Part 16 :

Concurrency Control and Recovery

Database System Concepts, 7th Ed.

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes:
  1. **exclusive** (X) mode. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction
  2. **shared** (S) mode. Data item can only be read. S-lock is requested using **lock-S** instruction
- Exclusive lock is also called **write lock**, and shared lock is also called **read lock**
- Lock requests are made to the lock manager. Transaction can proceed only after request is granted

Lock-Based Protocols

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks
- Locking protocols enforce serializability by restricting the set of possible schedules
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item

Deadlock

- Consider the partial schedule

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)	

- 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

Deadlock

- The potential for deadlock exists in most locking protocols
- **Starvation** occurs when a transaction cannot complete its task for an indefinite period of time while other transactions continue
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item
  - The same transaction is repeatedly rolled back due to deadlocks
- Concurrency control manager can be designed to prevent starvation

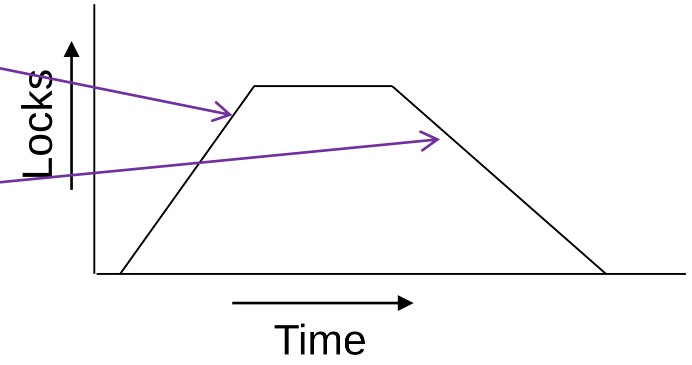
Deadlock Prevention

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state, e.g.,

- Require that each transaction locks all its data items before it begins execution (pre-declaration)
- A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back

The 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)



Lock Conversions

- Two-phase locking 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**)
- This protocol ensures serializability

Recovery Algorithms

- Suppose transaction T_i transfers \$50 from account *A* to account *B*
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database
  - A failure may occur after one of these modifications have been made but before both of them are made
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents

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
- **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

Storage Structure

- **Volatile storage**:
  - Does not survive system crashes
  - Examples: main memory, cache memory
- **Non-volatile storage**:
  - Survives system crashes
  - Examples: disk, tape, flash memory, non-volatile RAM
  - But may still fail, losing data
- **Stable storage**:
  - A mythical form of storage that survives all failures
  - Approximated by maintaining multiple copies on distinct nonvolatile media

