Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Chapter 6: Database Recovery Techniques

Database Management Systems (CO3021)

Computer Science Program

Dr. Võ Thị Ngọc Châu

(chauvtn@hcmut.edu.vn)

Semester 1 – 2018-2019

Course outline

- Overall Introduction to Database Management Systems
- Chapter 1. Disk Storage and Basic File Structures
- Chapter 2. Indexing Structures for Files
- Chapter 3. Algorithms for Query Processing and Optimization
- Chapter 4. Introduction to Transaction Processing Concepts and Theory
- Chapter 5. Concurrency Control Techniques
- **Chapter 6. Database Recovery Techniques**

References

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 6th Edition, Pearson- Addison Wesley, 2011.
  - ***R. Elmasri, S. R. Navathe, Fundamentals of Database Systems- 7th Edition, Pearson, 2016.***
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts –3rd Edition, McGraw-Hill, 1999.
- [Internet] …

Content

- 6.1. Purposes of Database Recovery
- 6.2. Recovery Concepts
- 6.3. NO-UNDO/REDO Recovery Based on Deferred Update
- 6.4. Recovery Techniques Based on Immediate Update
- 6.5. Shadow Paging
- 6.6. The ARIES Recovery Algorithm
- 6.7. Recovery in Multidatabase Systems
- 6.8. Database Backup and Recovery from Catastrophic Failures

6.1. Purposes of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).
- Example: If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect values. Thus, the database must be restored to the state before the transaction modified any of the accounts.

6.2. Recovery Concepts

- Recovery from transaction failures
- System log
- Recovery policies: deferred update and immediate update
- Caching (buffering) of disk blocks
- Write-ahead logging
- Steal/No-Steal and Force/No-Force
- Checkpoints and fuzzy checkpointing
- Rollback and cascading rollback

6.2. Recovery Concepts

The database may become unavailable for use due to

- Transaction failure: Transactions may fail because of incorrect input, deadlock, incorrect synchronization, etc.
- System failure: System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
- Media failure: Disk head crash, power disruption, etc.
- …

→*Recovery from transaction failures* restores the database to the most recent consistent state before the time of failure.

6.2. Recovery Concepts

- **System Log (Transaction Log)**

  For recovery from any type of failure data *values prior to modification* (BFIM - BeFore IMage) and the *new value after modification* (AFIM – AFter IMage) are required. These values and other information are stored in a sequential file called ***Transaction log***. A sample log is given below. **Back P** and **Next P** point to the previous and next log records of the same transaction.

	TID	Back P	Next P	Operation	Data item	BFIM	AFIM
1	T1	0	2	Begin			
2	T1	1	4	Write	X	X=100	X=200
3	T2	0	8	Begin			
4	T1	2	5	Write	Y	Y=50	Y=100
5	T1	4	7	Read	M	M=200	M=200
6	T3	0	9	Read	N	N=400	N=400
7	T1	5	NULL	End			

6.2. Recovery Concepts

- **Data Updates**
  - **Immediate Update**: As soon as a data item is modified in cache, the disk copy is updated.
  - **Deferred Update**: All modified data items in the cache are written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
  - **Shadow update**: The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
  - **In-place update**: The disk version of the data item is overwritten by the cache version.

6.2. Recovery Concepts

- Two main policies for recovery from noncatastrophic transaction failures
  - **Deferred update**: do not physically update the database on disk until *after* a transaction commits; then the updates are recorded in the database.
    - NO-UNDO/REDO
  - **Immediate update**: the database *may be updated* by some operations of a transaction *before* the transaction reaches its commit point. However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery possible.
    - UNDO/REDO, UNDO/NO-REDO

6.2. Recovery Concepts

- At the data/operation level
  - **UNDO**: reverse data changes by *undoing* write operations of *uncommitted* transactions
    - Restore all BFIMs on disk
  - **REDO**: *redo* operations of *committed* transactions
    - Restore all AFIMs on disk

→ The UNDO and REDO operations are required to be **idempotent** — executing an operation multiple times is equivalent to executing it just once.

→ The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery*!

6.2. Recovery Concepts

- **Data Caching**

  Data items to be modified are first stored into *database cache* by the Cache Manager and after modification they are flushed (written) to the disk. The flushing is controlled by **Modified** and **Pin-Unpin** bits.

  - **Pin-Unpin**: **pinned** (1) if it cannot be written back to disk; instructs the operating system not to flush the data item.
  - **Modified**: a dirty bit to indicate whether or not the buffer has been modified; i.e. indicate the AFIM of the data item.

6.2. Recovery Concepts

- **Write-Ahead Logging**

  When **in-place** update (immediate or deferred) is used, a log is necessary for recovery, available for the recovery manager. This is achieved by **Write-Ahead Logging** (WAL) protocol.

  - **Undo**: Before a data item's AFIM is flushed to the database disk (overwriting the BFIM), its **BFIM** must be written to the log and the log must be saved on a stable store (log disk).
  - **Redo**: Before a transaction executes its commit operation, all its **AFIMs** must be written to the log and the log must be saved on a stable store.

6.2. Recovery Concepts

- **Steal/No-Steal and Force/No-Force**

  Possible ways for flushing database cache to disk:

  Steal: *a* cache buffer page can be flushed *before* the transaction commits.

  No-Steal: *a* cache buffer page cannot be flushed *before* the transaction commits.

  Force: *all* the updated pages are immediately flushed (forced) to disk *before* the transaction commits.

  No-Force: *all* the updated pages are deferred *before* the transaction commits.

  → Four various ways for handling recovery:
    - Steal/No-Force (Undo/Redo)
    - Steal/Force (Undo/No-redo)
    - No-Steal/No-Force (No-undo/Redo)
    - No-Steal/Force (No-undo/No-redo)

6.2. Recovery Concepts

- **Checkpoints**

  From time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery. The following steps define a *checkpoint* operation:

  1. Suspend execution of transactions temporarily.
  2. Force-write modified buffer data to disk.
  3. Write a [*checkpoint*] record to the log, force-write the log to disk.
  4. Resume executing transactions.

