# Storing the Cardano ledger state on disk: final report for a high-performance backend

A TECHNICAL REPORT BY WELL-TYPED ON BEHALF OF INTERSECT

Duncan Coutts Joris Dral Wolfgang Jeltsch

July 2025

## Contents

# Changelog

**1.1** Joris Dral (July 2025) — Fixed an issue with rendering the table in the "Results of the primary benchmark" section

**1.0** Duncan Coutts, Joris Dral, Wolfgang Jeltsch (July 2025) — Final version

# 1 Introduction

As part of the project to reduce `cardano-node`'s memory use [1] by storing the bulk of the ledger state on disk, colloquially known as UTxO-HD[1], a high-performance disk backend was developed as an arm's-length project by Well-Typed LLP on behalf of the Cardano Development Foundation and Intersect[2]. The intent is for the backend to be integrated into the consensus layer of `cardano-node`, specifically to be used for storing the larger parts of the Cardano ledger state.

This backend is now complete. It satisfies all its functional requirements and meets all its performance requirements, including stretch targets.

The backend is implemented as a Haskell library called `lsm-tree` [2], which provides efficient on-disk key–value storage using log-structured merge-trees, or LSM-trees for short. An LSM-tree is a data structure for key–value mappings that is optimized for large tables with a high insertion rate, such as the UTxO set and other stake-related data. The library has a number of custom features that are primarily tailored towards use cases of the consensus layer, but the library should be useful for the broader Haskell community as well.

Currently, a UTxO-HD `cardano-node` already exists, but it is an MVP that uses off-the-shelf database software (LMDB) to store a part of the ledger state on disk [3]. Though the LMDB-based solution is suitable for the current state of the Cardano blockchain, it is not suitable to achieve Cardano's long-term business requirements [1, Sec. 3], such as high throughput with limited system resources. The goal of `lsm-tree` is to pave the way for achieving said business requirements, providing the necessary foundation on which technologies like Ouroboros Leios can build.

Prior to development, an analysis was conducted, leading to a comprehensive requirements document [4] outlining the functional and non-functional (performance) requirements for the `lsm-tree` library. The requirements document includes recommendations for the disk backend to meet the following criteria:

1. It helps achieve the higher performance targets.
2. It supports the new and improved operations required by the ledger to be able to move the stake-related tables to disk.
3. It is able to be tested in the context of the integrated consensus layer using the I/O fault testing framework.

This report aims to set out the requirements for the `lsm-tree` library as described in the original requirements document and to analyse how and to what extent these requirements are met by the implementation. The intent of this report is not to substantiate how the requirements came to be; the full context including technical details is available in the original requirements document for the interested reader.

[1] 'UTxO-HD' is a classic project-manager's misnomer. The project is not just about the UTxO but also other parts of the ledger state, and it is not about hard drives. Indeed, the performance of hard drives is too low to support the feature; faster drives like SSDs are needed instead. A better project name would be 'On-disk ledger state', but there is no way to remove poorly chosen names from people's heads once they are firmly engrained.

[2] In its early stages, it was developed on behalf of Input Output Global, Inc. (IOG). It was novated in early 2024.

It should be noted that the requirements of the `lsm-tree` component were specified in isolation from the consensus layer and `cardano-node`, but these requirements were of course chosen with the larger system in mind. This report only reviews the development of `lsm-tree` as a standalone component, while integration notes are provided in an accompanying document [5]. Integration of `lsm-tree` with the consensus layer will happen as a separate phase of the UTxO-HD project.

Readers are advised to familiarise themselves with the API of the library by reading through the Haddock documentation of the public API. A version of the Haddock documentation that tracks the `main` branch of the repository is hosted using GitHub Pages [6]. There are two modules that make up the public API: the `Database.LSMTree` module contains the full-featured public API, whereas the `Database.LSMTree.Simple` module offers a simplified version that is aimed at new users and use cases that do not require advanced features. Additional documentation can be found in the package description [7]. This and the simple module should be good places to start at before moving on to the full-featured module.

The version of the library that is used as the basis for this report is tagged `final-report` in the `lsm-tree` Git repository [2]. It can be checked out using the following commands:

```
git clone git@github.com:IntersectMBO/lsm-tree.git
cd lsm-tree
git checkout final-report
```

## 2 Development history

Before the `lsm-tree` project was officially commissioned, we performed some prototyping and design work. This preparatory work was important to de-risk key aspects of the design, such as achieving high SSD throughput using a Haskell program. The initial results from the prototypes were positive, which contributed to the decision to move forward with the project.

Development of `lsm-tree` officially started in early autumn 2023. The first stages focused mostly on additional prototyping and design but also included testing. Among the prototyping and design artefacts are the following:

- A prototype for the incremental merge algorithm [8]
- A library for high SSD throughput using asynchronous I/O [9]
- Specifications of the formats of on-disk files and directories [10]

In the spirit of test-driven development, we created a reference implementation for the library, modelling each of the basic and advanced features that the library would eventually have. Additionally, we implemented tests targeting the reference implementation. We designed these tests in such a way that they could later be reused by the actual implementation of the library, but initially they served as sanity checks for the reference implementation.

After the initial phase, we worked towards an MVP for the `lsm-tree` library. During this period, we wrote most of the core code of the library. We aimed at making the MVP a full implementation of a key–value store, covering the basic operations, namely insertion, deletion and lookup, but not necessarily advanced or custom features.

Over the course of time, we wrote tests and micro-benchmarks for the library internals to sanity-check our progress. In particular, we created micro-benchmarks to catch performance defects or regressions early.

We focused on defining the in-memory structures and algorithms of the MVP first before moving on to I/O code. This separation proved fruitful: the purely functional code could be tested and benchmarked before dealing with the more tricky aspects of impure code.

Towards completion of the MVP, we created a macro-benchmark that simulates the workload when handling the UTxO set – the largest component of the ledger state that UTxO-HD moves to disk and likely the most performance-sensitive dataset. This benchmark serves in particular as the basis for assessing the performance requirements, which will be discussed further in the *Performance requirements* section. Results of running the macro-benchmark on the MVP showed promising results.

With the core of the library implemented, we turned to adding more features to the library, beyond the basic key–value operations, including some special features that are not provided by off-the-shelf database software. The following features are worth mentioning:

- Upserts
- Separate blob retrieval
- Range lookups and cursors
- Incremental merges
- Multiple writeable table references
- Snapshots
- Table unions
- I/O fault testing

Most of these features will also be touched upon in the *Functional requirements* section.

Along the way, we optimised and refactored code to improve the quality of the library and its tests. For each completed feature, we enabled the corresponding tests that we had written for the reference implementation to check the real implementation as well. The passing of these tests served as a rite of passage for each completed feature.

In the final stages, we reviewed and improved the public API, tests, benchmarks, documentation and library packaging. We constructed the final deliverables, such as this report and additional integration notes [5], which should guide the integration of `lsm-tree` with the consensus layer. In April 2025, we reached the final milestone.

# 3 Functional requirements

This section outlines the functional requirements for the `lsm-tree` library and how they are satisfied. These requirements are described in the original requirements document [4, Sec. 18.2].

Several requirements specify that an appropriate test should demonstrate the desired functionality. Though the internals of the library are extensively tested directly, the tests that should be of most interest to the reader are those that are written against the public API. These tests can be found in the modules listed below, whose source code is under the `test` directory. Where it aids the report in the analysis of the functional requirements, we will reference specific tests. Beyond this, the reader is encouraged to peruse and review the listed modules.

| Module | Test style |
|---|---|
| `Test.Database.LSMTree.Class` | Classic |
| `Test.Database.LSMTree.StateMachine` | State machine |
| `Test.Database.LSMTree.StateMachine.DL` | State machine |
| `Test.Database.LSMTree.UnitTests` | Unit |

The tests are written in three styles:

- *Classic* QuickCheck-style property tests check for particular properties of the library's features. Many of these properties are tailored towards specific functional requirements, and their tests are usually grouped according to the requirements they refer to.
- *State machine* tests compare the library against a model (the reference implementation) for a broad spectrum of interleaved actions that are performed in lock-step by the library and its model. The state machine tests generate a variety of scenarios, some of which are similar to scenarios that the classic property tests comprise. A clear benefit of the state machine tests is that they are more likely than classic property tests to catch subtle edge-cases and interactions.
- *Unit* tests are used to exercise behaviour that is interesting in its own right but not suitable for inclusion in the state machine tests because it would negatively affect their coverage. For example, performing operations on closed tables will always throw the same exception; so not including this behaviour in the state machine tests means that these tests can do interesting things with *open* tables more often.

