Crash Recovery

Three Types of Failure

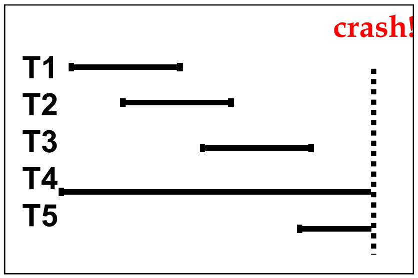
- Transaction failure
- Media Failure (Hard Crash)
- System Failure (Soft Crash)

Review: The ACID properties

- Atomicity: All actions in the transaction happen, or none happens
- Consistency: Transaction transforms DB from consistent states to consistent states
- Isolation: Execution of one transaction is isolated from that of other transactions
- Durability: If a transaction commits, its effects persist
- The *Recovery Manager* guarantees Atomicity & Durability

Motivation

- Atomicity: Transactions may abort
- Durability: DBMS may stop running



Desired behavior at system restart:

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).

Assumptions

- Concurrency control is in effect
- Updates are happening “in place”, i.e. data is overwritten on (deleted from) the disk

Buffer Management

- Features of buffer:
  - Volatile. Update in buffer may get lost at system crash
  - Limited space. At some point, no free buffer frame is available
- What can a DBMS do?
  - Update data in buffer frames, *force write* frames to disk if necessary
  - If no frame is available, *steal frame* from a running transaction (need to flush updated frames first)

Force vs Steal: Options

- Force every update.
  - Poor response time
- No force.
  - What if transaction commits & DBMS crashes before updates are written to disk?
- No steal.
  - Poor throughput
- Steal.
  - What if transaction aborts after some of its frames are flushed?

Force vs Steal: Ideas

	No Steal	Steal
Force	Simple	
No Force		**Desired**

- No force. Keep enough info on disk at commit time to prepare to *redo* committed updates
- Steal. Keep enough info on disk at flush time to prepare to *undo* aborted updates

Logging

- A *log* is a file of entries about DB updates
  - Sequentially recorded, first in buffer, then on *stable storage*
  - Contain minimal information required to perform undo & redo operations of transactions
- Recovery Manager keeps a log of DB operations
  - Record log during normal operation
  - Use log to redo/undo transactions at system restart time

Log Entries

- <T start>
  - Each transaction has a unique, system generated id
- <T, X, oldVal, newVal> (or <T, X, newVal>)
  - oldVal is needed for undo, newVal is needed for redo
- <T commit>
- <T abort>

Write-Ahead Logging (WAL)

- RM must coordinate the writing of log entries with writing of updated buffer frames
- The *Write-Ahead Logging Protocol* :
  - Must force the log entry for an update before the updated frame is written to disk (to guarantee Atomicity)
  - Must write all log entries for a transaction before it commits (guarantees Durability)

When to Update Database?

- Every update must be done in buffer first
- Every update must be recorded in log
- When can DB update start?
  - *Immediate Update* : As soon as an update is made in buffer & in log
  - *Deferred Update* : As soon as transaction is ready to commit
- When is DB update done?
  - Whenever updated frame is stolen
  - Whenever updated frame is forced out

Redo and Undo

- Undo
  - For each updated data, restore its old value
  - Must be done in reverse order
  - Must be repeatable
  - Needed for recovery & (normal) abort
- Redo
  - For each updated data, set to its new value
  - Must be done in forward order
  - Must be repeatable
  - Needed for recovery

Recovery Algorithms

- Each algorithm describes
  - Normal operations:

    read, write, commit, abort
  - Restart operation
- Types of algorithms:
  - Undo/redo
  - No-undo/redo
  - Undo/no-redo
  - No-undo/no-redo

Undo/Redo Algorithm

- Buffer Management:
  - Immediate Update
  - Steal/no-force
- Normal Operations:
  - Update in buffer, record in log, frame can be stolen
  - Commit places a log entry <T, commit>
  - Abort performs an undo in buffer, then appends a log entry <T, abort>

Undo/Redo Algorithm (cont.)

