Relational Database System Implementation

CS122 – Lecture 15

Winter Term, 2017-2018

Transaction Processing

- Last time, introduced transaction processing
- ACID properties:
  - Atomicity, consistency, isolation, durability
- Began talking about implementing atomicity and durability
- Shadow-copy technique:
  - When a transaction first writes to the database, make a complete copy of the database
  - A “db-pointer” refers to the current copy of the database
  - All writes go against the new copy of the database
  - At commit time, sync all files in the new copy, then update and sync the db-pointer
- Primary limitation of shadow copies is that database can only have one transaction in progress at a time

Write-Ahead Logging

- Instead of duplicating the entire database and writing to a copy, would like to write to database files *in-place*
- To provide atomicity and durability, maintain a single *log file* describing all changes made to the database
  - OS allows us to update this single file atomically
  - Can interleave changes from different txns in the log
- Require that the log file must reflect *all* data changes, sync'd to disk, *before* any table files are written
  - This technique is called *write-ahead logging* (WAL)
- When DB writes to the log that a txn is committed, it is!
  - This write must also make it to the disk itself (e.g. fsync())
  - A single atomic operation against persistent storage

Data Access

- As before, don't model transactions at the SQL level!
  - Rather, model simple operations against data items
- Also, must buffer disk access to improve performance
- Two-level storage hierarchy:
  - Database transactions interact with buffer pages
  - Buffer pages are transferred to and from disk storage
- input(B) – transfer physical block B to main memory
  - Data is transferred into a page of the Buffer Manager
- output(B) – transfer physical block B back to disk
  - May or may not also include a sync of the file that B is in

Data Access (2)

- Transactions perform computations in local variables, and simply read/write data items in buffer pages
- read(*X*) – read data item *X* into a local variable
  - If block B_X that *X* resides in isn't in memory, database also issues input(B_X) to read into memory
- write(*X*) – write a local variable into data item *X*
  - Does not require block B_X to be written back to disk!
- If DB crashes after write(*X*), the change could be lost!
  - To ensure new *X* is recorded, database must eventually *force* B_X to be stored to disk, by calling output(B_X)

Database Modifications

- Could require that a transaction does not modify any database state until it is committed
  - Called *deferred modification*
- Presents several challenges:
  - A transaction must make local copies of everything it modifies
  - If a transaction reads a value it has written, must make sure it reads the local copy, not the original value
- Could also allow a transaction to modify database state before it is committed
  - Called *immediate modification*
- With immediate modification, must ensure we can properly roll back all changes that any transaction might make!

Write-Ahead Log Records

- Log-file records important transaction state-changes
- Transaction-status log records:
  - <T_i start> Transaction T_i has been started
  - <T_i commit> Transaction T_i has been committed
  - <T_i abort> Transaction T_i has been aborted
- Every transaction has a unique ID
  - (usually a 32-bit or 64-bit integer value)
- Completed transactions will have a <T_i start> record, and either <T_i commit> or <T_i abort>, in the log file
- Incomplete transactions will only have a <T_i start> record in the log file

Write-Ahead Log Records (2)

- Log file also records all modifications to database state
- Update log records: <T_i, X_j, V_1, V_2>
  - Transaction T_i wrote to value X_j
  - Old value of X_j was V_1, and new value is V_2
- X_j specifies the data item that was written
  - In discussion, usually think of X_j as a specific column
  - In implementations, X_j is actually usually a page of a specific data file
    - e.g. store file, block no., old and new state of the block as deltas
- Other kinds of database updates too!
  - e.g. create a new data file; extend a file's size by one page

Write-Ahead Log Records (3)

- Write-ahead logging supports multiple concurrent transactions
  - Records for different transactions are interleaved in the log file
- Database is responsible for ensuring that transactions don't interfere with each other in nasty ways
  - i.e. that read(X_j) and write(X_j) operations from different txns are properly scheduled to maintain isolation
  - Mechanism is called *concurrency-control*
  - For now, we will assume this is properly taken care of!

Logging Operations

- Write-ahead log records every database state-change
  - Log is always written and synchronized to disk before any other data files are modified on disk
- Earlier example: transfer $50 from account *A* to *B*
  - Every write to a data item must be preceded by a record written to write-ahead log
  - Commit record must be written to log before transaction is reported as committed!

T_1: read(A);
$A := A - 50$;
write(A); ③
read(B);
$B := B + 50$;
write(B); ⑤
commit. ⑦

Write-Ahead Log:

T_1: start	①
T_1: A, 100, 50	②
T_1: B, 40, 90	④
T_1: commit	⑥

Rolling Back a Transaction

- We can rollback transactions with our write-ahead log!
- Transfer $50 from account *A* to *B*:
  - This time, transaction is aborted at attempt to read(*B*)
  - Must undo all state-changes made in the transaction
- Scan backward through write-ahead log, undoing all changes made by transaction T_1
  - Stop when we reach <T_1: start> record

T_1: read(*A*);
A := *A* – 50;
write(*A*); ③
read(*B*); ④ ABORT!

