Anna Östlin Pagh and Rasmus Pagh
IT University of Copenhagen

Advanced Database Technology

April 11, 2005

SYSTEM FAILURES

Lecture based on [GUW 17.1+17.2.4+5, 17.4]

Slides based on
Notes 08: Failure recovery
for Stanford CS 245, fall 2002
by Hector Garcia-Molina

This lecture

- Part 1:
  Logging of **transactions** in order to allow **recovery** in case of a system failure.
- Part 2:
  Reliable disk systems (RAID)

Transactions

- **Transactions** are user-defined groups of updates to the database.
- We have previously considered the possibility of many **concurrent transactions**, but for now assume that transactions occur one by one.
- Basic property: Transactions are **atomic** (to maintain consistency).

Handling failures during transactions?

Types of system events

Desired events: See product manuals.

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

That's it!!

Undesired unexpected: Everything else!

Undesired unexpected: Everything else!

Examples:

- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe...

We deal only with expected events

Simplified view of DB operations

- Input (x): block with x → memory
- Output (x): block with x → disk

} controlled by buffer manager

Assumption:
Storage is resilient and writes are atomic.

- Read (x,t): do Input(x) if necessary
  t ← value of x in block
- Write (x,t): do Input(x) if necessary
  value of x in block ← t

} controlled by trans-actions

Logging

- To enable recovery, database systems use **logging** of changes to data on disk.
- Arguably, the simplest logging strategy is **undo logging** (due to Hansel and Gretel, 782 AD; improved in 783 AD to durable undo logging).
- We consider the more flexible **undo/redo logging**.

Undo/redo logging

- Whenever a database element X is going to be changed (Write(x,t)) by transaction Ti, we must **first** write to the log an entry of the form:

  `<Ti, X, New X val, Old X val>`
- Whenever a transaction Ti commits, we write to the log the entry: `<COMMIT Ti>`
- Important that disk cache is flushed! (The flush **is** the commit.)

Problem session

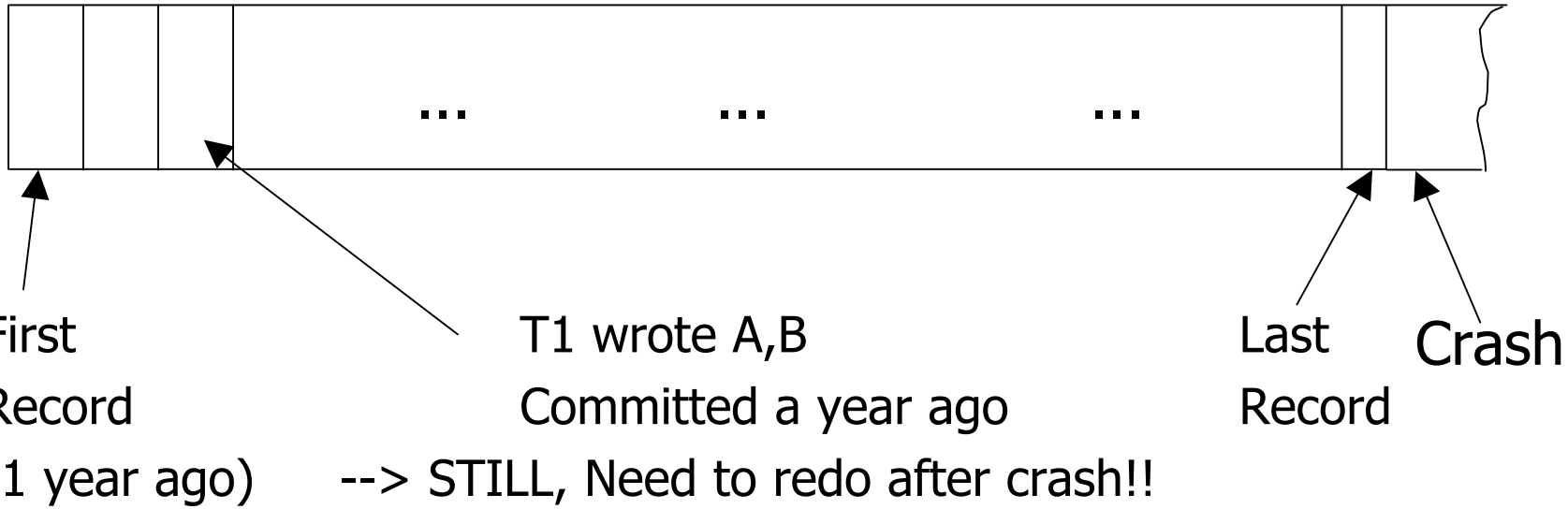
- Suppose that a DB crashes, and that an undo/redo log is available.
- How can we recover to a consistent state, i.e., one in which every transaction has either been fully executed, or not executed at all?

