Lecture 5: Transactions in SQL

Tuesday, February 6, 2007
Outline

- Transactions in SQL, the buffer manager
- Recovery
  - Chapter 17 in Ullman’s book
- Concurrency control
  - Chapter 18 in Ullman’s book
Comments on the Textbook

• Ullman’s book: chapters 17,18
  – Gentle introduction
  – We follow mostly this text in class

• Ramakrishnan: chapters 16,17,18
  – Describes quite accurately existing systems (e.g. Aries)
  – Not recommended as first reading
Transactions

• Major component of database systems
• Critical for most applications; arguably more so than SQL

• Turing awards to database researchers:
  – Charles Bachman 1973
  – Edgar Codd 1981 for inventing relational dbs
  – Jim Gray 1998 for inventing transactions
Why Do We Need Transactions

- Concurrency control
- Recovery
Concurrency control:
Three Famous anomalies

• Dirty read
  – T reads data written by T’ while T’ is running
  – Then T’ aborts

• 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’

• Inconsistent read
  – One task T sees some but not all changes made by T’
Dirty Reads

Client 1:
/* transfer $100 from account 1 to account 2 */

UPDATE Accounts
SET balance = balance + 100
WHERE accountNo = ‘11111’

X = SELECT balance
   FROM Accounts
   WHERE accountNo = ‘2222’

If X < 100 /* abort . . . . */
   then UPDATE Accounts
      SET balance = balance - 100
      WHERE accountNo = ‘11111’

Else UPDATE Accounts
      SET balance = balance - 100
      WHERE accountNo = ‘2222’

Client 2:
/* withdraw $100 from account 1 */

X = SELECT balance
   FROM Accounts
   WHERE accountNo = ‘1111’

If X > 100
   then UPDATE Accounts
      SET balance = balance - 100
      WHERE accountNo = ‘11111’
      . . . . . Dispense cash . . . . Cli

Else /* do nothing */
Lost Updates

Client 1:

```
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’
```

Client 2:

```
UPDATE Product
SET Price = Price*0.5
WHERE pname=‘Gizmo’
```

Two managers attempt to do a discount.
Will it work?
Inconsistent Read

Client 1:

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

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

Client 2:

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

What’s wrong?
Protection against crashes

Client 1:

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

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

What’s wrong ?
Definition

- **A transaction =** one or more operations, which reflects a single real-world transition
  - In the real world, this happened completely or not at all

- **Examples**
  - Transfer money between accounts
  - Purchase a group of products
  - Register for a class (either waitlist or allocated)

- If grouped in transactions, all problems in previous slides disappear
Transactions in SQL

• In “ad-hoc” SQL:
  – Default: each statement = one transaction

• In a program:
  START TRANSACTION
  [SQL statements]
  COMMIT or ROLLBACK (=ABORT)

May be omitted: first SQL query starts txn
Revised Code

Client 1: START TRANSACTION
UPDATE Product
SET Price = Price – 1.99
WHERE pname = ‘Gizmo’
COMMIT

Client 2: START TRANSACTION
UPDATE Product
SET Price = Price*0.5
WHERE pname=‘Gizmo’
COMMIT

Now it works like a charm
Transaction Properties

ACID

• **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 atxn has committed, its effects remain in the database
ROLLBACK

• If the app gets to a place where it can’t complete the transaction successfully, it can execute ROLLBACK

• This causes the system to “abort” the transaction
  – The database returns to the state without any of the previous changes made by activity of the transaction
Reasons for Rollback

• User changes their mind ("ctl-C"/cancel)
• Explicit in program, when app program finds a problem
  – e.g. when qty on hand < qty being sold
• System-initiated abort
  – System crash
  – Housekeeping
    • e.g. due to timeouts
Theory of Transaction Management

Two parts:

• Recovery from crashes:  ACID
• Concurrency control:     ACID

Both operate on the buffer pool
## Recovery

<table>
<thead>
<tr>
<th>Type of Crash</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong data entry</td>
<td>Constraints and Data cleaning</td>
</tr>
<tr>
<td>Disk crashes</td>
<td>Redundancy: e.g. RAID, archive</td>
</tr>
<tr>
<td>Fire, theft, bankruptcy…</td>
<td>Buy insurance, Change jobs…</td>
</tr>
<tr>
<td>System failures: e.g. power</td>
<td>DATABASE RECOVERY</td>
</tr>
</tbody>
</table>

