

ICCS240 Database Management

# **Transactions, ACID & Concurrency Controls**

# Transaction Concept

- A transaction is a unit of program execution that accesses and possibly updates various data items.

E.g. transaction to transfer \$50 from account A to account B:

1. READ(A)
2. A := A-50
3. WRITE(A)
4. READ(B)
5. B := B+50
6. WRITE(B)

- Two main issues to deal with
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

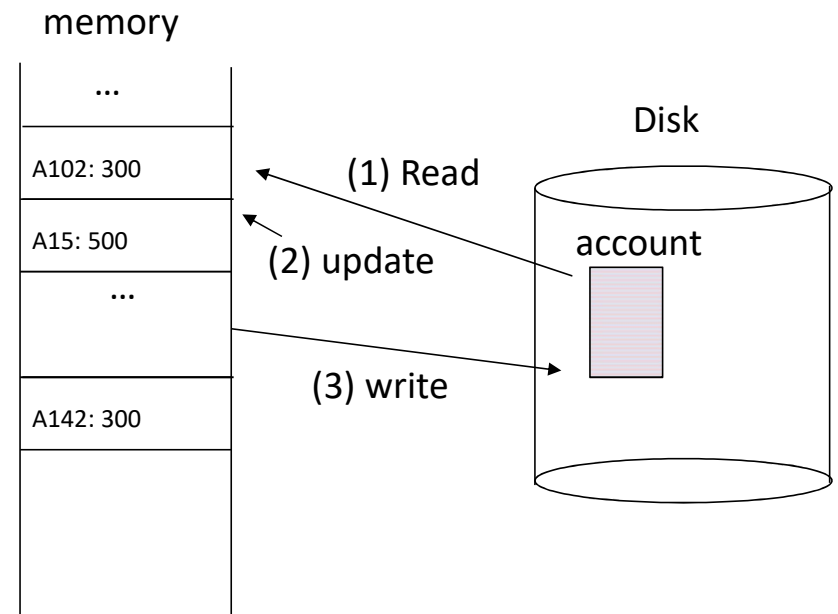
# Example of UPDATE in SQL

```
UPDATE account  
SET balance = balance - 50  
WHERE acct_no = A102
```

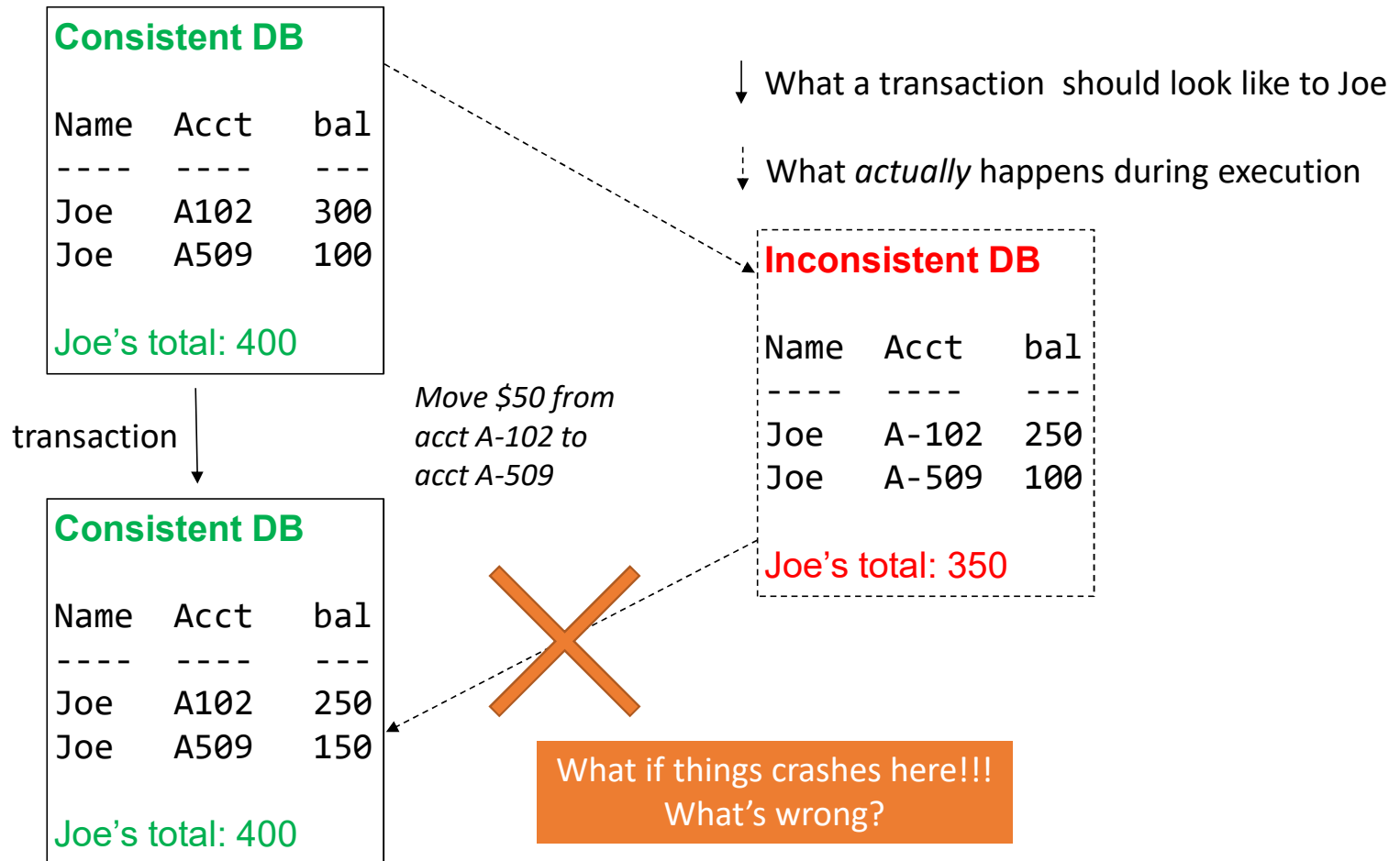
Transaction:

1. Read(A)
2.  $A \leftarrow A - 50$
3. Write(A)

What takes place:



# The Threat to Data Integrity: Crashes



# 3 Famous Anomalies

- Lost Updates
  - Two task  $T_1$  and  $T_2$  modify the same data and both commit
  - Final state shows effects of only  $T_1$  or  $T_2$ , but not both.
- Dirty Reads
  - $T$  reads data written by  $T'$  while  $T'$  has not yet committed.
- Inconsistent Read
  - One task  $T$  sees some but not all changes made by  $T'$

# 1st: Lost Updates

Client 1:

```
UPDATE Customer  
SET rentals= rentals + 1  
WHERE cname= 'Fred'
```

Client 2:

```
UPDATE Customer  
SET rentals= rentals + 1  
WHERE cname= 'Fred'
```

Two people attempt to rent two movies for Fred, from two different terminals.

What happens ?

## 2<sup>nd</sup>: Dirty Reads

**Client 1:** transfer \$100 acc1→acc2

```
X = Account1.balance
```

```
Account2.balance += 100
```

```
If (X>=100) Account1.balance -=100
```

```
else { /* rollback ! */
```

```
    account2.balance -= 100
```

```
    println("Denied !")
```

**Client 2:** transfer \$100 acc2→acc3

```
Y = Account2.balance
```

```
Account3.balance += 100
```

```
If (Y>=100) Account2.balance -=100
```

```
else { /* rollback ! */
```

```
    account3.balance -= 100
```

```
    println("Denied !")
```

What's wrong?

## 3rd: Inconsistent Reads

**Client 1:** move from gizmo→gadget

```
UPDATE Products  
SET quantity = quantity + 5  
WHERE product = 'gizmo'
```

```
UPDATE Products  
SET quantity = quantity - 5  
WHERE product = 'gadget'
```

**Client 2:** inventory....

```
SELECT sum(quantity)  
FROM Product
```



# Transactions

What?

- One or more operations, which reflect a single real-world transition.  
**A unit of work!**
- Can be executed concurrently

Why?

1. Updates can require multiple reads, writes on a DB.  
e.g., transfer \$50 from A102 to A509  
$$= \text{READ}(A); A \leftarrow A - 50; \text{WRITE}(A); \text{READ}(B); B \leftarrow B + 50; \text{WRITE}(B);$$
2. For performance reasons, DBs permit updates to be executed concurrently.

Concern: concurrent access/updates of data can compromise data integrity

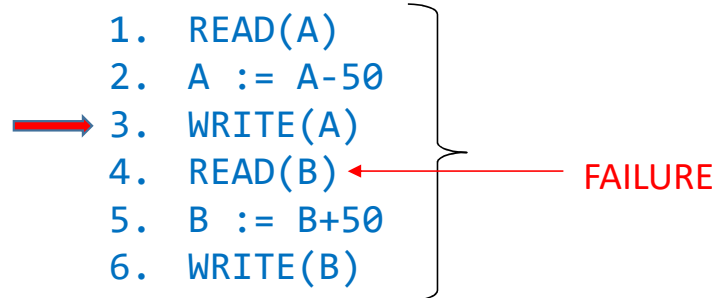
# ACID

Properties that transactions need to have:

- ✓ **Atomicity**: either all operations in a transaction take effect, or none
- ✓ **Consistency**: operations, taken together preserve DB consistency
- ✓ **Isolation**: intermediate, inconsistent states must be concealed from other transactions
- ✓ **Durability**: If a transaction successfully completes (“commits”), changes made to DB must persist, even if system crashes

# Demonstration of ACID

transaction to transfer \$50 from account A to account B:



**Consistency:** total value  $A+B$  unchanged by transaction

**Atomicity:** if transaction fails after 3. and before 6., then 3. should not affect DB

**Durability:** once user notified of transaction commit, updates to A,B should not be undone by system failure

**Isolation:** other transactions should not be able to see A, B between steps 3-6

# Threat to ACID

- Programmer Error
  - e.g.: \$50 subtracted from A, \$30 added to B
    - threatens **consistency**
- System Failures
  - e.g.: crash after write(A) and before write(B)
    - threatens **atomicity**
  - e.g.: crash after write(B)
    - threatens **durability**
- Concurrency
  - e.g.: concurrent transaction reads A, B between steps 3-6
    - threatens **isolation**

# Isolation ...

Simplest way to guarantee: *forbid concurrent transactions!?*

But, concurrency is desirable:

- Achieves better throughput (TPS: transactions per second)  
one transaction can use CPU while another is waiting for disk to service request
- Achieves better average response time  
short transactions don't need to get stuck behind long ones

Prohibiting concurrency is *not* an option!

So we need a **concurrency control**

# Concurrency Control

- Multiple concurrent transactions:  $T_1, T_2, \dots$
- Read/Write common elements:  $A_1, A_2, \dots$
- How to prevent unwanted interference?

The SCHEDULER is responsible for that

# Schedules

A **schedule** is a **sequence** of interleaved actions from all transactions.

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A, s)
READ(B, t)	READ(B, s)
t := t+100	s := s*2
WRITE(B, t)	WRITE(B, s)

# A Serial Schedule

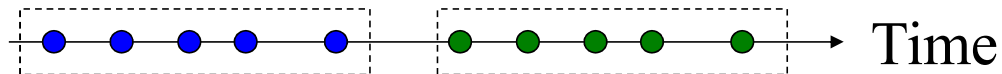
T1

```
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B, t)
```

T2

```
READ(A, s)
s := s*2
WRITE(A, s)
READ(B, s)
s := s*2
WRITE(B, s)
```

If any action of transaction  $T_1$  precedes any action of  $T_2$ , then all action of  $T_1$  precede all action of  $T_2$ .





A Schedule is serializable  
if it is equivalent to a serial schedule.

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A, s)
READ(B, t)	
t := t+100	
WRITE(B, t)	
	READ(B, s)
	s := s*2
	WRITE(B, s)

