19. Sequential Transactions with Caching

There are many situations in which we want to make a ‘big’ action atomic, either with respect to concurrent execution of other actions (everyone else sees that the big action has either not started or run to completion) or with respect to failures (after a crash the big action either has not started or has run to completion).

Some examples:
- Debit/credit: \( x := x + \Delta; \ y := y - \Delta \)
- Reserve airline seat
- Rename file
- Allocate file and update free space information
- Schedule meeting: room and six participants
- Prepare report on one million accounts

Why is atomicity important? There are two main reasons:

1. **Stability**: A large persistent state is hard to fix up if it gets corrupted. This can happen because of a system failure, or because an application fails in the middle of updating the state (presumably because of a bug). Manual fixup is impractical, and ad-hoc automatic fixup is hard to code correctly. Atomicity is a valuable automatic fixup mechanism.

2. **Consistency**: We want the state to change one big action at a time, so that between changes it is always ‘consistent’, that is, it always satisfies the system’s invariant and always reflects exactly the effects of all the big actions that have been applied so far. This has several advantages:
   - When the server storing the state crashes, it’s easy for the client to recover.
   - When the client crashes, the state remains consistent.
   - Concurrent clients always see a state that satisfies the invariant of the system. It’s much easier to code the client correctly if you can count on this invariant.

The simplest way to use the atomicity of transactions is to start each transaction with no volatile state. Then there is no need for an invariant that relates the volatile state to the stable state between atomic actions. Since these invariants are the trickiest part of easy concurrency, getting rid of them is a major simplification.

**Overview**

In this handout we treating failures without concurrency; handout 20 treats concurrency without failures. A grand unification is possible and is left as an exercise, since the two constructions are more or less orthogonal.

We can classify failures into four levels. We show how to recover from the first three.

- **Transaction abort**: not really a failure, in the sense that no work is lost except by request.
- **Crash**: the volatile state is lost, but the only effect is to abort all uncommitted transactions.
- **Media failure**: the stable state is lost, but it is recovered from the permanent log, so that the effect is the same as a crash.
- **Catastrophe or disaster**: the stable state and the permanent log are both lost, leaving the system in an undefined state.

We begin by repeating the `SequentialTr` spec and the `LogRecovery` code from handout 7 on file systems, with some refinements. Then we give much more efficient code that allows for caching data; this is usually necessary for decent performance. Unfortunately, it complicates matters considerably. We give a rather abstract version of the caching code, and then sketch the concrete specialization that is in common use. Finally we discuss some pragmatic issues.

**The spec**

An \( a \) is an encoded action, that is, a transition from one state to another that also returns a result value. Note that an \( a \) is a function, that is, a deterministic transition.

```haskell
MODULE NaiveSequentialTr [
  V, % Value of an action
  S WITH { s0: ()->S } % State, s0 initially
] EXPORT Do, Commit, Crash =

TYPE A = S->(V, S) % Action

VAR ss := S.s0() % stable state
  vs := S.s0() % volatile state

APROC Do(a) -> V = << VAR v | (v, vs) := a(vs); RET v >>
APROC Commit() = << vs := ss >> % 'aborts' the transaction
APROC Crash () = << vs := ss >>

END NaiveSequentialTr
```

Here is a simple example, with variables \( X \) and \( Y \) as the stable state, and \( x \) and \( y \) the volatile state.

---

**Handout 19. Sequential Transactions with Caching**

1

---

**Handout 19. Sequential Transactions with Caching**

2
<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do(x := x - 1);</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>idle</td>
</tr>
<tr>
<td>Do(y := y + 1)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>run</td>
</tr>
<tr>
<td>Commit</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>run</td>
</tr>
<tr>
<td>Crash before commit</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>idle</td>
</tr>
</tbody>
</table>

If we want to take account of the possibility that the server (specified by this module) may fail separately from the client, then the client needs a way to detect this. Otherwise a server failure and restart, and repeats its actions, the result would be $X = 3$, $Y = 6$. To avoid these problems, we introduce a new `Begin` action to mark the start of a transaction and `Commit` marks the end. We use another state variable `ph` (for phase) that keeps track of whether there is an uncommitted transaction in progress. A transaction in progress is aborted whenever there is a crash, or if another `Begin` action is invoked before it commits. We also introduce an `Abort` action so the client can choose to abandon a transaction explicitly.

This interface is slightly redundant, since `Abort = Begin; Commit`, but it’s the standard way to do things. Note that `Crash = Abort` also; this is not redundant, since the client can’t call `Crash`.

