# CS411
# Database Systems

**14: Concurrency Control**

**Kazuhiro Minami**

## Announcements

- Homework 5 due on Dec 1
- Graduate project is due on Dec 1
- Project stage 5 is due on Dec 3

- No office hour next Friday (Nov. 26)

## Undo/Redo Logging

## Redo/undo logs save both before-images and after-images.

<START T>
<COMMIT T>
<ABORT T>
<T, X, old_v, new_v>
- T has written element X; its **old** value was old_v, and its **new** value is new_v

## Undo/Redo-Logging Rule

UR1: If T modifies X, then <T,X,u,v> must be written to disk before X is written to disk

Note: we are free to OUTPUT early or late (I.e. before or after <COMMIT T>)

| Undo | Redo | Undo/redo |
|---|---|---|
| OUTPUT<br>OUTPUT<br><COMMIT T> | <COMMIT T><br>OUTPUT<br>OUTPUT | OUTPUT<br><COMMIT T><br>OUTPUT |

| Action | T | Mem A | Mem B | Disk A | Disk B | Log (memory) | Log (disk) |
|---|---|---|---|---|---|---|---|
| | | | | | | <START T> | |
| READ(A,t) | 8 | 8 | | 8 | 8 | | |
| t := t*2 | 16 | 8 | | 8 | 8 | | |
| WRITE(A,t) | 16 | 16 | | 8 | 8 | <T,A,8,16> | |
| READ(B,t) | 8 | 16 | 8 | 8 | 8 | | |
| t := t*2 | 16 | 16 | 8 | 8 | 8 | | |
| WRITE(B,t) | 16 | 16 | 16 | 8 | 8 | <T,B,8,16> | |
| FLUSH LOG | | | | | | | <START T><br><T, A, 8, 16><br><T, B, 8, 16> |
| OUTPUT(A) | 16 | 16 | 16 | 16 | 8 | | |
| | | | | | | **<COMMIT T>** | |
| FLUSH LOG | | | | | | | **<COMMIT T>** |
| OUTPUT(B) | 16 | 16 | 16 | 16 | 16 | | |

## Recovery is more complex with undo/redo logging.

1. Redo all committed transactions, starting at the beginning of the log
2. Undo all incomplete transactions, starting from the end of the log

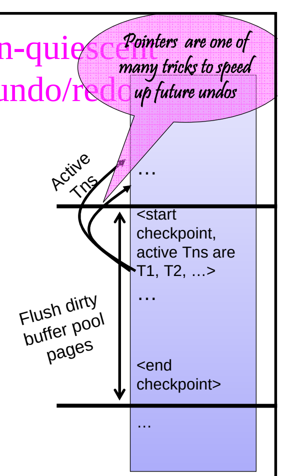
REDO ↓

```
<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
…
…
```

UNDO ↑

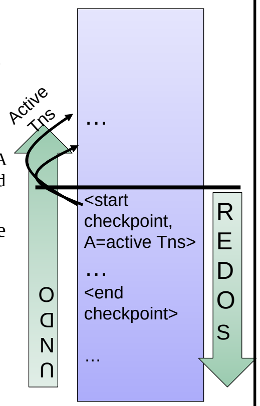
## Algorithm for non-quiescent checkpoint for undo/redo

1. Write <start checkpoint, list of all active transactions> to log
2. Flush log to disk
3. Write to disk **all** dirty buffers, whether or not their transaction has committed
   (this implies some log records may need to be written to disk)
4. Write <end checkpoint> to log
5. Flush log to disk



## Algorithm for undo/redo recovery with nonquiescent checkpoint

1. Backwards undo pass (end of log to start of last *completed* checkpoint)
   a. C = transactions that committed after the checkpoint started
   b. Undo actions of transactions that (are in A or started after the checkpoint started) and (are not in C)
2. Undo remaining actions by incomplete transactions
   a. Follow undo chains for transactions in (checkpoint active list) – C
3. Forward pass (start of last completed checkpoint to end of log)
   a. Redo actions of transactions in C




## Examples
what to do at recovery time?

…
T1 wrote A, …
…
checkpoint start (T1 active)
…
T1 wrote B, …
…
checkpoint end
…
T1 wrote C, …
…

