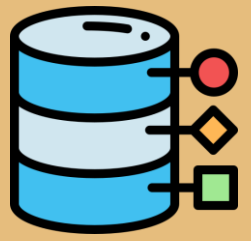


CSAI 302

Advanced Database Systems

Lec 09

Database recovery techniques

ACID principles: Reminder

Atomicity

- Transaction performed in its entirety or not at all
- Ensured by the **recovery subsystem**

Consistency

- The database should always remain consistent
- Ensured by the **transaction program**

Isolation

- Transaction should not interfere with others
- Ensured by the **concurrency control subsystem**

Durability

- Changes of committed transactions must persist
- Ensured by the **recovery subsystem**

Recoverable Transactions

- To ensure **durability** in ACID, a schedule may be

Recoverable	Cascadeless	Strict
• **T** does not commit before each **T'** from which **T** read have committed	• **T** does not read from uncommitted transactions	• **T** is cascadeless and does not overwrite any values written by uncommitted transactions

Recovery

Lecture content

Recovery concepts

- Intro
- Cache
- Log
- Checkpoint

Recovery protocols

- Shadowing
- Deferred-update
- Immediate-update
- ARIES

Recovery intro

- Recovery:
  - ★restores database to most **recent** **consistent** state before failure
- Types of failures ➔

Computer failure (system crash)
- hardware, software, network errors

Transaction failure
- division by zero, constraint violation, etc.

Local transaction errors
- no data found, programmed exception, etc.

Concurrency control enforced
- serializability violation, deadlock break, etc.

Disk crash
- persistent errors with disk reads or writes

Physical problems
- power cut, fire, catastrophe, etc.

Groups of failures

Disk is (significantly) **damaged** (catastrophe, disk crash, etc.)

restore the whole database from back-up storage.

Disk is **not damaged** (everything else)

recover consistency by **undoing** and **redoing** some operations following a recovery ***protocol***

use the **database** on the disk and a **log** of operations

need to take into account the **cache** in the main memory

Cache

Cache

- DBMSs use (dedicated) cache:
  - Collection of buffers (**pages** in main memory) ⇆ **blocks** on disk
- Some **information** (data itself, indexes, logs, etc.)
  - from the disk are kept in main-memory cache for quick access
- to perform an operation with data, the cache is first checked:
  - if the needed block is **in cache**, it is **used**
  - otherwise, an existing cache buffer is **replaced** by the needed block from disk using *buffer replacement policies*
  - if the operation is ***write***, then the buffer becomes **dirty**

Flushing approaches

- Each buffer in the cache has a **dirty bit** (flag) attached:
  - was the buffer modified or not
- *Flushing approaches*:
  - **in-place** flushing:
    - new (cache) version replaces the old (disk) one if dirty bit is 1
      - used in most cases
  - **shadowing**:
    - new version is written in ***another place*** and the old version is kept
      - overhead

Buffer replacement policies

methods to decide which cache buffers should be flushed

LRU

- Basic standard replacement policy is LRU:
  - the **least recently used** buffer is flushed (replaced)
  - not specialized for DBs and not effective, because different domains (types of information) may benefit from different treatment

Domain separation technique

- **Each** domain (data, indexes, logs, or relations) has its **own** dedicated cache
- LRU (least recently used) policy is **used** in each **individually**
  - ★ better results than common LRU
- Can be improved in several ways:
  - Hot set, clock sweep

Clock sweep with hot set

- Hot set:
  - blocks that should always be in cache until a certain point
  - decided externally (e.g., smaller relation in nested-loop join)
- Clock sweep over other buffers (for each domain separately):
  - each (non-hot) cache buffer has an integer count value
  - cache buffers are checked in a round-robin 'cycle' with some regularity, maintaining individual counts
  - if a buffer has started to be used since the last time visited, increment its count
  - otherwise, decrement its count
  - if we need to replace a buffer with a new block from the disk, the buffer with the least count is replaced

