CSEP 544: Lecture 5
Concurrency Control

April 28, 2009

Announcements

New deadlines:

- HW3 deadline: May 2nd, 11:45pm
- HW4 deadline: May 9th, 6:30 pm

Outline

- Chapters 16, 17

The Problem

- 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

Some Famous Anomalies

- Recall these anomalies:
  - Dirty reads (including inconsistent reads)
  - Unrepeatable reads
  - Lost updates

Many other things can go wrong too

Dirty Reads

Write-Read Conflict

T_1: WRITE(A)

T_1: ABORT

T_2: READ(A)

Inconsistent Read

Write-Read Conflict

T_1: A := 20; B := 20; T_1: WRITE(A) T_1: WRITE(B)	 T_2: READ(A); T_2: READ(B);

Unrepeatable Read

Read-Write Conflict

T_1: WRITE(A)

T_2: READ(A);

T_2: READ(A);

Lost Update

Write-Write Conflict

T1	T2
T_1: READ(A)	
	T_2: READ(A);
T_1: A := A+5	
	T_2: A := A*1.3
T_1: WRITE(A)	
	T_2: WRITE(A);

Schedules

- Given multiple transactions

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 ***serializable*** 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

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)	

Ignoring Details

- Sometimes transactions' actions can commute accidentally because of specific updates
  - Serializability is undecidable !

- Scheduler should not look at transaction details

- Assume worst case updates
  - Only care about reads r(A) and writes w(A)
  - Not the actual values involved

Notation

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

Conflict Serializability

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 $w_i(X); r_j(X)$

$r_i(X); w_j(X)$

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:

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

⇓

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

The Precedence Graph Test

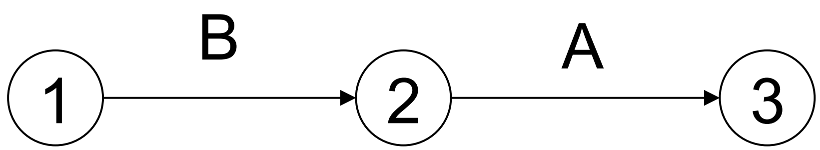
Is a schedule conflict-serializable ?

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

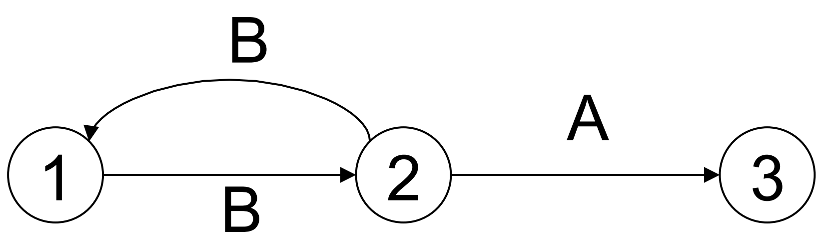
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

Example 2

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



This schedule is NOT conflict-serializable

View Equivalence

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

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

Lost write

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

Equivalent, but can't swap

View Equivalent

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

T1	T2	T3
W1(X)		
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’

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 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 equivalent to serial

Recoverability

- Recoverable
- Avoiding cascading deletes

Scheduler

- The scheduler is the module that schedules the transaction's actions, ensuring serializability
- How ? We discuss three techniques in class:
  - Locks
  - Time stamps
  - Validation

Locking 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

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

Example

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 !!!

Two Phase Locking (2PL)

The 2PL rule:

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

- This ensures conflict serializability ! (why?)

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

What about Aborts?

- 2PL enforces conflict-serializable schedules
- But does not enforce recoverable schedules

A 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

Strict 2PL

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

- Ensures that schedules are recoverable
  - Transactions commit only after all transactions whose changes they read also commit
- Avoids cascading rollbacks

Deadlock

- 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 !!

- Could be avoided, by ordering all elements (see book); or deadlock detection + rollback

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- U = update lock
  - Initially like S
  - Later may be upgraded to X
- I = increment lock (for A := A + something)
  - Increment operations commute

Read the book !

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’	

Suppose there are two blue products, X1, X2:

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

Conflict serializable ! But not serializable due to phantoms

Dealing with Phantoms

- 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
- Expensive ways of 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)

Serializable transactions are very expensive

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

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

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)

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

Note: this part follows Ullman, not R&G

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

- Conflict-serializable
- Recoverable
  - Even avoids cascading aborts
- Does NOT handle phantoms

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}, . . .

