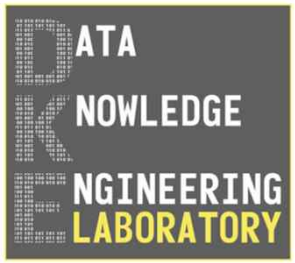




# TRANSACTION PROCESSING II

## Chapter 22

## - Database Recovery Techniques

Prof. Young-Kyoon Suh

# Introduction

- Recovery concepts
  - *Deferred update vs. Immediate update*
  - *In-place updates vs. shadow updates*
  - *Write-ahead logging protocol*
  - *Checkpointing*
- Recovery algorithms
  - Via deferred update or immediate update
  - ARIES
- Recovery techniques are often intertwined with the CC mechanisms
  - Certain recovery techniques best used with specific concurrency control methods

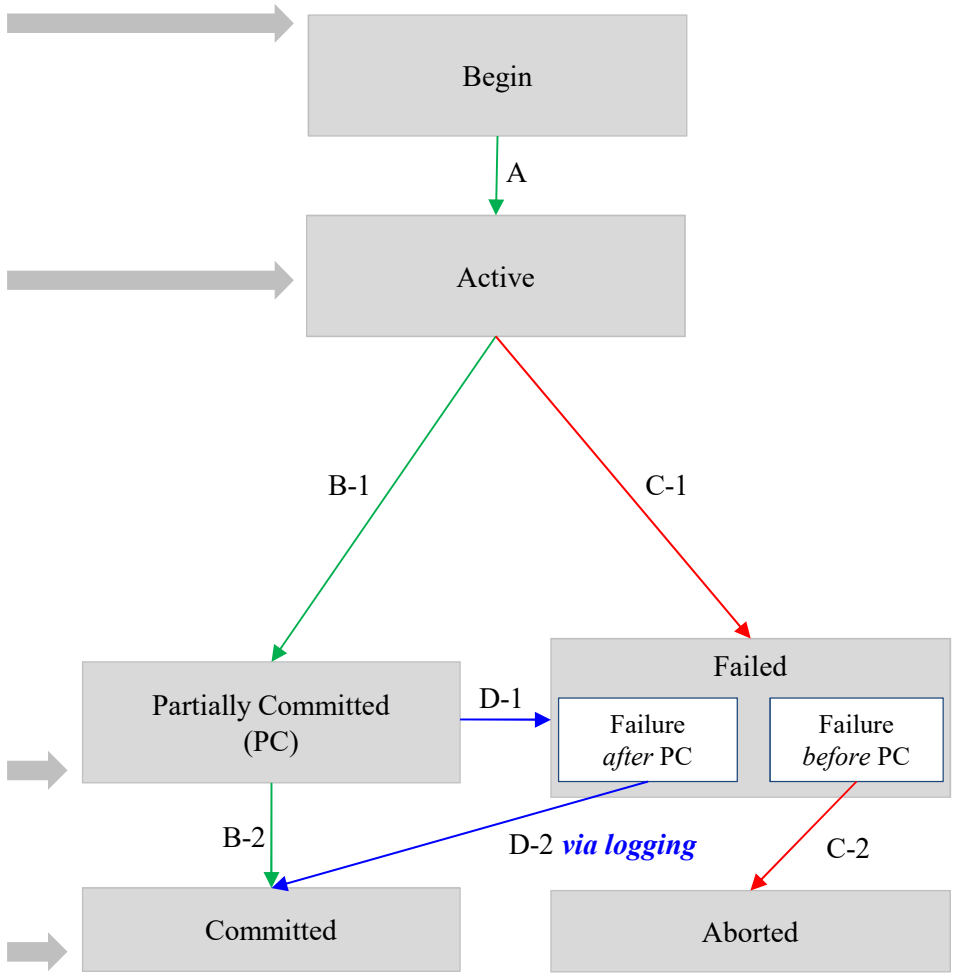
# Introduction (Cont'd)

- Recovery process restores database to most recent consistent state before time of failure.
  - As a reminder, information kept in system log.
- Typical recovery strategies
  1. Restore backed-up copy of database
     - Best in cases of *catastrophic* failure (치명적 실패) like disk crash
  2. Identify any changes that may cause inconsistency
     - Best in cases of *noncatastrophic* failure (비치명적 실패?)
     - Some operations may require redo

