

# CSE232A: Database System Principles

## More Concurrency Control and Transaction Processing

1

---

---

---

---

---

---

---

---

### Validation

Transactions have 3 phases:

(1) Read

- all DB values read
- writes to temporary storage
- no locking

(2) Validate

- check if schedule so far is serializable

(3) Write

- if validate ok, write to DB

2

---

---

---

---

---

---

---

---

### Key idea

- Make validation atomic
- If  $T_1, T_2, T_3, \dots$  is validation order, then resulting schedule will be conflict equivalent to  $S_s = T_1 T_2 T_3 \dots$

3

---

---

---

---

---

---

---

---

To implement validation, system keeps two sets:

- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

4

---

---

---

---

---

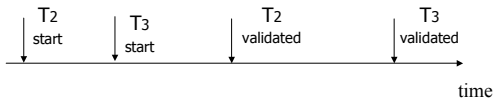
---

---

---

Example of what validation must prevent:

$RS(T_2) = \{B\}$       $RS(T_3) = \{A, B\} \neq \phi$   
 $WS(T_2) = \{B, D\}$       $WS(T_3) = \{C\}$



5

---

---

---

---

---

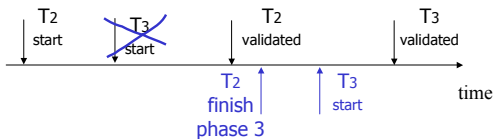
---

---

---

Example of what validation must <sup>allow</sup> prevent:

$RS(T_2) = \{B\}$       $RS(T_3) = \{A, B\} \neq \phi$   
 $WS(T_2) = \{B, D\}$       $WS(T_3) = \{C\}$



6

---

---

---

---

---

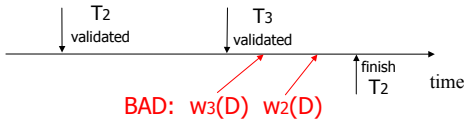
---

---

---

Another thing validation must prevent:

RS(T<sub>2</sub>)={A}      RS(T<sub>3</sub>)={A,B}  
 WS(T<sub>2</sub>)={D,E}    WS(T<sub>3</sub>)={C,D}




---

---

---

---

---

---

---

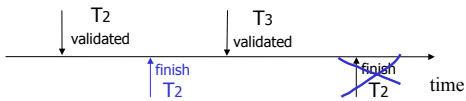
---

---

---

Another thing validation must prevent: <sup>allow</sup>

RS(T<sub>2</sub>)={A}      RS(T<sub>3</sub>)={A,B}  
 WS(T<sub>2</sub>)={D,E}    WS(T<sub>3</sub>)={C,D}




---

---

---

---

---

---

---

---

---

---

Validation rules for T<sub>j</sub>:

- (1) When T<sub>j</sub> starts phase 1:  
     ignore(T<sub>j</sub>) ← FIN
- (2) at T<sub>j</sub> Validation:  
     if check (T<sub>j</sub>) then  
         [ VAL ← VAL U {T<sub>j</sub>};  
         do write phase;  
         FIN ← FIN U {T<sub>j</sub>} ]

---

---

---

---

---

---

---

---

---

---

Check (Tj):

```
For Ti ∈ VAL - IGNORE (Tj) DO
  IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR
    Ti ∉ FIN ] THEN RETURN false;
RETURN true;
```

Is this check too restrictive ?

10

---

---

---

---

---

---

---

---

### Improving Check(Ti)

```
For Ti ∈ VAL - IGNORE (Tj) DO
  IF [ WS(Ti) ∩ RS(Tj) ≠ ∅ OR
    (Ti ∉ FIN AND WS(Ti) ∩ WS(Tj) ≠ ∅) ]
  THEN RETURN false;
RETURN true;
```

11

---

---

---

---

---

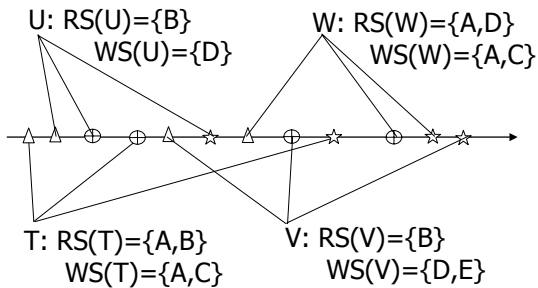
---

---

---

### Exercise:

△ start  
⊕ validate  
☆ finish



12

---

---

---

---

---

---

---

---

Validation (also called optimistic concurrency control) is useful in some cases:

- Conflicts rare
- System resources plentiful
- Have real time constraints

13

---

---

---

---

---

---

---

---

## Summary

Have studied C.C. mechanisms used in practice

- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation

14

---

---

---

---

---

---

---

---

## Chapter 10 More on transaction processing

Topics (which I doubt we'll get to all of them):

- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Long transactions (nested, compensation)

15

---

---

---

---

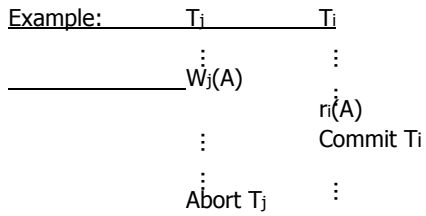
---

---

---

---

## Concurrency control & recovery



☛ Cascading rollback (Bad!)

