# COMP207
# Database Development

Lecture 9

Transaction Management:
Reconciling Conflict-Serialisability & Recovery

- Nonquiescent checkpoints will not be part of the exam

# Canceled lecture

- Tuesday 17-18 is canceled again
  - This time because of a networking event for second years in Sensor City that starts at 17 as well (no more slots though)
- Tuesday 13-14 is still going on though

# Review of Undo & Redo Logging

- Logs activities with the goal of:
  - (For Undo): “undoing” to a previous database state.
  - (For Redo): “redoing” a database state that has been lost.
- Log records (or log entries):
  - **<START T>**: Transaction T has started.
  - **<COMMIT T>**: Transaction T has committed.
  - **<ABORT T>**: Transaction T was aborted.
  - **<T, X, v>**: Transaction T has updated the value of database item X, and the (**old** for **undo** and **new** for **redo**) value of X was v.
    - Response to **write_item(X)**
    - If this entry occurs in the log, then the new value of X might not have been written to the database yet

# Undo/Redo Logging

- Good properties of undo logging and redo logging
- Log records:
  - Same as before, but replace **<T, X, v>**
  - **<T, X, v, w>**: “Transaction T has updated the value of database item X, and the **old**/**new** value of X is v/w.”
- Procedure:
  - Write all log records for all updates to database items first
  - Then write updates to disk
  - <COMMIT T> can be written to disk before or after all changes have been written to disk
- Recovery needs to process log in both directions

# Review of Undo/Redo Logging

5 min

| Time | Transaction $T_1$ | Transaction $T_2$ |
|---|---|---|
| 1 | read_item(X) | |
| 2 | X := X * 2 | |
| 3 | write_item(X) | |
| 4 | | read_item(X) |
| 5 | read_item(Y) | |
| 6 | | X := X * 3 |
| 7 | | write_item(X) |
| 8 | Y := X + Y | |
| 9 | write_item(Y) | |

X = 1
Y = 2

- How does undo/redo logging work on this schedule?
  - Which **log entries** are written to buffer/disk & when?
  - Which **other operations** must be executed & when?

| Time | Transaction $T_1$ | Transaction $T_2$ | Local $T_1$ | | Local $T_2$ | | Buffer | | Disk | | Buffer log |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | X | Y | X | Y | X | Y | X | Y | |
| 0 | | | | | | | | | 1 | 2 | <START $T_1$> |
| 1 | read_item(X) | | 1 | | | | 1 | | 1 | 2 | |
| 2 | X := X * 2 | | 2 | | | | 1 | | 1 | 2 | |
| 3 | write_item(X) | | 2 | | | | 2 | | 1 | 2 | <$T_1$, X, 1, 2> |
| 4 | | | 2 | | | | 2 | | 1 | 2 | <START $T_2$> |
| 5 | | read_item(X) | 2 | | 2 | | 2 | | 1 | 2 | |
| 6 | read_item(Y) | | 2 | 2 | 2 | | 2 | 2 | 1 | 2 | |
| 7 | | X := X * 3 | 2 | 2 | 6 | | 2 | 2 | 1 | 2 | |
| 8 | | write_item(X) | 2 | 2 | 6 | | 6 | 2 | 1 | 2 | <$T_2$, X, 2, 6> |
| 9 | Y := X + Y | | 2 | 4 | 6 | | 6 | 2 | 1 | 2 | |
| 10 | write_item(Y) | | 2 | 4 | 6 | | 6 | 4 | 1 | 2 | <$T_1$, Y, 2, 4> |
| 11 | | | 2 | 4 | 6 | | 6 | 4 | 1 | 2 | <COMMIT $T_1$> |
| 12 | **flush_log** | | 2 | 4 | 6 | | 6 | 4 | 1 | 2 | |
| 13 | **output(X)** | | 2 | 4 | 6 | | 6 | 6 | 6 | 2 | |
| 14 | **output(Y)** | | 2 | 4 | 6 | | 6 | 4 | 6 | 4 | |
| 15 | | | 2 | 4 | 6 | | 6 | 4 | 6 | 4 | <COMMIT $T_2$> |
| 16 | | **flush_log** | 2 | 4 | 6 | | 6 | 4 | 6 | 4 | |
| 17 | | **output(X)** | 2 | 4 | 6 | | 6 | 4 | 6 | 4 | |

Why are DBMS using Undo/Redo?