# Transaction & Recovery

```
BEGIN TRANSACTION
① /* 박지성 계좌를 읽어온다 */
② /* 김연아 계좌를 읽어온다 */
   /* 잔고 확인 */

③ /* 예금인출 박지성 */
UPDATE Customer
SET balance=balance-10000
WHERE name='박지성';

④ /* 예금입금 김연아 */
UPDATE Customer
SET balance=balance+10000
WHERE name='김연아';

COMMIT /* Partially Committed */
⑤ /* 박지성 계좌를 기록한다 */
⑥ /* 김연아 계좌를 기록한다 */
```

(a) Balance Transfer Transaction




(b) Transaction State Diagram

# Reminder: Logging

- Writing all database operations of transactions into log disk
  - Safe, permanently recorded
- Used to recover from failure
- Types of "Log Records":
  - [`start_transaction`, $T$], [`read_item`, $T$, $X$], [`write_item`, $T$, $X$, *old_value*, *new_value*] [`commit`, $T$], [`abort`, $T$]
    - $T$: a unique transaction identifier generated automatically
- Example of log records on the previous example:

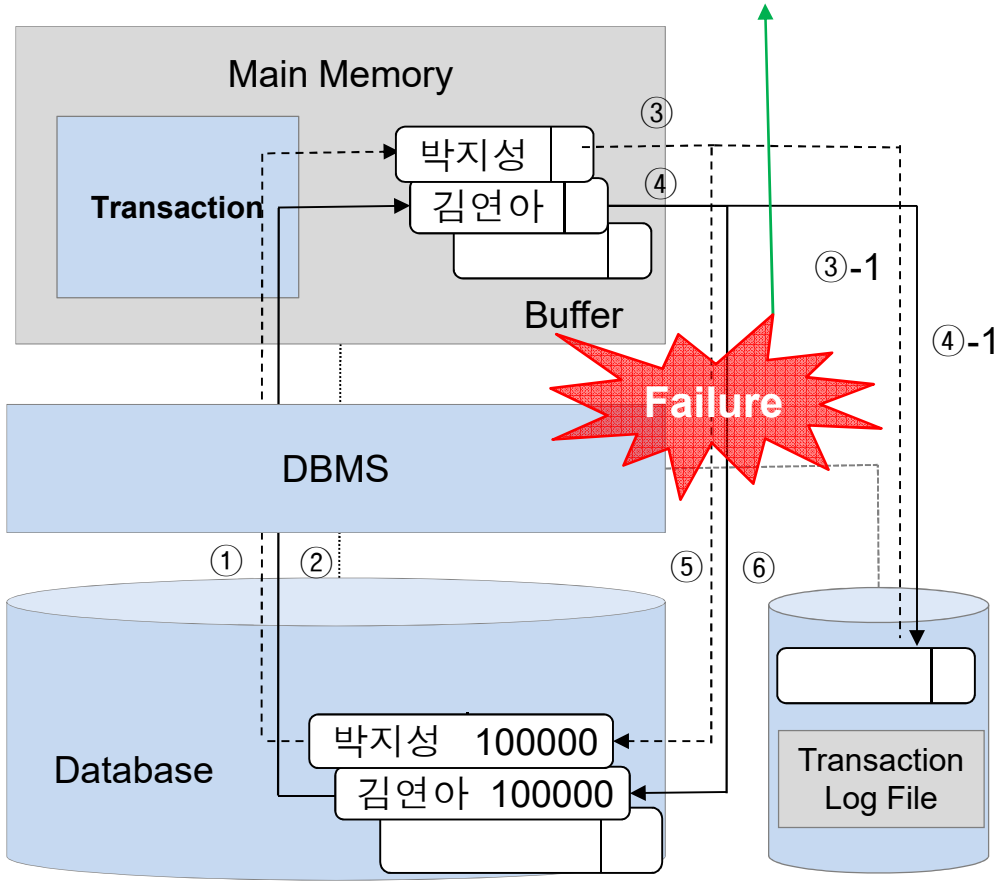
```
<T1, START>
<T1, UPDATE, Customer(박지성).balance, 100000, 90000>
<T1, UPDATE, Customer(김연아).balance, 100000, 110000>
<T1, COMMIT>
```

# Reminder: Logging (Cont'd)

```
BEGIN TRANSACTION
① /* 박지성 계좌를 읽어온다 */
② /* 김연아 계좌를 읽어온다 */
   /* 잔고 확인 */

③ /* 예금인출 박지성 */
UPDATE Customer
SET balance=balance-10000
WHERE name='박지성';

④ /* 예금입금 김연아 */
UPDATE Customer
SET balance=balance+10000
WHERE name='김연아';

COMMIT /* Partially Committed */
⑤ /* 박지성 계좌를 기록한다 */
⑥ /* 김연아 계좌를 기록한다 */
```

