# Lecture 11: Concurrency Control & Recovery

Dr. Kyong-Ha Lee
(kyongha@kisti.re.kr)

# Contents

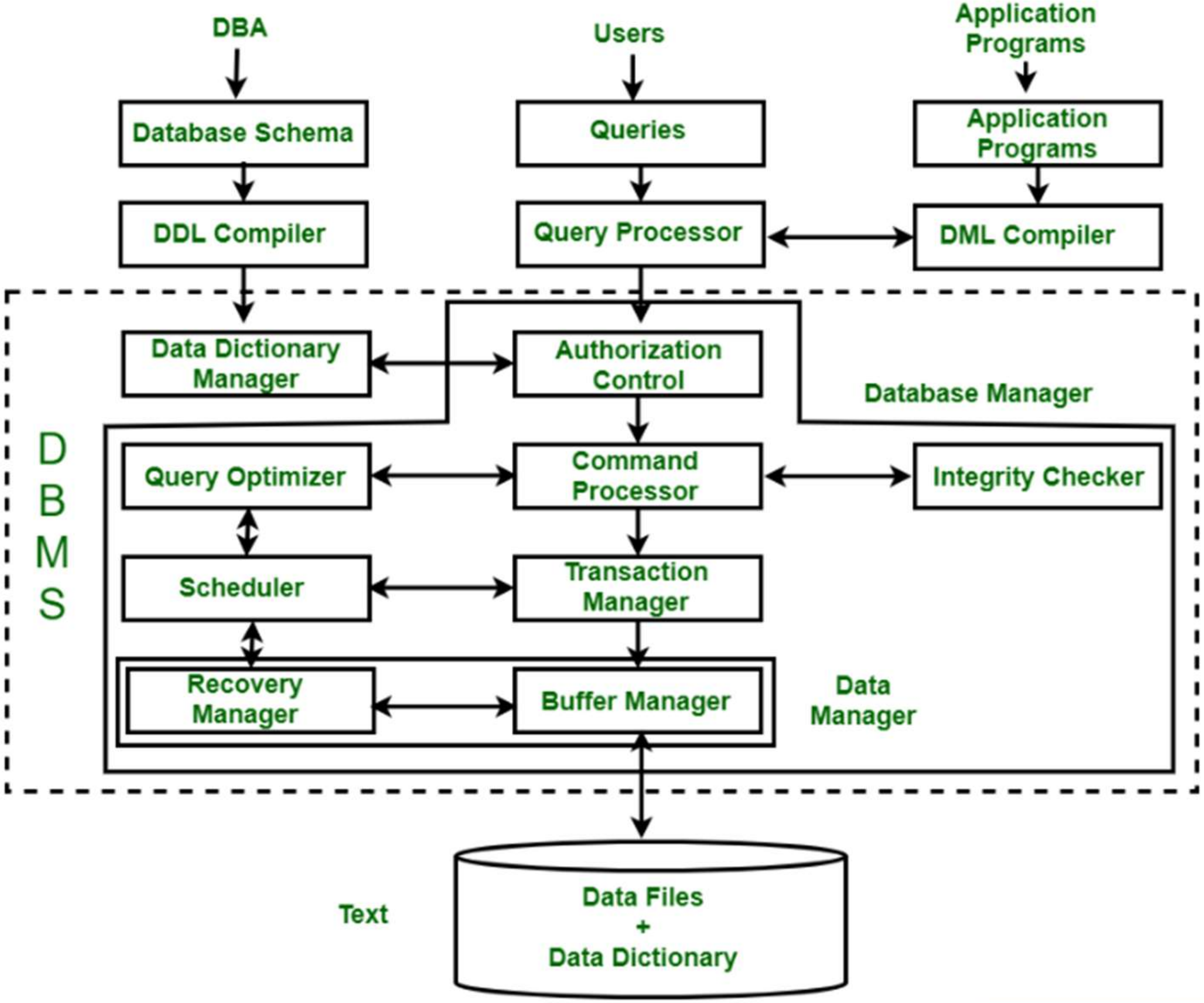
## Brief overview of this lecture

- Basic theories and principles about Index mechanisms used in database systems
- Not much discussion on implementation or tools, but will be happy to discuss them if there are any questions



*Disclaimer: these slides are based on the slides created by the authors of Database System Concepts 7th ed. and modified by K.H. Lee.