  During recovery, **redo** or **undo** is required to transactions appearing after the [*checkpoint*] record.

6.2. Recovery Concepts

- **Fuzzy checkpointing**
  - The time needed for force-writing all modified buffers may delay transaction processing because of step 1. To reduce this delay, use ***fuzzy checkpointing***.
  - In this technique, the system can resume transaction processing after the [*begin_checkpoint*] record is written to the log without waiting for step 2 to finish.
  - Until step 2 is completed, the previous [*checkpoint*] record should remain valid. To accomplish this, the system maintain a pointer to the valid checkpoint, which continues to point to the previous [*checkpoint*] record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

6.2. Recovery Concepts

- **Transaction Roll-back and Roll-Forward**
  - To maintain atomicity, a transaction's operations are **redone** or **undone** when rolled-forward or rolled-back for recovery.
  - Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two. These operations are recorded in the log as they happen.
  - Rollback: *uncommitted* transactions
  - Rollforward: *committed* transactions

6.2. Recovery Concepts

T_1
read_item(A)
read_item(D)
write_item(D)

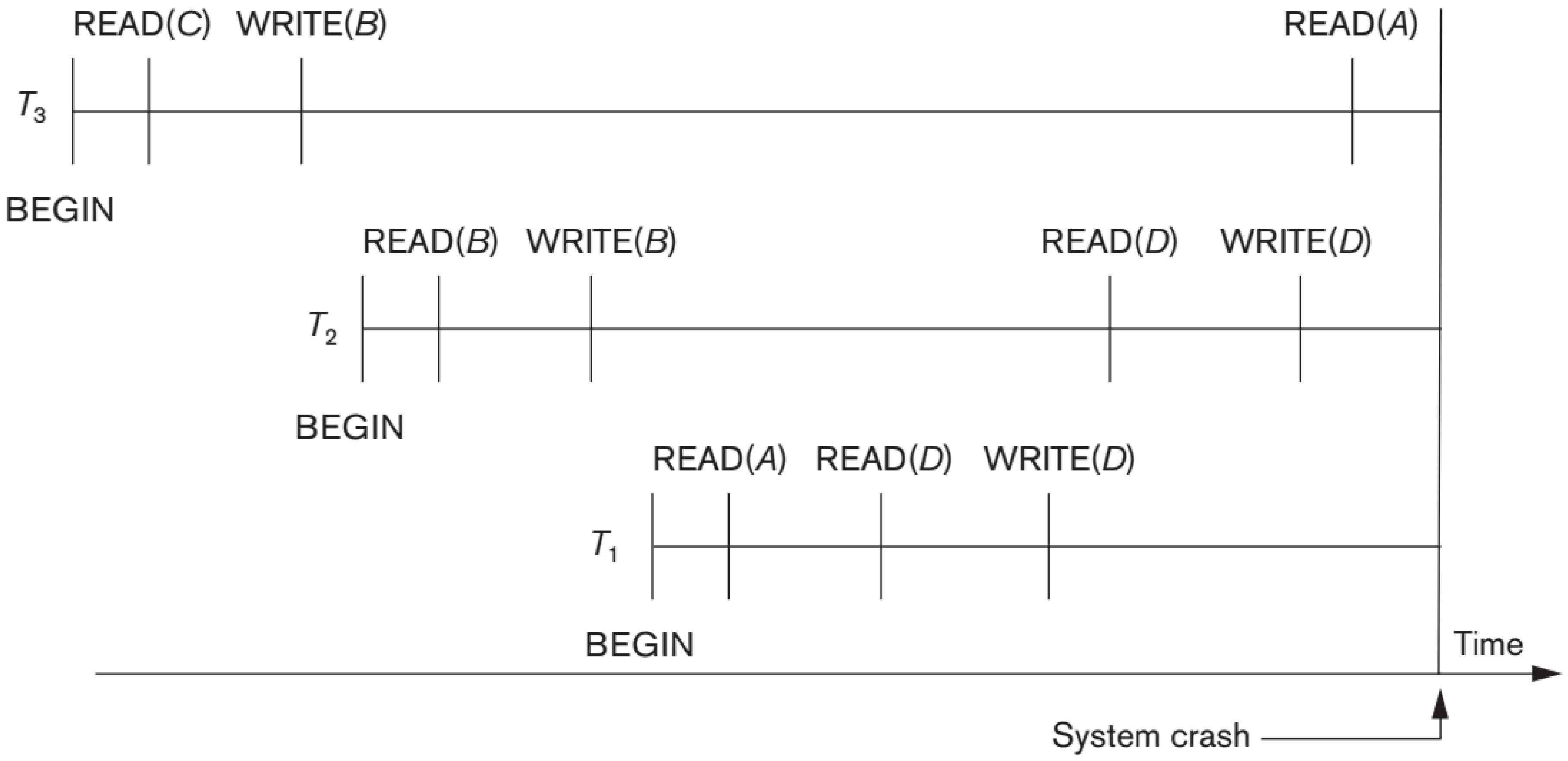
T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
write_item(A)

Transactions for illustrating cascading rollback (a process that never occurs in *strict* or *cascadeless* schedules)