This is NOT a serial schedule,  
but is *serializable*

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

# Exercise: Which of these are serializable?

Assume WRITE(x) may change values of x

```
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 Transaction and Schedules

We do not consider the details of local computation steps such as  $t := t + 100$   
Assume *worst case* updates: so only the READs and WRITEs matter

- Actions:  $r_i(X)$  or  $w_i(X)$
- Transaction  $T_i$ : a sequence of actions
- Schedule  $S$ : a sequence of actions from a set of transaction  $\mathcal{T}$ .

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

# Conflicts

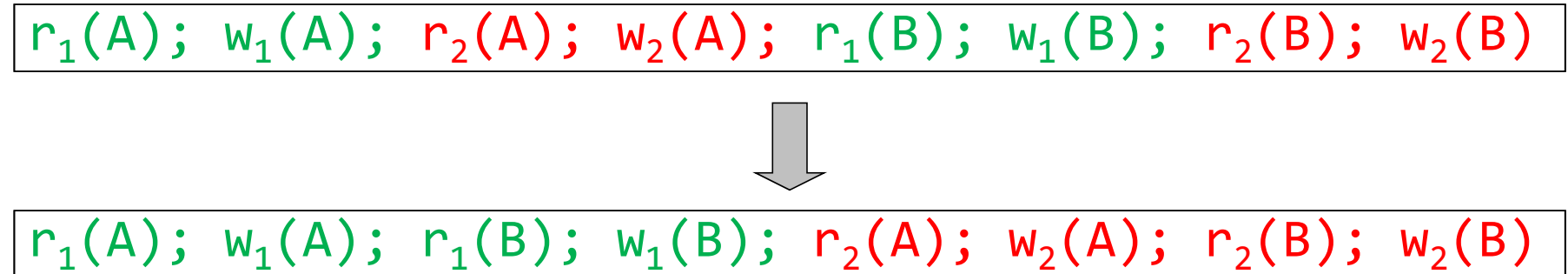
WRITE-READ	: WR
READ-WRITE	: RW
WRITE-WRITE	: WW

Two actions by same transaction $T_i$ :	$r_i(X); w_i(Y)$
Two writes by $T_i, T_j$ to the same element $X$ :	$w_i(X); w_j(X)$
Read/write by $T_i, T_j$ to same element:	$w_i(X); r_j(X)$ or $r_i(X); w_j(Y)$

A **conflict** means: you cannot *swap* the two operations

# Conflict Serializability

A schedule is **conflict serializable** if  
it can be transformed into a **serial schedule**  
by a **series of swaps** of *adjacent non-conflicting actions*



# The Precedence Graph Test

– Is a schedule  $S$  conflict-serializable?

Build a graph of all transaction  $T_i$  in  $S$  such that

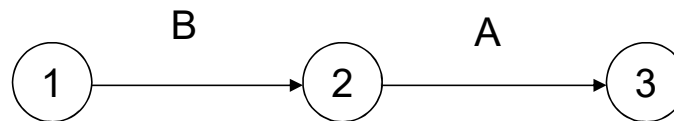
- Vertices are denoted by transaction  $T_i$
- Edges from  $T_i$  to  $T_j$  if  $T_i$  makes an action that conflicts with one of  $T_j$  and that  $T_i$ 's action comes first.

Then,

If the graph has no cycles, then  $S$  is conflict serializable.

## Example 1:

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



This schedule is conflict-serializable

## Exercise

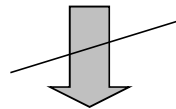
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$
--



# View Equivalence

A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$



$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but not conflict-equivalent

# View Equivalence

Two schedules  $S$  and  $S'$  are **view equivalent** if:

- If  $T$  reads an initial value of  $A$  in  $S$ ,  
then  $T$  also reads the initial value of  $A$  in  $S'$ .
- If  $T$  reads a value of  $A$  written by  $T'$  in  $S$ ,  
then  $T$  also reads a value of  $A$  written by  $T'$  in  $S'$ .
- If  $T$  writes the final value of  $A$  in  $S$ ,  
then it writes the final value of  $A$  in  $S'$ .

A schedule is view serializable if it is *view equivalent* to a serial schedule

If a schedule is **conflict serializable**, then it is also **view serializable**. But not vice versa

# Schedule with Aborted Transactions

When a transaction aborts, the recovery manager undoes its updates  
But some of its updates may have affected other transactions !

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Cannot abort T1 because cannot undo T2

# Recoverable Schedules

A schedule  $S$  is recoverable if:

- It is *conflict-serializable*, and
- Whenever a transaction  $T$  commits, *all* transactions who have written elements read by  $T$  have already committed

# Examples

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Non-recoverable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Abort	
	Commit

Recoverable

# Cascading Aborts

If a transaction  $T$  aborts, then we need to abort any other transaction  $T'$  that has read an element written by  $T$ .

A schedule is said to **avoid cascading aborts** if whenever a transaction read an element, the transaction that has last written it has already committed.

T1	T2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	...

Without cascading aborts

