Transaction Management Overview
R & G Chapter 16

There are three side effects of acid. Enhanced long term memory, decreased short term memory, and I forget the third.
- Timothy Leary

Concurrent Control & Recovery

- Very valuable properties of DBMSs
  - without these, DBMSs would be much less useful
- Based on concept of transactions with ACID properties
- Remainder of the lectures discuss these issues

Statement of Problem

- Concurrent execution of independent transactions
  - utilization/throughput ("hide" waiting for I/Os.)
  - response time
  - fairness
- Example:

  T1:
  \[ t0: \text{tmp1 := read}(X) \]
  \[ t1: \text{tmp1 := tmp1} - 20 \]
  \[ t2: \text{write tmp1 into X} \]

  T2:
  \[ t0: \text{tmp2 := read}(X) \]
  \[ t1: \text{tmp2 := tmp2 + 10} \]
  \[ t2: \text{write tmp2 into X} \]

Statement of problem (cont.)

- Arbitrary interleaving can lead to
  - Temporary inconsistency (ok, unavoidable)
  - "Permanent" inconsistency (bad!)
- Need formal correctness criteria.

Definitions

- A program may carry out many operations on the data retrieved from the database
- However, the DBMS is only concerned about what data is read/written from/to the database.
  - database - a fixed set of named data objects \((A, B, C, \ldots)\)
- transaction - a sequence of read and write operations \((\text{read}(A), \text{write}(B), \ldots)\)
  - DBMS’s abstract view of a user program
Correctness criteria: The **ACID** properties

- **A** tomicity: All actions in the Xact happen, or none happen.
- **C** onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one Xact is isolated from that of other Xacts.
- **D** urability: If a Xact commits, its effects persist.

**Atomicity of Transactions**

- Two possible outcomes of executing a transaction:
  - Xact might commit after completing all its actions
  - or it could abort (or be aborted by the DBMS) after executing some actions.
- DBMS guarantees that Xacts are **atomic**.
  - From user’s point of view: Xact always either executes all its actions, or executes no actions at all.

**Mechanisms for Ensuring Atomicity**

- **One approach**: LOGGING
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- **Another approach**: SHADOW PAGES
  - (ask me after class if you’re curious)
- Logging used by modern systems, because of need for audit trail and for efficiency reasons.

**Transaction Consistency**

- “Consistency” - data in DBMS is accurate in modeling real world and follows integrity constraints
- User must ensure transaction consistent by itself
  - I.e., if DBMS consistent before Xact, it will be after also
- Key point:
  ```
  consistent database S1  transaction T  consistent database S2
  ```

**Transaction Consistency (cont.)**

- Recall: Integrity constraints
  - must be true for DB to be considered consistent
  - Examples:
    1. FOREIGN KEY R.sid REFERENCES S
    2. ACCT-BAL >= 0
- System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).
  - Beyond this, DBMS does not understand the semantics of the data.
  - e.g., it does not understand how interest on a bank account is computed

**Isolation of Transactions**

- Users submit transactions, and
- Each transaction executes as if it was running by itself.
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- Many techniques have been developed. Fall into two basic categories:
  - Pessimistic – don’t let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them after they happen.
Example

- Consider two transactions (Xacts):
  
  \begin{align*}
  T1: & \text{BEGIN } A=A+100, \; B=B-100 \text{ END} \\
  T2: & \text{BEGIN } A=1.06^\ast A, \; B=1.06^\ast B \text{ END}
  \end{align*}

- 1st xact transfers $100 from B's account to A's
- 2nd credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2???
  
  \begin{itemize}
  \item $2000 \ast 1.06 = $2120
  \item There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running serially in some order.
  \end{itemize}

Example (Contd.)

