Storing the Cardano ledger state on disk: analysis and design options

AN IOHK TECHNICAL REPORT

Douglas Wilson
douglas@well-typed.com

Duncan Coutts
duncan@well-typed.com
duncan.coutts@iohk.io

Version 1.0, April 2021

Changelog

1.0 Duncan Coutts, Douglas Wilson (April 2021) — Final version

1 Introduction

The project is intended to solve the following problem: the Cardano node keeps its ledger state within memory (RAM) and as Cardano scales up this will not be sustainable because the ledger state will eventually grow too big.

The solution that this project is focused on is to move the bulk of the ledger state from memory to disk. This will involved developing and then integrating new infrastructure in the Cardano node (ledger and consensus layers) to allow large parts of the ledger state to be kept on disk rather than in memory.

This document is intended to capture the business and technical requirements, to present a small set of candidate solutions, and to evaluate and justify a preferred solution among those candidates. That solution should be justified in complexity by the captured requirements.

Contents

2 Summary

Memory (RAM) is very fast but memory space is relatively small and expensive. Disks – even SSDs – are relatively slow, but disk space is relatively plentiful and cheap.

Thus the challenge when adapting a design from memory to disk is to maintain adequate performance. For blockchains generally, and Cardano specifically, this is not a trivial problem. For example, for a long time Ethereum node performance was bottlenecked on disk I/O performance. We must consider performance in the design analysis or we will face the same problem.

Best case estimates indicate that the stretch goal of 200 TPS is *very* hard to achieve if one also wants the time to synchronise the chain to be reasonable (e.g. the first time Daedalus is used). Even threshold or mid targets of 20 and 50 TPS will be a challenge, while keeping sync times reasonable.

The reason for this is clear: if we were to aim for syncing being 1000 times faster than real time, then a year's worth of chain data would take just under 9 hours to validate. End users would have to wait 9 hours for each year that the chain had been operating at this rate. Arguably, even this is unreasonably slow, and yet note that this already requires syncing to be 1000 times faster. So if our target were 200 TPS, then the requirement for syncing would be 200,000 TPS. It is intuitively clear that 20,000 or 200,000 TPS is a hard target indeed.

Overall this points to the next major scaling bottleneck being synchronisation performance. Though it is out of scope for this project, it will likely be worth developing solutions that do not require all nodes (e.g. Daedalus nodes) to download and validate the entire chain.

2.1 Example estimate

The simplified optimistic estimate is as follows:

- Assume the 200 TPS stretch goal.
- Assume the typical 2 inputs and 2 outputs per tx.
- Assume the UTxO is the only part of the ledger state of interest. This is a simplifying assumption. In reality there are other parts of the ledger state that will make these estimates worse.
- Assume the UTxO mostly does not fit in memory.
- Assume the UTxO read access pattern has poor temporal and physical locality, and thus each lookup will typically require at least one "random" disk read.
- Assume that writes to the database are able to be efficiently dispatched in batches such that they are insignificant in cost relative to reads. Note that this is a rosy assumption that may be mostly true for LSM trees, but would not be true for many other data structures (e.g. B+ trees)
- Assume a DB read amplification factor of 1.5. (This is a rosy assumption.)
- Thus 200 TPS translates to $200 \times 2 \times 1.5 = 600$ disk I/O operations per second (IOPS).
- Assume a mid-range SSD rated at 10,000 IOPS for random reads at queue depth 1, and 100,000 IOPS for random reads at queue depth 32.
- Assume that we can fully utilise the parallel performance of the SSD, i.e. use queue depth 32.

- The ratio of SSD performance IOPS (100,000) to live system IOPS (600) gives the sync speed ratio, i.e. the factor that syncing the chain would be compared to real time. This is a factor of $100,000/600 = 167$ in our example.

- Thus for a chain growing at 200 TPS for a year, the time to sync that chain would be 1/167 of a year, which is 52.5 hours, more than two days.

2.2 Crucial factors

As we can see in the estimate above, the chain sync times are not reasonable. The example illustrates that the crucial factors are:

1. The target TPS
2. The read amplification factor
3. a database that can efficiently batch writes
4. The SSD random read performance

Thus if we reduce the target TPS by a factor of 10, we reduce the sync time bound correspondingly. With 20 rather than 200 TPS we could expect at best 5 hours of syncing to catch up a year of the chain.

A read amplification factor of only 1.5 is itself a challenge, and requires a good database choice.

Modern SSDs IOPS for random read range from 100,000 to 1,000,000 for the extreme end of the consumer market. Achieving these levels of performance requires utilising parallel I/O. If only serial I/O is used, one is limited to the approximately 10,000 – 20,000 IOPS. Using parallel I/O requires a more complex design and developing other additional software infrastructure.

So as we can see, even with 20 TPS, to achieve reasonable sync times will require a sophisticated choice of database, and the development effort needed to take advantage of parallel I/O. Or it requires relaxing the assumption that most of the ledger state does not fit in memory: allowing users with lots of memory to sync quickly, while other users sync slowly.

2.3 Recommendation

The recommended development path is to assume that initially (e.g. 12 months) the TPS will remain relatively low, and that the size of the ledger state will remain only somewhat bigger than memory. In this case it may be possible to develop a solution that does not initially use a very sophisticated database and does not use parallel I/O, but follows a design that can be extended to do so.

3 Requirements and targets

3.1 Transaction rate (TPS)

The transaction rate is the average number of *transactions per second* (TPS). It is a measure of the rate that data is added to a blockchain and is often used to compare different systems. It is a crude measure because it does not take account of the transaction size, which can vary wildly. Nevertheless, because it is the measure used to compare systems, our targets are also expressed in TPS.

Our targets are

Target	TPS
Threshold	20
Middle	50
Stretch	200

For comparison, these are the current transaction rates of Cardano and comparable systems[1].

System	TPS (approx)
Bitcoin	4
Ethereum, peak	17
Cardano, mainnet typical	1
Cardano, mainnet max	7
Cardano, benchmarks max	>50

It is out of scope for this storage project to demonstrate the node running at 50 or 200 TPS. It is within scope to demonstrate that the storage system would not be a bottleneck that would prevent the node running at 50 or 200 TPS.

3.2 Ledger state size

Cardano currently has approximately 2 million UTxO entries. For comparison, Bitcoin currently has approximately 75 million UTxO entries[2].

Cardano also has delegation. The current ratio between UTxO entries and registered stake keys is 5:1. We will assume this 5:1 ratio persists.

