Storing the Cardano ledger state on disk: API design concepts

AN IOHK TECHNICAL REPORT

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

Douglas Wilson
douglas@well-typed.com

Joris Dral
joris@well-typed.com
joris.dral@iohk.io

Version 0.6, August 2023

Changelog

0.6 Joris Dral (August 2023) — Editorial changes, final version

0.5 Duncan Coutts, Douglas Wilson (November 2021) — Unfinished version

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 models.

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 Ledger state handling in the current design

3.1 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.

3.2 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, which 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, 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.

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

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

4 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 transactions are an important concept in both databases and blockchain ledgers.

In this document we will mostly talk about transactions in the database sense. As it turns out, the processing of transactions in the database sense (usually[3]) corresponds to processing of whole blocks in the blockchain.

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, 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: new values are defined based on old values. It is also exactly how we define our ledger rules as functions on the ledger state: given an old state, it yields an updated state (with the old one still available). A database perspective on the blockchain ledger would say that the ledger state itself is the database and that the transactions on that database are the ledger state transformations arising from extending the chain with additional blocks. This is the perspective we will take.

5 General approach

We wish to deviate as little as possible from the general design of the current ledger and consensus layers, particularly with respect to 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 key-value mappings will be kept primarily on disk, and all other state will be kept in memory.

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

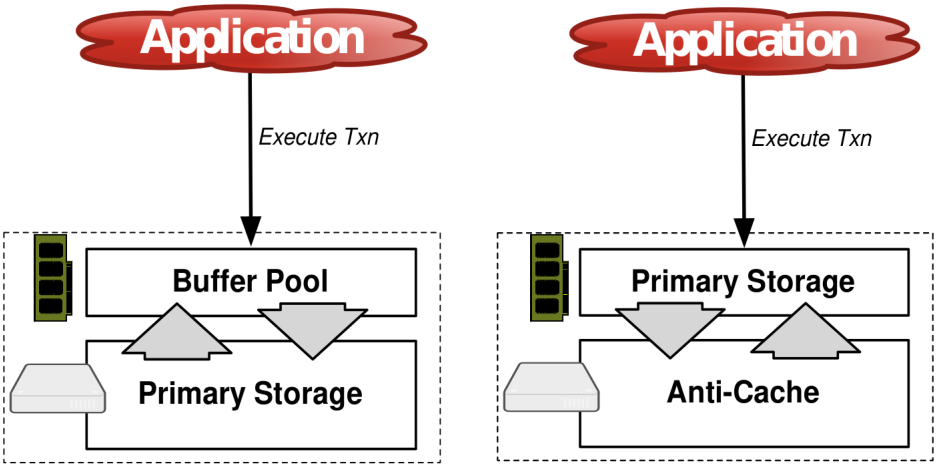
[2]For example `https://www.zdnet.com/article/security-flaws-found-in-26-low-end-cryptocurrencies/`

[3]For chain validation, the database transactions are the ledger state transformations arising from applying blocks to a chain. For mempool validation, the database transactions are the ledger state transformations arising from new blockchain transactions being added to the mempool.

5.1 Inspiration from the 'anti-caching' database architecture

Our general approach takes inspiration from DeBrabant et al. [2013] who describe an OLTP[4] database architecture that they call 'anti-caching'.

The anti-caching architecture reverses the usual notion that a database's value is represented on disk while some data is cached in memory for efficiency. Instead, the perspective is that the logical value of the database is represented in memory while much of the data is 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.



(a) Disk-oriented DBMS (b) Main Memory DBMS with Anti-Caching

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 overall memory budget.

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 on-disk anti-cache 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 into memory and retrying), but it is certainly simpler if all the required data can be identified up-front.

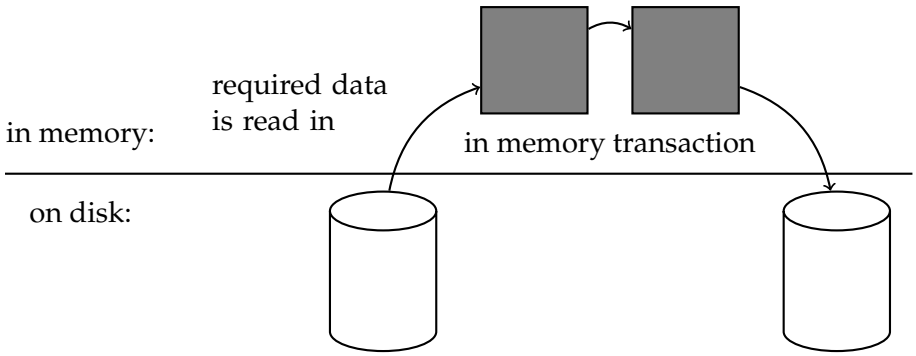
This is the other main idea that we borrow: to do all the database transaction processing in memory. This is a very attractive idea for our context because it enables the transaction processing to be implemented using pure functions operating on in-memory data structures. Unlike in the anti-caching design, we will rely on being able to identify all the data that will be needed to process a transaction *up-front* 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 that is appropriate for our use case.

[4]OLTP stands for Online Transaction Processing, which is a style of database workload. A blockchain ledger database would be subjected to such a workload.

5.2 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.

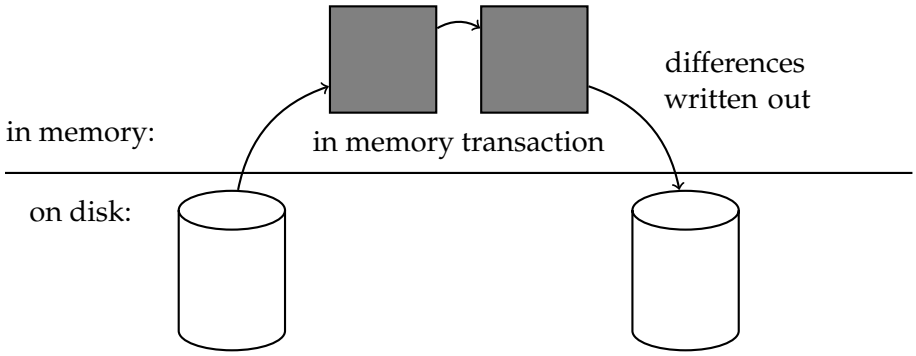


As an example of this, consider validating a ledger transaction including its UTxO inputs: we know we will need to look up the transaction inputs in the UTxO mapping. Which inputs we will need is clearly known in advance as they are explicit in the transaction itself. For the Cardano ledger rules in general, we believe that 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.

