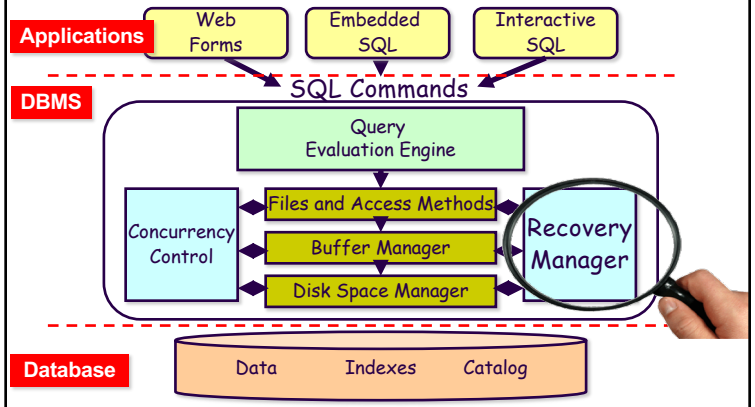
Transaction Recovery

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

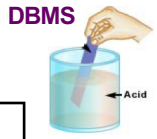
- *Bad data is inserted or good data is deleted*

Many Things Can Go Wrong

- User n
- disk he
- Buffer
- Accou
- Intege
- Error c
- Power
- **Interfe ies**

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

ACID Properties

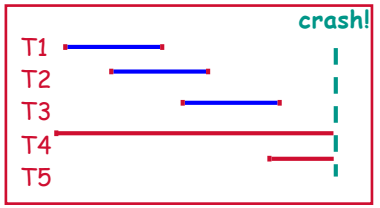


Property	Dealt with by
A, D	**Recovery Techniques**
I	Concurrency Control Techniques
C	Checks, Assertions,Triggers Applications Programmers

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*.
3. 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*
4. 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.

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.

- 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.

Redoing Writes

REDO Rule

T writes x
T commits
System crash

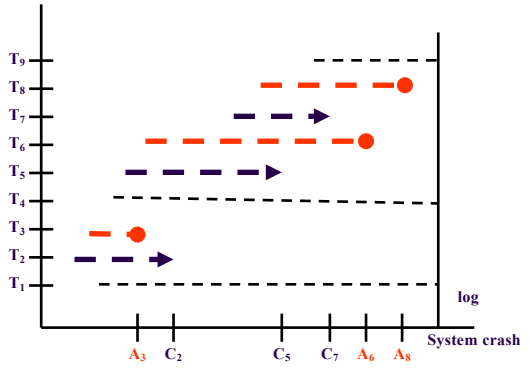
– If *x* was not transferred to disk, at *restart* time we need the *after image* of *x* to redo *T*'s update.

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

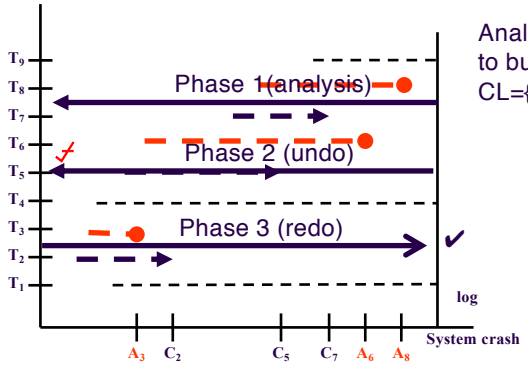
Restarts

- *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?

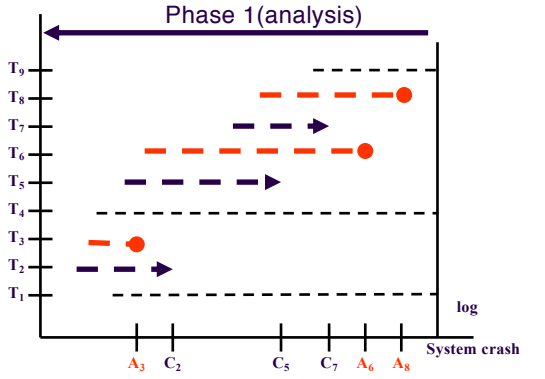


What should Restart do?



Analyze the Log to build the CL={} & AL={}

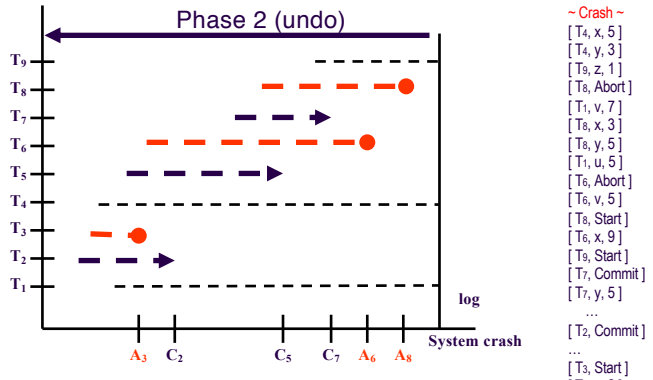
What should Restart do?



