Impossibile visualizzare l'immagine.

Recovery System

These slides are a modified version of the slides of the book "Database System Concepts" (Chapter 17), 5th Ed., McGraw-Hill, by Silberschatz, Korth and Sudarshan.
Original slides are available at www.db-book.com

Transactions: ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Consistency: Programmer

Isolation: Concurrency Control System

Atomicity and Durability: Recovery System

Transaction

- commit
  termination with success of the transaction
  all operations are executed and changes to the database are persistent
- abort (or rollback)
  abort of the transaction
  none operation is executed

Transfers $50 from account *A* to account *B*

```
start transaction;
update Account
        set balance = balance – $50  where Accout_number = A;
update Account
        set balance = balance + $50 where Account_number = B;
commit;
```

Abort of a Transaction

1) Abort if balance of A less than $50

```
start transaction;
update Account
        set balance = balance – $50  where Accout_number = A;
update Accont
        set balance = balance + $50 where Account_number = B;
select balance into V
        from Account where Account_number = A;
if (V>=0)    then commit
                 else abort;
```

2) Abort if the system has entered an undesirable state (e.g. deadlock)

3) Abort in presence of failures

Failures

- A computer system is subject to failures
- Causes are: disk failure, power outage, hardware or software errors, ….
- In any failure, information may be lost
- DBMS must take actions in advance to ensure that atomicity and durability properties of transactions are preserved in case of failures

Recovery System:
it can restore the database to the consistent state that existed before the failure

ASSUMPTIONS on failures:

- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**:
    non-volatile storage contents are not 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 drivers use checksums to detect failures

Recovery Algorithms

- Recovery algorithms are techniques to ensure database transaction **atomicity and durability despite failures**

- 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 and durability

Storage Structure

Resilience to failure classification:

- **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
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - **Information residing in stable storage is never lost!!!**
    (theoretically cannot be guaranteed - it can be closely approximated by techniques that make data loss extremely unlikely)

The recovery systems relies on stable storage

Data Access

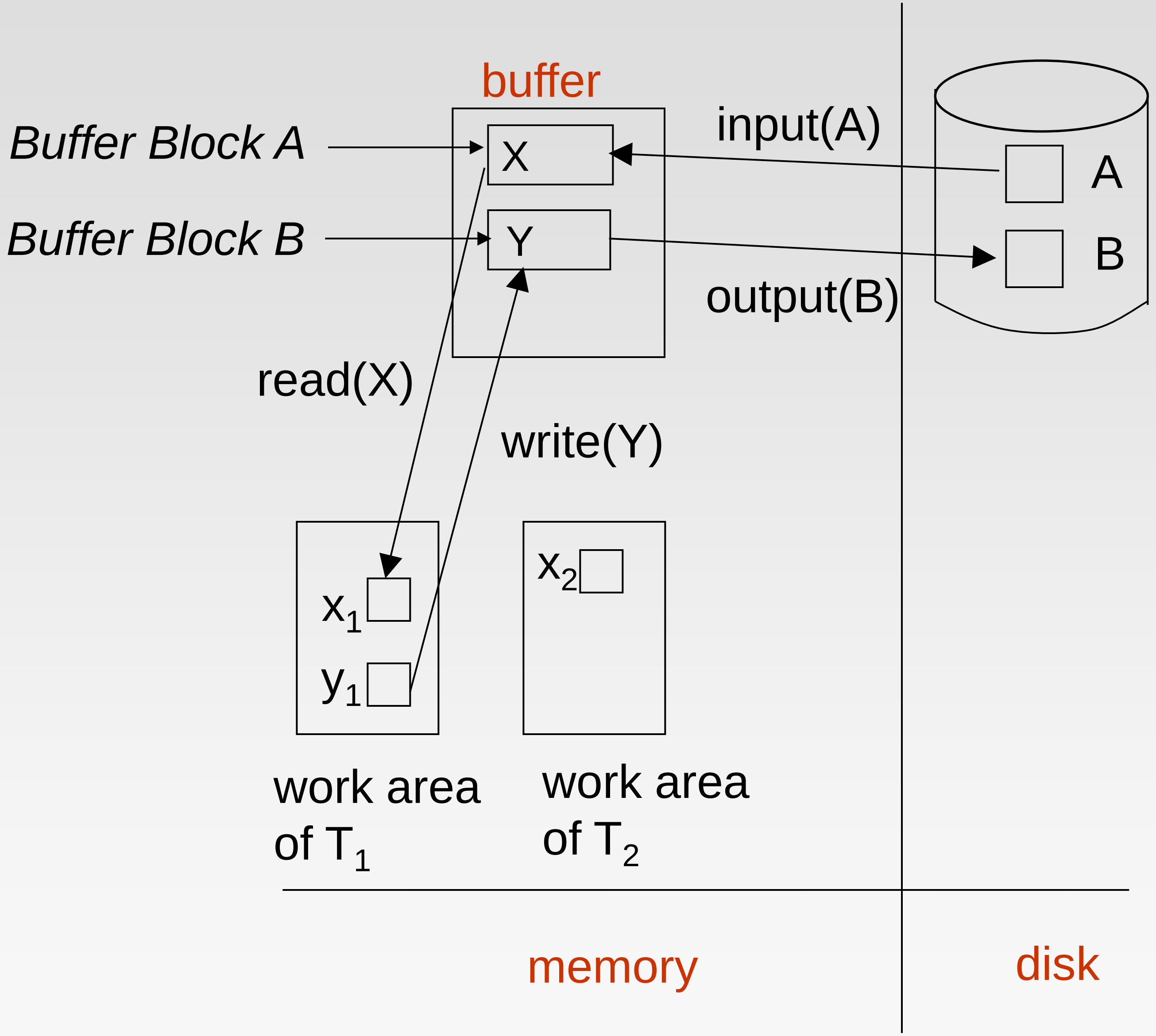
- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

