COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 23
RECOVERY SYSTEM

RECOVERY SYSTEM: OUTLINE

Overview

Log-based Recovery with Serial Transactions

Log-based Recovery with Concurrent Transactions

Log Record Buffering

Shadow Paging

FAILURE CLASSIFICATION

Transaction Failure

Logical errors: A transaction cannot complete due to some internal error condition (e.g., invalid input).

System errors: The database system must terminate an active transaction due to an error condition (e.g., deadlock).

System Crash

- Due to a power, hardware or software failure.

  Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by a system crash.

  - DBMSs have numerous integrity checks to prevent corruption of disk data.

Disk Failure

- A disk failure (e.g., head crash) destroys all or part of disk storage.
- Destruction is assumed to be detectable (e.g., disk drives use checksums to detect failures).

FAILURE RESPONSE

- To determine the system's response to a failure we need to:

  a) identify failure modes of storage devices.

  b) consider how failure modes affect database content.

  c) develop **recovery algorithms** to ensure, despite failures,

     ➢ transaction atomicity,

     ➢ database consistency and

     ➢ transaction durability

- 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 to a state that ensures atomicity, consistency and durability.

STORAGE CONSIDERATIONS

Volatile Storage

- Does not survive system crashes.

  Examples: main memory, cache memory.

Nonvolatile Storage

- Survives system crashes.

  Examples: disk, tape, flash memory, non-volatile (battery backed up) RAM.