(a) Balance Transfer Transaction

*Can recover the database state via the log file!*




(b) Transaction Processing Process

# Recovery Concepts

- Two main policies *for recovery from noncatastrophic failure*:
  - ***Deferred update*** techniques (지연 갱신 기법):
    - Don't physically update the database until after transaction commits
    - UNDO for an uncommitted transaction is NOT necessary; but REDO for a committed transaction may be needed; No-UNDO/REDO algorithm
  - ***Immediate update*** techniques (즉시 갱신 기법):
    - Database may be updated by some operations of a transaction before it reaches its commit point
      - Operations *must be recorded* in the log on disk, by "force-writing" before being applied to the database on disk, making recovery possible via the log
    - Both UNDO and REDO may be required during recovery; UNDO/REDO alg.
  - **UNDO** and **REDO**: ***idempotent***
    - No matter how many times they are done, equivalent to executing just once.
    - The result of recovery from a system crash *during recovery* = the result of recovering *when there's no crash during recovery*.

# Recovery Concepts (Cont'd)

- Two main strategies *for flushing (replacing) a modified buffer back to disk*:
  - ***In-place updating***:
    - Writes the buffer to the *same original disk location*
    - Overwrites old values of any changed data items: only a single copy
    - Needed to maintain a log for recovery
  - ***Shadowing***:
    - Writes an updated buffer at a different disk location, to maintain multiple versions of data items
    - Not used in practice; not strictly needed to maintain a log for recovery
- **BF**ore-**IM**age (**BFIM**): the old value of data item
- **AF**ter-**IM**age (**AFIM**): the new value of data item
  - If shadowing is used, then both the BFIM and AFIM can be kept on disk.

# Recovery Concepts (Cont'd)

- **W**rite-**A**head **L**ogging (**WAL**) protocol (쓰기-전 로깅 규약)
  - Used in a recovery mechanism via a log for in-place updates
  - Ensure the BFIM is recorded in the log
  - The log entry must be flushed to disk *before* the BFIM is overwritten with the AFIM in the database on disk.
  - Necessary for UNDO operation if needed
- Two types of log entry information for a write command:
  - UNDO-type log entries
    - Includes the BFIM of the item (for UNDO)
  - REDO-type log entries
    - Includes the AFIM of the item (for REDO)

# Recovery Concepts (Cont'd)

- Recovery terminologies: *steal/no-steal*, *force/no-force*
  - Specify rules governing when a page from the database cache can be written to disk

| Recovery methods | Notes |
|---|---|
| *No-steal* | If a cache buffer page is updated by a transaction, it can **NOT** be written back to disk *before* a transaction commits. |
| ***Steal*** | Writing an updated buffer to disk is *allowed* before a transaction commits |
| ***No-force*** | All pages updated by a transaction are *NOT immediately* written to disk before the transaction commits |
| *Force* | All pages updated by a transaction are *immediately* written to disk before the transaction commits |

- Typical DBMSes: use a **steal/no-force**; Why? It (i) avoids for very large buffer space and (ii) reduces disk I/O operations for heavily updated pages.

Recovery Concepts (Cont'd):

# Checkpoints in the System Log

- ***Checkpoint*** (검사점): another type of entry in the log
  - [`checkpoint`, *list of active transactions*]
    - *Active transactions*: that has started but yet committed
  - Written into the log periodically, at the point when database on disk gets synchronized with modified buffers. Why necessary?
- Steps taken for recording a checkpoint
  1. Suspend execution of all transactions temporarily
  2. Force-write all main memory buffers that have been modified on disk
  3. Write a checkpoint record to the log, and force-write the log to disk
  4. Resume executing transactions
  - After Step 2, additional information may be recorded due to rollback:
    - A list of active transaction ids, and
    - The locations of the first and most recent records in the log for each active transaction

Recovery Concepts (Cont'd):

# *Fuzzy Checkpointing*

- The DBMS can resume transaction processing after a `begin_checkpoint` record is written to the log
  - Without waiting for flushing all buffers to disk at Step 2

  (Note that Step 1 is very pricy.)
- Previous checkpoint record maintained until `end_checkpoint` record is written.
  - The DBMS maintains a file on disk containing a pointer to the valid checkpoint.
    - That is, the pointer points to the previous checkpoint record in the log.
- Once Step 2 is done, then that pointer is updated to point to the new checkpoint

