CSE544
Transactions: Concurrency Control

Lectures #5-6

Thursday, January 20, 2011

Tuesday, January 25, 2011

Reading Material for Lectures 5-7

Main textbook (Ramakrishnan and Gehrke):

- Chapters 16, 17, 18

Mike Franklin's paper

More background material: Garcia-Molina, Ullman, Widom:

- Chapters 17.2, 17.3, 17.4
- Chapters 18.1, 18.2, 18.3, 18.8, 18.9

Transactions

- The problem: An application must perform *several* writes and reads to the database, as a unity
- Solution: multiple actions of the application are bundled into one unit called *Transaction*

Turing Awards to Database Researchers

- Charles Bachman 1973 for CODASYL
- Edgar Codd 1981 for relational databases
- Jim Gray 1998 for transactions

The Need for Transactions

- What can go wrong ?
  - System crashes
  - Anomalies during concurrent access: three are famous

Crashes

Client 1:

```
UPDATE Accounts
SET balance= balance - 500
WHERE name= 'Fred'
```

Crash !

```
UPDATE Accounts
SET balance = balance + 500
WHERE name= 'Joe'
```

What's wrong ?

Three Famous Anomalies

- Lost update – what is it ?
- Dirty read – what is it ?
- Inconsistent read – what is it ?

The Three Famous anomalies

- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Dirty read
  - T reads data written by T' while T' has not committed
  - What can go wrong: T' write more data (which T has already read), or T' aborts
- Inconsistent read
  - One task T sees some but not all changes made by T'

1st Famous Anomaly: 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 ?

2nd Famous Anomaly: 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 Famous Anomaly: Inconsistent Read

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: Definition

- **A transaction** = one or more operations, which reflects a single real-world transition
  - Happens completely or not at all; all-or-nothing
- Examples
  - Transfer money between accounts
  - Rent a movie; return a rented movie
  - Purchase a group of products
  - Register for a class (either waitlisted or allocated)
- By using transactions, all previous problems disappear

Transactions in Applications

START TRANSACTION

[SQL statements]

COMMIT or ROLLBACK (=ABORT)

May be omitted: first SQL query starts txn

In ad-hoc SQL: each statement = one transaction

ACID Properties

- Atomic
  - What is it ?
- Consistent
  - What is it ?
- Isolated
  - What is it ?
- Durable
  - What is it ?

ACID Properties

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Concurrency Control

Multiple concurrent transactions T_1, T_2, …

They read/write common elements A_1, A_2, …

How can we prevent unwanted interference ?

The SCHEDULER is responsible for that

Schedules

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

Example

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

Serializable Schedule

A schedule is <u>*serializable*</u> if it is equivalent to a serial schedule

A Serializable 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 Non-Serializable Schedule

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

Serializable Schedules

The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

Serializable Schedules

The role of the scheduler is to ensure that the schedule is serializable

Q: Why not run only serial schedules ?
I.e. run one transaction after the other ?

A: Because of very poor throughput due to disk latency.

Lesson: main memory databases *may* do serial schedules only

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Schedule is serializable because t=t+100 and s=s+200 commute

We don’t expect the scheduler to schedule this

Ignoring Details

Assume worst case updates:

- We never commute actions done by transactions

As a consequence, we only care about reads and writes

- Transaction = sequence of R(A)'s and W(A)'s

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Conflicts

Write-Read – WR

Read-Write – RW

Write-Write – WW

Conflicts

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element

$w_i(X); r_j(X)$

$r_i(X); w_j(X)$

A "conflict" means: you can't swap the two operations

Conflict Serializability

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

Example:

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

$$\Downarrow$$

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

The Precedence Graph Test

Is a schedule conflict-serializable ?

Simple test:

Build a graph of all transactions T_i

Edge from T_i to T_j if T_i makes an action that conflicts with one of T_j and comes first

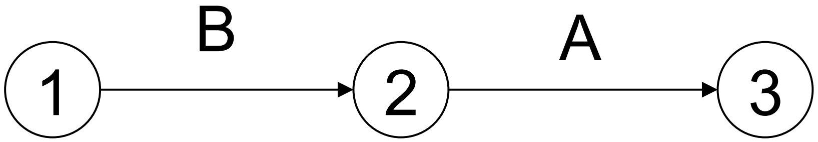
The test: if the graph has no cycles, then it is conflict serializable !

Example 1

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

Example 1

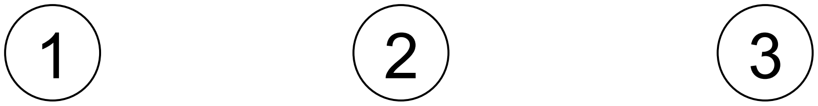
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

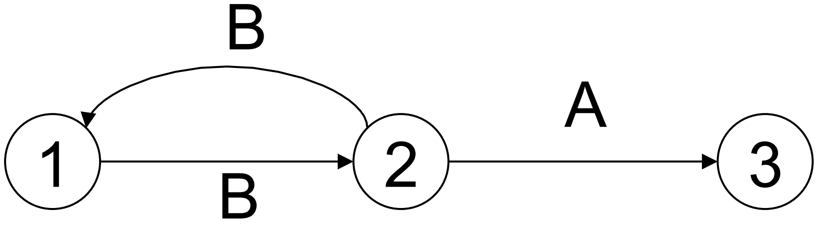
Example 2

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



Example 2

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



This schedule is NOT conflict-serializable

View Equivalence

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

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

Is this schedule conflict-serializable ?

