Storing the Cardano ledger state on disk: API design concepts

AN IOHK TECHNICAL REPORT

Douglas Wilson
douglas@well-typed.com

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

Version 0.4, November 2021

1 Purpose

This document is intended to explore and communicate – to a technical audience – design concepts for how to store the bulk of the Cardano ledger state on disk, within the context of the existing designs of the Cardano consensus and ledger layers. We will try to illustrate the ideas with prose, algebra, diagrams and simple prototypes in Haskell.

The reader is assumed to be familiar with the initial report on this topic, covering analysis and design options [Wilson and Coutts, 2021].

2 Acknowledgements

Thanks to the consensus team for many useful discussions and feedback. Thanks particularly to Edsko de Vries for critiques of early versions of these ideas. Thanks to Tim Sheard for the inspiration to use the ideas and notation of a calculus of changes.

Contents

3 Literate Haskell prototyping

We will try to illustrate some of the ideas in the form of a Haskell prototype. To keep ourselves honest, the source for this document is also executable Haskell code. It can be loaded within `ghci` for type checking, experimentation and testing.

We start with a typical module preamble.

```
 {-# LANGUAGE TypeFamilies #-}
 {-# LANGUAGE FlexibleContexts #-}
 {-# LANGUAGE FlexibleInstances #-}
 {-# LANGUAGE InstanceSigs #-}
 {-# LANGUAGE ScopedTypeVariables #-}
module UTxOAPI where

import           Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map
import qualified Data.Map.Merge.Strict as Map
```

4 Ledger state handling in the current design

4.1 In the ledger layer

The existing ledger layer relies heavily on using pure functions to manipulate and make use of the ledger state. In particular the ledger rules are written in a style where old and new states are passed around explicitly. The ledger rules are complex and non-trivial to test, so the benefits of using a pure style are substantial and not something that we would wish to sacrifice.

4.2 In the consensus layer

The consensus layer relies on the ledger layer for the functions to manipulate the ledger state, but the design of the consensus layer relies on these functions being pure.

The consensus layer also relies on the use of in-memory persistent[1] data structures. It keeps the ledger states for the last k blocks of the chain in memory. The consensus layer also evaluates speculative ledger states along candidate chains, that may be adopted or discarded. Overall, there is not just a single logical ledger state in use at once, there are many related ones. There are enormous opportunities for sharing between these related ledger states and the use of persistent data structures takes full advantage of that such that the cost is little more than the cost of a single ledger state. The incremental cost is proportional to the *differences* between the states.

Having quick and easy access to the ledger states of the last k blocks is not a design accident. An important design principle for the Cardano node has been to balance the resource use in all interactions between honest nodes and potentially dishonest nodes and thus resist DoS attacks. This design principle led us to an Ouroboros design that involves efficient access to recent ledger states. Kanjalkar et al. [2019] describe the consequences of the failure to adopt this design principle. They identify a pair of flaws in the design of many other PoS blockchain implementations which lead to resource imbalances that can be exploited to mount DoS attacks. This was reported in the popular press as the so-called 'fake stake attack'[2]. One of the design flaws involves not having efficient access to recent ledger states and consequently postponing block body validation to the last possible moment.

In our design to store much of the ledger state on disk it is essential that we preserve the ability to efficiently access and use the ledger states for the last k blocks. Our resistance to DoS attacks depends on it.

Furthermore, chain selection within consensus relies on the ability to evaluate the validity of candidate chains, without yet committing to them. This also relies on being able to compute derived ledger states

5 Terminology and our perspective on databases

We will make use of the terminology of databases as well as Cardano blockchain terminology. This is potentially confusing since databases involve transactions, and transactions are also an important concept in blockchain ledgers.

We will mostly talk about transactions in the database sense. As it turns out the processing of transactions in the database sense corresponds to processing of whole blocks in the blockchain (and not the transactions within block).

We will take a relatively abstract view of databases. For the most part we will consider databases simply as logical values, without any particular implication of a choice of representation. Where it is important to imagine that a representation would be in-memory or on-disk then we will try to be clear about it. In this same spirit we will talk (and reason) about multiple logical values of a database, even though real databases typically only allow access to the 'current' value of the database. In this abstract view of databases a transaction is simply a way to get from one logical value of the database to another.

This is exactly the same mathematical perspective we take with functional programming: that new values are defined based on old. It is also exactly how we define our ledger rules as functions on the ledger state: given an one state it yields an updated state (with the old one still available). A database perspective on the ledger state would say that the ledger state itself is the database and that applying blocks are the transactions on that database state. This is the perspective we will take.

[1]That is persistent in the sense of purely functional data structures, not persistent as in kept on disk

[2]For example `https://www.zdnet.com/article/security-flaws-found-in-26-low-end-cryptocurrencies/`

6 General approach

We wish to deviate as little as possible from the general design of the current ledger and consensus layers, particularly their style using pure functions and persistent data structures.

The ledger state will be maintained using a partitioned approach with both in-memory and on-disk storage. The large mappings will be kept primarily on disk, and all other state kept in memory.

We will manipulate the state using pure functions over pure in-memory data structures. This relies on two main ideas: reading the data into memory in advance, and working with differences of data structures.

6.1 Inspiration from the 'anti-caching' database architecture

Our general approach takes inspiration from DeBrabant et al. [2013] who describe an OLTP[3] database architecture that they call 'anti-caching'.

The anti-caching architecture reverses the usual notion that a database's value is represented on disk, with some of the data cached in memory for efficiency. Instead the perspective is that the logical value of the database is represented in memory, with much of the data evicted from memory to disk. This is the essence of the eponymous anti-cache: the disk is used as an anti-cache to evict in-memory data, rather than using memory as a cache for disk data.

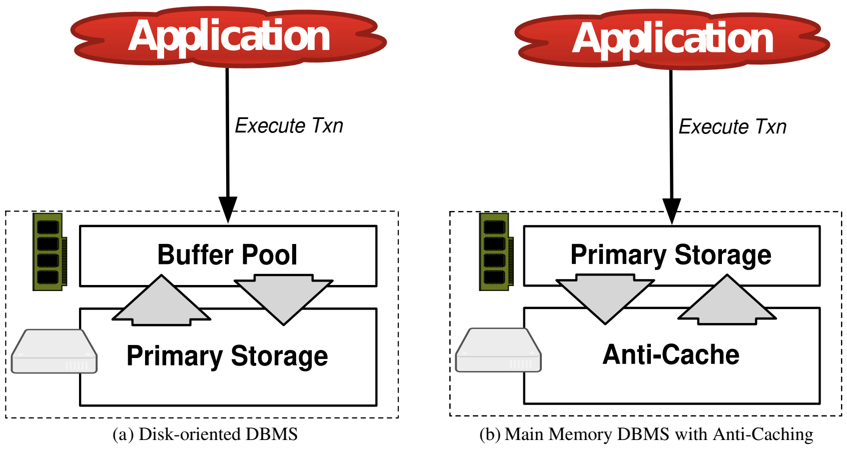


Diagram by DeBrabant et al. [2013, Figure 1] illustrating the difference in database architecture: "In (a), the disk is the primary storage for the database and data is brought into main memory as it is needed. With the anti-caching model in (b), memory is the primary storage and cold data is evicted to disk."

One of the ideas we borrow is that the logical value of the database is determined by the in-memory data, with large parts of the data residing on-disk. This means that as a matter of principle we do not need to constrain ourselves to keeping all data on disk. We merely need to keep most of it on disk. Where there are efficiency or design complexity benefits we can choose to keep data in memory, provided that we do not exhaust our memory budget overall.

Another feature of the anti-caching architecture is that all data required to process a database transaction must be in memory. If some or all of the data required by a transaction is not in memory then the database management system will first bring it back into memory from the anti-cache on disk and then retry the transaction. In principle this approach can deal with transactions that do dependent reads based on earlier reads (by multiple rounds of bringing data

[3]OLTP stands for Online Transaction Processing, and is a style of database workload. A blockchain ledger database is such a workload.