# NO-UNDO/REDO RECOVERY BASED ON DEFERRED UPDATE

Chapters 22.2

# *Deferred Update* (지연 갱신): No-Steal Policy

- Key idea
  - To defer or postpone any “actual” updates to the database on disk
    - Until $T$ completes its execution successfully and reaches its commit point
- Why **REDO** log entries only?
  - While $T$ is executed, updates are recorded “only in the log” and “in the cache buffers”.
  - The updates are recorded after (i) $T$ reaches its commit point and (ii) its log flushes to disk
  - If failure occurs after the commit point, may REDO updates from the log.
  - If $T$ fails before the commit point, **NO** effect on the database; thus, no UNDO log entries.
- Impractical although recovery process gets simpler
  - Why? Easily to run out of buffer space; many buffers will be pinned; unable to be replaced until $T$ reaches its commit point
  - Works only for shorter transactions

# Deferred Update (Cont'd)

- **Deferred update protocol** works in the following way:

  Step 1. $T$ cannot change the database on disk until reaching its commit point

  - All buffers changed by $T$ must be pinned until $T$ commits

  Step 2. A transaction doesn't reach its commit point *until*

  ($i$) All its REDO-type log entries are recorded in log, and

  ($ii$) Log buffer is force-written to disk

  - Restatement of the WAL protocol

# Deferred Update – Recovery Example (Cont'd)

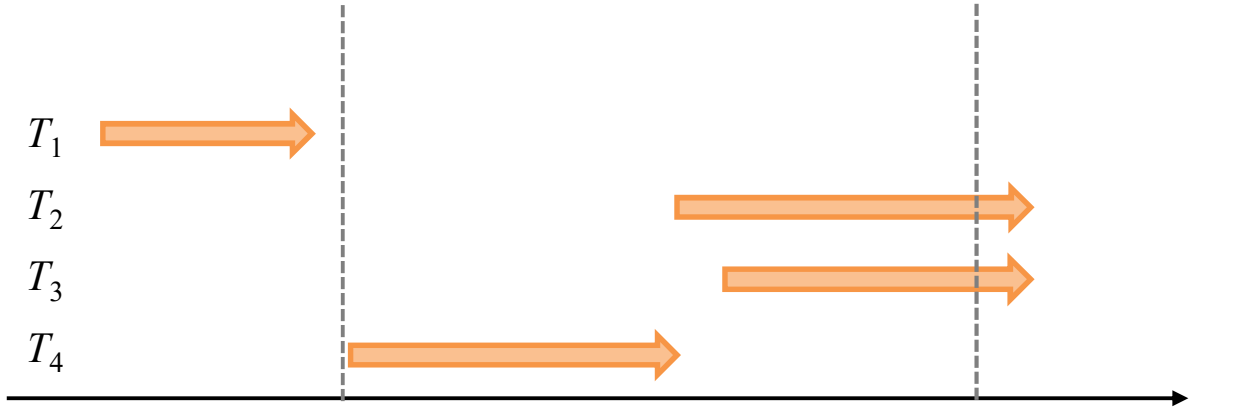
| $T_1$ |
|---|
| read_item(A) |
| read_item(D) |
| write_item(D) |

| $T_2$ |
|---|
| read_item(B) |
| write_item(B) |
| read_item(D) |
| write_item(D) |

| $T_3$ |
|---|
| read_item(A) |
| write_item(A) |
| read_item(C) |
| write_item(C) |

| $T_4$ |
|---|
| read_item(B) |
| write_item(B) |
| read_item(A) |
| write_item(A) |

| [System Log] |
|---|
| [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*

| UNDO | REDO | No Work Required |
|---|---|---|
| - | $T_4$ | $T_1$ (done), $T_2$(ignored), $T_3$(ignored) |

# RECOVERY TECHNIQUE BASED ON IMMEDIATE UPDATE

Chapters 22.3

# *Immediate Update* (즉시 갱신): Steal Policy

- Key idea
  - When $T$ issues an update command, the database on disk *can* be updated *immediately*, with no need to wait for $T$ to reach commit point
    - But **NOT** a requirement that every update be immediate
- Why **UNDO** log entries (as well as **REDO** log entries)?
  - To cancel the effect of updates by a *failed transaction* (atomicity)
- Two main recovery algorithms

  (1) **UNDO/REDO**: steal/no-force strategy; more commonly used in practice
    - $T$ is allowed to commit before all its changes are written to database

  (2) UNDO/NO-REDO: steal/force strategy
    - All updates of $T$ must be recorded in the database on disk *before $T$ commits*.
    - No need of REDO for committed transactions