System Log

System log

- System log keeps track of executed operations of transactions:
  - sequential, append-only file (i.e., a table of entries)
  - reading and writing the log are not affected by failure except disk or catastrophic failures
    - (i.e., logging is done by stand-alone system operations which is not a part of any transaction operations)
  - backed up periodically to guard against these failures

System log: Properties

- System log keeps track of **relevant** executed operations of transactions:
  - in particular, we do not need to *log computations*
  - log keeps the ***effects of*** data *writes* and maybe *reads* (not write and read ~~**operations**~~)
- log also keeps the **beginnings** and the **ends** (commit or abort) of transactions
  - (e.g., entries of the form **[commit, T]**)
- log keeps **other information**
  - e.g., about **checkpoints** and **dirty buffer** table

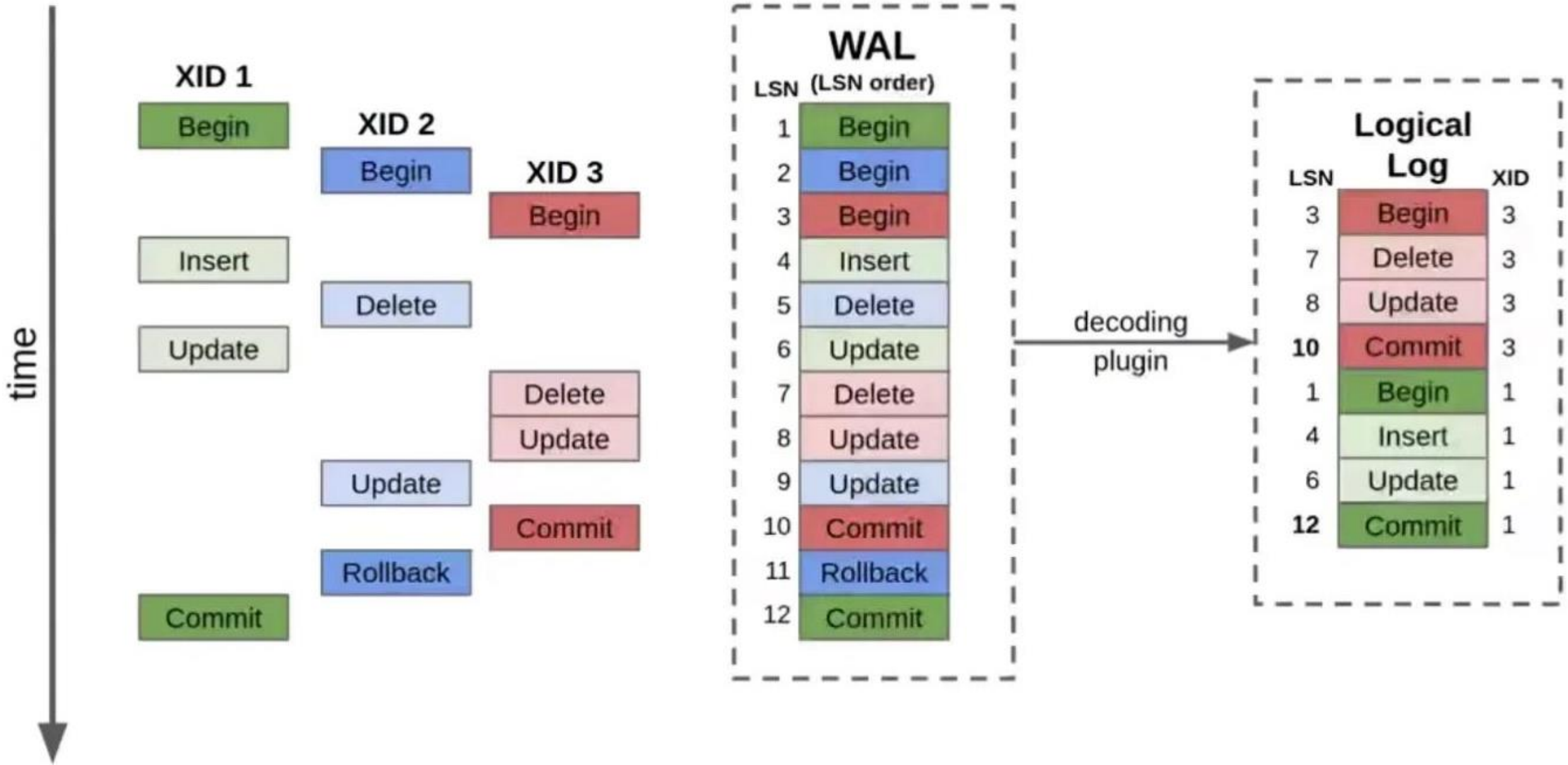
System log: Read and Write entries

- System log table can have read and write entries of several types:
  - UNDO- (OLD-) WRITE entry
    - keeps the before-image (i.e., ***old value***) of a written item used to **undo** the write if the transaction is aborted
  - REDO- (NEW-) WRITE entry
    - keeps the after-image (i.e., ***new value***) of a written item used for **redo** the write if the transaction is committed,
      - but the effect is not on the disk (only in a dirty cache buffer)
  - UNDO&REDO-WRITE entry
    - keeps both above

Log cache buffer

- System log cache buffer:
  - main memory buffer (cache)
  - keeps the last part of the log
  - when full, appended (i.e., flushed) to the log file on the disk
- Log cache buffer may also be flushed to ensure write-ahead logging:
  - log information about a transaction operation is flushed before the effect of the operation is flushed itself

Write-Ahead Logging (WAL)



Write-ahead logging: More detail

- A recovery protocol follows write-ahead logging if
  - before-image (the old version) of each item on the disk cannot be overwritten (by the flushing buffer) by its after-image (the new version) until all UNDO (OLD) log entries have been flushed to disk
  - a transaction cannot commit