5.3 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.

5.4 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 5.1, 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 5.2, 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 simplifies the problem.
- As mentioned in Section 5.3, we need to be able to represent and manage differences of data kept on disk. Differences of key-value mappings are straightforward, so 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.

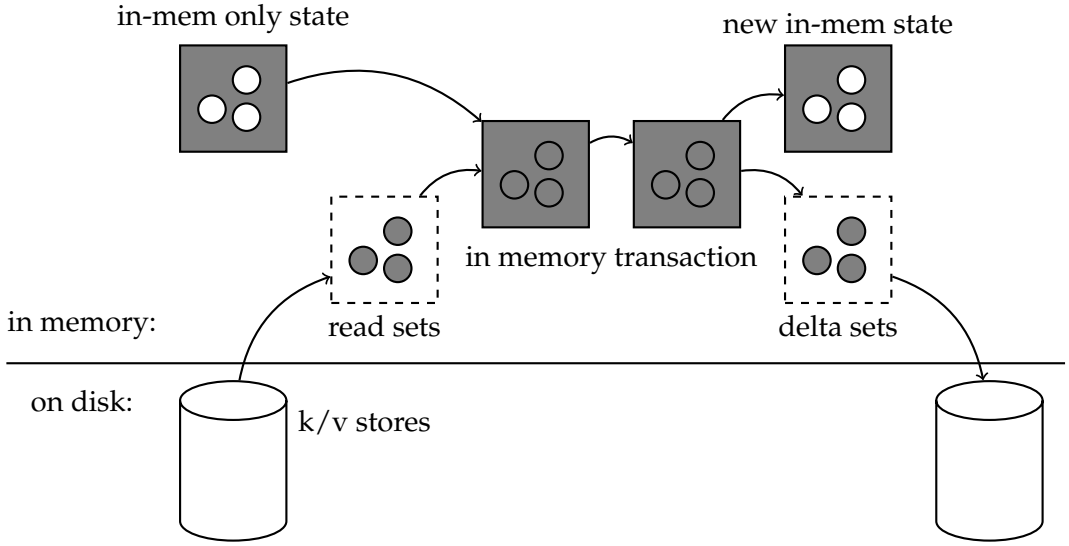
As for what counts as a large mapping, we draw the dividing line between 'large' and 'small' between those that are proportional to the number of stake addresses (or bigger) and those that are proportional to the number of stake pools (or smaller). That is, mappings with sizes proportional to the number 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

In particular, note 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 dictate 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 that 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 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 restore the state when the node 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.

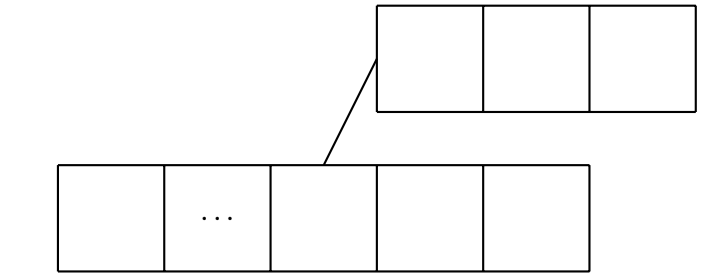
5.5 Access to multiple (logical) ledger states

As discussed in Section 3.2, 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 3.2 that the existing consensus design relies on persistent data structures so that keeping many ledger states costs little more than keeping 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 keep around 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.

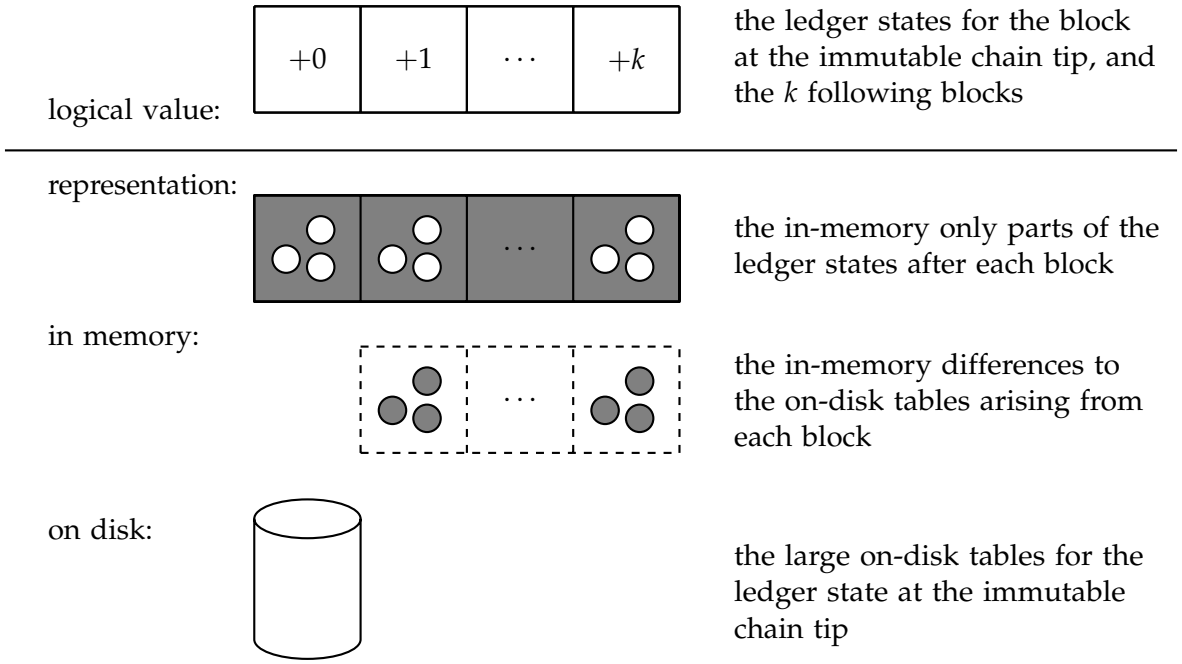


In summary, we know that the differences between all the ledger states that we need to manipulate are relatively small (compared to the size of the ledger state itself), and they are all derived from one ledger state at the 'immutable tip'. Using persistent data structures is one way to take advantage 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 correspond to the ledger state at the immutable tip, which is the common root 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:



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, we should expect the representation of the differences to require in 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.