Our targets for UTxO sizes are

Target (millions)	UTxO entries	Stake keys
Threshold	10	2
Middle	50	10
Stretch	100	20

We assume the number of registered stake pools will remain in the low thousands.

3.3 Resource requirements

The resource requirements of the Cardano node need to remain reasonable, to allow nodes runing on end user systems with Daedalus, and to keep hosting costs for SPOs reasonable. The purpose of this project is to limit the growth in the memory (RAM) requirement, while allowing for a substantial increase in size of the ledger state generally by keeping most of it on disk. So the memory requirement should remain steady or decrease while the disk requirements must increase.

3.3.1 Existing system requirements

At the time of writing, the published requirements for the Cardano node[3] and for Daedalus[4] are:

[1] Bitcoin TPS charts `https://blockchair.com/bitcoin/charts/transactions-per-second`
Ethereum TPS charts `https://blockchair.com/ethereum/charts/transactions-per-second`

[2] Bitcoin UTxO size charts `https://www.blockchain.com/charts/utxo-count`

[3] `https://github.com/IntersectMBO/cardano-node/releases/tag/1.26.2`

[4] `https://iohk.zendesk.com/hc/en-us/articles/360010496553-Daedalus-System-Requirements`

Resource	Node	Node + Daedalus
CPU	x86 processor, 2+ cores at 1.6GHz (2GHz for a stake pool or relay)	64-bit dual core processor
Memory	8 GB	8 GB
Disk space	10 GB, 20 GB for a stake pool	15 GB

These published requirements are not completely coherent, since the Daedalus case includes the node, wallet backend and the Daedalus frontend, all of which take resources.

3.3.2 CPU requirements

No change in CPU requirement is expected.

3.3.3 Memory requirements

The memory requirement is expected to remain steady or decrease:

Memory target	GB
Threshold	8
Middle	4
Stretch	2

3.3.4 Disk performance requirements

The crucial disk requirement will be disk random read performance (IOPS) not disk space. The proposed requirements are

Random 4k reads	IOPS
Queue depth 1	10,000
Queue depth 32	100,000

This implies a requirement for an SSD. Spinning disks cannot achieve these IOPS requirements.

3.3.5 Disk space requirements

There are also disk space requirements for storing the ledger state on an SSD. Given the target of 10–100 million UTxO entries, and corresponding stake keys, and three stake distribution snapshots, the expected disk requirement for the ledger state are

Target	UTxO entries (millions)	SSD disk space (GB)
Threshold	10	20
Middle	50	100
Stretch	100	200

In addition, disk space is needed to store the block chain itself. This does not require an SSD and can use cheaper spinning magnetic disk. The disk space needed grows continuously, at a rate that depends on the TPS.

	TPS	disk space per day (MB)	per year (GB)
Mainnet typical	1	39	14
Mainnet current max	7	270	96
Threshold target	20	770	275
Middle target	50	1,929	687
Stretch target	200	7,714	2,750

It is worth noting that the upper end is completely impractical for a public system where all partipants download and retain the entire chain as it would require everyone running a node to buy extra multi-terabyte hard drives per year.

3.4 Performance requirements

As described in the summary, we expect the time to synchronise the whole chain to be the hardest performance requirement to achieve. This is because it involves doing the same thing as the node does to validate the chain normally, but 100s of times faster, so that the node can sync in a reasonable time frame.

We have no specific requirements for sync time, but reasonable values would be

Target	faster than real time	hours to sync 1 year
Threshold	100×	88 hours
Stretch	1000×	8.8 hours

3.5 Functional requirements

The storage system must support the needs of the consensus and ledger for storing the bulk of the ledger state on disk. It must support all the operations that these components need to interact with the selected parts of the ledger state that are kept on disk.

The ledger state consists of small complicated parts and large simple parts. Analysis indicates that to meet the memory requirements, only the large simple parts of the ledger state need to be kept on disk. It is acceptable to keep the other parts of the ledger state in memory.

The large parts of the ledger state that must be kept on disk are:

UTxO The collection of all unspent transaction outputs.

Stake addresses and delegation The collection of all registered stake addresses, their corresponding reward account balances, and their choice of delegation to a stake pool. The collection of stake addresses also needs to be indexed by 'pointer': the numerical index of the location on the chain where the stake address was registered. This is needed to support pointer addresses.

Stake distribution An aggregation of the UTxO by stake address, recording for each state address the amount of stake controlled by that address.

Three snapshots of this stake distribution are required.

The storage system must support all the operations the consensus and ledger layers need for interacting with these parts of the ledger state.

For an incremental delivery of a solution, it would be acceptable to start with the UTxO and then proceede to the two stake collections. The UTxO is the largest of the collections, but the two stake collections are also of a substantial size. To achieve the ledger size requirements from Section 3.2 within the memory requirements from Section 3.3 will require all three large parts of the ledger state to be kept on disk.

4 Related components

The related components are from the consensus and ledger layers.

4.1 Consensus

The components that any new code would have to integrate with directly are from the consensus layer. The consensus layer already has three subcomponents that manage on-disk data structures as part of its storage subsystem, for storing:

- the old immutable part of the block chain
- the recent volatile part of the block chain
- snapshots of the ledger state

Storing parts of the 'live' ledger state would add a fourth such subcomponent.

The consensus layer performs all the state management on behalf of the ledger layer. Changes will be required in this state management to accomodate keeping parts of the ledger state on disk.

The design of the interface between the main consensus code and the extra storage subcomponent will be crucial for the success of the project: both for the effort and disruption to the consensus code and also the performance possiblities or limitations.

The (extensive) automated consensus tests rely on being able to mock out the underlying file system used by the storage components, replacing it with a simulted file system. The simulted file system supports deliberate scripted fault injection. The tests rely on this to be able to test that the storage components and the consensus layer as a whole is fully robust in the face of I/O failures and disk corruption. The existing storage components are written in terms of a file system interface, which allows both the real file system for production, and a mock file system instance for the tests.

To maintain the ability to test that integration of the whole consensus layer is robust to file system failure and corruption, will require that new storage components are also written against the file system interface, rather than using the file system directly.

4.2 Ledger

Although the ledger layer does not interact with disk storage directly, the design of the storage system is heavily influenced by the needs and structure of the ledger layer. The consensus and ledger components share an interface, enabling the consensus layer to perform all the state management on behalf of the ledger. Currently this interface relies on the full ledger state being kept in memory. This interface will have to be adjusted to enable the consensus layer to be able to keep the large parts of the ledger state on disk. This interface change will obviously have an impact on the ledger code.

