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 treat 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.

```plaintext
MODULE NaiveSequentialTr [ V, S ]
EXPORT Do, Commit, Crash =
TYPE A = S->(V, S)
VAR ss := S.s0()
VAR vs := S.s0()
APROC Do(a) -> V = << VAR v | (v, vs) := a(vs); RET v >>
APROC Commit() = << ss := vs >>
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.

<table>
<thead>
<tr>
<th>Action</th>
<th>( X )</th>
<th>( Y )</th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do(x := x - 1);</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Do(y := y + 1)</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

| Commit | 4 | 6 | 4 | 6 |

| Crash before commit | 5 | 5 | 5 | 5 |

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 after the decrement of \( x \) in the example could result in \( X = 5, Y = 6 \), because the client will continue with the decrement of \( y \) and the commit. Alternatively, if the client fails at that point, restarts, 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 as 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 exported interface is slightly redundant, since \( \text{Abort} = \text{Begin} \); \( \text{Commit} \). 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 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\(^1\), with copy-on-write used to keep track of the differences between \( vs \) and \( ss \).

Here is the previous example extended with the \( ph \) variable.

## 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.

Here is the previous example extended with the \( ph \) variable.

### MODULE SequentialTr

```plaintext
MODULE SequentialTr [
    \( \text{V}, \text{S} \) WITH \{ \( s0: () \rightarrow \text{S} \) \} \]
EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A = \( \text{S} \rightarrow (\text{V}, \text{S}) \)
VAR ss := S.s0()
VAR vs := S.s0()
VAR ph := \text{ENUM}[\text{idle, run}] := \text{idle}

EXCEPTION crashed

APROC Begin() = << Abort(); ph := run >>
APROC Do(a) -> \text{V} RAISES [crashed] = <<
    IF ph = run => VAR v | (v, vs) := a(vs); RET v [*] RAISE crashed FI >>
APROC Commit() RAISES [crashed] = <<
    IF ph = run => ss := vs; ph := idle [*] RAISE crashed FI >>
APROC Abort() = << vs := ss; ph := idle >>
APROC Crash() = << vs := ss; ph := idle >>

END SequentialTr
```

### 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\(^1\), with copy-on-write used to keep track of the differences between \( vs \) and \( ss \).

### MODULE LogRecovery

```plaintext
MODULE LogRecovery [
    \( \text{V}, \text{S} \) WITH \{ \( s0: () \rightarrow \text{S} \) \} \]
EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A = \( \text{S} \rightarrow (\text{V}, \text{S}) \)
U = \( \text{S} \rightarrow \text{S} \)
L = SEQ U
S = \( \text{S} \) WITH \{ \( \text{"+"} \rightarrow \text{DoLog} \) \}
Ph = \text{ENUM}[\text{idle, run}]

VAR ss := S.s0()
VAR vs := S.s0()
VAR sl := L{}
VAR vl := L{}
VAR ph := idle
VAR rec := false

EXCEPTION crashed

% 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 sl 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) -> \text{V} RAISES [crashed] = <<
    IF ph = run => VAR v | (v, vs + l) = a(vs) =>
    vs := vs + l; vl := l + 1; RET v [*] RAISE crashed FI >>
APROC Commit() RAISES [crashed] =
    IF ph = run => << sl := vl; ph := idle >> ; Redo() >> [*] RAISE crashed FI
APROC Abort() = << vs := ss + sl; vl := []; ph := idle >>

```
PROC Crash() = << vs := ss; vl := {}; ph := idle; rec := true >>; % what the crash does
vl := sl; Redo(); vs := ss; rec := false % the recovery action
PROC Redo() = % replay vl, then clear sl % sl := vl before this is called
DO vl # {} => << vs := ss + {vl.head} >>; << vl := vl.tail >> OD
<< sl := {} >>
FUNC DoLog(s, l) -> S = % apply U's in l to s
IF  l = {} => RET s % apply U's in l to s
FI
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:

<table>
<thead>
<tr>
<th>Original</th>
<th>Idempotent</th>
</tr>
</thead>
</table>
| The disk block | x: Int x: Int
| The update | vn = 22 => x := x + 1; vn := 23
| 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. The action in the code that corresponds to Commit in the spec is setting sph = commit.

MODULE IncrementalLog % implements LogRecovery ...
VAR ...

% ABSTRACTION to LogRecovery
LogRecovery.sl = (sph # commit => sl [*] {})
LogRecovery.ph = (sph # commit => vph [*] idle)
the identity elsewhere

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

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

1. Allow the volatile state vs 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: this is called installing the updates. Installing 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 install them all. We may also be able to group updates to adjacent regions of the disk.

3. Decouple crash recovery from installing updates. This is important once we have decoupled Commit from install, 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 installed but 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 installed, 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.