    - until all UNDO (OLD) and REDO (NEW) log entries for that transaction have been flushed to disk
- (Most of) recovery protocols are write-ahead logging

WAL steps

- Log First
  - DBMS writes the intended changes as WAL records to an append-only log file before modifying any database files.
- Flush to Disk
  - Before DBMS marks the transaction as committed, it flushes the WAL to disk.
    - This ***guarantees durability***: even if the system crashes before applying the changes to the main database files, the WAL ensures the transaction can be recovered.
- Apply Changes Later
  - DBMS applies the changes to the data files asynchronously during a process called **checkpointing**.
  - At each checkpoint, the dirty pages in memory (modified data) are written back to the storage to reduce reliance on WAL during recovery.
- If a crash happens between the WAL flush and the data file update, DBMS can use the WAL to "redo" the operations that occurred after the last checkpoint.

Checkpoints

Checkpoint idea

- a **recovery operation** that makes it possible to ***safely return to a consistent state***, in case of failure
  - (i.e., all committed transactions at a point of the failure are logically redone, all aborted undone) by
    - flushing all relevant buffers and
    - logging currently active transactions
- Ensures that everything committed so far is on the disk and does not need any action in case of a failure
- Repeated regularly (e.g., every 5 minutes) by recovery subsystem
  - more frequent—less recovery work
  - less frequent—more overhead

checkpoint operation

Suspend execution of all transactions

Flush all main non-pinned dirty memory buffers

Write **checkpoint** entry [checkpoint, list of active transactions] to log and flush the log to the disk

Resume executing transactions

Recovery protocols

Recovery & Concurrency control

- Recovery subsystem ensures **atomicity** and **durability**:
  - restores database to most recent consistent state before failure
    - one or several transactions **abort** and all their changes need to be **rolled back**
- Needs coordination with concurrency control:
  - Assume a system crash event that **aborts all** transactions as a model type failure
  - Assume **strict two-phase locking** as a model concurrency control protocol