View Equivalence

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

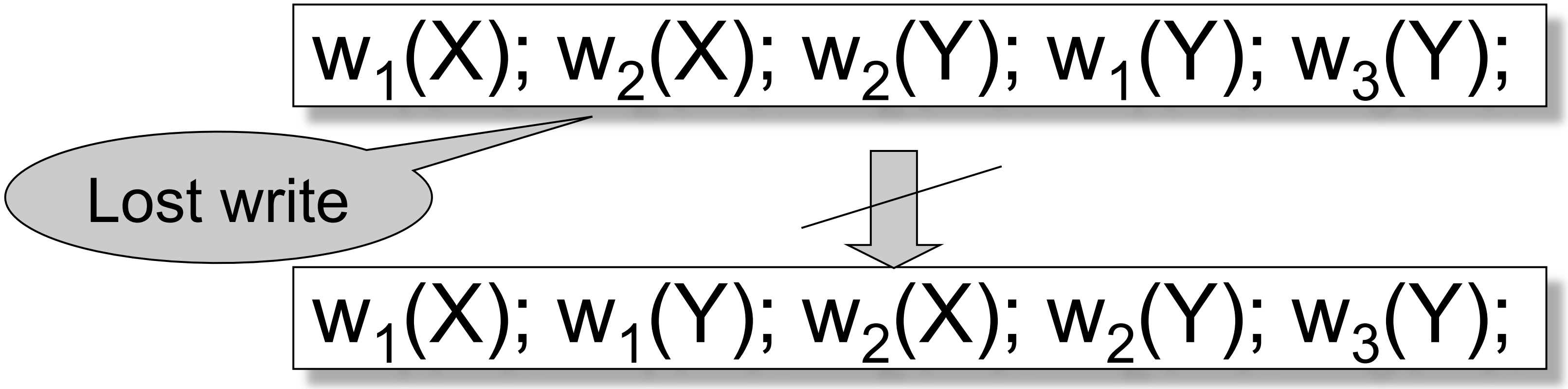
$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Is this schedule conflict-serializable ?

No…

View Equivalence

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



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

Lost write

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

Equivalent, but not conflict-equivalent

View Equivalence

T1	T2	T3
W1(X)		
	W2(X)	
	W2(Y)	
	CO2	
W1(Y)		
CO1		
		W3(Y)
		CO3

Lost

T1	T2	T3
W1(X)		
W1(Y)		
CO1		
	W2(X)	
	W2(Y)	
	CO2	
		W3(Y)
		CO3

Serializable, but not conflict serializable

View Equivalence

Two schedules S, 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’

View-Serializability

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

Remark:

If a schedule is *conflict serializable*, then it is also *view serializable*

But not vice versa

Schedules with Aborted Transactions

When a transaction aborts, the recovery manager undoes its updates

But some of its updates may have affected other transactions !

Schedules with Aborted 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 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

Recoverable Schedules

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

Nonrecoverable

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.

Avoiding Cascading Aborts

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

With cascading aborts

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

Without cascading aborts

Review of Schedules

Serializability

Serial

Serializable

Conflict serializable

View serializable

Recoverability

Recoverable

Avoiding cascading deletes

Review Questions

What is a *schedule* ?

What is a *serializable* schedule ?

What is a *conflict* ?

What is a *conflict-serializable* schedule ?

What is a *view-serializable* schedule ?

What is a *recoverable* schedule ?

When does a schedule avoid *cascading aborts* ?

Scheduler

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

Pessimistic Scheduler

Simple idea:

Each element has a unique lock

Each transaction must first acquire the lock before reading/writing that element

If the lock is taken by another transaction, then wait

The transaction must release the lock(s)

Notation

$l_i(A)$ = transaction T_i acquires lock for element A

$u_i(A)$ = transaction T_i releases lock for element A

A Non-Serializable Schedule

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

Example

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A); L_1(B)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s); U_2(A);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(B);

Scheduler has ensured a conflict-serializable schedule

But…

T1	T2
$L_1(A)$; READ(A, t)	
t := t+100	
WRITE(A, t); $U_1(A)$;	
	$L_2(A)$; READ(A,s)
	s := s*2
	WRITE(A,s); $U_2(A)$;
	$L_2(B)$; READ(B,s)
	s := s*2
	WRITE(B,s); $U_2(B)$;
$L_1(B)$; READ(B, t)	
t := t+100	
WRITE(B,t); $U_1(B)$;	

Locks did not enforce conflict-serializability !!! What's wrong ?

Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must preceed all unlock requests

This ensures conflict serializability ! (will prove this shortly)

Example: 2PL transactions

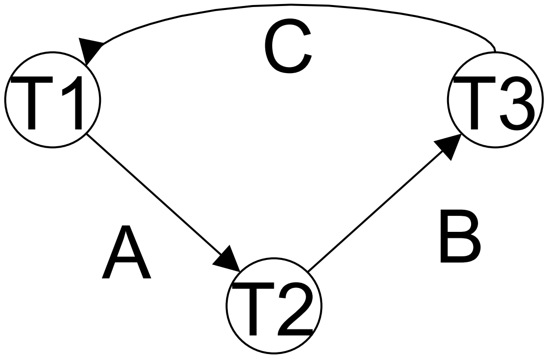
T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);

Now it is conflict-serializable

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following temporal cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$
$L_2(A) \rightarrow U_2(B)$
$U_2(B) \rightarrow L_3(B)$
$L_3(B) \rightarrow U_3(C)$
$U_3(C) \rightarrow L_1(C)$
$L_1(C) \rightarrow U_1(A)$

Contradiction

A New Problem: Non-recoverable Schedule

T1	T2
L_1(A); L_1(B); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(A); U_2(B);
Abort	**Commit**

What about Aborts?

2PL enforces conflict-serializable schedules

But does not enforce recoverable schedules

Strict 2PL

Strict 2PL: All locks held by a transaction are released when the transaction is completed

Schedule is recoverable

- Transactions commit only after all transactions whose changes they read also commit

Schedule avoids cascading aborts

- Transactions read only after the txn that wrote that element committed

Schedule is strict: read book

Lock Modes

Standard:

S = shared lock (for READ)

X = exclusive lock (for WRITE)

Lots of fancy locks:

U = update lock

- Initially like S
- Later may be upgraded to X

I = increment lock (for A := A + something)

- Increment operations commute

Lock Granularity

Fine granularity locking (e.g., tuples)

- High concurrency
- High overhead in managing locks

Coarse grain locking (e.g., tables, predicate locks)

- Many false conflicts
- Less overhead in managing locks

Alternative techniques

- Hierarchical locking (and intentional locks) [commercial DBMSs]
- Lock escalation

Deadlocks

Trasaction 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_{73} waits for a lock held by T_1 !!

Deadlocks

When T1 waits for T2, which waits for T3, which waits for T4, …, which waits for T1 – 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)

The Locking Scheduler

Task 1:
Add lock/unlock requests to transactions

Examine all READ(A) or WRITE(A) actions

Add appropriate lock requests

Ensure Strict 2PL !

The Locking Scheduler

Task 2:
Execute the locks accordingly

Lock table: a big, critical data structure in a DBMS !

When a lock is requested, check the lock table

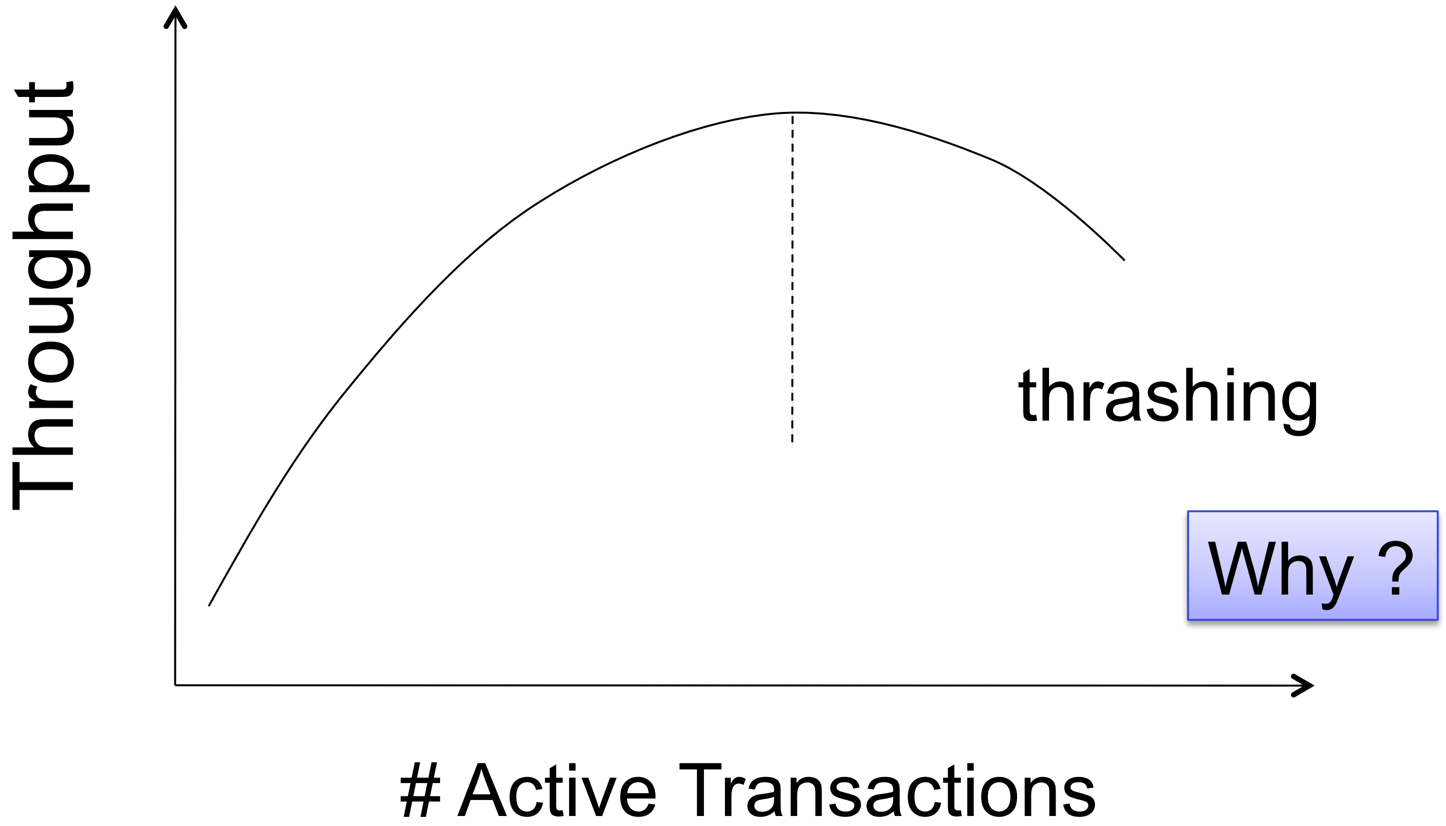
- Grant, or add the transaction to the element’s wait list