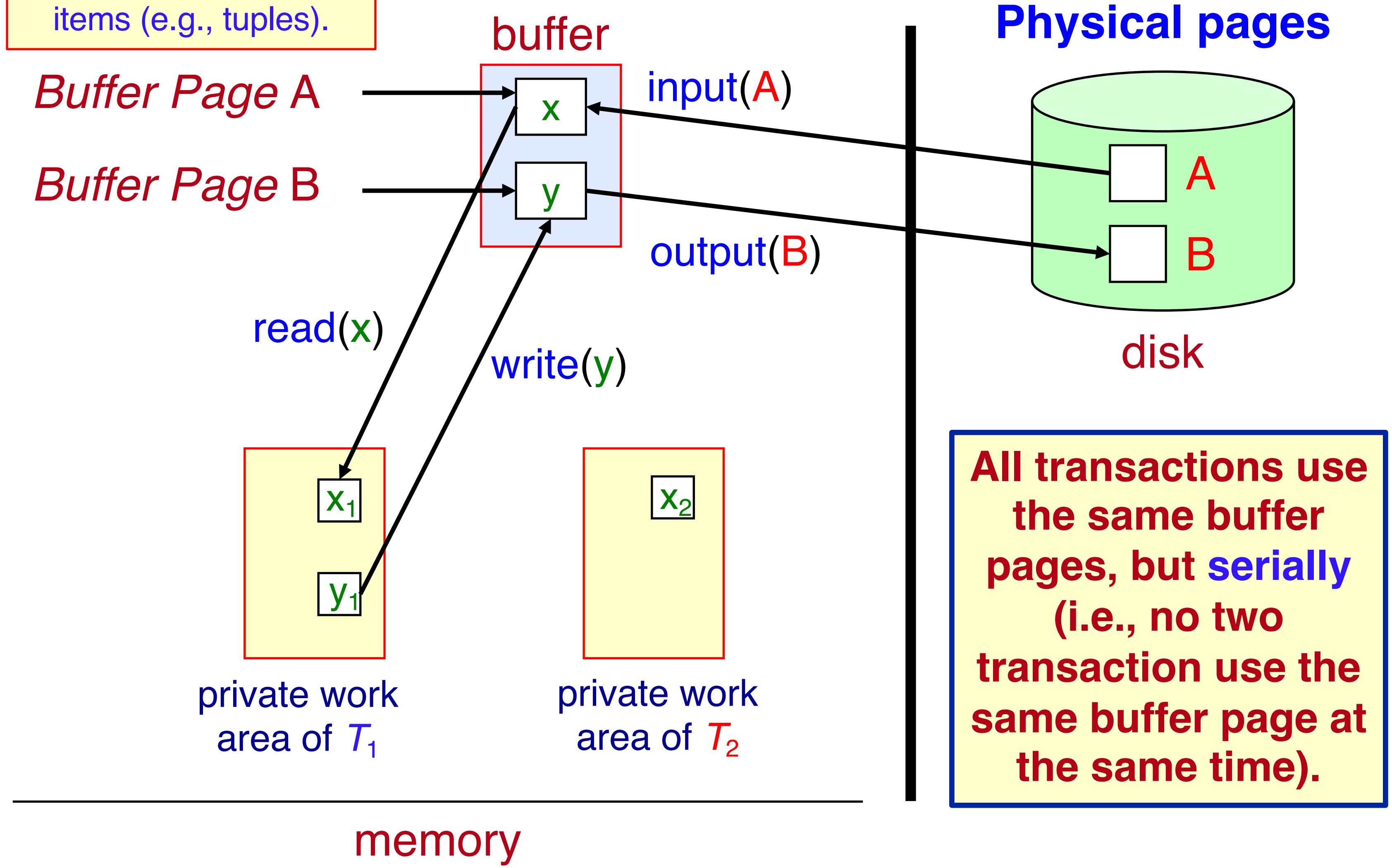
Stable Storage

- A *mythical* form of storage that survives all failures.

  In practice Several copies of the same data are kept in different storage devices (e.g., disks) so all of them cannot be lost at the same time. (*Two copies is the usual practice.*)

DATA ACCESS

x and y are database items (e.g., tuples).



All transactions use the same buffer pages, but serially (i.e., no two transaction use the same buffer page at the same time).

DATA ACCESS (CONT'D)

Physical pages – pages residing on the disk.

Buffer pages – pages residing temporarily in main memory.

- Page movements disk ⇔ main memory use two operations:

  input(*B*) transfers the physical page *B* to main memory.

  output(*B*) transfers the buffer page *B* to the disk and replaces the appropriate physical page on the disk.

- Each transaction T_i has its private work area where local copies of all database items accessed and updated by it are kept (e.g., a variable in a program).

  T_i's local copy of a database item x is called x_i.

- We assume, for simplicity, that each database item fits in, and is stored inside, a single page.

DATA ACCESS (CONT'D)

- A transaction transfers database items between the system buffer pages and its private work area using two operations:

  read(x) assigns the value of database item x to the local variable x_i.

  write(x) assigns the value of local variable x_i to database item x in the buffer page.

  ☞ **Both operations may require an input(B_X), if the page B_X in which x resides is not already in memory.**

- Transactions
  - Perform a read(x) when accessing x for the first time.
  - All subsequent accesses are to the local copy x_i.
  - Perform write(x) after the last access to x_i .

☞ The operation output(B_X) does not need to immediately follow write(x) as the system can perform the output operation when it deems it necessary.

RECOVERY AND ATOMICITY

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Several output operations may be required for T_1 (to output A and B).

A=100, B=200

	T_1	
A=100	read(A)	
	A := A - 50	
A=50	write(A)	output(B_A)
B=200	read(B)	
	B := B + 50	← System failure
	write(B)	
	commit	

- A failure may occur after one of these modifications have been made, but *before* all of them are made.
- To ensure atomicity despite failures, we need to first output information describing the modifications to stable storage without modifying the database itself.

☞ **Our goal is to perform either *all* database modifications made by T_1 or *none* of them.**

Assuming serial execution of transactions.

LOG-BASED RECOVERY

A log is a sequence of log records that maintains a record of all the update activities on the database.

☞ **A log is kept on stable storage (e.g., disk).**

Log File
⋮
<T_i start>
<T_i, x, v_1, v_2>
<T_i commit>
⋮

- When transaction T_i starts, it registers itself by writing a <T_i start> log record.
- *Before* T_i executes write(x), it writes a log record <T_i, x, v_1, v_2>, where v_1 is the *value* of x *before* the write, and v_2 is the *new value* to be written to x.
  - The log record notes that T_i has performed a write on data item x; x had value v_1 before the write and will have value v_2 after the write.
- When T_i finishes its last instruction, the log record <T_i commit> is written.
  - We assume, for now, that log records are written directly to stable storage (that is, they are not buffered in main memory).