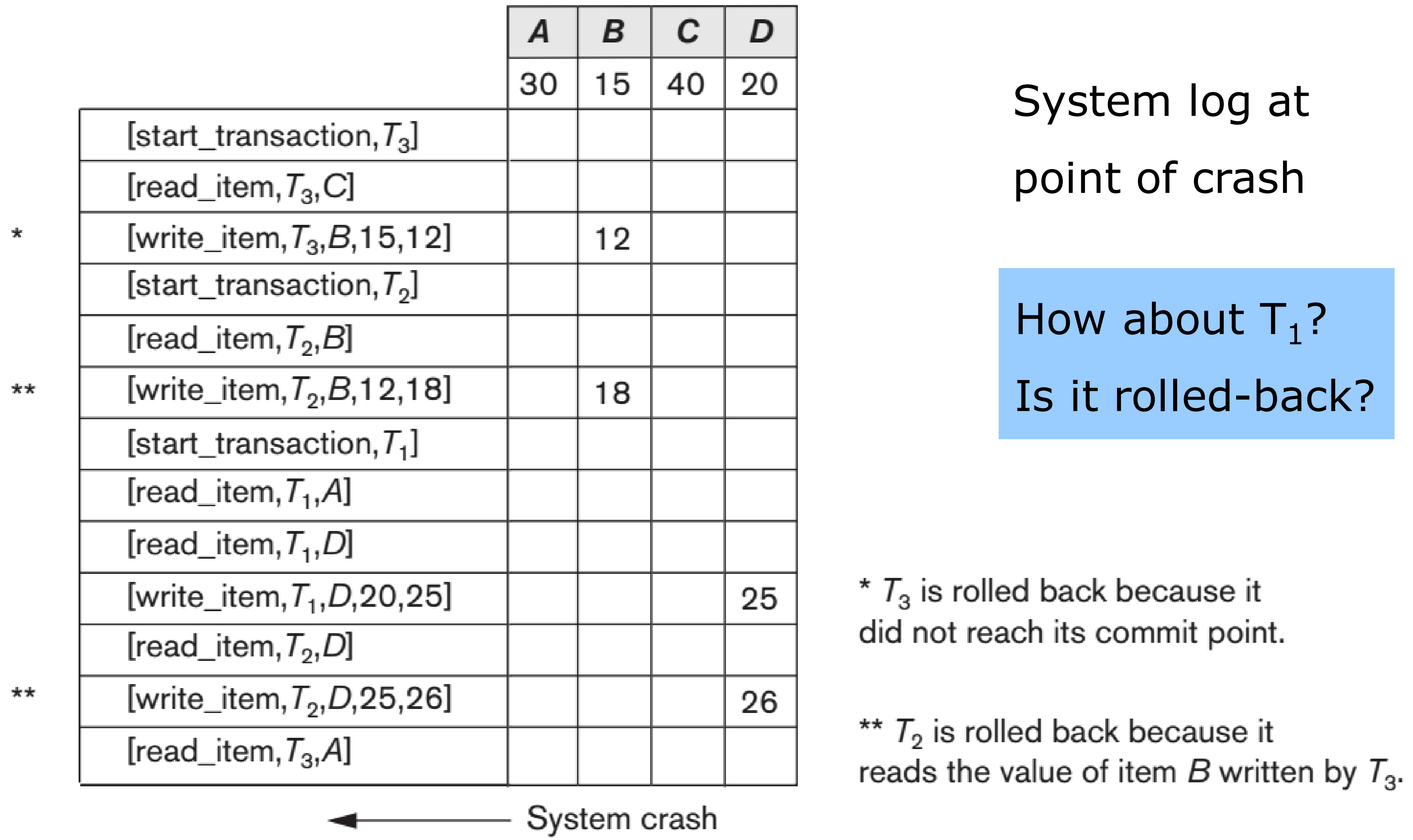
Figure 22.1, [1], pp. 820.

6.2. Recovery Concepts



Operations before the crash

6.2. Recovery Concepts



		A	B	C	D
		30	15	40	20
	[start_transaction,T_3]				
	[read_item,T_3,C]				
*	[write_item,T_3,B,15,12]		12		
	[start_transaction,T_2]				
	[read_item,T_2,B]				
**	[write_item,T_2,B,12,18]		18		
	[start_transaction,T_1]				
	[read_item,T_1,A]				
	[read_item,T_1,D]				
	[write_item,T_1,D,20,25]				25
	[read_item,T_2,D]				
**	[write_item,T_2,D,25,26]				26
	[read_item,T_3,A]				

$\leftarrow$ System crash

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item B written by T_3.

System log at point of crash

How about T_1?

Is it rolled-back?

Practical recovery methods *guarantee cascadeless or strict* schedules.
→ No need to record any *read_item* operations in the log

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

- The idea behind deferred update (no-steal) is to *defer or postpone* any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.
- During transaction execution, the updates are recorded *only in the log* and *in the cache buffers*.
- ***After*** the transaction reaches its *commit point* and the *log is force-written to disk*, the updates are recorded in the database.

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

- **Deferred Update (No-Undo/Redo)**

  1. A transaction cannot change the database on disk until it reaches its commit point; hence all buffers that have been changed by the transaction must be pinned (*not flushed to disk*) until the transaction commits (this corresponds to a *no-steal policy*).
  2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

  After rebooting from a failure, the log is used to redo all the committed transactions affected by this failure. No undo is required because no AFIM is flushed to the disk before a transaction commits.

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

Deferred update in a single-user system

There is no concurrent data sharing in a single user system. The data update goes as follows:

The algorithm RDU_S uses a REDO procedure for redoing certain *write_item* operations:

PROCEDURE RDU_S: use two lists of transactions: the committed transactions since the last checkpoint, and the active transaction. Apply the REDO operation to all the *write_item* operations of the committed transactions from the log in the order in which they are written to the log. Restart the active transaction.

The REDO procedure:

REDO(*write_item*): Redoing a *write_item* operation consisting of examining its log entry [*write_item*, *T*, *X*, *new_value*] and setting the value of *X* in the database to *new_value*, which is the after image (AFIM).