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

Lock Performance



Why ?

The Tree Protocol

An alternative to 2PL, for tree structures

E.g. B-trees (the indexes of choice in databases)

Because

- Indexes are hot spots!
- 2PL would lead to great lock contention

The Tree Protocol

Rules:

The first lock may be any node of the tree

Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B

Nodes can be unlocked in any order (no 2PL necessary)

"Crabbing"

- First lock parent then lock child
- Keep parent locked only if may need to update it
- Release lock on parent if child is not full

The tree protocol is NOT 2PL, yet ensures conflict-serializability !

Phantom Problem

So far we have assumed the database to be a *static* collection of elements (=tuples)

If tuples are inserted/deleted then the *phantom problem* appears

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Is this schedule serializable ?

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

This is conflict serializable ! What’s wrong ??

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘gizmo’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Suppose there are two blue products, X1, X2:

R1(X1),R1(X2),W2(X3),R1(X1),R1(X2),R1(X3)

Not serializable due to ***phantoms***

Phantom Problem

A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.

In our example:

T1: reads list of products

T2: inserts a new product

T1: re-reads: a new product appears !

Phantom Problem

In a ***static*** database:

Conflict serializability implies serializability

In a ***dynamic*** database, this may fail due to phantoms

Strict 2PL guarantees conflict serializability, but not serializability

Dealing With Phantoms

- Lock the entire table, or
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

Degrees of Isolation

Isolation level “serializable” (i.e. ACID)

- Golden standard
- Requires strict 2PL and predicate locking
- But often too inefficient
- Imagine there are few update operations and many long read operations

Weaker isolation levels

- Sacrifice correctness for efficiency
- Often used in practice (often **default**)
- Sometimes are hard to understand

Degrees of Isolation in SQL

Four levels of isolation

- All levels use **long-duration exclusive locks**
- READ UNCOMMITTED: no read locks
- READ COMMITTED: short duration read locks
- REPEATABLE READ:
  - Long duration read locks on individual items
- SERIALIZABLE:
  - All locks long duration and lock predicates

Trade-off: consistency vs concurrency

Commercial systems give choice of level

Isolation Levels in SQL

1. "Dirty reads"
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. "Committed reads"
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. "Repeatable reads"
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

ACID

Choosing Isolation Level

Trade-off: efficiency vs correctness

DBMSs give user choice of level

Always read docs!

Beware!!
- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly <u>A</u>CID

1. Isolation Level: Dirty Reads

"Long duration" WRITE locks

- Strict 2PL

No READ locks

- Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads

2. Isolation Level: Read Committed

"Long duration" WRITE locks

- Strict 2PL

"Short duration" READ locks

- Only acquire lock while reading (not 2PL)

Unrepeatable reads

- When reading same element twice, may get two different values

3. Isolation Level: Repeatable Read

“Long duration” READ and WRITE locks

- Strict 2PL

This is not serializable yet !!!

Why ?