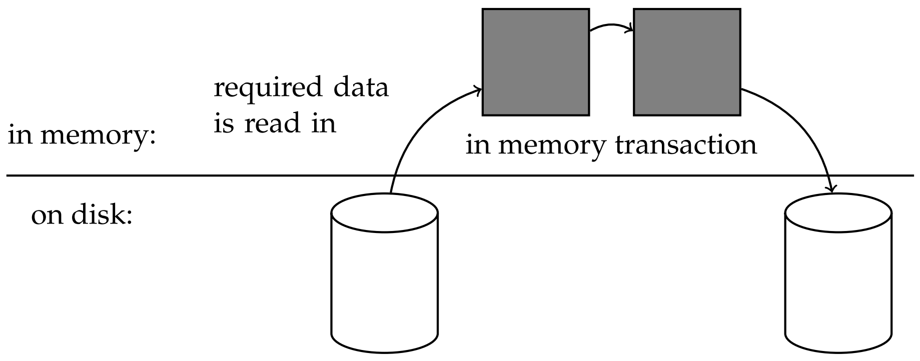
into memory and retrying) but it is certainly simpler if all the data required can be identified up front.

This is the other main idea that we borrow: to do all the database transaction processing in memory. This is a very attractive idea for our context because it enables the transaction processing to be implemented using pure functions operating on in-memory data structures. Unlike in the anti-caching design, we will rely on being able to identify *up front* all the data that will be needed to process a transaction so that there is no need to have a retry loop.

Our approach is not a full implementation of anti-caching. In particular we do not use a cache or an anti-cache to decide which data to keep in memory versus on disk. Instead we use a simple static policy which is appropriate for our use case.

6.2 Reading data into memory in advance

We will arrange to know in advance which parts of the ledger state may be used by the ledger rules, and we will read the data from disk into memory in advance. This allows the actual transformation to be performed on in-memory data structures and to be expressed as a pure function – minimising the required changes to the implementation of the ledger rules. We simply bring into memory the subset of the data that we will need. This subset is typically small.

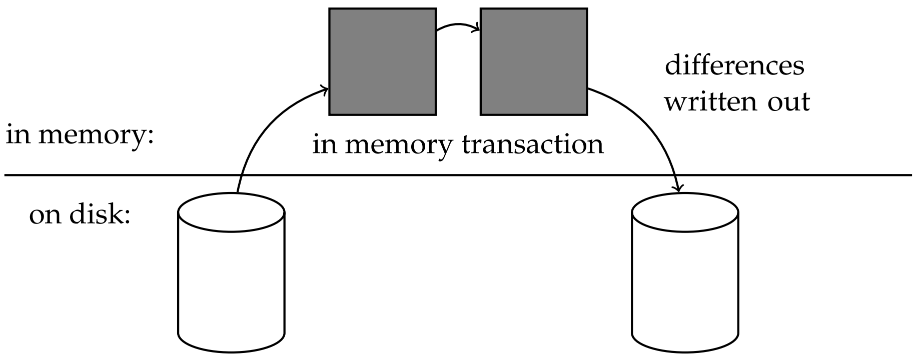


As an example of this, consider validating a ledger transaction including its UTxO inputs: we know we will need to look up the transaction inputs in the UTxO mapping. Which inputs we will need is clearly known in advance as they are explicit in the transaction itself. We believe that in general for the Cardano ledger rules we can determine all the required mapping entries and that there are no dynamic dependencies that cannot be discovered in advance. This fact will enable us to keep the design simpler.

We do not use a cache (or anti-cache): we will *always* read the required data into memory. As we have discussed previously [Wilson and Coutts, 2021] the data access patterns do not substantially benefit from caching. This choice keeps the design simpler.

6.3 Differences of data structures

We will make use of *differences* of data structures. In particular we will arrange for the ledger rules to return differences and it is these differences that can be applied to the on-disk data structures (e.g. as inserts, updates and deletes for on-disk tables).



The simplest scheme, as in the diagram above, would be to write differences back to disk immediately. As we will discuss, we will actually want to hold the differences in memory across many transactions and flush them to disk later.

6.4 Partitioned in-memory/on-disk representation

To meet our targets for memory use we must keep the bulk of the ledger state on disk, but as mentioned already (in Section 6.1) it is not necessary to keep the *entire* ledger state on disk. We can achieve a substantially simpler design if we partition the state such that *only* large key/value mappings are kept on disk, and all other data remains in memory. This approach is simpler in several ways.

- Rather than solve the general problem of keeping complex compound data on disk, we can reduce it to the well-understood problem of on-disk key-value stores.
- As mentioned in Section 6.2, we need to be able to predict which parts of the ledger state will need to be fetched from disk. If it is only the large mappings that are on disk then we do not need to consider which other 'miscellaneous' parts of the ledger state are needed since those parts are always in memory. This substantially reduces the problem.
- As mentioned in Section 6.3, we need to be able to represent and manage differences of data kept on disk. Differences of key/value mappings are straightforward and we can avoid the general problem of differences of complex data structures.

The observation that we have made about the Cardano ledger state is that while its structure is relatively complex, with many nested parts, most of it is relatively small. Only a few parts of the ledger state are really large, and those parts are all finite mappings. Thus we believe this approach will be sufficient to ensure the memory needed remains within the memory available. The requirements for scale and resource use are given in the previous document [Wilson and Coutts, 2021, Section 3].

Furthermore all the large finite mappings in the ledger state have relatively simple key and value types that can be represented as short byte strings. This allows them to be represented as on-disk key-value stores, which gives us a wide choice of store implementations. Previous analysis [Wilson and Coutts, 2021, Section 5] indicates that the performance requirements of the different mappings are all compatible, so we believe we can use a single key-value store implementation for all the on-disk mappings.

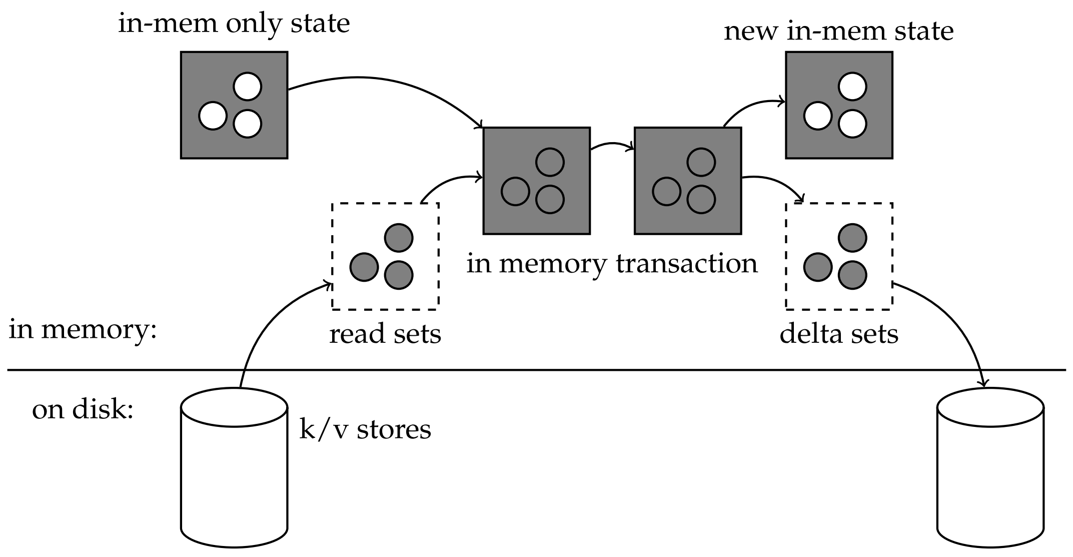
As for what counts as a large mapping, we draw the dividing line between 'large' and 'small' between those that are proportional to the number of stake addresses (or bigger) and those that are proportional to the number of stake pools (or smaller). That is mappings with sizes proportional to the number of stake pools – or something smaller than the number of stake pools – will be kept in memory. On the other hand mappings with sizes proportional to the number of stake address – or bigger than the number of stake addresses – should be kept on disk. The table below lists a selection of important mappings, what their size is proportional to, and whether they will be stored in memory or (primarily) on disk.

Mapping:	Size proportional to:	Location:
The UTxO	number of UTxO entries	on disk
Delegation choices	number of stake addresses	on disk
Reward account balances	number of stake addresses	on disk
Stake address pointers	number of stake addresses	on disk
Stake distribution (by stake address)	number of stake addresses	on disk
Stake distribution (by pool)	number of stake pools	in memory
Stake pool parameters	number of stake pools	in memory
Protocol parameters	constant	in memory

Note in particular that the consensus layer needs rapid and random access to the stake distribution (by block producer, i.e stake pool) to be able to validate block headers. Therefore performance concerns dictates that at least one mapping proportional to the number of stake pools needs to be kept in memory.

For mappings based on the same key it may or may not make sense to combine them into a single on-disk store. Combining them may save space but depending on the access pattern and on-disk store implementation we may obtain better performance by keeping them separate.