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

Deferred Update in a single-user system

(a)

T_1	T_2
read_item (A)	read_item (B)
read_item (D)	write_item (B)
write_item (D)	read_item (D)
	write_item (D)

(b)

[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit T_1]
[start_transaction, T_2]
[write_item, T_2, B, 10]
[write_item, T_2, D, 25] ← system crash

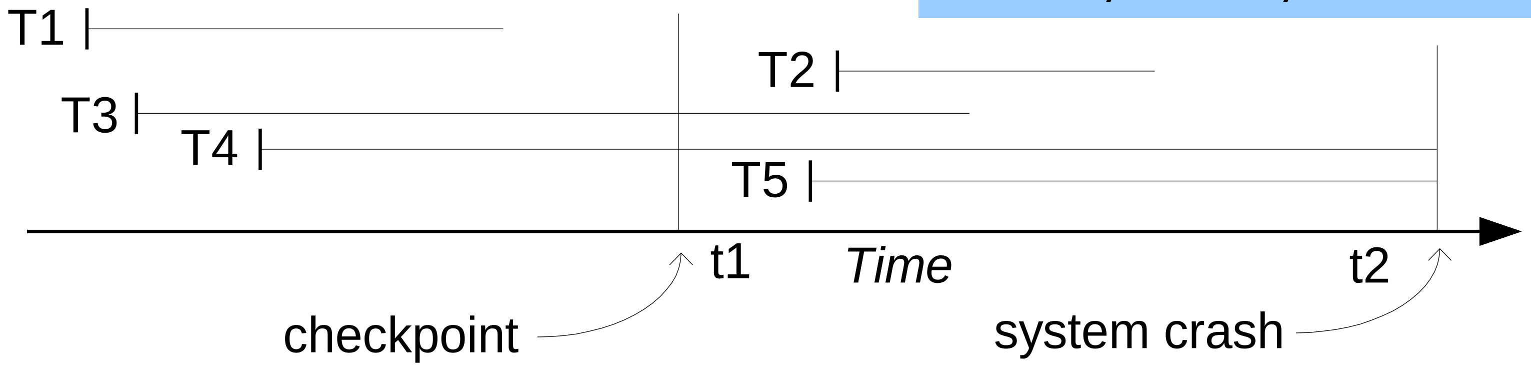
The [write_item, …] operation of T_1 is redone.
T_2 log entries are ignored by the recovery manager. (T_2 is not committed.)

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

Deferred Update with concurrent users

This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery, transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.

What should be done for recovery after system crash?



Recovery in a multiuser environment.

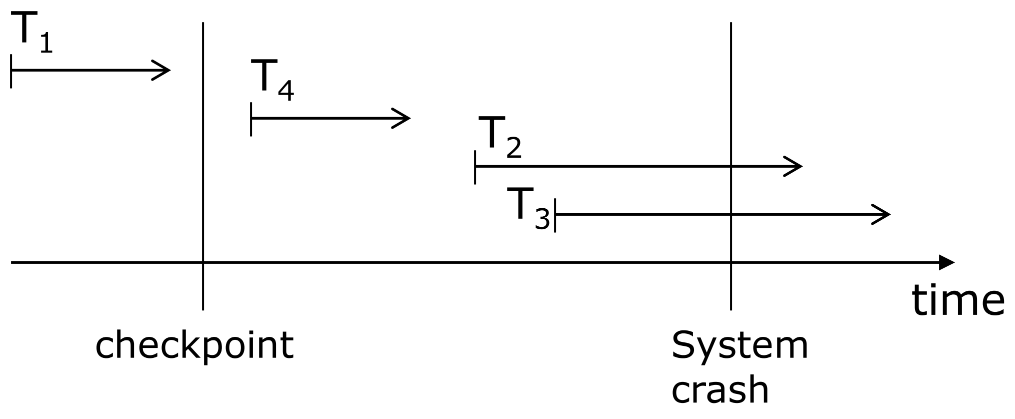
6.3. NO-UNDO/REDO Recovery Based on Deferred Update

Deferred Update with concurrent users

(a)

T_1	T_2	T_3	T_4
read_item (A)	read_item (B)	read_item (A)	read_item (B)
read_item (D)	write_item (B)	write_item (A)	write_item (B)
write_item (D)	read_item (D)	read_item (C)	read_item (A)
	write_item (D)	write_item (C)	write_item (A)

(b) [start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25] ← system crash



T_2 and T_3 are ignored because they did not reach their commit points.
T_4 is redone because its commit point is after the last checkpoint.

6.3. NO-UNDO/REDO Recovery Based on Deferred Update

Deferred Update with concurrent users

Two tables are required for implementing this protocol:

1. **Active table**: All active transactions are entered in this table.
2. **Commit table**: Transactions to be committed are entered in this table.

During recovery, all transactions of the **commit** table are redone and all transactions of **active** tables are ignored since none of their AFIMs reached the database. It is possible that a **commit** table transaction may be **redone** twice but this does not create any inconsistency because a redo operation is "**idempotent**", that is, one redo operation for an AFIM is equivalent to multiple redo operations for the same AFIM.

6.4. Recovery Techniques Based on Immediate Update

- Immediate Update
  - When a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point.
  - It is *not a requirement* that every update be applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.
    - UNDO/NO-REDO, UNDO/REDO
  - These methods follow a ***steal*** strategy for deciding when updated main memory buffers can be written back to disk.

6.4. Recovery Techniques Based on Immediate Update

Recovery Techniques Based on Immediate Update

Undo/No-redo Algorithm

In this algorithm, AFIMs of a transaction are flushed to the database disk under WAL before it commits.

For this reason, the recovery manager **undoes** all uncommitted transactions during recovery. It is possible that a transaction might have completed execution and ready to commit but this transaction has not yet committed and is also **undone**.

No committed transaction is **redone**.

6.4. Recovery Techniques Based on Immediate Update

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Single-user environment)

Recovery schemes of this category apply **undo** and also **redo** for recovery. In a single-user environment, no concurrency control is required but a log is maintained under WAL. Note that at any time, there will be one transaction in the system and it will be either in the commit table or in the active table.

The recovery manager performs:

1. **Undo** of a transaction if it is in the **active** table.
2. **Redo** of a transaction if it is in the **commit** table.

6.4. Recovery Techniques Based on Immediate Update

Recovery Techniques Based on Immediate Update

Undo/Redo Algorithm (Multiuser environment)

Recovery schemes of this category applies undo and also redo to recover the database from failure. In concurrent execution environment, a concurrency control is required and log is maintained under WAL. Commit table records transactions to be committed and active table records active transactions. To minimize the work of the recovery manager, checkpointing is used.