# DBMS Architecture OVerview



# Lock-based Protocols

- **Lock**
  - **A mechanism to control concurrent accesses to a data item**
  - Prevents non-serializable schedues from ever being produces
- Data items can be locked in two modes
  - **Execlusive(X) lock**
    - Data item can be both read as well as written by a TX
    - If any TX hods an exclusive lock on the item no other TX may hold any lock on the item
  - **Shared(S) lock**
    - Data item can only be read by a TX
    - Any # of TXs can hold shared locks on an item

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

- Lock requests are made to conccurency control manager
  - TX can proceed only after requirest is granted

# Example of Locking

- Locking itself is not sufficient to guarantee serializability
  - This schedule is not serializable (why?)
- **Locking protocol** is a set of rules followed by all TXs while requesting and releasing locks
  - It enforces serializability by restricting the set of possible schedules

| $T_1$ | $T_2$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| unlock($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | unlock($A$) |
| | lock-S($B$) |
| | read($B$) |
| | unlock($B$) |
| | display($A + B$) |
| lock-X($A$) | |
| read($A$) | |
| $A := A + 50$ | |
| write($A$) | |
| unlock($A$) | |

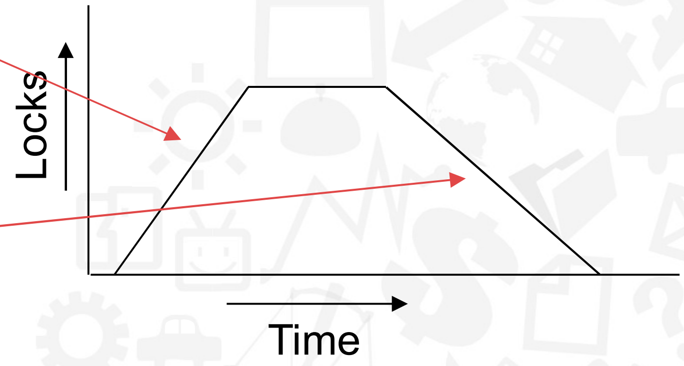
# Deadlock

- Neither $T_3$ nor $T_4$ can make progress
  - executing **lock-S**(*B*) causes $T_4$ to wait for $T_3$ to release its lock on *B*
  - while executing **lock-X**(*A*) causes $T_3$ to wait for $T_4$ to release its lock on *A*.
- Such a situation is called a **deadlock.**
  - To handle a deadlock one of $T_3$ or $T_4$ must be rolled back and its locks released
  - A necessary evil
    - The potential for deadlock exists in most locking protocols
- **Starvation**
  - A TX may be waiting for an X-lock on an item, while a sequence of other TXs are granted an S-lock on the same item
  - The same TX is repeatedly rolled back due to deadlocks

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

# Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks




- The protocol assures serializability
  - It can be proved that the transactions can be serialized in the order of their **lock points**
    - i.e., the point where a transaction acquired its final lock
  - ***Sufficient but not necessary***

# Two-Phase Locking Protocol(Cont.)

- It does NOT ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking**
    - A TX must hold all its exclusive locks till it commits/aborts
    - It ensures recoverability and avoids cascading rollbacks
  - **Rigorous two-phase locking**
    - TX can be serialized in the order in which they commit

- Note:
  - Most DBMSes implement rigorous two-phase locking, but refer to it as simply two-phase locking

# Two-Phase Locking Protocol(Cont.)

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable
- 2PL protocol with lock conversions
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)

| $T_1$ | $T_2$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| unlock($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | unlock($A$) |
| | lock-S($B$) |
| | read($B$) |
| | unlock($B$) |
| | display($A + B$) |
| lock-X($A$) | |
| read($A$) | |
| $A := A + 50$ | |
| write($A$) | |
| unlock($A$) | |

# Deadlock handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

- Solution
  - Deadlock prevention
    - Ensures that the system will never be deadlocked
  - Deadlock detection& recovery

# Deadlock Prevention

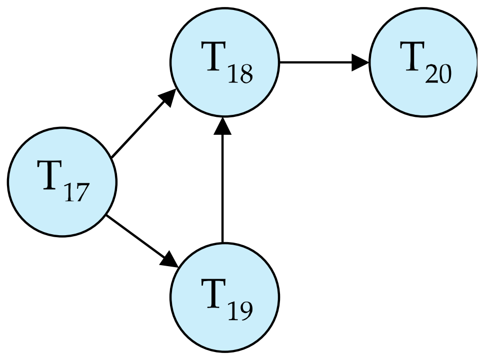
- Pre-declaration
  - Each TX locks all its data items before it begins execution
- Graph-based protocol
  - Impose partial ordering of all data items
  - TX can lock data items only in the order specified by the partial order