4. Isolation Level Serializable

Deals with phantoms too

READ-ONLY Transactions

```
Client 1: START TRANSACTION
          INSERT INTO SmallProduct(name, price)
                    SELECT pname, price
                    FROM Product
                    WHERE price <= 0.99

          DELETE  FROM Product
                     WHERE price <=0.99
          COMMIT

Client 2: SET TRANSACTION READ ONLY
          START TRANSACTION
          SELECT count(*)
          FROM Product

          SELECT count(*)
          FROM SmallProduct
          COMMIT
```

Can improve performance

Optimistic Concurrency Control Mechanisms

Pessimistic:

- Locks

Optimistic

- Timestamp based: basic, multiversion
- Validation
- Snapshot isolation: a variant of both

Timestamps

Each transaction receives a unique timestamp TS(T)

Could be:

The system's clock

A unique counter, incremented by the scheduler

Timestamps

Main invariant:

The timestamp order defines
the serialization order of the transaction

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

Main Idea

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

In each of these cases

$w_U(X) \ . \ . \ . \ r_T(X)$

$r_U(X) \ . \ . \ . \ w_T(X)$

$w_U(X) \ . \ . \ . \ w_T(X)$

Read too late ?

Write too late ?

When T requests $r_T(X)$, need to check $TS(U) <= TS(T)$

Timestamps

With each element X, associate

RT(X) = the highest timestamp of any transaction U that read X

WT(X) = the highest timestamp of any transaction U that wrote X

C(X) = the commit bit: true when transaction with highest timestamp that wrote X committed

If element = page, then these are associated with each page X in the buffer pool

Simplified Timestamp-based Scheduling

Only for transactions that do not abort

Otherwise, may result in non-recoverable schedule

Transaction wants to read element X

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

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

Transaction wants to write element X

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

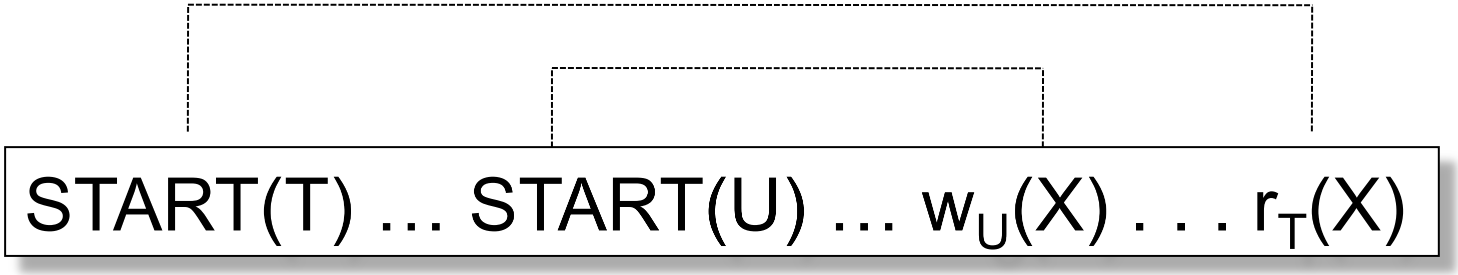
Else if TS(T) < WT(X) ignore write & continue (Thomas Write Rule)

Otherwise, WRITE and update WT(X) =TS(T)

Details

Read too late:

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



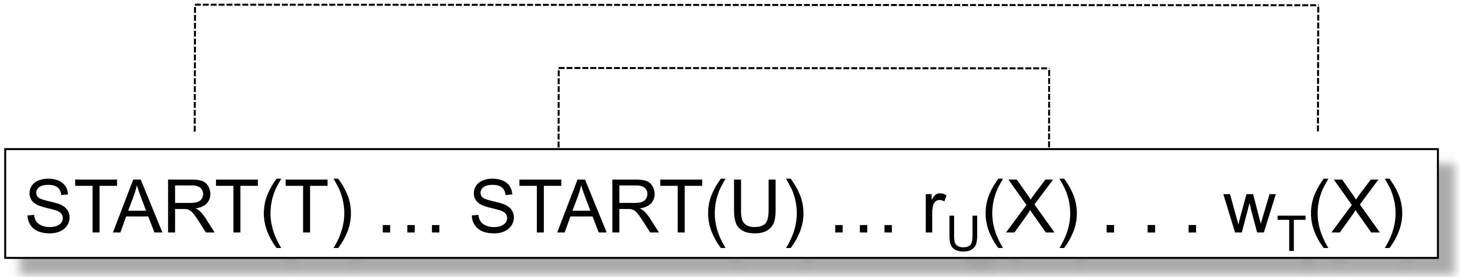
START(T) … START(U) … $w_U(X)$ . . . $r_T(X)$

Need to rollback T !