The recovery performs:

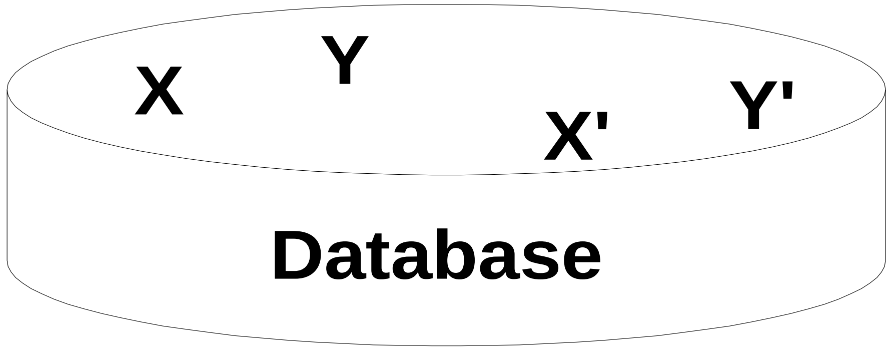
1. **Undo** of a transaction if it is in the active table.
2. **Redo** of a transaction if it is in the commit table.

6.5. Shadow Paging

- This recovery technique is based on shadow paging with shadow directory and current directory.
  - Shadow directory contains the most recent or current entry points to database pages on disk that are never modified.
  - Current directory contains the entry points to database pages on disk used and modified during transaction execution.
    - When a write_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten.*
    - For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

6.5. Shadow Paging

The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.



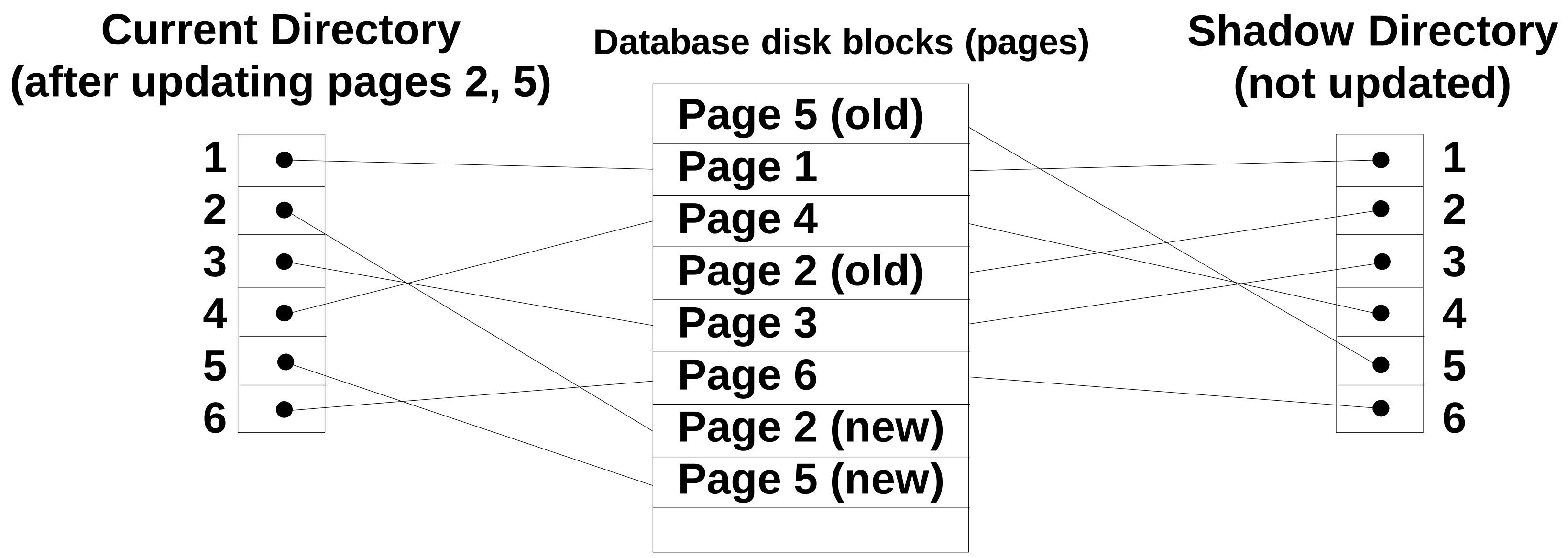
X and Y: Shadow copies of data items (BFIM)
X' and Y': Current copies of data items (AFIM)

NO-UNDO/NO-REDO for recovery

- NO-UNDO: discard Current Directory
- NO-REDO: discard Shadow Directory

6.5. Shadow Paging

To manage access of data items by concurrent transactions two directories (current and shadow) are used. The directory arrangement is illustrated below. Here a page is a data item.



Current Directory (after updating pages 2, 5)
Database disk blocks (pages)
Shadow Directory (not updated)
1
2
3
4
5
6
Page 5 (old)
Page 1
Page 4
Page 2 (old)
Page 3
Page 6
Page 2 (new)
Page 5 (new)

Need more consideration:

- Complex storage management
- Garbage collection
- Atomic migration between current and shadow directories

6.6. The ARIES Recovery Algorithm

- A recovery algorithm is used in many relational database-related products of IBM.
- ARIES uses a steal/no-force approach for writing.
  - UNDO/REDO
- Three concepts
  - WAL
  - Repeating history during redo
  - Logging changes during undo
- Three steps
  - Analysis
  - REDO
  - UNDO

6.6. The ARIES Recovery Algorithm

The ARIES Recovery Algorithm is based on:

1. WAL (Write Ahead Logging)
2. Repeating history during redo: ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred.

   Transactions that were uncommitted at the time of the crash (active transactions) are undone.
3. Logging changes during undo: prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

6.6. The ARIES Recovery Algorithm

The ARIES recovery algorithm consists of three steps:

1. **Analysis**: this step identifies the dirty (updated) pages in the buffer and the set of transactions active at the time of crash. The appropriate point in the log where redo is to start is also determined.
2. **Redo**: necessary redo operations are applied.
3. **Undo**: log is scanned backwards and the operations of transactions active at the time of crash are undone in reverse order.