# More deadlock prevention strategies

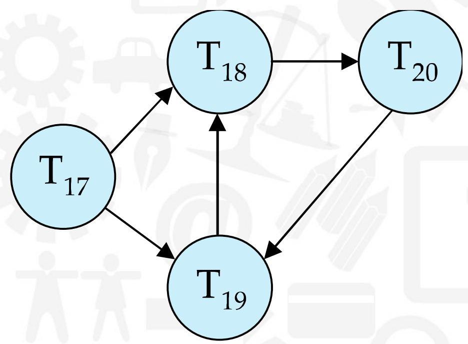
- Wait-die scheme (non-preemptive)
  - Older TX may wait for younger one to release data item
  - Younger TX never wait for older ones; they are rolled back instead
- Wound-wait scheme(preemptive)
  - Older TX forces wounds (forces rollback) of younger TX instead of waiting for it
  - Younger TX may wait for older ones
  - Fewer rollbacks than wait-die scheme
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

# Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if $T_i$ is waiting for a lock held in conflicting mode by $T_j$
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.




Wait-for graph without a cycle



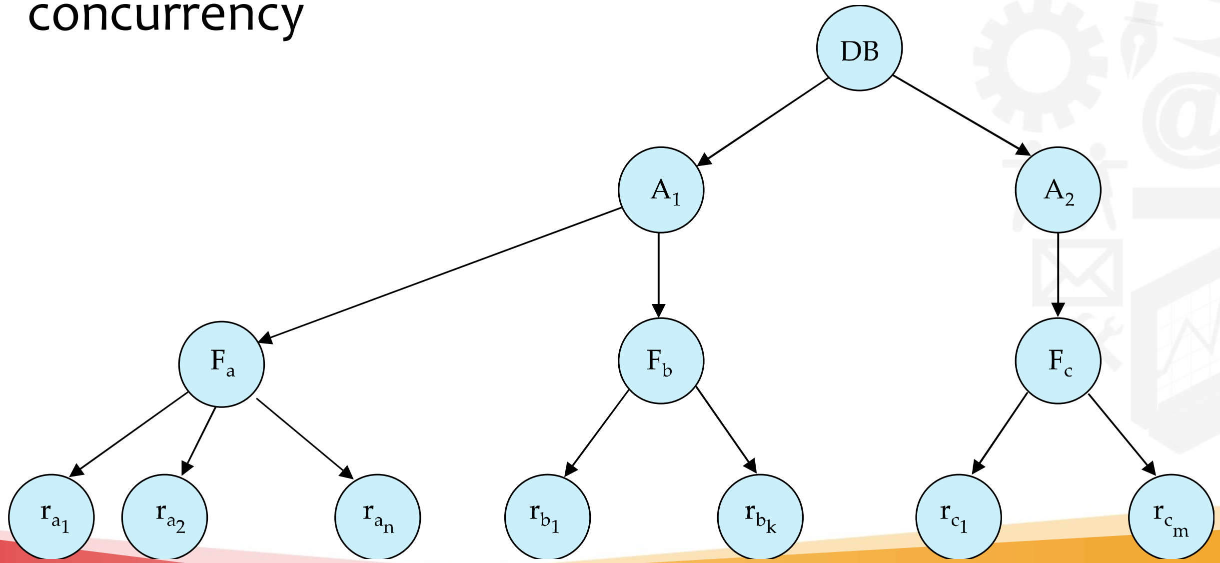

Wait-for graph with a cycle

# Deadlock Recovery

- When deadlock is  detected :
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

# Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities
- Can be represented graphically as a tree
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency




- *database*
- *area*
- *file*
- *record*

# Intention Lock Modes

- 3 additional lock modes with multiple granularity
  – ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks
  – ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  – ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | true | true | true | true | false |
| IX | true | true | false | false | false |
| S | true | false | true | false | false |
| SIX | true | false | false | false | false |
| X | false | false | false | false | false |

compatibility matrix

# Multiple Granularity Locking Scheme

- Transaction $T_i$ can lock a node $Q$, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node $Q$ can be locked by $T_i$ in S or IS mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or IS mode.
  4. A node $Q$ can be locked by $T_i$ in X, SIX, or IX mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or SIX mode.
  5. $T_i$ can lock a node only if it has not previously unlocked any node (that is, $T_i$ is two-phase).
  6. $T_i$ can unlock a node $Q$ only if none of the children of $Q$ are currently locked by $T_i$.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

# Timestamp-Based Protocols

- Each transaction $T_i$ is issued a timestamp TS($T_i$) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use (wall-clock time, logical counter) to ensure
- Timestamp-based protocols manage concurrent execution such that