### 3.1 Requirement 1

> It should have an interface that is capable of implementing the existing interface used by the existing consensus layer for its on-disk backends.

For the analysis of this functional requirement, we use a fixed version of the `ouroboros-consensus` repository [11]. This version can be checked out using the following commands:

```
git clone git@github.com:IntersectMBO/ouroboros-consensus.git
cd ouroboros-consensus
git checkout 9d41590555954c511d5f81682ccf7bc963659708
```

The consensus interface that has to be implemented using `lsm-tree` is given by the `LedgerTablesHandle` record type [12]. This type provides an abstract view on the table storage, so that the rest of the consensus layer does not have to concern itself with the concrete implementation of that storage, be it based on `lsm-tree` or not; `ouroboros-consensus` can freely pick any particular record as long as it constitutes a faithful implementation of the storage interface. This has advantages for initial functional integration because the integration effort is confined to the implementation of the record. To take full advantage of all of `lsm-tree`'s features, further integration efforts would be needed because changes to the interface and the rest of the consensus layer would be required. However, this is considered out of scope for the current phase of the UTxO-HD project.

Currently, the consensus layer has one implementation of table storage for the ledger, which stores all data in main memory [13]. This implementation preserves much of the behaviour of a pre-UTxO-HD node. A closer look at it shows that there are two pieces of implementation-specific functionality that are not covered by the `LedgerTablesHandle` record: creating a fresh such record and producing such a record from an on-disk snapshot. It makes sense that these are standalone functions, as they produce the records in the first place.

All in all, we are left with the following API to implement in the integration phase:

```
newLSMTreeLedgerTablesHandle
  :: _ -> m (LedgerTablesHandle m l)
  -- newTable

openLSMTreeLedgerTablesHandleSnapshot
  :: _ -> m (LedgerTablesHandle m l)
  -- openTableFromSnapshot
```

```
data LedgerTablesHandle m l = LedgerTablesHandle {
    close              :: _ -- closeTable
  , duplicate          :: _ -- duplicate
  , read               :: _ -- lookups
  , readRange          :: _ -- rangeLookup or Cursor functions
  , readAll            :: _ -- rangeLookup or Cursor functions
  , pushDiffs          :: _ -- updates
  , takeHandleSnapshot :: _ -- saveSnapshot
  , tablesSize         :: _ -- return Nothing
  }
```

For the sake of brevity, we have elided the types of the different functions, replacing them by underscores. To give a sense of how the `lsm-tree` library's interface fits the interface used in `ouroboros-consensus`, each of the functions above has a comment that lists the corresponding function or functions from `lsm-tree`'s public API that should be used to implement it.

Note that `tablesSize` should always return `Nothing` in the case of the `lsm-tree`-based implementation. While the in-memory implementation can determine the number of entries in a table in constant time, given that this value is cached, the `lsm-tree`-based implementation would have to read all the *physical* entries from disk in order to compute the number of *logical* entries because it would have to ensure that key duplicates are ignored. We already anticipated this when we defined the `LedgerTablesHandle` type and consequently accounted for it by using `Maybe Int` as the return type of `tablesSize`.

The analysis above offers a simplified view on how the `lsm-tree` and consensus interfaces fit together; so this report is accompanied by integration notes [5] that provide further guidance. These notes include, for example, an explanation of the need to store a session context in the ledger database. However, implementation details like these are not considered to be blockers for the integration efforts, as there are clear paths forward.

### 3.2 Requirement 2

> The basic properties of being a key value store should be demonstrated using an appropriate test or tests.

`lsm-tree` has been supporting the basic operations `insert`, `delete` and `lookup` as well as their bulk versions `inserts`, `deletes` and `lookups` since the MVP was finished. It also provides operations `update` and `updates`, realised as combinations of inserts and deletes.

We generally advise to prefer the bulk operations over the elementary ones. On Linux systems, lookups in particular will better utilise the storage bandwidth when the bulk version is used, especially in a concurrent setting. This is due to the method used to perform batches of I/O, which employs the `blockio-uring` package [9]: submitting many batches of I/O concurrently will lead to many I/O requests being in flight at once, so that the SSD bandwidth can be saturated. This is particularly relevant for the consensus layer, which will have to employ concurrent batching to meet higher performance targets, for example by using a pipelining design.

It is not part of the requirements but does deserve to be mentioned that there is specific support for storing blobs (binary large objects). Morally, blobs are part of values, but they are stored and retrieved separately. The reasoning behind this feature is that there are use cases where not all of the data in a value is needed by the user. For example, when a Cardano node is replaying the current chain, the consensus layer skips expensive checks like script validation. Scripts are part of UTxO values, and we could skip reading them during replay if we stored them as blobs. Note

that the performance improvements from using blob storage are only with lookups; *updates* involving blobs are about as expensive as if the blobs' contents were included in the values.

The implementation of updates in an LSM tree involves merging files together. A naïve implementation of updates would merge files all in one go, which entails a high latency for the occasional update that triggers a full file merge. The `lsm-tree` library can avoid such spikes by spreading out I/O over time, using an incremental merge algorithm: the algorithm that we prototyped at the start of the `lsm-tree` project [8]. Avoiding latency spikes is essential for `cardano-node` because `cardano-node` is a real-time system, which has to respond to input promptly. The use of the incremental merge algorithm does not improve the time complexity of a sequence of updates as such, but it turns the *amortised* time complexity of the naive solution into a *worst-case* time complexity.

### 3.3 Requirement 3

> It should have an extended interface that supports key–range lookups, and this should be demonstrated using an appropriate test or tests.

The library offers two alternatives for key–range lookups:

- *The `rangeLookup` function* requires the user to specify a range of keys via a lower and an upper bound and reads the corresponding entries from a table.
- A *cursor* can be used to read consecutive segments of a table. The user can position a cursor at a specific key and then read database entries from there either one by one or in bulk, with bulk reading being the recommended method. A cursor offers a stable view of a table much like an independently writeable reference (see functional requirement 5), meaning that updates to the original table are not visible through the cursor. Currently, reading from a cursor simultaneously advances the cursor position.

Internally, `rangeLookup` is defined in terms of the cursor interface but is provided for convenience nonetheless. A range lookup always returns as many entries as there are in the specified table segment, while reading through a cursor returns as many entries as requested, unless the end of the table is hit.

### 3.4 Requirement 4

> It should have an extended interface that supports a 'monoidal update' operation in addition to the normal insert, delete and lookup. The choice of monoid should be per table/mapping (not per operation). The behaviour of this operation should be demonstrated using an appropriate test.

The terminology related to monoidal updates in `lsm-tree` is different from the terminology used in the original functional requirement. From this point onwards, we will say 'upsert' instead of 'monoidal update' and 'resolve function' instead of 'monoid'.

Where `lsm-tree`'s `insert` and `delete` behave like the functions `insert` and `delete` from `Data.Map`, `upsert` behaves like `insertWith`: if the table contains no value for the given key, the given value is inserted; otherwise, the given value is combined with the one in the table to form the new value. The combining of values is done with a user-supplied resolve function. It is required that this function is associative. Examples of associative functions are `(+)` and `(*)` but also `const` and `max`. If the resolve function is `const`, an upsert behaves like an insert. The choice of resolve function is per table, and this is enforced using a `ResolveValue` class constraint. The user can implement the resolve function on unserialised values, serialised values or both.

