Lecture 17: Transaction Recovery

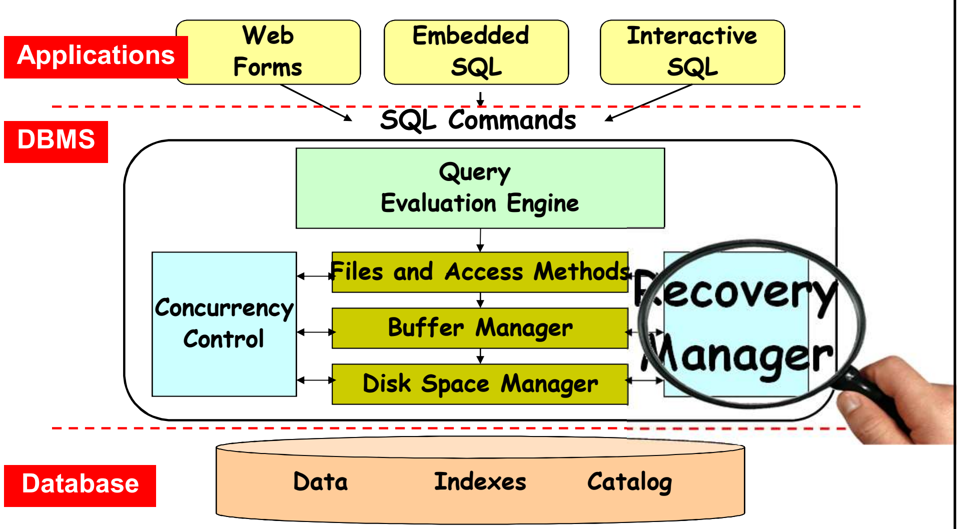
CS 1555: Database Management Systems

Constantinos Costa
http://db.cs.pitt.edu/courses/cs1555/current.term/

April 9, 2019, 16:00-17:15
University of Pittsburgh, Pittsburgh, PA

Lectures based: P. Chrysanthis & N. Farnan Lectures

Database Management System (DBMS)



Many Things Can Go Wrong

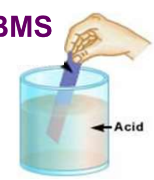
- **Interference with other concurrent activities**

- User may decide to interrupt the program
- disk head crash, system goes down
- Buffer congestion
- Account number does not exist
- Integer overflow
- Error during data transfer
- Power failure

- *Bad data is inserted or good data is deleted*

ACID Properties

DBMS



Property	Dealt with by
A, D	**Recovery Techniques**
I	Concurrency Control Techniques
C	Checks, Assertions,Triggers Applications Programmers

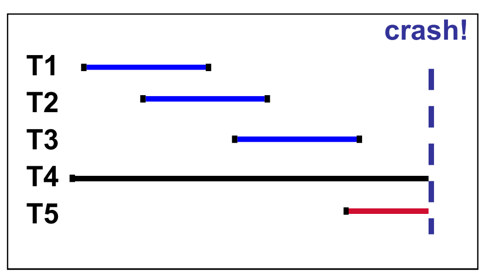
Failure Types

- *Program Failures*
  - logical errors, bad input, unavailable data, user cancellation
  - resource limits
- *System Failures*
  - computer hardware malfunction, power failures
  - bugs in O.S, operator error
- *Media Failures*
  - disk head crash, data transfer error,
  - disk controller failure
- *Unrecoverable errors*
  - failure to make archive dumps
  - destruction of archives

Atomicity & Durability

- Atomicity:
  - Transactions may abort ("Rollback")
- Durability:
  - What if DBMS stops running?

- **Desired Behavior after system restarts:**
  - **T1, T2 & T3 should be durable**
  - **T4 & T5 should be aborted (effects not seen)**



Goal of Recovery

1. When a transaction *T* ***commits***
   - Make the updates permanent in the database so that they can survive subsequent failures.

2. When a transaction *T* ***aborts***
   - Obliterate any updates on data items by aborted transactions in the database.
   - Obliterate the effects of *T* on other transactions; i.e., transactions that read data items updated by *T*.

2. When the system ***crashes*** after a system or media failure
   - Bring the database to its most recent consistent state.

Recovery Actions

- Recovery protocols implement two actions:

  - ***Undo*** action: required for atomicity.

    Undoes all updates on the stable storage by an uncommitted transaction.

  - ***Redo*** action: required for durability

    Redoes the update (on the stable storage) of committed transaction.

Recovering from Failures

- *Program Failures* *Transaction Undo*

  - Removes all the updates of the aborted transaction

  - with Isolation does not affect any other transaction

- *System Failures* *Global Undo*

  Partial Redo

  - Effects of committed transactions are reflected in the database

- *Media Failures* *Global Redo*

Recovery Techniques

1. Undo/Redo Algorithm
   – most commonly used one

2. Undo/No-Redo

3. No-Undo/Redo
   – also called *logging with deferred updates*

1. No-Undo/No-Redo
   – also called *shadowing*

Logging

- A *Log or journal* is a sequence of records which represent all modifications to the database in the order in which they actually occurred
- Log records may describe either *physical* changes or *logical* database operations
  - A *physical log* contains information about the actual values of data items written by transactions.
    - state before change, ***before image***
    - state after change, ***after image***
    - transition causing the change
  - A *logical log* represents higher level operations; e.g., insert this key in an index.