Details

Write too late:

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



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

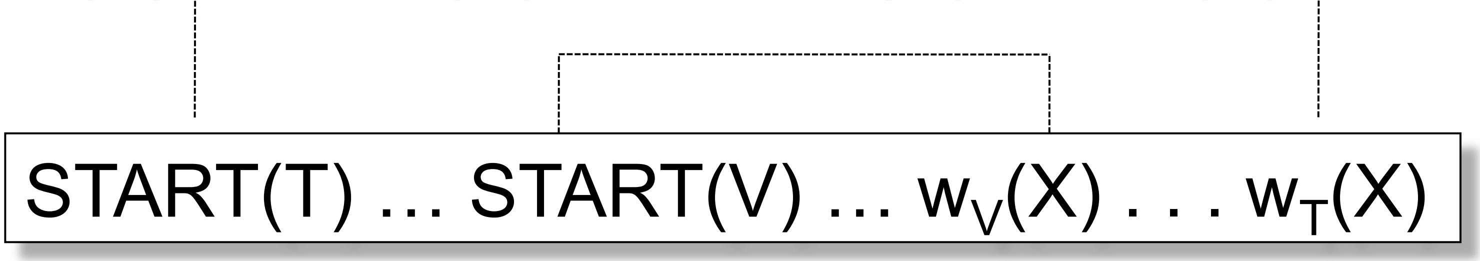
Need to rollback T !

Details

Write too late, but we can still handle it:

T wants to write X, and

TS(T) >= RT(X) but WT(X) > TS(T)



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

Don’t write X at all !

(Thomas’ rule)

View-Serializability

By using Thomas' rule we do not obtain a conflict-serializable schedule

But we obtain a view-serializable schedule

Ensuring Recoverable Schedules

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

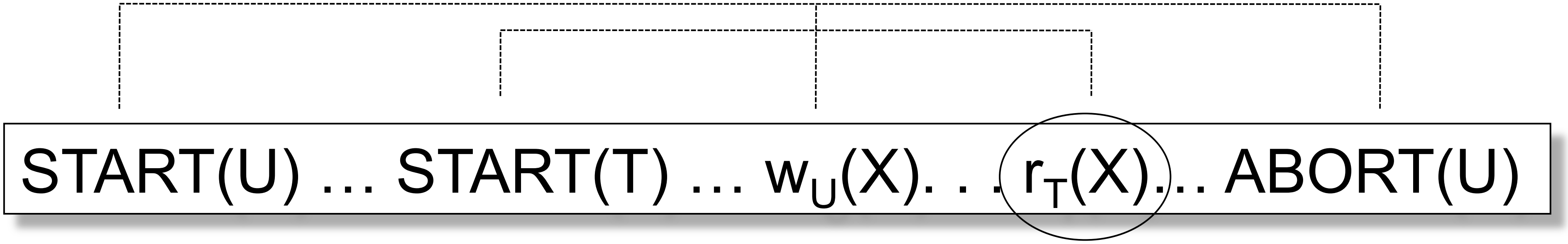
Use the commit bit C(X) to keep track if the transaction that last wrote X has committed

Ensuring Recoverable Schedules

Read dirty data:

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

Seems OK, but…



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

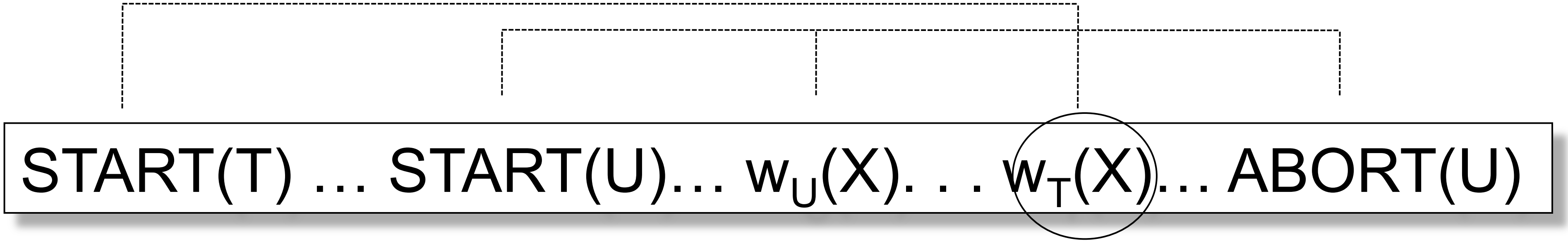
If C(X)=false, T needs to wait for it to become true

Ensuring Recoverable Schedules

Thomas' rule needs to be revised:

T wants to write X, and WT(X) > TS(T)

Seems OK not to write at all, but …



START(T) … START(U)… $w_U(X)$. . . $w_T(X)$… ABORT(U)

If C(X)=false, T needs to wait for it to become true

Timestamp-based Scheduling

```
Transaction wants to READ element X
    If TS(T) < WT(X)  then ROLLBACK
    Else If C(X) = false, then WAIT
    Else READ and update RT(X) to larger of TS(T) or RT(X)
```

```
Transaction wants to WRITE element X
    If TS(T) < RT(X) then ROLLBACK
    Else if TS(T) < WT(X)
        Then If C(X) = false then WAIT
                else IGNORE write (Thomas Write Rule)
    Otherwise, WRITE, and update WT(X)=TS(T), C(X)=false
```