Undo T1 (undo A, B, C)

no <T1 commit>

## Examples
what to do at recovery time?

…
T1 wrote A, …
…
checkpoint start (T1 active)
…
T1 wrote B, …
…
checkpoint end
…
T1 wrote C, …
…
T1 commit

Redo T1: (redo B, C)

## Concurrency Control

## A transaction is a sequence of operations that must be executed as a whole.

**BUSEY SAVINGS**
Winslett
$1000

Transfer $500

1. Debit savings
2. Credit checking

**BUSEY CHECKING**
Winslett
$0

**Either both (1) and (2) happen or neither!**

**Every DB action takes place inside a transaction.**

## We abstract away most of the application code when thinking about transactions.

*User's point of view*

Transfer $500

*Code writer's point of view*

1. Debit savings
2. Credit checking

*Concurrency control's & recovery's point of view*

Read Balance1
Write Balance1
Read Balance2
Write Balance2

## Schedule: The order of execution of operations of two or more transactions.

Schedule S1

| Transaction1 | Transaction2 |
| --- | --- |
| R(A) | |
| R(C) | |
| W(A) | |
| R(B) | |
| W(C) | |
| | R(B) |
| | W(B) |
| W(B) | |

Read data item A (typically a tuple)

Time

## Why do we need transactions?

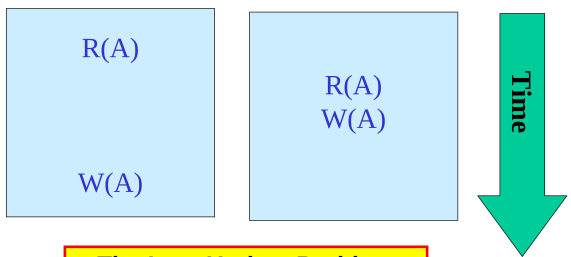
| **Transaction 1:** Add $100 to account A | **Transaction 2:** Add $200 to account A |
| --- | --- |
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |

Time

## What will be the final account balance?

**Transaction 1:** Add $100 to account A

**Transaction 2:** Add $200 to account A

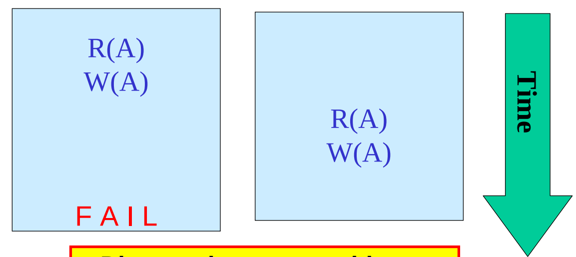



**The Lost Update Problem**

## What will be the final account balance?

**Transaction 1:** Add $100 to account A

**Transaction 2:** Add $200 to account A




**Dirty reads cause problems.**

## **Abort** or **roll back** are the official words for "fail".

Commit

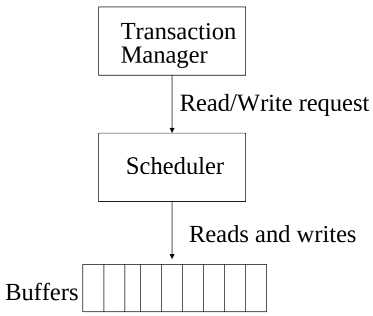
All your writes will definitely absolutely be recorded and will not be undone, and all the values you read are committed too.

Abort/rollback

Undo all of your writes!

*The concurrent execution of transactions must be such that each transaction appears to execute in isolation.*

## Scheduler

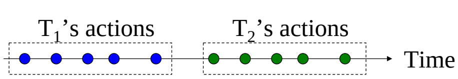



## Schedule

- Time-ordered sequence of the important actions taken by one or more transactions
- Consider only the READ and WRITE actions, and their orders; ignore the INPUT and OUTPUT actions
  - An element in a buffer is accessed by multiple transactions

## Serial Schedule

- If any action of transaction $T_1$ precedes any action of $T_2$, then all action of $T_1$ precede all action of $T_2$
- The correctness principle tells us that every serial schedule will preserve consistency of the database state




