

# Database Recovery

- Face it —  $\$ \# ! \dagger$  happens
- While all systems should have some sort of *recovery scheme* in the event of failure, this is particularly important for databases
- Specifically, we want a database recovery scheme to restore a database to the most recent possible *consistent state* that existed before the failure
- We also want *high availability*: minimize the time during which the database is not usable

## Failure Classification

- *Transaction failure*: Individual transactions fail
  - ◆ *Logical error*: Internal problem within the transaction
  - ◆ *System error*: External problem during transaction execution (e.g., deadlock)
- *System crash*: Problem with overall database server execution; terminates the current process
  - ◆ *Fail-stop assumption*: Data in non-volatile storage is unharmed in the event of a system crash
- *Disk failure*: Problem with storage media

# Data Storage

- *Volatile storage*: Main or cache memory; very fast, but does not survive a system crash
- *Nonvolatile storage*: Disks, tapes; significantly slower due to eletromechanical element, survives system crashes
  - ◇ *Flash memory* is nonvolatile but fast — traditionally too small for databases, but that is changing
- *Stable storage*: Survives *everything*
  - ◇ Of course, this is just a theoretical ideal — the best we can do is make data loss sufficiently unlikely
- We can approximate stable storage through redundant copies, copies at different physical locations, and controlled updates of these copies
  - ◇ Write to one copy first; upon successful initial write, then write to second copy
  - ◇ Recovery involves comparing the copies, and replacing the copy with a detectable error (i.e., checksum) or replacing the copy that was written first if no detectable error
- Data access involves different layers of storage space: *physical blocks* on disk, *buffer blocks* in a designated main memory *disk buffer*, and *private work areas* for each transaction, also in main memory
  - ◇ *input(B)* and *output(B)* transfer between physical and buffer blocks
  - ◇ *read(X)* and *write(X)* transfer between buffer blocks and the work area, possibly triggering *input(B<sub>x</sub>)* automatically
  - ◇ A *buffer manager* decides when buffer blocks should be written back to physical blocks
  - ◇ When needed, the database system can perform a *force-output* with a direct *output(B)* call

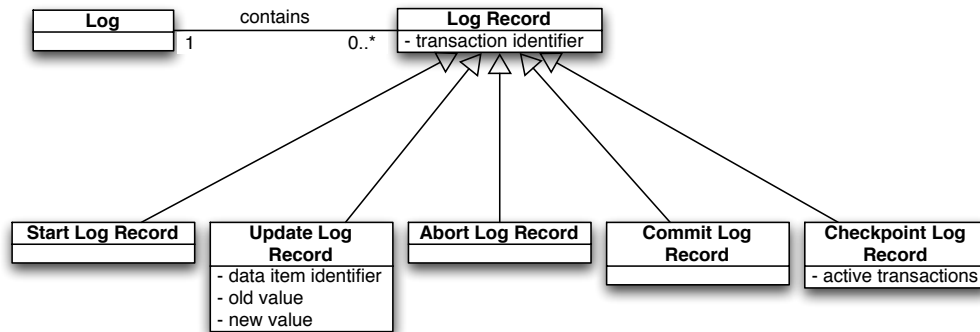
# Recovery and Atomicity

- We can't perform *output(B)* in the middle of a transaction — if it fails midway, then the database will be in an inconsistent state, thus violating a transaction's atomicity property
- Rule of thumb: modify the nonvolatile database storage only after we are sure that a transaction will commit
- General approach: first record a transaction's modifications in a separate space, then transfer from that space to database blocks only after the transaction commits successfully

## Log-Based Recovery

- *Logs* are the most widely-used approach for recording database modifications
- A log consists of a sequence of *log records*
- A database write is recorded in an *update log record*, which consists of:
  - ◇ *Transaction identifier* indicates the writing transaction
  - ◇ *Data-item identifier* indicates what was written (typically on-disk location of item)
  - ◇ *Old, new values* of the item before and after the write

# Log Data Model



- Write the log record before modifying the database
- Undo is possible through the *old value* attribute
- Log should be written to stable storage

## Deferred Database Modification

- Perform *write* operations only after the transaction partially commits (i.e., performs last action)
- Ignore log records for failed transactions
- $redo(T_i)$  operation rewrites all of the data items modified by  $T_i$  (as recorded in the log) — must be *idempotent*, meaning that executing  $redo(T_i)$  once must have the exact effect as executing it multiple times
- To recover after a failure: perform  $redo(T_i)$  for  $T_i$  that have both *start* and *commit* log records

# Immediate Database Modification

- Write database modifications immediately; while a transaction is active, its modifications are called *uncommitted modifications*
- Upon failure, perform  $undo(T_i)$  to write old values back to the database for failed transactions — again, must be idempotent
- To recover, perform  $undo(T_i)$  for transactions that have a *start* record but not *commit*, and perform  $redo(T_i)$  for transactions that have both *start* and *commit* records