Summary of Timestamp-based Scheduling

View-serializable

Recoverable

  Even avoids cascading aborts

Does NOT handle phantoms

  These need to be handled separately, e.g. predicate locks

Multiversion Timestamp

When transaction T requests r(X)
but WT(X) > TS(T), then T must rollback

Idea: keep multiple versions of X:
$X_t, X_{t-1}, X_{t-2}, \ldots$

$$\boxed{TS(X_t) > TS(X_{t-1}) > TS(X_{t-2}) > \ldots}$$

Let T read an older version, with appropriate timestamp

Details

When $w_T(X)$ occurs,

create a new version, denoted X_t where $t = TS(T)$

When $r_T(X)$ occurs,

find most recent version X_t such that $t < TS(T)$

Notes:

$WT(X_t) = t$ and it never changes

$RT(X_t)$ must still be maintained to check legality of writes

Can delete X_t if we have a later version X_{t1} and all active transactions T have $TS(T) > t1$

Concurrency Control by Validation

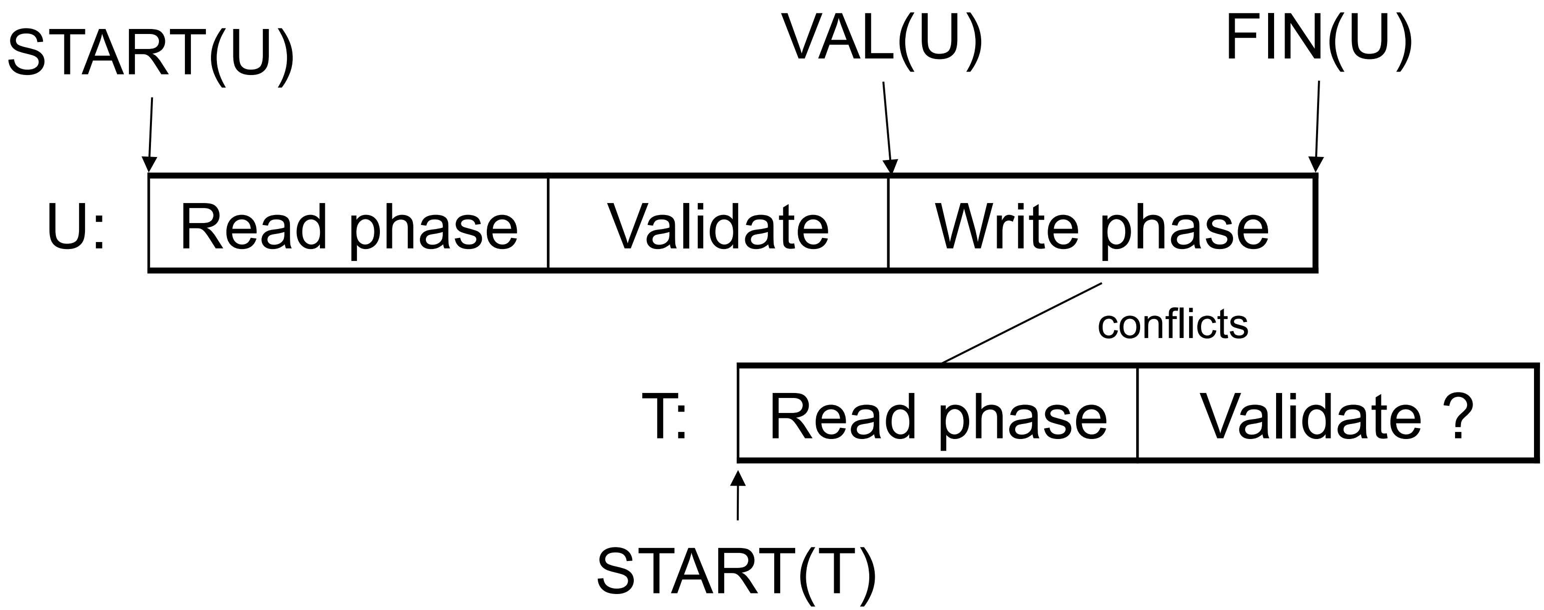
Each transaction T defines a *read set* RS(T) and a *write set* WS(T)

Each transaction proceeds in three phases:

- Read all elements in RS(T). Time = START(T)
- Validate (may need to rollback). Time = VAL(T)
- Write all elements in WS(T). Time = FIN(T)

Main invariant: the serialization order is VAL(T)