The information needed for ARIES to accomplish its recovery procedure includes the *log*, the *Transaction Table*, and the *Dirty Page Table*. Additionally, *checkpointing* is used.

6.6. The ARIES Recovery Algorithm

The Log and Log Sequence Number (LSN)

A unique LSN is associated with every log record. LSN increases monotonically and indicates the disk address of the log record it is associated with. In addition, each data page stores the LSN of the latest log record corresponding to a change for that page.

A log record is written for:

(a) data update,

(b) transaction commit,

(c) transaction abort,

(d) undo (in this case, a compensating log record is written.),

(e) transaction end (an *end log record* is written.).

6.6. The ARIES Recovery Algorithm

The Log and Log Sequence Number (LSN)

A log record stores:

1. Previous LSN of that transaction: It links to the log record of each transaction. It is like a back pointer that points to the previous record of the same transaction.
2. Transaction ID
3. Type of log record.

For a *write* operation, the following additional information is logged:

4. Page ID for the page that includes the item
5. Length of the updated item
6. Its offset from the beginning of the page
7. BFIM of the item
8. AFIM of the item

6.6. The ARIES Recovery Algorithm

The Transaction table and the Dirty Page table

For efficient recovery, these tables are needed, maintained by the transaction manager, and rebuilt in the analysis phase of recovery when a crash occurs:

Transaction table: Contains an entry for each ***active*** transaction, with information such as *transaction ID*, *transaction status* and the *LSN of the most recent log record for the transaction*.

Dirty Page table: Contains an entry for each ***dirty page*** in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

6.6. The ARIES Recovery Algorithm

Checkpointing

1. Writes a *begin_checkpoint* record in the log
2. Writes an *end_checkpoint* record in the log. With this record, the contents of transaction table and dirty page table are appended to the end of the log.
3. Writes the LSN of the *begin_checkpoint* record to a special file. This special file is accessed during recovery to locate the last checkpoint information.

To reduce the cost of checkpointing and allow the system to continue to execute transactions, ARIES uses “fuzzy checkpointing”.

6.6. The ARIES Recovery Algorithm

1. **Analysis phase**: Start at the *begin_checkpoint* record and proceed to the *end_checkpoint* record, until reaching the end of the log. Access *transaction table and dirty page table* that are appended to the end of the log. During this phase, some other log records may be written to the *log and transaction table* may be modified. The analysis phase compiles the set of **redo** and **undo** to be performed.
2. **Redo phase**: Starts from the point in the log up to where all dirty pages have been flushed, and move forward to the end of the log. It determines this point by finding the smallest LSN, *M*, of all the pages in Dirty Page Table. Any change with LSN $\geq$ M is redone.
3. **Undo phase**: Starts from the *end of the log* (or the largest LSN of the active transaction) and proceeds backward while performing appropriate undo. For each undo, it writes a *compensating log record* in the log.

(a)

LSN	LAST-LSN	TRAN-ID	TYPE	PAGE-ID	Other Info.
1	0	T1	update	C	-----
2	0	T2	update	B	-----
3	1	T1	commit		-----
4	begin checkpoint				
5	end checkpoint				
6	0	T3	update	A	-----
7	2	T2	update	C	-----
8	7	T2	commit		-----

At time of checkpoint (b)

TRANSACTION TABLE

TRANSACTION ID	LAST LSN	STATUS
T1	3	commit
T2	2	in progress

DIRTY PAGE TABLE

PAGE ID	LSN
C	1
B	2

After the ANALYSIS phase (c)

(start from LSN 4 till the end)

TRANSACTION TABLE

TRANSACTION ID	LAST LSN	STATUS
T1	3	commit
T2	8	commit
T3	6	in progress

DIRTY PAGE TABLE

PAGE ID	LSN
C	1
B	2
A	6

The REDO phase: start from LSN 1 (the smallest in Dirty Page table) and reapply for LSNs 1, 2, 6, and 7.

The UNDO phase: start from LSN 6 (for uncommitted transactions in Transaction table) and proceed backward in the log.

6.7. Recovery in Multidatabase Systems

A multidatabase system is a special distributed database system where one node may be running a relational database system under Unix, another may be running an object-oriented database system under Windows, and so on. A transaction may run in a distributed fashion at multiple nodes, requiring access to multiple databases. Such a transaction is called multidatabase transaction.

In this execution scenario, the transaction commits only when all these multiple nodes agree to commit individually the part of the transaction they were executing.

This commit scheme is referred to as "*two-phase commit*" (2PC). If any one of these nodes fails or cannot commit the part of the transaction, then the transaction is aborted. Each node recovers the transaction under its own recovery protocol.

6.7. Recovery in Multidatabase Systems

- Each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs.
- To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism.
- A **global recovery manager**, or **coordinator**, is needed to maintain information needed for recovery, in addition to the **local recovery managers** and the information they maintain (log, tables).
  - The coordinator usually follows the **two-phase commit protocol**.

6.8. Database Backup and Recovery from Catastrophic Failures

- *Non*catastrophic failures: *log-based recovery* or *shadow paging* to bring the database back to a consistent state
- Catastrophic failures: ***database backup***, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices
  - Database, log
  - Frequency
  - Human resource

Summary

- Recovery from transaction failures in *both single and multiuser environments* brings a database to a consistent state.
  - Log-based recovery methods
  - Shadow paging
  - Backup
- Undo and redo operations: idempotent
- Transaction rollback and roll-forward
  - ACID properties: Atomic, Durability

Summary

- System log
  - BFIM (undo), AFIM (redo)
- Checkpoints and fuzzy checkpointing
- When to update vs. Where to update
  - Deferred update vs. immediate update
  - In-place updating vs. shadowing
- Write-Ahead Logging
- When a page from the database cache can be written to disk
  - Steal/No-Steal
  - Force/No-Force