The design of this interface will also be crucial to success of the project: both for the effort and disruption to the ledger code and also the performance opportunities or limitations.

4.3 Cardano DB-Sync and Cardano API clients

The Cardano node provides an interface for other client applications to access the blockchain data. Some client applications also need access to the ledger state as they process the blockchain. In particular the DB-Sync component makes use of this. More generally the Cardano API is in the process of being extended to make it easier to write client applications that make use of the ledger state. Currently these client applications keep their own copy of the ledger state in memory.

Eventually it will be necessary for these client applications, and DB-Sync in particular, to also transition to keeping their copies of the ledger state on disk. The need for this transition is less urgent for applications like DB-Sync than it is for the node itself, because these applications are typically hosted only on servers with more plentiful resources, whereas the node is intended to be deployed in more locations where the resource limits are tighter.

It would save development effort overall if client applications can reuse the ledger state storage components developed for the node.

5 Analysis of on-disk data structures

The most important design question is the choice of an appropriate on-disk data structure. This data structure must satisfy the functional requirements from Section 3.5: the operations needed by the ledger rules and consensus layer. We should start therefore with a review of the operations required.

Beyond that, it is clear from initial estimates that the most pressing constraints are performance constraints. We must therefore assess if the required rates of operations are feasible for an on-disk data structure, and if so what the options and trade-offs are.

There is a fundamental trade-off in the performance of on-disk data structures between read performance, write performance and the amount of main memory used. Athanassoulis et al. [2016] describe this trade-off as the "RUM conjecture": for read vs update vs memory. To see where we fit in this trade-off we must analyse the expected mix of reads vs updates, and the amount of memory available vs the expected size of the dataset.

Finally, to assess feasibility we need a quantitative – albeit approximate – assessment of the I/O performance for different general design options vs the hardware capabilities. This requires a brief review of hardware capabilities, i.e. SSD performance.

We review these issues in turn in the remainder of this section.

5.1 Operations

Here we describe the operations the on-disk data store must support. These requirements are driven by the operation the ledger uses on those parts of the ledger state that we will move to disk. As a reminder, the functional requirements from Section 3.5 are that we must keep the following parts of the ledger state on disk

- The UTxO
- Stake addresses and delegation choices
- Stake distribution

5.1.1 UTxO operations

While processing transactions, the only operations on the UTxO are point LOOKUP, INSERT and DELETE.

Depending on how the stake distribution is managed we may also require a bulk SCAN over a stable snapshot of the UTxO.

5.1.2 Stake address operations

Transactions involving stake addresses can do the following: register, de-register, withdraw rewards and set the delegation choice. These only need LOOKUP, INSERT and DELETE.

Maintaining an index of stake addresses by pointer also requires only LOOKUP, INSERT and DELETE.

There is also a bulk INSERT needed when paying out stake pool rewards. This can be implemented as multiple point INSERT operations, but some data structures may have more efficient bulk operations.

5.1.3 Stake distribution operations

The ledger layer needs to change the way it manages the stake distribution – for scalability reasons unrelated to the disk storage. There are a couple of primary design options. Ideally the on-disk storage should support either of these primary design options.

1. One option is to compute the stake distribution by doing a bulk SCAN over the UTxO to aggregate the stake by the stake keys. This would require a bulk SCAN over a stable snapshot of the UTxO to produce a new immutable table holding a stake distribution snapshot. This snapshot would also need to support a bulk SCAN to aggregate the stake pool distribution.
2. Another option is to maintain the current stake distribution incrementally as each transaction changes the UTxO. This would involve an UPDATE operation, which could be implemented as a LOOKUP and INSERT, or some data structures may support it directly more efficiently. This design would also involve taking a stable snapshot of the stake distribution, and performing a bulk SCAN to aggregate the stake pool distribution.

5.1.4 Consensus operations

In addition to the operations required by the ledger layer, the consensus layer has to do the overall state management and it has extra requirements.

The consensus layer needs to be able to roll back to any ledger state within the last K blocks. For small K in particular this needs to be efficient since short forks are common. It is not necessary to support every single intermediate rollback point in the last K, since it is possible to roll forward from any point. So at minimum what is required is sparse rollback points, covering at least K blocks.

It is also necessary to be able to snapshot the entire ledger state to disk from time to time. This is needed to be able to shut down and restart, without having to replay the chain from genesis. The consensus layer currently does this by writing out the whole in-memory ledger state to disk. With parts of the ledger state on disk and parts in memory, it will still be necessary to be able to construct consistent overall snapshots. The easiest way to do this would be for the on-disk storage to support cheap snapshots – cheap in the sense that little time is needed to take the snapshot. There will only be a limited number of such snapshots, e.g. 3, so it is acceptable for these to incur extra storage costs.

It is worth noting that the consensus layer does *not* require the disk operations to be completely durable – the D in ACID. The consensus layer can recover from data loss – especially loss of recent data – by replaying the chain or in the worst case by fetching the chain again from its peers. This is an important point for performance because achieving durability generally entails an `fsync()` operation which impedes efficient use of write buffers and large sequential writes. The consensus layer already takes advantage of this for its other on-disk storage subcomponents.

5.1.5 Cardano API clients operations

Most Cardano API clients simply iterate the ledger state forward, and hence use the same operations as needed by the ledger. Some also need to follow the live chain through rollbacks and hence need the same operations as consensus does. Some API clients may also need bulk SCANs over a snapshot of the UTxO, or other collections.

5.1.6 Overall operations

Overall we require the following operations

1. LOOKUP

2. INSERT

3. DELETE

4. UPDATE: One of the design options for the stake distribution would benefit from a update operation that performs an insert with monoidial merge with any existing value. This can always be implemented with a LOOKUP followed by an INSERT operation, but some data structures support a more efficient UPDATE directly.

5. BULK UPDATE: this is optional since it can always be implemented as multiple UPDATE operations, but some data structures may support it directly.

6. SNAPSHOT: preserving a consistent snapshot of the collection.

7. RESTORE: restoring from a previous a snapshot.

8. SCAN: a bulk scan over a snapshot, or immutable table, in an arbitrary order.

9. ROLLBACK The consensus layer will rollback ledger state whenever it switches chain forks. The data store must support this in some way, with at least sparse rollback points, covering at least K blocks. This could be implemented in terms of SNAPSHOT and RESTORE or more directly. Rollback by a small number of blocks will be frequent and needs to be efficient.

