Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a ''validation test'' to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time; this is done to increase concurrency.
  - Thus, TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B \quad 50$
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of 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, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions <Q_1, Q_2,...., Q_m>. 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
- When a transaction T_i creates a new version Q_k of Q, Q_k's W-timestamp and R-timestamp are initialized to TS(T_i).
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k, and TS(T_j) > R-timestamp(Q_k).

Multiversion Timestamp Ordering (Cont)

- 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.
  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. else a new version of Q is created.
- Observe that
  - 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

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading the current value of **ts-counter** before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads.

Multiversion Two-Phase Locking (Cont.)

- When an update transaction wants to read a data item:
  - it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction T_i completes, commit processing occurs:
  - T_i sets timestamp on the versions it has created to **ts-counter** + 1
  - T_i increments **ts-counter** by 1
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g. if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again

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 1: Give logical "snapshot" of database state to read only transactions, read-write transactions use normal locking
  - Multiversion 2-phase locking
  - Works well, but how does system know a transaction is read only?
- Solution 2: Give snapshot of database state to every transaction, updates alone use 2-phase locking to guard against concurrent updates
  - Problem: variety of anomalies such as lost update can result
  - Partial solution: snapshot isolation level (next slide)
    - Proposed by Berenson et al, SIGMOD 1995
    - Variants implemented in many database systems
      - E.g. Oracle, PostgreSQL, SQL Server 2005

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

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

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

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

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)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

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)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits of SI

- Reading is *never* blocked,
  - and also doesn't block other txns activities
- Performance similar to Read Committed
- Avoids the usual anomalies
  - No dirty read
  - No lost update
  - No non-repeatable read
  - Predicate based selects are repeatable (no phantoms)
- 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

Snapshot Isolation

- E.g. of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

Snapshot Isolation Anomalies

- SI breaks serializability when txns modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify too (like a total quantity) so SI will abort one of them
  - But does occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

Deadlocks

- Consider the following two transactions:

T_1: write (X) write(Y)

T_2: write(Y) write(X)

- S

T_1	T_2
lock-X on A write (A)	
	lock-X on B write (B) wait for **lock-X** on A
wait for **lock-X** on B	

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

Recovery Algorithms

- Consider transaction T_i that 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 to a state that ensures atomicity, consistency and durability

Storage Structure

- **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
  - 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
  - See book for more details on how to implement stable storage

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.
    2. When the first write successfully completes, write the same information onto the second physical block.
    3. The output is completed only after the second write successfully completes.

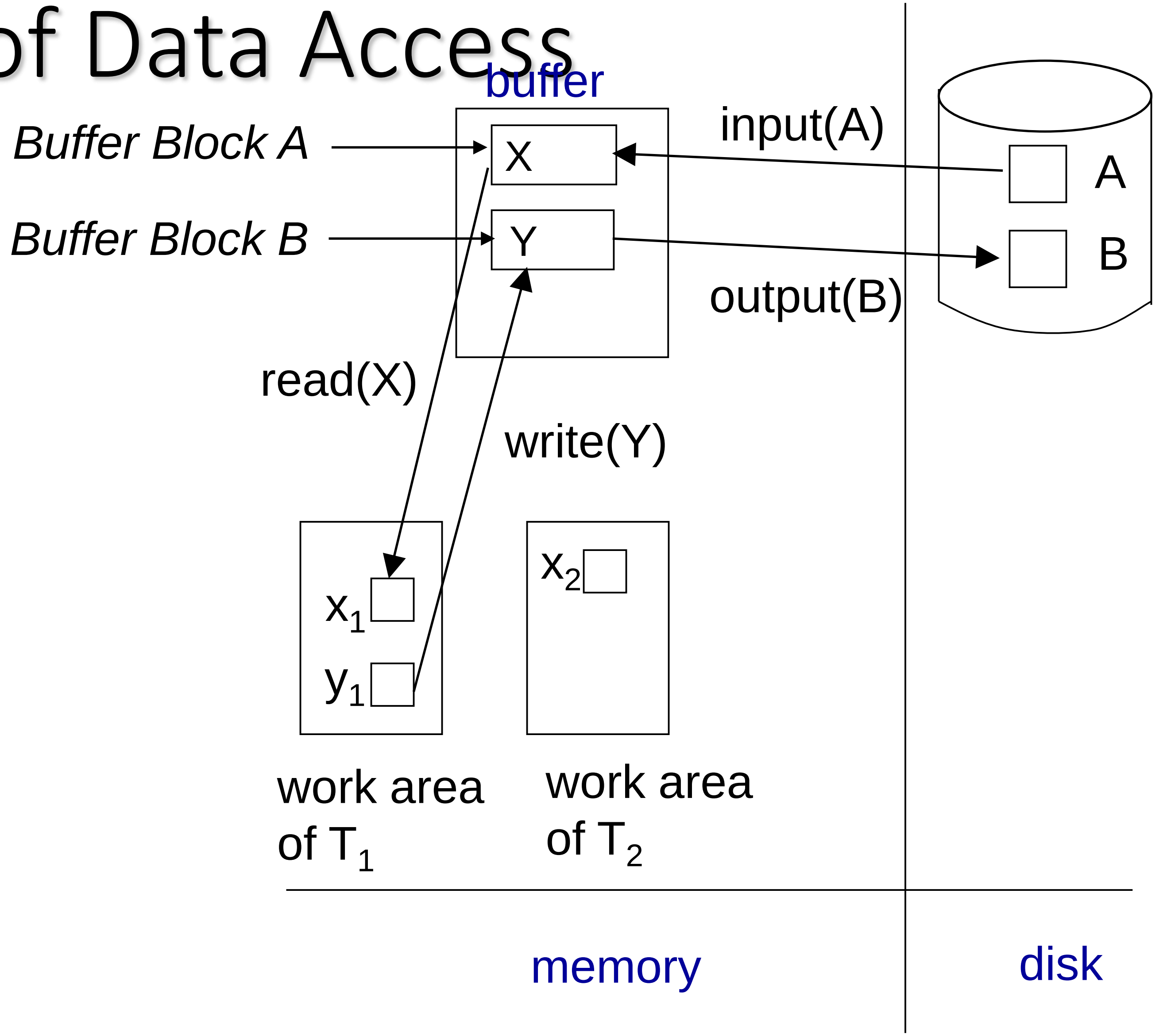
Stable-Storage Implementation (Cont.)

- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

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, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



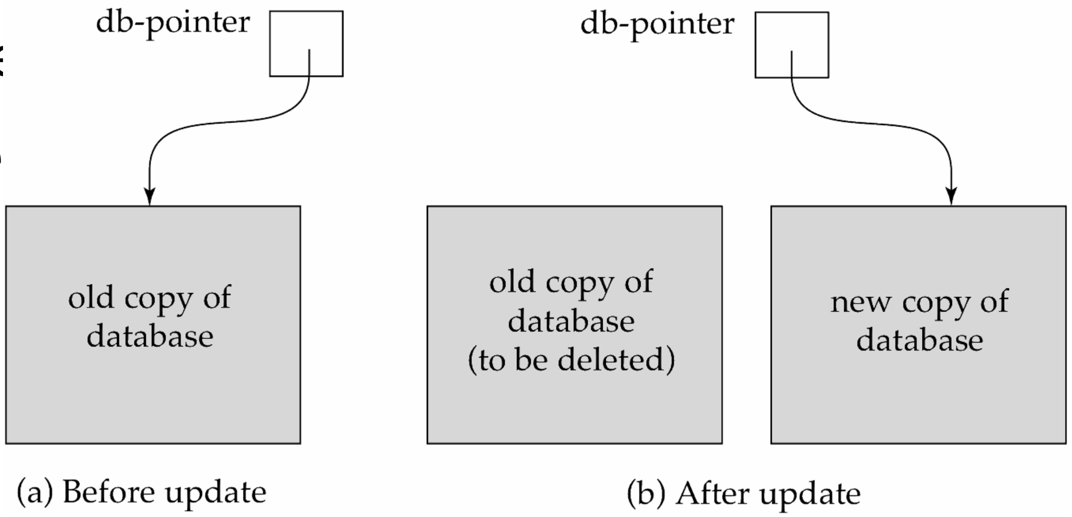
Data Access (Cont.)

- 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 before the transaction commits

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the
- Less used alternative (brief details in book)

shadow-copy



Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- 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 **commi**t> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate 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
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to stable storage 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 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.

