The UNIVERSITY of VERMONT

Benjamin Fenster
CS-201 (Arslan)
11 December 2005

Notes – Recovery from Crashes

I. Definitions
   a. A transaction is a collection of operations that perform a logical function.
   b. We would like transactions to be atomic.
   c. Problems
      i. Some data are brought from disk to memory
      ii. It may take time before data in memory are written back to disk.
      iii. The system may crash before the data are written back.
      iv. Some part of the transaction may be written back, but not all, so it won't be atomic anymore. That's bad.

II. Log-Based Recovery
   a. Assume that before any update we record the old value and new value in a log record
      i. Transaction name
      ii. Data item name
      iii. Old value
      iv. New value
   b. Write-ahead logging
      i. Write a log record before modifying the data.
      ii. Record when the transaction starts and commits.
      iii. The writes up until the commit should be permanent
      iv. The entire logical function (transaction) is done.
   c. Operations
      i. Undo T_i: Restore old values (changed as part of transaction T_i)
      ii. Redo T_i: Set the new value that was updated as part of T_i
   d. Checkpoints
      i. Output the log to non-volatile storage ("stable"). Before this time the log is stored in memory.
      ii. Output all modified data from volatile storage to stable storage
      iii. Output the log record <checkpoint> to stable storage.
   e. In the event of a crash:
      i. Find the last checkpoint. Everything up to that point is already on stable storage
      ii. Any transaction starting after a checkpoint needs to be reprocessed in some way.
      iii. For transaction T_k where <T_k commits> is in the log, Redo T_k
      iv. If <T_k starts> is in the log but not <T_k commits> then Undo T_k. It was started but wasn't finished.
      v. We'll lose some changes but the most important goal is to restore to some *consistent* point (so it's better to lose data than to have a transaction partially completed).

III. Serializability
   a. A schedule is a sequence of execute (e.g. T_1 reads A, T_2 reads B, …
   b. A schedule where every transaction is executed atomically is a serial schedule.
   c. Non-serial schedules have overlapping operations from different transactions. That doesn't inherently mean it's incorrect. It may still be perfectly fine.
   d. Two operations conflict if they operate on the same variable and at least one is a write.
   e. Any two adjacent operations in a schedule that *don't* conflict can be swapped.
   f. If a serial schedule can be obtained through such swaps, then it is serializable

IV. Locking Protocol
   a. Two types of locks:
      i. Shared: If T_i has a shared lock on data item Q, then T_i can read Q but not write. Some T_J can also read Q but not write.
      ii. Exclusive: If T_i has an exclusive lock on Q, then T_i can read and write Q. No other transaction can read or write Q.
   b. Obtaining a lock depends on existing locks being released.

c. Two-Phase Locking Protocol
    i. Growing phase: Obtain all the locks needed but do not release any locks.
    ii. Shrinking phase: Release locks but do not acquire any new ones.
    iii. This ensures any schedule obtained under this protocol will be serializable.
d. Timestamp Protocol
    i. Assume there is a common clock for all transactions
    ii. Each transaction gets a unique timestamp TS(T)
    iii. If TS(S) < TS(T) then the produced schedule will be equivalent to the serial schedule where S appears before T.
    iv. Each data item Q has:
        1. Write-Timestamp(Q): Largest timestamp of any transaction that wrote Q
        2. Read-Timestamp(Q): Largest timestamp of any transaction that read Q.
    v. Use timestamps to decide if an operation should be executed or an entire transaction should be rolled back
        1. Reading:
            a. If TS(T) < W-timestamp(Q), rollback T. T needs the old value.
            b. If TS(T) > W-timestamp(Q) perform the read, and update R-timestamp(Q) = max(R-timestamp(Q), TS(T))
            c. We're pretending that all instructions for T occur at "time" TS(T)
        2. Writing:
            a. If TS(T) < R-timestamp(Q), rollback T. Another transaction, supposedly occurring after T, has read the value; if we update it then that other transaction will have gotten the wrong value.
            b. If TS(T) < W-timestamp(Q), rollback T. Another transaction already wrote a newer value.
            c. Otherwise accept the write, update W-timestamp(Q) = TS(T)
    vi. After any rollback, restart T with a new timestamp (joins the ready queue; treated as a new transaction).
e. Neither protocol recognizes all serializable schedules! They're conservative ways of getting schedules that are guaranteed to be serializable.