# Immediate Update(UNDO/REDO)– Recovery Example (Cont'd)

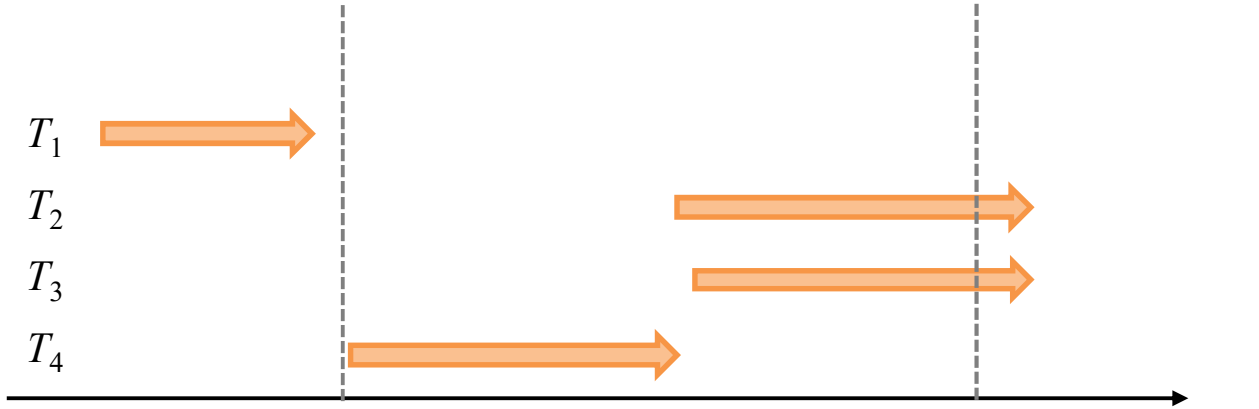
| $T_1$ |
|---|
| read_item(A) |
| read_item(D) |
| write_item(D) |

| $T_2$ |
|---|
| read_item(B) |
| write_item(B) |
| read_item(D) |
| write_item(D) |

| $T_3$ |
|---|
| read_item(A) |
| write_item(A) |
| read_item(C) |
| write_item(C) |

| $T_4$ |
|---|
| read_item(B) |
| write_item(B) |
| read_item(A) |
| write_item(A) |

| [System Log] |
|---|
| [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] |




| UNDO | REDO | No Work Required |
|---|---|---|
| $T_2$, $T_3$ | $T_4$ | $T_1$ |

*System Crash*

# THE ARIES RECOVERY ALGORITHM

## Chapters 22.5

ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

[from wiki]

C. MOHAN
IBM Almaden Research Center
and
DON HADERLE
IBM Santa Teresa Laboratory
and
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

Authors' addresses: C Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.



ACM Transactions on Database Systems, Vol 17, No. 1, March 1992, Pages 94–162

# **ARIES**: **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics

- Used in many IBM relational database products
- Uses a **steal**/**no-force** approach for writing
- Based on three concepts:
  1) **WAL protocol**
  2) **Repeating history** (during **REDO**)
     - Means retracing all database system actions prior to crash
       - To reconstruct database state when crash occurred
  3) **Logging during UNDO**
     - Prevents ARIES from repeating the “completed undo” operations if failure occurs during recovery
       - Otherwise, a restart of the recovery process happens.

# The ARIES Algorithm (Cont'd)

- Consists of three main steps: *analysis*, *REDO*, and *UNDO*.
- **Analysis step**
  - Identifies (i) dirty (updated) pages in the buffer and (ii) the set of active transactions at the time of crash
  - Determines appropriate start point in the log for the REDO operation
- **REDO step**
  - Actually *reapplies updates from the log* to the database *until the end* of the log is reached.
    - In general, REDO is applied to committed transactions but NOT in ARIES.
  - *Only necessary REDO operations* are applied during recovery.
    - ARIES will determine whether the REDO operation to be redone has actually been applied to the database and hence doesn't need to be reapplied,
      - Using information stored by ARIES and in the data pages

# The ARIES Algorithm (Cont'd)

- **UNDO step**
  - The log is scanned backward.
  - Operations of transactions that were active at the time of crash are undone in reverse order.
- Every log record has an associated **log sequence number** (LSN):
  - Monotonically increasing
  - Indicates the address of the log record on disk
  - Corresponds to a specific change (action) of some transaction
- Log record type: `write`, `commit`, `abort`, `undo` (of an update), `end`
  - When an update is undone, a *compensation log record* (CLR) is written.
  - When $T$ ends, an `end` log record is written.
- Each *data page* will store the LSN of the latest log record corresponding to a change for that data page.