Write-Ahead Log:

T_1: start	①
T_1: *A*, 100, 50	②

scan →

Rolling Back a Transaction (2)

- Update record specifies that *A* was 100 before write…
  - Roll back the change by restoring *A* to 100
- When undoing change, write a *compensation log record* to the write-ahead log
  - Compensates for previous state-change being undone
  - Also called a *redo-only* log record: this write is rolling back a state-change, so it will never be undone

T_1: read(A);
$A := A - 50$;
write(A); (3)
read(B); (4) ABORT!
$A := 100$;
write(A); (6)

Write-Ahead Log:

	T_1: start	(1)
scan →	T_1: A, 100, 50	(2)
	T_1 redo-only A: 100	(5)

Rolling Back a Transaction (3)

- When all T_1 state-changes have been reversed, record <T_1: abort> record to the log
- Transaction is now aborted.
  - All state changes have been rolled back
  - Write-ahead log records both the compensating writes, and the final transaction status

T_1: read(A);
A := A – 50;
write(A); ③
read(B); ④ ABORT!
A := 100;
write(A); ⑥
abort. ⑧

scan

Write-Ahead Log:

T_1: start	①
T_1: A, 100, 50	②
T_1 redo-only A: 100	⑤
T_1: abort	⑦

Force, or No-Force?

- Write-ahead logging rule (a.k.a. WAL rule):
  - All database state changes must be recorded to the log on disk, before any table-files are changed on disk
- At commit time, are we *required* to force all modified table-pages out to disk?
  - In other words, can a transaction be reported to the client as "committed," if not all table files have been written?
- We are *not required* to write all modified table pages at commit time, if the database follows the WAL rule
  - We know that all changes are recorded in log file on disk, even if the table pages themselves haven't yet been flushed to disk
  - Won't violate durability by reporting transaction "committed"

Force, or No-Force? (2)

- *Force* policy:
  - Database force-outputs all dirty table-pages before a transaction is reported as "committed"
- *No-force* policy:
  - Database can report a transaction as "committed" before all dirty table-pages are output
- No-force policy is much faster than force policy:
  - Writes from multiple transactions can be performed against in-memory table pages without incurring disk IO
- As long as the DB records all data-changes to the WAL on disk at commit time, it can use the no-force policy

Steal, or No-Steal?

- A similar question:
- Are we *forbidden* from writing modified table-pages out to disk before a transaction commits?
  - In other words, can we allow table changes performed by an incomplete transaction to reach the disk?
- We are *not* forbidden from writing dirty table-pages for active txns, as long as we follow the WAL-rule:
  - Not only does the log record the new value for each modified value, but it also records the old value
  - Log will always contain sufficient information, on disk, to undo any state changes written to table pages on disk

Steal, or No-Steal? (2)

- *Steal* policy:
  - Database is allowed to write dirty table-pages to disk, even if the transaction is still active
- *No-steal* policy:
  - Database is not allowed to write dirty table-pages to disk until the transaction is being committed
- Steal policy allows much larger database updates to be performed
  - Doesn't require a large amount of buffer memory to hold uncommitted changes
  - Modified pages can be written to disk to free up buffer space
- As long as DB follows WAL rule, it can use the steal policy

Crash Recovery!

- Write-ahead logging rule:
  - All database state changes must be recorded to the log on disk, before any table-files are changed on disk
- If the system crashes, all important state changes will already be recorded to the log file
  - All completed transactions will record:
    - All modifications performed by the transaction
    - A <T_i: commit> or <T_i: abort> record for the transaction
  - All incomplete transactions will record:
    - All modifications performed by transaction before the crash
- Table files won't necessarily reflect *all* of these changes

Crash Recovery! (2)

- The *recovery* process performs two critical tasks:
  - It synchronizes the current state of all data files with the current state of the write-ahead log
  - It completes all incomplete transactions

- Policy for incomplete transactions:
  - At recovery, incomplete transactions are <u>aborted</u>

Crash Recovery! (3)

- Completed transactions:
  - Log will contain a <T_i: start> record, plus a matching <T_i: commit> or <T_i: abort> record
  - Ensure that data files properly reflect all transaction state-changes
- Incomplete transactions:
  - Log will contain a <T_i: start> record, but <u>no</u> matching <T_i: commit> or <T_i: abort> record
  - Ensure that all transaction state-changes are properly removed from the data files
- Recovery is performed in two phases

