Storing the Cardano ledger state on disk: integration notes for high-performance backend

Duncan Coutts Joris Dral Wolfgang Jeltsch

July 2025

Contents

1 Sessions

Creating new empty tables or opening tables from snapshots requires a `Session`. The session can be created using `openSession`, which has to be done in the consensus layer. The session should be shared between all tables. Sharing between a table and its duplicates, which are created using `duplicate`, is automatic. Once the session is created, it could be stored in the `LedgerDB`. When the `LedgerDB` is closed, all tables and the session should be closed. Closing the session will automatically close all tables, but this is only intended to be a backup functionality: ideally the user closes all tables manually.

2 The compact index

The compact index is a memory-efficient data structure that maintains serialised keys. Rather than storing full keys, it only stores the first 64 bits of each key.

The compact index only works properly if in most cases it can determine the order of two serialised keys by looking at their 64-bit prefixes. This is the case, for example, when the keys are hashes: the probability that two hashes have the same 64-bit prefixes is 2^{-64} and thus very small. If the hashes are 256 bits in size, then the compact index uses 4 times less memory than if it would store the full keys.

There is a backup mechanism in place for the case when the 64-bit prefixes of keys are not sufficient to make a comparison. This backup mechanism is less memory-efficient and less

performant. That said, if the probability of prefix clashes is very small, like in the example above, then in practice the backup mechanism will never be used.

UTXO keys are *almost* uniformly distributed. Each UTXO key consist of a 32-byte hash and a 2-byte index. While the distribution of hashes is uniform, the distribution of indexes is not, as indexes are counters that always start at 0. A typical transaction has two inputs and two outputs and thus requires storing two UTXO keys that have the same hash part, albeit not the same index part. If we serialise UTXO keys naively, putting the hash part before the index part, then the 64-bit prefixes will often not be sufficient to make comparisons between keys. As a result, the backup mechanism will kick in way too often, which will severely hamper performance.

The solution is to change the serialisation of UTXO keys such that the first 64 bits of a serialised key comprise the 2-byte index and just 48 bits of the hash. This way, comparisons of keys with equal hashes will succeed, as the indexes will be taken into account. On the other hand, it becomes more likely that the covered bits of hashes are not enough to distinguish between different hashes, but the propability of this should still be so low that the backup mechanism will not kick in in practice.

Importantly, range lookups and cursor reads return key–value pairs in the order of their *serialised* keys. With the described change to UTXO key serialisation, the ordering of serialised keys no longer matches the ordering of actual, unserialised keys. This is fine for `lsm-tree`, for which any total ordering of keys is as good as any other total ordering. However, the consensus layer will face the situation where a range lookup or a cursor read returns key–value pairs slightly out of order. Currently, we do not expect this to cause problems.

3 Snapshots

Snapshots currently require support for hard links. This means that on Windows the library only works when using NTFS. Support for other file systems could be added by providing an alternative snapshotting method, but such a method would likely involve copying file contents, which is slower than hard-linking.

Creating a snapshot outside the session directory while still using hard links should be possible as long as the directory for the snapshot is on the same disk volume as the session directory, but this feature is currently not implemented. Hard-linking across different volumes is generally not possible; therefore, placing a snapshot on a volume that does not hold the associated session directory requires a different snapshotting implementation, which would probably also rely on copying file contents.

A copying snapshotting implementation would probably kill two birds with one stone by removing the two current limitations just discussed.

Presumably, `cardano-node` will eventually be required to support storing snapshots on a different volume than where the session is placed, for example on a cheaper non-SSD drive. This feature was unfortunately not anticipated in the project specification and so is not currently included. As discussed above, it could be added with some additional work.

4 Value resolving

When instantiating the `ResolveValue` class, it is usually advisable to implement `resolveValue` such that it works directly on the serialised values. This is typically cheaper than having `resolveValue` deserialise the values, composing them, and then serialising the result. For

example, when the resolve function is intended to work like `(+)`, then `resolveValue` could add the raw bytes of the serialised values and would likely achieve better performance this way.

5 `io-classes` incompatibility

At the time of writing, various packages in the `cardano-node` stack depend on `io-classes-1.5` and the 1.5-versions of its daughter packages, like `strict-stm`. For example, the build dependencies in `ouroboros-consensus.cabal` contain the following:

- `io-classes ^>= 1.5`
- `strict-stm ^>= 1.5`

However, `lsm-tree` needs `io-classes-1.6` or `io-classes-1.7`, and this leads to a dependency conflict. One would hope that a package could have loose enough bounds that it could be built with `io-classes-1.5`, `io-classes-1.6`, and `io-classes-1.7`. Unfortunately, this is not the case, because, starting with the `io-classes-1.6` release, daughter packages like `strict-stm` are sublibraries of `io-classes`. For example, the build dependencies in `lsm-tree.cabal` contain the following:

- `io-classes ^>= 1.6 || ^>= 1.7`
- `io-classes:strict-stm`

Sadly, there is currently no way to express both sets of build dependencies within a single `build-depends` field, as Cabal's support for conditional expressions is not powerful enough for this.