~ Crash ~
[T4, x, 5]
[T4, y, 3]
[T9, z, 1]
[T8, Abort]
[T1, v, 7]
[T8, x, 3]
[T8, y, 5]
[T1, u, 5]
[T6, Abort]
[T6, v, 5]
[T8, Start]
[T6, x, 9]
[T9, Start]
[T7, Commit]
[T7, y, 5]
...
[T2, Commit]
...
[T3, Start]
[T5, x, 2]
[T2, Start]

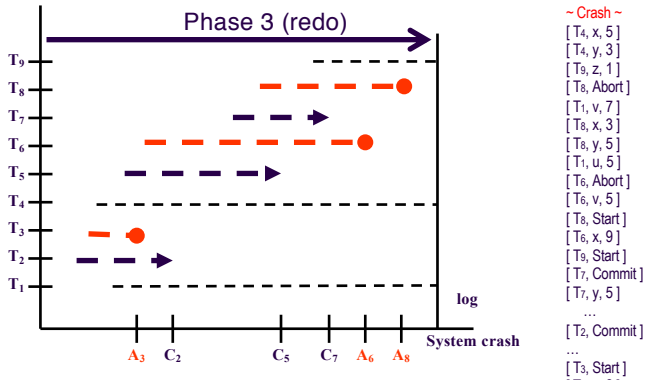
CL={T_7,T_5,T_2} AL={T_4,T_9,T_8,T_1,T_6,T_3}

What should Restart do?



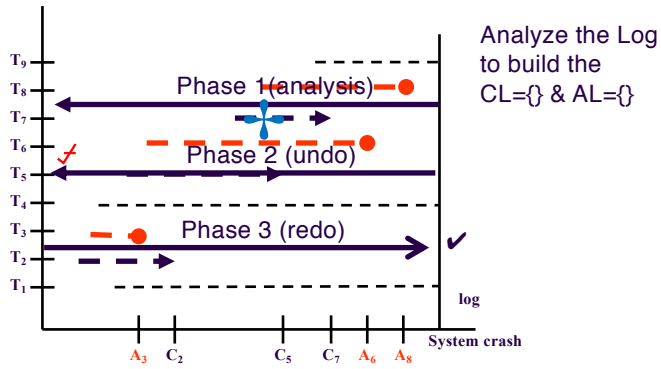
CL={T_7, T_5, T_2} AL={$T_4, T_9, T_8, T_1, T_6, T_3$}

What should Restart do?

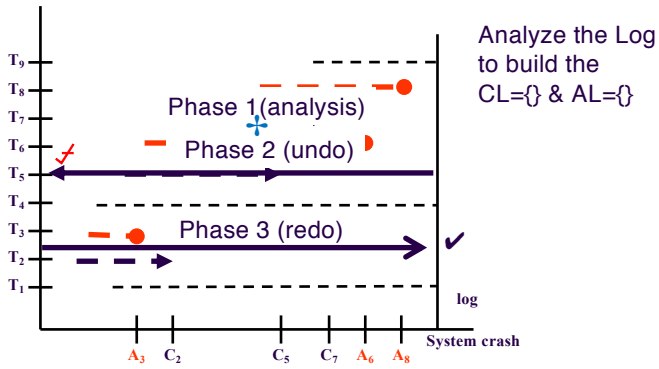


CL={T_7, T_5, T_2} AL={$T_4, T_9, T_8, T_1, T_6, T_3$}

What should Restart do?



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

Cost of Recovery

- 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
- Observation: 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

Garbage Collection & Checkpoints

- Recycling space in the log occupied by unnecessary info
- The amount of work Restart has to do after a system failure can be reduced by ***checkpointing***
  - **Force** the updates that have been performed up to a certain time to materialize in the database
  - Checkpoint Record: include a list of transactions that were active at checkpoint time

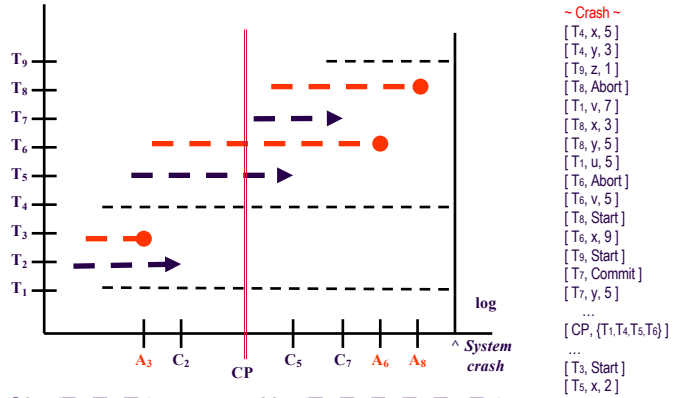
    [checkpoint, Ac]

Restart with Checkpointing

- Restart may proceed as before, i.e.:
  - redo updates of transactions that have been committed, 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?)
- In addition, the following scenario for a transaction *T* is possible:
  - *T* was active when the system crashed.
  - But *T* did not perform any Write operation since the last checkpoint
    - there is no log record for *T* after the last checkpoint

Example: Restart with Checkpoint



CL={$T_7, T_5, \sout{T_2}$} AL={$T_4, T_9, T_8, T_1, T6, \sout{T3}$}

ARIES [IBM]

- Works in conjunction with in-place 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*