Log Records

- For the moment, we will assume that a log record may be one of the following types:
  - Start Record
    - [T_i, start]
  - Commit Record
    - [T_i, commit]
  - Abort Record
    - [T_i, abort]

Log Records

- Update Record for physical state logging at page level
  - [T_i, x, b, a]
    - T_i : the id of the transaction that performed a Write operation on x
    - x: the id of data item x
    - b: *before* image of x
    - *a: after* image of x
  - Assuming Strict Executions
    [T_j, x, b]: T_j wrote into x before T_i
    [T_i, x, a]

Logical Logging on the Record Level

- Simply record the operation and its arguments
  [T_i, Op, Inv-op, Arg]
  - Op = {Insert, Delete, Update} [REDO]
  - Inv-op = inverse operation [UNDO]
  - Arg = arguments

=> It is not possible in all models to automatically generate the inverse; e.g., the network model.

Update Log Records Structure

o LSN: [T_i, x, b, a, old-LSN(x), prev-LSN(T_i)]

- T_i : the id of the transaction that issued the Write
- **x**: the address of the block being modified and the offset and length
- **b**: the before image of the modified portion of the block
- **a**: the after image of the modified portion of the block
- **old-LSN(x)**: the LSN of x's buffer before this update
- **prev-LSN(T_i)**: the LSN of the preceding log record of this transaction (null if it's the first)

Undoing Writes

UNDO Rule (*WAL, Write Ahead Logging principle*)

T writes x

T aborts or System crash

- If x was transferred to disk, then we need the *before image* of *x* to *undo* this update.

- Thus, when x is updated by *T*, the DM should store first the *before image* of x in the log on stable storage and then x itself in the stable database.

Table for Buffered Log

Disk Id	dirty bit	fix count	Page LSN	Page numbe
x	0	0	812	0
y	1	1	10	1
z	0	1	123	2
⋮	⋮	⋮	⋮	⋮

- The Undo rule is:
  - Before the Buffer/Cache Manager replaces a buffer page it should flush all log entries whose LSN is less than or equal to the LSN recorded on this buffer page

Redoing Writes

REDO Rule

T writes x
T commits
System crash

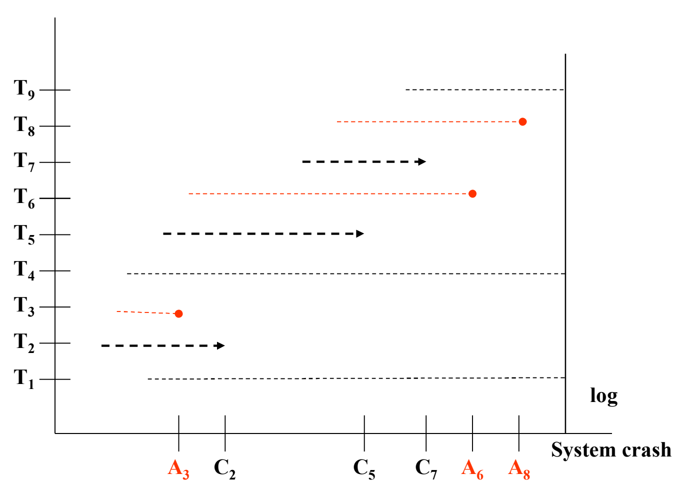
- If *x* was not transferred to disk, at *restart* time we need the *after image* of *x* to redo *T*'s update.

- Thus, the DM should not commit a transaction *T* until the *after image* of each data item written by *T* is in stable storage.

Restarts

- *Restart*: consult the log and for each transaction T_i do the following:
  - **redo** the updates of T_i if there is a commit record of T_i in the log
  - **Undo** the updates of T_i if there is no such record in log, i.e.,
    - T_i had been aborted, or
    - T_i was active when the system crashed

What should Restart do?