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 T1 updates A, then T2 updates A and commits, and finally T1 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** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **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
    - No logging is done in this case

Undo and Redo on 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**>
- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
  - **such a redo redoes all the original actions *including the steps that restored old values***
    - Known as **repeating history**
    - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

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

<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)	(b)	(c)

Recovery actions in each case above are:

(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

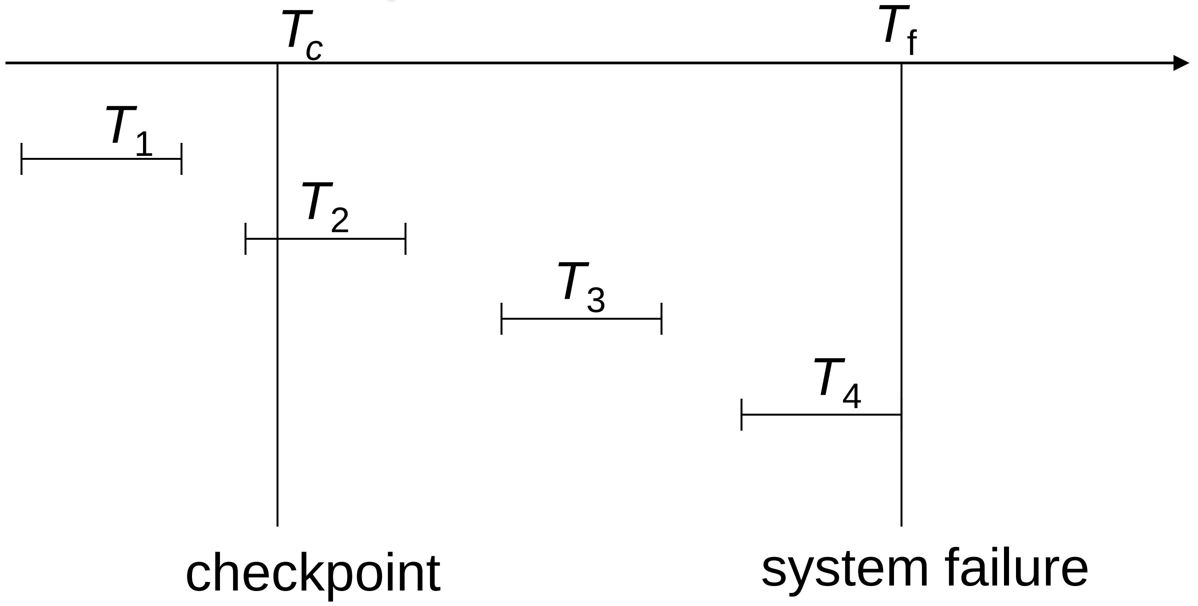
Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
  1. processing the entire log is time-consuming if the system has run for a long time
  2. 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.
  - 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.
  1. 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
  1. 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 Algorithm

- **So far:** we covered key concepts
- **Now**: we present the components of the basic recovery algorithm
- **Later**: we present extensions to allow more concurrency

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**>

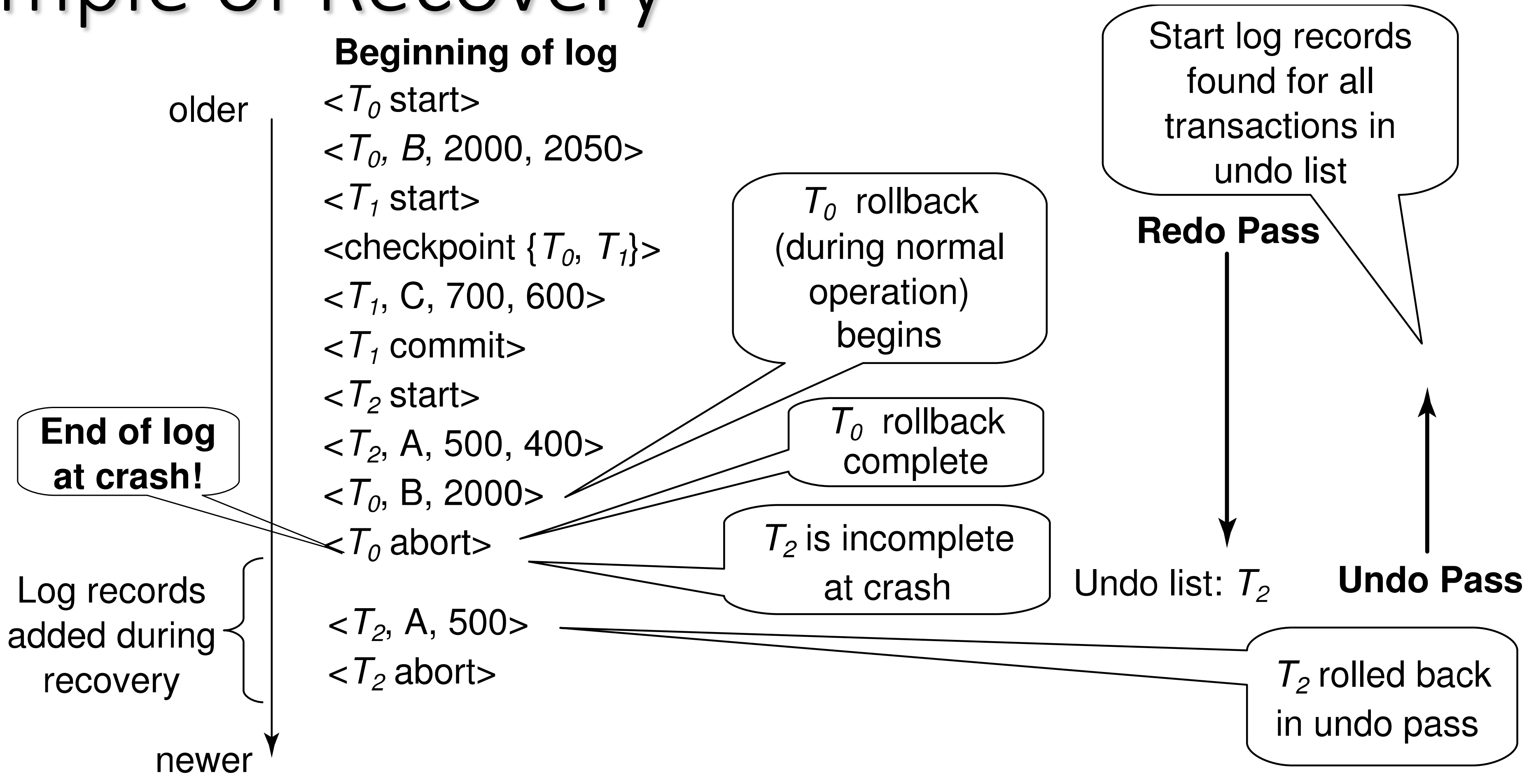
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record <T_i **start**> is found, add T_i to undo-list
     3. Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record $<T_i , X_j, V_1>$
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