# The ARIES Algorithm (Cont'd)

- Common fields in all log records
  - *Previous (Last) LSN* for that transaction (that ends)
    - Links the log records (in reverse order)
  - *Transaction ID*
  - The *type of log record*
    - For *update*: additionally includes *page ID*, *the length* of the updated item, and *its offset* from the page beginning, the *BFIM*, and the *AFIM* of the item

| Lsn | Last_lsn | Tran_id | Type | Page_id | Other_information |
|---|---|---|---|---|---|
| 1 | 0 | $T_1$ | update | *C* | ... |
| 2 | 0 | $T_2$ | update | *B* | ... |
| 3 | 1 | $T_1$ | commit | | ... |
| 4 | begin checkpoint | | | | |
| 5 | end checkpoint | | | | |
| 6 | 0 | $T_3$ | update | *A* | ... |
| 7 | 2 | $T_2$ | update | *C* | ... |
| 8 | 7 | $T_2$ | commit | | ... |

[Log records in ARIES]

# The ARIES Algorithm (Cont'd)

- Two tables needed for efficient recovery (to be exemplified)
  - ***Transaction Table***
    - Contains an entry for *each active transaction*, with information:
      - *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 DBMS cache, which includes:
      - *Page ID* and *the LSN corresponding to the earliest update to that page*
  - Both maintained by the transaction manager
  - When a crash occurs, these tables are rebuilt in the analysis phase
    - Will be discussed in more details on Slide 30

# The ARIES Algorithm (Cont'd)

- Checkpointing consists of the following:
  - Writing a `begin_checkpoint` record to the log
  - Writing an `end_checkpoint` record to the log
    - Also, the contents of the Transaction Table and Dirty Page Table are appended to the end of the log.
  - Writing the LSN of the `begin_checkpoint` record to a special file.
    - The special file is accessed during recovery to locate the previous checkpoint.
    - If a crash occurs during checkpointing, the file is also referred to get the previous checkpoint, which would be used for recovery.
- Fuzzy checkpoint: used to reduce the cost in ARIES
  - So that the DBMS can continue to execute transactions during checkpointing

# The ARIES Algorithm (Cont'd)

- After a crash, the ARIES manager works in the following:
  - Information from the last checkpoint is accessed through the special file.
  - First, the **analysis phase** proceeds from the `begin_checkpoint` record to the end of the log.
    - When the `end_checkpoint` record is encountered, the Transaction Table and Dirty Page Table are accessed.
    - During analysis, some modifications may be made to these tables.
      - E.g., If an `end` log record was present for a certain $T$, then the entry for $T$ can be deleted from the Transaction Table.

# The ARIES Algorithm (Cont'd)

- Second, the **REDO phase** starts.
  - The recovery manager finds the smallest LSN, say $M$, of all the dirty pages in the Dirty Page Table.
    - Why? To locate where to start! Before $M$, all changes in disk!
  - It then scans forward to the end of the log.
    - For each change in the log, the REDO algorithm determines whether the change has to be reapplied or not.
    - If reapplying a change recorded in the log is necessary, then dirty page $P$ is read from disk and $\mathrm{LSN}(P)$ is compared with the LSN (say, $N$) of the change.
      - If $N < \mathrm{LSN}(P)$, then **NO** need for reapply (already reflected on disk).
      - Otherwise, the change should be reapplied on disk.
  - After the REDO phase, the database is restored to *the exact state* where it crashed.

# The ARIES Algorithm (Cont'd)

(Recall that the set of active transactions (say, `undo_set`) has been identified in the Transaction Table during the analysis phase.)

- Thirdly and lastly, the **UNDO phase** starts.
  - The manager scans backward from the end of the log.
    - Undoes the appropriate actions.
  - The UNDO reads backward in the log until every action of the set of transactions in the `undo_set` has been **undone**.
    - A **CLR** is written for each action that is undone.
      - Why? So that the manager won't undo the action later.

- After the **UNDO phase**, the recovery process is **FINISHED**.
  - Then normal processing can begin again.

*In Step 1, the Analysis Phase starts at checkpoint beginning time*

# ARIES – Recovery Example

| Lsn | Last_lsn | Tran_id | Type | Page_id | Other_information |
|---|---|---|---|---|---|
| 1 | 0 | $T_1$ | update | C | ... |
| 2 | 0 | $T_2$ | update | B | ... |
| 3 | 1 | $T_1$ | commit | | ... |
| 4 | begin checkpoint | | | | |
| 5 | end checkpoint | | | | |
| 6 | 0 | $T_3$ | update | A | ... |
| 7 | 2 | $T_2$ | update | C | ... |
| 8 | 7 | $T_2$ | commit | | ... |

REDO

*DB State before crash*

[The Log at Crash (same as before)]

Before checkpointing
- $T_1$ updates $C$; *commits*.
- $T_2$ updates $B$; not yet committed.