Here is the previous example extended with the `ph` variable.

<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do(x := x - 1);</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>idle</td>
</tr>
<tr>
<td>Do(y := y + 1)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>run</td>
</tr>
<tr>
<td>Commit</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>run</td>
</tr>
<tr>
<td>Crash before commit</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>idle</td>
</tr>
</tbody>
</table>

**Uncached code**

Next we give the simple uncached code based on logging; it is basically the same as the `LogRecovery` module of handout 7 on file systems, with the addition of `ph`. Note that `ss` is not the same as the `ss` of `SequentialTr`; the abstraction function gives the relation between them.

This code may seem impractical, since it makes no provision for caching the volatile state `vs`. We will study how to do this caching in general later in the handout. Here we point out that a scheme very similar to this one is used in Lightweight Recoverable Virtual Memory, with copy-on-write used to keep track of the differences between `vs` and `ss`.

---

% ABSTRACTION to SequentialTr
SequentialTr.ss = ss + sl
SequentialTr.vs = (~ rec => vs [*] rec => ss + sl)
SequentialTr.ph = ph

% INVARIANT
- rec ==> vs = ss + sl + vl
(EXISTS l | l <= vl \ ss + sl = ss + l)
% Applying s l to ss is equivalent to applying a prefix l of vl. That is, the
% state after a crash will be the volatile state minus some updates at the end.

APROC Begin() = << vs := ss; sl := {}; vl := {}; ph := run >>
APROC Do(a) -> V RAISES {crashed} = <<
IF  ph = run => VAR v, l | (v, vs + l) = a(vs) =>
vs := vs + l; vl := vl + l; RET v [*] RAISE crashed
FI >>
PROC Commit()RAISES {crashed} =
IF ph = run => << sl := vl; ph := idle >> ; Redo() >> [*] RAISE crashed FI
PROC Crash() = << vs := ss; sl := {}; ph := idle >>;
PROC Abort() = << vs := ss; sl := {}; vl := {}; ph := idle >>
PROC Redo() = << sl := vl; ph := idle; rec := true >>; % what the crash does
PROC Redo() = << vs := ss; sl := {}; vl := {}; ph := idle >>

% apply U's in l to s
FUNC DoLog(s, l) -> S =
IF l = {} => RET s [*] RET DoLog((l.head)(s), l.tail))
FI

END LogRecovery

Here is what this code does for the previous example, assuming for simplicity that A = U. You
may wish to apply the abstraction function to the state at each point and check that each action
simulates an action of the spec.

<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>sl</th>
<th>vl</th>
<th>ph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin();</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>{}</td>
<td>{}</td>
<td>idle</td>
</tr>
<tr>
<td>Do(x := x - 1);</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>{}</td>
<td>{x=4; y=6}</td>
<td>run</td>
</tr>
<tr>
<td>Do(y := y + 1)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>{x=4; y=6}</td>
<td>{y=6}</td>
<td>idle</td>
</tr>
<tr>
<td>Commit</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>{x=4; y=6}</td>
<td>{}</td>
<td>idle</td>
</tr>
<tr>
<td>Redo: apply x := 4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>{x=4; y=6}</td>
<td>{}</td>
<td>idle</td>
</tr>
<tr>
<td>Redo: apply y := 6</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>{x=4; y=6}</td>
<td>{}</td>
<td>idle</td>
</tr>
<tr>
<td>Redo: erase sl</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>{}</td>
<td>{}</td>
<td>idle</td>
</tr>
</tbody>
</table>

Crash before commit

|      | 5 | 5 | 5 | 5 | {} | {} | idle |

Log idempotence

For this redo crash recovery to work, we need idempotence of logs: s + l + l = s + l, since
a crash can happen during recovery. From this we get (remember that "<=" on sequences is
"prefix")
11 <= 12 ==> s + 11 + 12 = s + 12,
and more generally
(ALL l1, l2 | IsHiccups(l1, l2) ==> s + 11 + 12 = s + 12)
where

FUNC IsHiccups(k: Log, l) -> Bool =
% k is a sequence of attempts to complete l
/% (EXISTS k', l' | k = k' + l' \ l' # {} \ l' <= l
\ IsHiccups(k', l) )
because we can keep absorbing the last hiccup l' into the final complete l. See handout 7 for
more detail.