# “Schedule” Summary

## Serializability

Serial

Serializable

Conflict Serializable

View Serializable

## Recoverability

Recoverable

(Avoid) Cascade Aborts

# Scheduler ensures serializability

The scheduler is the module that schedules the transaction's actions, ensuring serializability.

Two main approaches

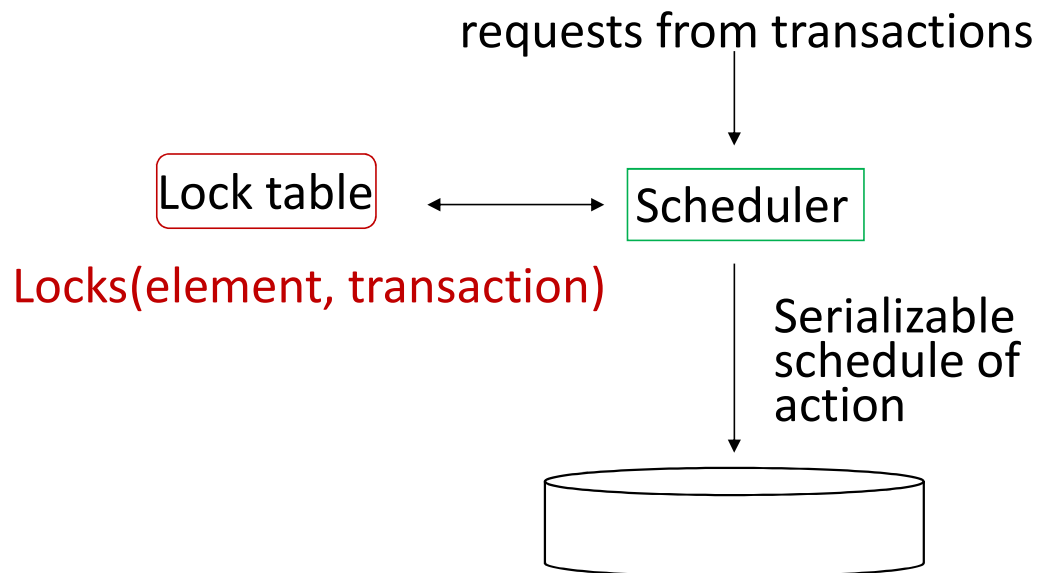
- *Pessimistic* scheduler: uses **LOCKS**
- *Optimistic* scheduler: **time stamps, validation**

Not covered!



# Locks

A scheduler uses a lock table to guide decisions



Notion:  $l_i(X) = T_i$  requests lock on element  $X$   
 $u_i(X) = T_i$  releases lock on element  $X$

## Consistency of transactions:

1. A transaction can only read or write an element if it previously requested a lock on that element and hasn't yet released the lock
2. If a transaction locks an element, it must later unlock that element

**Legality:** No two transactions may have locked the same element without one having first released the lock

## Example: A legal but not serializable schedule

T1

---

$L_1(A)$ ; READ(A, t)  
t := t+100  
WRITE(A, t);  $U_1(A)$ ;

$L_1(B)$ ; READ(B, t)  
t := t+100  
WRITE(B, t);  $U_1(B)$ ;

T2

$L_2(A)$ ; READ(A, s)  
s := s\*2  
WRITE(A, s);  $U_2(A)$ ;  
 $L_2(B)$ ; READ(B, s)  
s := s\*2  
WRITE(B, s);  $U_2(B)$ ;

# 2-Phase Locking (2PL)

## – no new locks once you've given up

- In every transaction, all lock requests must precede all unlock requests.
- This ensures conflict serializability!

T1	T2
<code>L<sub>1</sub>(A); L<sub>1</sub>(B); READ(A, t)</code> <code>t := t+100</code> <code>WRITE(A, t); U<sub>1</sub>(A)</code>	
	<code>L<sub>2</sub>(A); READ(A, s)</code> <code>s := s*2</code> <code>WRITE(A, s);</code> <code>L<sub>2</sub>(B); DENIED...</code>
<code>READ(B, t)</code> <code>t := t+100</code> <code>WRITE(B, t); U<sub>1</sub>(B);</code>	
	<code>...GRANTED; READ(B, s)</code> <code>s := s*2</code> <code>WRITE(B, s); U<sub>2</sub>(A); U<sub>2</sub>(B);</code>

Now it is conflict-serializable

# Now, it is non-recoverable ☹️

T1

$L_1(A)$ ;  $L_1(B)$ ; READ(A, t)

t := t+100

WRITE(A, t);  $U_1(A)$

READ(B, t)

t := t+100

WRITE(B,t);  $U_1(B)$ ;

**Abort/Rollback**

T2

$L_2(A)$ ; READ(A,s)

s := s\*2

WRITE(A,s);

$L_2(B)$ ; **DENIED...**

**...GRANTED**; READ(B,s)

s := s\*2

WRITE(B,s);  $U_2(A)$ ;  $U_2(B)$ ;

**Commit**

*We should never  
have let T2 commit.*

## 2PL does not guarantee recoverable, so ...

**Commit** transaction  $T$  only after all transactions that wrote data that  $T$  read have committed

**Or** only let a transaction read an item after the transaction that last wrote this item has committed

### **Strict 2PL:**

2PL + a transaction releases its locks *only after* it has committed.

# Deadlocks

Transaction  $T_1$  waits for a lock held by  $T_2$ ;

But  $T_2$  waits for a lock held by  $T_3$ ;

While  $T_3$  waits for . . . .

... and  $T_{100}$  waits for a lock held by  $T_1$ !! ---- **Cycle!**

## Deadlock avoidance

- Acquire locks in pre-defined order
- Acquire all locks at once before starting

## Deadlock detection

Timeouts

Wait-for graph (*this is what commercial systems use*)

# In general, the Locking Scheduler looks like ...

## Task 1:

Add lock/unlock requests to transactions

- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- Ensure Strict 2PL !

## Task 2:

Execute the locks accordingly

- When a lock is requested, check the lock table
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

# Concurrency Control by Timestamps

Main variant:

The timestamp order defines  
the serialization order of the transaction

Will generate a schedule that is view-equivalent to a serial schedule, and recoverable



# Timestamp

$TS(T)$  is a timestamp of transaction  $T$ .