*After* checkpointing
- $T_2$ updates $C$; *commits*.
- $T_3$ updates $B$; not yet committed.

**TRANSACTION TABLE**

| Transaction_id | Last_lsn | Status |
|---|---|---|
| $T_1$ | 3 | commit |
| $T_2$ | 2 | in progress |

**DIRTY PAGE TABLE**

| Page_id | Lsn |
|---|---|
| C | 1 |
| B | 2 |

<= The **REDO phase** starting point (Step 2)

[The Two Tables Stored at Checkpoint]

*Updated*

**TRANSACTION TABLE**

| Transaction_id | Last_lsn | Status |
|---|---|---|
| $T_1$ | 3 | commit |
| $T_2$ | 8 | commit |
| $T_3$ | 6 | in progress |

*Added*

**DIRTY PAGE TABLE**

| Page_id | Lsn |
|---|---|
| C | 7 |
| B | 2 |
| A | 6 |

- In Step 3, **UNDO phase** starts; $T_3$ is *undone*,

- Thereafter, the recovery process is **FINISHED**.

[The Transaction/Dirty Page Tables *after* the **Analysis Phase**]

# Summary

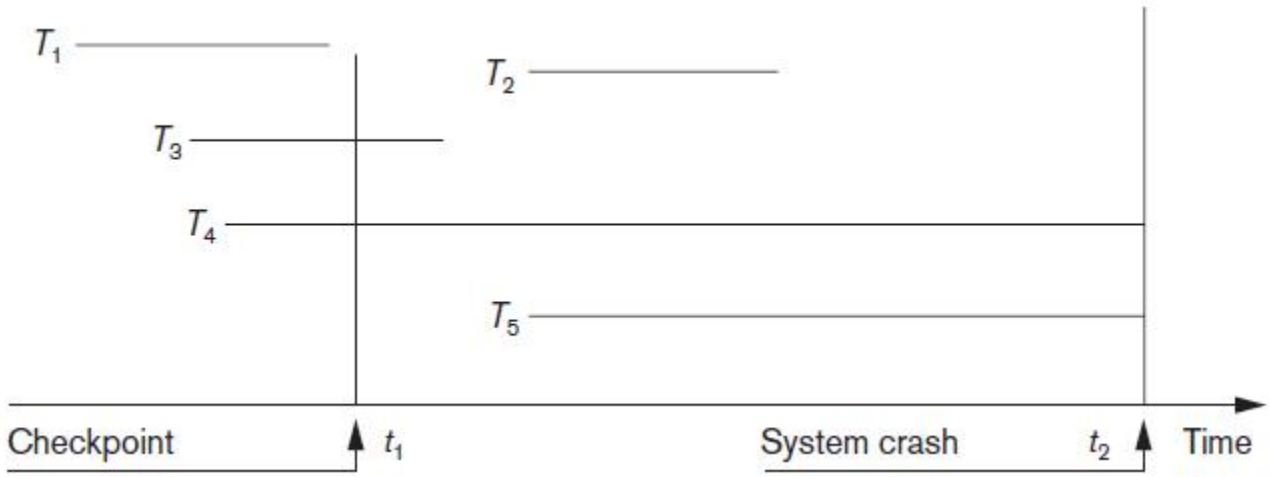
- Recovery from transaction failures
  - The database is restored to the most recent state before crash
- Write-Ahead Logging protocol
- Checkpoint: fuzzy checkpoint in practice
- Database recovery techniques
  - Recovery via deferred update (NO-UNDO/REDO)
  - Recovery via immediate update (UNDO/REDO)
  - The ARIES algorithm: analysis, redo, and undo

# APPENDIX

- Some slides from the Lab Textbook

# Deferred Update (Cont'd)

- An example of a recovery timeline to illustrate the effect of checkpointing under the deferred update protocol




  - "Find which transactions were committed or active at the time of crash."
    - Commit list: $T_2$, $T_3$; Active list: $T_4$, $T_5$
    - REDO all the WRITE operations in each transaction in the commit list.

# Recovery via Checkpoint (Cont'd)

- During recovery when checkpoint is available
  - Case 1) [`Commit`, $T$] exists **before** a checkpoint:
    - No work is required; everything is synchronized.
  - Case 2) [`Commit`, $T$] exists **after** a checkpoint:
    - Proceed with `REDO`. Why?
  - Case 3) [`Commit`, $T$] do **NOT** exist **after** a checkpoint:
    - 3-1) Immediate update: Proceed with `UNDO`. Why?
    - 3-2) Deferred update: no work required. Why?