We can get log idempotence if the U's commute and are idempotent, or if they are all writes like
the assignments to x and y in the example, or writes of disk blocks. Often, however, we want to
make more general updates atomic, for instance, inserting an item into a page of a B-tree. We can
make general U's log idempotent by attaching a UID to each one and recording it in S:

TYPE S = [ss, tags: SET UID]
U = [uu: SS->SS, tag: UID] WITH { meaning:=Meaning }

FUNC Meaning(u, s)->S =
IF u.tag IN s.tags => RET s [*] RET S{ ss := (u.uu)(s.ss), tags := s.tags + {u.tag} }
FI
If all the \( u \)'s in \( l \) have different tags, we get log idempotence. The way to think about this is that the modified updates have the meaning: if the tag isn’t already in the state, do the original update, otherwise don’t do it.

Most practical code for this makes each \( U \) operate on a single variable (that is, map one value to another without looking at any other part of \( S \); in the usual application, a variable is one disk block). It assigns a version number \( VN \) to each \( U \) and keeps the \( VN \) of the most recently applied \( U \) with each block. Then you can tell whether a \( U \) has already been applied just by comparing its \( VN \) with the \( VN \) of the block. For example, if the version number of the update is 23:

Original

<table>
<thead>
<tr>
<th>The disk block</th>
<th>x: Int</th>
<th>x: Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>vn: Int</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The update

\[
x := x + 1 \\
IF vn = 22 => x := x + 1; \\
vn := 23
\]

[*] SKIP FI

Note: \( vn = 22 \) implies that exactly updates 1, 2, ..., 22 have been applied.

Writing the log atomically

This code is still not practical, because it requires writing the entire log atomically in `Commit`, and the log might be bigger than the one disk block that the disk hardware writes atomically. There are various ways to get around this, but the standard one is to add a stable \( sph \) variable that can be `idle` or `commit`. We view `LogRecovery` as a spec for our new code, in which the \( sl \) of the spec is empty unless \( sph = commit \). The module below includes only the parts that are different from `LogRecovery`. It changes \( sl \) only one update at a time.

```plaintext
MODULE IncrementalLog % implements LogRecovery
...
VAR ...
sph : ENUM[Idle, commit] := idle % stable phase
vph : ENUM[Idle, run, commit] := idle % volatile phase
% ABSTRACTION to LogRecovery
LogRecovery.sl = (sph = commit => sl [*] {})
LogRecovery.ph = (sph = commit => vph [*] idle)
the identity elsewhere

APROC Begin() = << vs := ss; vl := {}; sl := {}; sph := run >>
APROC Do(a) -> V RAISES [crashed] = <<
  IF vph = run ... % the rest as before

PROC Commit() =
  IF vph = run =>
    % copy vl to sl a bit at a time
    VAR l := vl | DO l # {} => << sl := sl {l.head}; l := l.tail >> OD;
    sph := commit; vph := commit >>;
    Redo()
[*] RAISE crashed FI
PROC Crash() =
  << vs := ss; vl := {}; sph := idle >>; % what the crash does
  vph := sph; vl := (vph = idle => [] [*] sl); % the recovery
  Redo(); vs := ss % action
PROC Redo() =
  % replay vl, then clear sl
  DO vl # {} => << sl := sl {vl.head}; vl := vl.tail >> OD;
  DO sl # {} => << sl := sl.tail >> OD;
  << sph := idle; vph := idle >>
END IncrementalLog
```

And here is the example again.

<table>
<thead>
<tr>
<th>Action</th>
<th>X</th>
<th>Y</th>
<th>x</th>
<th>y</th>
<th>sl</th>
<th>vl</th>
<th>sph</th>
<th>vph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin, Do, Do</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>{}</td>
<td>{x:=4; y:=6}</td>
<td>idle</td>
<td>run</td>
</tr>
<tr>
<td>Commit</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>{x:=4; y:=6}</td>
<td>{x:=4; y:=6}</td>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td>Redo: x:=4; y:=6</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>{x:=4; y:=6}</td>
<td>{}</td>
<td>commit</td>
<td>commit</td>
</tr>
<tr>
<td>Redo: cleanup</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>6</td>
<td>{}</td>
<td>{}</td>
<td>idle</td>
<td>idle</td>
</tr>
</tbody>
</table>

We have described \( sph \) as a separate stable variable, but in practice each transaction is labeled with a unique transaction identifier, and \( sph = commit \) for a given transaction is represented by the presence of a `commit` record in the log that includes the transaction’s identifier. Conversely, \( sph = idle \) is represented by the absence of a commit record or by the presence of a later “transaction end” record in the log. The advantages of this representation are that writing \( sph \) can be batched with all the other log writes, and that no storage management is needed for the \( sph \) variables of different transactions.