  **time-stamp order = serializability order**
- Several alternative protocols based on timestamps

# Timestamp-Based Protocols(Cont.)

The **timestamp ordering (TSO) protocol**

- Maintains for each data $Q$ two timestamp values:
  - **W-timestamp**($Q$) is the largest time-stamp of any transaction that executed **write**($Q$) successfully.
  - **R-timestamp**($Q$) is the largest time-stamp of any transaction that executed **read**($Q$) successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

# Timestamp-Based Protocols(Cont.)

- Suppose a transaction $T_i$ issues a **read**($Q$)

1. If TS($T_i$) < **W**-timestamp($Q$),
   then $T_i$ needs to read a value of $Q$ that was already over-written
   - Hence, the **read** operation is rejected, and $T_i$ is rolled back.
2. If TS($T_i$) ≥ **W**-timestamp($Q$),
   then the **read** operation is executed, and R-timestamp($Q$) is set to **max**(R-timestamp($Q$), TS($T_i$)).

# Timestamp-Based Protocols(Cont.)

- Suppose that transaction $T_i$ issues **write**($Q$)
  1. If TS($T_i$) < R-timestamp($Q$)
     then the **write** operation is rejected, and $T_i$ is rolled back.
  2. If TS($T_i$) < W-timestamp($Q$)
     then $T_i$ is attempting to write an obsolete value of $Q$
     - Hence, this **write** operation is rejected, and $T_i$ is rolled back.
  3. Otherwise
     the **write** operation is executed, and W-timestamp($Q$) is set to TS($T_i$).

# Example of Schedule Under TSO

Is this schedule valid under TSO?

Assume that initially:
R-TS(A) = W-TS(A) = 0
R-TS(B) = W-TS(B) = 0
Assume TS($T_{25}$) = 25 and
TS($T_{26}$) = 26

| $T_{25}$ | $T_{26}$ |
|---|---|
| read($B$) | |
| | read($B$) |
| | $B := B - 50$ |
| | write($B$) |
| read($A$) | |
| | read($A$) |
| display($A + B$) | |
| | $A := A + 50$ |
| | write($A$) |
| | display($A + B$) |

- How about this one, where initially
  R-TS(Q)=W-TS(Q)=0

| $T_{27}$ | $T_{28}$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

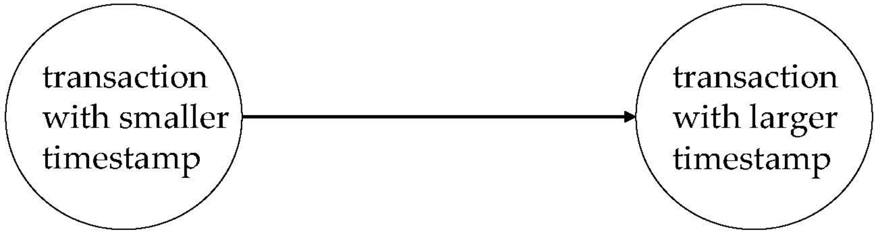
# Another Example Under TSO

- A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | read ($X$) |
| | read ($Y$) | | | |
| read ($Y$) | | | | |
| | | write ($Y$) | | |
| | | write ($Z$) | | |
| | | | | read ($Z$) |
| | read ($Z$) | | | |
| | abort | | | |
| read ($X$) | | | | |
| | | | read ($W$) | |
| | | write ($W$) | | |
| | | abort | | |
| | | | | write ($Y$) |
| | | | | write ($Z$) |

# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:




  Thus, there will be no cycles in the precedence graph
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may  not even be recoverable.

# Multiversion Concurrency Control(MVCC)

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** creates **a new version of the data item** written.
  - **Use timestamps to label versions**
  - When a **read**(Q) operation is issued, **select an appropriate version of Q based on the timestamp of the transaction** issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately

# Multiversion Concurrency Control

- Each data item $Q$ has a sequence of versions <$Q_1$, $Q_2$,...., $Q_m$> and each version $Q_k$ contains three data fields:
  - **Content** -- the value of version $Q_k$.
  - **W-timestamp**($Q_k$) -- timestamp of the transaction that created (wrote) version $Q_k$
  - **R-timestamp**($Q_k$) -- largest timestamp of a transaction that successfully read version $Q_k$
- Suppose that transaction $T_i$ issues a **read**($Q$) or **write**($Q$) operation. Let $Q_k$ denote the version of $Q$ whose write timestamp is the largest write timestamp less than or equal to TS($T_i$).
  1. If transaction $T_i$ issues a **read**($Q$), then
     - the value returned is the content of version $Q_k$
     - If R-timestamp($Q_k$) < TS($T_i$), set R-timestamp($Q_k$) = TS($T_i$),
  2. If transaction $T_i$ issues a **write**($Q$)
     1. if TS($T_i$) < R-timestamp($Q_k$), then transaction $T_i$ is rolled back.
     2. if TS($T_i$) = W-timestamp($Q_k$), the contents of $Q_k$ are overwritten
     3. Otherwise, a new version $Q_i$ of $Q$ is created
        - W-timestamp($Q_i$) and R-timestamp($Q_i$) are initialized to TS($T_i$).