While not strictly essential, development will be considerably simplified if a single on-disk data structure can support all operation for each of the three parts of the ledger state we need to store.

There are three data collections to keep on disk. We will focus the analysis on the UTxO initially. The UTxO is the collection that is most frequently used, as it is consulted and updated for every single transaction.

5.2 UTxO access pattern

The access pattern for the UTxO data structure is *very* write heavy. In the usual case, we expect exactly three operations for each UTxO entry over its lifetime:

1. One INSERT operation when the UTxO entry is created.

2. One LOOKUP operation when validating the block in which the UTxO entry is spent.

3. One DELETE operation when the block that spends the UTxO entry is added to the chain.

This means that processing a typical transaction with 2 inputs and 2 outputs will entail:

- Two LOOKUP operations: one for each input.
- Two DELETE operations: one for each input.
- Two INSERT operations: one for each output.

This gives us a 1:2 read:write ratio; for each LOOKUP there is one INSERT and DELETE. This is very write heavy, and is very different from typical database applications that are read heavy (e.g. a 10:1 read:write ratio).

Note that some UTxOs will be read more often – if validation of a block fails – or if a block is validated multiple times. We expect the vast majority of UTxOs to be read exactly once, and this applies moreso while syncing, where we do not expect validation to fail. By definition, UTxOs will be written at most twice: once to create and once to delete.

5.3 UTxO data access locality

Many data structures exploit locality in their data. For example, a time series database will store values together that are adjacent in time. In this way, queries which inspect values that are close in time (most all queries to a time series database!) are able to read a single disk page, and obtain many values of interest at once.

There are two dimensions for locality for the UTxO: the time of creation and the key/identifier. The key/identifier in the UTxO is the pair of the transaction id – which is a 32 byte cryptographic hash – and the index of the unspent output within that transaction.

We can expect no locality whatsoever in the identifier dimension. Indeed cryptographic hashes are the worst case for locality because by design they are effectively perfectly random. One might hope there is a small locality effect from accessing multiple outputs of the same transaction at the same time. This would only provide a small benefit, but in practice multiple outputs tend to be used independently because they tend to belong to different parties.

Less obviously, there is apparently relatively little temporal locality in UTxO blockchains. The best available comparable data comes from bitcoin, for which there are charts[5] of the age distribution of the UTxO set. Ideally we would like the distribution of the lifespan of UTxO entries, rather than the distribution of age of entries within the UTxO. So this is not the perfect indicator, but under reasonable assumptions it is indicative. For locality to be really effective in reducing I/O costs we would want to see a heavily skewed distribution with the bulk of the activity corresponding to a small fraction (e.g. 10% or less) of the UTxO set. This is because that would enable keeping the small active fraction in memory which would avoid I/O costs. What the bitcoin age distribution indicates is that there is some temporal effect, but it is not strong.

The conclusion we should draw is that there is little data locality to take advantage of. There may be limited temporal locality, but it is unlikely to be a decisive factor in choosing between design options. There are a few corollaries of this conclusion:

- Caching the UTxO data (as opposed to indexes) is not likely to be very effective, unless the memory cache is a large fraction of the total data size – which is contrary to our requirements from Sections 3.2 and 3.3 for ledger state size and memory use.
- We cannot expect to do better than 1 disk read per LOOKUP operation. This is because due to the poor data locality we cannot expect each disk read to fetch more than one element, and because our requirements from Sections 3.2 and 3.3 are to manage ledger states that are substantially bigger than the available memory.
- Thus, no matter the choice of on-disk data structure, we can expect the LOOKUP operation performance to be bounded by the I/O performance of 'random' reads.

5.4 SSD performance

To assess feasibility we need a sense of typical SSD performance. We will include the performance of spinning disks but only to illustrate that their performance is so poor as to be out of the question.

To make things concrete we look at the example of Samsung SSDs. Samsung manufacture a range medium to high end consumer and server SSDs. They helpfully publish rated performance numbers including low and high queue depth. We select a vaguely representative sample of their range, picking 500GB models where possible. For our spinning disk comparison we select a couple models from Western Digital. Western Digital do not consistently publish performance numbers for their disk drives, so some numbers below are taken from 3rd party reviews, or are omitted entirely.

[5] For example `https://hodlwave.com/`

Model	Capacity	Interface type	Comment
WD Blue	1TB	SATA	Mainstream, low cost spinning disk
WD Black	1TB	SATA	'High performance' spinning disk
870 QVO	1TB	SATA	Latest gen, high capacity, lower performance
860 PRO	512GB	SATA	Previous gen high end SATA
960 EVO	512GB	NVMe PCIe 3	Previous gen high end NVMe
970 EVO Plus	500GB	NVMe PCIe 3	Latest gen high end NVMe
980 PRO	500GB	NVMe PCIe 4	Latest gen ultra-high performance NVMe

The random read performance is given at "queue depth 1" and at "queue depth 32". Queue depth 1 means issuing all I/O operations serially, whereas a high queue depth implies continuously issuing many I/O operations in parallel, so that the hardware it given 32 operations to do at any one time.

Model	4k reads at QD1	4k reads at QD32	QD32 speedup
WD Blue	–	128 IOPS	–
WB Black	200 IOPS	500 IOPS	2.5×
870 QVO	11,000 IOPS	98,000 IOPS	9×
860 PRO	11,000 IOPS	100,000 IOPS	9×
960 EVO	14,000 IOPS	330,000 IOPS	23×
970 EVO Plus	19,000 IOPS	480,000 IOPS	25×
980 PRO	22,000 IOPS	800,000 IOPS	35×

On the write side, some on-disk data structures are sensitive to random writes, while others rely on sequential writes. The WD Blue model is omitted as the numbers are not readily available.

Model	4k writes at QD1	4k writes at QD32	sequential writes
WB Black	450 IOPS	460 IOPS	176 MB/s
870 QVO	35,000 IOPS	88,000 IOPS	530 MB/s
860 PRO	43,000 IOPS	90,000 IOPS	530 MB/s
960 EVO	50,000 IOPS	330,000 IOPS	1,800 MB/s
970 EVO Plus	60,000 IOPS	550,000 IOPS	3,200 MB/s
980 PRO	60,000 IOPS	1,000,000 IOPS	5,000 MB/s

There are a number of things to note:

- These are all relatively high end and expensive SSDs. Minimum system requirements can only be at the lower end of this range.
- There is huge difference between serial and parallel performance: typically a factor of 10 to 20 times.
- Read performance at queue depth 1 has only slightly improved over time.
- Read performance at high queue depths continues to improve from one generation to the next.
- Sequential write performance is even higher than random 4k writes at high queue depth, typically by an extra 40–50%.
- Spinning hard drives have extremely low random access performance, though moderate sequential performance.