16

---

---

---

---

---

---

---

---

---

---

- Schedule is conflict serializable
- $T_j \rightarrow T_i$
- But not recoverable

17

---

---

---

---

---

---

---

---

---

---

- Need to make "final" decision for each transaction:
  - **commit decision** - system guarantees transaction will or has completed, no matter what
  - **abort decision** - system guarantees transaction will or has been rolled back (has no effect)

18

---

---

---

---

---

---

---

---

---

---

To model this, two new actions:

- $C_i$  - transaction  $T_i$  commits
- $A_i$  - transaction  $T_i$  aborts

19

---

---

---

---

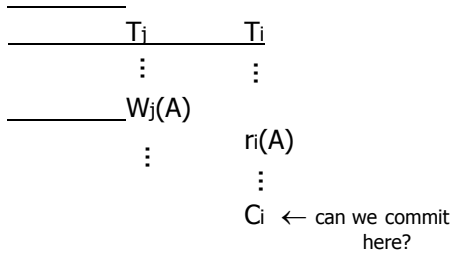
---

---

---

---

Back to example:



20

---

---

---

---

---

---

---

---

Definition

$T_i$  reads from  $T_j$  in  $S$  ( $T_j \Rightarrow_S T_i$ ) if

- (1)  $w_j(A) <_S r_i(A)$
- (2)  $a_j \not<_S r_i(A)$  ( $\not<$  : does not precede)
- (3) If  $w_j(A) <_S w_k(A) <_S r_i(A)$  then  $a_k <_S r_i(A)$

21

---

---

---

---

---

---

---

---

## Definition

Schedule  $S$  is recoverable if  
whenever  $T_j \Rightarrow_S T_i$  and  $j \neq i$  and  $C_i \in S$   
then  $C_j <_S C_i$

22

---

---

---

---

---

---

---

---

Note: in transactions, reads and writes  
precede commit or abort

- $\Rightarrow$  If  $C_i \in T_i$ , then  $r_i(A) < C_i$   
 $w_i(A) < C_i$
- $\Rightarrow$  If  $A_i \in T_i$ , then  $r_i(A) < A_i$   
 $w_i(A) < A_i$

- Also, one of  $C_i, A_i$  per transaction

23

---

---

---

---

---

---

---

---

How to achieve recoverable schedules?

24

---

---

---

---

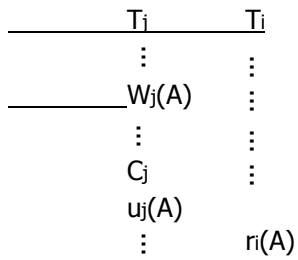
---

---

---

---

⇒ With 2PL, hold write locks to commit (strict 2PL)



---

---

---

---

---

---

---

---

⇒ With validation, no change!

---

---

---

---

---

---

---

---

- S is recoverable if each transaction *commits* only after all transactions from which it read have committed.
- S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.

---

---

---

---

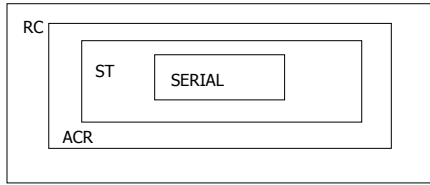
---

---

---

---

- S is strict if each transaction may *read and write* only items previously written by committed transactions.



28

---

---

---

---

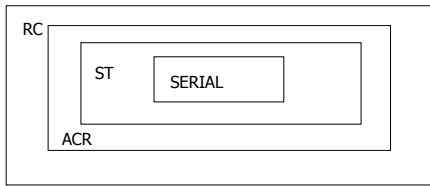
---

---

---

---

Where are serializable schedules?



29

---

---

---

---

---

---

---

---

### Examples

- Recoverable:
  - $w_1(A) w_1(B) w_2(A) r_2(B) c_1 c_2$
- Avoids Cascading Rollback:
  - $w_1(A) w_1(B) w_2(A) c_1 r_2(B) c_2$
- Strict:
  - $w_1(A) w_1(B) c_1 w_2(A) r_2(B) c_2$

Assumes  $w_2(A)$  is done without reading

30

---

---

---

---

---

---

---

---

## Deadlocks

- Detection
  - Wait-for graph
- Prevention
  - Resource ordering
  - Timeout
  - Wait-die
  - Wound-wait