- Restart Operation:
  - Undo aborted & uncommitted transactions
    - Read log backwards & make a commit list (CL) (transactions with <T, commit> in log)
    - At meantime, undo every update made by transactions not in CL (done in buffer)
  - Redo committed transactions
    - Read log forward and redo every update of committed transactions (in buffer)

Undo/Redo Algorithm: Example

- Log on stable storage at system crash:

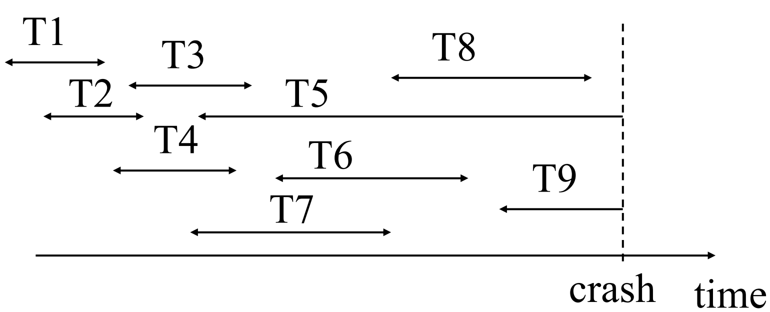
<T1, start><T1, D, 25, 20><T1, commit>
<T4, start><T4, B, 10, 15><T2, start>
<T2, D, 20, 12><T4, A, 10, 24><T4, commit>
<T3, start><T3, A, 24, 30><T2, D, 12, 35>

- Restart operations:
  - CL = {T4, T1}
  - Undo T2, T3: D=20, A=24
  - Redo T1, T4: D = 20, B = 15, A = 24

Undo/Redo Algorithm: Discussion

- Guarantees both Atomicity and durability.
- Efficient normal operations
- Higher degree of concurrency may be obtained.
  - Intermediate result of a transaction is visible to other transactions, after it is written on disk.
- May cause cascading abort.
  - Best used with cascadeless schedules

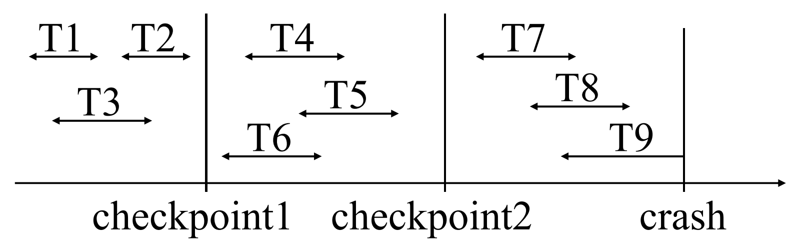
How Far To Trace Back?



- How far should the log be traced back?
  - May have to go back to the beginning
  - May perform many unnecessary redo/undo

Checkpoint: The Idea

- Create *checkpoint* at which database is up-to-date and consistent.



- Only redo T7, T8 and undo T9.

Checkpoint Operations

- Stop accepting new transaction
- Suspend running transactions
- Force write log buffer frame
- Force write all updated frames
- Append <checkpoint, AL> to log, where AL is a list of active transactions at checkpoint time
- Resume normal execution

Recovery With Checkpoint

- Log entries at system crash:

<T1, start><T1, D, 25, 20><T1, commit>
<T4, start><T4, B, 10, 15><T2, start>
<T2, D, 20, 12><checkpoint, (T2, T4)>
<T4, A, 10, 24><T4, commit><T3, start>
<T3, A, 24, 30><T2, D, 12, 35>

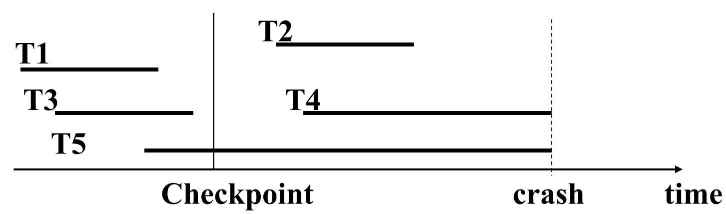
- Restart operations:
  - Undo T2, T3 (all operations)
  - Redo T4 (only operations after checkpoint)

No-Undo/Redo Algorithm

- Buffer Management:
  - Deferred Update
  - No-force, No steal
- Normal Operations:
  - Updates are done in buffer & recorded in log, no steal of buffer frames is allowed
  - Abort appends <T, abort>
  - Commit appends <T, commit>, and allows steal of buffer frames

No-Undo/Redo Algorithm (cont.)

- Restart operation
  - Redo transactions that commit after the most recent checkpoint (in forward order)
  - Restart transactions that are active at the time of crash



- Redo T2

No-Undo/Redo: Discussion

- Guarantees Atomicity and durability
- Efficient normal operations
- No undo
- Lower degree of concurrency since no updated value of a transaction is visible to other transactions before the transaction commits.
  - All transactions that need to access the result of T have to wait until T completes.
  - System performance may suffer.