Assuming serial execution of transactions.

DEFERRED DATABASE MODIFICATION

All modifications are recorded to the log, but all the writes are deferred until after *partial commit*.

☞ **The old value is not needed in this scheme.**

Log File
⋮
<T_i start>
<T_i, x, v>
<T_i commit>
⋮

- A transaction starts by writing a <T_i start> record to the log.
- A write(x) operation writes a log record <T_i, x, v>, where v is the new value for x.
  - The write of x is not actually performed 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 (i.e., *output to the disk*) the previously deferred writes.

Assuming serial execution of transactions.

DEFERRED DATABASE MODIFICATION (CONT'D)

- During recovery after a crash, a transaction needs to be redone *if and only if* both <T_i start> and <T_i commit> are in the log (otherwise, log entries about T_i are ignored and T_i is re-executed).

- Redoing a transaction
  - redo(T_i) sets the value of all database items updated by the transaction to the new values (using the log information).

- Crashes can occur while
  - the transaction is executing the original updates, or
  - during a recovery action.

Assuming serial execution of transactions.

DEFERRED DATABASE MODIFICATION
(CONT'D)

Log File

The log below shows how it appears at three instances of time (a), (b) and (c).

<T_1 start>	<T_1 start>	<T_1 start>
<T_1, a, 950>	<T_1, a, 950>	<T_1, a, 950>
<T_1, b, 2050>	<T_1, b, 2050>	<T_1, b, 2050>
	<T_1 commit>	<T_1 commit>
	<T_2 start>	<T_2 start>
	<T_2, c, 600>	<T_2, c, 600>
		<T_2 commit>
Time (a)	Time (b)	Time (c)

Example

T_1 executes *before* T_2 with initial values a=1000, b=2000, c=700.

T_1	T_2
read(a)	read(c)
a := a - 50	c := c - 100
write(a)	write(c)
read(b)	
b := b + 50	
write(b)	

If the log on stable storage at the time of the failure is as in case:

(a) [after write(b)] No redo actions need to be taken since there is no commit for any transaction.

(b) [after write(c)] Perform redo(T_1) since <T_1 commit> is present.

(c) [after <T_2 commit>] Perform redo(T_1) followed by redo(T_2) since <T_1 commit> and <T_2 commit> are both present.

Assuming serial execution of transactions.

IMMEDIATE DATABASE MODIFICATION

Database updates of an uncommitted transaction are allowed to be made as the writes are issued.

☞ **Since undoing may be needed, update logs must have both the old value and the new value.**

- The update log record must be written *before* the database item is written.
  - We assume that the log record is *output directly to stable storage*.
  - This can be extended to postpone log record output, so long as prior to performing an output(*B*) operation for a data page *B*, all log records corresponding to items in *B* have been output to stable storage.
- The output of updated pages can take place at any time before or after transaction commit.
- The order in which pages are output to disk can be different from the order in which they are written to in the buffer.

Assuming serial execution of transactions.

IMMEDIATE DATABASE MODIFICATION (CONT'D)

Example

T_1 executes *before* T_2 with initial values a=1000, b=2000, c=700.

T_1	T_2
read(a)	read(c)
a := a - 50	c := c - 100
write(a)	write(c)
read(b)	
b := b + 50	
write(b)	

Log	Write of database item	Page Output
<T_1 start>		
<T_1, a, 1000, 950>		
<T_1, b, 2000, 2050>		
	a = 950	
	b = 2050	
<T_1 commit>		
<T_2 start>		
<T_2, c, 700, 600>		
	c = 600	
		B_b, B_c
<T_2 commit>		
		B_a

Note: B_x denotes the page containing x.

Assuming serial execution of transactions.

IMMEDIATE DATABASE MODIFICATION
(CONT'D)

- The recovery procedure has two operations instead of one:

  undo(T_i) restores the value of all database items updated by T_i to their old values, going backwards from the last log record for T_i.

  redo(T_i) sets the value of all database items updated by T_i to the new values, going forward from the first log record for T_i.