In a design where the state is partitioned between large on-disk tables and all other state in memory, the pattern for performing a transaction is as depicted below. We read the required data from the on-disk tables and combine it with the in-memory state, to give a combined state we can use to perform the transaction. The transaction result is split again between the new in-memory state, and differences on the large mappings which can then be applied to the on-disk tables.



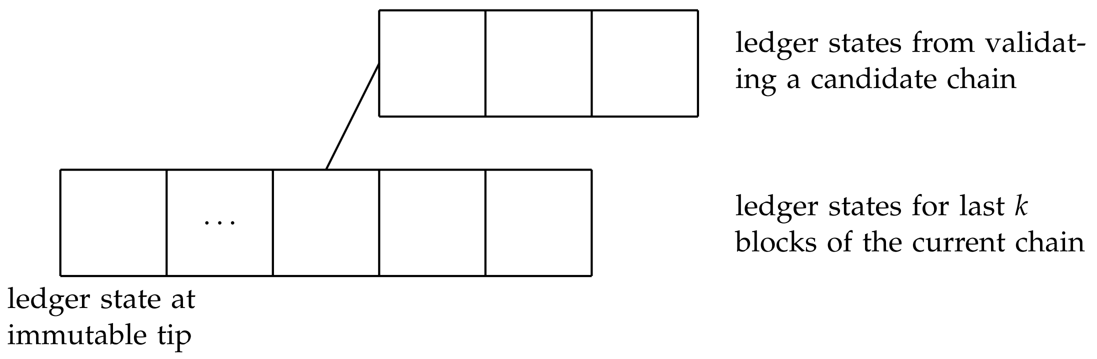
Finally, note that for the parts of the ledger state that are kept in memory, there does still need to be a mechanism to save and restore the state when the node shuts down and restarts. The intention is to use the same approach as the consensus layer uses now: writing snapshots to disk from time to time and replaying from the most recent snapshot upon start up. Only minor changes to this scheme are necessary to account for the on-disk mappings. To achieve a consistent snapshot of the overall state it will be necessary to take snapshots of the on-disk mappings and of the in-memory data for the exact same state (i.e. corresponding to the ledger state of the same chain). If the snapshots of the on-disk and in-memory parts were not synchronised it would not be possible to replay the subsequent changes upon start-up.

6.5 Access to multiple (logical) ledger states

As discussed in Section 4.2, the consensus design relies on having efficient access to multiple ledger states, corresponding to the k most recent blocks on the current chain. Furthermore, the chain selection algorithm needs to compute ledger states along candidate chains without yet committing to them. Evaluating the validity of candidate chains involves computing the ledger state, block by block, but if the chain turns out to be invalid then we must discard it and the corresponding ledger state. In particular in this situation we must not change our current chain or its corresponding ledger state.

Thus the consensus design demands that we have the ability to manipulate multiple logical ledger states. On the face of it this requirement would appear to be hard to satisfy using traditional on-disk data structures or database management systems which only provide a single 'current' value.

We also discussed in Section 4.2 that the existing consensus design relies on persistent data structures so that keeping many ledger states costs little more than a single state. We noted that the incremental cost of each extra copy is proportional to the differences between the states. Of course this only works because the states are *closely related*. More specifically, all the ledger states the node needs to handle are derived from a common state: the ledger state of the 'immutable tip' of the current chain. This is the ledger state for the tip of the chain if were to remove the most recent k blocks. Obviously all the ledger states for the last k blocks are related to this state by application of the ledger rules. The same holds for the ledger states of any candidate chains that we need to validate since they must have an intersection within the last k blocks.

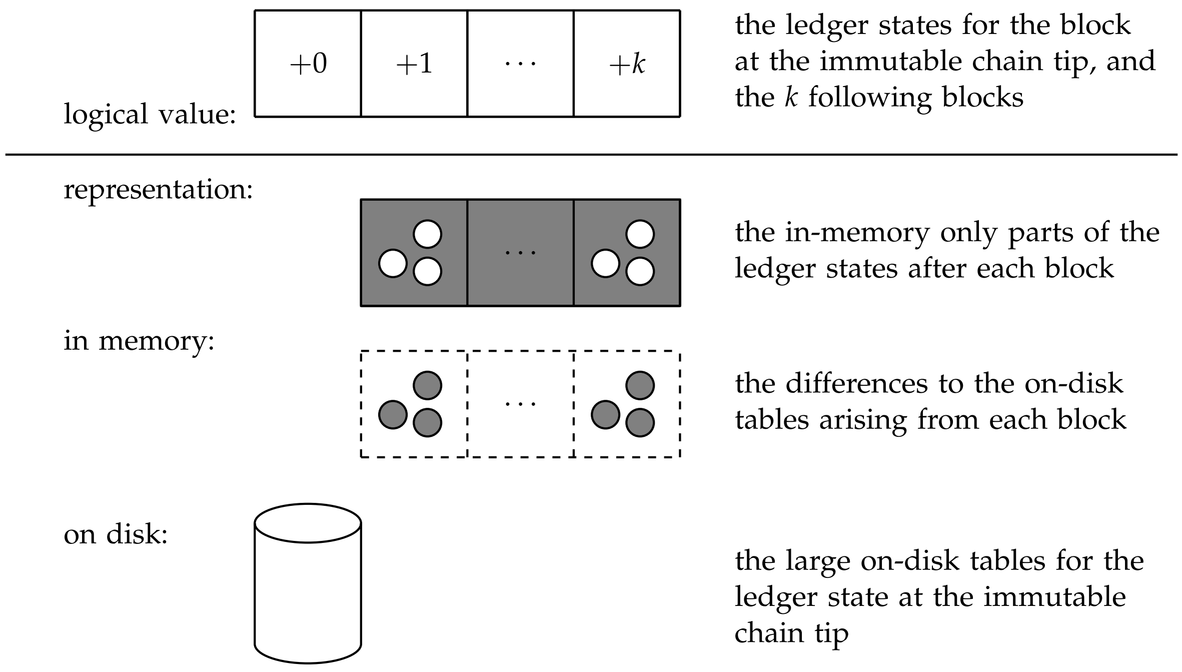

ledger states from validating a candidate chain
...
ledger states for last k blocks of the current chain
ledger state at immutable tip

In summary, we know that the differences between all the ledger states that we need to manipulate are relatively small (compared to the size of the ledger state itself), and they are all derived from one ledger state at the 'immutable tip'. Using persistent data structures is one way to take advantage of this property. Another way is to *represent the differences explicitly*, and use that to construct (on-demand) the multiple logical states (or parts thereof).

The design we choose to take is as follows:

- we will keep a single copy of the ledger state (k/v mappings) on disk;
- that copy will corresponding to the ledger state at the immutable tip, which is the common point of all other states;
- we will represent all other derived ledger states using differences from the common state;
- all these differences will be maintained in memory.

Or in diagram form:


logical value:
+0
+1
···
+k
the ledger states for the block at the immutable chain tip, and the k following blocks
representation:
···
the in-memory only parts of the ledger states after each block
in memory:
···
the differences to the on-disk tables arising from each block
on disk:
the large on-disk tables for the ledger state at the immutable chain tip

This design resolves the tension: it uses only a single on-disk value at any one time, which lets us use traditional database techniques, and yet it also lets us efficiently work with multiple logical values of the database state at once.

We must of course assess the memory use of this approach. It involves keeping the changes from the last *k* blocks in memory. Rough estimates suggest that by the time we hit the stretch target of 200 TPS then we should expect the representation of the differences to need on the order of a few gigabytes of memory. As we noted previously [Wilson and Coutts, 2021, section 3.3.5] it would be impractical to operate a public system at such TPS rates because of the high resource use, but private instances may be practical and in such cases using a few GB of memory would be acceptable.

6.6 Enabling I/O pipelining

As discussed in the initial report [Wilson and Coutts, 2021, sections 6.1 and 8.8] it is expected that ultimately it will be necessary to make use of parallel I/O to hit the stretch performance targets. This is because the expectation is that disk I/O (rather than network or CPU) would be the bottleneck for very high throughput validation of blockchains.

We may not make use of parallelism in an initial implementation but if we are to keep open the option to use parallelism later then it is necessary for the interface between the application and the disk storage layer to expose the opportunities for parallelism. Thus we wish to find an interface that allows for I/O parallelism.

It is worth keeping in mind how much parallel I/O we need to saturate a modern SSD. It is on the order of 32 – 64 IOPS.

Since blockchains are mostly linear in nature (being a chain) the opportunities for I/O parallelism come from batching and pipelining. In context we can think of these opportunities as follows

Batching: This is submitting a batch of I/O operations and waiting and collecting them all. For example we could submit all the I/O reads for a single block in one go.