Note that you still have to be careful about the order of disk writes: all the log data must really be on the disk before \( sph \) is set to `commit`. This complication, called “write-ahead log” or ‘WAL’ in the database literature, is below the level of abstraction of this discussion.
Caching

We would like to have code for `SequentialTr` that can run fast. To this end it should:

1. Allow the volatile state to be cached so that the frequently used parts of it are fast to access, but not require a complete copy of the parts that are the same in the stable state.
2. Decouple `Commit` from actually applying the updates that have been written to the stable log, because applying the updates slows down `Commit` and therefore holds up the client, and it also does a lot of random disk writes that do not make good use of the disk. By waiting to write out changes until the main memory space is needed, we have a chance of accumulating many changes to a single disk block and paying only one disk write to record them all. We may also be able to group updates to adjacent regions of the disk.
3. Decouple crash recovery from actually applying updates. This is important once we have decoupled `Commit` from applying updates, since a lot of updates can now pile up and recovery can take a long time. Also, we get it more or less for free.
4. Allow uncommitted updates to be written to the stable log, and even applied to the stable state. This saves a lot of bookkeeping to keep track of which parts of the cache go with uncommitted transactions, and it allows a transaction to make more updates than will fit in the cache.

Our new caching code has a stable state; as in `LogRecovery`, the committed state is the stable state plus the updates in the stable log. Unlike `LogRecovery`, the stable state may not include all the committed updates. `Commit` need only write the updates to the stable log, since this gets them into the abstract stable state `SequentialTr.ss`; a background thread updates the concrete stable state `LogAndCache.ss`. We keep the volatile state up to date so that `Do` can return its result quickly. The price paid in performance for this scheme is that we have to reconstruct the volatile state from the stable state and the log after a crash, rather than reading it directly from the committed stable state, which no longer exists. So there’s an incentive to limit the amount by which the background process runs behind.

Normally the volatile state consists of entries in the cache. Although the abstract code below does not make this explicit, the cache usually contains the most recent values of variables, that is, the values they have when all the updates have been done. Thus the stable state is updated simply by writing out variables from the cache. If the write operations write complete disk blocks, as is most often the case, it’s convenient for the cached variables to be disk blocks also. If the variables are smaller, you have to read a disk block before writing it; this is called an ‘installation read’. The advantage of smaller variables, of course, is that they take up less space in the cache.

The cache together with the stable state represents the volatile state. The cache is usually called a ‘buffer pool’ in the database literature, where these techniques originated.

We want to ‘install’ parts of the cache to the stable state independently of what is committed (for a processor cache, install is usually called ‘flush’, and for a file system cache it is usually called ‘sync’). Otherwise we might run out of cache space if there are transactions that don’t commit for a long time. Even if all transactions are short, a popular part of the cache might always be touched by a transaction that hasn’t yet committed, so we couldn’t install it and therefore couldn’t truncate the log. Thus the stable state may run ahead of the committed state as well as behind. This means that the stable log must include “undo” operations that can be used to reverse the uncommitted updates in case the transaction aborts instead of committing. In order to keep undoing simple when the abort is caused by a crash, we arrange things so that before applying an undo, we use the stable log to completely do the action that is being undone. Hence an undo is always applied to an “action consistent” state, and we don’t have to worry about the interaction between an undo and the smaller atomic updates that together comprise the action. To implement this rule we need to add an action’s updates and its undo to the log atomically.

To be sure that we can abort a transaction after installing some parts of the cache to the stable state, we have to follow the “write ahead log” or WAL rule, which says that before a cache entry can be installed, all the actions that affected that entry (and therefore all their undo’s) must be in the stable log.

Although we don’t want to be forced to keep the stable state up with the log, we do want to discard old log entries after they have been applied to the stable state, whether or not the transaction has committed, so the log space can be reused. Of course, log entries for undo’s can’t be discarded until `Commit`.

Finally, we want to be able to keep discarded log entries forever in a “permanent log” so that we can recover the stable state in case it is corrupted by a media failure. The permanent log is usually kept on magnetic tape.

Here is a summary of our requirements:

- Cache that can be installed independently of locking or commits.
- Crash recovery (or ‘redo’) log that can be truncated.
- Separate undo log to simplify truncating the crash recovery log.
- Complete permanent log for media recovery.