It is advisable to at least implement it on serialised values because this way serialisation and deserialisation of values can be avoided.

Like inserts and deletes, upserts can be submitted either one by one or in bulk. There is also an `updates` function that allows submitting a mix of upserts, inserts and deletes.

### 3.5 Requirement 5

> It should have an extended interface that exposes the ability to support multiple independently writable references to different versions of the datastore. The behaviour of this functionality should be demonstrated using an appropriate test. For further details see Section 17.

Multiple independently writable references can be created using `duplicate`. `duplicate` is relatively cheap because we do not need to copy data around: Haskell provides us with data structure persistence, sharing and garbage collection for in-memory data, and we simulate these features for on-disk data. However, it is important to make a distinction between *immutable* and *mutable* data. We will focus on immutable data first.

Immutable in-memory data can be shared between tables and their duplicates 'for free' by relying on Haskell's persistent data structures, and the garbage collector will free memory 'for free' when it becomes unused. What is inherent to LSM-trees is that on-disk data is immutable once created, which means we can also share immutable on-disk data 'for free'. Garbage collection for on-disk data does not come for free however. Therefore, internally all shared disk resources (like files and file handles) are reference-counted using a custom reference counting scheme. When the last reference to a disk resource is released, the disk resource is deleted. Our reference counting scheme is quite elaborate: it can be run in a debug mode that checks, amongst other things, that all references are ultimately released.

The story for mutable data is slightly more tricky. Most importantly, incremental merges (see functional requirement 2) are shared between tables and their duplicates. Continuing the analogy with Haskell's persistent data structures and sharing, we can view an incremental merge as similar to a thunk. Incremental merges only progress, which does not involve modifying data in place, and each table and its duplicates share the same progress. Special care is taken to ensure that incremental merges can progress concurrently without threads waiting much on other threads. The design here is to only perform actual I/O in batches, while tracking in memory how much I/O should be performed and when. The in-memory tracking data can be updated concurrently and relatively quickly, and ideally only one of the threads will do I/O from time to time. Garbage collection for shared incremental merges uses the same reference counting approach as garbage collection for immutable on-disk data.

### 3.6 Requirement 6

> It should have an extended interface that supports taking snapshots of tables. This should have $O(\log n)$ time complexity. The behaviour of this operation should be demonstrated using an appropriate test. For further details see Sections 7.5 and 7.6.

Snapshots can be created using `saveSnapshot` and later be opened using `openTableFromSnapshot`. Non-snapshotted contents of a table are lost when the table is closed (and thus in particular when its session is closed). Therefore, it is necessary to take a snapshot if one wants to drop access to a table and later regain it. When a snapshot is created, the library ensures durability of the snapshot by flushing to disk the contents and file system metadata of files and directories involved in the snapshot.

Creating a snapshot is relatively cheap: hard links to all the table-specific files are put in a dedicated snapshot directory, together with some metadata and a serialised write buffer. This way, most files are shared between active tables and snapshots, but this is okay since these files are immutable by design. Hard links are supported by all POSIX systems, and Windows supports them for NTFS file systems. The number of files in a table scales logarithmically with the number of physical key–value entries in the table. Hence, if the number of entries in a table is $n$, then taking a snapshot requires $O(\log n)$ disk operations.

Opening a snapshot is more costly because the contents of all files involved in the snapshot are validated (see functional requirement 8). Moreover, the state of ongoing merges is not stored in a snapshot but recomputed when the snapshot is opened, which takes additional time. Another downside of not storing merge state is that any sharing of in-memory data is lost when creating a snapshot and opening it later. A more sophisticated design could conceivably restore sharing, but we chose the current design for the sake of simplicity.

### 3.7 Requirement 7

> It should have an extended interface that supports merging monoidal tables. This should have $O(\log n)$ time complexity at the point it is used, on the assumption that it is used at most every $n$ steps. The behaviour of this operation should be demonstrated using an appropriate test. For further details see Section 7.7.

Since the term 'merge' is already part of the LSM-tree terminology, we chose to call this operation a table *union* instead. Moreover, 'union' is a more fitting name, since the behaviour of table union is similar to that of `Data.Map.unionWith`: all logical key–value pairs with unique keys are preserved, but pairs that have the same key are combined using the resolve function that is also used for upserts (see functional requirement 4). When the resolve function is `const`, table union behaves like `Data.Map.union`, which computes a left-biased union.

We make a distinction between *immediate* and *incremental* unions:

- *Immediate unions* (`union`, `unions`) perform all necessary computation immediately. This can be costly; in general, it requires that all key–value pairs of the involved tables are read from disk, combined using the resolve function and then written back to disk.
- *Incremental unions* (`incrementalUnion`, `incrementalUnions`) offer a way to spread out computation cost over time. Initially, an incremental union just combines the internal trees of the input tables to form a new tree for the output table. This is relatively cheap, as it requires mostly in-memory operations and induces only a constant amount of disk I/O. Afterwards, the user can repeatedly request a next computation batch to be performed until the whole computation is completed (`supplyUnionCredits`). It is up to the user to decide on the sizes of these batches, and this decision can be made based on the amount of work still to be done, which can be queried (`remainingUnionDebt`).

  The whole computation requires a linear amount of disk I/O, regardless of the batching strategy. However, usually not every read or write of a key–value pair requires I/O, since in most cases multiple key–value pairs fit into a single disk page and only a read or write of a disk page induces I/O.

Internally, immediate unions are implemented in terms of incremental unions.

Initially, performing a lookup in an incremental union table will cost roughly the same as performing a lookup in each of the input tables separately. However, as the computation of the union progresses, the internal tree is progressively merged into a single run, and finally a lookup induces only a constant amount of I/O. Since immediate unions are completed immediately, lookups in their results are immediately cheap.

In Cardano, unions should be used for combining tables of staking rewards at epoch boundaries. Because Cardano is a real-time system, latency spikes during normal operation must be prevented; so Cardano should use incremental unions for combining staking reward tables. When doing so, the computation of each union can be spread out over the course of the following epoch and be finished right before the next incremental union starts.

### 3.8 Requirement 8

> It should be able to run within the `io-sim` simulator, and do all disk I/O operations via the `fs-sim` simulation API (which may be extended as needed). It should be able to detect file data corruption upon startup/restoration. Detection of corruption during startup should be reported by an appropriate mechanism. During normal operation any I/O exceptions should be reported by an appropriate mechanism, but it need not detect 'silent' data corruption. The behaviour of this corrupted detection should be demonstrated using an appropriate test. For further details see Section 14.2.

The `lsm-tree` library supports both real I/O and I/O simulation, including file system simulation. Its public functions work with arbitrary `IO`-like monads and can therefore be used in `IO` as well as `IOSim` computations. When opening a session (`openSession`), the user provides a value of the `HasFS` record type from the `fs-api` package to specify the file system implementation to use. This way, the user can choose between the real file system and the file system simulation provided by the `fs-sim` package.

We made some smaller changes to `fs-api` and `fs-sim` to facilitate the development of `lsm-tree`. Furthermore, we created an extension to `HasFS` called `HasBlockIO`. It captures both the submission of batches of I/O, for example using `blockio-uring` [9], and some functionality unrelated to batching that is nonetheless useful for `lsm-tree`. The latter could eventually be included in `fs-api` and `fs-sim`.

In the context of `lsm-tree`, startup or restoration means opening a table from a table snapshot. In the consensus layer, table snapshots would be part of the ledger state snapshots that are created periodically. When a Cardano node starts up, the most recent, uncorrupted ledger state snapshot has to be restored, which requires checking table snapshots for corruption.