Shadowing recovery protocol

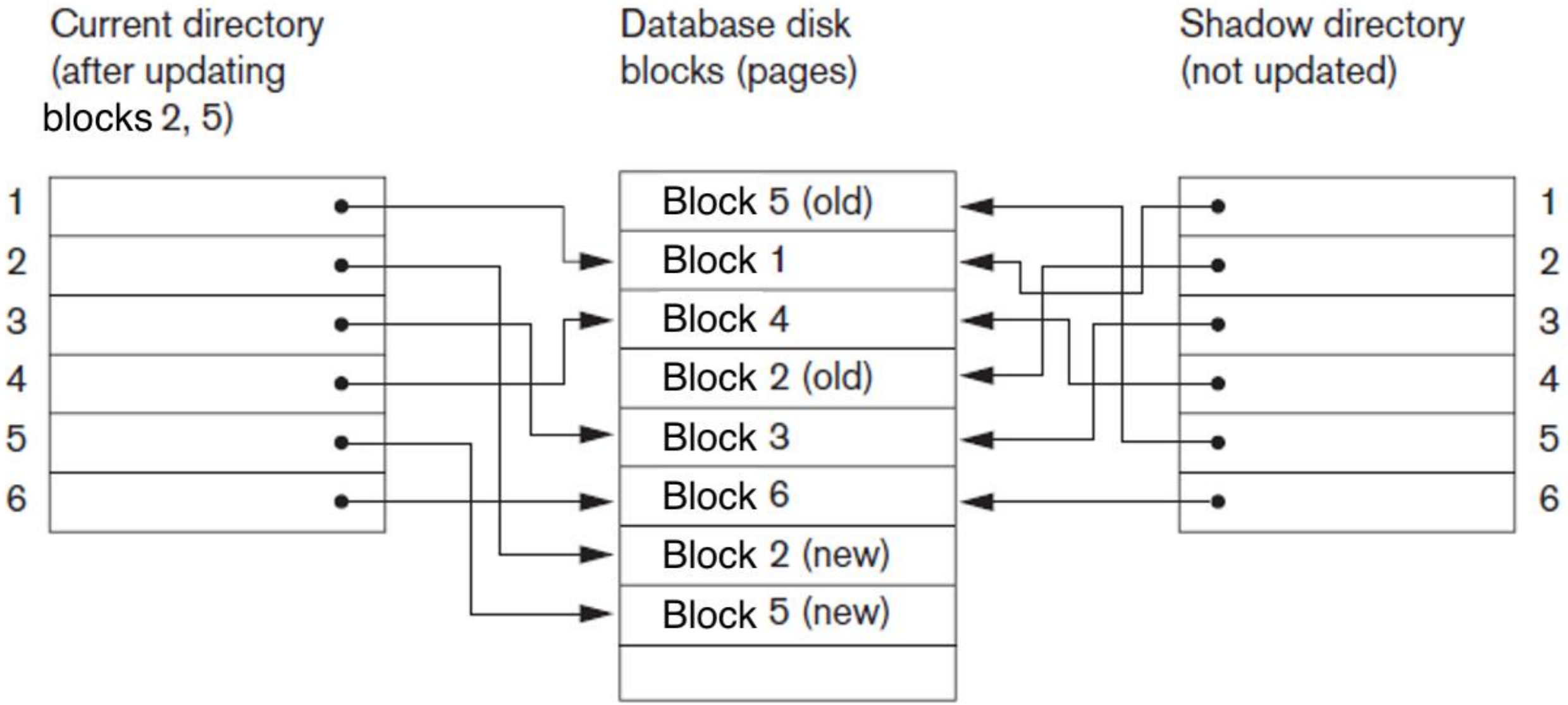
Shadowing recovery protocol: Idea

- flush each modified buffer immediately to a **new place** on disk, the old (shadow) version is kept on disk (and *never modified*)
  - ★ if a transaction **commits**, the **current** version is used
  - ★ if a transaction **aborts**, the **shadow** version is used
  - ***A directory: a table with pointers to disk blocks***