- Both operations must be idempotent.
  - Even if the operation is executed multiple times the effect is the same as if it is executed only once.
  - This is needed since operations may get re-executed during recovery.

- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i start>, but does not contain the record <T_i commit>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i start> and the record <T_i commit>.

☞ **Undo operations are performed first, then redo operations.**

Assuming serial execution of transactions.

IMMEDIATE DATABASE MODIFICATION (CONT'D)

Log File

The log below shows how it appears at three instances of time (a), (b) and (c).

<T_1 start>	<T_1 start>	<T_1 start>
<T_1, a, 1000, 950>	<T_1, a, 1000, 950>	<T_1, a, 1000, 950>
<T_1, b, 2000, 2050>	<T_1, b, 2000, 2050>	<T_1, b, 2000, 2050>
	<T_1 commit>	<T_1 commit>
	<T_2 start>	<T_2 start>
	<T_2, c, 700, 600>	<T_2, c, 700, 600>
		<T_2 commit>
Time (a)	Time (b)	Time (c)

Example

T_1 executes *before* T_2 with initial values a=1000, b=2000, c=700.

T_1	T_2
read(a)	read(c)
a := a - 50	c := c - 100
write(a)	write(c)
read(b)	
b := b + 50	
write(b)	

If the log on stable storage at the time of the failure is as in case:

(a) [after write(b)] undo(T_1): restore b to 2000 and a to 1000.

(b) [after write(c)] First undo(T_2) restoring c to 700. Then redo(T_1) setting a and b to 950 and 2050, respectively.

(c) [after <T_2 commit>] First redo(T_1) setting a and b to 950 and 2050, respectively. Then redo(T_2) setting c to 600.

CHECKPOINTS

Recovery Procedure Problems

1. Searching the entire log is time-consuming.
2. We might unnecessarily redo transactions which have already output their updates to the database.

- We can streamline the recovery procedure by periodically performing checkpointing, which will
  1. output all log records currently residing in main memory onto stable storage.
  2. output all modified buffer pages to the disk.
  3. write a log record <checkpoint> onto stable storage.

☞ **Transactions cannot perform any update actions while a checkpoint is in progress.**

Assuming serial execution of transactions.

CHECKPOINTS (CONT'D)

During recovery we need to consider only

i. the most recent transaction T_i that started *before* the checkpoint and *has not committed*.

ii. all transactions that *started after* the checkpoint.

1. Scan backwards from the end of the log to find the most recent <checkpoint> record.
2. Continue scanning backwards until a <T_i start> record is found.
3. We need only consider the part of the log following this start record. The part of the log before this record can be ignored during recovery and can be erased whenever desired (*since we are assuming serial execution of transactions*).
4. Scan forward in the log starting from T_i.