- Legal outcomes: A=1166, B=954 or A=1160, B=960
- Consider a possible interleaved schedule:
  
  \begin{align*}
  T1: & \text{A=A+100, B=B-100} \\
  T2: & \text{A=1.06^\ast A, B=1.06^\ast B}
  \end{align*}

- This is OK (same as T1;T2). But what about:
  
  \begin{align*}
  T1: & \text{A=A+100, } \; B=B-100 \\
  T2: & \text{A=1.06^\ast A, } \; B=1.06^\ast B
  \end{align*}

- Result: A=1166, B=960; A+B = 2126, bank loses $6
- The DBMS's view of the second schedule:
  
  \begin{align*}
  T1: & \text{R(A), W(A), R(B), W(B)} \\
  T2: & \text{R(A), W(A), R(B), W(B)}
  \end{align*}

Formal Properties of Schedules

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.
- **Equivalent schedules**: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.
  (Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- **Reading Uncommitted Data (WR Conflicts, "dirty reads")**:
  
  \begin{align*}
  T1: & \text{R(A), W(A), R(B), W(B), Abort} \\
  T2: & \text{R(A), W(A), C}
  \end{align*}

- **Unrepeatable Reads (RW Conflicts)**:
  
  \begin{align*}
  T1: & \text{R(A), R(A), W(A), C} \\
  T2: & \text{R(A), W(A), C}
  \end{align*}

Anomalies (Continued)

- **Overwriting Uncommitted Data (WW Conflicts)**:
  
  \begin{align*}
  T1: & \text{W(A), W(B), C} \\
  T2: & \text{W(A), W(B), C}
  \end{align*}

Lock-Based Concurrency Control

- Here's a simple way to allow concurrency but avoid the anomalies just described...
- **Strict Two-phase Locking (Strict 2PL) Protocol**
  
  - Each Xact must obtain an S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - System can obtain these locks automatically.
  - Lock rules:
    - If an Xact holds an X lock on an object, no other Xact can acquire a lock (S or X) on that object.
    - If an Xact holds an S lock, no other Xact can get an X lock on that object.
  - Two phases: acquiring locks, and releasing them
  - No lock is ever acquired after one has been released
  - All locks held by a transaction are released when the xact completes
- **Strict 2PL allows only serializable schedules.**
Aborting a Transaction (i.e., Rollback)

- If an xact Ti aborted, all actions must be undone.
- Also, if Tj reads object last written by Ti, Tj must be aborted!
  - Most systems avoid such cascading aborts by releasing locks only at EOT (i.e., strict locking).
  - If Ti writes an object, Tj can read this only after Ti finishes.
- To undo actions of an aborted transaction, DBMS maintains log which records every write.
- Log also used to recover from system crashes: All active Xacts at time of crash are aborted when system comes back up.

Logging (cont.)

- Write-Ahead Logging protocol
  - Log record must go to disk before the changed page!
    - implemented via a handshake between log manager and the buffer manager.
  - All log records for a transaction (including its commit record) must be written to disk before the transaction is considered “Committed”.
- All logging and CC-related activities are handled transparently by the DBMS.

The Log

- Log consists of “records” that are written sequentially.
  - Typically chained together by Xact id
  - Log is often archived on stable storage.
- Need for UNDO and/or REDO depend on Buffer Mgr.
  - UNDO required if: uncommitted data can overwrite stable version of committed data (STEAL buffer management).
  - REDO required if: xact can commit before all its updates are on disk (NO FORCE buffer management).
- The following actions are recorded in the log:
  - If Ti writes an object, write a log record with:
    - If UNDO required need “before image”
    - IF REDO required need “after image”.
  - Ti commits/aborts: a log record indicating this action.

Durability - Recovering From a Crash

- Three phases:
  - Analysis: Scan the log (forward from the most recent checkpoint) to identify all Xacts that were active at the time of the crash.
  - Redo: Redo updates as needed to ensure that all logged updates are in fact carried out and written to disk.
  - Undo: Undo writes of all Xacts that were active at the crash, working backwards in the log.
- At the end – all committed updates and only those updates are reflected in the database.
- Some care must be taken to handle the case of a crash occurring during the recovery process!

(Review) Goal: The ACID properties

- A tomicity: All actions in the Xact happen, or none happen.
- C consistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- I solution: Execution of one Xact is isolated from that of other Xacts.
- D urability: If a Xact commits, its effects persist.

What happens if system crashes between commit and flushing modified data to disk?

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Concurrency control is automatic
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts
  - Property ensured: resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) and the recovery protocol are used to:
  1. undo the actions of aborted transactions, and
  2. restore the system to a consistent state after a crash.