A block with 16 transactions with 2 UTxO inputs each would generate 32 read IOPS. So we can see that large blocks could individually temporarily saturate an SSD. Note that with just batching there is no overlapping of computation with I/O, since we wait for the I/O to complete and then use the results.

Pipelining: This is submitting a (typically) continuous stream of I/O operations in advance of when their results are needed, and collecting each result (usually) in time before it is needed. A useful analogy is a queue where new I/O operations are added at one end and results arrive eventually at the other end, and the queue should be kept sufficiently full to maximise performance.

For example while we are doing the CPU work to process one block we can have submitted the I/O operations for one or more subsequent blocks so that their results are available by the time we come to process them.

Note that this involves overlapping computation with I/O. It also in principle the I/O queue can be kept full: we can avoid any gaps between blocks when no I/O is being performed.

Given this, it is clear that pipelining is superior in terms of achieving enough I/O parallelism to saturate an SSD, but is also more complex to arrange. The opportunity for batching arises naturally from processing blocks as a unit. In practice if pipelining is used, it would be used with batching as a pipeline of batches (per block).

For our purposes for this stage of design we simply need to ensure that the scheme for disk I/O makes it possible to take advantage of I/O pipelining. It is then a design decision for the consensus layer to decide where it is practical to take advantage of pipelining. It may only be worth attempting to use pipelining for bulk sync situations, and not attempting to use it opportunistically such as when switching forks. Using only batching is likely to be sufficient for normal operation when the node is already in sync. Indeed using batching only may be sufficient for an initial integration that is not yet aiming for the higher throughput targets.

7 Notation and properties of differences

In this section we briefly review the notation and properties of differences that we will use in later sections. We roughly follow the presentation by Atkey [2015].

We start with a set A of values of our data structure of interest. This set will typically be a finite mapping, or some small collection of finite mappings.

For some sets we may be able to find a corresponding set ΔA of *differences* or *changes* on values from the set A. In particular we will see that this is possible for finite mappings. For $a \in A$ we will write about corresponding differences on a as $\delta a \in \Delta A$. Note that δ is not an operator but a naming convention for values that are differences.

We can *apply* a difference to a value to produce a new value: given $a \in A$ and $\delta a \in \Delta A$ we can use the apply operator $\triangleleft$ to give us $(a \triangleleft \delta a) \in A$

The differences ΔA form a monoid with an associative operator $\diamond$ and a zero element $\mathbf{0} \in \Delta A$. This means we can composes changes, and we can always have no change.

In addition to the monoid laws we have a couple straightforward laws involving applying changes. The zero change is indeed no change at all

$$\forall a \in A.\ a \triangleleft \mathbf{0} = a \tag{1}$$

and applying multiple changes is the same as applying the composition of the changes

$$\forall a \in A.\ \forall \delta b, \delta c \in \Delta A.\ (a \triangleleft \delta b) \triangleleft \delta c = a \triangleleft (\delta b \diamond \delta c) \tag{2}$$

We will use the following $\Diamond\!\sum$ notation for a n-way monoidial sum, meaning simply repeated use of the associative $\diamond$ operator. An empty 0-way sum is of course defined as the zero change.

$$\Diamond\!\sum_{i=0}^{n} \delta a_i = \delta a_0 \diamond \delta a_1 \diamond \delta a_2 \diamond \ldots \delta a_n$$

We will also talk about functions that transform values, and corresponding functions that compute differences. That is if we have a function $f : A \to A$ then a corresponding difference function would be $\delta f : A \to \Delta A$. A difference function must satisfy the property that applying the change gives the same result as the original function

$$a \lhd \delta f(a) = f(a) \tag{3}$$

We will sometimes be able to derive difference functions from the original transformation functions.

8 Abstract models of hybrid on-disk/in-memory databases

In this section we will look at abstract mathematical models of databases. This is not specific to the Cardano ledger state, or indeed any specific database. Our use of the term 'database' in this section is quite abstract: we mean only some collection of data. We will discuss how different models correspond to different implementation strategies, including in-memory, on-disk or hybrid data storage.

There are a few reasons to consider these models.

Lucid descriptions There is the usual reason for abstraction that, omiting unnecessary details can make the ideas easier and shorter to describe.

Imagining implementation strategies Another reason is that although these are abstract models, they can be seen to correspond to certain implementation approaches. For example we will refer to certain terms as represting on-disk or in-memory data structures. Obviously mathematically there is no such distinction but it is very useful to see the correspondence to an implementation strategy. It lets us start to think about implementation constraints and whether a design seems plausible.

Seeing equivalences By describing the models in mathematical terms we may be able to show mathematical equivalences between models. This is especially useful when we have a model that corresponds to an preferred implementation approach and we can show that it is mathematically equivalent to some reference model.

This section makes use of the idea and notation of the calculus of differences from the previous section.

8.1 Simple sequential databases

This is a very simple model. In this model there is an initial database state $db_0 \in DB$ and a series of state transformation functions $tx_0, tx_1, \ldots \in DB \to DB$ which we can also think of as database transactions. This gives rise to a series of database states $db_0, db_1, \ldots$ just by applying each transformation function to the previous state $db_{n+1} = tx_n(db_n)$.

$$\begin{tikzcd}
db_0 \arrow[r, bend right, "tx_0"'] & db_1 \arrow[r, bend right, "tx_1"'] & db_2 \arrow[r, bend right, "tx_2"'] & db_3 & \cdots
\end{tikzcd}$$

$$\begin{aligned}
db_1 &= tx_0(db_0) \\
db_2 &= tx_1(db_1) \\
&\vdots
\end{aligned}$$

This model serves as an important semantic reference point for the more complicated models below. we will want to show that some of our more complicated models are semantically equivalent to this simple one.

If we think of this model in terms of what kind of implementation strategy it most clearly represents, then it would probably be a simple in-memory design. That is a design where the whole database is a simple program value that is transformed with pure functions.

8.2 Change-based databases

In this model we want to introduce two concepts

1. the use of transaction difference functions and applying differences; and
2. identifying the subset of values that each transaction needs.

It is otherwise just a simple sequence of changes. The use of these two concepts makes this a simple but reasonable model of an on-disk database with in-memory transaction processing. That is, a database where the data is stored on disk but all transaction processing is performed on in-memory data structures. Using a subset of values corresponds to reading the data in from disk, while obtaining and applying differences corresponds to writing changes back to disk.

Whereas in the previous model we had a series of transaction functions $tx_0, tx_1, \ldots$, in this model we will have difference functions $\delta tx_0, \delta tx_1, \ldots : DB \to \Delta DB$. These are required to be proper difference functions, obeying the property

$$db \triangleleft \delta tx(db) = tx(db) \tag{4}$$

For each transaction we will also identify the subset of the database state that the transaction needs. This will typically take the form of a set of keys $ks \in \operatorname{dom} DB$ (or collection of sets of keys) and performing a domain restriction $db|_{ks} \in DB$. So there will key sets $ks_0, ks_1, \ldots$ corresponding to the transactions $\delta tx_0, \delta tx_1, \ldots$. We will require that the transaction really does only make use of the subset by requiring the property that the transaction function gives the same result on the subset as the whole state

$$\delta tx_i(\,db|_{ks_i}) = \delta tx_i(db) \tag{5}$$

A lemma that will prove useful later is that domain restriction commutes with applying changes

$$db|_{ks} \triangleleft \delta db = \ (db \triangleleft \delta db)|_{ks} \tag{6}$$

This lemma either needs to be proved universally or we will need to make it a required property of the differences.