Recovery using undo/redo log

- Status: Some of the logged DB changes have been written to the DB, others have not.
- We may redo all transactions T **with** <COMMIT T> in the log, in the order that DB elements were changed.
- We may undo all transactions T **without** <COMMIT T> in the log, in the opposite order of that in which DB elements were changed.

Recovery can be very, very SLOW

Log:



Checkpoints (simple version)

Periodically:

(1) Do not accept new transactions

(2) Wait until all transactions finish

(3) Flush all log records to disk (log)

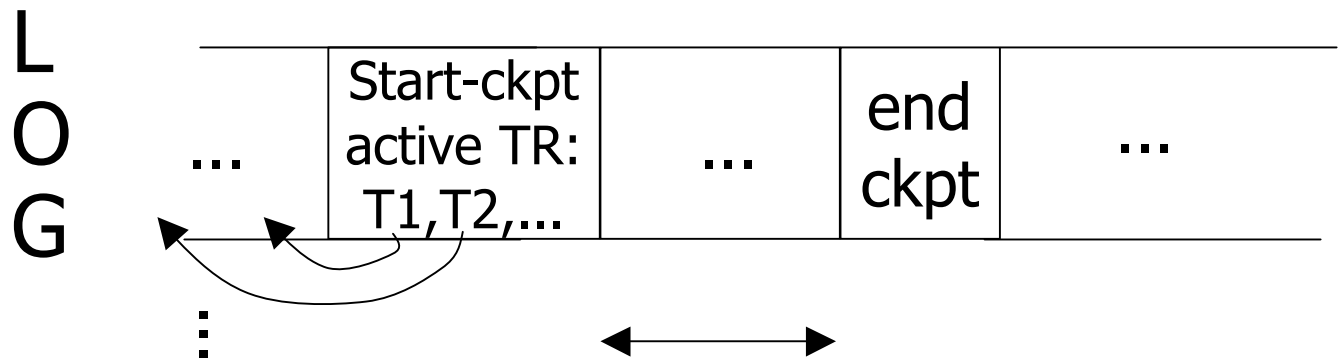
(4) Flush all DB buffers to disk (don't discard buffers)

(5) Write **checkpoint** record on disk (log)

(6) Resume transaction processing

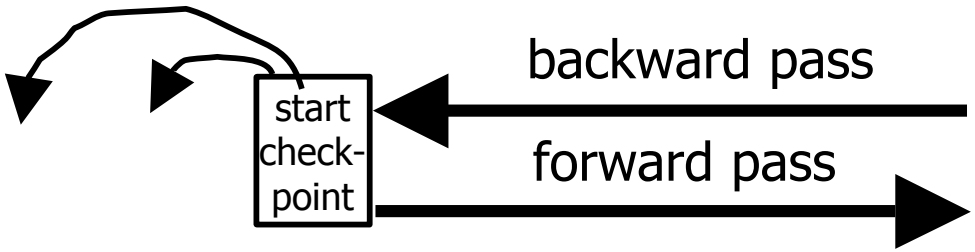
Non-quiescent checkpoints

Idea: Record ongoing transactions at checkpoint.



