Lecture 14
Recovery System

Shuigeng Zhou

June 18, 2014

School of Computer Science
Fudan University

Outline

- Database failures
- Data access model
- Log-based recovery
  - Serial schedule
  - Checkpointing
  - Concurrent schedule

Database Failures

- Transaction failure
  - Logical errors
  - System errors
- System crash
  - A power failure or other hardware or software failure causes the system to crash
- Disk failure
  - A head crash or similar disk failure destroys all or part of disk storage

Recovery Algorithms

- Techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing
  2. Actions taken after a failure

Data Access

Storage Structure

- **Volatile storage**
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**
  - survives system crashes
  - examples: disk, tape, flash memory
- **Stable storage**
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

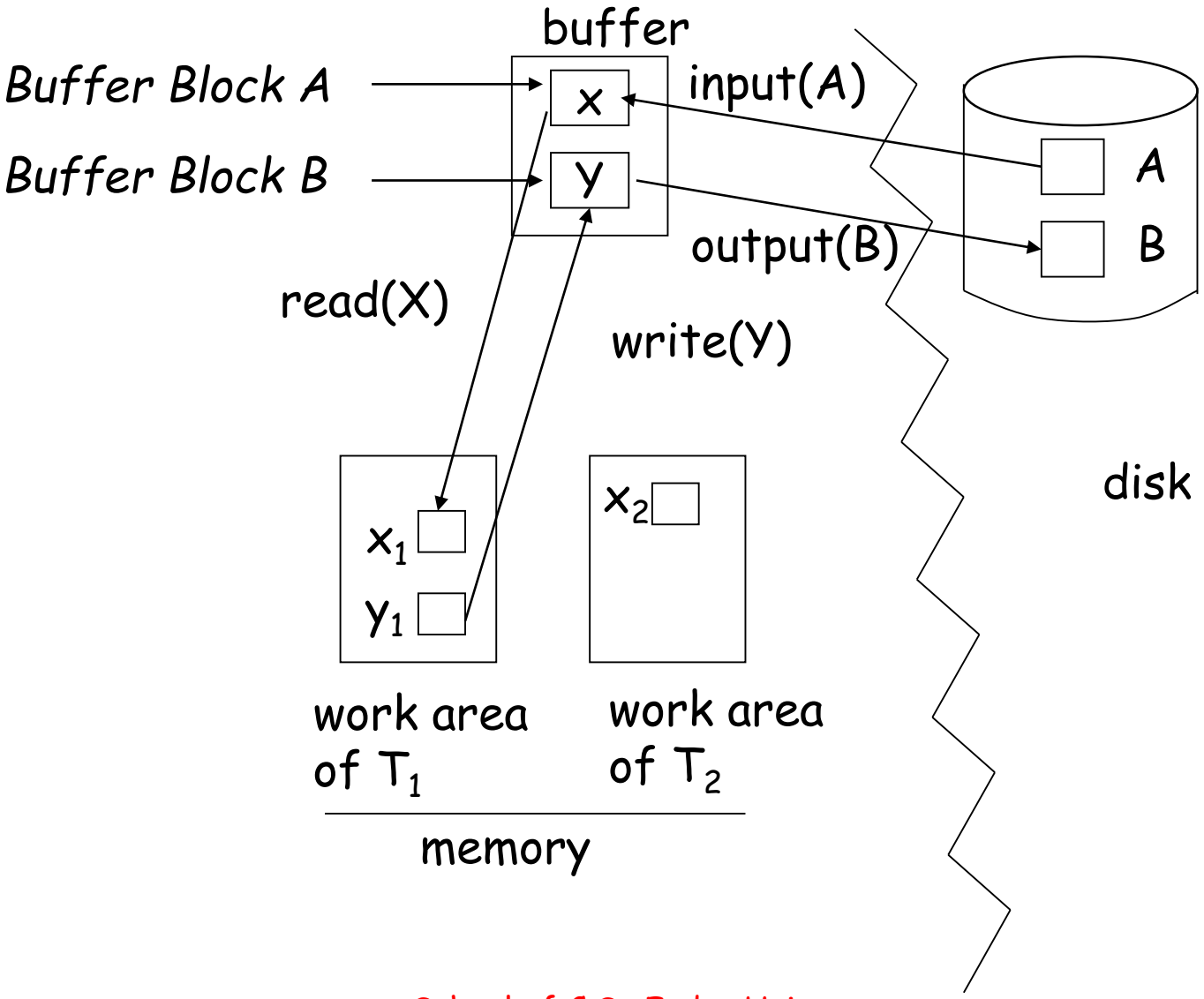
Data Access

- **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 :
  - **input**(*B*)
  - **output**(*B*)
- Each transaction T_i has its private work-area
  - T_i's local copy of a data item *X* is called x_i.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using
  - **read**(X)
  - **write**(X)
- Transactions
  - Perform **read**(X) while accessing X for the first time
  - All subsequent accesses are to the local copy
  - After last access, transaction executes **write**(X)
- **output**(B_X) need not immediately follow **write**(X)
  - System can perform the **output** operation when it deems fit

Example of Data Access



Recovery and Atomicity

- Modifying the database (without ensuring that the transaction will commit) may leave the database in an inconsistent state.
  - Consider transaction T_i that transfers \$50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
  - Several output operations may be required for T_i (to output *A* and *B*).
  - A failure may occur after one of these modifications have been made but before all of them are made.

Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- There are two approaches:
  - **log-based recovery (基于日志的恢复)**
  - **shadow-paging (影子页)**
- We assume (initially) that transactions run serially, that is, one after the other

Log-based Recovery

Log-Based Recovery

- A **log** is kept on stable storage
  - The log is a sequence of **log records**
  - We assume that log records are not buffered.
- When transaction T_i starts, it registers itself by writing a $\langle T_i$ **start**$\rangle$ log record
- *Before* T_i executes **write**(X), a log record $\langle T_i, X, V_1, V_2\rangle$ is written
- When T_i finishes it's last statement, the log record $\langle T_i$, **commit**$\rangle$ is written.
- Two approaches using logs
  - Deferred database modification (延迟数据库修改)
  - Immediate database modification (即刻数据库修改)

Deferred Database Modification

- Records all modifications to the log, but defers all the **writes** to after partial commit.
- Transaction starts by writing $<T_i$ **start**$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written.
  - The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes

Deferred Database Modification (Cont.)

- During recovery after a crash
  - a transaction needs to be redone iff both $<T_i$ **start**$>$ and $<T_i$ **commit**$>$ are there in the log.
- **Redo(T_i)** sets the value of all data items updated by the transaction to the new values.
- example (T_0 executes before T_1):

T_0: **read** (A)
A: - A - 50
Write (A)
read (B)
B:- B + 50
write (B)

T_1 : **read** (C)
C:- C- 100
write (C)

Deferred Database Modification (Cont.)

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) redo(T_0) must be performed
  - (c) redo(T_0) must be performed followed by redo(T_1)

Immediate Database Modification

- Allows database updates of an uncommitted transaction to be made as the writes are issued
- Update log record must be written *before* database item is written
- Output of updated blocks 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.

Immediate 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

- Note: B_X denotes block containing X

Immediate Database Modification (Cont.)

- Recovery procedure has two operations:
  - undo(T_i)
  - redo(T_i)
- Both operations must be **idempotent**(等幂/幂等的)
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain <T_i **commit**>
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and <T_i **commit**>
- Undo operations are performed first, then redo operations

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>

- Recovery actions in each case above are:
  - (a) undo(T_0)
  - (b) undo(T_1) and redo(T_0)
  - (c) redo(T_0) and redo(T_1)

Checkpoints

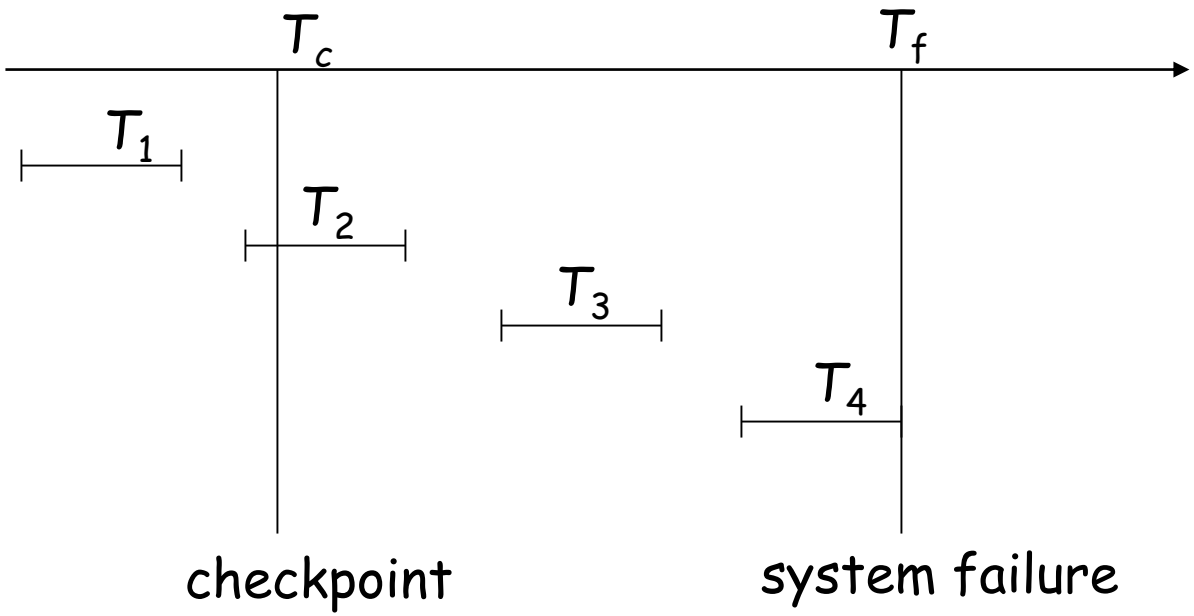
Checkpoints (检查点)

- Problems in the recovery procedure:
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database
- 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>** onto stable storage.