# Multiversion Timestamp Ordering (Cont)

- Observations
  - Reads always succeed
  - A write by $T_i$ is rejected if some other transaction $T_j$ that (in the serialization order defined by the timestamp values) should read $T_i$'s write, has already read a version created by a transaction older than $T_i$.
- Protocol guarantees serializability

# Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution : **Give snapshot of database state to every transaction**
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level

# Snapshot Isolation

- A transaction T1:
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent TXs are not visible to T1
  - Writes of T1 complete when it commits
  - First-committer-wins rule
    - Commits only if no other concurrent TX has already written data that T1 intends to write

| T1 | T2 | T3 |
|---|---|---|
| W(Y := 1)<br>Commit | | |
| | Start<br>R(X) → 0<br>R(Y)→ 1 | |
| | | W(X:=2)<br>W(Z:=3)<br>Commit |
| | R(Z) → 0<br>R(Y) → 1<br>W(X:=3)<br>Commit-Req<br>Abort | |

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

# Snapshot Read & Write

- Concurrent updates invisible to snapshot read
- First Committer Wins

$X_0 = 100, Y_0 = 0$

| $T_1$ deposits 50 in $Y$ | $T_2$ withdraws 50 from $X$ |
|---|---|
| $r_1(X_0, 100)$ | |
| $r_1(Y_0, 0)$ | |
| | $r_2(Y_0, 0)$ |
| | $r_2(X_0, 100)$ |
| | $w_2(X_2, 50)$ |
| $w_1(Y_1, 50)$ | |
| $r_1(X_0, 100)$ (update by $T_2$ not seen) | |
| $r_1(Y_1, 50)$ (can see its own updates) | |
| | $r_2(Y_0, 0)$ (update by $T_1$ not seen) |

$X_2 = 50, Y_1 = 50$

$X_0 = 100$

| $T_1$ deposits 50 in $X$ | $T_2$ withdraws 50 from $X$ |
|---|---|
| $r_1(X_0, 100)$ | |
| | $r_2(X_0, 100)$ |
| | $w_2(X_2, 50)$ |
| $w_1(X_1, 150)$ | |
| $commit_1$ | |
| | $commit_2$ (Serialization Error $T_2$ is rolled back) |

$X_1 = 150$

# Pros and COns

- Reads are *never* blocked,
  - and also don't block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - I.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

# Recovery System

# Failure Classification

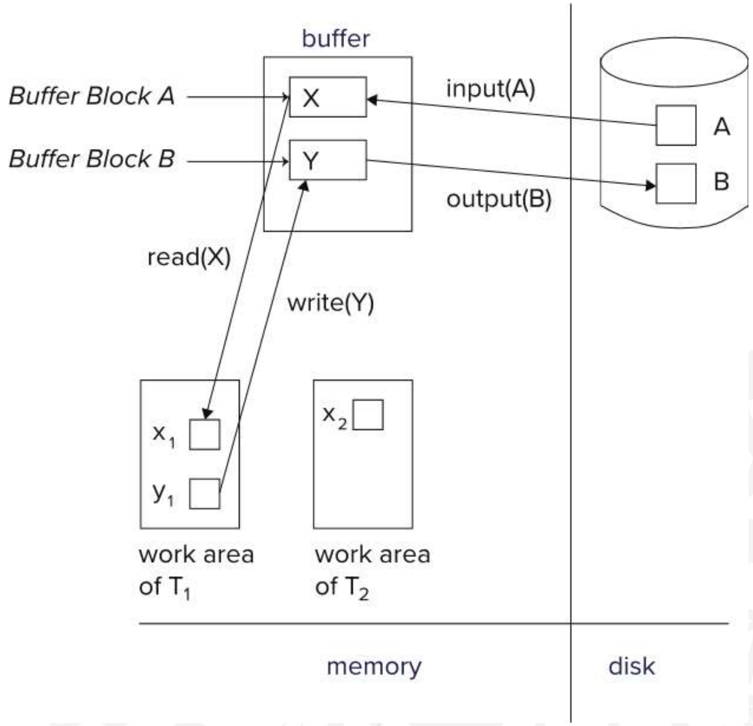
- **Transaction failure :**
  - **Logical errors:** transaction cannot complete due to some internal error condition
  - **System errors:** the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash:** a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption:** non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure:** a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