- 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.
- 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 the following operations :
  - **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.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- 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.

- We study
  - **Log-based recovery**

We assume (initially) that transactions run serially, that is, one after the other.

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 log record

  <T_i **start**>

- *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, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.

- When T_i finishes it last statement (**commit** statement) (**partial commit of the transaction**), the following log record is written:

  <T_i **commit**>

Log-Based Recovery

Transaction **rollback** during normal operation

- If T_i executes the **abort** statement, the transaction is undone:

  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 (**write**(*X*))
  - a special log record is written out

    < T_i, X, V>

    such log records are called compensation log records

  - when undo of a transaction is complete, the following log record is written:

    <T_i **abort**>

- We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Log-Based Recovery

output(B_X) need not immediately follow **write**(X).
System can perform the **output** operation when it deems fit.

- Possible schemes for the execution of the **output**(B_X) operations are:
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to disk **after partial commit**
  - the **output**(B_X) operation executed after the partial commit
- old value of X is not needed in the log file for this scheme

- 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, where V is the new value for X
- The write is not performed on X, but it is deferred after the partial commit.
- When T_i partially commits, <T_i **commit**> is written to the log

At the checkpoint, 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** if and only if both <T_i **start**> and<T_i **commit**> are there in the Log.
- Redoing a transaction T_i (**redo**T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions T_0 and T_1 (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 DB Modification Recovery Example

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

Assume A= 1000, B=2000, C =700

<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>
(a)	(b)	(c)

Assume we have a crash.

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since <T_0 **commit**> is present

(c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i **commit**> are present

Immediate Database Modification

- The **immediate database modification** scheme allows database updates **output**(*B*) of an uncommitted transaction to be made as the writes are issued
- since undoing may be needed, update logs must have both **old value** and new value $<T_i, X, V_1, V_2>$
- **Output of updated blocks can take place at any time (before or after transaction commit)**
- **Update Log record must be written *before* database item is written**
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **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
  - **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
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - 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.

Immediate DB Modification Recovery Example

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

(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 is restored to 2000 and A to 1000.