The `LogAndCache` code below is a full description of a practical transaction system, except that it doesn’t deal with concurrency (see handout 20) or with distributing the transaction among multiple `SequentialTr` modules (see handout 27). The strategy is to:

- Factor the state into independent components, for example, disk blocks.
- Factor the actions into log updates called `u`’s and cache updates called `w`’s. Each cache update not only is atomic but works on only one state component. Cache updates for different components commute. Log updates do not need either of these properties.
- Define an `undo` action for each action (not each update). The action followed by its undo leaves the state unchanged. An undo action is a full-fledged action, so it may itself involve multiple updates.
- Keep separate log and undo log, both stable and volatile.

\[ \text{Log} : \text{sequence of updates} \]
**UndoLog** : sequence of undo actions (not updates)

The essential step is installing a cache update into the stable state. This is an internal action, so it must not change the abstract stable or volatile state. As we shall see, there are many ways to satisfy this requirement.

The code in LogAndCache is rather abstract. We give the usual concrete form in BufferPool below. Here \( W \) (a cached update) is just \( s(ba) := d \) (that is, set the contents of a block of the stable state to a new value). This is classical caching, and it may be helpful to bear it in mind as concrete code for these ideas, which is worked out in detail in BufferPool. Note that this kind of caching has another important property: we can get the current value of \( s(ba) \) from the cache.

This property isn’t essential for correctness, but it certainly makes it easier for \( Do \) to be fast.

```plaintext
MODULE LogAndCache [ % implements SequentialTr
V, % Value of an action
S0 WITH {s0:=() -> S0} % abstract State; s0 initially
] = EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A = S -> (V, S) % Action
S = S WITH {"++" := DoLog, % State with ops
"-" := UndoLog, "++" := DoCache,
"-" := UndoLog}

Tag = ENUM[commit] % Update
U = S -> S % Undo
Un = (A + ENUM[cancel]) % Undo
W = S -> S % update in cache
L = SEQ (SEQ U + Tag) % Log
UL = SEQ Un % undo log
C = SET W WITH {"++" := CombineCache} % Cache
Ph = ENUM[idle, run] % Phase

VAR ss := S.s0{} % Stable State
s1 := L{} % Stable Log
sul := UL{} % Stable Undo Log
c := C := {} % Cache (dirty part)
v1 := L{} % Volatile Log
vul := UL{} % Volatile Undo Log
vph := idle % Permanent Log
pl := L{} undoing := false
```

Note that there are two logs, called \( L \) and \( UL \) (for undo log). A \( L \) records groups of updates; the difference between an update \( U \) and an action \( A \) is that an action can be an arbitrary state change, while an update must interact properly with a cache update \( W \). To achieve this, a single action must in general be broken down into several updates. All the updates from one action form one group in the log. The reason for grouping the updates of an action is that, as we have seen, we have to add all the updates of an action, together with its undo, to the stable log atomically.

There are various ways to represent a group, for example as a sequence of updates delimited by a special mark token, but we omit these details.

A cache update \( W \) must be applied atomically to the stable state. For example, if the stable state is just raw disk pages, the only atomic operation is to write a single disk page, so an update must be small enough that it changes only one page.

\( UL \) records undo actions \( Un \) that reverse the effect of actions. These are actions, not updates; a \( Un \) is converted into a suitable sequence of \( U \)'s when it is applied. When we apply an undo, we treat it like any other action, except that it doesn’t need an undo itself; this is because we only apply undo’s to abort a transaction, and we never change our minds about aborting a transaction. We do need to ensure either that each undo is applied only once, or that undo actions have log idempotence. Since we don’t require log idempotence for ordinary actions (only for updates), it’s unpleasant to require it for undo’s. Instead, we arrange to remove each undo action from the undo log atomically by applying it. We code this idea with a special \( Un \) called cancel that means: don’t apply the next earlier \( Un \) in the log that hasn’t already been canceled, and we write a cancel to \( Vul \) atomically as we write the updates of the undo action to \( Vl / Sl \).

For example, after \( un0 \), \( un1 \), and \( un2 \) have been processed, \( Vl \) might be:

\[
\text{un0 un1 un2 cancel un3 cancel cancel} = \text{un0 un1 cancel} = \text{un0}
\]

Of course many other encodings are possible, but this one is simple and fits in well with the rest of the code.

Examples of actions:

- \( f(x) := y \) % simple overwriting
- \( f(x) := f(x) + y \) % not idempotent
- \( f := f(x -> ) \) % delete

\( \text{split B-tree node} \)

This code has commit records in the log rather than a separate \( sph \) variable as in IncrementalLog. This makes it easier to have multiple transactions in the stable log. For brevity we omit the machinations with \( sph \).