5.6 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) could well 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 in the order of 32 – 64 concurrent I/O operations being performed at all times. A useful abstraction is to think of it as a queue of in-progress operations 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 saturate the SSD's throughput. Due to the high throughput and timing jitter, it is better to 'over-fill' the queue by some amount, e.g., $2\times$. That is, in order to ensure the SSD queue 'depth' does not drain to below 64, it may be necessary to aim to keep double that number of operations in progress at the application level. The appropriate amount to use can be tuned based on I/O profiling tools, but the overall point is clear: to fully exploit the throughput of an SSD, we need to keep a substantial number of operations in progress at once – and on a continuous basis.

Since blockchains are mostly linear in nature (being a chain), the opportunities for I/O parallelism come from batching and pipelining.

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

A block with 64 transactions with 2 UTxO inputs each would generate 128 read I/O operations. So, we can see that large enough 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.

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. 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 clear that it is more complex to arrange. The opportunity for

batching arises naturally from processing blocks as a unit. In practice, if pipelining is used, it would also be used with batching as a pipeline of batches (per block).

For this stage of the design, we simply need to ensure that the scheme for disk I/O makes it possible to take advantage of I/O pipelining. Where it is practical to take advantage of pipelining is then a design decision for the consensus layer. 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 only batching may be sufficient for an initial integration that is not yet aiming for the higher throughput targets.

6 Notation and properties of differences

To make the design ideas from the previous section more precise, we will use a more formal presentation 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. For some sets[5], we are able to find a corresponding set ΔA of *differences* or *changes* on values from the set A. We will use a naming convention $\overset{\Delta}{a} \in \Delta A$, using a Δ-accent, to remind ourselves which variables represent differences. So note that $\overset{\Delta}{a}$ is not an operator on a variable a, as it is simply a variable naming convention.

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

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

In addition to the monoid laws, we have a couple of 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.

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

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

$$\Diamond\sum_{i=0}^{n} \overset{\Delta}{a}_i = \overset{\Delta}{a}_0 \diamond \overset{\Delta}{a}_1 \diamond \overset{\Delta}{a}_2 \diamond \ldots \overset{\Delta}{a}_n$$

We will also talk about functions that transform values, and corresponding functions that compute differences. Given a function $f : A \to A$, a difference function $\overset{\Delta}{f} : A \to \Delta A$ corresponds to the original function f if it satisfies the property that applying the change gives the same result as the original function.

$$a \triangleleft \overset{\Delta}{f}(a) = f(a) \tag{3}$$

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

[5]Our data structures of interest will typically be finite mappings, and we will see that differences for finite mappings are straightforward.

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

In this section, we will look at an abstract formal treatment of databases based on differences. The abstract models we will look at are motivated by the ideas discussed in Section 5. We will discuss how different models correspond to different implementation strategies, including in-memory and on-disk data storage. Where appropriate and possible, we will sketch proofs to reassure ourselves that we will get correct results. There are a few reasons to consider these models.

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

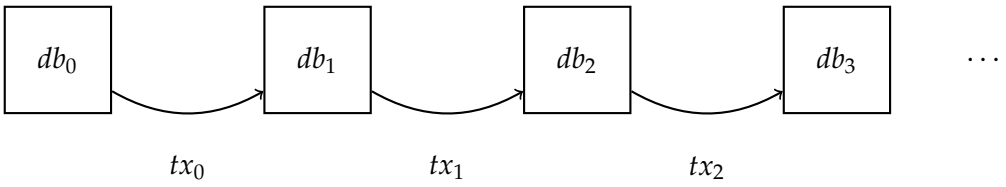
Imagining implementation strategies Another reason is that, although these are abstract models, they can be seen to correspond to certain implementation approaches. For example, we will refer to certain terms as representing on-disk or in-memory data structures. Obviously, mathematically there is no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

As mentioned in Section 4, our use of the term 'database' in this section is quite abstract: we mean only some collection of data. This section makes use of the idea and notation of the calculus of differences from the previous section.

7.1 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. Applying each transformation function to the previous state by $db_{n+1} = tx_n(db_n)$ gives rise to a series of database states $db_0, db_1, \ldots$.



$$
\begin{aligned}
db_1 &= tx_0(db_0) \\
db_2 &= tx_1(db_1) \\
&\vdots \\
&\text{And in general} \\
db_{i+1} &= tx_i(db_i) \qquad (4)
\end{aligned}
$$

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 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.

7.2 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 database values. The use of these two concepts makes this a simple but reasonable model of an on-disk database with in-memory transaction processing (as introduced in Sections 5.2 and 5.3). 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.

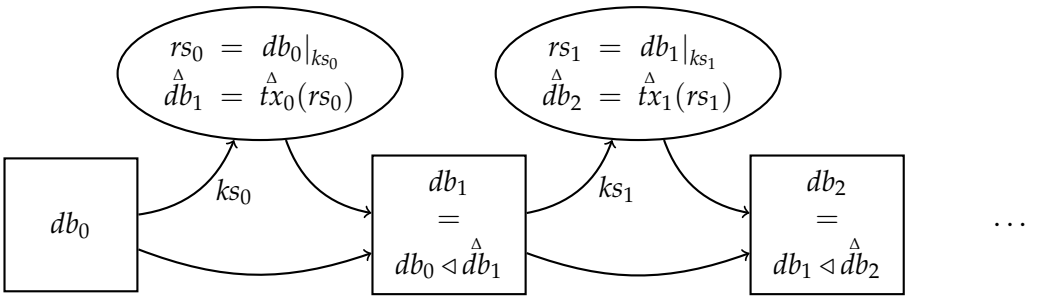
Whereas in the previous model we had a series of transaction functions $tx_0, tx_1, \ldots$, in this model we will have difference functions $\overset{\Delta}{tx}_0, \overset{\Delta}{tx}_1, \ldots : DB \to \Delta DB$. These are required to be proper difference functions, satisfying the property

$$db \triangleleft \overset{\Delta}{tx}(db) = tx(db) \tag{5}$$

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 $ks \subseteq \operatorname{dom} DB$ (or collection of sets of keys) which will be used to perform a domain restriction $db|_{ks} \subseteq DB$. So, there will key sets $ks_0, ks_1, \ldots$ corresponding to the transactions $\overset{\Delta}{tx}_0, \overset{\Delta}{tx}_1, \ldots$. 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 on the whole state

$$\overset{\Delta}{tx}(db|_{ks}) = \overset{\Delta}{tx}(db) \tag{6}$$

The domain restriction on the database value corresponds to reading a set of keys ks from the database, and we call the result a *read set*. We will define each read set as $rs_i = db_i|_{ks_i}$ and the changes from each transaction as $\overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i(rs_i)$. The series of database states $db_0, db_1, \ldots$ can now be constructed by applying the changes from each transaction to the previous database state.



