; we do not fsync, etc., and instead make sure that we never read data left behind after an OS crash or a reboot.[3] We do so by working in a subdirectory tagged with the boot id (/proc/sys/kernel/random/boot_id) and boot time (approximated by stating /proc/1). This also means that, while the database file may live on reliable persistent storage (e.g., EBS), we can spool snapshots on faster / cheaper local disks.

Inside each database file's spooling directory, writers maintain two subdirectories: the "ready" subdirectory is only replaced once its contents are copied to the remote blob store, while the "staging" subdirectory is always updated to match the database file after a write transaction[4] and before releasing the read lock (which may let another write transaction commit).

Both the "ready" and the "staging" subdirectories contain a pair of subsubdirectories: "{ready,staging}/chunks" contains content-addressed chunk blobs, and "{ready,staging}/meta" contains named directory blobs that refer to the chunks in their sibling directory. Each file in "chunks" is named after the corresponding blob, and similarly for "meta". Copiers thus simply have to read each file, and upload the file's contents to a blob with the same name as that file (slashes and other special characters are url-encoded).

Writers always keep "staging" up to date after a sqlite write transaction, and update "ready" whenever it is empty or missing. We recover from crashes between a write transaction and the corresponding update to "staging" by tagging sqlite files with version ids, and tracking the current tag in snapshots: whenever the latest snapshot does not match the current tag for the sqlite db file, we know the snapshot is out of sync and must not be trusted.

Copiers can always consume the "ready" subdirectory and upload its contents to the blob store when it exists, regardless of churn in the "staging" subdirectory (no livelock). However, the "ready" subdirectory may not be updated with fresh contents after copiers have consumed it, if no one writes to the sqlite database for a while. Copiers will thus optimistically consume from the "staging" subdirectory with a sequence-lock-style approach that always succeeds when no one writes to the sqlite database (no deadlock). The two subdirectories provide complementary progress guarantees, without any synchronisation between copiers and writers, except for atomic POSIX filesystem operations. We will see that the result is wait-free for copiers and writers, and inherently crash-safe.

---

[3] It would make sense to seed the local state with a copy of the most recent directory blob, but our databases are small enough and our hosts hopefully stable enough that we can afford to resynchronise everything after a reboot or crash.

[4] Writers can crash after committing the transaction to the local sqlite file, but before completing the snapshot to the spooling directory. Verneuil detects that case by comparing version tags between the local sqlite db and the most recent staged directory file.

Files in the ready directory are only deleted once uploaded to remote storage. The “ready” directory contains two subdirectories “ready/chunks” for content-addressed chunk blobs, and “ready/meta” for the named directory blob.

Copiers acquire a consistent pointer to the directory by opening the “ready” directory with O_PATH and accessing its contents through /proc/self/fd/: this guarantees that a copier will always consume from the same ready directory, and not from an old ready directory and later a newer one that might have been published after deleting the old one.[5]

The chunks are uploaded to the chunks bucket first, and deleted as they are uploaded. Once all the chunks have been deleted (and thus uploaded), the “ready/chunks” subdirectory can be deleted. Copiers then race to upload the contents of the “ready/meta” subdirectory to the directory bucket. Now that blob stores (even S3) provide strong consistency, this ordering guarantees that, once a directory blob is visible, so are the chunks it refers to. Once the copier has fully uploaded the contents of the “ready” subdirectory, the subdirectory can be removed; POSIX guarantees that this rmdir call will only succeed if “ready” is empty (and thus if all its contents have been deleted).[6]

Writers should update the “ready” directory whenever it is missing or empty.[7] They do so by constructing the new “ready” directory at a temporary path, and publish that with an atomic rename that only succeeds if the “ready” directory is missing or empty. When they construct the temporary ready directory, writers populate the “chunks” subdirectory with everything in the “staging” subdirectory that is useful for the current snapshot, and populate the “meta” subdirectory with the directory proto for the current snapshot.

Once a writer has successfully populated the “ready” directory, it can delete all staged chunks: a chunk is either not useful anymore, or is now in the “ready” directory, and will only be deleted once it has been successfully copied to remote storage.