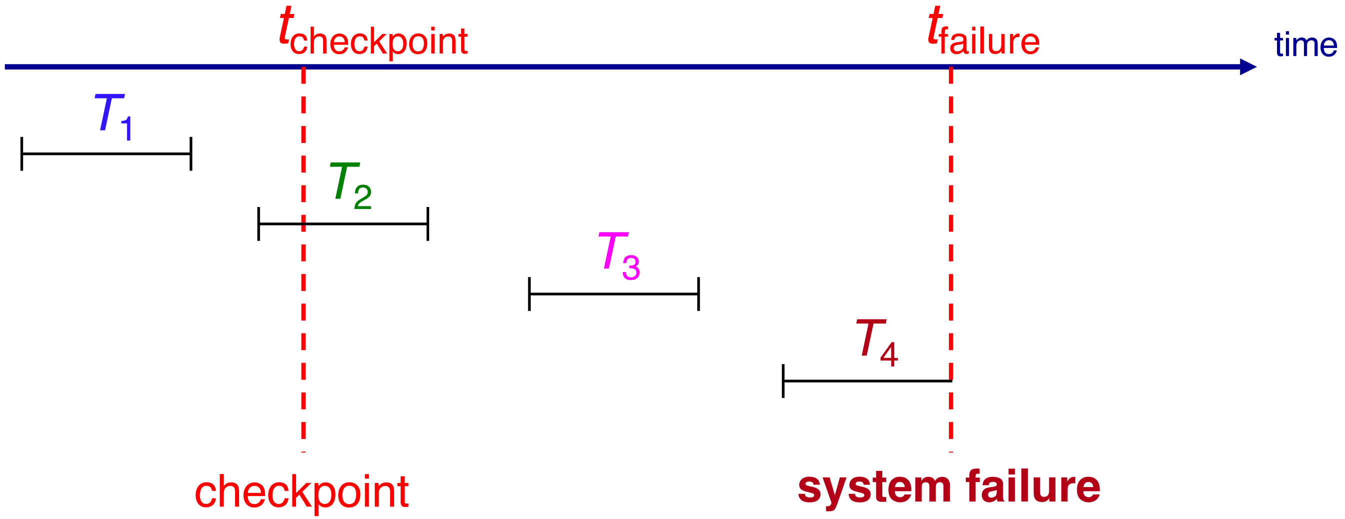
   a) For all transactions with no <T_i commit> or <T_i abort> record, execute undo(T_i). (Done only in the case of immediate modification.)

   b) For all transactions with a <T_i commit>, execute redo(T_i).

Assuming serial execution of transactions.

CHECKPOINTS EXAMPLE

Log File
$<T_1$ start$>$
⋮
$<T_1$ commit$>$
$<T_2$ start$>$
⋮
<checkpoint>
⋮
$<T_2$ commit$>$
$<T_3$ start$>$
⋮
$<T_3$ commit$>$
$<T_4$ start$>$
⋮
system failure



T_1 can be ignored since its updates were already output to disk due to the checkpoint.

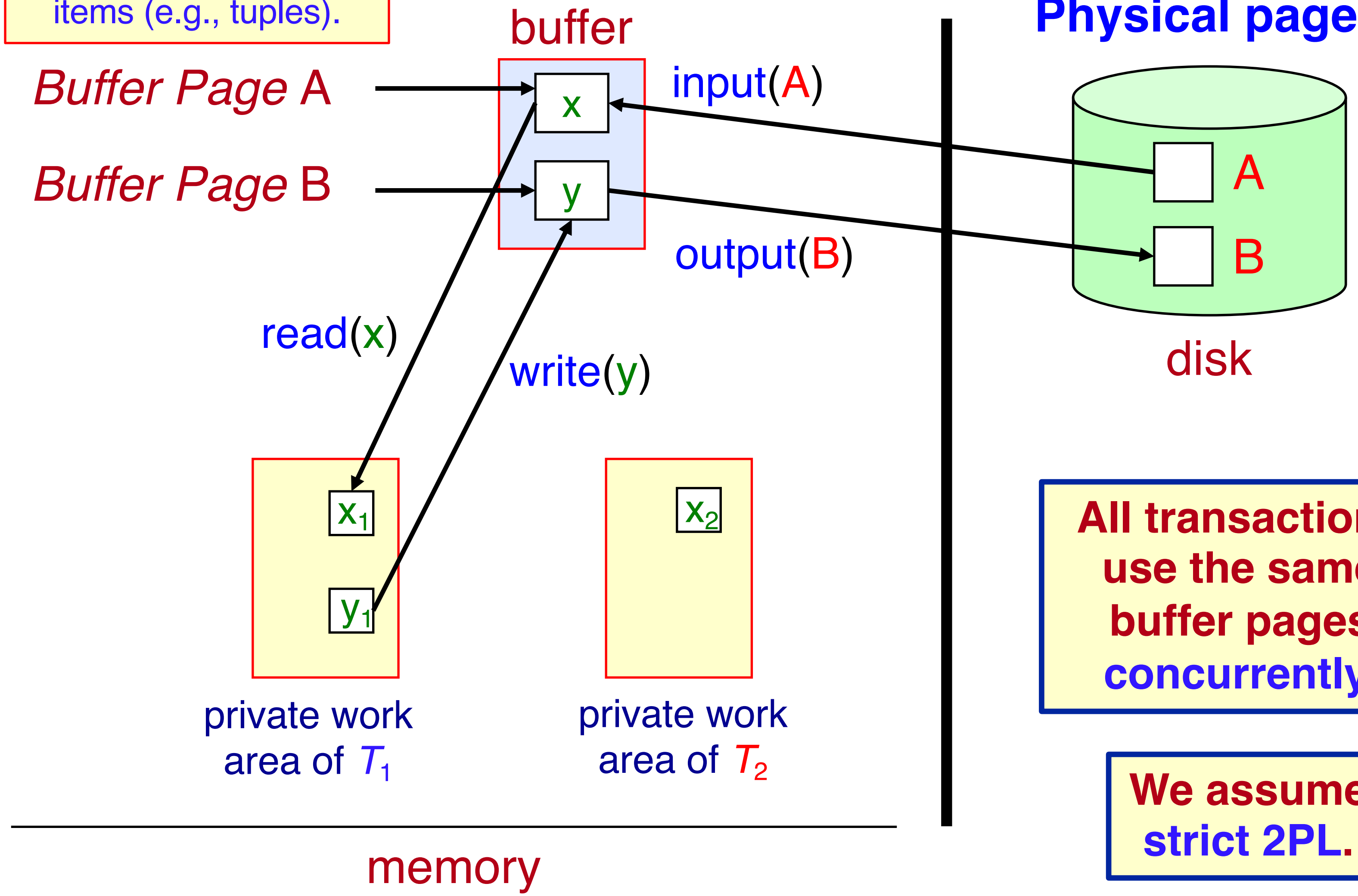
T_2 and T_3 need to be redone.