Shadowing recovery protocol

- Shadowing (shadow paging) recovery protocol (for a single transaction):
  - at the **beginning** of a transaction,
    - a current directory with pointers to all blocks is created, and it is copied to shadow directory
  - **after each write**,
    - affected buffers are immediately flushed to a new disk block, and the current directory is updated
  - if a transaction commits, the current version is taken
  - if a transaction aborts, the shadow version is restored

Example of shadowing

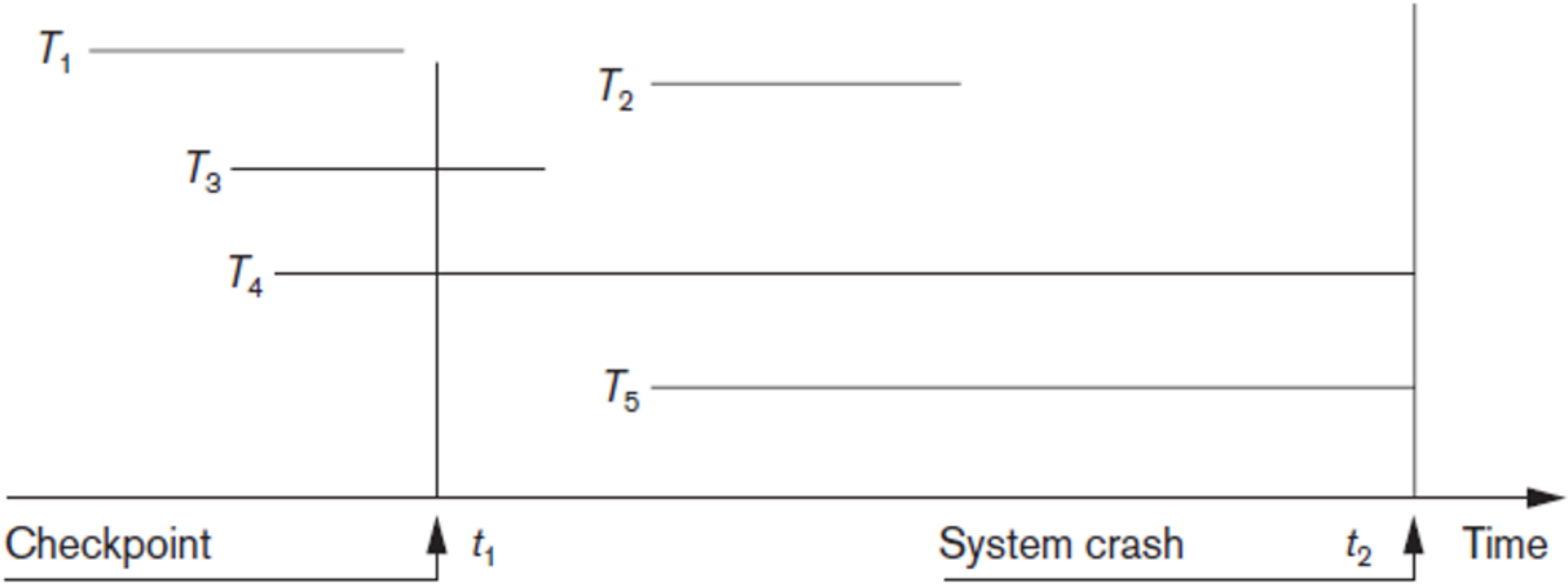


Shadowing properties

- no log required in strict (or single-user) environment
  - so, sometimes called NO-UNDO/NO-REDO
- no checkpoints either
  - essentially, a copy of cache on disk
- results in a lot of overhead and random-ordered blocks on disk
- requires complicated garbage collection