With each element  $X$ , associate:

- $RT(X)$  = the highest timestamp of any transaction  $T$  that read  $X$ .
- $WT(X)$  = the highest timestamp of any transaction  $T$  that wrote  $X$ .
- $C(X)$  = the commit bit:
  - true when transaction with highest timestamp that wrote  $X$  committed.

# Example

For any two *conflicting actions*, ensure that their order is the serialized order:

In each of these cases:

- $w_U(X) \dots r_T(X)$
- $r_U(X) \dots w_T(X)$
- $w_U(X) \dots w_T(X)$



Read too late ?

When  $T$  requests  $r_T(X)$ , need to check  $TS(U) \leq TS(T)$

# Timestamp to ensure recoverable

Recall the definition:

if a transaction reads an element,  
then the transaction that wrote it must have already committed

Use the commit bit  $C(X)$  to keep track if the transaction that last wrote  $X$  has committed

## Some consideration:

- T wants to read X, and  $TS(T) < WT(X)$

START(T) ... START(U) ...  $w_U(X)$  ...  $r_T(X)$

- T wants to write X, and  $TS(T) < RT(X)$

START(T) ... START(U) ...  $r_U(X)$  ...  $w_T(X)$

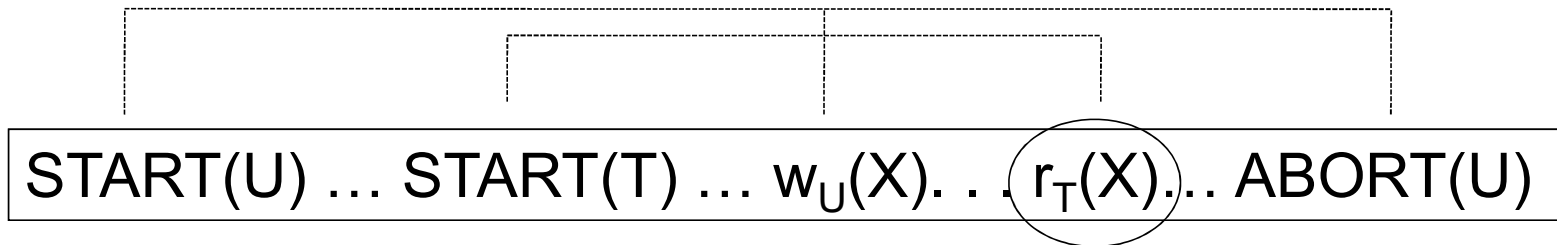
- T wants to write X, and  $TS(T) \geq RT(X)$  but  $WT(X) > TS(T)$

START(T) ... START(V) ...  $w_V(X)$  ...  $w_T(X)$

**So need to ROLLBACK T**

# Ensuring Recoverability (1)

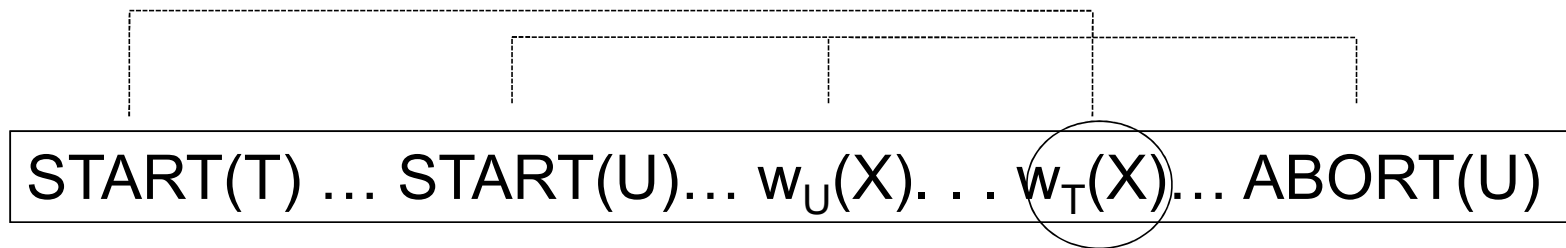
- T wants to read X, and  $WT(X) < TS(T)$
- Seems OK, but...



If  $C(X)=\text{false}$ , T needs to wait for it to become true

## Ensuring Recoverability (2)

- T wants to read X, and  $WT(X) > TS(T)$
- Seems OK not to write at all, but...



If  $C(X)=\text{false}$ , T needs to wait for it to become true

# Simplified Timestamp-based Scheduling

Only for transactions that do not abort

## Transaction $T$ wants to read element $X$

If  $TS(T) < WT(X)$ , then ROLLBACK

Else if  $C(X) = \text{false}$ , then WAIT until  $C(X) = \text{true}$  or ABORT

Else READ and update  $RT(X)$  to larger of  $TS(T)$  or  $RT(X)$

## Transaction $T$ wants to write element $X$

If  $TS(T) < RT(X)$ , then ROLLBACK

Else if  $TS(T) < WT(X)$ ,

    If  $C(X) = \text{false}$ , then WAIT

    Else IGNORE WRITE & continue --- (*Thomas WRITE rule*)

Otherwise, WRITE and update  $WT(X) = TS(T)$  and set  $C(X) = \text{false}$

# Example

T1	T2	T3	A	B	C
200	150	175	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
r1(B)					
	r2(A)				
		r3(C)			
w1(B)					
w1(A)					
	w2(C)				
		w3(A)			

TS(T2) ≥ RT(C) & WT(C) < TS(T2)  
Writing too late! Check C(C);



# Timestamp vs. Lock

## Timestamp

- Optimistic
  - Poor when there are many conflicts (rollbacks)
  - Great when there are few conflicts
- Storage: Read and write times for recently accessed database elements

## Lock

- *Lock delay transactions by avoid rollback!*
- Pessimistic
  - Great when there are many conflicts
  - Poor when there are few conflict (lock delays)
- Storage: space in the lock table  $\propto$  # elements locked

## Compromise

READ ONLY transactions  $\rightarrow$  timestamps

READ/WRITE transactions  $\rightarrow$  locks

# Multi-version Concurrency Control

- When update data element, database will not overwrite original item with new data, but creates a new version of the element.
- Thus, there are multiple versions stored.
- The version that each transaction sees depends on isolation level implemented, e.g., snapshot isolation – a transaction observes a state of data as when the transaction started.
- Issues to consider is how/when to remove version that become obsolete and will no longer be read.

Not cover in this class,  
but you may read more from the textbook.

# Transaction in MySQL

Example from <https://www.w3resource.com/mysql/mysql-transaction.php>

# Transaction in MySQL