# Data Accesses

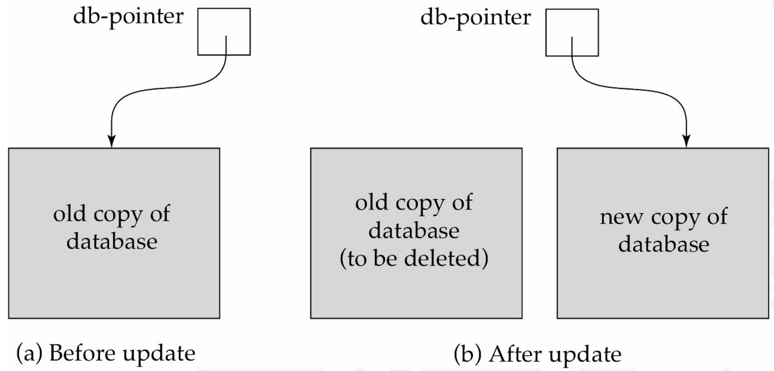
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage

  ➔ **Log**

- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging**




**Example of DA**




**shadow-copy**

# Log-Based Recovery

- A **log** is a sequence of **log records.** The records keep information about update activities on the database.
  - The **log** is kept on stable storage
- When transaction $T_i$ starts, it registers itself by writing a
  <$T_i$ **start**> log record
- *Before* $T_i$ executes **write**($X$), a log record
  <$T_i$, $X$, $V_1$, $V_2$>
  is written, where $V_1$ is the value of $X$ before the write (the **old value**), and $V_2$ is the value to be written to $X$ (the **new value**).
- When $T_i$ finishes it last statement, the log record <$T_i$ **commit**> is written.
- Two approaches using logs
  - Immediate database modification
  - Deferred database modification.

# Immediate Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
- Output of updated blocks to disk can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
  - Simplifies some aspects of recovery
  - But has overhead of storing local copy

# Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
  - All previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

# Immediate Modification Example

| Log | Write | Output |
|---|---|---|
| <$T_0$ **start**> | | |
| <$T_0$, A, 1000, 950> | | |
| <$T_0$, B, 2000, 2050> | | |
| | $A$ = 950 | |
| | $B$ = 2050 | |
| <$T_0$ **commit**> | | |
| <$T_1$ **start**> | | |
| <$T_1$, C, 700, 600> | | |
| | $C$ = 600 | |
| | | $B_B$ , $B_C$ |
| <$T_1$ **commit**> | | |
| | | $B_A$ |

$B_C$ output before $T_1$ commits

$B_A$ output after $T_0$ commits

- Note: $B_X$ denotes block containing $X$.

# Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction $T_i$ has modified an item, no other transaction can modify the same item until $T_i$ has committed or aborted*
  - i.e., the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise, how to perform undo if $T_1$ updates A, then $T_2$ updates A and commits, and finally $T_1$ has to abort?
  - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- **Log records of different transactions may be interspersed** in the log.

# Undo and Redo operations

- **undo**($T_i$)
  - restores the value of all data items updated by $T_i$ to their old values, going backwards from the last log record for $T_i$
    - Each time a data item X is restored to its old value V a special log record <$T_i$ , $X$, $V$> is written out
    - When undo of a transaction is completed, a log record <$T_i$ **abort**> is written out.
- **redo**($T_i$)
  - sets the value of all data items updated by $T_i$ to the new values, going forward from the first log record for $T_i$
    - No logging is done in this case

# Recovering from Failure

- When recovering after failure:
  - Transaction $T_i$ needs to be undone if the log
    - Contains the record <$T_i$ **start**>,
    - But does not contain either the record <$T_i$ **commit**> *or* <$T_i$ **abort**>.
  - Transaction $T_i$ needs to be redone if the log
    - Contains the records <$T_i$ **start**>
    - And contains the record <$T_i$ **commit**> *or* <$T_i$ **abort**>
- If transaction $T_i$ was undone earlier and the <$T_i$ **abort**> record was written to the log, and then a failure occurs,
  - On recovery from failure, transaction $T_i$ is redone
  - Such a **redo** redoes all the original actions of transaction $T_i$ *including the steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

# Immediate DB Modification Recovery Example

| (a) | (b) | (c) |
|---|---|---|
| <$T_0$ start> | <$T_0$ start> | <$T_0$ start> |
| <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> | <$T_0$, $A$, 1000, 950> |
| <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> | <$T_0$, $B$, 2000, 2050> |
| | <$T_0$ commit> | <$T_0$ commit> |
| | <$T_1$ start> | <$T_1$ start> |
| | <$T_1$, $C$, 700, 600> | <$T_1$, $C$, 700, 600> |
| | | <$T_1$ commit> |

(a) undo ($T_0$): B is restored to 2000 and A to 1000, and log records <$T_0$, B, 2000>, <$T_0$, A, 1000>, <$T_0$, **abort**> are written out

(b) redo ($T_0$) and undo ($T_1$): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <$T_1$, C, 700>, <$T_1$, **abort**> are written out.