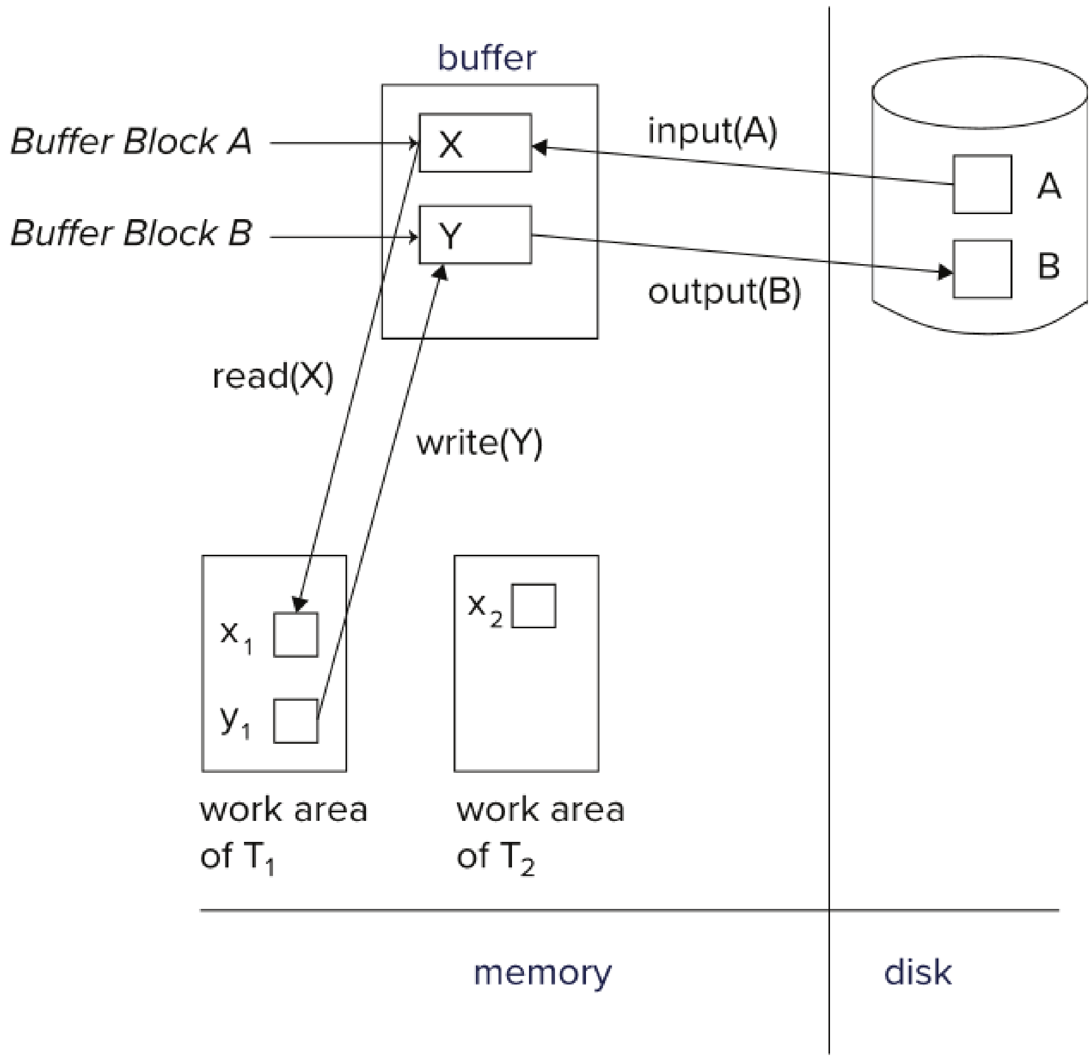
Data Access

- **Physical blocks** are those blocks residing on the disk
- **Buffer blocks** are the blocks residing temporarily in main memory
- Block movements between disk and main memory are initiated through the following two operations:
  - **input** (*B*) transfers the physical block *B* to main memory
  - **output** (*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there
- We assume that each data item fits in, and is stored inside, a single block

Data Access

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i
  - **write**(X) assigns the value of local variable x_i to data item X in the buffer block
  - Note: **output**(B_X) need not immediately follow **write**(X)
    - System can perform the **output** operation when it deems fit
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time

Example of Data Access



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** or **before image BFIM**)**,** and V_2 is the value to be written to X (the **new value** or **after image AFIM**).
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.

Database Modification

- We say a transaction *modifies the database* if it performs an update on a disk buffer or on the disk itself
  - Updates to the private part of main memory do not count as database 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
  - 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
- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit

Commit Point

- A transaction is said to reach its **commit point** when the effect of all the transaction operations on the database have been output to the log
- A transaction is said to have committed (beyond the commit point) when **its commit log record is output to stable storage**
  - All previous log records of the transaction must also have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Database 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

Undo and Redo Operations

- **Undo and Redo of Transactions**
  - **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 complete, 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

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 for all completed transactions if the log
    - Contains the records <T_i **start**>
    - And contains the record <T_i **commit**> *or* <T_i **abort**>
- Suppose that failed transaction T_i was undone before system failure and the <T_i **abort**> record was written to the log, and then a system failure occurs
- On recovery from system failure, transaction T_i is redone
  - Such a **redo** redoes all the original actions of transaction T_i *including the (roll back) steps that restored old values*
    - Known as **repeating history**
    - Seems wasteful in making all the changes and then rolling them back; repeating history for failed transactions simplifies recovery

Recovery Example

Below we show the log as it appears at three instances of time preceding a failure.

(a)	(b)	(c)
$< T_0 \text{ start} >$	$< T_0 \text{ start} >$	$< T_0 \text{ start} >$
$< T_0, A, 1000, 950 >$	$< T_0, A, 1000, 950 >$	$< T_0, A, 1000, 950 >$
$< T_0, B, 2000, 2050 >$	$< T_0, B, 2000, 2050 >$	$< T_0, B, 2000, 2050 >$
	$< T_0 \text{ commit} >$	$< T_0 \text{ commit} >$
	$< T_1 \text{ start} >$	$< T_1 \text{ start} >$
	$< T_1, C, 700, 600 >$	$< T_1, C, 700, 600 >$
		$< T_1 \text{ commit} >$