```
START TRANSACTION
```

```
{ command1 }
```

```
{ command2 }
```

```
...
```

```
COMMIT (or ROLLBACK)
```

```
SET autocommit = {0 | 1}
```

By default, MySQL runs with **autocommit mode enabled**.

This means that as soon as you execute a statement that updates (modifies) a table, MySQL stores the update on disk to make it permanent.

The change cannot be rolled back.

```
mysql> /* CASE STUDY 1: */
```

```
mysql> select * from student;
```

STUDENT_ID	NAME	REG_CLASS
2	Neena Kochhar	9
3	Lex De Haan	9
4	Alexander Hunold	11

```
mysql> update STUDENT set ST_CLASS=8 where STUDENT_ID=2;
```

```
mysql> select * from STUDENT;
```

STUDENT_ID	NAME	REG_CLASS
2	Neena Kochhar	8
3	Lex De Haan	9
4	Alexander Hunold	11

```
mysql> ROLLBACK;
```

```
mysql> select * from student;
```

STUDENT_ID	NAME	REG_CLASS
2	Neena Kochhar	8
3	Lex De Haan	9
4	Alexander Hunold	11

```
mysql> /* There is no rollback as MySQL runs with autocommit  
mode enabled!!! */
```

```
mysql> /* CASE STUDY 2: */
```

SET autocommit=0;

```
mysql> START TRANSACTION;
```

```
mysql> update STUDENT set ST_CLASS=10 where STUDENT_ID=2;
```

```
mysql> select * from STUDENT;
```

STUDENT_ID	NAME	REG_CLASS
2	Neena Kochhar	10
3	Lex De Haan	9
4	Alexander Hunold	11