Deferred-update protocol

Transaction picture



Deferred update: Idea

- Deferred update recovery protocol idea:
  - no-steal approach:
    - postpone all flushes of (dirty buffers) to disk until the transaction commits
  - REDO- (NEW-) log entries are needed:
    - to recover committed transactions after a system crash
  - UNDO- (OLD-) log entries are not needed:
    - no changes of aborted transactions are on the disk
  - thus, NO-UNDO/REDO

Properties and Assumptions

- Deferred update recovery protocol properties:
  - effective only for short transactions with few changes
  - cache size is an issue with longer transactions
- Simplifying assumptions, some already mentioned
  - (can be relaxed but rules and procedures may need adjustments):
  - strict two-phase locking with blocks as items
  - system crash as a fail (i.e., all active transactions abort)
  - all blocks changed by a transaction fit into cache

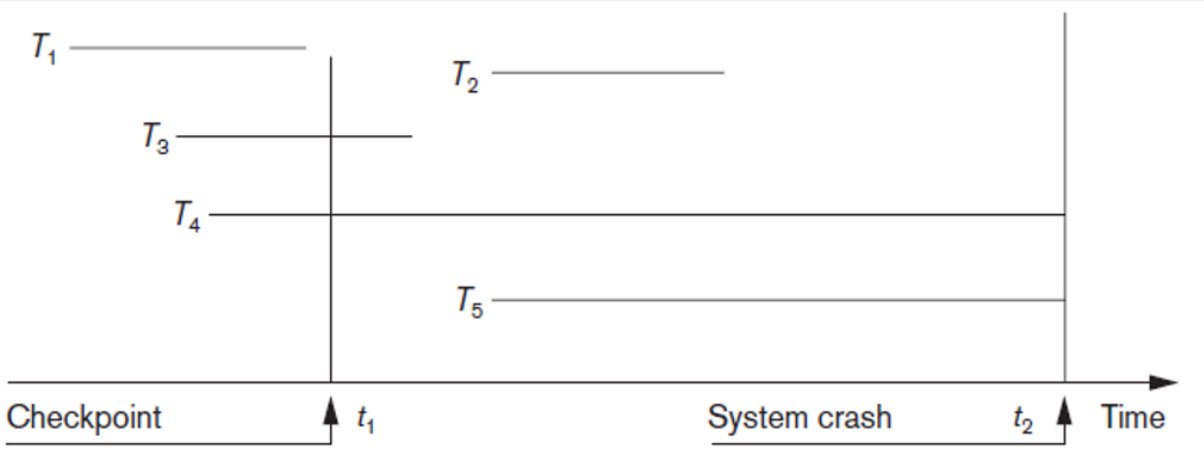
Deferred update rules

- use log with REDO entries and regular checkpoints
- all buffers changed by a transaction are pinned until commit
  - (i.e., no-steal approach)
- transaction does not commit until all its REDO-type log entries are recorded in log and log buffer is flushed to disk
  - (i.e., the relevant REDO-part of the write-ahead approach)

Deferred update: Recovery procedure

- (evaluated by the recovery subsystem after a system crash):
- find the active (i.e., latest) checkpoint in the log on the disk
  - all effects of committed transactions are on the disk by the checkpoint
- using the checkpoint's active transactions and the log after the checkpoint, construct the set of all committed transactions between the checkpoint and the crash
  - their commit log entries are on the disk
- redo all operations of the committed transactions from the set
  - all their log entries are on the disk

Deferred update protocol: Example



- T1: already committed and flushed, no action required
- T2: may have non-flushed buffers,
  - but the log (including begin_transaction) is flushed, so all changes
  - can be recovered from the log (considered from from the checkpoint)
- T3: may have non-flushed buffers,
  - but the log is flushed and the checkpoint remembers that it is active, so
  - all the changes can be recovered from the log
- T4, T5: aborted and no buffers are flushed, no action required