$$db_1 = db_0 \triangleleft \overset{\Delta}{db}_1 \quad \textbf{where} \quad \overset{\Delta}{db}_1 = \overset{\Delta}{tx}_0(rs_0) \quad \textbf{and} \quad rs_0 = db_0|_{ks_0}$$
$$db_2 = db_1 \triangleleft \overset{\Delta}{db}_2 \quad \textbf{where} \quad \overset{\Delta}{db}_2 = \overset{\Delta}{tx}_1(rs_1) \quad \textbf{and} \quad rs_1 = db_1|_{ks_1}$$

$$\vdots$$

And in general

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \quad \textbf{where} \quad \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i(rs_i) \quad \textbf{and} \quad rs_i = db_i|_{ks_i}$$

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

$$
\begin{aligned}
& db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i(db_i|_{ks_i}) \\
\equiv \quad & \{\text{by the restriction property Equation (6) that } \overset{\Delta}{tx}_i(db|_{ks_i}) = \overset{\Delta}{tx}_i(db)\} \\
& db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i(db_i) \\
\equiv \quad & \{\text{by the difference function property Equation (5) that } db \triangleleft \overset{\Delta}{tx}(db) = tx(db)\} \\
& db_{i+1} = tx_i(db_i)
\end{aligned}
$$

Which is the same as Equation (4): the recurrence for the simple model.

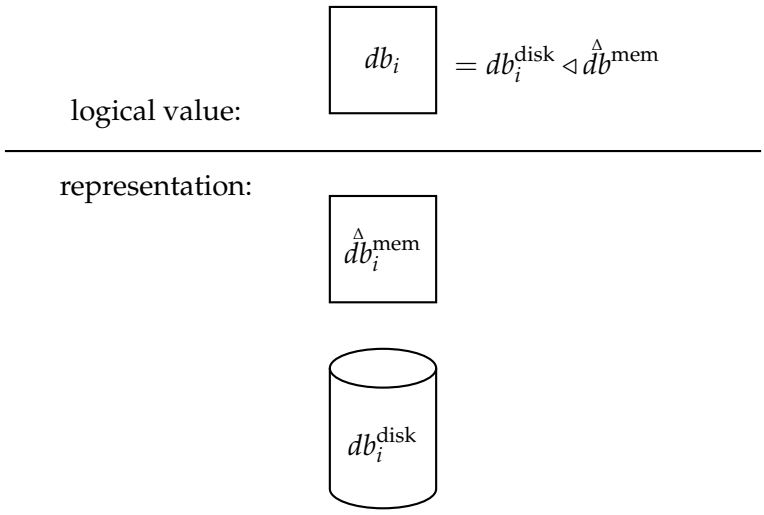
7.3 Hybrid on-disk / in-memory databases

This model covers the bare essence of the idea from Section 5.5: that is, representing the ledger state as the combination of data on disk and differences in memory. This model is just about representing a *single* ledger state. We will look at representing *multiple* states in Section 7.4.

In this model, 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 on-disk part will be ordinary database values $db_i^{\text{disk}} \in DB$, while the in-memory part will be database differences $\overset{\Delta}{db}_i^{\text{mem}} \in \Delta DB$. We define the overall logical value of the database to be the on-disk part with the in-memory changes applied 'on top'.

$$db_i = db_i^{\text{disk}} \triangleleft \overset{\Delta}{db}_i^{\text{mem}} \tag{7}$$

In a visual notation, we will depict that idea as follows. Above the line is the logical value of the database, while below the line is the corresponding representation.



There are a couple of lemmas that will prove useful later when reasoning about this representation. One is that if we have a domain restriction 'outside' of the application of changes, then it does not make any difference if we also use domain restriction 'inside' or not.

$$\left(db_a \triangleleft \overset{\Delta}{db}_b\right)\Big|_{ks} = \left(db_a|_{ks} \triangleleft \overset{\Delta}{db}_b\right)\Big|_{ks} \tag{8}$$

This lemma either needs to be proved universally or we will need to make it a required property of the apply changes operator for the choice of ΔDB.

The other useful lemma is a simple corollary of Equation (6) and Equation (8)

$$\overset{\Delta}{tx}\left(\left(db_a \triangleleft \overset{\Delta}{db}_b\right)\Big|_{ks}\right) = \overset{\Delta}{tx}\left(db_a|_{ks} \triangleleft \overset{\Delta}{db}_b\right) \tag{9}$$

This lets us 'shift' a domain restriction from outside to inside the application of changes, within the context of a 'proper' transaction function $\overset{\Delta}{tx}$ that satisfies Equation (6).

