Switcher Role with MVCC

Concurrent Readers and a Single Writer -
A Locking Model for Distributed Systems with Shared Storage

2021-03-21 v4 Dan Forsberg, D.Sc.

Motivation and Goals

- We are living in the era of compute and storage separation, where both can scale independently - access to the storage has higher latency than local disk, whilst the throughput can compete (especially with concurrent readers and high perf net)
- Our overall goal is to **make an OLAP MVCC RDBM support concurrent readers and a Writer in separate machines** (i.e. no signalling, no shared memory/disk, no clustering, no direct network connections, ..).
- In other words, allowing a Writer to push bigger batches (using WAL) to the db concurrently with high number of Readers - e.g. writing from a stream to the db while allowing aggregation queries (possibly long running) to run all the time

- We want to **optimise Reader performance** for low latency and avoid long breaks for version upgrades, as we see that as characteristic for an OLAP system (we are not optimising TPS per se)
- We want to achieve **low interference (Readers) version upgrades while high number of concurrent Readers** (hundreds) are being served
- We want to try optimistic approaches, where failover may take more time than fully locked initial approach in the hopes that optimistic approach works almost all the time
- **Writer writes (big) import batches with low or very low TPS** (until the "dataset is ready")

High-level Description

- Optimistic approach: **Readers** check the current version without initial lock, and eventually have shared lock on the correct current version*
- **"Switcher"**: Goal is to avoid Writer starvation by making any new Reader or Writer to **switch current version and truncate WAL** iff the WAL version and old version wal version match and is high enough (old version == current version + WAL).
- Normally, Writer switches the version if it gets the EXCLUSIVE lock on current version with N tries, so that the "world freeze tick" is avoided

*) Readers retry as long as the version matches when they check again with the lock acquired. This allows readers to acquire lock on the correct current version but also have potentially less operations to execute, if the lock is correctly set on current version from the start (alternatively could start acquiring lock always on h1, but that would be wrong choice when the current version is h2)

- **Writers** write to WAL and copy to old version whenever they get the lock with first try to old version**
- Versions are stored on main db file as "headers"
- Headers include the version number for h1 and h2, highest being the current, but also WAL version that has been copied to the old version. Current version has always wal version 0***

**) Only when old version Readers are gone, Writer can start to merge updates to old version to avoid disk image mismatch for old version Reader. No lock waiting needed and one try is enough to ensure fast and smooth writes.
***) Readers read the WAL with current version always, no tracking needed.

Version Upgrade - The Challenge and Solution

- **Challenge**: if there are continuous new readers having SHARED lock on current version, Writer never gets EXCLUSIVE lock on current version and thus can not update current version (DoS)
- **Solution**: Any new Reader or Writer adopts “Switcher” role for **switching current version and truncate WAL** when the conditions for version switching are met..
- **"Switcher" waits for EXCLUSIVE lock on current version** so that the current version can be switched and WAL truncated without causing new reader errors*

*) New readers will read the current version + WAL, but the WAL may have been truncated before the reader opens WAL if there is no EXCLUSIVE locking by the "Switcher"

- Check the headers and WAL again to see if the switch is still needed, if not, clear lock and yield/abandon the "Switcher" role
- WAL is truncated first*, then current version switched. This transfers the “current EXCLUSIVE” lock to “old version EXCLUSIVE” lock and essentially all new readers are free to read the new current version immediately**.

) Once the switch happens, new readers can immediately acquire SHARED lock on current version while the (switcher) process having still the EXCLUSIVE lock, **now on old version, can finish. In case a "Switcher" accidentally waited EXCLUSIVE lock(*) on wrong version (version is checked without locking!), it will delay new version Readers for short time period as the “Switcher" waiters will release their locks one by one.

Version Upgrade - Reader perspective

- Reader starts reading headers and WAL version (no locking), if the VERSION UPGRADE condition is met**, Reader switches to "Switcher" role
- In "Switcher" role, the Reader starts to wait for EXCLUSIVE on current version (blocking)
- When reader gets the EXCLUSIVE, it checks whether the switch was already done or not
- If already done, it clears the EXCLUSIVE, and reverts back to normal SHARED lock on the new current version (starts from the beginning again)
- **If switch is still needed, *it executes it* and then clears EXCLUSIVE, and acquires SHARED lock on the new current version**

) To avoid too frequent forced VERSION UPGRADES, the WAL version must be **high enough to trigger the procedure. When this condition is met the Writer will not update WAL anymore either, so no locking is needed. If Writer gets EXCLUSIVE on the current version too, it can do switch immediately.