We have the following requirements on \( U, Un, \) and \( W \):

- We can atomically add to \( s1 \) both all the \( U \)'s of an action and the action's undo (\text{ForceOne} does this).
- Applying a \( W \) to \( ss \) is atomic (\text{Install} does this).
- Applying a \( W \) to \( ss \) doesn't change the abstract \( ss \) or \( vs \). This is a key property that replaces the log idempotence of \text{LogRecovery}.
- A \( W \) looks only at a small part of \( s \) when it is applied, normally only one component (\text{DoCache} does this).
- Mapping \( U \) to \( W \) is cheap and requires looking only at a small part (normally only one component) of \( ss \), at least most of the time (\text{Apply} does this).
• All \( w \)'s in the cache commute. This ensures that we can install cache entries in any order (CombineCache assures this).

% ABSTRACTION \( \varnothing \)SequentialTr
SequentialTr.ss = ss + sl – sul
SequentialTr.vs = (~undoing => ss + (sl+vl)-(sul+vul))
SequentialTr.ph = (~undoing => vph idle)

% INVARIANTS

[1] (ALL l1, l2 | sl = l1 + {commit} + l2)
\( /\backslash \) – commit IN l2
\( \Rightarrow \) ss + l1 = ss + sl - sul
% Stable undos cancel
% uncommitted tail of sl

[2] ss + sl = ss + sl + vl – vul
% Volatile undos cancel vl

[3] ~undoing \( \Rightarrow \) ss + sl + vl = ss + c
% Cache is correct; this is vs

[4] (ALL w1:IN c, w2:IN c | w1 * w2 = w2 * w1).
% All W's in c commute

[5] S.s0() + pl + sl – sul = ss
% Permanent log is complete

[6] (ALL w:IN c | ss + (w) + sl = ss + sl)
% Any cache entry can be installed

% External interface

PROC Begin() = << IF vph = run => Abort() [*] SKIP FI; vph := run >>
PROC Do(a) -> V RAISES {crashed} = << IF vph = run => VAR v | v := Apply(a, AToUn(a, ss + c)); RET v [*] RAISE crashed FI >>
PROC Commit() RAISES {crashed} = << IF ~undoing => ForceAll(); sl := sl + {commit}; sul := {}; vph := idle >>
PROC Abort() = undoing := true; Undo(); vph := idle
PROC Checkpoint() = VAR c' := c, w | DO c' # {} => w := Install(); c' := c' - (w) OD;
PROC Crash() = << vl := {}; vul := {}; undoing := true >>;
PROC Redo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |
PROC Undo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |

% Internal procedures invoked from Do and Commit

APROC Apply(a, un) -> V = << VAR v, l, vs := ss + c | (v, l) := AToUn(a, vs); vl := vl + {un}; VAR c' | ss + c + c' = ss + c + l 
\( \Rightarrow \) c := c + c';
RET v >>
PROC ForceAll() = DO vl # {} => ForceOne() OD; RET
PROC ForceOne() = << VAR ll, l2 | sl := sl + {vl.head}; sul := {}; vl := vl.tail; vph := run >>
PROC Redo() = VAR l := + : sl |
PROC Undo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |
PROC Undo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |
PROC Crash() = << vl := {}; vul := {}; c := {}; undoing := true >>;

% Internal procedures invoked from Crash or Abort

PROC ForceAll() = DO vl # {} => ForceOne() OD; RET
PROC ForceOne() = << VAR ll, l2 | sl := sl + {vl.head}; sul := {}; vl := vl.tail; vph := run >>
PROC Redo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |
PROC Undo() = VAR ul := sul + vul, i := 0 | DO ul # {} => VAR un := ul.last |
PROC Crash() = << vl := {}; vul := {}; c := {}; undoing := true >>;

% BACKGROUND actions to install updates from c to ss and to truncate sl

THREAD Background() =

% Enough of these implement WAL
Install()
[ ] ForceOne()
[ ] Drop()
[ ] Truncate()
[ ] SKIP
OD

APROC Install() -> W = << VAR w :IN c |
PROC Drop() = << VAR w :IN c | ss + w = ss \( \Rightarrow \) c := c – [w] >>
PROC Truncate() = << VAR ll, l2 |
PROC Crash() = << vl := {}; vul := {}; c := {}; undoing := true >>;

% Media recovery

The idea is to reconstruct the stable state ss from the permanent log pl by redoing all the updates, starting with a fixed initial ss. Details are left as an exercise.
% Miscellaneous functions

FUNCTION AToL(a, s) -> (V, L) = VAR v, l |
  l.size = 1 /\ (v, s + l) = a(s) => RET (v, l)
END AToL

FUNCTION AToUn(a, s) -> Un = VAR un, v, s' |
  (v, s') = a(s) /\ (nil, s) = un(s') => RET un
END AToUn

The remaining functions are only used in guards and invariants.