```
MODULE LogAndCache

V, S0 WITH {s0:=()->S0}
] = EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A = S->(V, S) % Action
S = S0 WITH ("+":DoLog, "++": DoCache, "-" := UndoLog)
Tag = ENUM[commit]
U = S -> S % Update
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
sl := L{} % Stable Log
sul := UL{} % Stable Undo Log
c := C := () % Cache (dirty part)
vil := L{} % Volatile Log
vul := UL{} % Volatile Undo Log
vph := idle % Volatile PHase
pl := idle % Permanent Log
undoing := false % true during recovery; just for the abstraction function
```

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 \( s \) (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.
records undo actions \( \text{Un} \) that reverse the effect of actions. These are actions, not updates; a \( \text{Un} \) 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 with applying it. We code this idea with a special \( \text{Un} \) called cancel that means: don’t apply the next earlier \( \text{Un} \) in the log that hasn’t already been canceled, and we write a cancel to \( \text{vl}/\text{vs} \) atomically as we write the updates of the undo action to \( \text{vl}/\text{vs} \). For example, after \( \text{un}_1, \text{un}_2, \text{un}_3 \) have been processed, \( \text{ul} \) might be

\[
\begin{align*}
\text{un}_0 & \text{ un}_1 \text{ un}_2 \text{ cancel un}_3 \text{ cancel cancel} \\
= \text{ un}_0 & \text{ un}_1 \text{ un}_2 \text{ cancel cancel} \\
= \text{ un}_0 & \text{ un}_1 \text{ cancel cancel} \\
= & \text{ un}_0 \\
\end{align*}
\]

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:

\[
\begin{align*}
f(x) &= y \quad \% \text{ simple overwriting} \\
f(x) &= f(x) + y \quad \% \text{ not idempotent} \\
f &= f(x) \Rightarrow \% \text{ delete} \\
\text{split B-tree node} \\
\end{align*}
\]

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

Here is a summary of the requirements we have derived on \( \text{Un}, \text{Un} \), and \( \text{W} \):

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

% External interface

\[
\begin{align*}
\text{PROC Begin()} &= \langle \text{IF vph = run => Abort()} \rangle \text{[\*] SKIP FI; vph := run} >> \\
\text{PROC Do(a) -> V RAISES (crashed)} &= \langle \text{IF vph = run => VAR v \mid v := Apply(a, AToUn(a, ss ++ c)); RET v [\*]} \rangle \text{RAISE crashed FI} >> \\
\text{PROC Commit() RAISES (crashed)} &= \langle \text{IF vph = run => ForceAll(); \langle ss := ss + {commit}; sul := {}; vph := idle [\*]\rangle RAISE crashed FI} >>
\end{align*}
\]

% Internal procedures invoked from Do and Commit

\[
\begin{align*}
\text{APROC Apply(a, un) -> V} &= \langle \text{VAR v, l, vs := ss ++ c | (v, l) := AToL(a, vs); vl := vl + l; vul := vul + {un}; VAR c' | ss ++ c ++ c' := ss ++ c + l \\ & \Rightarrow c := c ++ c'; \text{RET v} \rangle >> \\
\text{APROC ForceAll()} &= \langle \text{DO vl \# {} \Rightarrow ForceOne() OD; RET} \rangle \text{[\*] more all of vl to sl} >>
\end{align*}
\]

% Internal procedures invoked from Crash or Abort

\[
\begin{align*}
\text{PROC Redo() = VAR l := + : sl | \langle \text{VAR l := + : sl | DO \langle l \# {} \Rightarrow VAR vs := ss ++ c, w | vs ++ [w] = vs + L([l.head]) \Rightarrow c := c ++ [w]; l := l.tail \rangle OD} \rangle \text{[\*] Restore c from sl after crash} \\
\text{PROC Undo()} &= \langle \text{VAR ul := sul + vul; i := 0} \rangle \text{[\*] Apply sul + vul to vs} >>
\end{align*}
\]
A cache is a set of commuting update functions. When we combine two caches \( c_1 \) and \( c_2 \), as we do in \text{CombineCache} below, we want the total effect of \( c_1 \) and \( c_2 \), and all the updates still have to commute and be atomic updates. The \text{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 \( u_1 \cdot u_2 = u_2 \) if both are writes of the same variable.

\[
\text{CombineCache}(c_1, c_2) \rightarrow C = \begin{cases} c_1 + c_2 & \text{if both are writes of the same variable.} \\
(\forall w : \text{IN } c \mid (w) = c_{1 \cdot w} \cdot c_{2 \cdot w}) & \text{else.}\end{cases}
\]