Real world numbers are less than the manufacturer rated numbers, once practical details like file systems and OS I/O APIs are taken into account. The authors have benchmarked a 960 EVO 250GB model under Linux using the ext4 file system on an encrypted block device, using the `fio` benchmarking tool. While rated at 330,000 IOPS for random reads at queue depth 32, the `fio` benchmark shows it achieves around 220,000 IOPS for random reads at queue depth 32.

Using parallel I/O would give us the opportunity to get an extra factor of ×10–×20. It is likely to be necessary to take advantage of this parallel I/O to achieve the system performance requirements.

5.5 Memory vs data set size

The requirements from Section 3.2 are for a UTxO of 10–100 million entries, and 2–20 million stake addresses. The memory requirement from Section 3.3 are for the node overall to take from 2–8GB. Thus we can assume that we can dedicate at least 1GB of main memory as part of an on-disk data structure design.

The UTxO entries are expected on the order of 100 bytes for most entries, and around 128 for EUTxO entries used by Plutus scripts. This means we can expect a 100 million UTxO to take around 9–11GB on disk.

The UTxO is of course not the only on-disk data collection, but even if only half of the 1GB is dedicated to the UTxO then this gives us a rather favourable ratio of available main memory to the size of the data set on-disk. It is not enough to cache large parts of the data set, but is enough to enable a number of more memory intensive options for on-disk data structures.

5.6 The 'RUM' trade-off

As described by Athanassoulis et al. [2016], there is a 'RUM' trade-off in the design of on-disk data structures between *read* performance, *update* performance, and amount of main *memory* required. Understanding where we fit in this trade-off will help to assess and select appropriate design options.

In Section 5.2 we established that we have a very write heavy access pattern, with a read:write ratio of 1:2 for the UTxO. In Section 5.5 we noted that we have a relatively generous amount of memory available compared to our data set size. Given this, the RUM trade-off immediately tells us that we should select an on-disk data structure that is optimised for writes at the expense of reads and at the expense of the amount of main memory used.

There are write-optimised on-disk data structures that are able to batch all update operations into a small number of write I/O operations. For example this would pack 30–40 UTxO INSERT operations, or over 100 UTxO DELETE operations into a single 4k write. Furthermore, these can be large sequential writes rather than 4k random writes, making optimal use of SSD write performance.

By contrast, many read-optimised data structures entail that each update operation (i.e. INSERT and DELETE) correspond to several small write I/O operations. Given our write-heavy workload, the bottleneck would be I/O write operations.

On the other hand, as we established in Section 5.3, we cannot expect to do better than a single read I/O operation for each LOOKUP operation. This would be the case even for read-optimised data structures.

The RUM trade-off makes the choice clear: we should select a write-optimised design and use our generous memory to dataset ratio to tune the design to get the read cost as close as reasonably possible to the lower bound of 1 read I/O operation per LOOKUP operation. We can then expect the performance bottleneck to be the hardware I/O random read performance.

5.7 Assessing I/O performance feasibility

Given the targets for TPS in Section 3.1 and sync speed in Section 3.4, it is useful to see these means as a requirement for the storage system. It is illustrative to look at the effective TPS that would be needed to be able to sync 100 or 1000 times faster than real time, for different TPS targets. The table below also shows the corresponding number of UTxO LOOKUP and INSERT/DELETE operations per second, assuming the typical transaction of 2 inputs and 2 outputs.

Target TPS	target sync factor	effective TPS	LOOKUP ops/s	INSERT/DELETE ops/s
20 TPS	100×	2,000 TPS	4,000	8,000
20 TPS	1000×	20,000 TPS	40,000	80,000
200 TPS	1000×	200,000 TPS	400,000	800,000

If we assume a write-optimised data structure that can use sequential writes then we are interested in the raw data rate of the UTxO operations. Assuming 128 bytes for INSERT and 36 bytes DELETE operations, the raw data rate for the last row of the table would be only 63MB/s. In reality, write-optimised data structures have to write each data item on average several times, but even a factor of 5 would still keep us well within the capabilities of the sequential write speed of low end SATA SSDs. In short, with a write-optimised data store, write performance will be well within the hardware performance capabilities of SSDs from Section 5.4.

On the read side, if we assume we can get relatively close to the lower bound of 1 read I/O operation per LOOKUP operation, e.g. perhaps 1.2–1.5×, then we can see that the last row of the table would correspond to 500,000–600,000 IOPS which is outside the capability of all but the latest and most expensive desktop or server NVMe PCIe4 SSDs, and when using parallel I/O to utilise a high queue depth.

Dropping down by a factor of 10 to the middle row gives us something that is within the realms of possibility for mainstream SSDs, but only when using parallel I/O to utilise a high queue depth.

Dropping down by a factor of 10 to the first row gives us something that is well within the realms of possibility for mainstream SSDs, even when not using parallel I/O, and so using only queue depth 1.

Finally we should be aware of some caveats. This analysis only covers the UTxO and not the other components, so we must keep in mind that some I/O bandwidth is needed for the other data stores. We must also remember that SSD read and write bandwidth compete with each other, so we cannot use the full rated performance of both simultaneously.

6 Additional design considerations

6.1 Utilising parallel I/O

As discussed in Section 5.4, modern SSDs can achieve IOPS in the hundreds of thousands by operating concurrently on many I/O requests at once. Without this concurrency they can achieve only tens of thousands of IOPS. Low end SSDs will typically scale well up to 8-16 operations at the same time, whereas high end SSDs will scale well up to 32, and up to 128 for some of the latest generation high end models.

As we discussed in Section 5.7, we expect saturating the disk bandwidth will be necessary to achieve reasonable sync times. For software to saturate the bandwidth of the disk however, that software must be designed to issue many operations in parallel. There is a trade-off here between performance and complexity, and hence development time. To saturate the bandwidth of the disk by issuing operations in parallel will require more complexity in the solution. This complexity is not confined to the storage subsystem. The opportunity for parallelism must be present in the consensus and ledger design and in the interface between the consensus and storage subsystem.