(c) redo ($T_0$) and redo ($T_1$): A and B are set to 950 and 2050 respectively. Then C is set to 600

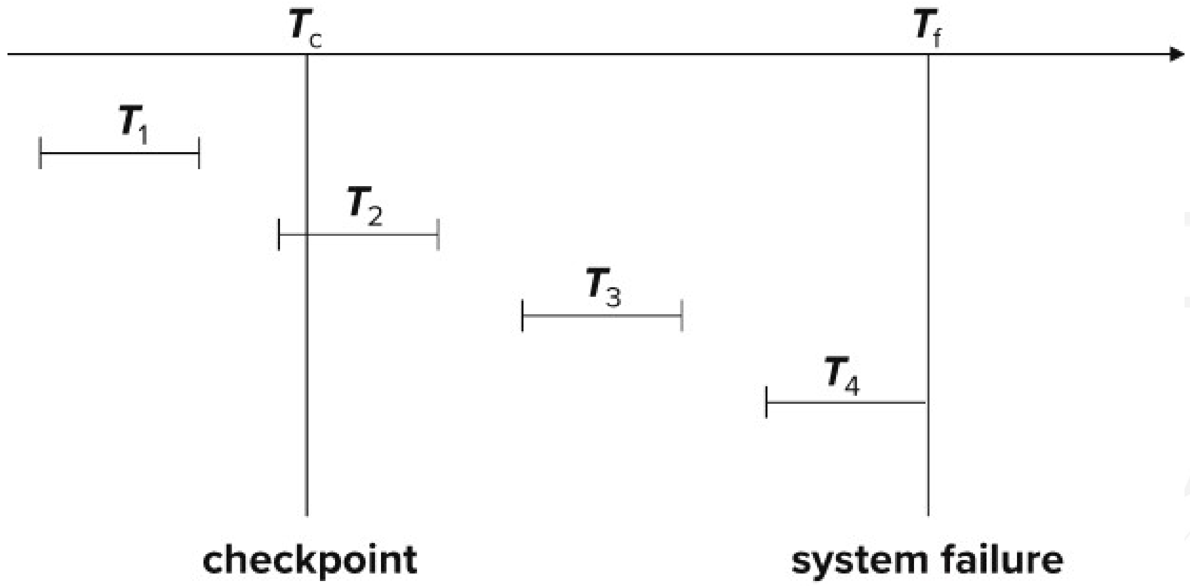
# Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  - Processing the entire log is time-consuming if the system has run for a long time
  - We might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
  4. All updates are stopped while doing checkpointing