We can now construct the series of database states $db_0, db_1, \ldots$ by applying the changes from each transaction to the previous database state.

$$\begin{tikzcd}
db_0 \arrow[r, bend right, "{\delta tx_0, ks_0}"'] & db_1 \arrow[r, bend right, "{\delta tx_1, ks_1}"'] & db_2 \arrow[r, bend right, "{\delta tx_2, ks_2}"'] & db_3 & \cdots
\end{tikzcd}$$

$$\begin{aligned}
db_1 &= db_0 \triangleleft \delta tx_0(\,db_0|_{ks_0}) \\
db_2 &= db_1 \triangleleft \delta tx_1(\,db_1|_{ks_1}) \\
&\vdots
\end{aligned}$$

It is straightforward to see how this is equivalent to the simple model.

$$\begin{aligned}
& db_{i+1} = db_i \lhd \delta tx_i(db_i|_{ks_i}) \\
\equiv\quad & \{\text{by the restriction property Equation (5) that } \delta tx_i(db|_{ks_i}) = \delta tx_i(db)\} \\
& db_{i+1} = db_i \lhd \delta tx_i(db_i) \\
\equiv\quad & \{\text{by the difference function property Equation (4) that } db \lhd \delta tx(db) = tx(db)\} \\
& db_{i+1} = tx_i(db_i)
\end{aligned}$$

And hence, just as in the simple model, it is the case that

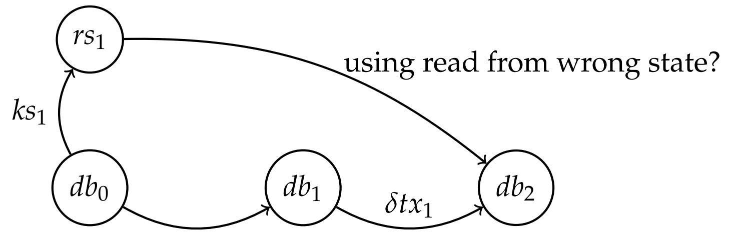
$$\begin{aligned}
db_1 &= tx_0(db_0) \\
db_2 &= tx_1(db_1) \\
&\vdots
\end{aligned}$$

8.3 Change-based pipelined databases

Here is where things start to get interesting and tricky. As discussed in Section 6.6 we wish to pipeline reads from disk to provide the opportunity to use parallel I/O. Providing this opportunity is not something that we can hide, and it will have to be explicit in how we manage the logical state of the database. So the purpose of this model is to provide a reasonable correspond to an implementation that could use pipelined reads. We will also want to show that it is nevertheless mathematically equivalent to the simple model.

The goal with the pipelining of I/O reads is to initiate the I/O operations early so the results are already available in memory by the time they are needed later. This allows the I/O to be overlapped with computation, and it also allows a substantial number of I/O operations to be in progress at once, which is what provides the opportunity to use parallel I/O.

The difficulty however is that initiating the reads early means that the state of the database in which we initiated the reads is not necessarily the same state as the one in which we use the read results to perform a transaction. In the diagram below we see an example where a read of key set ks_1 is performed against database state db_0 to give us a read set $rs_1 = db_0|_{ks_1}$. The transaction δtx_1 using this read set rs_1 is being applied starting from database state db_1 which is *not* the same as the database state db_0 the read was performed against. Any updates applied in db_1 that might affect rs_1 would be lost and we would get the wrong result.



Contrast this with the model in the previous section. That model uses 'reads' from the database (modelled as $db_i|_{ks_i}$) but the reads are done from the same database state as the transaction itself.

Apparently with pipelining we no longer have straightforwardly sequential state updates. Of course pipelining is only acceptable if can find some way to make it semantically equivalent to the sequential version.

$$\begin{tikzcd}
rs_1 \arrow[drr, bend left, "\text{adjust } rs_1 \text{ as-if performed against } db_1"] & & \\
db_0 \arrow[u, bend left, "ks_1"] \arrow[r, bend right] & db_1 \arrow[r, bend right, "\delta tx_1"] & db_2
\end{tikzcd}$$

The trick to restoring equivalence to the sequential version is to adjust the result of the reads so that it is as if they had been performed against the right state of the database.

We observed before that the problem was that any updates applied in db_1 that might affect rs_1 would be lost. Since we are working with differences we of course know *exactly* what changes were applied to db_1. We can apply those same changes to the read set rs_1. In the example above those changes are exactly the ones performed by the previous transaction δtx_0.

More generally the changes we want to apply are all those that occurred between the state against which the read was performed and the state in which the transaction using the read results is to be applied. Thus in our designs we will have to carefully track and apply the changes from where a read was initiated to where it is used.

If we can do so successfully however it seems clear that we can obtain a arbitrary depths of pipelining, at the memory cost of tracking the intervening changes. In the diagram below we illustrate a series of transactions using pipelined reads with a fixed depth of one.

$$\begin{tikzcd}
rs_1 \arrow[ddr, "\delta tx_1"] & rs_2 \arrow[ddr, "\delta tx_2"] & rs_3 \arrow[ddr, "\delta tx_3"] & rs_4 & \cdots \\
rs_0 \arrow[dr, "\delta tx_0"] & & & & \\
db_0 \arrow[u, bend left, "ks_0"] \arrow[uu, bend right, "ks_1"'] \arrow[r, bend right, "\_ \lhd \_"'] & db_1 \arrow[uu, bend left, "ks_2"] \arrow[r, bend right, "\_ \lhd \_"'] & db_2 \arrow[uu, bend left, "ks_3"] \arrow[r, bend right, "\_ \lhd \_"'] & db_3 \arrow[uu, bend left, "ks_4"] \arrow[r, bend right, "\_ \lhd \_"'] & db_4 & \cdots
\end{tikzcd}$$

$$\begin{aligned}
db_1 &= db_0 \lhd \delta db_0 && \textbf{where} && \delta db_0 = \delta tx_0(rs_0) && \textbf{and} && rs_0 = db_0|_{ks_0} \\
db_2 &= db_1 \lhd \delta db_1 && \textbf{where} && \delta db_1 = \delta tx_1\left((rs_1 \lhd \delta db_0)|_{ks_1}\right) && \textbf{and} && rs_1 = db_0|_{ks_1} \\
db_3 &= db_2 \lhd \delta db_2 && \textbf{where} && \delta db_2 = \delta tx_2\left((rs_2 \lhd \delta db_1)|_{ks_2}\right) && \textbf{and} && rs_2 = db_1|_{ks_2} \\
&\vdots
\end{aligned}$$

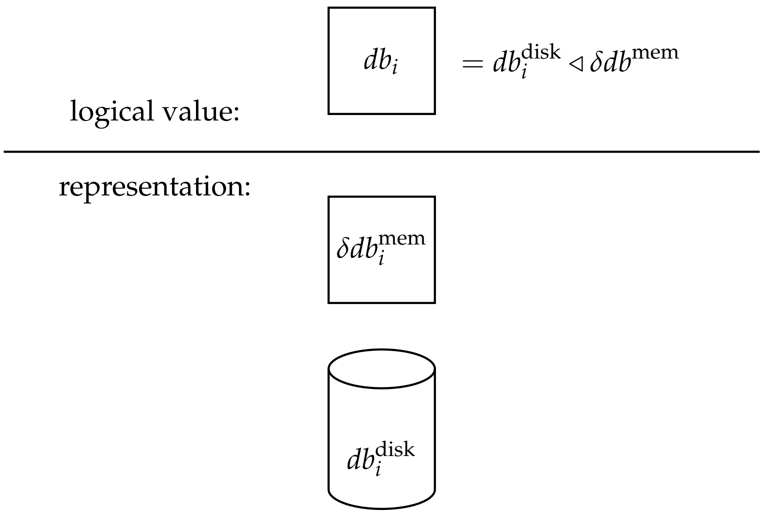
We need to clarify how exactly this is equivalent to the simple sequential model. We now have a more interesting recurrence relation than in previous models so we argue by induction. Due to the setup step for the pipelining, we start from $i = 1$ rather than $i = 0$.

$$
\begin{aligned}
& db_{i+1} = db_i \lhd \delta db_i \quad \textbf{where} \quad \delta db_i = \delta tx_i \left((rs_i \lhd \delta db_{i-1})|_{ks_i} \right) \quad \textbf{and} \quad rs_i = db_{i-1}|_{ks_i} \\
\equiv \quad & \{\text{by substitution of } rs_i \} \\
& db_{i+1} = db_i \lhd \delta db_i \quad \textbf{where} \quad \delta db_i = \delta tx_i \left(\left(db_{i-1}|_{ks_i} \lhd \delta db_{i-1} \right) \Big|_{ks_i} \right) \\
\equiv \quad & \{\text{by the domain restriction lemma Equation (6), and idempotency of restriction} \} \\
& db_{i+1} = db_i \lhd \delta db_i \quad \textbf{where} \quad \delta db_i = \delta tx_i \left((db_{i-1} \lhd \delta db_{i-1})|_{ks_i} \right) \\
\equiv \quad & \{\text{by induction hypothesis } db_i = db_{i-1} \lhd \delta db_{i-1} \} \\
& db_{i+1} = db_i \lhd \delta db_i \quad \textbf{where} \quad \delta db_i = \delta tx_i \left(db_i|_{ks_i} \right) \\
\equiv \quad & \{\text{by substitution of } \delta db_i\} \\
& db_{i+1} = db_i \lhd \delta tx_i(db_i|_{ks_i}) \\
\equiv \quad & \{\text{by the restriction property Equation (5) that } \delta tx_i(db|_{ks_i}) = \delta tx_i(db)\} \\
& db_{i+1} = db_i \lhd \delta tx_i(db_i) \\
\equiv \quad & \{\text{by the difference function property Equation (4) that } db \lhd \delta tx(db) = tx(db)\} \\
& db_{i+1} = tx_i(db_i)
\end{aligned}
$$