Immediate-update recovery

Immediate update

- Immediate update recovery protocol idea:
  - possible steal approach:
    - allow to flush of dirty buffers
    - both before and after the transaction commits
  - REDO- (NEW-) log entries log entries are needed:
    - to redo committed transactions after a system crash
  - UNDO- (OLD-) log entries log entries are needed: to undo aborted transactions
    - thus, UNDO/REDO (generalises deferred-update)
- Immediate update recovery protocol properties:
  - more general than deferred-update
  - more difficult

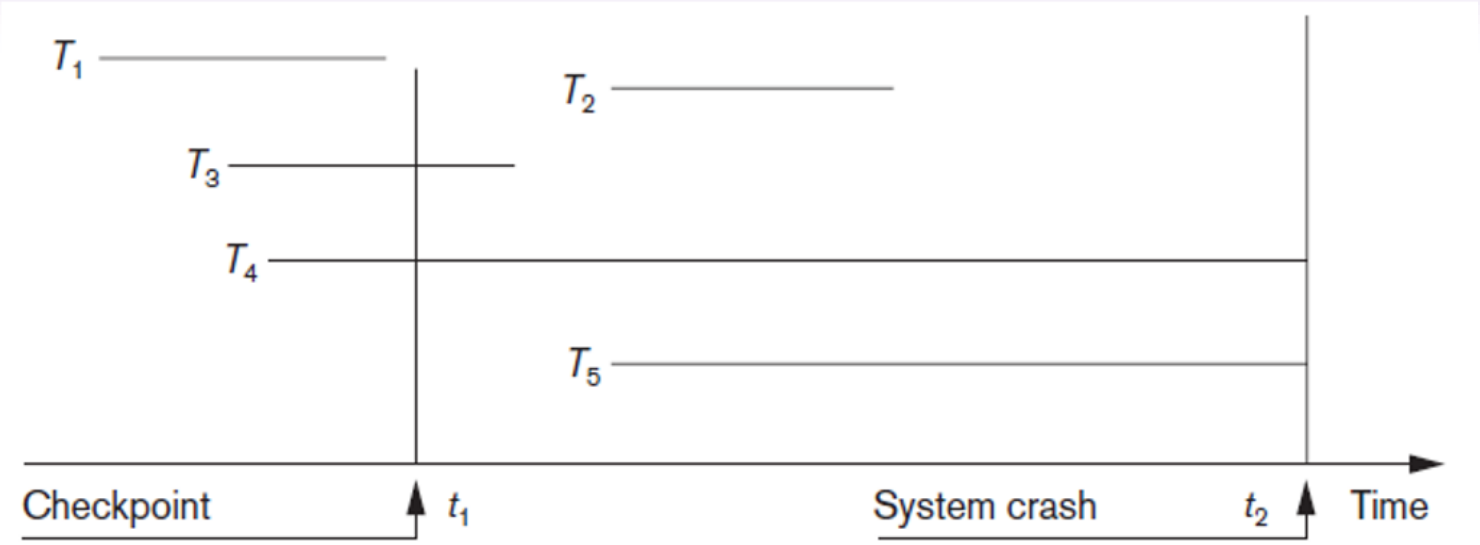
assumptions and rules

- Simplifying assumptions, less than above
  - strict two-phase locking with blocks as items (not essential here)
  - system crash as a fail (i.e., all active transactions abort)
- Immediate update protocol rules (just write-ahead logging in full):
  - use log with REDO/UNDO entries and regular checkpoints
  - a dirty buffer does not flush until all relevant log entries are recorded in log and log buffer is flushed to disk
    - (i.e., the first part of the write-ahead approach)
  - transaction does not commit until all its log entries are recorded in log and log buffer is flushed to disk
    - (i.e., the second part of the write-ahead approach)