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

- 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

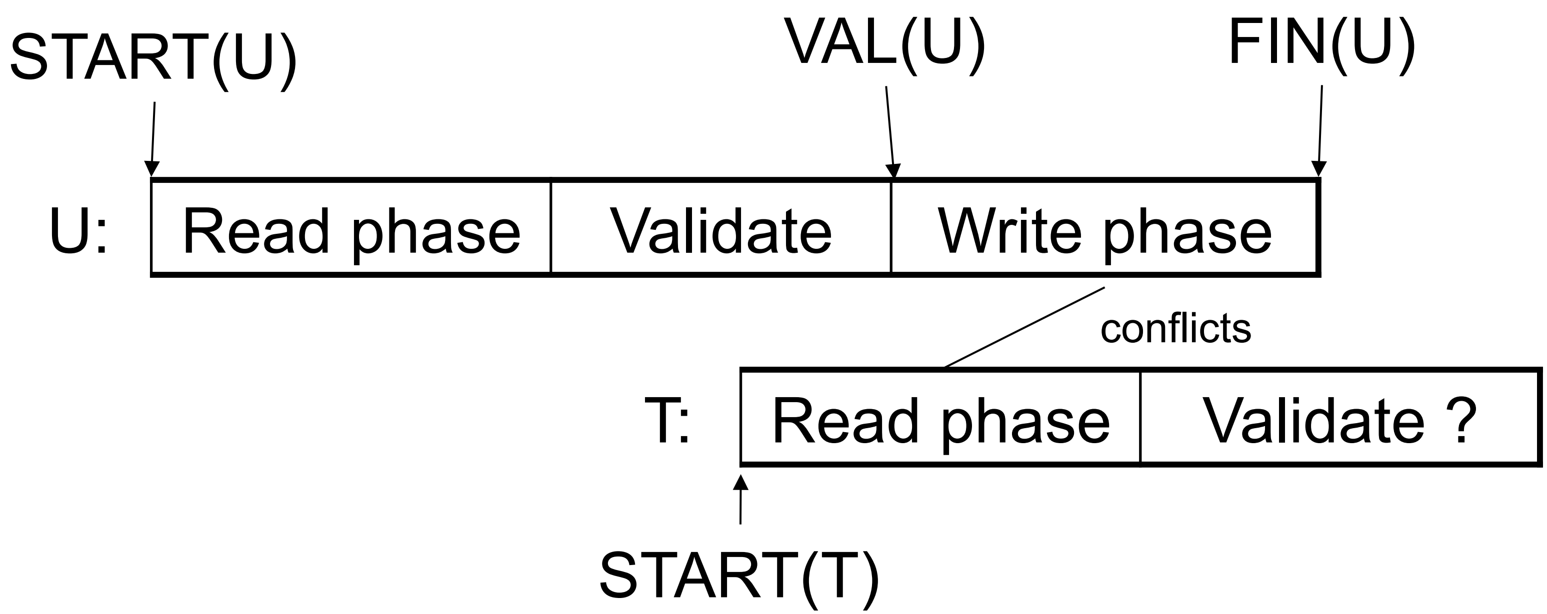
- Can delete X_t if we have a later version X_{t1} and all active transactions T have $TS(T) > t1$

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)