# Recovery via Log – REDO/UNDO (Cont'd)

- REDO operation
  - Performed when [start_transaction] and [commit] records exist in the log.
    - Transaction was successfully committed but the changed buffer contents may not yet have been written to disk.
    - So while examining the log the contents need be written to disk.
- UNDO operation
  - Performed when [start_transaction] exists but [commit]
    - Transaction was incompletely successful.
    - So the work of the transaction need be cancelled.
    - Its effect may have been reflected on database, so it has to be undone while examining the log

# RECOVERY ALONG WITH LOGGING POLICIES

## Using Immediate Update and Deferred Update

# Example: Two Transactions

- `SELECT` => `read_item()`, `UPDATE` => `update_item()`
- Each transaction reads or writes data items with the respective initial values: *A*=100, *B*=100, *C*=300, D=400.

| $T_1$ | $T_2$ |
|---|---|
| read_item(A); | read_item(A); |
| A=A+10; | A=A+10; |
| read_item(B); | write_item(A); |
| B=B+10; | read_item(D); |
| write_item(B); | D=D+10; |
| read_item(C); | read_item(B); |
| C=C+10; | B=B+10; |
| write_item(C); | write_item(B); |
| write_item(A); | write_item(D); |

Example: Two Transactions

# Log File Contents

- Recorded when $T_1$ -> $T_2$ in sequence

| $T_1$ | $T_2$ |
|---|---|
| read_item(A); | read_item(A); |
| A=A+10; | A=A+10; |
| read_item(B); | write_item(A); |
| B=B+10; | read_item(D); |
| write_item(B); | D=D+10; |
| read_item(C); | read_item(B); |
| C=C+10; | B=B+10; |
| write_item(C); | write_item(B); |
| write_item(A); | write_item(D); |

| Log Sequence Number (LSN) | Log Records |
|---|---|
| 1 | [T1, START] |
| 2 | [T1, UPDATE, B, 200, 210] |
| 3 | [T1, UPDATE, C, 300, 310] |
| 4 | [T1, UPDATE, A, 100, 110] |
| 5 | [T1, COMMIT] |
| 6 | [T2, START] |
| 7 | [T2, UPDATE, A, 110, 120] |
| 8 | [T2, UPDATE, B, 210, 220] |
| 9 | [T2, UPDATE, D, 400, 410] |
| 10 | [T2, COMMIT] |

Assume that we don't record a `READ` record.

Example: Two Transactions

# Recovery via Immediate Update

| LSN | Any work | Result |
|---|---|---|
| i=0 | No work required | T1/T2 not started |
| 1<=i<=4 | UNDO(T1) | T1 cancelled (no effect) |
| 5<=i<=9 | REDO(T1), UNDO(T2) | T1 successfully executed, but T2 cancelled (no effect) |
| i = 10 | REDO(T2) | T2 successfully executed. |

| LSN | Log Records |
|---|---|
| 1 | [T1, START] |
| 2 | [T1, UPDATE, B, 200, 210] |
| 3 | [T1, UPDATE, C, 300, 310] |
| 4 | [T1, UPDATE, A, 100, 110] |
| 5 | [T1, COMMIT] |
| 6 | [T2, START] |
| 7 | [T2, UPDATE, A, 110, 120] |
| 8 | [T2, UPDATE, B, 210, 220] |
| 9 | [T2, UPDATE, D, 400, 410] |
| 10 | [T2, COMMIT] |

Example: Two Transactions

# Recovery via Deferred Update

| LSN | Any work | Result |
|---|---|---|
| i=0 | No work required | T1/T2 not started |
| 1<=i<=4 | No work required for T1 | T1 cancelled (no effect) |
| 5<=i<=9 | REDO(T1) | T1 successfully executed, T2 cancelled (no effect) |
| i = 10 | REDO(T2) | T2 successfully executed. |

| LSN | Log Records |
|---|---|
| 1 | [T1, START] |
| 2 | [T1, UPDATE, B, 200, 210] |
| 3 | [T1, UPDATE, C, 300, 310] |
| 4 | [T1, UPDATE, A, 100, 110] |
| 5 | [T1, COMMIT] |
| 6 | [T2, START] |
| 7 | [T2, UPDATE, A, 110, 120] |
| 8 | [T2, UPDATE, B, 210, 220] |
| 9 | [T2, UPDATE, D, 400, 410] |
| 10 | [T2, COMMIT] |