7.3.1 Performing transactions

We now need to see how performing transactions works in this representation. We will, of course, use the change-based style of transactions, as in the previous model (in Section 7.2). Given the changes made by a transaction $\overset{\Delta}{tx}_i(db_i|_{ks_i})$, we would normally obtain the new state of the database by applying the changes to the previous state

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i(db_i|_{ks_i})$$

With the hybrid representation (for db_i), that is

$$db_{i+1} = \left(db_i^{\text{disk}} \triangleleft \overset{\Delta}{db}_i^{\text{mem}}\right) \triangleleft \overset{\Delta}{tx}_i(db_i|_{ks_i})$$

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.

$$db_{i+1} = db_i^{\text{disk}} \triangleleft \left(\overset{\Delta}{db}_i^{\text{mem}} \diamond \overset{\Delta}{tx}_i(db_i|_{ks_i})\right)$$

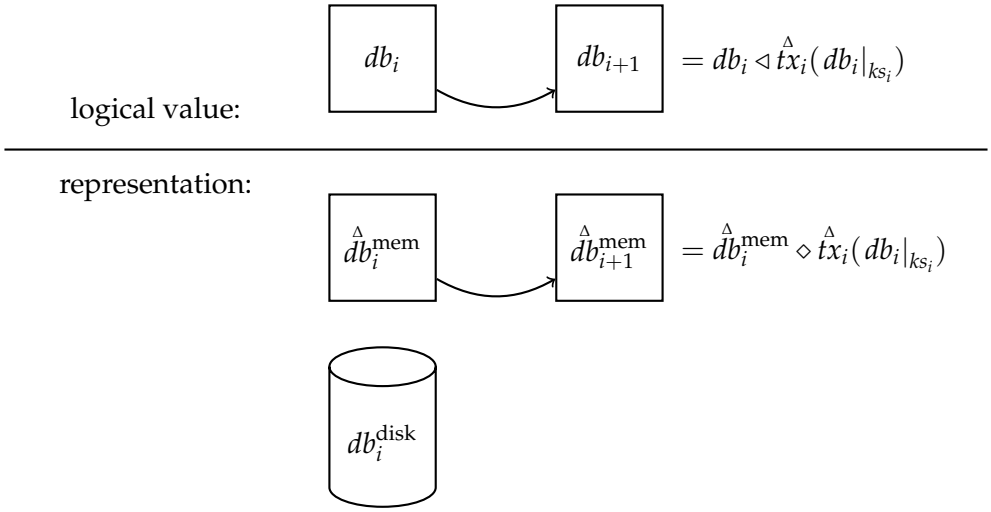
This now fits the same hybrid representation. We can define the new in-memory value to be

$$\overset{\Delta}{db}_{i+1}^{\text{mem}} = \overset{\Delta}{db}_i^{\text{mem}} \diamond \overset{\Delta}{tx}_i(db_i|_{ks_i})$$

to get

$$db_{i+1} = db_i^{\text{disk}} \triangleleft \overset{\Delta}{db}_{i+1}^{\text{mem}}$$

Or pictorially



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 size of the in-memory part will grow without bound.

7.3.2 Flushing changes to disk

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

$$db = db^{\text{disk}} \triangleleft \overset{\Delta}{db}^{\text{mem}}$$

Suppose further that the in-memory part consists of the composition of (at least) two sets of changes.

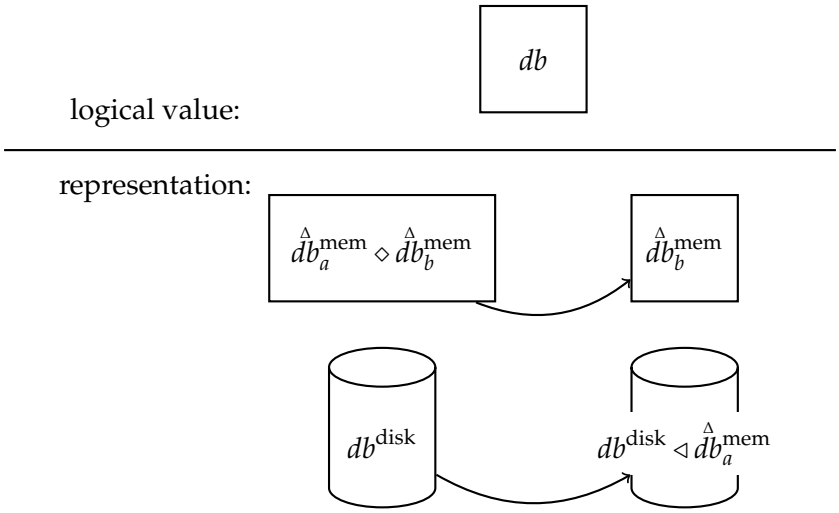
$$\overset{\Delta}{db}^{\text{mem}} = \overset{\Delta}{db}_a^{\text{mem}} \diamond \overset{\Delta}{db}_b^{\text{mem}}$$

This will be a typical situation since, as we saw above, applying each transaction gives an extra composition of changes. 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, in particular at any boundary between changes from transactions. For example, we could choose to flush everything to disk by picking $\overset{\Delta}{db}_b^{\text{mem}} = \mathbf{0}$, but we can also choose to flush just a prefix of changes.

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

$$\begin{aligned} & db = db^{\text{disk}} \triangleleft \left(\overset{\Delta}{db}_a^{\text{mem}} \diamond \overset{\Delta}{db}_b^{\text{mem}} \right) \\ \equiv & \\ & db = \left(db^{\text{disk}} \triangleleft \overset{\Delta}{db}_a^{\text{mem}} \right) \triangleleft \overset{\Delta}{db}_b^{\text{mem}} \end{aligned}$$

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



7.3.3 Performing reads

A detail we glossed over above is how we perform reads in this representation. We said above (in Section 7.3.1) that

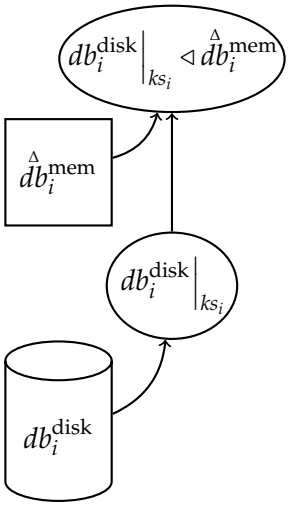
$$db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i \left(db_i|_{ks_i} \right)$$

but we glossed over how we obtain the read set $db_i|_{ks_i}$. In the context of the hybrid in-memory / on-disk database representation, we are interested in how this corresponds to operations

involving the on-disk and in-memory parts. So let us look at the read set in the context of using it with the transaction function and rewrite it into a more useful form

$$
\begin{aligned}
& \overset{\Delta}{tx}_i \left(db_i|_{ks_i} \right) \\
\equiv \quad & \{\text{by Equation (7), expanding the definition } db_i = db_i^{\text{disk}} \triangleleft \overset{\Delta}{db}_i^{\text{mem}}\} \\
& \overset{\Delta}{tx}_i \left(\left(db_i^{\text{disk}} \triangleleft \overset{\Delta}{db}_i^{\text{mem}} \right) \Big|_{ks_i} \right) \\
\equiv \quad & \{\text{by Equation (9), the corollary that } \overset{\Delta}{tx}_i \left(\left(db \triangleleft \overset{\Delta}{db} \right) \Big|_{ks} \right) = \overset{\Delta}{tx}_i \left(db|_{ks} \triangleleft \overset{\Delta}{db} \right)\} \\
& \overset{\Delta}{tx}_i \left(db_i^{\text{disk}} \Big|_{ks_i} \triangleleft \overset{\Delta}{db}_i^{\text{mem}} \right)
\end{aligned}
$$