It is very likely that a solution required to perform well with a growing blockchain will need to exploit parallel I/O to achieve performance targets. Since the use of parallel I/O will significantly affect the design of the interface between the consensus layer and its storage subsystem, it makes sense to consider this in the design from the start, rather than to leave it to a later phase of development. Indeed even if the parallelism is not fully exploited initially, it probably still makes sense to design the interface to support parallel I/O.

6.2 Integration with `io-sim`

The Ledger and Consensus codebases have achieved impressive quality and reliability, in no small part due to their use of the `io-sim` Haskell library to test against a broad array of failure modes. This relies upon integrating at the Haskell source level.

The solution should similarly demonstrate it's correctness with an io-sim test suite. This is not a strict requirement, however it is difficult to see how one could be confident in the correctness of the solution without it.

6.3 Changes to the ledger scheme

While the Byron ledger rules have a separate implementation that is verified against the executable specification, Shelley (and the following era's) use the executable specification directly. This was a pragmatic choice at the time it was made, however it does introduce tensions. The Ledger team are limited in their ability to modifying the implementation of those rules to meet business requirements, because changing the executable specification risks introducing unsoundness to the ledger rules.

The solution should minimise disruption to the implementation of the ledger rules. This may require us to separate the implementation of the Shelley (and forward) ledger rules from their executable specification.

7 Existing on-disk data structures

In the light of the RUM trade-off discussed in Section 5.6, it is interesting to review the the existing on-disk data structures used in the node.

The existing consensus layer contains three on-disk data structures. All three of these on-disk data structures are bespoke implementations. Two of them are special purpose key value stores.

The immutable block database This data structure stores the blocks from the old part of the chain: the blocks that are no longer at risk from being rolled back. It is a special purpose key value store. It maps the pair of a slot number and block hash to the serialised byte representation of the block itself. The design of this store trades away read performance to get minimum memory use: in general individual reads can take up to 3 I/O operations, but it does not require any in-memory indexes. This design trade-off makes sense because the chain itself is large and ever-growing, but we cannot dedicate an ever-growing amount of memory for indexes.

The access pattern for blocks tends to be sequential, and so the design supports efficient sequential reads by placing blocks next to each other in the disk files. This substantially reduces the I/O cost for sequential reads.

The volatile block database This data structure stores the blocks from the new part of the chain, as well as blocks before they are added to the chain, and from forks that are no longer part of the chain. It is a special purpose key value store. It maps the pair of a slot number and block hash to the serialised byte representation of the block itself. It follows a very different design compared to the immutable block database. It is what is known in the literature as a log-structured hash table (LSH table). It keeps a full in-memory index of all the keys, mapping to the location on disk where the value can be found. This design of key value store trades-off memory use to minimise the read and write I/O costs. It use memory to keeps a full index in memory, but this allows reading with a singe I/O worst case, and all writes are sequential and append-only. This design choice is appropriate because read and write performance for recent blocks is important as these are the most frequently accessed ones, including in potentially adversarial situations. The memory cost is acceptable because the size of the volatile block database is relatively small.

The ledger snapshot database This on-disk store keeps relatively recent snapshots of the ledger state. Its purpose is to allow the node to recover its in-memory version of the ledger state without having to travers the whole blockchain from the start. It does not need to keep many snapshots because the ledger state can be recovered by replaying the chain from any snapshot.

The RUM trade-off For the two key value stores, it is interesting to note – even though they both do the same thing of mapping slot/hash pairs to blocks – that due to differing access patterns and dataset sizes they sit in different parts of the 'RUM' trade-off space. The two designs are optimised for their different locations in the 'RUM' trade-off space and are thus quite different from each other.

Implementing bespoke on-disk data structures It is worth reflecting on the fact that the node already contains two bespoke key value stores, optimised for different use cases. These were both developed in reasonable time and to a very high quality.

The key value store for the large parts of the ledger state will be somewhat more complex as it needs to support more operations (see Section 5.1).

8 Design choices

It is clear that we will need to integrate an on-disk key-value store capable of serving the ledger state, and supporting the operations identified in Section 5.1.

There are many choices in the design space that trade off development effort, delivery risk and scalability. We have factored these into a set of semi-independent choices:

- The choice of the class of on-disk data structure: e.g. B+ tree vs. LSM tree vs. other.
- The choice to reuse an existing on-disk data structure implementation or to implement one bespoke.
- The choice of the style of interface between the consensus layer and the new ledger state storage subsystem.
- The choice to use serial or parallel I/O.
- The option to add a special caching scheme to optionally take advantage of large amounts of memory to increase performance.

In the remainder of this section we discuss each choice and how it affects our ability to meet our requirements. In the following section we discussed the preferred option, which is a combination of these choices.

8.1 Data Structure

In Section 5.6 we came to the following conclusion:

> The RUM trade-off makes the choice clear: we should select a write-optimised design and use our generous memory to dataset ratio to tune the design to get the read cost as close as reasonably possible to the lower bound of 1 read I/O operation per LOOKUP operation.

This conclusion is based solely on scalability and meeting the middle to higher end of the performance targets. There are also other choices with different trade-offs for development effort.

8.1.1 LSM trees

We have identified LSM trees [Dayan et al., 2017] as the on-disk data structure most suitable for our workload (see Section 5.6). LSM trees are the most widely used design for write-optimised key value stores. They are used for example by the main bitcoin implementation for storing the UTxO. LSM trees offer:

1. Append-only writes to disk.

   This enables very good write performance. It also means that the slow `fsync()` operation is not needed to ensure data consistency. It would only be needed for prompt durability guarantees – which we do not require.

2. A cheap snapshotting mechanism.

   Both the ledger rules and the consensus layer need to be able to take snapshots of parts or all of the ledger state. It is helpful if this operation can be cheap.

3. A cheap rollback mechanism.

   In normal operation the Cardano node needs to be able to roll back the ledger state to an earlier point within the last K blocks, see Section 5.1.6.

4. Workload

   There is a well specified recipe, described by Dayan et al. [2017], for tuning the data structure to various workloads. Our workload is quite unusual (see Section 5.2) . We would be able to tune the LSM tree to match that workload, and moreover, if the workload changes in the future as Cardano evolves, we will have the opportunity to revisit these decisions without reworking the entire data structure.