FUNCTION DoLog(s, l) -> S = |
  IF l = {} => RET s
  (*) VAR us := l.head |
  RET DoLog((us IS Tag / us = {} => s) | (us.head)(s), (us.tail) + l.tail)
END DoLog

FUNCTION DoCache(s, c) -> S = |
  DO VAR w :IN c | s := w(s), c := c - {w} OD; RET s
END DoCache

FUNCTION UndoLog(s, ul) -> S = |
  IF ul = {} => RET s
  (*) VAR ul1, un, ul2 | un # cancel /
    ul = ul1 + {un, cancel} + ul2 => RET UndoLog(s, ul1 + ul2)
END UndoLog

A cache is a set of commuting update functions. When we combine two caches c1 and c2, as we do in apply, we want the total effect of c1 and c2, and all the updates still have to commute and be atomic updates. The CombineCache below just states this requirement, without saying how to achieve it. Usually it’s done by requiring updates that don’t commute to compose into a single update that is still atomic. In the usual case updates are writes of single variables, which do have this property, since u1 * u2 = u2 if both are writes of the same variable.

FUNCTION CombineCache(c1, c2) -> C = |
  VAR c | (*) : c1.seq) * (* : c2.seq) /\ (ALL w1 :IN c, w2 :IN c | w1 # w2 => w1 * w2 = w2 * w1) => RET c
END CombineCache

We can summarize the ideas in LogAndCache:

- Writing stable state before committing a transaction requires undo. We need to write before committing because cache space for changed state is limited, while the size of a transaction may not be limited, and also to avoid starvation that keeps us from installing some cache entries that are always part of an uncommitted transaction.

- Every uncommitted log entry has a logged undo. The entry and the undo are made stable by a single atomic action (using some low-level coding trick that we leave as an exercise for the reader). We must log an action and its undo before installing a cache entry affected by it; this is write ahead logging.

- Recovery is complete redo followed by undo of uncommitted transactions. Because of the complete redo, undo’s are always from a clean state, and hence can be actions.

- An undo is executed like a regular action, that is, logged. The undo of the undo is a special cancel action.

- Writing a & to stable state doesn’t change the abstract stable state. This means that redo recovery works. It’s a strong constraint on the relation between logged U’s and cached W’s.

### Multi-level recovery

Although in our examples an update is usually a write of one disk block, the LogAndCache code works on top of any kind of atomic update, for example a B-tree or even another transactional system. The latter case codes each as a transaction in terms of updates at a still lower level. Of course this idea can be applied recursively to yield a n level system. This is called ‘multi-level recovery’. It’s possible to make a multi-level system more efficient by merging the logs from several levels into a single sequential structure.

Why would you want to complicate things in this way instead of just using a single-level transaction, which would have the same atomicity? There are at least two reasons:

- The lower level may already exist in a form that you can’t change, without the necessary externally accessible locking or commit operations. A simple example is a file system, which typically provides atomic file renaming, on which you can build something more general. Or you might have several existing database systems, on top of which you want to build transactions that change more than one database. We show in handout 27 how to do this using two-phase commit. But if the existing systems don’t implement two-phase commit, you can still build a multi-level system on them.

- Often you can get more concurrency by allowing lower level transactions to commit and release their locks. For example, a B-tree typically holds locks on whole disk blocks to maintain the integrity of the index data structures, but at the higher level only individual entries need to be locked.

### Buffer pools

The standard code for the ideas in LogAndCache makes a U and a W read and write a single block of data. The W just gives the current value of the block, and the U maps one such block into another. Both W’s (that is, cache blocks) and U’s carry sequence numbers so that we can get the log idempotence property without restricting the kind of mapping that a U does, using the method described earlier; these are called ‘log sequence numbers’ or LSN’s in the database literature.

The LSN’s are also used to code the WAL guard in Install and the guard in Truncate. It’s OK to install a W if the LSN of the last entry in s1 is at least as big as the n of the W. It’s OK to drop a U from the front of s1 if every uninstalled W in the cache has a bigger LSN.