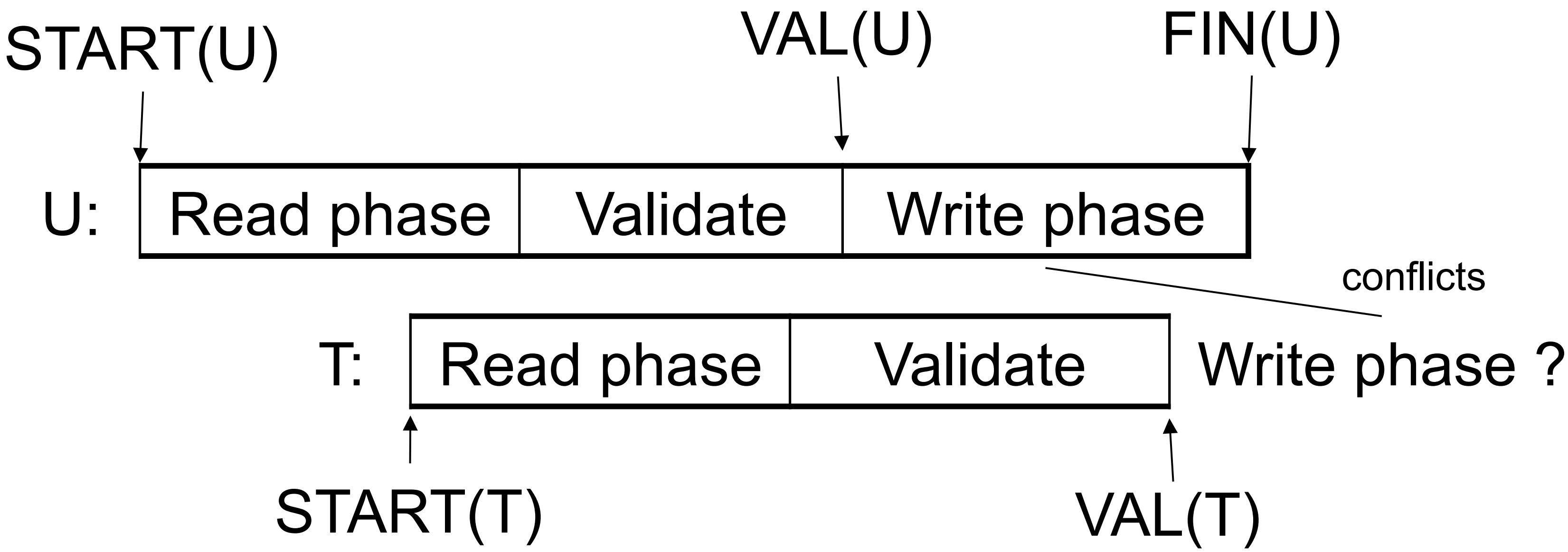
Avoid $r_T(X)$ - $w_U(X)$ Conflicts



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



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
- 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

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

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 ACID

1. Isolation Level: Dirty Reads

Implementation using locks:

- "Long duration" WRITE locks
  - Strict Two Phase Locking (you knew that !)
- No READ locks
  - Read-only transactions are never delayed

Possible pbs: dirty and inconsistent reads

2. Isolation Level: Read Committed

Implementation using locks:

- “Long duration” WRITE locks
- “Short duration” READ locks
  - Only acquire lock while reading (not 2PL)

Unrepeatable reads
When reading same element twice, may get two different values

2. Read Committed in Java

In the handout: isolation.java - Transaction 1:

```
db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
db.setAutoCommit(false);
readAccount();
Thread.sleep(5000);
readAccount();
db.commit();
```

Can see a different value

In the handout: isolation.java – Transaction 2:

```
db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
db.setAutoCommit(false);
writeAccount();
db.commit();
```

3. Isolation Level: Repeatable Read

Implementation using locks:

- "Long duration" READ and WRITE locks
  - Full Strict Two Phase Locking

Why ?

This is not serializable yet !!!

3. Repeatable Read in Java

In the handout: isolation.java - Transaction 1:

```
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
readAccount();
Thread.sleep(5000);
readAccount();
db.commit();
```

Now sees the same value

In the handout: isolation.java – Transaction 2:

```
db.setTransactionIsolation(Connection. TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
writeAccount();
db.commit();
```

3. Repeatable Read in Java

```
In the handout: isolation.java – Transaction 3:
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
countAccounts();
Thread.sleep(5000);
countAccounts();
db.commit();
```

Can see a different count

```
In the handout: isolation.java – Transaction 4:
db.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
db.setAutoCommit(false);
insertAccount();
db.commit();
```

This shows that they are not serializable !

4. Serializable in Java

```
In the handout: isolation.java – Transaction 3:
db.setTransactionIsolation(Connection. TRANSACTION_SERIALIZABLE);
db.setAutoCommit(false);
countAccounts();
Thread.sleep(5000);
countAccounts();
db.commit();
```

Now should see same count

```
In the handout: isolation.java – Transaction 4:
db.setTransactionIsolation(Connection. TRANSACTION_SERIALIZABLE);
db.setAutoCommit(false);
insertAccount();
db.commit();
```

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