Summary

- Recovery based on deferred update
  - NO-UNDO/REDO
- Recovery based on immediate update
  - UNDO/REDO
  - UNDO/NO-REDO
- Shadow paging: shadow/current directories
  - NO-UNDO/NO-REDO
- ARIES with the log, transaction table, and dirty page table in three phases: analysis, redo, and undo
  - UNDO/REDO

Summary

- Recovery in a multidatabase system
  - Multidatabase transaction
  - Two-phase commit
- Recovery from catastrophic failures
  - Database backup
    - Database, log
    - Frequency
    - Human resource

Chapter 6: Database Recovery Techniques

Check for Understandings

- 6.1. Describe some non-catastrophic failures and catastrophic failures. Introduce some approaches to deal with these failures.
- 6.2. State the purposes of database recovery from transaction failures.
- 6.3. What are transaction commit points? Why are they important?
- 6.4. What are checkpoints? Why are they important? What is fuzzy checkpointing?
- 6.5. Distinguish deferred update with immediate update.

Check for Understandings

- 6.6. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?
- 6.7. Describe UNDO and REDO operations. Distinguish them from transaction rollback and roll-forward.
- 6.8. Describe write-ahead logging.
- 6.9. Describe the policies when a page from the database cache can be written to disk: steal/no-steal, force/no-force.

Check for Understandings

- 6.10. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?
- 6.11. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update? What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update?

Check for Understandings

- 6.12. Describe the shadow paging recovery technique. Under what circumstances does it not require a log? Why is it called a NO-UNDO /NO-REDO method?
- 6.13. Describe the three phases of the ARIES recovery method.
- 6.14. Describe the two-phase commit protocol for multidatabase transactions.
- 6.15. Discuss how disaster recovery from catastrophic failures is handled.

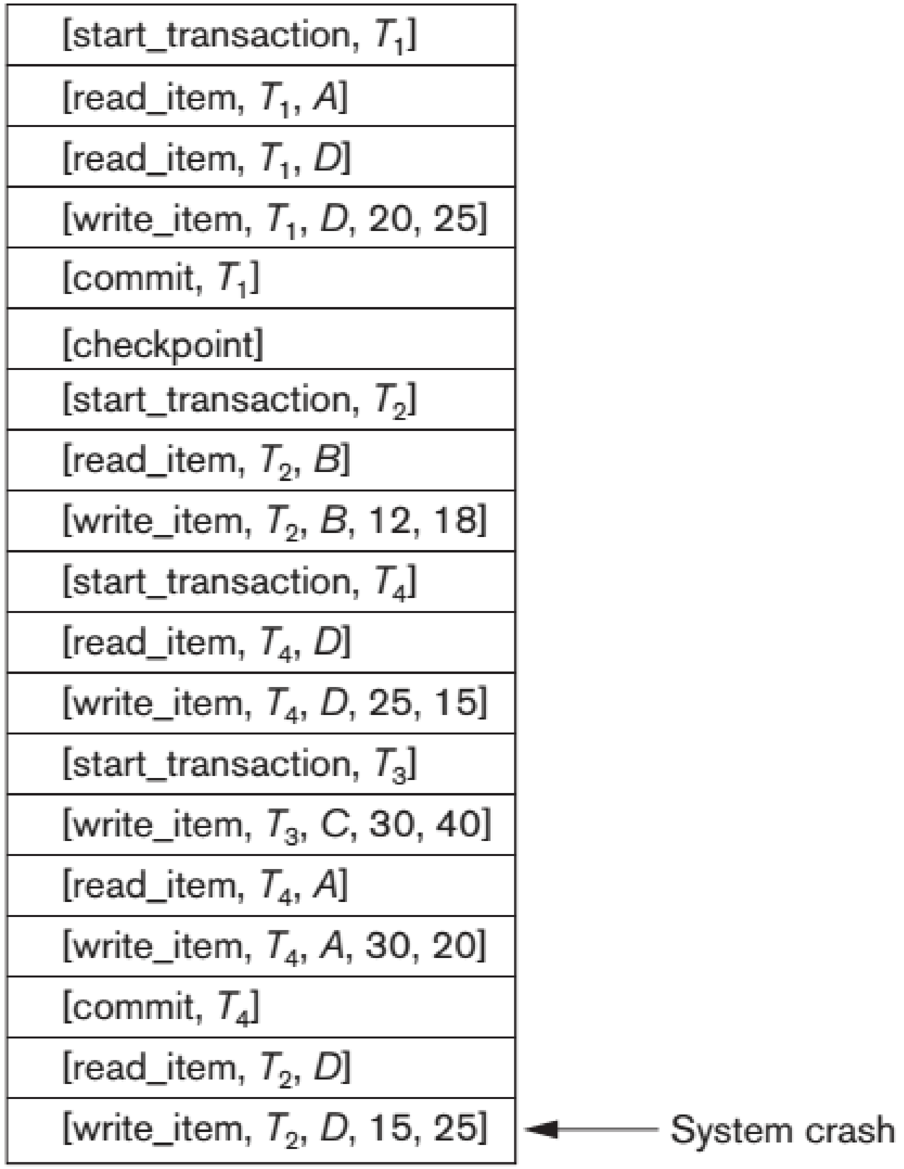
Check for Understandings

- 6.16. Given a system log as follows. Suppose that:
- (1). the system crashes before the **[read_item, *T3*, *A*]** entry is written to the log
- (2). the system crashes before the **[write_item, *T2*, *D*, 25, 26]** entry is written to the log.
- Describe the recovery process for each case.

