Lecture 16 — Atomicity via Logging **4/7/2014**

Last time: General model of transactions

Two key properties:

- Atomicity ("All or nothing", even w/ failures)
  (Show example) — abort means undo; uncommitted should be undoe
- Isolation (Concurrent operations can't see each other's writes until commit — intermediate state is never exposed.)

Saw a simple scheme to preserve atomicity via shadow copies

This time: still focusing on atomicity, but going to look at a more general scheme;

- doesn't require all writes to go to same file
- doesn't rewrite whole file on each commit

Key idea: keep a log of updates a transaction made while it ran, and whether it committed or aborted

Start with a simple scheme that is slow, and then enhance it to make it run faster.

Remember for now we are considering just one transaction running at a time (no concurrency, so no need to worry about isolation.)

So, for this lecture, our primary concern is that if the system crashes mid-transaction, we don't want the effects of those transactions to be visible after the system comes back online ("recovers").

To see how logging works — consider our bank account example again.

Two accounts: A and B.

These are the persistent values that we need to preserve atomicity for. May be helpful to think of them as a database of values, arranged in a table, e.g.:

Accounts

Id	Balance
A	100
B	50

This tabular representation is how most database work.

Accounts start out empty.
Run these all-or-nothing actions:

```
begin  //T1
A = 100
B = 50
commit

begin //T2
A = A - 20                    // note the we just log the new value, not arithmetic
B = B + 20
commit

begin  //T3
A = A + 30
--CRASH--
```

What goes into the log?

We assign every all-or-nothing action a unique transaction ID.
Need to distinguish multiple actions in progress at the same time.
Two kinds of records in the log:
UPDATE records: both new and old value of some variable.
(we'll see in a bit why we need the old values..)
COMMIT/ABORT records: specify whether that action committed or aborted.

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

Ok, so how can we use this log to implement atomic persistent storage?

Need to consider what we do when app begin / ends a transaction, or reads / writes a record

begin: allocate a new transaction ID.
write variable: append an entry to the log.
read variable: scan the log looking for last committed value.
[slide: read with a log]
As an aside: how to see your own updates?
Read uncommitted values from your own tid.
[show updated slide]

commit: write a commit record.
Expectedly, writing a commit record is the "commit point" for action, because of the way read works (looks for commit record). However, writing log records better be all-or-nothing. One approach, from last time: make each record fit within one sector.

Alternative — put some kind of checksum on each record; since we're only worried about half-written records when there's a crash, partially written records are ok to skip over because their effects won't be included in any committed transaction anyway.

abort: do nothing (could write an abort record, but not strictly needed).
recover from a crash: do nothing.

Quick demo:

```
rm DB LOG
cat l16-demo.txt
./wal-sys.py < l16-demo.txt
cat LOG
```

How can we optimize read performance?

Keep both a log and "cell storage".

Log is just as before: authoritative, provides all-or-nothing atomicity.
Cell storage: provides fast reads, but cannot provide all-or-nothing.
[board: log, cell storage; updates going to both, read from cell storage]

We will say we "log" an update when it's written to the log.
We will say we "install" an update when it's written to cell storage.

[slide: read/write with cell storage]

Two questions we have to answer now:

- how to recover cell storage from the authoritative log after crash?
- why did we implement write in the way that we did (e.g., log before cell write)?

What if we crash in setting w/ cell storage?

Let's look at the above example in our situation.

Log still contains the same things.
As we're running, maintain cell storage for A and B.
Except one problem: after crash, A's value in cell storage is wrong.
Last action aborted (due to crash), but its changes to A are visible.
We're going to have to repair this in our recovery function.
Good thing we have the log to provide authoritative information.

cell storage after crash:
A: ~~100~~ ~~80~~ 110 <— wrong!
B: ~~50~~ 70

Recovering cell storage.

What happens if we log an update, install it, but then crash/abort?
Need to undo that installed update.

Plan: scan log, determine what actions aborted ("losers"), undo them.

[slide: recover cell storage from log]

Why do we have to scan backwards?
Need to undo newest to oldest.
Also need to find outcome of action before we decide whether to undo.
In our example: done will be {1, 2}, we will set cellStorage[A] to 80.