Version Upgrade - Writer perspective

- Writer acquires EXCLUSIVE lock on the old version, but if it fails (old version readers still exist) it will update WAL only*
- When it has EXCLUSIVE on old version and when old version wal version matches with WAL version and is above threshold, writer will clear EXCLUSIVE on old version** and wait for EXCLUSIVE on current version, as it adopts the "Switcher" role
- Otherwise, Writer updates both old version and WAL, and tries EXCLUSIVE on current version N times
- Once Writer gets EXCLUSIVE on current version is also checks whether switch is already done or not
- If not done, it executes the switch procedures
- If switch is already done, clear lock***
- If still needs to write, re-acquire EXCLUSIVE lock on old version
- Writer (re-)opens (now truncated) WAL, writes new data there and then syncs that to old version while having the lock on it, then it clears the lock

*) If EXCLUSIVE lock fails on old version, it means that WAL can not be copied to old version so version switch can not happen either

**) Allows new readers to get SHARED lock on new current version as the old version will become the new current version

***) Allows all other EXCLUSIVE lock waiters "queue" to be purged. Note: inside OS, signal handlers would be used to notify all "waiters", but we need the solution to work between machines. Using distributed locking services is left for future study.

Header structure and versioning

- *Headers include the version number for h1 and h2 (DuckDB), highest being the current, but also WAL version that has been copied to the version. Current version has always WAL version value 0, because current version is not being updated. Readers read current version and the WAL if it exists. Version upgrade (switch) can happen when WAL version W equals the old version header WAL version and is above threshold. At that point old version is the same as current version plus WAL*
- h1 is current version. No WAL yet.
  - `h1: n,0; h2: n-1,0 :: h1 (current) == h2 (old)`
  - **⇒ both version are the same**
- h1 is current version. WAL version is W > W'
  - `h1: n,0; h2: n-1,W' :: h1 (current) == h2 (old)`
  - **⇒ versions differ, old version does not have <u>all</u> the WAL copied. Writer updates old version as long as W == W´ (no conflict with readers reading current version, but old readers will block copying, so writer updates WAL only as long as it can not get exclusive lock on old version)**
- h1 is current version. WAL version is W == W´
  - `h1: n,0; h2: n-1,W' :: h1 (current) == h2 (old) + W (wal)`
  - **⇒ both versions are the same when WAL is added to current (no old readers anymore and Writer has been updating the old version along with WAL)**
- **⇒ "Switcher" Role executes version upgrade ⇐**
- h2 is current version. No WAL yet.
  - `h1: n,0; h2: n+1,0 :: h2 (current) == h1 (old)`
  - **⇒ both version are the same**
- h2 is current version. WAL version is W2 == W2'
  - `h1: n,W2'; h2: n+1,0 :: h2 (current) == h1 (old) + W2 (wal)`
  - **⇒ both versions are the same when WAL is added to current (no old readers anymore and Writer has been updating the old version along with WAL)**

Considerations

- The "whole world" stops for a short period of time when the forced switch conditions are met - this can be tuned with the threshold
- The switch condition must include greater than check as the WAL version may grow independently while there are still Readers on the old version
- If the WAL has grown a lot once there are no old version Readers anymore, Writer may impose a bigger latency spike if it is to synchronise all WAL data to the old version, thus Writer could do this in steps instead (optimisation)
- It is better to make the contention case to not involve any actual reading or writing despite acquiring the locks and verifying that switch is either still needed or done already - this way hopefully - the latency hit taken is on average small and acceptable overall
- **Future work**: it is probably possible to extend the model to support multiple Writers too. When this model is already in place, much of the complexity of MVCC with multiple Writers too is already materialised

Improvements and Alternatives

- Could think about using distributed Redis locking instead of a shared filesystem file descriptor locking, depending on the latency between the two and the number of "locks" that are needed/supported. Compare also pro/con of having a requirement for an existing Redis service. Like advisory locks do not prevent reads/writes.
- If Writer is trying locking for N times, think about alternatively using wait lock and another thread sending "timeout signal" to break the wait (removes polling)
- Count round trips (like read headers, acquire lock, read WAL version, etc.) for a Reader and Writer and think about using multiple threads for running concurrent operations when it is possible and reduces overall latency