Notes on Crashes & Recovery
15-440, Fall 2012
Carnegie Mellon University
Randal E. Bryant

Reading: Wikipedia entries:
Write-ahead_logging
Shadow_paging
Algorithms_for_Recovery_and_Isolation_Exploiting_Semantics

Goal: Make transactions reliable in the presence of failures.

Standard failure model:

* Machines are prone to crashes:
  Disk contents OK (nonvolatile)
  Memory contents lost (volatile)
  "Fail recoverable"
  Machines don’t misbehave

* Networks are flaky
  Drop messages, but this can be handled with timeout & retransmit
  Corruption detected by checksums

If only store state of database in memory, then a crash will lead to a loss of durability: once a transaction has been committed, it’s effects must be permanent. Can also violate "failure atomicity" even when system crashes, must be able to recover in a way that all uncommitted transactions are either aborted or committed.

General scheme: store enough information on disk that can safely determine global state.

Challenges:

1. Disk performance is poor, and modern systems have very high transaction speed requirements. Although one implementation would be to always save results of transaction to disk, this would have terrible performance.

2. Getting information onto disk in a way that can handle arbitrary crash is very tricky. (It’s even tricky to get all data onto a disk. Most disk & SS drives have write buffers that use volatile memory.)

General scheme: Make sure enough information is stored to disk so that can recover to valid state after crash. Two major schemes to do this:

1. Shadow pages. When start writing to page, make a copy (called the "shadow"). All updates on this one. Retain original copy.
- Abort: Discard shadow page
- Commit: Make the shadow page become the real page. Must then update any pointers to data on this page from other pages. Requires recursive updating

2. Write-Ahead Logging (WAL): Create log file recording every operation performed on database. Consider update to be reliable when log entry stored on disk. Keep updated versions of pages in memory (page cache). When have crash, can recover by replaying log entries to reconstruct correct state.

WAL is more common. Fewer disk operations. Can consider transaction to be committed once logfile entry stored on disk. Don’t need to write out tail of logfile until encounter commit operation.

WAL details. Example based on ARIES, WAL used in IBM & Microsoft databases. (Mohan, 1992). Considered the gold standard.

View log file as sequence of entries. Each numbered sequentially with log sequence number (LSN). (Typically LSN is the address of the first byte in the entry.) Database organized as set of fixed size PAGES, each of which may have multiple DB records. Manage storage at page level.

At any time, have copies of pages on disk, and some subset of them in memory. Memory pages in memory that differ from those on disk are termed "dirty". Termed a "page cache"; analogous to physical vs. virtual memory. Set of pages stored on disk need not represent a globally consistent state. But, it should be possible to reconstruct a globally consistent state from a combination of the log files + what is stored on disk, or log files + disk contents + page cache.

Log file consists of a sequence of records, ordered according to the events that the database processes. Record possibilities:

```
Begin  LSN, TID            # Begin transaction
End    LSN, TID, PrevLSN   # Finish transaction (either abort or commit)
# To record update to state
Update LSN, TID, PrevLSN, pageID, offset, old value, new value
```

PrevLSN forms a backward chain of operations for each transaction.

Storing both old & new values makes it possible to either REDO an update when trying to bring page up to date, or UNDO an update, reverting to an earlier version.

Other (in memory) data structures:

Transaction table: Records all transactions that have not yet been recorded permanently to disk. PrevLSN field to serve as head of list for that transaction.

Dirty Page Table: List of all pages held in memory that have not yet been written to disk.

Committing a transaction:

Not difficult. Simply make sure log file stored to disk up through commit entry. Don’t need to update actual disk pages, since log file gives instructions on how to bring them up to date.

Can keep "tail" of logfile in memory. These will contain most recent entries, but none of them are commits. If have a crash and the tail gets wiped out, then the partially executed transactions will lost, but can still bring system to reliable state.

Aborting a transaction:

Locate last entry from transaction table. Undo all updates that have occurred thus far. Use chain of PrevLSN entries for transaction to revert in-memory pages to their state at beginning of transaction. Might reach point where page stored on disk requires undo operations. Don’t change disk pages. (Come back to this later)

Recovery requires passing over log file 3 times:

1. Analysis. Reconstruct transaction table & dirty page table. Get copies of all of these pages at beginning. Figure out what is required for other 2 phases.

2. Recovery. Replay log file forward, making all updates to the dirty pages. Bring everything to state that occurred at time of crash. This involves even executing those transactions that were not yet committed at time of crash. (Required to get consistent state of locks.)

3. Undo. Replay log file backwards, reverting any changes made by transactions that had not committed at time of crash. This is done using the PrevLSN pointers for the uncommitted transactions. Once reach a Begin transaction entry, write an End transaction record to log file.

A subtle point. When performing undo operations, add "compensation log records" (CLRs) to log file. Fields include:

CLR LSN, TID, PrevLSN, page ID, offset, new value, UndoNxtLSN

The "new value" here is the old value of the update record being processed. The UndoNxtLSN field is the PrevLSN number of the update record being undo.

Why CLRs:

* In case have crash during recovery.
* In case need to abort transaction where part of state already stored to disk.
* Allows us to get rid of earlier parts of log file.

How this works: Turns future undo operation into redo operation. E.g., if aborting but copy on disk contains some modifications due to transaction, then CLRs become a record of how to go from that page to one where all updates reverted.

Integration with two-phase commit.

Can generalize WAL to deal with 2PC by having log file entries that capture 2PC operations. Additional log file entries:

* Coordinator. Includes list of participants
* Participant. Indicates coordinator.
* Vote to abort or commit
* Indication from coordinator to abort or commit.

As described, would need to replay operations back to beginning of time every time need to recover, and would need to keep log file forever. In fact, entire state of database would be captures in log file, not in data pages stored on disk.

In practice, do periodic CHECKPOINT of system state and pruning of log file. Can:

* Store any dirty pages back to disk. Must put indication in log file that have done so.
* Save copies of TT and DPT
* Prune initial portion of log file such that all transactions up to that point have been committed or aborted.