It is known to us that the `ouroboros-consensus` stack has not been updated to `io-classes-1.7` due to a bug related to Nix. For more information, see https://github.com/IntersectMBO/ouroboros-network/pull/4951. We would advise to fix this Nix-related bug rather than downgrading `lsm-tree`'s dependency on `io-classes` to version 1.5.

6 Security of hash-based data structures

Data structures based on hashing have to be considered carefully when they may be used with untrusted data. For example, an attacker who can control the keys in a hash table may be able to provoke hash collisions and cause unexpected performance problems this way. This is why the Haskell Cardano node implementation does not use hash tables but ordering-based containers, such as those provided by `Data.Map`.

The Bloom filters in an LSM-Tree are hash-based data structures. For the sake of performance, they do not use cryptographic hashes. Thus, without additional measures, an attacker can in principle choose keys whose hashes identify mostly the same bits. This is a potential problem for the UTxO and other stake-related tables in Cardano, since it is the users who get to pick their UTxO keys (TxIn) and stake keys (verification key hashes) and these keys will hash the same way on all other Cardano nodes.

This issue was not considered in the original project specification, but we have taken it into account and have included a mitigation in `lsm-tree`. The mitigation is that, on the initial creation of a session, a random salt is conjured and stored persistenly as part of the session. This salt is then used as part of the Bloom filter hashing for all runs in all tables of the session.

The consequence is that, while it is in principle still possible to produce hash collisions in the Bloom filter, this now depends on knowing the salt. However, every node should have a different salt, in which case no single block can be used to attack every node in the system. It is

only plausible to target individual nodes, but discovering a node's salt is extremely difficult. In principle there is a timing side channel, in that collisions will cause more I/O and thus cause longer running times. To exploit this, an attacker would need to get upstream of a victim node, supply a valid block on top of the current chain and measure the timing of receiving the block downstream. There would, however, be a large amount of noise spoiling such measurements, necessitating many samples. Creating many samples requires creating many blocks that the victim node will adopt, which requires substantial stake (or successfully executing an eclispse attack).

Overall, our judgement is that our mitigation is sufficient, but it merits a security review from others who may make a different judgement. It is also worth noting that the described hash clash issue may occur in other LSM-tree implementations used in other software, related and unrelated to Cardano. In particular, RocksDB does not appear to use a salt at all.

Note that using a per-run or per-table hash salt would incur non-trivial costs, because it would reduce the sharing available in bulk Bloom filter lookups, where several keys are looked up in several filters. Given that the Bloom filter lookup is a performance-sensitive part of the overall database implementation, such an approach to salting does not seem feasible. Therefore, we chose to generate hash salts per session.

In the Cardano context, a downside of picking Bloom filter salts per session and thus per node is that this interacts poorly with sharing of pre-created databases. While it would still be possible to copy a whole database session, since this includes the salt, doing so would result in the salt being shared between nodes. If SPOs shared databases widely with each other, to avoid processing the entire chain, then the salt diversity would be lost.

Picking Bloom filter salts per session is particularly problematic for Mithril. The current Mithril PoC works by copying the node's on-disk file formats. This design has numerous drawbacks, but would be particularly bad in this context because it would share the same Bloom filter salt to all Mithril users. If Mithril were to use a proper externally defined snapshot format, rather than just copying the node's on-disk formats, then restoring a snapshot would naturally involve creating a new LSM tree session and thus a fresh local salt. This would solve the problem.

7 Possible incompatibility with the XFS file system

We have seen at least one failure when disabling disk caching via the table configuration, using the `DiskCacheNone` setting. Albeit it is unconfirmed, we suspect that some versions of Linux's XFS file system implementation, in particular the one used by the default AWS Amazon Linux 2023 AMI, do not support the system call that underlies `fileSetCaching` from the `unix` package. This is an `fcntl` call, used to set the file status flag `O_DIRECT`. XFS certainly supports `O_DIRECT`, but it may support it only when the file in question is opened using this flag, not when trying to set this flag for an already open file.

This problem can be worked around by using the ext4 file system or by using `DiskCacheAll` in the table configuration, the latter at the cost of using more memory and putting pressure on the page cache. If this problem is confirmed to be widespread, it may become necessary to extend the `unix` package to allow setting the `O_DIRECT` flag upon file opening.