## Example 1: ($T_1$, $T_2$)

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| | | 25 | 25 |
| READ(A, t) | | | |
| t := t + 100 | | | |
| WRITE(A, t) | | 125 | |
| READ(B, t) | | | |
| t := t + 100 | | | |
| WRITE(B, t) | | | 125 |
| | READ(A, s) | | |
| | s := s * 2 | | |
| | WRITE(A, s) | 250 | |
| | READ(B, s) | | |
| | s := s * 2 | | |
| | WRITE(B, s) | | 250 |

## Example 2: ($T_2$, $T_1$)

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| | | 25 | 25 |
| | READ(A, s)<br>s := s * 2<br>WRITE(A, s) | 50 | |
| | READ(B, s)<br>s := s * 2<br>WRITE(B, s) | | 50 |
| READ(A, t)<br>t := t + 100<br>WRITE(A, t) | | 150 | |
| READ(B, t)<br>t := t + 100<br>WRITE(B, t) | | | 150 |

## Serial Schedule is Not Necessarily Desirable

- Improved throughput
  - I/O activity can be done in parallel with processing at CPU
- Reduced average waiting time
  - If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete

## A schedule is serializable if it is guaranteed to give the same final result as some serial schedule.

Which of these are serializable?

| | |
|---|---|
| Read(A) | |
| | Read(A) |
| | Write(A) |
| Write(A) | |
| Read(B) | |
| Write(B) | |
| | Read(B) |
| | Write(B) |

| | |
|---|---|
| Read(A) | |
| | Read(A) |
| Write(A) | |
| | Write(A) |
| Read(B) | |
| Write(B) | |
| | Read(B) |
| | Write(B) |

| | |
|---|---|
| Read(A) | |
| Write(A) | |
| | Read(A) |
| | Write(A) |
| Read(B) | |
| Write(B) | |
| | Read(B) |
| | Write(B) |

## Notation for Transactions and Schedules

- We do not consider the details of local computation steps such as t := t + 100
- Only the reads and writes matter
- Action: $r_i(X)$ or $w_i(X)$
- Transaction Ti: a sequence of actions with subscript i
- Schedule S: a sequence of actions from a set of transactions T

## Examples

- T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;
- T2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$;
- S: $r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;

## Conflict-Serializability

- Commercial systems generally support *conflict-serializability*
  - Stronger notion than serializability
- Based on the idea of a conflict
- Turn a given schedule to a serial one by make as many nonconflicting swaps as we wish

## Conflicts

- A pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change

## Conflicting Swaps

- Two actions of the same transaction
  - E.g., $r_i(X)$; $w_i(Y)$
- Two writes of the same database element
  - E.g., $w_i(X)$; $w_j(X)$
- A read and a write of the same database element
  - E.g., $r_i(X)$; $w_j(X)$

## Nonconflicting swaps

- Any two actions of different transactions may be swapped unless:
  - They involve the same database element, and
  - At least one is a write
- Examples:
  1. $r_i(X); r_j(Y)$
  2. $r_i(X); w_j(Y)$ if X!= Y
  3. $w_i(X); r_j(Y)$ if X != Y
  4. $w_i(X); w_j(Y)$ if X != Y

## Conflict-serializable

- Two schedules are *conflict-equivalent* if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions
- A schedule is *conflict-serializable* if it is conflict-equivalent to a serial schedule
- Easy to check whether a schedule is conflict-serializable by examining a precedence graph

## Example

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

## Test for Conflict-Serializability

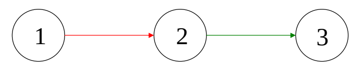
- Can decide whether or not a schedule S is conflict-serializable
- Ideas:
  - when there are conflicting actions that appear anywhere in S, the transactions performing those actions must appear in the same order in any conflict-equivalent serial schedule
  - Summarize those conflicting actions in a *precedence graph*

## Precedence Graphs

- $T_1$ takes precedence over $T_2$ ($T_1 <_S T_2$), if there are actions $A_1$ of $T_1$ and $A_2$ of $T_2$, s.t.
  - $A_1$ is ahead of $A_2$ in S
  - Both $A_1$ and $A_2$ involve the same database element
  - At least one of $A_1$ and $A_2$ is a written action
- Construct a precedence graph and ask if there are any cycles

## Example

S: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$;




S': $r_1(B)$; $w_1(B)$; $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$; $r_3(A)$; $w_3(A)$;

## Example

$S_1$: $r_2(A)$; $r_1(B)$; $w_2(A)$; $r_2(B)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $w_2(B)$;