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

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

DB Modification: An Example

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

- Note: B_X denotes block containing X.

Checkpoints

- Problems in 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.

- 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**> onto stable storage

Transactions are not allowed to execute any actions while a checkpoint is in progress.

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**> record
  2. Continue scanning backwards till a record <T_i **start**> is found for transaction in the checkpoint.
  3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, or <T_i **abort**>, execute **undo(**T_i). (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**> *or* <T_i **abort**>, execute **redo(**T_i).

Checkpoints (Cont.)

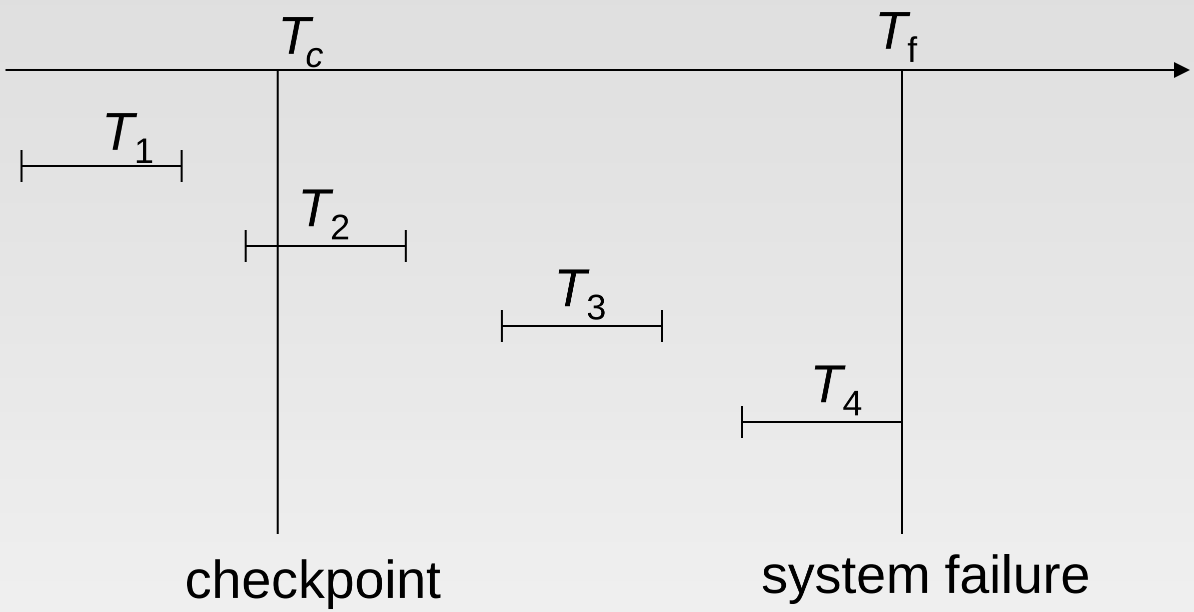
Note that

- 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

Example of Checkpoints



Recovery from system failure

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

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**;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
- 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.)

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

< **checkpoint** *L*>

where *L* is the list of **transactions active** at the time of the checkpoint

- When the system recovers from a crash, it 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.
     For each record found during the backward scan:
     - if the record is <T_i **commit**>/<T_i **abort**>, add T_i to *redo-list*
     - if the record is <T_i **start**>, then if 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.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when <T_i **start**> records have been encountered for every T_i in L.
     - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
  2. Scan log forwards from the <T_i **start**> oldest record found at step 1 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:

  <T_0 **start**>

  <T_0, A, 0, 10>

  <T_0 **commit**>

  <T_1 **start**> /* Scan at step 1 comes up to here */

  <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**>

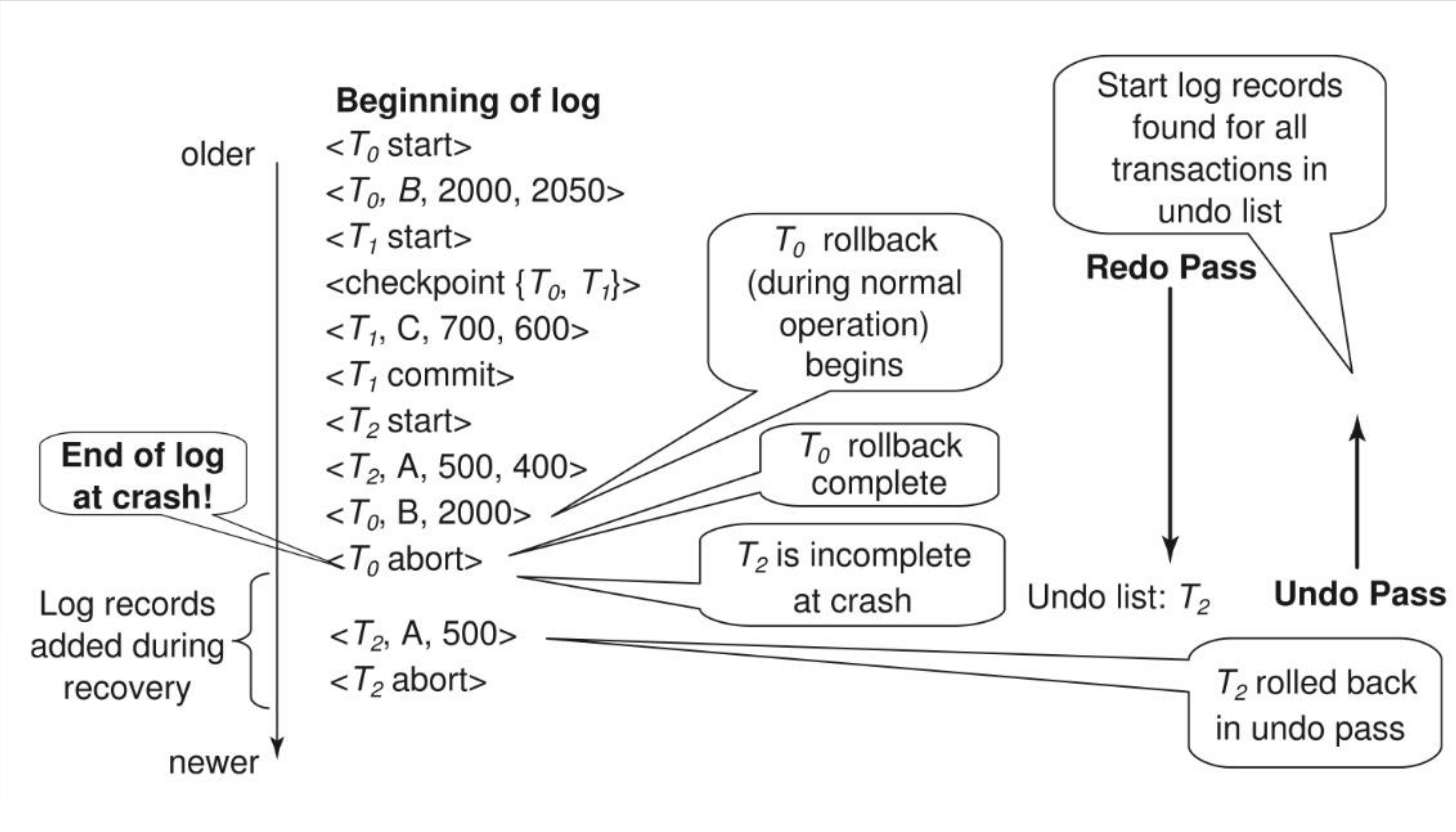
  <T_3, A, 10, 20>

  <T_3, D, 0, 10>

  <T_3 **commit**>

  crash

Example of recovery (<T0 abort>)



Log Record Buffering

- **Log record buffering**: log records are 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.

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. **Log force** is performed to commit a transaction by forcing all its log records (including the commit record) 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
- 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.

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.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
  - Apply the Log Recovery