31

---

---

---

---

---

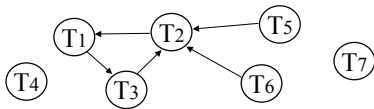
---

---

---

## Deadlock Detection

- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim



32

---

---

---

---

---

---

---

---

## Resource Ordering

- Order all elements  $A_1, A_2, \dots, A_n$
- A transaction  $T$  can lock  $A_i$  after  $A_j$  only if  $i > j$

Problem : Ordered lock requests not realistic in most cases

33

---

---

---

---

---

---

---

---

## Timeout

- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L

34

---

---

---

---

---

---

---

---

## Wait-die

- Transactions given a timestamp when they arrive ....  $ts(T_i)$
- $T_i$  can only wait for  $T_j$  if  $ts(T_i) < ts(T_j)$  ...else die

35

---

---

---

---

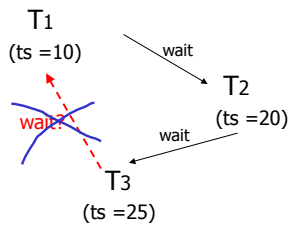
---

---

---

---

## Example:



36

---

---

---

---

---

---

---

---

### Wound-wait

- Transactions given a timestamp when they arrive ...  $ts(T_i)$
- $T_i$  wounds  $T_j$  if  $ts(T_i) < ts(T_j)$   
    else  $T_i$  waits

“Wound”:  $T_j$  rolls back and gives lock to  $T_i$

---

---

---

---

---

---

---

---

### User/Program commands

Lots of variations, but in general

- Begin\_work
- Commit\_work
- Abort\_work

---

---

---

---

---

---

---

---

### Nested transactions

User program:

⋮

  Begin\_work;

  ⋮

  ⋮

  If results\_ok, then commit work  
  else abort\_work

---

---

---

---

---

---

---

---

## Nested transactions

User program:

```
⋮  
Begin_work;  
  Begin_work;  
  ⋮  
  If results_ok, then commit work  
  else {abort_work; try something else...}  
⋮  
If results_ok, then commit work  
else abort_work
```

40

---

---

---

---

---

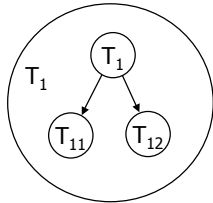
---

---

---

## Parallel Nested Transactions

```
T1: begin-work  
  ⋮  
  parallel:  
    T11: begin_work  
        ⋮  
        commit_work  
    T12: begin_work  
        ⋮  
        commit_work  
  ⋮  
  commit_work
```



41

---

---

---

---

---

---

---

---

Locking

Locking

What are we really locking?



42

---

---

---

---

---

---

---

---

Example:

T<sub>i</sub>     :  
  Read record r<sub>1</sub>  
      :  
  Read record r<sub>1</sub>  
      :  
  Modify record r<sub>3</sub>  
      :  
      :

          > do record locking

---

---

---

---

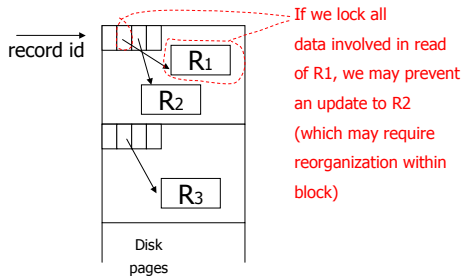
---

---

---

---

But underneath:



---

---

---

---

---

---

---

---

Solution: view DB at two levels

Top level: record actions  
          record locks  
          undo/redo actions — logical

e.g., Insert record(X,Y,Z)  
      Redo: insert(X,Y,Z)  
      Undo: delete

---

---

---

---

---

---

---

---

Low level: deal with physical details  
latch page during action  
 (release at end of action)

---

---

---

---

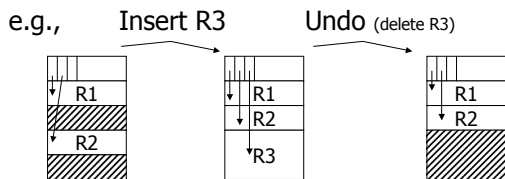
---

---

---

---

Note: undo does not return physical DB  
 to original state; only same logical state




---

---

---

---

---

---

---

---

**Related idea: Sagas**

- Long running activity:  $T_1, T_2, \dots, T_n$
- Each step/transaction  $T_i$  has a compensating transaction  $T_i^{-1}$
- Semantic atomicity: execute one of
  - $T_1, T_2, \dots, T_n$
  - $T_1, T_2, \dots, T_{n-1}, T_{n-1}^{-1}, T_{n-2}^{-1}, \dots, T_1^{-1}$
  - $T_1, T_2, \dots, T_{n-2}, T_{n-2}^{-1}, T_{n-3}^{-1}, \dots, T_1^{-1}$
  - $\vdots$
  - $T_1, T_1^{-1}$
  - nothing

---

---

---

---

---

---

---

---

## Summary

- Cascading rollback  
Recoverable schedule
- Deadlock
  - Prevention
  - Detectoin
- Nested transactions
- Multi-level view

---

---

---

---

---

---

---

---