8.4 Hybrid on-disk / in-memory databases

In this model we go back to a sequential style for simplicity, but we represent the data as a combination of two parts. One will stand for data on disk and the other will stand for data in memory. The part on disk will be ordinary database values $db_i^{\text{disk}} \in DB$, while the part in memory will be database differences $\delta db_i^{\text{mem}} \in \Delta DB$. We define the overall logical value of the database to be the on-disk part with the in-memory changes applied on top.

$$db_i = db_i^{\text{disk}} \lhd \delta db_i^{\text{mem}}$$


db_i
$= db_i^{\text{disk}} \lhd \delta db^{\text{mem}}$
logical value:
representation:
δdb_i^{mem}
db_i^{disk}

We now need to see how performing transactions works in this representation. We will of course use the change-based style of transactions. Given the changes made by a transaction $\delta tx_i(db_i|_{ks_i})$ we would normally obtain the new state of the database by applying the changes to the previous state

$$db_{i+1} = db_i \lhd \delta tx_i(db_i|_{ks_i})$$

With the hybrid representation that is

$$db_{i+1} = \left(db_i^{\text{disk}} \lhd \delta db_i^{\text{mem}}\right) \lhd \delta tx_i(db_i|_{ks_i})$$

As we know from Equation (2), applying two sets of changes is equivalent to applying the composition of the changes, and we choose to make use of this.

$$db_{i+1} = db_i^{\text{disk}} \lhd \left(\delta db_i^{\text{mem}} \diamond \delta tx_i(db_i|_{ks_i})\right)$$

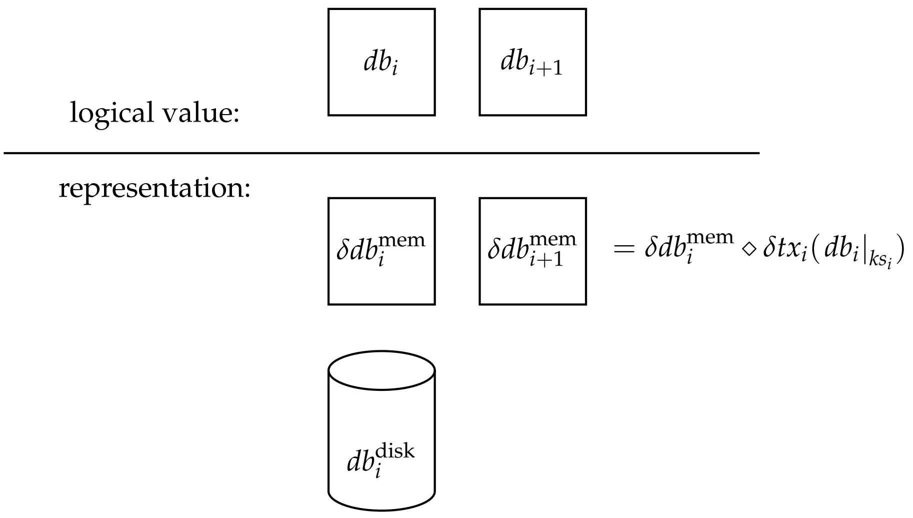
This now fits the same hybrid representation. We can define the new in-memory value to be

$$\delta db_{i+1}^{\text{mem}} = \delta db_i^{\text{mem}} \diamond \delta tx_i(db_i|_{ks_i})$$

to get

$$db_{i+1} = db_i^{\text{disk}} \lhd \delta db_{i+1}^{\text{mem}}$$

Or pictorally



db_i
db_{i+1}
logical value:
representation:
δdb_i^{mem}
$\delta db_{i+1}^{\text{mem}}$
$= \delta db_i^{\text{mem}} \diamond \delta tx_i(db_i|_{ks_i})$
db_i^{disk}

Notice that we have applied the transaction exclusively to the in-memory part, without changing the on-disk part. Obviously we cannot do this indefinitely or the part kept in memory will grow without bound.

In this approach we must changes to disk from time to time flush. Let us see how that might work. Of course flushing is not supposed to change the logical value of the database, just to shuffle data from the in-memory part to the on-disk part. Suppose we start from a state

$$db = db^{\text{disk}} \lhd \delta db^{\text{mem}}$$

We can in principle split the in-memory part into the composition of two sets of changes.

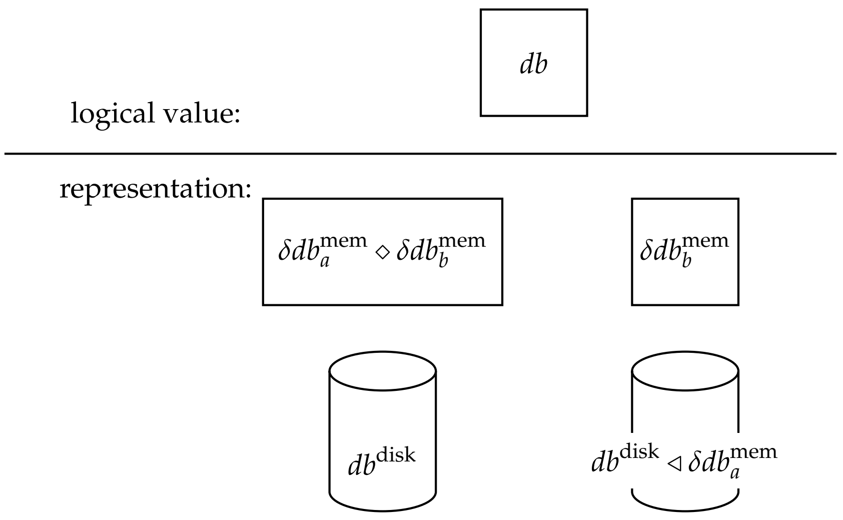
$$\delta db^{\text{mem}} = \delta db_a^{\text{mem}} \diamond \delta db_b^{\text{mem}}$$

The idea is that the first part will be flushed to disk and the second part will remain in memory. We have a lot of choice here. We can split this in any way we like. We could choose to flush everything to disk for example, by picking $\delta db_b^{\text{mem}} = \mathbf{0}$, but we can also choose to flush just a subset of changes.

Doing the flush is another straightforward application of Equation (2), but in the opposite direction.

$$db = db^{\text{disk}} \lhd (\delta db_a^{\text{mem}} \diamond \delta db_b^{\text{mem}})$$
$$\equiv$$
$$db = \left(db^{\text{disk}} \lhd \delta db_a^{\text{mem}}\right) \lhd \delta db_b^{\text{mem}}$$

We can interpret the application of changes $db^{\text{disk}} \lhd \delta db_a^{\text{mem}}$ as performing the writes to the on-disk database. Here is the same in pictorial style:



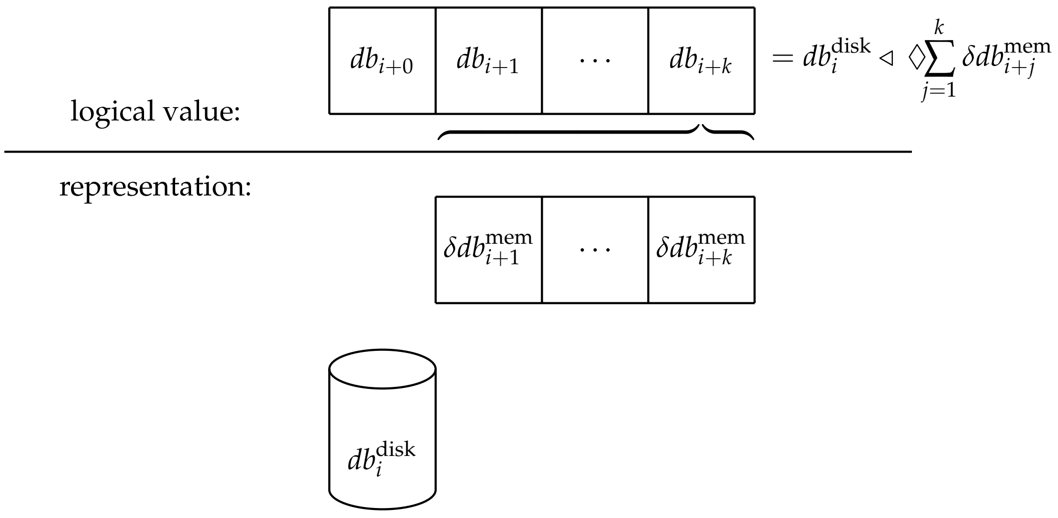
logical value:
db
representation:
$\delta db_a^{\text{mem}} \diamond \delta db_b^{\text{mem}}$
δdb_b^{mem}
db^{disk}
$db^{\text{disk}} \lhd \delta db_a^{\text{mem}}$

8.5 Multiple logical database states

In the models so far, while we have of course defined many logical values of a database, we have implicitly assumed that these are the evolving states of a single database over time. We now wish to model a situation in which we want to maintain and have efficient access to many logical values of a database at once.