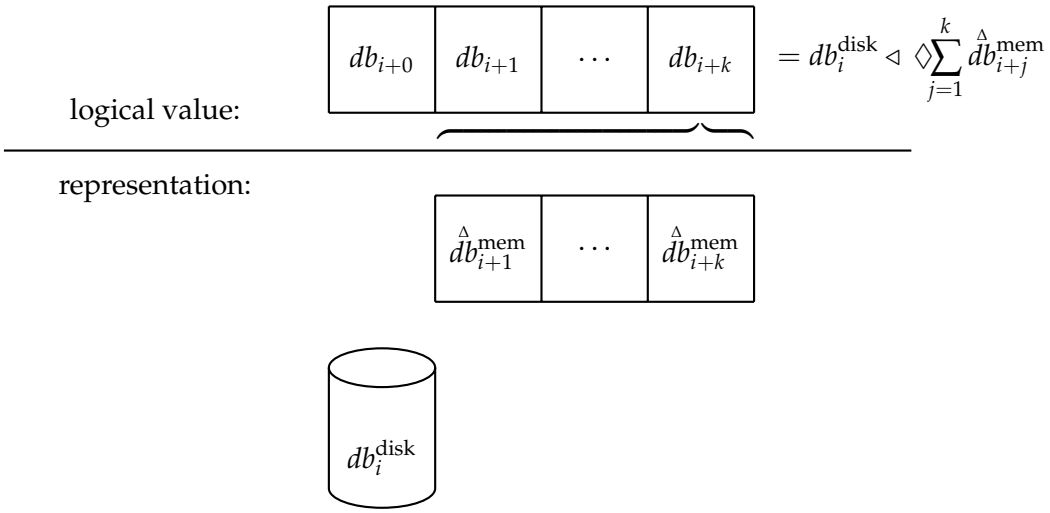
This is a very useful result. The value $db_i^{\text{disk}}\Big|_{ks_i}$ corresponds to performing a set of reads from the on-disk part of the database. Notice that we needed the context of using the read set in the transaction to apply Equation (9). The intuition is that pushing the domain restriction inside of applying the in-memory changes could increase the size of the read set, but the transaction is guaranteed not to look at anything outside the ks_i subset.



Overall, we see that we can perform reads on the hybrid representation simply by performing the reads on the on-disk part and then applying the in-memory differences to the result. This is something that can be implemented in a relatively simple and efficient way.

7.4 Multiple logical database states

This model covers the full idea from Section 5.5, in which we want to maintain and have efficient access to many logical values of a database at once. In particular, we want to model having a single on-disk state at once and a whole series of in-memory differences, giving us a series of logical database values.



As depicted 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 transaction. That is, we keep each difference and do not compose them together prematurely. Each logical database value is the value of the on-disk state with the monoidal composition of the appropriate changes applied on top. That is, for the k$^{\text{th}}$ database value beyond the on-disk state we have

$$db_{i+k} = db_i^{\text{disk}} \triangleleft \Diamond\sum_{j=0}^{k} \overset{\triangle}{db}_{i+j}^{\text{mem}}$$

which of course for the zero case is simply

$$db_{i+0} = db_i^{\text{disk}}$$

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 monoidal compositions of the remaining differences.

One interesting implementation idea to manage both a sequence of k differences and also the efficient monoidal composition of them is to use a *finger tree* data structure [Hinze and Paterson, 2006]. A finger tree is parametrised over a monoidal measure of subsequences. The choice of measure for a subsequence in this case would be the range of slot numbers and also the composition of the differences. The slot number range is included to support splitting at slot numbers. We would rely on splitting to obtain the subsequence of differences for evaluating a chain fork that starts from a recent point on the chain. Including 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, monoidal measure values. Finger trees are already used extensively within the consensus implementation.

There are a few operations we need to be able to perform in this setting:

1. Perform reads of sets of keys and '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 subsequence can start anywhere in the sequence, and the replacement subsequence can have any length. 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 on-disk store.

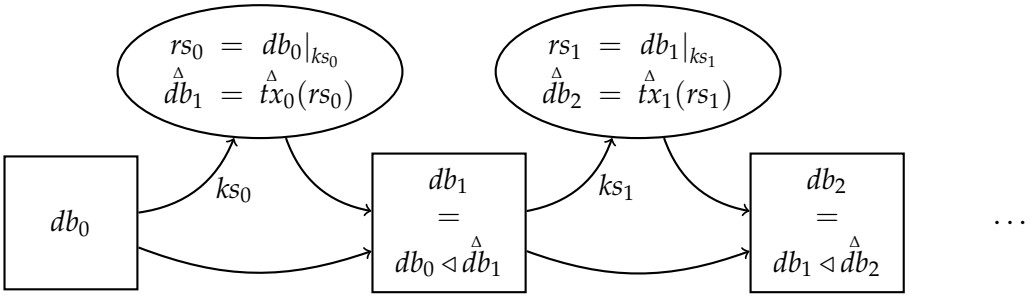
These operations are straightforward generalisations of the operations we have already seen in Section 7.3. The first two are about constructing new logical database values by making new in-memory differences. In this representation, it is achieved by simply appending or replacing a suffix of a sequence of changes. The flushing to disk is a straightforward instance of the flush operation from Section 7.3 where we choose a point in the sequence that splits the differences we wish to flush from the remaining sequence of differences that should be kept in memory. The value that we apply to the on-disk state is the monoidal composition of the sequence of differences we wish to flush.