Idempotence of Restarts

- The restart operation may be interrupted because of a failure
- Incomplete executions of Restart followed by a completed Restart must have the same effect as just one completed Restart

Garbage Collection

- Recycling space in the log occupied by unnecessary info
- Garbage Collection Rule:

  The entry [T_i ,*x,v*] can be removed from the log *iff*

  1. T_i has aborted
  2. T_i has committed but some other committed transaction wrote into x after T_i did
     - Note that the last committed value of a data item x must be in a log, if undo is possible

  1. [T_i ,*x,v*] can be removed from the log if *v* is the last committed value of x and *v* is the value of x in the stable storage (db) and there are no other entries of x

Checkpoints

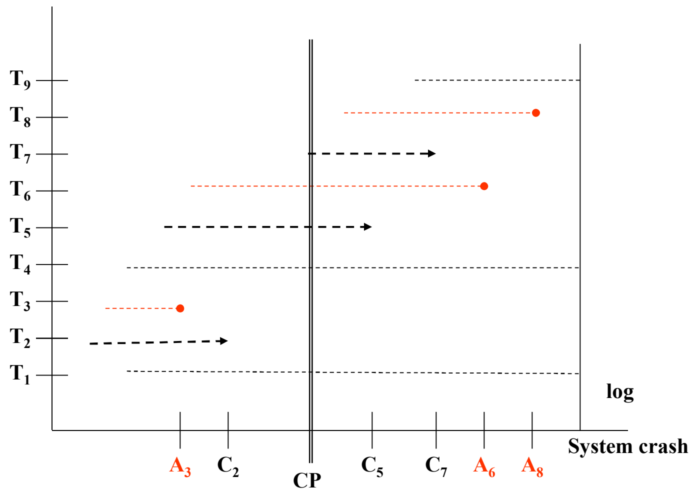
- To Restart, we need to scan the entire log !
  - The Restart operation will be prohibitively slow
  - The Log file may become very long and may not fit on disk
- Most of the transactions that need to be redone have already written their updates to stable database (why?)
  - Thus, most of the Restart operations are unnecessarily performed
- The amount of work Restart has to do after a system failure can be reduced by ***check pointing*** the updates that have been performed up to a certain time

Restart with Checkpointing

- Restart may proceed as before, i.e.:
  - *redo updates* of transactions that have been committed after the checkpoint (why?)
  - *undo updates* of transactions that have not been committed

    Notice: The undo procedure may require reading log records written before the most recent checkpoint point (why?)
- What about *T* that was active when the system crashed but did not perform any Write since the last checkpoint?
  - i.e., there is no record for *T* in the log after the last checkpoint.
- Checkpoint Record includes a list of transactions that were active at checkpoint time: [checkpoint, Ac]

Example: Restart with Checkpoint

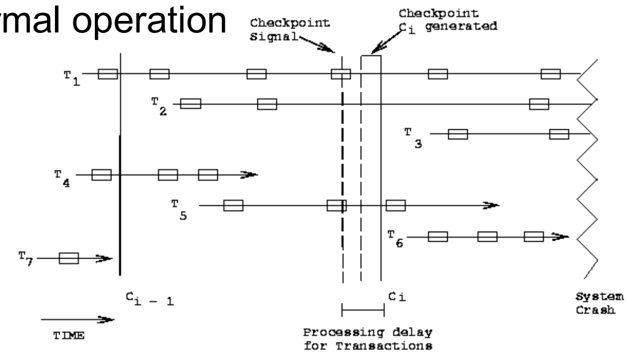


Checkpoint Schemes

- Action-Consistent Checkpoint
- Transaction-Consistent Checkepoint
- Fuzzy Checkpoint

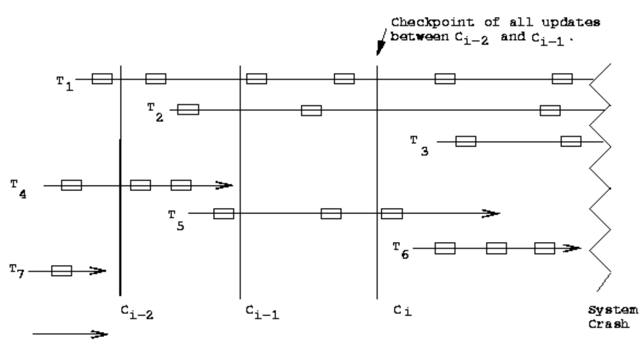
Action Consistent Checkpoint
(Cache Consistent Checkpoint)

1. Stop accepting new *operations*
2. Active transactions are blocked
3. Flush all dirty buffer blocks to disk
4. Force-write a <checkpoint> record to the log file
5. Resume normal operation



Fuzzy Checkpointing

1. Initiate low priority flush of all dirty pages
2. Don't checkpoint again until all of the last checkpoint's dirty pages are flushed
3. Restart begins at second-to-last checkpoint



ARIES [IBM]

- Works in conjunction with inplace updates, WAL, Fuzzy checkpoints
- Novel aspects:
  - Hybrid logging:
    - Page-oriented redo
    - Operation-oriented undo
  - Three passes:
    - *Analysis Pass:* Forward pass from checkpoint till end
    - *Redo Pass:* Repeat history -- reestablish database state as of failure
    - *Undo Pass:* Undo aborted/uncommitted transactions

Media Failure

No surprises here…

- The only hope to implement stable storage is by *data replication*.
  - Number of copies
  - Where these copies are stored ?
- Goal
  - Minimize the probability that all copies will be destroyed
- Two common solutions:
  - Have a second disk (*mirror*) for each used disk
  - Periodically *backup* the db to an *archive db*