The technique for detecting whether a table snapshot is corrupted consists of two parts:

- Each snapshot includes a metadata file that captures the structure of the table's internal tree and stores the table's configuration options. Any mismatch between the metadata file and other files in the snapshot directory will result in an exception.
- All files in the snapshot, including the metadata file, have associated checksum files. Checksums for run files are computed incrementally; so their computation does not induce latency spikes. Like the other table files, also checksum files are not copied but referenced via hard links when a snapshot is created. When a snapshot is opened, the checksum for each file is recomputed and checked against the stored checksum. Any checksum mismatch will be communicated to the user using an exception.

Note that the success of this technique is independent of the nature of the I/O faults that caused a corruption: corruption from both noisy and silent I/O faults is detected.

Corruption detection upon snapshot opening is tested by the state machine tests using their random interleaving of table operations, which include an operation that creates a snapshot and corrupts it. However, there is also a dedicated test for corruption detection, which can be run using the following command:

```
cabal run lsm-tree-test -- -p prop_flipSnapshotBit
```

This test creates a snapshot of a randomly populated table and flips a random bit in one of the snapshot's files. Afterwards, it opens the snapshot and verifies that an exception is raised.

In addition to performing the corruption detection described above, the library does not further attempt to detect silent I/O faults, but it raises exceptions for all noisy I/O faults, that is, faults that are detected by the underlying software or hardware. To verify that this reporting of noisy I/O faults works properly, we implemented a variant of the state machine tests, which can be run using the following command:

```
cabal run lsm-tree-test -- -p prop_noSwallowedExceptions
```

This variant aggressively injects I/O faults through the `fs-sim` file system simulation and checks whether it can observe an exception for each injected fault.

A noble goal for the library would be to be fully exception-safe, ensuring that tables are left in a consistent state even after noisy I/O faults (assuming proper masking by the user where required). This was not a functional requirement, however, since in case of an exception the consensus layer will shut down anyway and thus drop all table data outside of snapshots. Nonetheless, we took some steps to ensure exception safety for the core library, but did not achieve it fully. We also extended the state machine tests with machinery that injects noisy I/O faults and checks whether the tables are left in a consistent state. If there should be a desire in the future to make the library fully exception-safe, then this machinery should help achieve this goal.

# 4 Performance requirements

This section outlines the performance requirements for the `lsm-tree` library and how they are satisfied. These requirements are described in the original requirements document [4, Sec. 18.3] and are reproduced in full below.

The summary of our performance results is that we can meet all the the targets: the *threshold*, *middle* and *stretch* targets.

1. We can exceed the *threshold* target using one core even on relatively weak hardware.
2. We can exceed the *middle* target using one core when using sufficiently capable hardware.
3. The *stretch* target is double the middle target, but two cores may be used to reach it. We can get a speedup of approximately 40 % with two cores, which on sufficiently capable hardware is enough to exceed the stretch target. On some higher end hardware we can meet the stretch target even with one core.

## 4.1 The requirements in detail

The on-disk backend should meet the following requirements, with the given assumptions.

1. Assume that the SSD hardware meets the minimum requirement for 4 k reads of 10 k IOPS at QD 1 and 100 k IOPS at QD 32 – as measured by the `fio` tool on the benchmark target system.
2. The performance should be evaluated by use of a benchmark with the following characteristics. The performance targets are specified in terms of this benchmark. The benchmark is intended to reasonably accurately reflect the UTxO workload, which is believed to be the most demanding individual workload within the overall design.

1. The benchmark should use the external interface of the disk backend, and no internal interfaces.
2. The benchmark should use a workload ratio of 1 insert, to 1 delete, to 1 lookup. This is the workload ratio for the UTxO. Thus the performance is to be evaluated on the combination of the operations, not on operations individually.
3. The benchmark should use 34 byte keys, and 60 byte values. This corresponds roughly to the UTxO.
4. The benchmark should use keys that are evenly spread through the key space, such as cryptographic hashes.
5. The benchmark should start with a table of 100 million entries. This corresponds to the stretch target for the UTxO size. This table may be pre-generated. The time to construct the table should not be included in the benchmark time.
6. The benchmark workload should ensure that all inserts are for 'fresh' keys, and that all lookups and deletes are for keys that are present. This is the typical workload for the UTxO.
7. It is acceptable to pre-generate the sequence of operations for the benchmark, or to take any other measure to exclude the cost of generating or reading the sequence of operations.
8. The benchmark should use the external interface of the disk backend to present batches of operations: a first batch consisting of 256 lookups, followed by a second batch consisting of 256 inserts plus 256 deletes. This corresponds to the UTxO workload using 64 kb blocks, with 512 byte txs with 2 inputs and 2 outputs.
9. The benchmark should be able to run in two modes, using the external interface of the disk backend in two ways: serially (in batches), or fully pipelined (in batches).

3. The *threshold* target for performance in this benchmark should be to achieve 5 k lookups, inserts and deletes per second, when using the benchmark in serial mode, while using 100 % of one CPU core.
4. The *middle* target for performance in this benchmark should be to achieve 50 k lookups, inserts and deletes per second, when using the benchmark in parallel mode, while using the equivalent of 100 % of a CPU core.
5. The *stretch* target for performance in this benchmark should be to achieve 100 k lookups, inserts and deletes per second – when using an SSD that can achieve 200 k IOPS at QD 32 – using the benchmark in parallel mode, while using the equivalent of 200 % of a CPU core. This target would demonstrate that the design can scale to higher throughput with more or faster hardware.
6. A benchmark should demonstrate that the performance characteristics of the monoidial update operation should be similar to that of the insert or delete operations, and substantially better than the combination of a lookup followed by an insert.
7. A benchmark should demonstrate that the memory use of a table with 10 M entries is within 100 Mb, and a 100 M entry table is within 1 Gb. This should be for key value sizes as in the primary benchmark (34 + 60 bytes).

## 4.2 Interpretion of the requirements

The performance requirements are specified in terms of one primary benchmark and one secondary benchmark. The primary benchmark is used to demonstrate overall throughput (items 3–5) and memory use (item 7), while the secondary benchmark is used to demonstrate that the upsert operation has the expected performance characteristics (item 6).

The primary benchmark is designed to be reasonably close to the expected UTxO workload. Throughput is specified in terms of batches of operations that match the UTxO workload for large blocks with many small transactions. The requirements contain three targets:

- A *threshold* target: 5 k lookups, inserts and deletes per second, using one CPU core
- A *middle* target: 50 k lookups, inserts and deletes per second, using one CPU core
- A *stretch* target: 100 k lookups, inserts and deletes per second, using two CPU cores

The requirements document does not specify the hardware except for the SSD, which must be capable of performing random reads with at least 10 k IOPS at queue depth 1 and at least 100 k IOPS at queue depth 32, as measured by the `fio` I/O benchmarking tool. We have interpreted this specification such that it refers to `fio` running on one core and using direct I/O, where the latter ensures that we measure actual SSD hardware performance. The subsection *SSD benchmarking script* in the appendix shows the `fio` script that we have used for determining SSD performance.

All our performance results are for Linux only. This is not a deficiency, since the performance requirements concern only Linux, while the functional requirements concern Linux, Windows and macOS. The `lsm-tree` library reflects this difference in that it supports parallel I/O only for Linux (employing `io_uring`) and always uses serial I/O on other platforms. This limits those other platforms to the QD 1 SSD performance, which is typically no better than 10 k IOPS. The code is modular, so that parallel I/O backends for other platforms could be implemented in the future if they were required.

The `lsm-tree` library is designed to work well without disk caching, but it does support disk caching as an option. The main performance results we present here do not rely on disk caching and are achieved with no more than 1 GiB of memory. We also present performance results with disk caching enabled. The benchmark workload has no temporal or spatial locality and therefore benefits very little from caching, except when the memory is large enough to hold at least most of the database. The 100 M entry table takes around 9.4 GiB on disk. Thus, for machines that have more than around 12 GiB of RAM, we can expect the whole database to fit in memory, so that we can expect a performance boost. There may also be a performance benefit for the machines with 8 GiB or less, but there is also a cost when cache thrashing occurs.