# Undo without Redo

- Undo essentially ensures Atomicity
- Can ensure durability using Force
  - **Force** the writing of updates to disk before commit
  - (**No Force** is not to require this)
  - Force is expensive in disk operations

# Redo without Undo

- Redo essentially ensures Durability
- Can ensure atomicity using No Steal
  - **No Steal** means that uncommitted data may not overwrite committed data on disk
  - (**Steal** is not to require this)
  - **No Steal** is expensive to ensure

# Ensuring A and D

- Could ensure Atomicity and Durability without log using No Steal/Force
  - Very hard and expensive to ensure
  - (Must commit **and** write every change to disk at once)
- In practice:
  - Want Steal/No Force (cheapest in time) → Use Undo/Redo

# More Efficient Recovery via Checkpoints

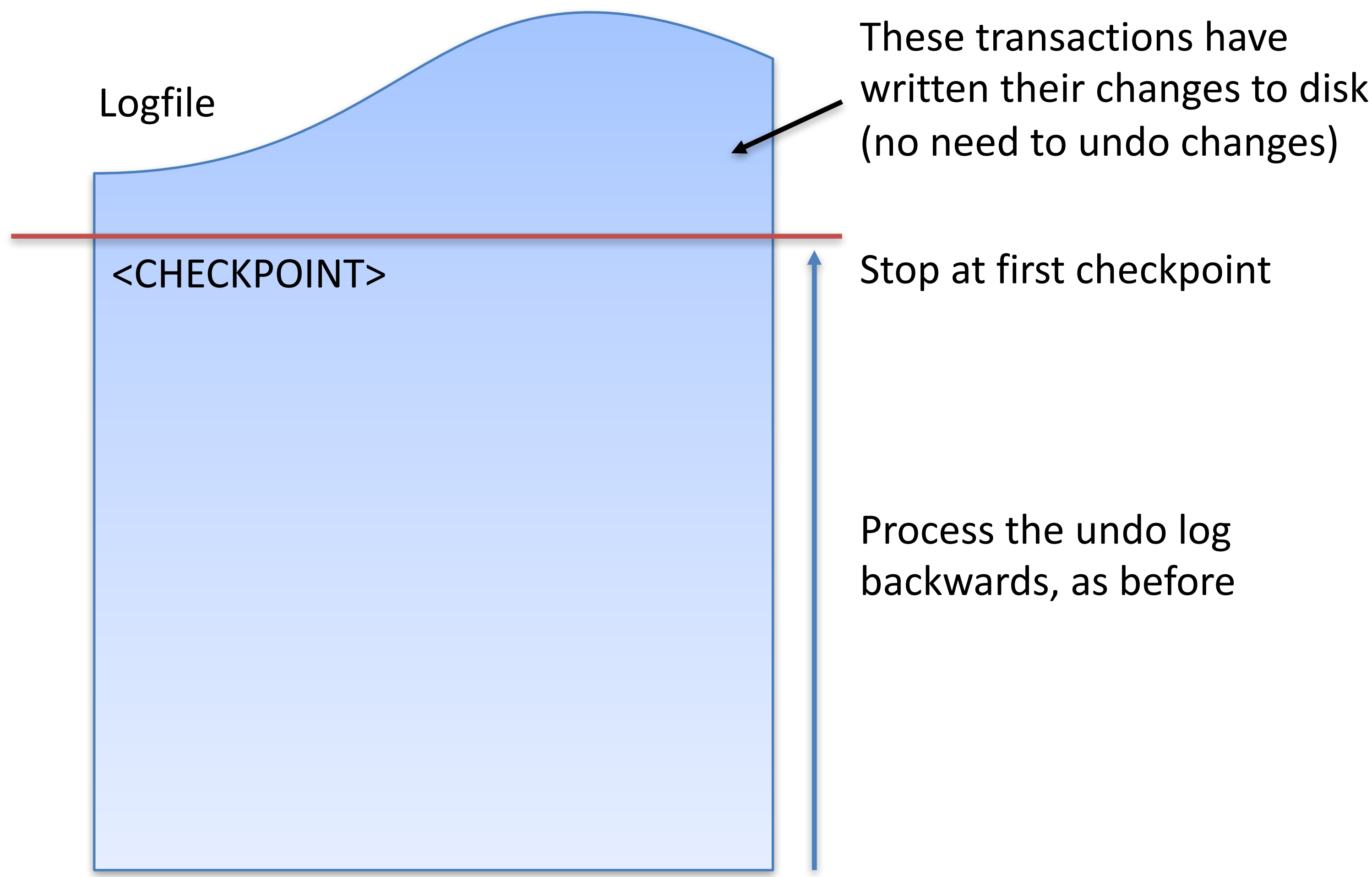
# Simple Checkpointing

- Idea: *checkpoint* the log periodically
  - Every *m* min., after *t* transactions since last checkpoint, …
  - No need to undo transactions before t
- Procedure:
  1. Stop accepting new transactions
  2. Wait until all active transactions finish and have written COMMIT or ABORT record to the log.