We can summarize the ideas in \text{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 which 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 \( w \) 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.

\text{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 update as a transaction in terms of updates at a still lower level. Of course this idea can be applied recursively to yield \( n \) 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, and it may lack 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

\[\text{DoLog}(s, l) \rightarrow S = \begin{cases} s + l & \text{DoLog}(s, l) \\
1 = (\{} \Rightarrow \text{RET } s \\
1 \cdot \text{IF } us \text{ IS Tag} \backslash \text{us} = \{\} \Rightarrow s \text{ } \text{RET } \text{DoLog}(\{ \text{us}.head\}(s), \{\text{us.tail} + l\} \text{tail})) \\
\text{FI}
\end{cases}
\]

\[\text{DoCache}(s, c) \rightarrow S = \begin{cases} s \cdot \text{DO } w : \text{IN } c \mid s = w(s), c := c - \{w\} \text{ } \text{OD; RET } s \\
\text{FI}
\end{cases}
\]

\[\text{UndoLog}(s, u1) \rightarrow S = \begin{cases} s \cdot \text{IF } ul = (\{} \Rightarrow \text{RET } s \\
1 \cdot \text{IF } ul \cdot \text{cancel} \Rightarrow \text{RET } \text{UndoLog}(u1.last)(s), ul.reml) \\
\text{FI}
\end{cases}
\]

\[\text{CombineCache}(c1, c2) \rightarrow C = \begin{cases} c1 + c2 & \text{if both are writes of the same variable.} \\
\text{FI}
\end{cases}
\]

\[\text{c1++c2 = CombineCache(c1,c2)}
\]

\[\text{Var c | (* : c.seq) = (* : c1.seq) \cdot (* : c2.seq)}
\]

\[\text{\\ (ALL w1 : IN c, w2 : IN c | w1 \# w2 \Rightarrow w1 \cdot w2 = w2 \cdot w1)} \Rightarrow \text{RET c}
\]

\[\text{END LogAndCache}
\]

\[\text{Handout 19. Sequential Transactions with Caching}
\]

\]
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 $\text{Install}$ and the guard in $\text{Truncate}$. It’s OK to install a $W$ if the LSN of the last entry in $sl$ is at least as big as the $n$ of the $W$. It’s OK to drop a $U$ from the front of $sl$ 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 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.

**MODULE BufferPool**

\[
\begin{align*}
\text{TYPE} & \quad A = S \rightarrow (V, S) \\
& \quad S = BA \rightarrow LB \\
& \quad U = [sn, ba, f: LB \rightarrow LB] \\
& \quad W = [ba, lb] \\
\text{VAR} & \quad ss := S.s0() \\
\text{FUNC} & \quad vs(ba) \rightarrow LB = \text{VAR w IN c | w.ba = ba \Rightarrow RET w.lb [\ast] RET ss(ba)} \\
& \quad \text{This is the standard abstraction function for a cache}
\end{align*}
\]

The essential property is that updates to different blocks commute: $w.ba \neq u.ba \Rightarrow w$ commutes with $u$, because $u$ only looks at $u.ba$. Stated precisely:

\[
\begin{align*}
(\forall s \quad (\forall ba \quad ba \neq u.ba \Rightarrow u(s)(ba) = s(ba)) \\
\quad \land \ (\forall s' \quad s(u.ba) = s'(u.ba) \Rightarrow u(s)(u.ba) = u(s')(u.ba)))
\end{align*}
\]

So the guard in $\text{Install}$ testing whether $w$ is already installed is just

\[
(\exists u \quad u \in vl \land u.ba = w.a)
\]

because in $\text{Do we get w}$ as $w[ba := u.ba, lb := u(vs)(u.ba)]$.

**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.

Transactions as fairy dust

Atomicity in spite of faults is only one aspect of transactions. Atomicity in spite of concurrency is another aspect; it is the subject of the next handout. A third aspect is load balancing: when many transactions run against shared state stored in a database, there is a lot of freedom in allocating CPU, memory and communications resources to them.

A complete transaction processing system puts all three aspects together, and the result is something that is unique in computing: you can start with a collection of straightforward sequential programs that are written without any concern for fault-tolerance, concurrency, or scheduling, sprinkle transaction processing fairy dust on them, and automatically obtain a fault-tolerant, concurrent program that runs efficiently on a large collection of processors, memory, and disks. Nowhere else do we know how to spin straw into gold in this way.