## 4.3 The benchmark machines

Since the requirements do not mention specific machines to be used for the benchmarks, we have employed a variety of setups, covering weaker and stronger machines. We have also included several low-end machines that fall below the 100 k IOPS requirement. These are useful to illustrate the performance on weaker hardware but obviously cannot demonstrate meeting the middle or stretch targets. We have tried as far as practical to use standard setups that others could use to reproduce the results. Our collection of machines consists of a selection of AWS EC2 cloud instances, a Raspberry Pi 5 and a relatively high-performance, bare-metal laptop machine. Concretely, we have used the following systems, listed here from low-end to high-end:

| Machine | CPU | CPU year | AWS vCPUs | Threads | Cores | RAM (GiB) | Rated IOPS |
|---|---|---|---|---|---|---|---|
| RPi5 | aarch64 | 2023 | – | 4 | 4 | 8.00 | 50,000 |
| m6gd.medium | aarch64 | 2019 | 1 | 1 | 1 | 4.00 | 13,400 |
| m5d.large | x86-64 | 2017 | 2 | 2 | 1 | 8.00 | 30,000 |
| i3.large | x86-64 | 2016 | 2 | 2 | 1 | 15.25 | 103,000 |
| i7i.xlarge | x86-64 | 2023 | 4 | 4 | 2 | 32.00 | 150,000 |
| i8g.xlarge | aarch64 | 2024 | 4 | 4 | 4 | 32.00 | 150,000 |
| dev laptop | x86-64 | 2023 | – | 16 | 8 | 64.00 | 1,400,000 |

The RPi5 is a standard Raspberry Pi 5, with an official NVMe HAT and SSD. The dev laptop is a laptop belonging to one of the developers, with an AMD Ryzen 7 8845HS CPU and a Samsung 990 PRO NVMe SSD.

Full specifications of the AWS instances are available from the AWS EC2 website. Note, however, that AWS uses a 'vCPU' nomenclature in its specifications. For their x86 machines, a vCPU corresponds to a hyperthread of a physical core, while, for their ARM machines, a vCPU corresponds to a physical core. When determining parallel speedup in the context of the stretch target, we are only interested in multiple physical cores, not hyperthreads. This makes the m6gd.medium, m5d.large and i3.large instances irrelevant with respect to the stretch target.

Finally, note that the first three of the machines listed in the table above have SSDs that are not capable of 100 k IOPS.

#### 4.3.1 Micro-benchmarks of the benchmark machines

To help evaluate the `lsm-tree` benchmark results across these different machines, it is useful to have a rough sense of their CPU and I/O performance. Therefore, we have determined the machines' scores according to standard performance benchmarks:

- For CPU performance, we have used a simple `sysbench` benchmark:

  ```
  sysbench cpu --cpu-max-prime=20000 --time=0 --events=10000 run
  ```

  We report the `sysbench` CPU results as events per second, with more being better.

- For I/O performance, we have used the `fio` benchmark tool with the configuration detailed in the appendix in the subsection *SSD benchmarking script*, measuring random 4 KiB reads at QD 32. For submitting the I/O requests, we have used one core as well as two cores, the latter only for machines with two or more *physical* cores. Note that the two-core results we report are aggregates across both cores, not per-core results.

Furthermore, we have determined the results of a couple of micro-benchmarks created as part of the project:

- We have a Bloom filter benchmark[3] that measures the time to perform a fixed number of lookups in Bloom filters whose sizes resemble those that will occur with the UTxO workload. This is primarily a memory benchmark, being highly dependent on memory latency, throughput and the number of memory reads that the CPU can issue in parallel from a single core using software prefetching. The performance of `lsm-tree` lookups is highly dependent on the performance of the Bloom filters.

[3]This is the `lsm-tree-bench-bloomfilter` benchmark. Use `cabal run lsm-tree-bench-bloomfilter` to run it yourself.

- We also have a Haskell version of the `fio` benchmark, which uses the `blockio-uring` library that `lsm-tree` employs. The results of this benchmark typically follow the `fio` numbers quite closely, demonstrating that there is little performance loss from using the high level Haskell I/O API provided by `blockio-uring` versus using a low-level C API. We report the 1 core results below, but this benchmark also scales to multiple cores and gets results close to those for `fio` for the same number of cores.

#### 4.3.2 Micro-benchmark results

The results of all these benchmarks are as follows:

| Machine name | `sysbench` (events/sec) | Bloom filter (seconds) | `fio`, 1 core (IOPS) | `fio`, 2 cores (IOPS) | `blockio-uring`, 1 core (IOPS) |
|---|---|---|---|---|---|
| RPi5 | 1,038 | 31.33 | 77,600 | 77,900 | 77,191 |
| m6gd.medium | 1,075 | 25.77 | 14,900 | – | 14,171 |
| m5d.large | 414 | 23.12 | 33,900 | – | 33,919 |
| i3.large | 901 | 20.99 | 170,000 | – | 170,945 |
| i7i.xlarge | 1,153 | 12.07 | 351,000 | 210,000 | 352,387 |
| i8g.xlarge | 1,249 | 18.78 | 351,000 | 210,000 | 351,574 |
| dev laptop | 2,134 | 7.79 | 261,970 | 487,000 | 275,008 |

Note that the RPi5, i7i.xlarge and i8g.xlarge all produce I/O results that are above their rated IOPS. In the case of the RPi5, this is due to the rated IOPS being specified for PCIe 2.0, while in practice the RPi5 is capable of PCI 3.0, which increases the available bandwidth.

The i7i.xlarge and i8g.xlarge AWS EC2 instances exhibit somewhat strange behaviour:

**Their IOPS scores are much higher than advertised.** This can be explained by the fact that these machines are virtual and use shared hardware, so that one can presume that their rated IOPS are guaranteed minimums that can be exceeded when the physical machines' IOPS are under-utilised.

**Their I/O is bursty.** The bursty behaviour is that initial `fio` runs achieve even higher IOPS scores (500 k and 450 k, respectively, on one core), but after several runs the scores settle down to the numbers reported in the table above. This behaviour is documented by AWS itself: it is due to disk I/O 'credits' that allow short bursts at higher performance, which cannot be sustained.

**The IOPS scores scale negatively when adding more cores.** This is actually quite bizarre: the more cores used to submit I/O, the lower the aggregate IOPS that is measured. Apparently, this is not the result of a measurement artefact but shows a real effect, and it is *opposite* to what happens with physical hardware. Running `fio` on the i8g.xlarge machine with 4 cores results in 175 k IOPS (which is near to the rated 150 k IOPS), showing that the negative scaling continues beyond two cores. One can but speculate as to the reason for this behaviour. It is probably an artefact of the way the Nitro hypervisor limits IOPS on the VMs, but it is unclear why it would allow exceeding the minimum rated IOPS by a greater proportion when submitting I/O from fewer cores.

**The IOPS scores of i7i.xlarge and i8g.xlarge are the same.** This can be explained by the fact that both machines use the same generation of the Nitro hypervisor and I/O hardware.

These behaviours are worth keeping in mind for deployment. In particular, a benchmark may give too rosy a picture if the physical machine was not saturated at the time the benchmark

was run, or the benchmark was not run long enough, or the benchmark does not use the same amount of I/O parallelism as the deployment.

Finally, the dev laptop's `fio` scores of 262 k IOPS on 1 core and 487 k on 2 cores are well below the laptop's rated 1.4 M IOPS. The explanation is that the 1.4 M is for the SSD device as a whole and it takes multiple cores to saturate the SSD, while the benchmark numbers above are for a single core. On the same machine, the scores of the `fio` and `blockio-uring` benchmarks for 8 cores are 1,449 k and 1,337 k IOPS respectively.

## 4.4 Setup of the primary benchmark

The primary benchmark handles the required UTxO-like workload. Its starting point is a table with 100 million key–value pairs, with keys and values having the required sizes and keys being uniformly distributed. The benchmark reads this table from a snapshot and executes a sequence of batches of operations on it. Each batch consists of 256 lookups, 256 inserts and 256 deletes. The benchmark measures the time to execute the sequence of batches of operations (but not the time to initially create the table, nor the time for opening the snapshot). To achieve reasonable running times, we use 10 k batches of operations for the serial mode and 100 k batches for the pipelined mode.