  3. Flush the log to disk.
  4. Write a log record **<CHECKPOINT>**.
  5. Flush the log to disk again.
  6. Resume accepting transactions.

There are variants of checkpointing that avoid this
See later!

# Recovery via Simple Checkpoints

# Recovery With Simple Checkpoints



| Time | Transaction $T_1$ | Transaction $T_2$ | Log (buffer) | Log (disk) |
|---|---|---|---|---|
| 0 | | | <START $T_1$> | |
| 1 | read_item(X) | | | |
| 2 | X := X * 2 | | | |
| 3 | write_item(X) | | <$T_1$, X, 1> | |
| 4 | | | <START $T_2$> | |
| 5 | | read_item(X) | | |
| 6 | read_item(Y) | | | |
| 7 | | X := X * 3 | | |
| 8 | | write_item(X) | <$T_2$, X, 2> | |
| 9 | Y := X + Y | | | |
| 10 | write_item(Y) | | <$T_1$, Y, 2> | |
| 11 | **flush_log** | | | |
| 12 | **output(X)** | | | |
| 13 | **output(Y)** | | | |
| 14 | | | <COMMIT $T_1$> | |
| 15 | **flush_log** | | | |
| 16 | | **flush_log** | | |
| | | **output(X)** | | |
| | | | <COMMIT $T_2$> | |
| 19 | | **flush_log** | | |

Checkpointing starts

Transaction $T_3$ is submitted

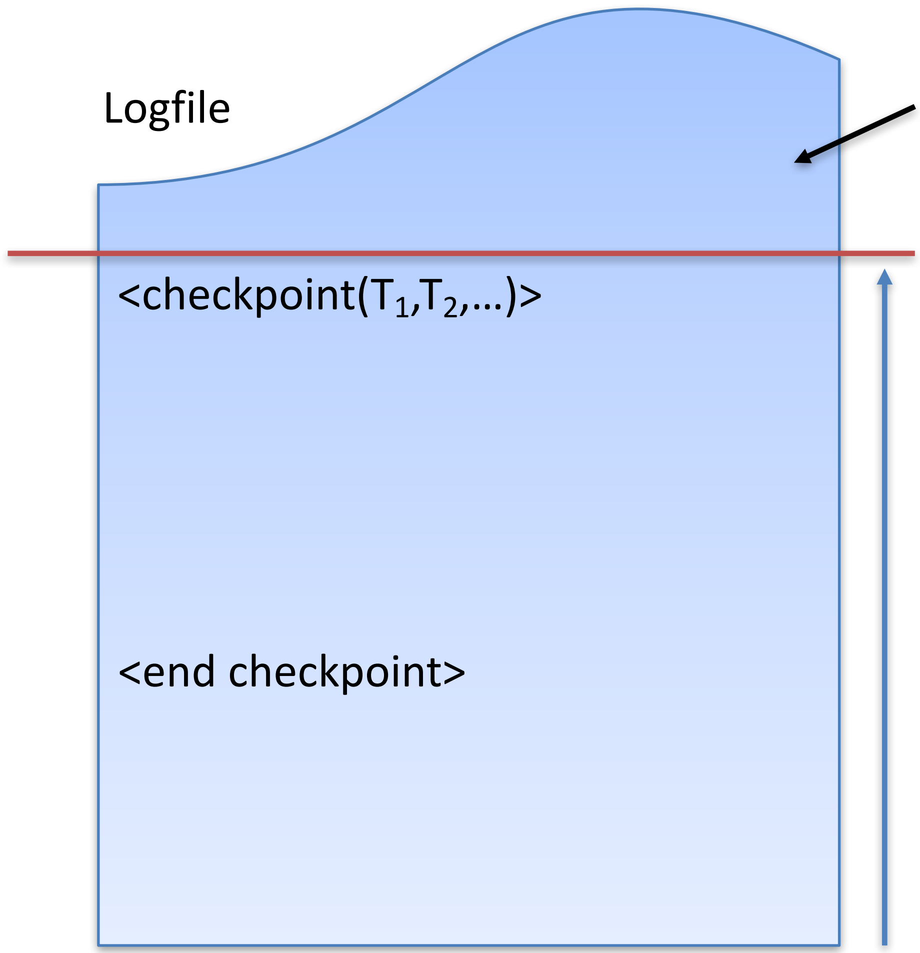
What will happen?