Quick demo:

```
rm DB LOG
cat l16-demo.txt
./wal-sys.py -undo < l16-demo.txt
cat LOG
cat DB
./wal-sys.py -undo
  show_state
```

Ordering of logging and installing.

Why did we order log write before cell storage write?
Because the two together don't have all-or-nothing atomicity.
Can crash in between, so just one of the two might have taken place.

What happens if we install first and then log?

If we crash, no idea what happened to cell storage, or how to fix it. Log can't help us because log hasn't been written yet.

Golden rule of logging: "Write-ahead-log protocol" (WAL).

==> Log the update before installing it. <==

If we crash, log is authoritative and intact, can look at cell storage and make sure it is consistent w/ committed xactions.

What's the performance going to be like now?

Writes might still be OK (but we do write twice: log & install).
Reads are fast: just look up in cell storage.
Recovery requires scanning the entire log

Remaining performance problems:

We have to write to disk twice.
Scanning the log will take longer and longer, as the log grows.

Optimization 1: defer installing updates, by storing them in a cache.
[slide: read/write with a cache]
writes can now be fast: usually just one write, instead of two.
the hope is that variable is modified several times in cache before flush.
cache can be flushed sequentially

reads go through the cache, since cache may contain more up-to-date values.
(show picture on board)

atomicity problem: cell storage (on disk) may be out-of-date.
is it possible to have changes that should be in cell storage, but aren't?
yes: might not have flushed the latest commits.
is it possible to have changes that shouldn't be in cell storage, but are?
yes: flushed some changes that then aborted

during recovery, need to go through and re-apply changes to cell storage.
undo every abort / uncommitted transaction
redo every commit.

[slide: recovery for caching]

Quick demo:

```
rm DB LOG
cat l16-demo.txt
./wal-sys.py < l16-demo.txt
cat LOG
cat DB
./wal-sys.py
show_state
```

Back to the log records: why do we need all of those parts?
ID: might need to distinguish between multiple actions at the same time.
Undo: roll back losers, in case we wrote to cell storage before abort/crash.
Redo: apply commits, in case we didn't write to cell storage before commit.

Optimization 2: truncate the log.
Current design requires log to grow without bound: not practical.
What part of the log can be discarded?
We must know the outcome of every action in that part of log.
Cell storage must reflect all of those log records (commits, aborts).

Truncating mechanism:
Wait for pending actions to stop (“quiesce”)
Flush all cached updates to cell storage.
Write a checkpoint record, to save our place in the log.
Truncate log prior to checkpoint record.

Couple of details:

What if the programmer decides to abort explicitly, without crashing?
Use the log to find all update records that were part of this action.
Reset cell storage to old values from those records.
Make sure writes during abort are WAL writes — would be in trouble if we crashed mid-abort
[show code]

Do we need to write an abort record, or can we skip & pretend we crashed?

Because we undo all actions not in done (included those that aborted), and redo those in done, ok to skip.

So no need to write abort record. But writing it allows us to skip undo'ing aborted transactions

What to do about external actions, e.g., dispensed cash, or fire a missile, or shipped a package? Can't undo those things (usually). And aren't idempotent, so don't want to redo them!

Require special care. One approach: wait for software that controls action to commit, then take the action afterwards, but have some special way to detect if action already happened (e.g., sensor of amount of cash, or whether missile bay is empty / full.)

Summary.
Logging is a general technique for achieving all-or-nothing atomicity.
Widely used: databases, file systems, ..
Can achieve reasonable performance with logging.
Writes are always fast: sequential.
Reads can be fast with cell storage.
Key idea 1: write-ahead logging, makes it safe to update cell storage.
Key idea 2: recovery protocol, undo losers / redo winners.

Next time in recitation will look at more details about how this works in database systems in practice; in particular:

- Optimized checkpoints
- Optimized log record format
- What to do if system crashes while recovering

On Wednesday, we will finally talk about how to achieve isolation (this is partly covered in reading for Tuesday as well.)