Recovery Processing

- Phase 1: redo phase
  - Scan forward through log, redoing updates from all txns, in the exact order they appear in the transaction log
  - For every update record <T_i, X_j, V_1, V_2> in the log, set X_j to the new value V_2 recorded in the log
  - For every redo-only record <T_i, X_j, V>, set X_j to V
  - This is called *repeating history*
- During this phase, maintain a set of incomplete txns:
  - When a <T_i: start> record is found, add T_i to incompletes
  - When a <T_i: commit> or <T_i: abort> record is found, remove T_i from incompletes

Recovery Processing (3)

- Phase 2: undo phase
  - Scan backward through log, rolling back incomplete txns
  - Procedure is identical to rolling back a single transaction
- If record's transaction ID is in the set of incompletes:
  - If the record is a normal update record: $<T_i, X_j, V_1, V_2>$
    - Write a redo-only record $<T_i, X_j, V_1>$ to end of log
    - Undo the state change: Restore X_j to the old value V_1
  - If the record is a <T_i: start> record:
    - Write a <T_i: abort> record to end of log
    - Remove transaction T_i from the set of incompletes
- Undo phase is done when the incomplete-set is empty

Redo-Only Records

- Redo-only logs greatly simplify recovery processing
- To rollback a transaction, must undo its state-changes in reverse order of its updates
  - A txn may write to a given data item multiple times…
  - At end of rollback, item must reflect the *original* value
- Cannot handle previously aborted txns in the undo phase:
  - Could undo a write performed by another committed txn!
- Example:
  - T_1 changes A from 100 to 200, then aborts.
  - T_2 changes A from 100 to 50, then commits.
  - Rolling back T_1 in undo phase would overwrite T_2's write to A in redo phase ☹

Write-Ahead Log:

T_1: start
T_1: A, 100, 200
T_1: abort
T_2: start
T_2: A, 100, 50
T_2: commit

Redo-Only Records (2)

- In cases of previously-aborted transactions, must undo the transaction's writes during the redo phase
  - Scan backwards through the log file, undoing all changes made specifically by T_1…
  - Very slow – introduces many extra disk seeks!
- Redo-only records make it fast to replay the rollback of previously aborted txns
  - Just keep scanning forward through the log, applying redo-only records for T_1

Write-Ahead Log:

T_1: start
T_1: A, 100, 200
T_1: abort
T_2: start
T_2: A, 100, 50
T_2: commit

Crashes During Recovery

- System could also crash during recovery…
  - Must still be able to recover, even if the last crash occurred during recovery processing!
- Recovery procedure must be *idempotent*:
  - Results of recovery processing must be the same, whether it is applied once or multiple times
- As described, this recovery procedure is idempotent
  - We record the actual old and new values of data items
  - Logged values aren’t relative to other operations
  - Worst case is that a data item will be “restored” multiple times (extra writes, but we don’t mind)

Read-Only Transactions

- Frequently have many transactions that only read the database
  - Nothing to redo or undo for such transactions…
  - No need to represent them in the write-ahead log!
- Only record a transaction to the write-ahead log when it actually changes state in the database
  - e.g. at first state-change, write <T_i: start> and also the first update-record to the log

Logging Performance

- So far, assumed that new write-ahead log records are always written and sync'd to the log file immediately
- Imposes a very expensive IO penalty on the system!
- Would rather write multiple log records to disk at once
  - Log file is written in units of blocks, just like table files
- Database loads pages of table-files into buffer space…
  - Table data is modified in memory
- Database system can control when these buffer pages are flushed back to disk
  - Database can coordinate the output of table blocks, with the writing of log records

Logging Performance (2)

- A transaction T_i cannot be reported as "committed" until:
  - A <T_i: commit> record is written to the log, and sync'd to disk
- Before the <T_i: commit> record can be logged:
  - All other log records for T_i must also be written to the WAL
  - These can be sync'd at same time <T_i: commit> is sync'd
  - (In other words, these can remain in buffer until it is time to write the <T_i: commit> record.)
- For this to work, must constrain that a table-page cannot be flushed from the Buffer Manager to disk until:
  - All write-ahead log records for that page have been written to the log file, and sync'd to disk
  - (This is the WAL rule)

Logging Performance (3)

- General rules:
  - Before a transaction is reported as "committed", must ensure that all logs have been sync'd to disk
  - Before a dirty table-page is flushed to disk, must ensure that all logs pertaining to that page have been sync'd to disk
- These rules specify the absolute latest that log records must be written and sync'd to disk
  - If current log-page is only partially full at this point, write it out anyway! Required for atomicity, durability.
- Can certainly write logs to disk earlier, if we need to free up buffer space
  - Still don't require syncing until one of the conditions above