## Checkpoints

- Not practical to scan the *entire* log upon failure — after all, a lot of writes don't need to be redone
- So we perform a periodic *checkpoint operation*: output (1) pending log records to stable storage, (2) modified buffer blocks to disk, then (3) *checkpoint log record* to stable storage
- On failure, (1) find the most recent checkpoint, (2) find the last transaction started before that checkpoint, then (3) perform recovery for that transaction and all those after

# Recovery with Concurrent Transactions

Some things to consider when multiple concurrent transactions are active (note that we still have one disk buffer and one log):

- During an undo, we should scan the log backward, to ensure the proper restoration of an item that has been written multiple times
- Any transaction that writes  $Q$  must either commit or be rolled back (with accompanying *undos* from the log) before another transaction writes  $Q$  again — strict two-phase locking will ensure this
  
- Checkpoint log records must include a list of active transactions at the time of the checkpoint — otherwise, we wouldn't know how when to stop scanning the log
- Recovery algorithm then requires two passes: one to determine the transactions to undo and redo, and another to perform the actual undo and redo
  - ◆ Scan the log backward, up to the most recent checkpoint log record
  - ◆ Every committed transaction goes to the *redo-list*
  - ◆ Every started transaction not already in the redo-list goes to the undo-list
  - ◆ Transactions in the checkpoint log record not already in the redo-list go to the undo-list
- The second pass then performs the data modifications:
  - ◆ Scanning backward, undo all updates by transactions in the undo-list
  - ◆ Going forward from the most recent checkpoint log record, redo all updates made by transactions in the redo-list

# Buffer Management

- Recall that the journey from main memory to disk still goes through a volatile section — the disk buffer
- Unlike main memory, the disk buffer's minimum storage matches the disk — typically *blocks*
- Since the information we read/write (log records, data items) tend to be much smaller than a disk block, it's impractical to write out entire blocks for every single logical read/write from main memory
- So we need some tweaks for efficiency's sake, without detracting from our recovery techniques

## Log-Record Buffers

- Since log records are small compared to blocks, it's costly to touch stable storage for every log record
- So we buffer the logs too...but then, those would be lost on a system crash, so we need to make sure that log-record buffering doesn't break recoverability:
  - ◆ Transactions get the commit state only after their commit log records reach stable storage
  - ◆ The commit log record should be the last log record to reach stable storage
  - ◆ Before a data block is written to the database, all log records for data items in that block must reach stable storage
- That last rule is called the *write-ahead logging (WAL) rule* — keep that in mind, because you'll hear it again later

# Database Buffers

- Database size is generally much larger than available main memory, so we sometimes need to *swap out* blocks as we read data items from the database
  - ◇ Sounds like virtual memory? That's because it is!
- With the write-ahead logging rule, we can't swap out a block until log records for that block's data are written
- While waiting on the write-ahead, we acquire a *latch* — a small, short-duration lock — on the block whose log entries are being written, so that no writes on that block can take place during the write-ahead

## But What About Disk Failures?

- So far, all of these recovery techniques pertain to *system crash* errors — loss of volatile storage
- To protect against disk failures, where our nonvolatile storage media goes bad, we perform operations that are very similar to standard file backup — we *dump* or copy the entire database to another form of nonvolatile or stable storage
- We disallow transactions during a dump, and dumps should include log records too; we can even create a new *dump log record* type so the dump goes in the log

# Remote Backup

- To complete our recovery picture, we must realize that failures may have external causes — fires, floods, black holes — and so we need to have some form of physical separation for our storage devices
- We say that our main database server resides (in the real world) in the *primary site*, and we establish a *remote backup* or *secondary site* that is sufficiently separated from the primary site to protect against external disasters
  
- Practical synchronization can be done by sending log records from the primary to the secondary site; full replication (dumps) may occur, but less frequently
- Upon primary site failure, the secondary site (1) performs recovery as discussed previously, then (2) takes over transaction processing
- A few things to consider with remote backups:
  - ◆ Reliable detection of failure (i.e., no false alarms)
  - ◆ Transfer of control — when the primary site comes back, it can either become primary again, or become the new secondary site
  - ◆ Minimal recovery time — ideally, a secondary site should act upon the log records that it receives, making it a *hot spare* of the primary site
  - ◆ Time to commit — a tradeoff exists here between full transaction durability and commit times for a transaction: to ensure full durability, a transaction at the primary site shouldn't commit until its log records have reached the secondary site, thus requiring more time

# Advanced Recovery Techniques

- Note how recovery may trade off concurrency and availability to protect against lost or inconsistent data
- While this is worthwhile, more sophisticated recovery techniques seek to have their cake and eat it too
- The state of the art in recovery is the *algorithm for recovery and isolation exploiting semantics (ARIES)* — details in the text and the literature
- While more complex, ARIES achieves less overhead and greater efficiency while maintaining recoverability

## Recovery in PostgreSQL

- Again, concepts hit reality fairly directly with PostgreSQL — it uses write-ahead logging (WAL) to facilitate recovery
- Assorted parameters configure the recovery system:
  - ◆ Buffer-to-disk (“sync” in PostgreSQL terms) settings
  - ◆ Commit delays to allow multiple transaction log entries to reach the disk in a single call
  - ◆ Maximum “distance” between checkpoints (both in terms of log size and passage of time)