T_4 needs to be undone.

1. Scan backwards from end of log to find most recent <checkpoint> record.
2. Continue scanning backwards until a $<T_i$ start$>$ record is found.
3. Scan forward in the log starting from T_i.
   - a) For all transactions with no $<T_i$ commit$>$ or $<T_i$ abort$>$ record, execute undo(T_i). (Done only in the case of immediate modification.)
   - b) For all transactions with a $<T_i$ commit$>$, execute redo(T_i).

RECOVERY WITH CONCURRENT TRANSACTIONS

x and y are database items (e.g., tuples).



All transactions use the same buffer pages concurrently.

We assume strict 2PL.

RECOVERY WITH CONCURRENT TRANSACTIONS
(CONT'D)

- 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 page can have database items updated by one or more transactions.
- We assume concurrency control using *strict two-phase locking*.
  - Recall that in strict 2PL a transaction holds all its exclusive locks until it commits – therefore, other transactions can only read "final" data and the schedule is cascadeless.
- 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 since several transactions may be active when a checkpoint is performed.

RECOVERY WITH CONCURRENT TRANSACTIONS
(CONT'D)

- The checkpoint log record is now of the form <checkpoint *L*> where *L* is the list of active transactions at the time of the checkpoint.

  ☞ **We assume no updates are in progress while the checkpoint is carried out.**

- When recovering from a crash, the system first does the following:

  1. Initialize *undo-list* and *redo-list* to empty.
  2. Scan the log backwards from the end, stopping when the first <checkpoint *L*> record is found.
  3. For each record found during the backward scan:
     i. If the record is <T_i commit>, add T_i to the *redo-list*.
     ii. If the record is <T_i start>, then if T_i is not in the *redo-list*, add T_i to the *undo-list*.
  4. For every T_i in *L*, if T_i is not in the *redo-list*, add T_i to the *undo-list*.