This corresponds roughly to the situation we have in the consensus design where it is necessary to have efficient access to ledger states corresponding to the last K blocks[4]. We need these ledger states to be able to evaluate the validity of candidate chains that intersect with our own at any point within the last K blocks. In the current wholly in-memory consensus design we can maintain K ledger states at very little additional memory cost compared to a single ledger state. This takes advantage of persistent data structures. This works because the K ledger states are very closely related: being the ledger states obtained from applying a series of blocks.

The idea we want to model here is having a single on-disk state at once, but a whole series of in-memory differences, giving us a series of logical database values.



logical value:
db_{i+0}
db_{i+1}
$\cdots$
db_{i+k}
$= db_i^{\text{disk}} \lhd \Diamond\sum_{j=1}^{k} \delta db_{i+j}^{\text{mem}}$
representation:
$\delta db_{i+1}^{\text{mem}}$
$\cdots$
$\delta db_{i+k}^{\text{mem}}$
db_i^{disk}

As depicted in the diagram above, the representation consists of a single disk state and a series of in-memory differences. Each difference is the individual difference from performing a

[4]On the Cardano mainnet K is 2160, so quite a few

transaction. That is, we keep each difference and do not compose them together prematurely. Then each logical database value is the value of the on-disk state with the monoidial composition of the appropriate changes applied on top. That is, for the kth database value beyond the on disk state we have

$$db_{i+k} = db_i^{\text{disk}} \lhd \Diamond\sum_{j=0}^{k} \delta db_{i+j}^{\text{mem}}$$

Although we are interested in the compositions of differences we must keep the individual differences. The reason is that when we do flush changes to disk we need to discard the oldest in-memory differences which entails computing new monoidial compositions of the remaining differences.

One interesting implementation idea to manage both a sequence of K differences and also the efficient monoidial composition of them is to use a *finger tree* data structure [Hinze and Paterson, 2006]. A finger tree is parametrised over a monoidal measure. The choice in this case would be both the length and the differences. The length is the usual measure to support splitting at indexes. We would rely on splitting to obtain the sub-sequence of differences for evaluating a chain fork that starts from a recent point on the chain. The other part of the measure – the differences – would mean that the measure of any sequence or sub-sequence would be the overall monoidal composition of all the differences in that (sub-)sequence. The finger tree data structure takes care of efficiently caching the intermediate monoid measure values. Finger trees are already used extensively within the consensus implementation – including with measures other than simply length.

There's a few operations we need to be able to perform in this setting

1. Perform reads of sets of keys and 'fast forward' the resulting read set using the accumulated sequence of differences.
2. Replace any suffix of the sequence of logical database values. This corresponds to switching to a fork. The replacement sub-sequence can start anywhere in the sequence and the replacement can have any length sequence of differences arising from blocks. The very common special case is to append to the end of the sequence of logical database values. This corresponds to adding a block to the end of the chain, or adding more transactions to a mempool.
3. Flush changes to disk. We need to be able to take some of the oldest changes and apply them to the disk store.

These operations are straightforward generalisations of the operations we have already seen in Section 8.4. The first two are about constructing new logical database values by making new in-memory differences. In this representation it is simply appending or replacing a suffix of a sequence of changes. The flushing to disk is a straightforward instance of the flush operation from Section 8.4 where we choose the a point in the sequence that splits the differences we wish to flush from the remaining sequence of differences. The value that we apply to the on-disk state is the monoidal composition of the sequence of differences we wish to flush.

In section ?? we will look at a design for the consensus ledger state management and chain selection based on the abstract model here, relying only on the operations we have identified here.

8.6 Haskell encoding

We can encode these definitions in Haskell using a type class with an associated data family

```
class Monoid (Diff a) ⇒ Changes a where
  data Diff a
```

```
    applyDiff :: a → Diff a → a
```

The *Diff a* corresponds to the ΔA and *applyDiff* to the $\triangleleft$ operator. The *Monoid* (*Diff a*) superclass constraint captures the point that differences form a monoid. We notate the monoid unit element as $\varnothing$ and use $\diamond$ for composition operator.

8.7 Differences on *Map*

To give an intersting and useful example, we provide an instance for ordinary finite mappings based on ordered keys: Haskell's usual *Map* type.

The representation for a difference of a map will be itself a map, but with an element type that tracks the changes to the element.

```
data MapDiffElem a = MapElemDelete | MapElemInsert !a | MapElemUpdate !a
```

The *Changes* instance gives the representation and the apply operator. For maps the *applyDiff* is a merge of the underlying map with the map of changes. The strategy for the merge is:

- for elements only in the underlying map, to preserve them;
- for elements only in the overlay, to insert them; and
- for elements in both maps to apply the per-element change to the underlying value.

The code for the merge follows straightforwardly from the above.

```
instance (Ord k, Monoid a) ⇒ Changes (Map k a) where
  newtype Diff (Map k a) = MapDiff (Map k (MapDiffElem a))
  applyDiff :: Map k a → Diff (Map k a) → Map k a
  applyDiff m (MapDiff md) =
      Map.merge
        Map.preserveMissing
        (Map.mapMaybeMissing      insert)
        (Map.zipWithMaybeMatched apply)
        m
        md
    where
      insert :: k → MapDiffElem a → Maybe a
      insert _  MapElemDelete      = Nothing
      insert _ (MapElemInsert x)   = Just x
      insert _ (MapElemUpdate x)   = Just x
      apply :: k → a → MapDiffElem a → Maybe a
      apply _ _  MapElemDelete     = Nothing
      apply _ _ (MapElemInsert y)  = Just y
      apply _ x (MapElemUpdate y)  = Just (x ◇ y)
```

Finally we need that changes on maps be monoidal: with empty maps and composition of changes. We define the composition of differences on maps to be the point-wise composition of matching elements, which in turn relies on the *MapDiffElem* forming a semi-group.

```
instance (Ord k, Semigroup a) ⇒ Monoid (Diff (Map k a)) where
  ∅ = MapDiff Map.empty
instance (Ord k, Semigroup a) ⇒ Semigroup (Diff (Map k a)) where
  MapDiff a ◇ MapDiff b = MapDiff (Map.unionWith (◇) a b)
```

```
instance Semigroup a ⇒ Semigroup (MapDiffElem a) where
  MapElemDelete    ◇ MapElemDelete    = MapElemDelete
  MapElemInsert  _ ◇ MapElemInsert  y = MapElemInsert    y
  MapElemUpdate x  ◇ MapElemUpdate  y = MapElemUpdate (x ◇ y)

  MapElemInsert  _ ◇ MapElemDelete    = MapElemDelete
  MapElemDelete    ◇ MapElemInsert  y = MapElemInsert    y

  MapElemUpdate _  ◇ MapElemDelete    = MapElemDelete
  MapElemDelete    ◇ MapElemUpdate  y = MapElemInsert    y

  MapElemInsert  x ◇ MapElemUpdate  y = MapElemInsert (x ◇ y)
  MapElemUpdate _  ◇ MapElemInsert  y = MapElemInsert    y
```

9 Introducing a simple ledger example

To help us illustrate the API ideas we will want to use a simple but realistic example.

We will use a running example of a simplified ledger, with ledger state and ledger rules for validating blocks containing transactions. We start with various basic types needed to represent transactions and their constituent parts.

```
newtype Block  = Block [Tx]            deriving (Eq, Show)
data    Tx     = Tx [TxIn] [TxOut]     deriving (Eq, Show)
data    TxIn   = TxIn !TxId !TxIx      deriving (Eq, Ord, Show)
newtype TxId   = TxId Int              deriving (Eq, Ord, Show)
newtype TxIx   = TxIx Int              deriving (Eq, Ord, Show)
data    TxOut  = TxOut !Addr !Coin     deriving (Eq, Show)
newtype Addr   = Addr Int              deriving (Eq, Ord, Show)
newtype Coin   = Coin Int              deriving (Eq, Show)

instance Semigroup Coin where Coin a ◇ Coin b = Coin (a + b)
```

We will flesh out the example further in later sections.