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively, and *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**, during which
  1. All updates are stopped while the checkpoint operation is in progress
  2. Output all log records currently residing in main memory onto stable storage
  3. Output all modified buffer blocks to the disk
  4. Output a log record < **checkpoint** *L*> onto stable storage, where *L* is a list of all transactions active at the time of checkpoint

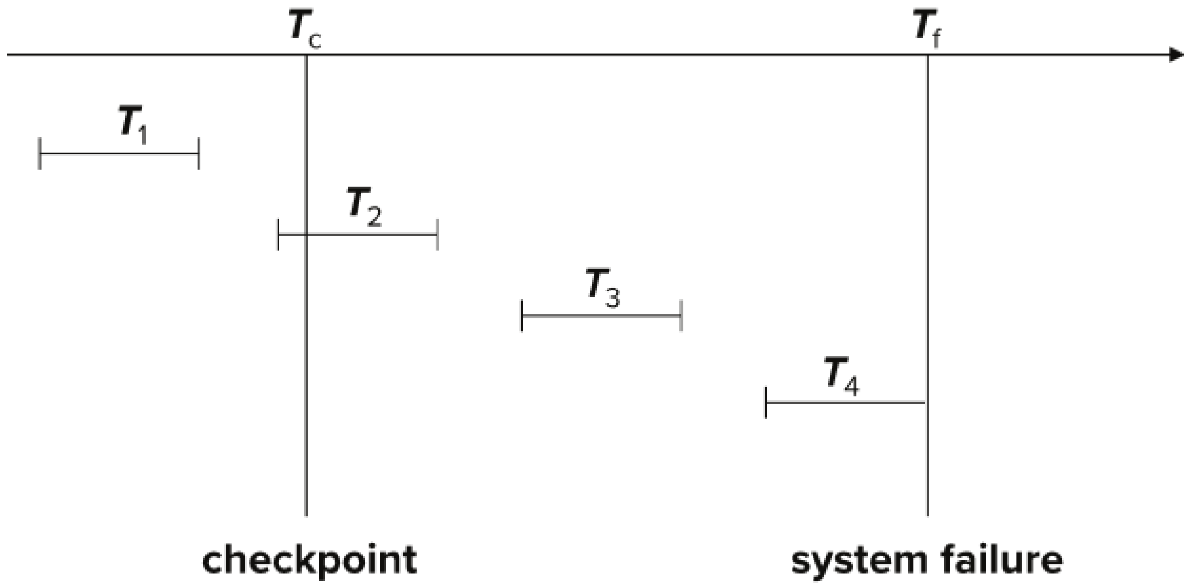
Checkpoints

- Consider a transaction T_i that completed prior to the checkpoint
- For such a transaction, the <T_i **commit**> record (or <T_i **abort**>) record appears in the log before the < **checkpoint** L> record
- Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself
- Thus, at recovery time, there is no need to perform a **redo** operation on T_i

Checkpoints

- During recovery
  - 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
- The redo or undo operations need to be applied only to transactions in *L*, and to all transactions that started execution after the <checkpoint *L*> record was written to the log. Denote this set of transactions as *T*.
  - For all transactions T_k in *T* that have no <T_k commit> record or <T_k abort> record in the log, execute undo(T_k)
  - For all transactions T_k in *T* such that either the record <T_k commit> or the record <T_k abort> appears in the log, execute redo(T_k)

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 Algorithm

- **Logging** (during normal operation):
  - <T_i **start**> at transaction start
  - <T_i, X_j, V_1, V_2> for each update, and
  - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
    - Perform the undo by writing V_1 to X_j,
    - Write a log record <T_i , X_j, V_1>
      - such log records are called **compensation log records**
  - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>

Log Record Buffering

- **Log record buffering**: log records are sometimes buffered in main memory, instead of being output directly to stable storage
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed
- Log force is performed to commit a transaction by forcing (force-write) all its log records (including the commit record) to stable storage

Write-Ahead Logging

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created
  - Transaction T_i enters the commit state only when the log record
    <T_i **commit**> has been output to stable storage
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage
    - This rule is called the **write-ahead logging** or **WAL** rule

Shadow Paging

- Shadow paging considers the database to be made up of *n* fixed-size disk pages
  - Directory with *n* entries is constructed
  - When transaction begins executing, directory copied into shadow directory to save while the current (new) directory is being used
  - Shadow directory is never modified
- During transaction execution
  - New copy of the modified page is created and stored elsewhere
  - Current directory modified to point to new disk block
  - Shadow directory still points to old disk block
- Committing a transaction
  - Switching from the shadow directory to the current directory
- Failure recovery
  - Reinstate shadow directory

Shadow Paging Example