### 4.4.1 Serial and pipelined execution

There are two execution modes that the specification calls for and that the primary benchmark provides: serial and fully pipelined. In the serial mode, we execute the batches sequentially and within each batch perform the lookups before the updates. In the pipelined mode, we do not impose such strict ordering, to overlap I/O with CPU work and to take advantage of multiple cores.

Note that both modes use parallel I/O (on Linux). The serial or parallel modes of the benchmark refers to the sequencing of the API operations on the database, not to disk I/O operations.

The importance of the pipelined mode in the benchmark – and the reason it is included in the project requirements [4, Sec. 16.2] – is that the benchmark corresponds to a (slightly) simplified version of the UTxO workload, and extracting parallelism in the real UTxO workload is important for the `cardano-node` to achieve higher peformance targets. The implementation of pipelined execution in this benchmark can in fact be considered a prototype of a pipelined version of the UTxO workload.

While the performance requirements demand that the benchmark is capable of pipelined execution, they also specify an order for the operations within each batch, which seems contradictory. The solution is to interpret the specification of operation order not physically but logically: as merely stating that the lookup results and the output content of the table must match those that are achieved when executing the operations in the specified order. The benchmark then conforms to the specification by adhering to the following dataflow constraints:

- The output table content of each batch is used as the input table content of the next batch.
- The lookups of each batch are performed on the input table content of the batch.
- The updates of each batch are performed on the input table content of the batch and their final result is used as the output table content of the batch.

These constraints leave room for concurrent and parallel execution, because they allow overlapped execution of multiple batches in a pipelined way. The reason why such an execution is possible is somewhat subtle though. We provide a high level summary here. For a full formal treatment see our previous work [3, Secs. 7, 7.5].

The updates have to be executed serially, but the lookups can be executed out of order, provided the results we ultimately report for the lookups are correct. The trick is to perform the lookups using an older value of the database and then adjust their results using the updates from the later batches. This allows starting the lookups earlier and thus having multiple lookups not only overlapping with each other but also with updates.

As an illustration, the following figure depicts such pipelined execution and its dataflow for the case of four concurrent pipeline stages, achieved using two cores with two threads running on each. The bars represent threads doing work over time. The blue portions of the bars represents threads doing CPU work, while the green portions represents threads waiting on I/O to complete. The key observation from this diagram is that multiple cores can be submitting and waiting on I/O concurrently in a staggered way. One can also observe that there is the opportunity on a single core to overlap CPU work with waiting on I/O to complete. Note however that this diagram is theoretical: it shows the opportunity for concurrency given the data flows and plausible timings. It does not show actual relative timings.

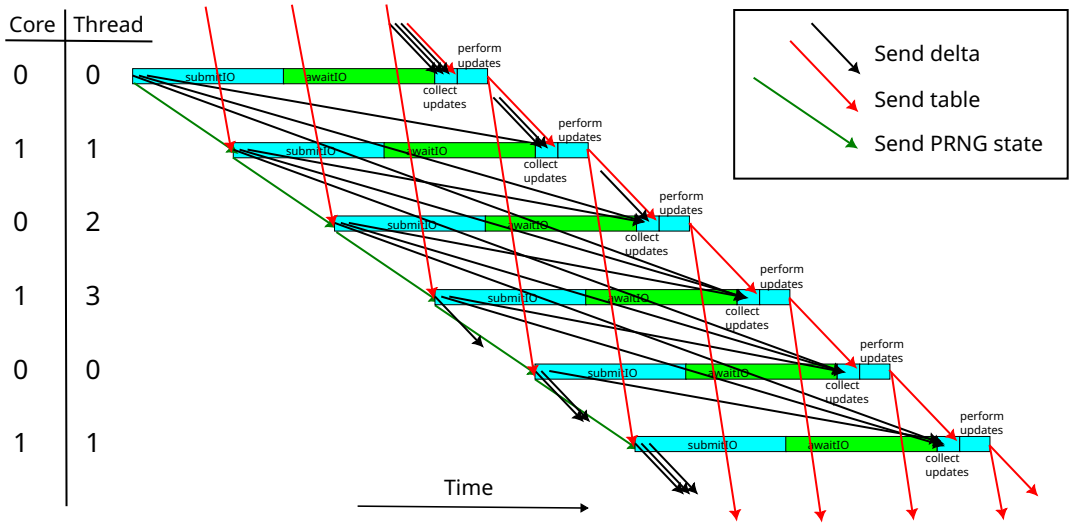


Figure 1: Concurrency in the pipelined benchmark mode

Note that this scheme of pipelined execution can be conveniently implemented with multiple independent handles to a table, a feature that the `lsm-tree` library provides. By using different handles, we can isolate lookups from subsequent updates while still allowing parallel execution. Traditional transactional databases also support transaction isolation but require additional synchronisation for this. We elaborate on this topic in the appendix in the subsection *Functional persistence*.

The advanced design of the pipelined mode makes its correctness less obvious than the correctness of the straightforward serial mode. Therefore, we compare the lookup results of the benchmark in the pipelined mode with the corresponding lookup results of a reference implementation. The benchmark has a special checking mode to perform this comparison.

### 4.4.2 Table configuration

The `lsm-tree` library has a number of table configuration parameters that directly affect performance. For the primary benchmark we use the following settings:

**No disk caching.** This ensures that we do not use excessive amounts of memory but stay within the specified memory limit of 1 GiB.

**Bloom filters with a false positive rate (FPR) of** $1/1000$**.** This results in nearly 16 bits per key.

The memory limit of 1 GiB is relatively generous; so we can use quite large Bloom filters to minimise unnecessary I/O and thus maximise performance. If the memory constraint was tighter, we would use smaller Bloom filters with a correspondingly higher FRP, losing some performance this way.

**Indexes of the compact type.** This takes advantage of the keys being roughly uniformly distributed. It reduces the size of the in-memory indexes, leaving more memory available for the Bloom filters and the write buffer.

**Write buffer size of 20,000 elements.** This is a reasonable size for a 100 M entry table and the memory budget of the benchmark.

### 4.5 Results of the primary benchmark

The following table shows the results of the primary benchmark run in serial mode with 10 k batches of operations, along with related data:

| Machine | `fio`, 1-core (IOPS) | Primary benchmark (ops/sec) | Lookup only (ops/sec) | Primary benchmark (MiB) |
|---|---|---|---|---|
| RPi5 | 77,600 | 25,458 | 62,357 | 865 |
| m6gd.medium | 14,900 | 10,628 | 13,320 | 634 |
| m5d.large | 33,900 | 22,553 | 30,039 | 860 |
| i3.large | 170,000 | 42,158 | 101,861 | 865 |
| i7i.xlarge | 351,000 | 95,907 | 157,560 | 944 |
| i8g.xlarge | 351,000 | 58,838 | 159,103 | 949 |
| dev laptop | 261,970 | 86,527 | 156,452 | 940 |

The data in this table is to be interpreted as follows:

**`fio` (IOPS).** This is the one-core `fio` I/O score, repeated here to help interpret the benchmark results.

**Primary benchmark (ops/sec).** This is the number of operations per second achieved by the primary benchmark.

**Lookup only (ops/sec).** This is the analogous result of a benchmark variant that only performs the lookups, not the inserts and deletes. It is here to indicate the read-only performance.

**Primary benchmark (MiB).** This is the peak amount of memory used by the primary benchmark. Concretely, it is the maximum resident set size (RSS) queried from the Linux kernel using GNU Time. This value does not cover page cache memory used by the operating system on behalf of the application. However, the primary benchmark makes only little use of the page cache: it employs it only for streaming reads and writes of run files, not for actual caching. Unlike the timing, the peak memory measure does include the loading the snapshot, though in practice this makes little difference.