5. Efficient monoidial UPDATE operation.

   The UPDATE operation can be implemented as a write operation without needing a LOOKUP. This would be a major benefit if it needs to be used frequently because LOOKUP operations are the performance bottleneck. We would use this at a very high frequency if we choose to maintain the stake distribution incrementally.

There are no existing suitable Haskell implementations of LSM trees available on hackage. There are existing non-Haskell implementations including LevelDB and RocksDB . Were we to develop a bespoke LSM tree we would follow the blueprint laid out by Dayan et al. [2017].

8.1.2 B+ trees

Rather than an LSM tree, one might use a B+ tree [Comer, 1979]. B+ trees are the main on-disk data structures used by most traditional databases, including for example SQLite and PostgreSQL. There is also an existing Haskell implementation of a B+ tree on hackage[6].

We do not expect to be able to meet the minimum performance targets (see Section 3.4) with a B+ tree based solution. The ability to reuse an existing implementation is a clear benefit for development time. This could therefore be a useful interim solution, on route to the final delivery.

8.2 Off-the-shelf or bespoke

There are many existing on-disk data stores, for example: SQLite[7], PostgreSQL[8], LevelDB[9] and RocksDB[10] to name just a few. We could build our data store on one of these technologies, or alternatively, we could develop a bespoke data store, either from scratch or on top of a lower level technology.

The trade-offs are as follows.

8.2.1 Development time

We would expect most solutions utilising an existing, mature, technology to consume less development time than developing a bespoke solution. Though there is clearly a saving on developing the actual data store, significant work would remain in cleanly integrating and testing the technology with our related components.

8.2.2 Integration time and complexity

Integrating an Off-the-Shelf data store will, for some choices of that data store, introduce additional complexity to the deployment and operation of the Cardano Node. For this reason, we can rule out out-of-process database, such as PostgreSQL, and limit our choices to embeddable solutions such as SQLite, RocksDB, or LevelDB.

A bespoke solution could be designed and built to integrate seamlessly with the Cardano node.

8.2.3 Quality

A solution built using a non-Haskell data store would preclude the ability to use the simulation-based testing technique used by the existing consensus automated test suite (see Section 6.2). The existing technique ensures two things:

1. that the integration of the on-disk data stores works in normal operation; and
2. that the combination of consensus components is robust in the presence of I/O failures and silent disk corruption.

The first aspect could be covered by new integration tests, with an obvious development and ongoing maintenance cost. The second point is exceedingly hard to replicate without the ability to simulate a file system with fault injection.

[6] https://hackage.haskell.org/package/haskey

[7] https://www.sqlite.org/

[8] https://www.postgresql.org/

[9] https://github.com/google/leveldb

[10] https://rocksdb.org/

The previous Cardano node implementation (`cardano-sl`) used an off-the-shelf data store (RocksDB) and lacked this style of testing. Throughout its lifetime the previous node implementation suffered from unrecoverable disk corruption problems. These problems occurred in production with users relatively frequently but were nevertheless extremely hard to reproduce in a development setting – precisely because they depend on rare I/O faults occurring at particular times. This had a high cost on users and the technical support team, and generated a poor reputation for Daedalus amongst users. These problems were fully resolved by the new node implementation which was designed and developed along with the simulation and fault injection testing framework.

Loosing this testing would mean we could no longer be confident that the node can recover from disk problems. As the `cardano-sl` history makes clear, this would be a significant retrograde step.

Certain existing data stores written entirely in Haskell – and any bespoke data store developed in Haskell – would have access to the `io-sim` tooling, and would integrate cleanly into the existing consensus test suite. Existing Haskell implementations would need to support replacing their disk data access layer and to be run in a caller-supplied monad. This is true of the `haskey` library for example but is not true of any of the Haskell bindings to external libraries such as RocksDB.

8.2.4 Data structure and workload

As described in Section 5.6, we have a clear idea of the trade-offs to which the data store should be tuned. In Section 8.1 above, we described the main choices we have in data structure. For any off-the-shelf solution we will be constrained to the on-disk data structure that implementation provides. We will be constrained to tune that implementation only with the adjustable parameters offered by that implementation.

A bespoke data store will allow us the greatest flexibility in choosing and tuning the on-disk data structure to handle our specific workload.

8.3 Parallel or serial I/O

As described in Section 6.1, we have the choice to use parallel I/O, at the cost of additional complexity in the implementation. A scalable solution that meets the performance targets (Section 3.4) will certainly require parallel I/O. It may be possible and reasonable to start with a simpler, serial I/O implementation, and then add parallel I/O once we have demonstrated the correctness of that simpler implementation.

An off-the-shelf implementation comes with a pre-existing fixed choice of serial or parallel I/O. Thus, demanding parallel I/O restricts our choice of off-the-shelf implementations. Using parallel I/O in an existing implementation will likely require using the database interface in a particular way to expose the opportunities for parallelism, which would likely increase integration complexity.

A bespoke implementation would require extra effort to take advantage of parallel I/O, but the interface could be designed to minimise the integration complexity.

8.4 Interface style

There will be an interface between the consensus layer and the new storage subsystem. There are a few main styles for this interface:

- A naïve 'CRUD' interface
- A batch mode 'CRUD' interface
- A non-standard 'pipelined' interface

One of the important considerations for development time is the changes that would be needed to the consensus or ledger code bases to be able to adapt to using the on-disk storage. These are both large and complex code bases, and hence minimising the changes to them can be a very substantial saving. It could be worth using a non-standard interface if it enables less invasive changes in the existing components.

A naïve 'CRUD' interface to the data structure would be simple to provide on top of any existing or bespoke underlying data store. A significant disadvantage to this style however is that it precludes the use of parallel I/O because the interface is incapable of exposing any parallelism of read operations to the underlying data store.

A batch mode 'CRUD' interface allows homogeneous batches of LOOKUP operations, batches of INSERT operations etc. This enables the use of parallel I/O to some degree, if the underlying data store is capable of it. It cannot necessarily fully utilise the full parallel I/O capacity of the hardware because the homogeneous nature of the batches limits their size, and there are always gaps between batches.

A disadvantage to both 'CRUD' style interfaces is that it requires the ledger rules to be written to make the LOOKUP, INSERT and DELETE operations explicit. This would be a significant change to the way the ledger rules are written. The ledger rules are currently expressed as pure functions. In addition to the development cost of a change in style, there are good reasons for the existing design as simple pure functions: it greatly simplifies testing and auditing against the specification. The consensus design also takes advantage of the fact that the ledger rules are pure functions.