10 Partitioned and hybrid in-memory/on-disk storage

As discussed in Section 6.4, only the large mappings will be kept on disk, with all other state kept in memory. So we have both partitioned and hybrid in-memory/on-disk storage. The distinction we are drawing is:

partitioned in the sense of keeping some parts of the state in memory, with other parts (primarily) on disk; and

hybrid in the sense of Section 8.4 where recent changes are kept in memory, while the bulk of the data is kept on disk.

In this section we will focus on the partitioning idea. We will not include the ideas of hybrid representations from Section 8.4 or of multiple logical database values discussed in Section 8.5. We will postpone incorporating these for later sections.

10.1 Parametrising over large mappings

The approach we will take is the following. The ledger state is a relatively complicated compound type consisting of many nested records and tuples. Consider this compound type as a tree where type containment corresponds to child-node relationships and atomic types correspond to leaf nodes. For example a pair of integers would correspond to a root node, for the pair itself, and two child nodes, for the two integers. Given this way of thinking of the type we then say that we will keep the root of the type in memory. Specific leaf nodes corresponding to the large mappings will be kept on disk. All other leaf nodes and interior nodes will also be in memory. So overall our ambition is to simulate a type that is in memory except for certain designated finite mappings which are kept on disk.

To realise this ambition we will use a number of tricks with the choice of the representation of mappings in the ledger state type. To do so we will parametrise the ledger state over the choice of the mapping type. We will then pick different choices of mapping for different purposes.

10.2 The example ledger state

To illustrate this idea we will return to our example ledger and define our ledger state. Note that the ledger state type is parametrised by the type of large mappings.

```
data LedgerState map =
    LedgerState {
      utxos    :: UTxOState map,
      pparams  :: PParams
    }
```

The *pparams* stands in for other state, like protocol parameters. The exact content of this does not matter for the example. It is merely a place-holder.

```
data PParams = PParams
```

The *utxos* represents some nested part of the overall ledger state. In particular it demonstrates two large mappings, a UTxO and an aggregation of the coin values in the UTxO by the address. Here is where we actually use the map collection type.

```
data UTxOState map =
    UTxOState {
      utxo     :: map TxIn TxOut,
      utxoagg  :: map Addr Coin
    }
```

We can recover a 'normal' in-memory ledger state representation by choosing to instantiate with the ordinary in-memory map type *Map* to give us *LedgerState Map*. We might use this for a reference implementation that does everything in-memory.

10.3 Always-empty mappings

We can also define an always-empty map type that has a nullary representation.

```
data EmptyMap k v = EmptyMap
```

If we pick this map type, i.e. *LedgerState EmptyMap*, it would give us a ledger state with no data for the large mappings at all. This gives us a kind of ledger containing only those parts of the ledger state that are only kept in memory. This will be useful.

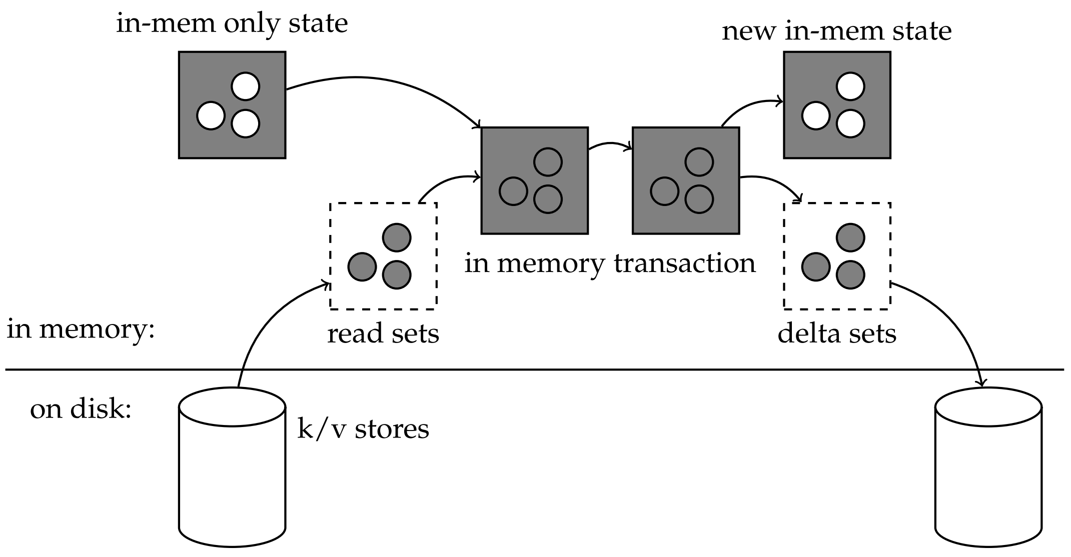
10.4 Selecting only the mappings

In addition to taking a ledger state type and picking the type of the large mappings within it, we will also need to work with *only* the large mappings without all of the other parts of the ledger state. We still want to be able to select the mapping representation.

We will use a class to capture the notion of

class *HasOnDiskMappings* (*state* :: $(* \to * \to *) \to *$) **where**
  data *OnDiskMappings state* :: $(* \to * \to *) \to *$
  projectOnDiskMappings :: *state map* $\to$ *OnDiskMappings state map*
  injectOnDiskMappings :: *OnDiskMappings state map* $\to$ *state any* $\to$ *state map*

Performing a transaction in this partitioned setting is depicted as follows.



We start on the left hand side with an on-disk store and an in-memory value. This value is those parts of the ledger state that are only kept in memory. The leaf types corresponding to the large mappings are replaced with nullary place holders.

11 Partial tracking mappings

In Section 8 we assumed that we could find difference functions δtx corresponding to the ordinary transaction functions tx. The correspondence we required was given in Equation (4), reproduced here as an aide-mémoire:

$$db \lhd \delta tx(db) = tx(db)$$

We would of course like to avoid manually writing such variants and proving the correspondence.

For the special case of functions on finite mappings, that use a limited number of operations, we can derive such difference functions automatically. This also generalises to combinations of finite mappings. As discussed in Section 6.4 ...

class *Mapping map* **where**
  lookup :: $Ord\ k \Rightarrow k \to map\ k\ a \to Maybe\ a$
  insert :: $Ord\ k \Rightarrow k \to a \to map\ k\ a \to map\ k\ a$
  delete :: $Ord\ k \Rightarrow k \to map\ k\ a \to map\ k\ a$
  update :: $Semigroup\ a \Rightarrow Ord\ k \Rightarrow k \to a \to map\ k\ a \to map\ k\ a$

A straightforward way to define

data *PTMap k a* = *PTMap* !(*Map k* (*Maybe a*))
                    !(*Diff* (*Map k a*))

```
instance Mapping PTMap where
  lookup k (PTMap m _) =
    case Map.lookup k m of
      Nothing → error "PTMap.lookup: used a key not fetched from disk"
      Just v    → v
  insert k v (PTMap m (MapDiff d)) =
    PTMap (            Map.insert k (Just v)              m)
          (MapDiff (Map.insert k (MapElemInsert v) d))
  delete k (PTMap m (MapDiff d)) =
    PTMap (            Map.insert k Nothing           m)
          (MapDiff (Map.insert k MapElemDelete d))
  update k v (PTMap m (MapDiff d)) =
    PTMap (            Map.insertWith (◇) k (Just v)              m)
          (MapDiff (Map.insertWith (◇) k (MapElemUpdate v) d))
```

References

Robert Atkey. The incremental λ-calculus and relational parametricity, 2015. https://bentnib.org/posts/2015-04-23-incremental-lambda-calculus-and-parametricity.html.

Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, September 2013. ISSN 2150-8097. doi: 10.14778/2556549.2556575. URL https://doi.org/10.14778/2556549.2556575.

Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006. doi: 10.1017/S0956796805005769.

Sanket Kanjalkar, Joseph Kuo, Yunqi Li, and Andrew Miller. *Short Paper: I Can't Believe It's Not Stake! Resource Exhaustion Attacks on PoS*, pages 62–69. 09 2019. ISBN 978-3-030-32100-0. doi: 10.1007/978-3-030-32101-7_4.

Douglas Wilson and Duncan Coutts. Storing the cardano ledger state on disk: analysis and design options, 2021.