# Better checkpoints!

# Nonquiescent Checkpoints

- Requirements:
  - Undo/Redo logging
  - Transactions do not write to buffers(!) before they are sure they want to commit
- Procedure:
  - Write <Checkpoint($T_1$, $T_2$,…)> in log and flush it
    - » $T_1$, $T_2$,… are the transaction in progress (i.e. not committed and not aborted)
  - Write the content of the **changed** buffer to disk (i.e. **output**)
  - Write <End Checkpoint> in log and flush it

# Recovery via Nonquiescent Checkpoints




These transactions have written their changes to disk (no need to undo changes)

Stop at first <checkpoint($T_1$,$T_2$,…)> with corresponding <end checkpoint>

Process the undo/redo log, as before

EXCEPT: only redo part of committed transactions in $T_1$,$T_2$,… after <checkpoint($T_1$,$T_2$,…)> AND undo all of uncommited transactions in $T_1$,$T_2$,… incl. before <checkpoint($T_1$,$T_2$,…)>

# Nonquiescent advantages

Nonquiescent checkpoints…

- does not require delaying transactions
- can be forcefully finished

# Dirty reads

# "Dirty Reads"

Read something written by an uncommited transaction

- In practice, the isolation property is often not fully enforced (→ "dirty reads" may occur)
- Reason: efficiency!
  - Spend less time on preventing "dirty reads"
  - Gain "more parallelism" by executing some transactions that would have to wait to prevent "dirty reads"
- You can decide:

Other option: READ ONLY

Other levels in SQL: READ COMMITTED, REPEATABLE READ, SERIALIZABLE

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

- "Dirty reads" can slow down the system when transactions have to abort

# Dirty Reads and Rollbacks

| Time | Transaction $T_1$ | Transaction $T_2$ | X | Y |
|---|---|---|---|---|
| 0 | lock(X) | | 1 | 2 |
| 1 | read_item(X) | | | |
| 2 | X := X + 100 | | 101 | |
| 3 | write_item(X) | | | |
| 4 | lock(Y) | | | |
| 5 | unlock(X) | | | |
| 6 | | lock(X) | | |
| 7 | | read_item(X) | | |
| 8 | | X := X * 2 | 202 | |
| 9 | | write_item(X) | | |
| 10 | | **lock(Y)** denied | | |
| 11 | read_item(Y) | | | |
| 12 | **abort** scheduler automatically unlocks Y | | | |
| 13 | | lock(Y) | | |
| 14 | | … | | |
| 15 | | **commit** | | |

$T_2$ depends on "dirty data" from $T_1$
→ must abort $T_2$, too!