Avoid $r_T(X)$ - $w_U(X)$ Conflicts

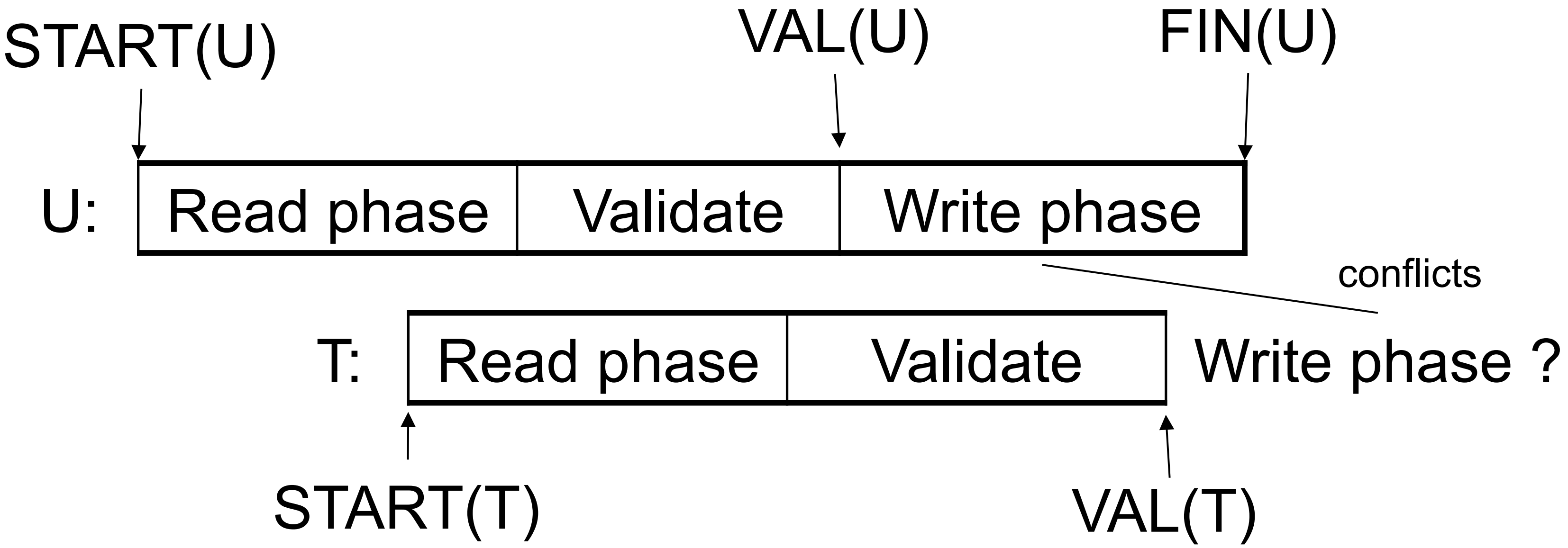


IF RS(T) ∩ WS(U) and FIN(U) > START(T)

(U has validated and U has not finished before T begun)

Then ROLLBACK(T)

Avoid $w_T(X)$ - $w_U(X)$ Conflicts



START(U)

VAL(U)

FIN(U)

U: Read phase | Validate | Write phase

conflicts

T: Read phase | Validate

Write phase ?

START(T)

VAL(T)

IF WS(T) ∩ WS(U) and FIN(U) > VAL(T)

(U has validated and U has not finished before T validates)

Then ROLLBACK(T)

Snapshot Isolation

Another optimistic concurrency control method

Very efficient, and very popular

- Oracle, Postgres, SQL Server 2005

WARNING: Not serializable, yet ORACLE uses it even for SERIALIZABLE transactions !

Snapshot Isolation Rules

Each transactions receives a timestamp TS(T)

Tnx sees the snapshot at time TS(T) of database

When T commits, updated pages written to disk

Write/write conflicts are resolved by the "**first committer wins**" rule

Snapshot Isolation (Details)

Multiversion concurrency control:

Versions of X: X_{t1}, X_{t2}, X_{t3}, . . .

When T reads X, return $X_{TS(T)}$.

When T writes X: if other transaction updated X, abort

Not faithful to “first committer” rule, because the other transaction U might have committed after T. But once we abort T, U becomes the first committer ☺

What Works and What Not

No dirty reads (Why ?)

No unconsistent reads (Why ?)

No lost updates (“first committer wins”)

Moreover: no reads are ever delayed

However: read-write conflicts not caught !

Write Skew

```
T1:
  READ(X);
  if X >= 50
      then Y = -50; WRITE(Y)
  COMMIT
```

```
T2:
  READ(Y);
  if Y >= 50
      then X = -50; WRITE(X)
  COMMIT
```

In our notation:

$R_1(X), R_2(Y), W_1(Y), W_2(X), C_1, C_2$

Starting with X=50,Y=50, we end with X=-50, Y=-50.
Non-serializable !!!

Write Skews Can Be Serious

ACIDland had two viceroys, Delta and Rho

Budget had two registers: ta**X**es, and spend**Y**ng

They had HIGH taxes and LOW spending…

```
Delta:
  READ(X);
  if X= ‘HIGH’
       then { Y= ‘HIGH’;
                WRITE(Y) }
  COMMIT
```

```
Rho:
  READ(Y);
  if Y= ‘LOW’
       then {X= ‘LOW’;
                WRITE(X) }
  COMMIT
```

… and they ran a deficit ever since.

Tradeoffs

Pessimistic Concurrency Control (Locks):

- Great when there are many conflicts
- Poor when there are few conflicts

Optimistic Concurrency Control (Timestamps):

- Poor when there are many conflicts (rollbacks)
- Great when there are few conflicts

Compromise

- READ ONLY transactions → timestamps
- READ/WRITE transactions → locks

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