	A	*B*	*C*	*D*
	30	15	40	20
[start_transaction, T_3]				
[read_item, T_3, C]				
[write_item, T_3, B, 15, 12]		12		
[start_transaction, T_2]				
[read_item, T_2, B]				
[write_item, T_2, B, 12, 18]		18		
[start_transaction, T_1]				
[read_item, T_1, A]				
[read_item, T_1, D]				
[write_item, T_1, D, 20, 25]				25
[read_item, T_2, D]				
[write_item, T_2, D, 25, 26]				26
[read_item, T_3, A]				

Check for Understandings

- 6.17. Given the log corresponding to a particular schedule at the point of a system crash for four transactions *T*1, *T*2, *T*3, *T*4.
- Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and whether any cascading rollback takes place.
  - (1). use the deferred update protocol with checkpointing
  - (2). use the *immediate update protocol* with checkpointing

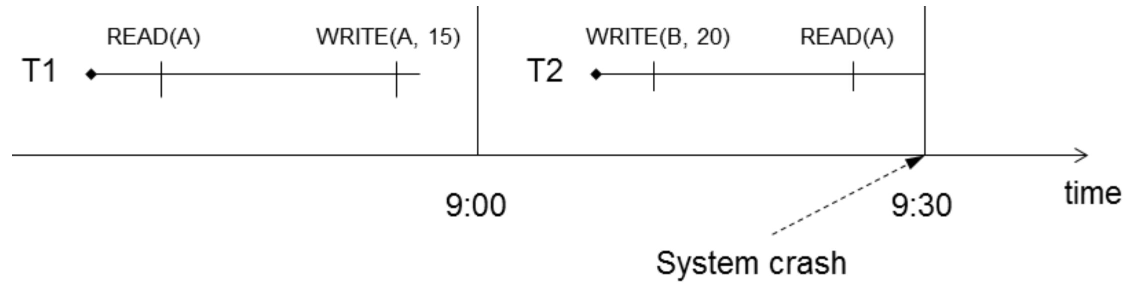


[start_transaction, T_1]	
[read_item, T_1, *A*]	
[read_item, T_1, *D*]	
[write_item, T_1, *D*, 20, 25]	
[commit, T_1]	
[checkpoint]	
[start_transaction, T_2]	
[read_item, T_2, *B*]	
[write_item, T_2, *B*, 12, 18]	
[start_transaction, T_4]	
[read_item, T_4, *D*]	
[write_item, T_4, *D*, 25, 15]	
[start_transaction, T_3]	
[write_item, T_3, *C*, 30, 40]	
[read_item, T_4, *A*]	
[write_item, T_4, *A*, 30, 20]	
[commit, T_4]	
[read_item, T_2, *D*]	
[write_item, T_2, *D*, 15, 25]	← System crash

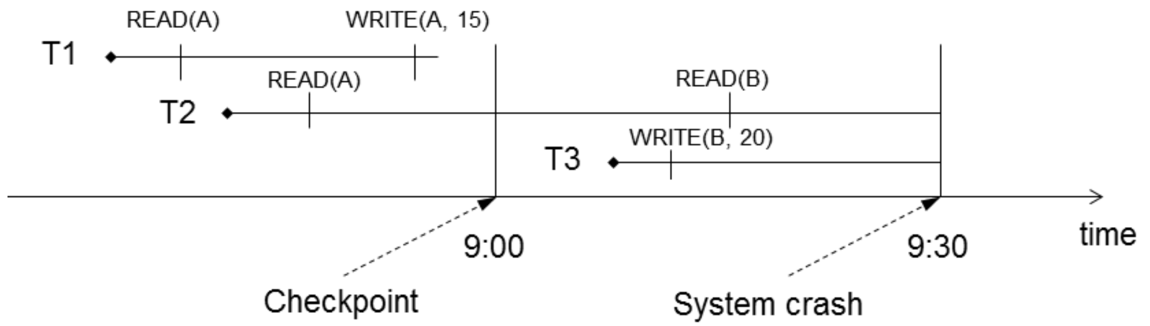
Check for Understandings

- 6.18. Given the execution of the transactions as follows. For each execution, describe the recovery process after the system crashes using the deferred update protocol with checkpointing. Repeat the question with the immediate update protocol.

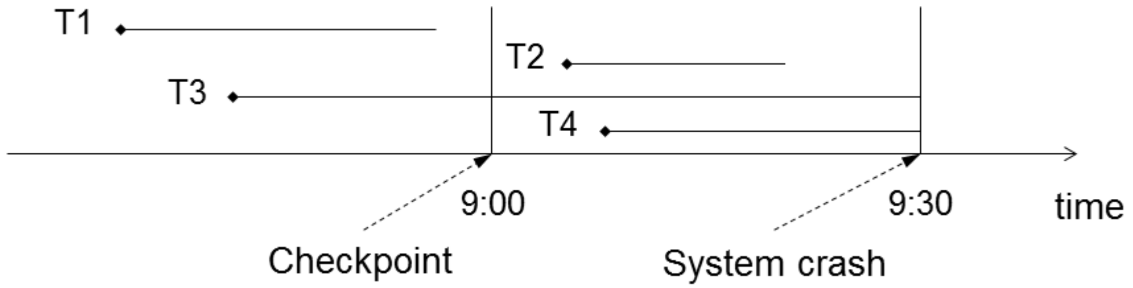
(a)



(b)



(c)



Check for Understandings

- 6.19. Given the log when the system crashes, the Transaction and Dirty Page tables at the checkpoint as follows. Describe the recovery process using ARIES.

LSN	LAST_LSN	TRAN_ID	TYPE	PAGE_ID	…
1	0	T1	update	C	…
2	1	T1	update	B	…
3	0	T2	update	C	…
4	begin_checkpoint				
5	end_checkpoint				
6	2	T1	commit		…
7	0	T3	update	A	…

TRANSACTION TABLE			DIRTY PAGE TABLE	
TRANSACTION ID	LAST LSN	STATUS	PAGE ID	LSN
T1	2	in progress	C	1
T2	3	in progress	B	2

Check for Understandings

- 6.20. Given the log when the system crashes as follows. Describe the recovery process using: (1). deferred update, (2). immediate update, and (3). ARIES methods.

LSN	LAST_LSN	TRAN_ID	TYPE	PAGE_ID	...
0	begin_checkpoint				
1	end_checkpoint				
2	0	T1	update	P5	...
3	0	T2	update	P3	...
4	3	T2	commit		
5	4	T2	end		
6	0	T3	update	P3	...
7	2	T1	update	P2	...