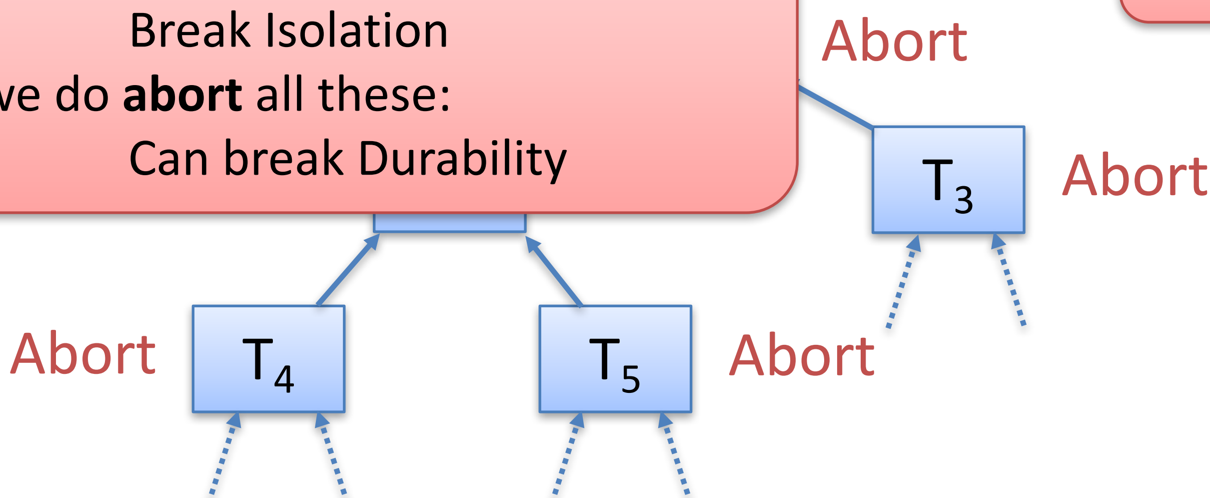
# Cascading Rollback

If a transaction T aborts:

- Find all transactions that have read items written by T.
- Recursively abort all transactions that have read items written by an aborted transaction.

If we do **not abort** all these:
Break Isolation
If we do **abort** all these:
Can break Durability

Very slow → want to avoid this



# Isolation vs Durability

| Time | Transaction $T_1$ | Transaction $T_2$ | X | Y |
|---|---|---|---|---|
| 0 | lock(X) | | 1 | 2 |
| 1 | read_item(X) | | | |
| 2 | X := X + 100 | | 101 | |
| 3 | write_item(X) | | | |
| 4 | lock(Y) | | | |
| 5 | unlock(X) | | | |
| | | lock(X) | | |
| | | read_item(X) | | |
| | | X := X * 2 | 202 | |
| | | write_item(X) | | |
| | | **commit** | | |
| 11 | read_item(Y) | | | |
| 12 | **abort** | | | |

If we do **not abort** $T_2$:
Break Isolation
If we do **abort** $T_2$:
Break Durability

# Conflict-Serialisability vs Recovery

**Conflict-Serialisability**

- Many nice properties:
  - Equivalent to serial schedules
  - Ensure consistency / correctness
- Can be enforced by two-phase locking (2PL)

**Logging and Recovery**

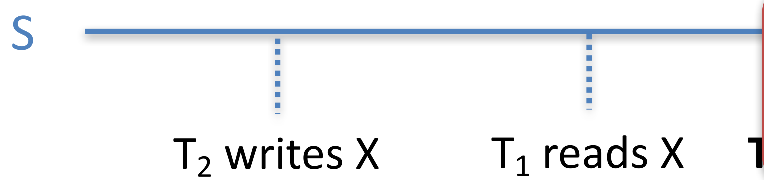
- Suitable logging techniques ensure that we can restore desired database states
  - Undo incomplete transactions
  - Redo committed transactions
  - Undo a single or a selected number of transactions
- Robust: works even after system failures

Problem: cascading rollbacks may be necessary!

# Recoverable Schedules

- The problem for Durability in regards to cascading rollbacks occur because a transaction $T_1$ reads data from some transaction $T_2$, then $T_1$ commits and afterwards $T_2$ aborts.

- A schedule S is **recoverable** if the following is true:
  - if a transaction $T_1$ commits and has read an item X that was written before by a different transaction $T_2$, …
  - then **$T_2$ must commit before $T_1$ commits**.




Can still do cascading rollbacks, but only active transactions can be forced to abort

# Example

- A **recoverable** schedule:

T2 reads data that was written before by T1

$$S_1: w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_1; c_2$$

$T_1$ must commit before $T_2$ can commit

- A **non-recoverable** schedule:

$$S_2: w_1(X); w_1(Y); w_2(X); r_2(Y); w_2(Y); c_2; c_1$$

$T_2$ reads data that was written before by $T_1$

But: $T_2$ commits first

- Note:
  - $S_1$ is *not* serialisable.
  - $S_2$ is serialisable.

# Summary

- Undo/Redo logging and why it is used
- Reconciliation of conflict-serialisability and recovery
  – Can lead to problems (**cascading rollbacks**) if done naively
  – Avoiding cascading rollbacks requires a smarter way of scheduling transactions
- Ideas:
  – **Recoverable schedules**: T commits only if all transactions that T has read from have committed