7.5 Change-based pipelined databases

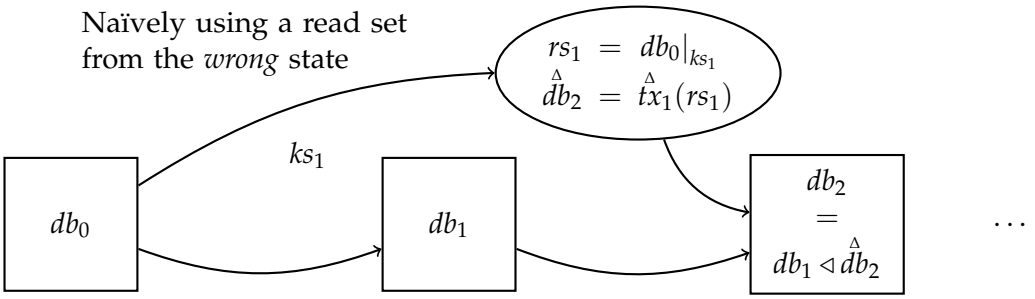
Here is where things start to get interesting and tricky. As discussed in Section 5.6, we wish to pipeline reads from disk to provide the opportunity to use parallel I/O. Providing this opportunity is not something that we can hide away in some low-level I/O layer: it will have to be explicit in how we manage the logical state of the database. So, the purpose of this model is to provide a reasonable correspondence 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 such that 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.

Recall the sequential pattern from Section 7.2 where the reads and transactions are based on the immediately preceding database value.

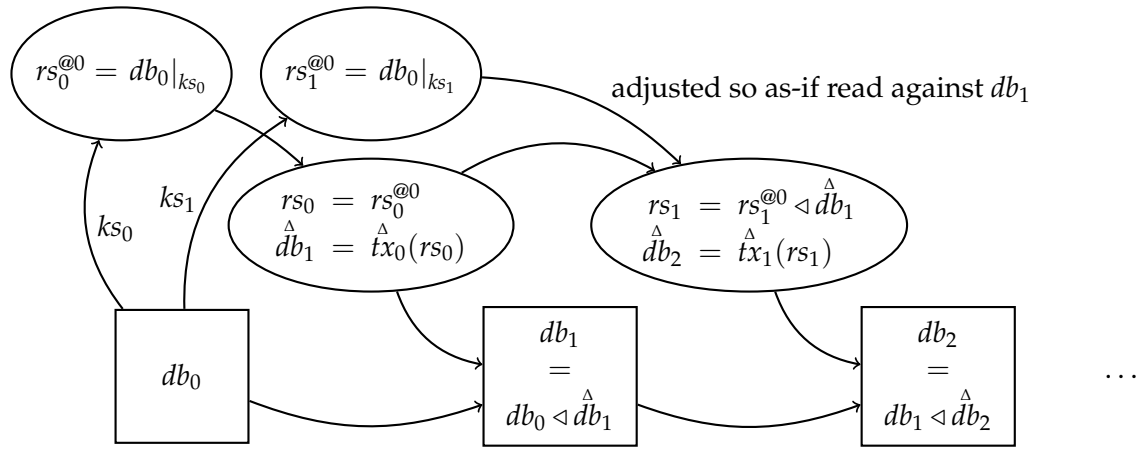


The difficulty with initiating the reads early is that the state of the database in which we initiated the reads is not the same state as the one in which we use the read results to perform a transaction. For example, if we naïvely started the reads for ks_1 from db_0, expecting to use it later with $\overset{\Delta}{tx}_1$, then any updates applied by $\overset{\Delta}{tx}_0$ in db_1 that might affect the read set would be lost and we would get the wrong result.

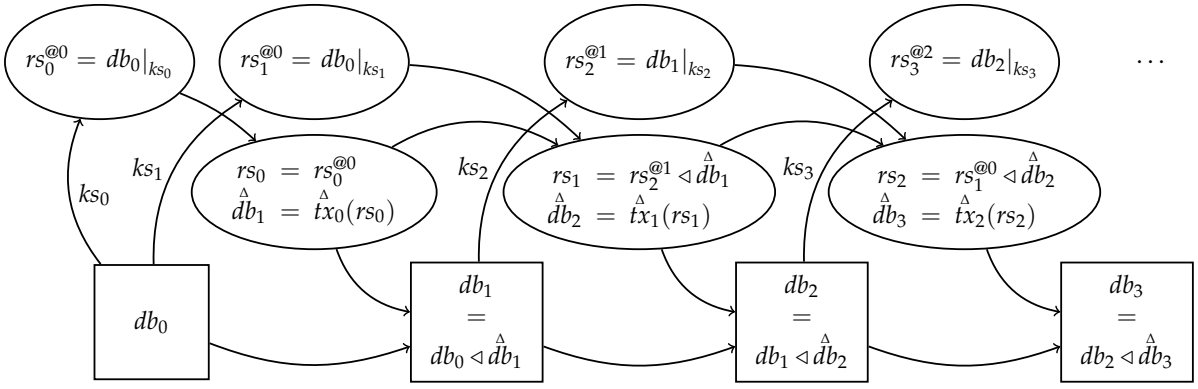


Of course, pipelining is only acceptable if we can find some way to make it semantically equivalent to the sequential version. The trick to do so is to adjust the result of the reads so that it is *as if* they had been performed against the right state of the database.

Notice in the sequential example above that db_1 is db_0 with some changes applied to it, and a read set from db_0 is simply a subset of db_0, so if we apply the same changes to the read set, then that should be the same as if we had performed the read from db_1 in the first place. In pictorial form, it would look like the following, with the reads of ks_1 intended to be used with $\overset{\Delta}{tx}_1$ being started against db_0, and then later adjusted by applying the changes from $\overset{\Delta}{tx}_0$.