Checkpoints (Cont.)

- During recovery :
  1. Scan backwards from end of log to find the most recent **<checkpoint>** record
  2. Continue scanning backwards till a record $<T_i$ **start>** is found (We have assume that all transactions are executed serially)
  3. Need only consider the part of log following above **start** record.
  4. For all transactions (starting from T_i or later) with no $<T_i$ **commit>**, execute **undo(T_i).**
  5. Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit>**, execute **redo(T_i).**

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

Concurrent Transactions

Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently
  - 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 concurrency control using strict two-phase locking
- Logging is done as described earlier
  - Log records of different transactions may be interspersed in the log
- The checkpointing technique and actions taken on recovery have to be changed

Recovery With Concurrent Transactions (Cont.)

- Checkpoints are performed as before, except that the checkpoint log record is the form

  <checkpoint *L*>

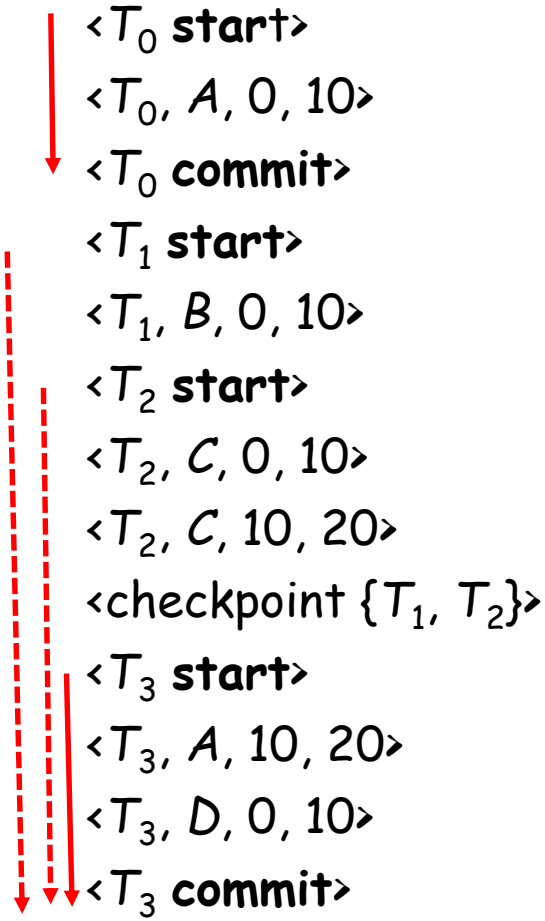
  - *L* is a list of transactions active at the time of the checkpoint
  - We assume no update is in progress while the checkpoint is carried out (this can be relaxed)
- When the system recovers from a crash:
  1. Initialize ***undo-list*** and ***redo-list*** to empty
  2. Scan the log backwards until a <**checkpoint** *L*> record is found:
     - if there is a record <T_i **commit**>, add T_i to *redo-list*
     - if there is a record <T_i **start**> and T_i is not in *redo-list*, add T_i to *undo-list*
  3. For every T_i in *L*, if T_i is not in *redo-list*, add T_i to *undo-list*

Recovery with Concurrent Transactions (Cont.)

- Recovery now continues as follows:
  1. Scan log backwards from the end of the log
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Locate the most recent <**checkpoint** *L*> record.
  3. Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
     - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery

- Go over the steps of the recovery algorithm on the following log:



Redo-list: T3
Undo-list: T1, T2

Log Record Buffering

- Log record buffering: log records are buffered in main memory, instead of 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 all its log records (including the commit record) to stable storage
- Several log records can thus be output using a single output operation, reducing the I/O cost

Log Record Buffering (Cont.)

- 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

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- No update should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**
  - Before a block is output to disk, the system acquires an exclusive latch on the block
    - Ensures no update can be in progress on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned beforehand between database buffer and applications, limiting flexibility
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When OS needs to evict a page that has been modified, the page is written to ***swap space*** on disk
  - When DB decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem
  - Ideally when swapping out a database buffer page, operating system should pass control to database, which in turn outputs page to database instead of to swap space (making sure to output log records first)
    - Dual paging can thus be avoided, but common operating systems do not support such functionality

Failure with Loss of Nonvolatile 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
  - To recover from disk failure
    - restore database from most recent dump
    - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as fuzzy dump or online dump

Course Review

Chapters Related to the Final Exam

- Chapter 1 Introduction
- Chapter 2 Relational Model
- Chapter 3 SQL
- Chapter 4 Advanced SQL
- Chapter 6 Database Design and the ER Model
- Chapter 7 Relational Database Design
- Chapter 10 XML Management
- Chapter 12 Indexing and Hashing
- Chapter 13 Query Processing
- Chapter 14 Query Optimization
- Chapter 15 Transactions
- Chapter 16 Concurrency Control
- Chapter 17 Recovery System

*The sections with ** are omitted*