Before doing that, writers *always* update the “staging” directory with an up-to-date snapshot of the database file. They do so by first populating chunks in the “staging/chunks” subdirectory, and then atomically linking over the directory proto file in “staging/meta”. In the worst case, writers can always enumerate all relevant chunks by reading and fingerprinting all 64 KB-chunks in the sqlite file. However, writers can usually start from a directory file that was up-to-date just before their write transaction. This lets writers read and fingerprint only chunks that the transaction wrote to. In fact, when the writes are aligned to 64 KB boundaries, writers publish

[5] We could also use the same seqlock approach taken for the “staging” directory: in that case, failure (mismatched “meta” files) would only occur if another copier had already made enough progress on the “ready” directory for writers to overwrite it. However, we’d also want to tag the contents of the “meta” directory with a pseudorandom value, to avoid ABA when copiers delete files they have uploaded.
[6] Misuse could leave extra files or subdirectories in there. We should add logic to move those to “dead letter” subdirectories.
[7] TODO: while we currently check for that case by listing the directory, we should probably just try to rmdir. If it succeeds or fails with ENOENT, the directory is now gone (:

the chunks before even updating the sqlite database file: it's never incorrect to publish or upload a chunk file (as long as its name matches its contents), even if later proves to be useless.

Once the staging directory is up to date, writers can garbage collect useless chunks. That's not necessary for correctness, but bounds space usage when replication gets stuck (e.g., partial network outage). It is safe to delete unused chunks because the writer thread still holds a read lock on the sqlite file, so no new chunk may unexpectedly become useful. A marking GC can waste a lot of time when it doesn't remove anything, so we use a PRNG to trigger GCs once the number of useless chunks is roughly proportional to the number of useful ones.

All these operations may be performed by an arbitrary number of concurrent "writer" processes, as long as they hold a read lock on the sqlite file: the lock guarantees that they work off the same file, and thus agree on what the snapshot should look like in the end.

Copiers can always consume (upload and then delete) files from the "staging/chunks" subdirectory: as long as the chunks are correctly named, it's not incorrect to have a few unreferenced chunks. They then use an asymmetric synchronisation protocol, like sequence locks, to determine when it's safe to upload the staged directory proto file in "staging/meta". They:

1. Construct a unique identifier for the proto file from its inode number, birth time, and high-precision ctime.
2. Check that the "staged/chunks" directory is still empty.
3. Check that there is no "ready" directory (otherwise "staged/chunks" could be empty merely because the chunks now live in "ready/chunks").
4. Check that the staged directory proto file in "staging/meta" still has the same unique identifier; if so, it can be uploaded to the remote blob store.

Having confirmed that the directory file hasn't changed in step 4, copiers know that there was a time when all the chunks for that directory file had been uploaded (there was nothing left in "staged/chunks" and the "ready" directory was empty), and it's thus safe to upload the staged directory blob.

What if the "ready" directory is only empty because it was populated and then consumed? That's still OK because consumption implies that the chunks were uploaded. In fact, since the staged proto file hasn't changed, it even implies that the "ready" directory matched the current staged snapshot!

What if the "chunks" directory was populated after we checked that it was empty? That's fine because it was empty at some point, so anything newly added isn't useful for the directory file we're about to upload.

Altogether, we thus have a fully asynchronous buffer between writers that publish new snapshot to the spooling directory, and copiers that upload spooled data to remote storage. Any number of copiers can cooperate on the same spooling directory, copiers never block writers, and any number of writers can cooperate, as long as they do so while holding a read lock on the replicated db file.

Finally, the size of the buffer is bounded: in the worst case, we have one full copy of the DB in the “ready” directory, and a few copies in the “staging” directory (GC is randomised, so could be delayed, but it will most likely run before we have more than 3-4 copies of the DB file).[8]

This decoupling of writers (sqlite commit code) and copiers, along with the space bound on spooled data, means that we can trivially control replication costs by pacing copiers.

We have good bounds on space usage for spooled data, but nothing once the data has been uploaded to remote storage. That’s mostly because we assume remote storage is much more capacious and affordable than local storage. However, unbounded growth is never good. The plan is to let the bucket for directory protobuf blobs be versioned, and apply retention rules on old versions, and to impose a long (order of several months) TTL to the content-addressed chunks bucket. Copiers will have logic to “touch” chunks that are still active, at least once a day.

[8] We should make the probability tunable, to make it possible to force a GC by opening and closing a transaction on the sqlite db.