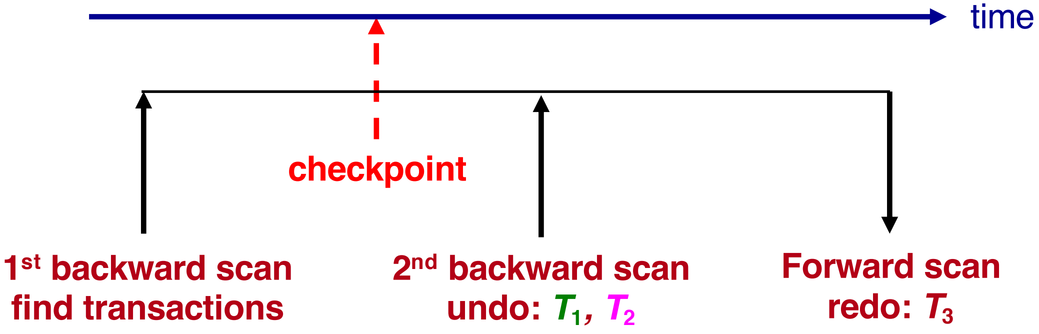
RECOVERY WITH CONCURRENT TRANSACTIONS (CONT'D)

- At this point the *undo-list* consists of incomplete transactions which must be undone, and the *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan the log backward from the most recent record, stopping when all the <T_i start> records have been encountered for every T_i in the *undo-list*.
     - ➢ During the scan, perform undo for each log record that belongs to a transaction in the *undo-list*.
  2. Locate the most recent <checkpoint *L*> record.
  3. Scan the log forward from the <checkpoint *L*> record until the end of the log.
     - ➢ During the scan, perform redo for each log record that belongs to a transaction in the *redo-list*.

☞ **Undoing backwards restores the original values, while redoing forward sets each item to the most recent value.**

Recovery with Concurrent Transactions Example

Log File
<T_0 start>
<T_0, A, 0, 10>
<T_0 commit>
<T_1 start>
<T_1, B, 0, 10>
<T_2 start>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 start>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 commit>
system failure



time

checkpoint

1st backward scan find transactions

2nd backward scan undo: T_1, T_2

Forward scan redo: T_3

Undo:

Redo:

1st backward scan - find transactions:

1. If <T_i commit>, add T_i to the *redo-list*.
2. If <T_i start>, then if T_i is not in *redo-list*, add T_i to *undo-list*.
3. For every T_i in <checkpoint L>, if T_i is not in *redo-list*, add T_i to *undo-list*.

2nd backward scan - undo transactions:

1. Stop when all <T_i start> records for every T_i in *undo-list* is found.
   - Perform undo for each transaction in *undo-list*.

Forward scan - redo transactions:

1. Locate most recent <checkpoint L> record.
2. Perform redo for each transaction in *redo-list*.

LOG RECORD BUFFERING

- Instead of writing log records individually, they can be buffered in main memory and output to stable storage when a page is full or a log force operation is executed.

- A log force operation commits a transaction by forcing all its log records (including the commit record) to stable storage.
  - ☞ **Several log records can be output by a single output operation.**

- The following rule, called the write-ahead logging or WAL rule, must be followed if log records are buffered.
  1. Log records are output to stable storage in their create order.
  2. Transaction T_i enters the commit state only when the log record <T_i commit> has been output to stable storage.
  3. Before a page of data in main memory is output to the database, all log records pertaining to data in that page must have been output to stable storage.