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
      - Strictly speaking WAL only requires undo information to be output

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
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**:i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates 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**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     1. Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch 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 before-hand 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.

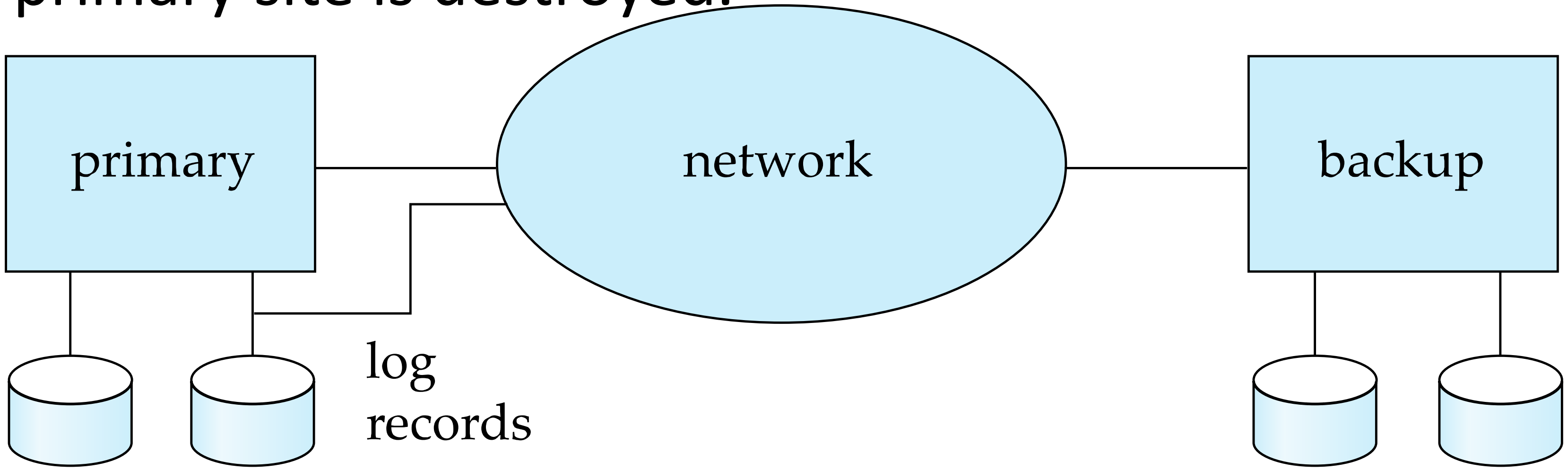
Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database 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 OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use

    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages
- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically proceses the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19

Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe**: commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe**: commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe**: proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.