# Checkpoints (Cont.)

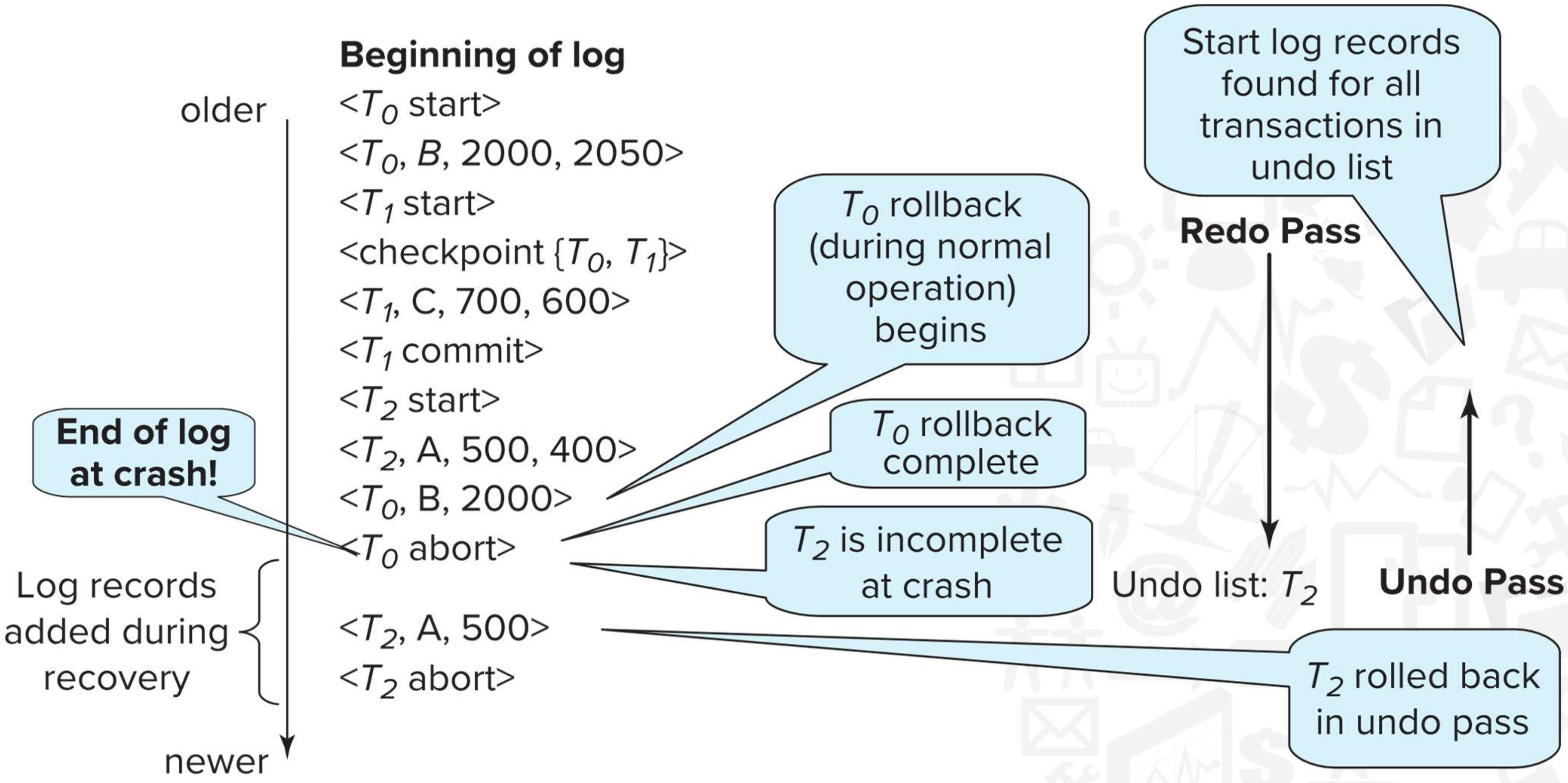
- During recovery we need to consider only the most recent transaction $T_i$ that started before the checkpoint, and transactions that started after $T_i$.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <$T_i$ **start**> is found for every transaction $T_i$ in *L*.
  - Parts of log prior to earliest <$T_i$ **start**> record above are not needed for recovery, and can be erased whenever desired.

# Example of Checkpoints




- $T_1$ can be ignored (updates already output to disk due to checkpoint)
- $T_2$ and $T_3$ redone.
- $T_4$ undone

# Recovery Example

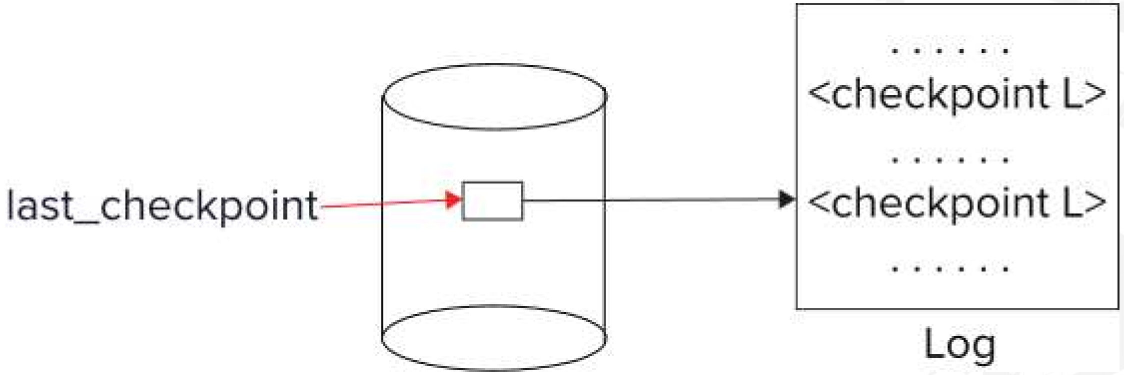


# Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** *L*> log record and force log to stable storage
  3. Note list *M* of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list *M*
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

# Fuzzy Checkpointing (Cont.)

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



# Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

# Question?

–source: https://www.fox.com/the-simpsons