SHADOW PAGING

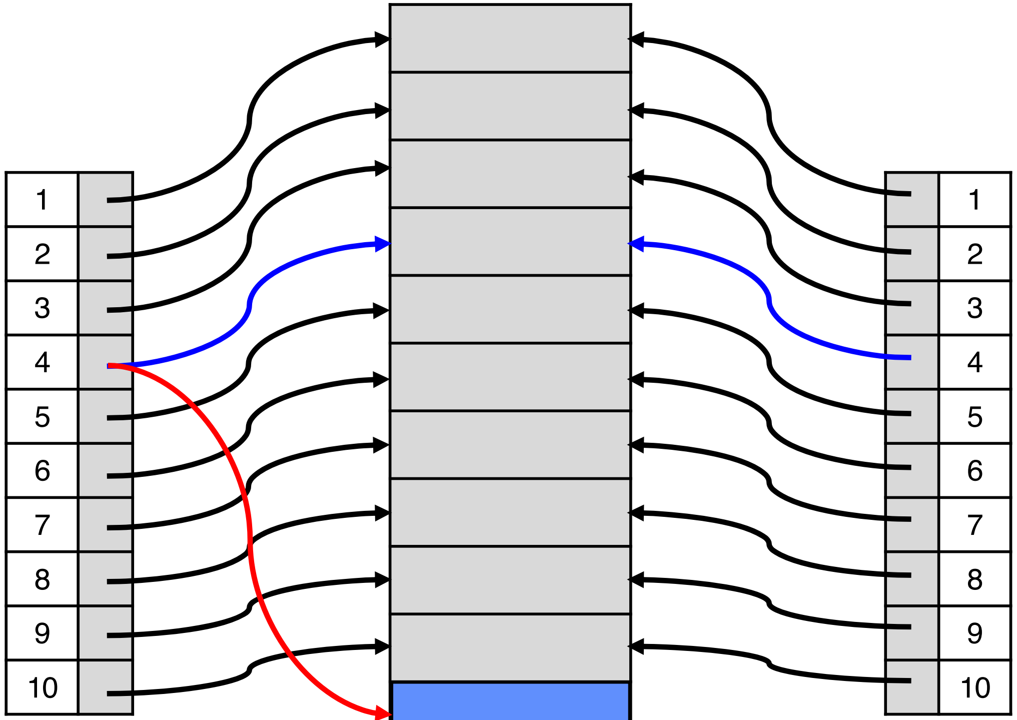
Idea: Maintain *two* page tables during the lifetime of a transaction – the *current* page table and the *shadow* page table.

Whenever any page is about to be written for the first time:

1. A copy of this page is made onto an unused page.
2. The current page table is then made to point to the copy.
3. The update is performed on the copy.

To commit a transaction:

1. Output all modified pages in main memory to disk.
2. Output the current page table to disk.
3. Make the current page table the new shadow page table.



current page table
(points to disk pages)

shadow page table
(points to disk pages)

pages on disk

No recovery needed after a crash!

SHADOW PAGING (CONT'D)

- The shadow page table is stored in nonvolatile storage, such that the state of the database prior to transaction execution can be recovered.
  - ☞ **The shadow page table is never modified during execution.**

- Initially, both page tables are identical.

- Only the current page table is used for database item accesses during execution of the transaction.

- Whenever any page is about to be written for the first time:
  1. A copy of this page is made onto an unused page.
  2. The current page table is then made to point to the copy.
  3. The update is performed on the copy.

SHADOW PAGING (CONT'D)

- To commit a transaction:
  1. Output all modified pages in main memory to disk.
  2. Output the current page table to disk.
  3. Make the current page table the new shadow page table, as follows:
     - i. keep a pointer to the shadow page table at a fixed (known) location on disk.
     - ii. to make the current page table the new shadow page table, simply update the pointer to point to the current page table on disk.
- Once the pointer to the shadow page table has been written, the transaction is committed.
- **No recovery is needed after a crash** — new transactions can start right away, using the shadow page table.
- Pages not pointed to from the current/shadow page table should be freed (garbage collected).

SHADOW PAGING (CONT'D)

Advantages

- No overhead of writing log records.
- Recovery is trivial.

Disadvantages

- Copying the entire shadow page table is very expensive.
  - The cost can be reduced by using a page table structured like a B^+-tree.
  - There is no need to copy the entire tree; only need to copy paths in the tree that lead to updated leaf nodes.
- The commit overhead is high even with the above extension.
  - Need to output every updated page and the page table.
- The data gets fragmented (related pages get separated on disk).
- After every transaction completion, the database pages containing old versions of modified data need to be garbage collected.
- Hard to extend algorithm to allow transactions to run concurrently.
  - Easier to extend log based schemes.

RECOVERY SYSTEM: SUMMARY

- Database recovery is an essential part of a DBMS.

  ☞ **Since the computer system and/or a transaction may fail.**

- Each transaction must be atomic to preserve consistency.

- A recovery scheme ensures transaction atomicity, consistency and durability.

- Recovery schemes use:
  - logs with immediate or deferred modification (most common approach).
  - shadow paging (not practical for large databases).

- Checkpoints reduce overhead of searching the log and redoing transactions.