#### 4.5.1 Meeting the middle target

Recall that for the middle target the machine is expected to support at least 100 k IOPS as reported by `fio`. Thus the results for the first three machines are interesting to gauge the performance that can be expected from cheaper machines but do not help establish if the middle target has been met.

The i3.large instance does have the required I/O performance but does not quite reach the target of 50,000 ops/sec. This is almost certainly due to its relatively weak CPU: it has the oldest

CPU of all the benchmark machines (from 2016), has a relatively low score on the `sysbench` CPU benchmark and has a comparatively poor Bloom filter timing. All the other benchmark machines with the required I/O performance easily surpass the target of 50,000 ops/sec.

The specification of the middle target calls for using the benchmark in parallel mode. While we do use parallel I/O, we do not need to use the parallel pipelined mode to meet the middle target. Nevertheless, the subset of machines that meet the 50k ops/sec target in serial mode, also meet the same target in parallel mode on one core. The results for this are presented in the section below on *meeting the stretch target*, where we discuss the parallel pipelined mode in more detail.

### 4.5.2 Meeting the threshold target

The specification does not make it entirely clear under what conditions the threshold target is supposed to be met. A reasonable intepretation is that the conditions are the same as for the middle target. Under this assumption, we do meet the threshold target since we meet the middle target, as established above. An alternative interpretation is that the threshold target is to be met using only serial I/O on hardware that is only capable of 10 k IOPS with queue depth 1. Trying to reach the target under these conditions is obviously more ambitious, but it is interesting since it gives an indication of performance on more constrained systems and on non-Linux systems, where parallel I/O is not used. We can demonstrate that the target is met even in this more demanding setting.

In order to furnish this proof, we have to run the benchmark using only serial I/O. The `lsm-tree` library uses a library `blockio` that has two implementations: one based on `blockio-uring`, which can only be used on Linux, and one that employs serial I/O. While the latter is intended for non-Linux systems, it can be used on Linux as well. We make use of this possibility in order to run the benchmark on Linux under the above-described tighter constraints.

On the Raspberry Pi 5, with serial I/O (QD 1), the `fio` benchmark achieves 9,897 IOPS, which is less than the 10 k assumed, and the primary benchmark nevertheless achieves 6,758 operations per second, which is more than the 5 k required for the threshold target. Thus we can reasonably claim that we can achieve the threshold performance on the weakest plausible hardware.

### 4.5.3 Meeting the stretch target

The stretch target is 100,000 ops/sec, and it has to be achievable on a machine with 200 k IOPS using two cores.

In connection with this target, we have run the benchmark in the pipelined mode on one and on two cores as well as in the serial mode, using those benchmark machines that have more than one *physical* core. For these benchmark runs, we have used 100 k batches of operations instead of 10 k, this way achieving longer running times. The following table lists the benchmarking results, along with the speedups of the pipelined runs on two cores relative to the corresponding serial runs:

| Machine | Serial | Pipelined, 1 core | Pipelined, 2 cores | Speedup |
|---|---:|---:|---:|---:|
| RPi5 | 27,743 | 31,341 | 44,614 | 60.8 % |
| i7i.xlarge | 114,723 | 109,228 | 130,529 | 13.7 % |
| i8g.xlarge | 72,390 | 65,820 | 101,445 | 40.1 % |
| dev laptop | 89,593 | 80,860 | 125,955 | 40.6 % |

Comparing the data for the serial mode with the corresponding data from the previous table shows that using longer running times improves the results for the serial case somewhat.

As can be seen, the pipelined version, when running on one core, is slower than the serial version. This is expected and mainly due to the following reasons:

- The pipelined version has to do more work, because it has to modify the lookup results according to the updates from later batches.
- Running the benchmark in the pipelined mode involves thread synchronisation.
- For the pipelined mode, we turn off merge batching. Merge batching improves performance in the serial mode. In the pipelining mode, however, it causes an imbalance across the workloads of the different cores and so we turn it off. One can consider merge batching as a single-threaded optimisation. The pipelined mode on one core doesn't get the benefit of that optimisation

Running the pipelined version on two cores instead of one takes us over the stretch target of 100 k operations per second on several of the benchmark machines, though the i7i.xlarge machine exceeds this target already on one core.

Comparing the results of running on two cores to the results of running in serial mode, we observe quite a wide range of speedups across the different benchmark machines. We can speculate as to what causes the substantial differences between machines. The most important issue is likely what the limiting resource is in the one-core case, CPU or SSD, and how that changes for the two-core case.

- The RPi5 is probably CPU-limited, since its `fio` score of 77 k IOPS is well above the number of operations per second in serial mode. We know that the RPi5's IOPS do not scale when adding more cores. So a second core simply improves the limiting resource: the CPU. This yields a substantial parallel speedup of 60 %.
- The i7i.xlarge machine has excellent performance with one core, exceeding the 100 k target already in this setting. We know it has higher one-core performance than i8g.xlarge, with an advantage of approximately 40 % in the Bloom filter micro-benchmark. Nevertheless, it is probably limited by CPU, not by SSD, since its one-core IOPS value is so high (350 k). We know from the subsection *micro-benchmark results* that this machine's IOPS value scales *negatively* when going to two cores, from 350 k down to 210 k aggregated across both cores. This is probably the cause of its poor speedup: the machine goes from being limited by CPU to being limited by SSD.
- The i8g.xlarge machine is clearly limited by CPU in the one-core case. Adding a second core improves its CPU performance substantially but does not push the scores so high that it starts to be limited by SSD as i7i.xlarge is.
- The dev laptop also exhibits good speedup. This machine's `fio` IOPS value scales by 1.85x when going from one to two cores, so the balance of CPU and I/O resources remains roughly the same, and thus we expect no significant change in the limiting resource.

The pipelining approach can profit in principle from further increasing the number of cores, so that for some applications it may be worth using more cores than just two. If an application performs a considerable amount of CPU work that can be interleaved with the database operations then using more cores can improve performance even more. With the real UTxO workload, we are in this situation, of course, because there is transaction validation work to do.

#### 4.5.4 Meeting the memory targets

Item 7 of the performance requirements states the following:

> A benchmark should demonstrate that the memory use of a table with 10 M entries is within 100 Mb, and a 100 M entry table is within 1 Gb. This should be for key value sizes as in the primary benchmark (34 + 60 bytes).

The benchmark results shown at the beginning of the section *Results of the primary benchmark* are for a database table with 100 M entries. They contain the amount of memory for each run, which is the maximum RSS as reported by the operating system. We can see from the listed values that all the benchmark runs use less than 1 GiB (1024 MiB).

For the target of a 100 MiB maximum when using a 10 M entry table, we run the same benchmark with a slightly different configuration:

- Of course, we use a smaller table, one with an initial size of 10 M entries.
- We scale down the size of the write buffer correspondingly, from 20 k entries to 2 k entries.
- We tell the GHC RTS to limit its heap size to 100 MiB, using `+RTS -M100m`.

With this configuration, the maximum RSS is reported as 85,220 KiB (83.2 MiB), which is less than the target of 100 MiB.

When reproducing this result, one minor trap to avoid is to run the benchmark using `cabal run` instead of launching its executable directly. The problem with the former is that GNU Time reports the largest RSS of any subprocess, which may turn out to be the `cabal` process and not the benchmark process.

By the way, we also get excellent speed with the 10 M entry table: 150 k ops/sec, much more than the circa 86 k ops/sec we get with the 100 M entry table. The reason is that for smaller tables there is less work to do generally: less merging work, fewer runs, fewer Bloom filters.

### 4.6 The upsert benchmarks

Item 6 of the performance requirements states the following:

> A benchmark should demonstrate that the performance characteristics of the monoidial update operation should be similar to that of the insert or delete operations, and substantially better than the combination of a lookup followed by an insert.