Immediate update: Recovery procedure

- (evaluated by the recovery subsystem in case of a crash):
- find the active (i.e., latest) checkpoint in the log on the disk
  - all effects of committed transactions are on the disk by the checkpoint
- construct the sets of all committed and non-committed (i.e., aborted) transactions between the checkpoint and the crash
  - their commit log entries are on the disk
- redo all operations of the committed transactions
  - all their log entries are on the disk
- undo all known operations of the non-committed transactions
  - some of their log entries may not be on the disk, but their corresponding changes were not flushed

Immediate update protocol: Example



- T1, T2, T3: same as before
- T4, T5: aborted, so should be undone;
  - some changes may be already flushed by t2 and some may not, but the log entries of all flushed changes are also flushed

ARIES recovery protocol

ARIES: Overview

- ARIES protocol main properties:
  - Algorithm for Recovery and Isolation Exploiting Semantics
  - an improved version of immediate-update protocol
    - (possible steals, REDO/UNDO log entries, write-ahead logging)
  - used in practice
  - again, we concentrate on strict schedules
- ARIES update recovery protocol improvement ideas:
  - different checkpoints (main improvement): instead of flushing everything,
    - remember the table of currently dirty buffers
  - other improvements (e.g., logging of recovery): not discussed here

ARIES Checkpoints

- During execution, ARIES maintains (in main memory):
  - Transaction table of currently active and committed transactions
    - (ID, pointer to the most recent relevant log entry, status)
  - Dirty buffer table of currently dirty buffers in the cache (ID, pointer to the earliest update log entry)
- ARIES checkpoint (fuzzy in general)
  - the begin_checkpoint entry identifies the start point for the recovery, and the state of the transaction table and dirty buffer table to store
  - the end_checkpoint entry has the tables (at the begin-point state) appended in the log
    - the log must be flushed at the end of checkpoint

ARIES recovery protocol: Example

(a) The log

(b) Possible transaction and dirty block table at the begin checkpoint (appended to end_checkpoint)

(c) Possible tables at the moment of the end of the log (observe that 7 in dirty buffer table implies that C was flushed between Lsn 5 and 7)

(a)

Lsn	Last_lsn	Tran_id	Type	Block _id	Other_information
1	0	T_1	write	C	...
2	0	T_2	write	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	write	A	...
7	2	T_2	write	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	2	in progress

DIRTY BUFFER TABLE

Block _id	Lsn
C	1

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	8	committed
T_3	6	in progress

DIRTY BUFFER TABLE

Block _id	Lsn
C	7

ARIES recovery main steps

- ARIES recovery procedure (evaluates at the crash event):
- Analysis step:
  - identifies the earliest update log entry of a dirty buffer
  - at the checkpoint (and collects other relevant information)
- Redo step:
  - go through the log from the identified point to the end and redo the necessary operations (i.e., the writes of dirty blocks in the table before the begin checkpoint and all writes after it)
- Undo step:
  - go through the log from the end to the beginning of all uncommitted (by the log end) transactions and undo necessary operations
  - (i.e., all writes that are not overwritten by a redo)
- Evaluated undo's and redo's are also logged,
- so that a crash during update does not cause a need to re-evaluate

ARIES recovery protocol: Example

Lsn	Last_lsn	Tran_id	Type	Block _id	Other_information
1	0	T_1	write	C	...
2	0	T_2	write	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	write	A	...
7	2	T_2	write	C	...
8	7	T_2	commit		...

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	committed
T_2	2	in progress

DIRTY BUFFER TABLE

Block _id	Lsn
C	1

Let the log above be in on the disk after a crash. ARIES steps:

1. Analysis step: identifies the earliest log entry: Lsn 1
2. Redo step: redo the writes with Lsn 1, 6, 7 (no 2)
3. Undo step (only T_3): undo update with Lsn 6

(Lsn 6 is redone and undone;

think of advantages and disadvanatages of such strategy)

THANK YOU