The simplest case is a block equal to a single disk block, for which we have an atomic write. Often a block is chosen to be several disk blocks, to reduce the size of indexes and improve the

---

\[2 \text{ D. Lomet. MLR: A recovery method for multi-level systems.} \text{ Proc. SIGMOD Conf., May, 1992, pp 185-194.} \]
efficiency of disk transfers. In this case care must be taken that the write is still atomic; many
commercial systems get this wrong.

The following module is incomplete.

```haskell
MODULE BufferPool
| % implements LogAndCache
| V, % Value of an action
| SO WITH {s0:=() -> S0}, % abstract State
|
| EXPORT ...
|

TYPE A = S->(V, S) % Action
SN = Int % Sequence Number
BA = Int % Block Address
LB = [sn, data] % Logical Block
S = BA -> LB % State
U = [sn, ba, f: LB->LB] % Update
W = [ba, lb] % update in cache
Un = A L = SEQ U UL = SEQ Un C = SET W
VAR ss := S.s0() % Stable State

FUNC vs(ba) -> LB = VAR w IN c | w.ba = ba => RET w.lb [\*] RET ss(ba)
% This is the standard abstraction function for a cache

The essential property is that updates to different blocks commute: w.ba # u.ba => w
commutes with u, because u only looks at u.ba. Stated precisely:

\( \forall s \ | \ (\forall ba | ba # u.ba ==> u(s)(ba) = s(ba) \)
/\ (\forall s' | s(u.ba) = s'(u.ba) ==> u(s)(u.ba) = u(s')(u.ba)) \)

So the guard in Install testing whether w is already installed is just

(EXISTS u | u IN vl /\ u.ba = w.a)

because in Do we get w as \#(ba:=u.ba, lb:=u(vs)(u.ba)).

END BufferPool
```

Transactions meet the real world

Various problems arise in using the transaction abstraction we have been discussing to code
actual transactions such as ATM withdrawals or airline reservations. We mention the most
important ones briefly.

The most serious problems arise when a transaction includes changes to state that is not
completely under the computer’s control. An ATM machine, for example, dispenses cash; once
this has been done, there’s no straightforward way for a program to take back the cash, or even to
be certain that the cash was actually dispensed. So neither undo nor log idempotence may be
possible. Changing the state of a disk block has neither of these problems.

So the first question is: Did it get done? The jargon for this is “testability”. Carefully engineered
systems do as much as possible to provide the computer with feedback about what happened in
the real world, whether it’s dispensing money, printing a check, or unlocking a door. This means
having sensors that are independent of actuators and have a high probability of being able to
detect whether or not an intended physical state transition occurred. It also means designing the
computer-device interface so that after a crash the computer can test whether the device received
and performed a particular action; hence the name “testability”.

The second question is: Is undo possible? The jargon for this is “compensation”. Carefully
engineered systems include methods for undoing at least the important effects of changes in the
state of the world. This might involve executing some kind of compensating transaction, for
instance, to reduce the size of the next order if too much is sent in the current one, or to issue a
stop payment order for a check that was printed erroneously. Or it might require manual
intervention, for instance, calling up the bank customer and asking what really happened in
yesterday’s ATM transaction. Usually compensation is not perfect.

Because compensation is complicated and imperfect, the first line of defense against the
problems of undo is to minimize the probability that a transaction involving real-world state
changes will have to abort. To do this, break it into two transactions. The first runs entirely inside
the computer, and it makes all the internal state changes as well as posting instructions for the
external state changes that are required. The second is as simple as possible; it just executes the
posted external changes. Often the second transaction is thought of as a message system that is
responsible for reliably delivering an action message to a physical device, and also for using the
testability features to ensure that the action is taken exactly once.

The other major difficulty in transactions that interact with the world arises only with concurrent
transactions. It has to do with input: if the transaction requires a round trip from the computer to
a user and back it might take a long time, because users are slow and easily distracted. For
example, a reservation transaction might accept a travel request, display flight availability, and
ask the user to choose a flight. If the transaction is supposed to be atomic, seats on the displayed
flights must remain available until the user makes her choice, and hence can’t be sold to other
customers. To avoid these problems, systems usually insist that a single transaction begin with
user input, end with user output, and involve no other interactions with the user. So the
reservation example would be broken into two transactions, one inquiring about flight
availability and the other attempting to reserve a seat. Handout 20 on concurrent transactions
discusses this issue in more detail.