As already mentioned in the discussion on functional requirement 4, the `lsm-tree` library and its documentation now use the term 'upsert' for this monoidal update operation, to follow standard database terminology.

In line with the above requirement, we have created the following benchmarks:

- A benchmark of the time to insert a large number of key–value pairs using the insert operation and a benchmark of the time to insert the same key–value pairs using the upsert operation
- A benchmark of the time to repeatedly upsert values of certain keys and a benchmark of the time to repeatedly emulate an upsert the values of these keys by looking up their current values, modifying them and writing them back

The benchmarks use the following parameters:

- 64 bit as the size of keys and values
- 80,000 elements (generated using a PRNG)
- Addition as the upsert combining operation
- 250 operations per batch

- No disk caching
- 1,000 elements as the write buffer capacity
- 10 upserts per key in case of the second two benchmarks

The benchmarks are implemented using Criterion, which performs multiple benchmark runs and combines the results in a sound statistical manner. For our benchmarks, the variance of the results across the different runs, as reported by Criterion, is relatively low. We have executed the benchmarks on the dev laptop machine. However, the absolute running times of these benchmarks is of little interest; the interesting point is the relative timings.

The result are as follows:

- The difference in running time between the insert and the corresponding upsert benchmark is less than 0.4 % (932.8 ms vs. 929.4 ms), so that insert and upsert performance clearly qualify as 'similar'.
- Using the combination of lookup and insert takes 2.4 times as long as using upsert (2.857 s vs. 1.188 s). We can thus reasonably conclude that the performance of upsert is 'substantially better' than the performance of a lookup followed by an insert.

### 4.7 Reproducing the results

We encourage interested readers to reproduce the benchmark results for themselves. To that end we have mostly used benchmark machines which are available to all. It is of course also informative to run these benchmarks (and the micro-benchmarks like `fio` and `sysbench`) on the hardware where the library is intended to be used.

The version of the library that is used as the basis for this report is tagged `final-report` in the `lsm-tree` Git repository [2]. It can be checked out using the following commands:

```
git clone git@github.com:IntersectMBO/lsm-tree.git
cd lsm-tree
git checkout final-report
```

The primary benchmark's code is in the repository in `bench/macro/utxo-bench.hs`. It can be executed using commands of the following shape:

```
cabal run utxo-bench -- [subcommand] [options]
```

For the full command line help use the following commands:

- Global help:

  ```
  cabal run utxo-bench -- --help
  ```

- Help on the `setup` subcommand:

  ```
  cabal run utxo-bench -- setup --help
  ```

- Help on the `run` subcommand:

  ```
  cabal run utxo-bench -- run --help
  ```

The execution of the benchmark is in two phases:

- During the *setup* phase, the initial database is prepared. The user can decide, amongst other things, on the number of database entries (default: 100 M) and the Bloom filter FPR (default: 1/1000). After preparation, the database can be used by multiple benchmark runs, picking different values for the remaining parameters.

- During the *run* phase, the actual benchmark is performed based on the database created during the setup phase. The user can decide, amongst other things, on the number of batches, the disk cache policy and the benchmark mode: serial (the default), pipelined or lookup only.

The checking mode, which compares the lookup results of each batch against the corresponding results of the reference implementation, can be activated using the `--check` option. One can inspect the code of the reference implementation to assure oneself that it is correct and then rely on the checking mode for assurance that the actual implementation is correct. This is important in general but especially so for the pipelined implementation, which is non-trivial.

## References


[1] D. Coutts and D. Wilson, "Storing the Cardano ledger state on disk: analysis and design options," 2021. Available: https://github.com/IntersectMBO/lsm-tree/blob/final-report/doc/final-report/references/utxo-db.pdf

[2] *lsm-tree*. Available: https://github.com/IntersectMBO/lsm-tree/

[3] D. Coutts, J. Dral, and D. Wilson, "Storing the Cardano ledger state on disk: API design concepts," 2021. Available: https://github.com/IntersectMBO/lsm-tree/blob/final-report/doc/final-report/references/utxo-db-api.pdf

[4] D. Coutts, "Storing the Cardano ledger state on disk: requirements for high performance backend," 2023. Available: https://github.com/IntersectMBO/lsm-tree/blob/final-report/doc/final-report/references/utxo-db-lsm.pdf

[5] D. Coutts, J. Dral, and W. Jeltsch, "Storing the Cardano ledger state on disk: integration notes for high performance backend," 2025.

[6] "`lsm-tree` API documentation." [Online]. Available: https://intersectmbo.github.io/lsm-tree/index.html

[7] "`lsm-tree` package description." [Online]. Available: https://github.com/IntersectMBO/lsm-tree/blob/final-report/lsm-tree.cabal

[8] "`lsm-tree` prototype." [Online]. Available: https://github.com/IntersectMBO/lsm-tree/blob/final-report/prototypes/ScheduledMerges.hs

[9] *blockio-uring*. Available: https://github.com/well-typed/blockio-uring

[10] "`lsm-tree` format specifications." [Online]. Available: https://github.com/IntersectMBO/lsm-tree/tree/final-report/doc

[11] *ouroboros-consensus*. Available: https://github.com/IntersectMBO/ouroboros-consensus/commit/9d41590555954c511d5f81682ccf7bc963659708

[12] *ouroboros-consensus LedgerSeq module*. Available: https://github.com/IntersectMBO/ouroboros-consensus/blob/9d41590555954c511d5f81682ccf7bc963659708/ouroboros-consensus/src/ouroboros-consensus/Ouroboros/Consensus/Storage/LedgerDB/V2/LedgerSeq.hs#L72-L96

[13] *ouroboros-consensus InMemory module*. Available: https://github.com/IntersectMBO/ouroboros-consensus/blob/9d41590555954c511d5f81682ccf7bc963659708/ouroboros-consensus/src/ouroboros-consensus/Ouroboros/Consensus/Storage/LedgerDB/V2/InMemory.hs

# Appendix

## SSD benchmarking script

To assess the performance of the SSDs used in the `lsm-tree` benchmarks, we have employed the following `fio` script to measure the IOPS for random 4 KiB reads at queue depth 32:

```
; Read a single file with io_uring at different depths
[global]
ioengine=io_uring
direct=1
rw=randread
size=1024m
directory=benchmark

[benchfile]
iodepth=96
iodepth_batch_submit=32
iodepth_low=64
bs=4k
```

## Functional persistence

In the subsubsection *Serial and pipelined execution*, we described how to implement a form of pipelining using the `lsm-tree` package and briefly mentioned that implementing such pipelining using traditional databases is only possible with additional synchronization. In this subsection, we elaborate on this.

Suppose we have the following two transactions:

- Transaction A, which performs only lookups and thus has no side effects
- Transaction B, which performs updates

Suppose further that we want both transactions to start with the same database state. Our goal is to gain speedup from parallelisation as far as possible.

We can obviously get the correct result by executing A and B in sequence, since A has no side effects, but we would not gain benefit from parallelisation this way. Note that we cannot execute them in reverse order, first B and then A, since in this case A could be run with a wrong database state. When using a traditional transactional database system capable of full ('serialisable') isolation, executing the two transactions concurrently would produce the same result as either executing A before B or executing B before A, hopefully with some parallel speedup. Given that we can only accept the result of the former, such concurrent execution would not help us.

Also with a traditional database system, it is possible, albeit non-trivial, to run two transactions concurrently and have them use the same initial state. The solution is to start both transactions (for example, using the SQL command `BEGIN TRANSACTION`) and then synchronize the threads or connections that issue the transactions, to ensure that both transactions have been started before the bodies of the transactions are processed. Such synchronisation can be done by the application directly or by employing non-standard server functionality, as present for example in recent versions of PostgreSQL.

By contrast, with functional persistence using explicit database handles, implementing the desired behaviour is straightforward. We generate two independent handles based on the same initial database state and then let one thread execute A using one of the handles and

another thread execute B using the other handle. This is not only simpler but also involves less synchronisation, since synchronization has to be only one-way rather than two-way.