Undo/redo log recovery

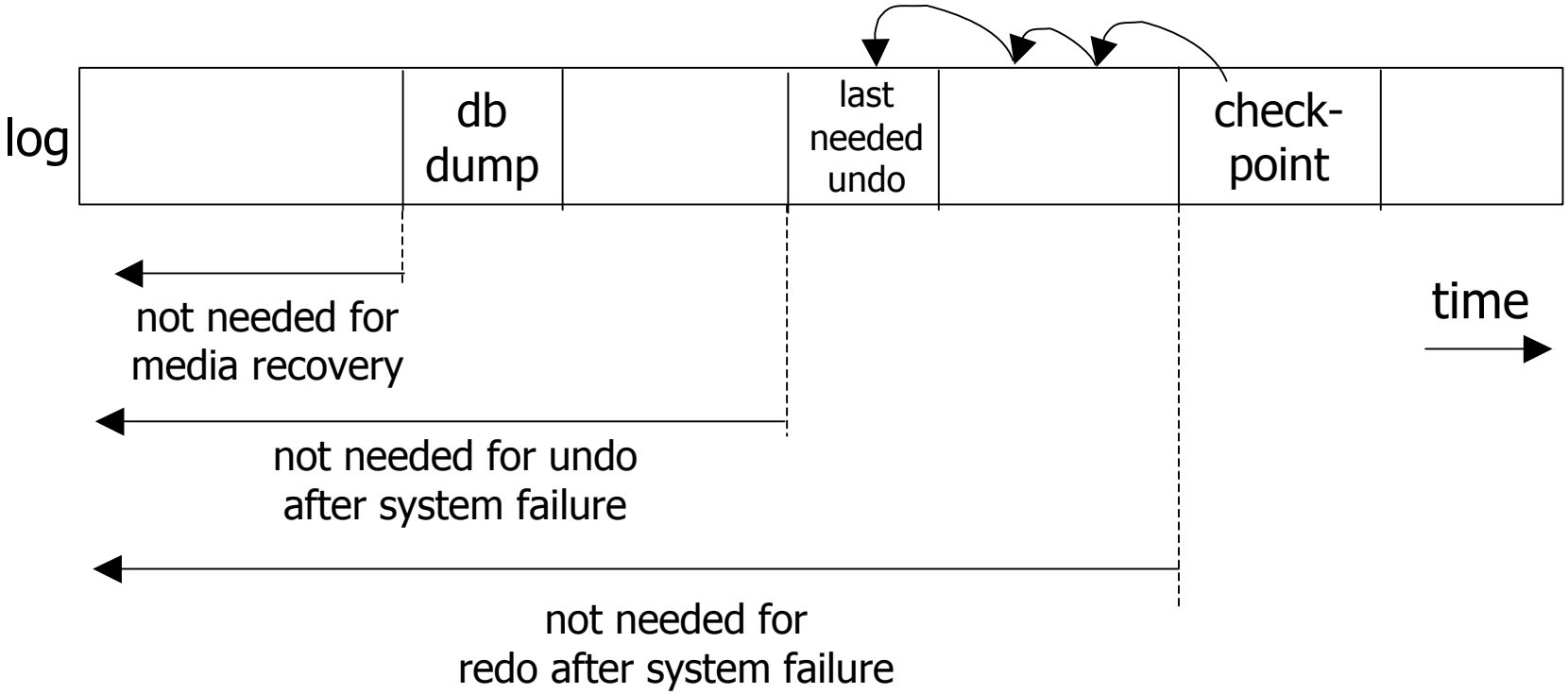
- Backwards pass (back to latest completed checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo pending transactions
  - follow undo chains for transactions in checkpoint active list and not in S
- Forward pass (latest checkpoint start to end of log)
  - redo actions of S transactions



Problem session

- What if the DB crashes during undo/redo recovery?
- How does undo/redo logging work with concurrent transactions (locking)? Any problems?

When can log be discarded?



Summary

- We can ensure that transactions are atomic, even in the presence of system failures, using undo/redo logging.
- Underlying assumption: Storage is resilient and writes are atomic.
- **Next:**
  Separate techniques such as RAID ensure resilient storage.