Instead, we recommend a non-standard 'pipelined' interface which:

- allows the ledger state changes to remain as pure functions; and
- exposes arbitrary amounts of pipeline style parallelism in the operations.

The essential idea of this interface is that in advance of processing a transaction or block, we identify all the keys that will need to be accessed – such as the transaction inputs which are the keys of the UTxO – and perform the disk reads in advance to bring the results into memory. Based on these read sets, the transactions are processed wholly in memory using pure functions. The result of the transaction processing includes the *differences* to the on-disk tables, and these differences are then flushed out to disk afterwards. This would still involve some changes to the ledger rules to output the differences in the large mappings rather than the whole mappings. This is a relatively mechanical change and keeps the general style as pure functions. We intend to elaborate on this interface, and propose a design, in a following document.

Note that committing to this interface does not require us to implement parallel I/O operations immediately. The interface simply makes it possible by exposing the opportunity for parallelism.

8.5 Caching layer

Although we expect caching to perform poorly over our data structure, due to the lack of data access locality (see Section 5.3), perhaps a poor cache would be significantly better than nothing.

One option to mitigate long sync times may be to allow users to add a large in-memory cache over the data structure while syncing. In this way, users could sync on a large cloud machine (Say 64GB RAM), save their state to disk, then run their node on a smaller cloud machine using that synced state.

This is somewhat speculative, and is included here due to the lack of other options for improving sync times.

9 Preferred option

Our preferred option is a hybrid, seeking to balance our ability to meet the performance targets in the long term with developing an interim solution that lets us start to scale up the size of the ledger state.

- We recommend the non-standard 'pipelined' style of interface (see Section 8.4) between the consensus and storage subsystem. This should minimise the necessary changes in the consensus and ledger layers and keep open the opportunity to take advantage of parallel I/O. The choice of interface is hard to change later so it is best to pick the final best choice, rather than taking an interim approach.

- We recommend an in-Haskell implementation of the on-disk data structure. As discussed in Section 8.2.3, choice of an in-Haskell implementation preserves the investment in the consensus testing infrastructure and assures that the node remains robust in the presence of disk I/O failures.

- We recommend trying the existing `haskey` B+ tree implementation as an interim solution, and a bespoke LSM tree implementation as a eventual solution. This balances the time to develop a first working version term with the need to meet the performance targets in the long term.

- We recommend that it will be necessary eventually to add support for parallel I/O to scale to higher transaction rate targets while maintaining acceptable bulk sync times. This will buy us approximately a factor of 10 in performance (throughput or sync times).

Building a version on top of `haskey`, and even delivering it to production, would solve the upcoming problem of the ledger state not fitting in memory, and may have acceptable sync times in the short term, assuming that the chain is still not too large and transaction rates are not too high.

This preferred option will *not* meet the performance targets in the short term. With this option they would only start to be met once the bespoke LSM tree is in place, and fully met once parallel I/O is implemented.

10 Preferred development approach

Our preferred development approach seeks to minimize delivery risk, deliver some improvement early, and give the option to pause some of the work before completion. We will design the API early and implement several iterations of the data structure, each with more complexity and better performance than the previous.

A key strategic question is early or late integration. Early integration would be clearly preferable to minimise risks. On the other hand early integration may require too much attention from the teams responsible for the components involved in the integration (ledger and consensus teams) at a time when their attention is necessarily elsewhere.

Early integration In an early integration approach we would immediately prioritise a draft API for agreement with stakeholders (ledger and consensus teams). Once agreed, we would deliver a trivial implementation of a data store that was still entirely in memory, using the existing in-memory representations, but which uses the newly agreed API to interact with the data. The next phase would be the integration of that interface, driven primarily by the teams responsible for the existing components, including the necessary changes to the ledger rules to use 'difference' types for output.

Late integration In a late integration approach we would still prioritise a draft API for agreement with stakeholders. Once agreed however we would do prototype and exploratory integrations without significant input from the teams responsible for the existing components. This would reduce but not eliminate the risks of integration problems later.

Store development Irrespective of the early or late integration strategy, in parallel with the actual or exploratory integration, we will begin implementing a B+ tree. Preferably this will be done by reusing or modifying and reusing the existing `haskey` B+ tree library. The focus will be to produce a working solution, not to achieve good performance. We will verify the correctness of this implementation with QuickCheck property tests and `io-sim` where appropriate. The intention will be to use these tests as a basis to test an LSM tree later.

Once the B+ tree is feature complete, and the related components have completed their integration, we would have the option to tune the performance of the B+ tree so that it could be deployed into production. Whether we would do this or not would depend on the urgency of lowering the memory footprint of the Cardano node.

We will then replace the B+ tree in the previous solution with a bespoke LSM tree. Our implementation will be designed with parallel I/O in mind from the outset. We will verify its correctness with the test suite developed for the B+ tree implementation. We will iterate on this LSM tree, improving its performance as required, including by using parallel I/O.

11 Risks

There are a numbers of risks in the project, and in the preferred option in particular.

11.1 API design

The design of the API between the consensus layer, ledger layer and the new storage subsystem will have far reaching impact on the success of the project. We need to carefully balance the need to integrate smoothly with the ledger and consensus layers and the requirement that the API will be able to support parallel I/O. This points one towards fixing the API early, so that integration in the related components can start, however this risks increasing the costs, or may render impossible, incorporating the outcome of our experience implementing the data structure into the design of the final API.

11.2 Performance of bulk chain synchronisation

As discussed in Section 5.7 there is reason to expect that any solution will have sync times in the order of days per chain-year at even moderate transaction rates. This is the critical constraint that guides the majority of our design. There is some opportunity to mitigate this (see e.g. Section 8.5), but this will remain a looming threat if transaction rates increase substantially. We recommend pursuing research to investigate ways that nodes can catch up with the blockchain without having to validate the entire history of the chain.

References

Manos Athanassoulis, Michael Kester, Lukas Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. Designing access methods: The rum conjecture. In *Proceedings of the 19th International Conference on Extending Database Technology*, EDBT 2016, 2016. doi: 10.5441/002/edbt.2016.42. URL `https://openproceedings.org/2016/conf/edbt/paper-12.pdf`.

Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979. URL `http://dblp.uni-trier.de/db/journals/csur/csur11.html#Comer79`.

Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, 2017. doi: 10.1145/3035918.3064054. URL `https://doi.org/10.1145/3035918.3064054`.