```
mysql> ROLLBACK;
```

```
mysql> select * from STUDENT;
```

STUDENT_ID	NAME	REG_CLASS
2	Neena Kochhar	8
3	Lex De Haan	9
4	Alexander Hunold	11

# SAVEPOINT

- A **SAVEPOINT** is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction.

**SAVEPOINT SAVEPOINT\_NAME;**

SQL> **SAVEPOINT** SP1;  
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;  
1 row deleted.

SQL> **SAVEPOINT** SP2;  
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;  
1 row deleted.

SQL> **SAVEPOINT** SP3;  
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;  
1 row deleted.

SQL> **ROLLBACK TO** SP2;  
Rollback complete.

SQL> **RELEASE** SAVEPOINT SP\_3;  
Savepoint removed.



# The SET Transaction Command

- You can specify a transaction to be read only or read write.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

# Unused slides

Not covered in this class

# Concurrency Control by Validation

- Another type of **optimistic** concurrency control
- Maintains a record of what active transactions are doing
- Just before a transaction starts to write,  
it goes through a “validation phase”
- If there is a risk of physically unrealizable behavior, the transaction is rolled back

# Validation-based Scheduler

- Keep track of each transaction  $T$ 's
  - Read set  $RS(T)$ : the set of elements  $T$  read
  - Write set  $WS(T)$ : the set of elements  $T$  write
- Execute transactions in three phases:
  1. **Read.**  $T$  reads all the elements in  $RS(T)$
  2. **Validate.** Validate  $T$  by comparing its  $RS(T)$  and  $WS(T)$  with those in other transactions. If the validation fails,  $T$  is rolled back
  3. **Write.**  $T$  writes its values for the elements in  $WS(T)$

# Validation Rules

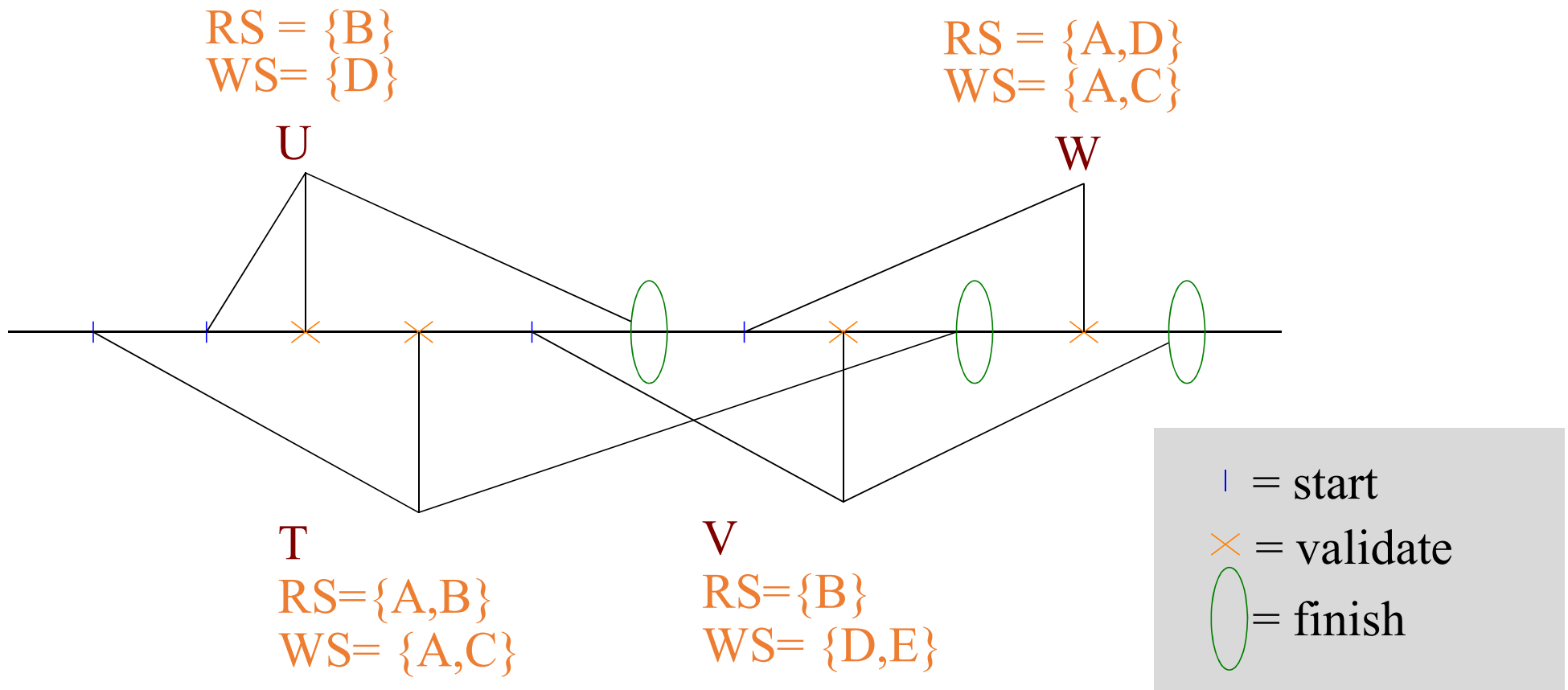
For each  $T$ ,

- maintain  $\langle T, START(T) \rangle$  : set of transactions that have started but not yet completed validation.
- maintain  $\langle T, START(T), VAL(T) \rangle$  : set of transactions that have been validated, but not yet finished.
- maintain  $\langle T, START(T), VAL(T), FIN(T) \rangle$  : set of transactions that have completed.

To validate a transaction  $T$ ,

1. Check that  $RS(T) \cap WS(U) = \emptyset$  for any *validated*  $U$  and  $START(T) < FIN(U)$ .
2. Check that  $WS(T) \cap WS(U) = \emptyset$  for any *validated*  $U$  that did not finish before  $T$  validated, i.e., if  $VAL(T) < FIN(U)$ .

# Example



# Technique Choices in Commercial Systems

- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL:
  - Multiversion concurrency control
- Oracle
  - Snapshot isolation even for SERIALIZABLE