We can now extend this example so that after the initial step (where we perform two reads from db_0 to get things going) we always initiate a read to be used by the next-but-one transaction. This gives us pipelining of depth one. This pattern could be extended indefinitely, and greater (or variable) depth pipelining could be used.



$$db_1 = db_0 \triangleleft \overset{\Delta}{db}_1 \quad \textbf{where} \quad \overset{\Delta}{db}_1 = \overset{\Delta}{tx}_0(rs_0) \quad \textbf{and} \quad rs_0 = rs_0^{@0} \quad \textbf{and} \quad rs_0^{@0} = db_0|_{ks_0}$$
$$db_2 = db_1 \triangleleft \overset{\Delta}{db}_2 \quad \textbf{where} \quad \overset{\Delta}{db}_2 = \overset{\Delta}{tx}_1(rs_1) \quad \textbf{and} \quad rs_1 = rs_1^{@0} \triangleleft \overset{\Delta}{db}_1 \quad \textbf{and} \quad rs_1^{@0} = db_0|_{ks_1}$$
$$db_3 = db_2 \triangleleft \overset{\Delta}{db}_3 \quad \textbf{where} \quad \overset{\Delta}{db}_3 = \overset{\Delta}{tx}_2(rs_2) \quad \textbf{and} \quad rs_2 = rs_2^{@1} \triangleleft \overset{\Delta}{db}_2 \quad \textbf{and} \quad rs_2^{@1} = db_1|_{ks_2}$$

$$\vdots$$

And in general (for $i > 1$)

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \textbf{ where } \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i(rs_i) \quad \textbf{and} \quad rs_i = rs_i^{@i-1} \triangleleft \overset{\Delta}{db}_i \quad \textbf{and} \quad rs_i^{@i-1} = db_{i-1}|_{ks_i}$$

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 $i = 0$.

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \quad \textbf{where} \quad \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i(rs_i) \quad \textbf{and} \quad rs_i = rs_i^{@i-1} \triangleleft \overset{\Delta}{db}_i \quad \textbf{and} \quad rs_i^{@i-1} = db_{i-1}|_{ks_i}$$

$\equiv$ {by substitution of rs_i and $rs_i^{@i-1}$ }

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \quad \textbf{where} \quad \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i \left(db_{i-1}|_{ks_i} \triangleleft \overset{\Delta}{db}_i \right)$$

$\equiv$ {by the domain restriction shifting lemma Equation (9) }

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \quad \textbf{where} \quad \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i \left(\left(db_{i-1} \triangleleft \overset{\Delta}{db}_i \right) \Big|_{ks_i} \right)$$

$\equiv$ {by induction hypothesis $db_i = db_{i-1} \triangleleft \overset{\Delta}{db}_i$ }

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{db}_{i+1} \quad \textbf{where} \quad \overset{\Delta}{db}_{i+1} = \overset{\Delta}{tx}_i \left(db_i|_{ks_i} \right)$$

$\equiv$ {by substitution of $\overset{\Delta}{db}_{i+1}$}

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i \left(db_i|_{ks_i} \right)$$

$\equiv$ {by the restriction property Equation (6) that $\overset{\Delta}{tx}_i \left(db|_{ks_i} \right) = \overset{\Delta}{tx}_i(db)$}

$$db_{i+1} = db_i \triangleleft \overset{\Delta}{tx}_i(db_i)$$

$\equiv$ {by the difference function property Equation (5) that $db \triangleleft \overset{\Delta}{tx}(db) = tx(db)$}

$$db_{i+1} = tx_i(db_i)$$

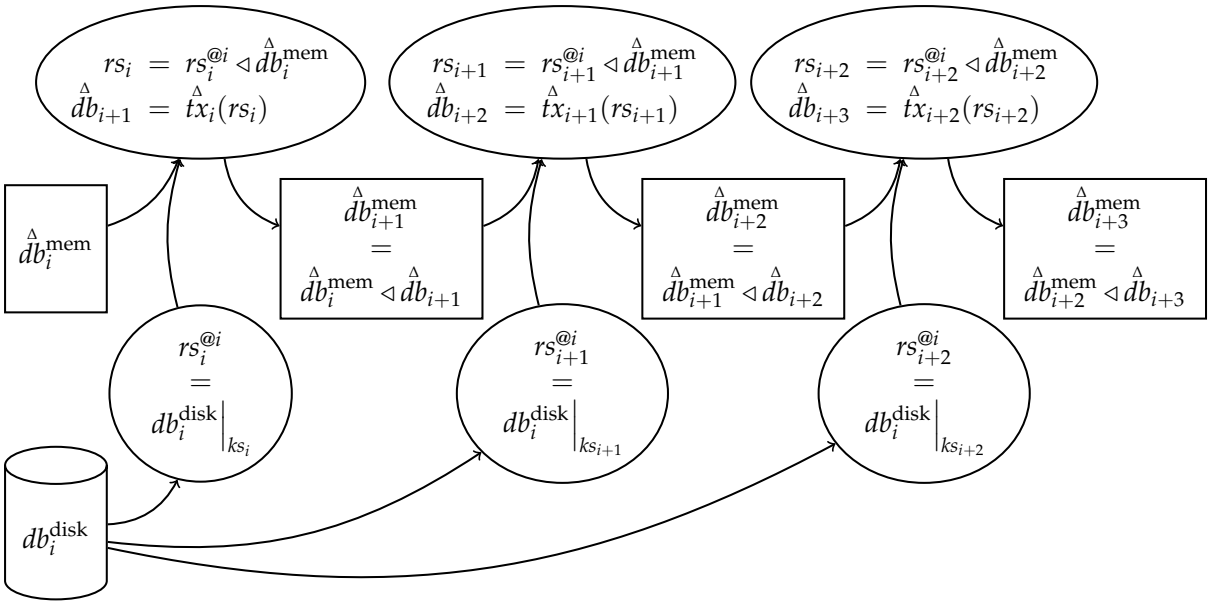
Which is the same as Equation (4): the recurrence for the simple model.

7.6 Change-based pipelined databases in the hybrid representation

More generally, the changes we want to apply are all those that occurred between the database state against which the read was performed and the state in which the transaction using the read results is to be applied. Thus, 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 an arbitrary depth of pipelining, at the memory cost of tracking the intervening changes.

Fortunately, tracking the intervening changes is relatively straightforward to do using the hybrid representations (from Sections 7.3 and 7.4) since they already keep the recent changes in memory.

The diagram below shows an example of the hybrid representation where several transactions are performed starting from the same on-disk state. Notice how all the disk reads are independent and so can be started early. Only the in-memory adjustments to read sets prior to processing transactions is still sequential.



This example does not include flushing changes to disk, which does also have to be done eventually. Doing so only marginally complicates the scheme. It involves keeping in memory the changes between when a read was initiated and when it is used – even if some older changes have been flushed to disk in the meantime.

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.