*Most frequent*
The Mechanics of Disk

Mechanical characteristics:
• Rotation speed (5400RPM)
• Number of platters (1-30)
• Number of tracks (\(\leq 10000\))
• Number of bytes/track\((10^5)\)

Unit of read or write: **disk block**
Once in memory: **page**
Typically: 4k or 8k or 16k
Disk Access Characteristics

• Disk latency = time between when command is issued and when data is in memory

• Disk latency = seek time + rotational latency
  – Seek time = time for the head to reach cylinder
    • 10ms – 40ms
  – Rotational latency = time for the sector to rotate
    • Rotation time = 10ms
    • Average latency = 10ms/2

• Transfer time = typically 40MB/s
• Disks read/write one block at a time
RAID: Protect against HW Failure

Several disks that work in parallel
• Redundancy: use parity to recover from disk failure
• Speed: read from several disks at once

Various configurations (called levels):
• RAID 1 = mirror
• RAID 4 = n disks + 1 parity disk
• RAID 5 = n+1 disks, assign parity blocks round robin
• RAID 6 = “Hamming codes”
Buffer Management in a DBMS

Page Requests from Higher Levels

- Data must be in RAM for DBMS to operate on it!
- Table of <frame#, pageid> pairs is maintained

choice of frame dictated by replacement policy
Buffer Manager

Needs to decide on page replacement policy

- LRU
- Clock algorithm

Both work well in OS, but not always in DB

Enables the higher levels of the DBMS to assume that the needed data is in main memory.
Least Recently Used (LRU)

- Order pages by the time of last accessed
- Always replace the least recently accessed

P5, P2, P8, P4, P1, P9, P6, P3, P7

Access P6

P6, P5, P2, P8, P4, P1, P9, P3, P7

LRU is expensive (why ?); the clock algorithm is good approx
Buffer Manager

Why not use the Operating System for the task??

Two reasons:

• May improve performance by knowing the access pattern

• Need fined-grained access to the operations to ensure ACID semantics
Transaction Manager

Operates on the buffer pool

• Recovery:
  – ‘log-file write-ahead’,
  – policy about which pages to force to disk

• Concurrency:
  – locks at the page level,
  – Or multiversion concurrency control
Transactions

• Assumption: the database is composed of elements
  – Usually 1 element = 1 block
  – Can be smaller (=1 record) or larger (=1 relation)

• Assumption: each transaction reads/writes some elements
Primitive Operations of Transactions

- **READ(X,t)**
  - copy element X to transaction local variable t

- **WRITE(X,t)**
  - copy transaction local variable t to element X

- **INPUT(X)**
  - read element X to memory buffer

- **OUTPUT(X)**
  - write element X to disk
Example

START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;

Atomicity:
BOTH A and B are multiplied by 2
<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Action</td>
<td>t</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Crash occurs after OUTPUT(A), before OUTPUT(B)
We lose atomicity
The Log

• An append-only file containing log records
• Note: multiple transactions run concurrently, log records are interleaved
• After a system crash, use log to:
  – Redo some transaction that didn’t commit
  – Undo other transactions that didn’t commit
• Three kinds of logs: undo, redo, undo/redo
Undo Logging

Log records
- \(<\text{START } T>\) transaction $T$ has begun
- \(<\text{COMMIT } T>\) $T$ has committed
- \(<\text{ABORT } T>\) $T$ has aborted
- \(<T,X,v>\) $T$ has updated element $X$, and its old value was $v$
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>Action</td>
<td>T</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
<td>Log</td>
</tr>
<tr>
<td>--------------</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

WHAT DO WE DO?
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

WHAT DO WE DO?
After Crash

• In the first example:
  – We UNDO both changes: A=8, B=8
  – The transaction is atomic, since none of its actions has been executed

• In the second example
  – We don’t undo anything
  – The transaction is atomic, since both it’s actions have been executed
Undo-Logging Rules

U1: If T modifies X, then \(<T,X,v>\) must be written to disk before \(\text{OUTPUT}(X)\)

U2: If T commits, then \(\text{OUTPUT}(X)\) must be written to disk before \(<\text{COMMIT T}>\)

• Hence: OUTPUTs are done \textit{early}, before the transaction commits
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Recovery with Undo Log

After system’s crash, run recovery manager

• Idea 1. Decide for each transaction T whether it is completed or not
  – <START T>…..<COMMIT T>…. = yes
  – <START T>…..<ABORT T>…….. = yes
  – <START T>…………………… = no

• Idea 2. Undo all modifications by incomplete transactions
Recovery with Undo Log

Recovery manager:

- Read log from the end; cases:
  - <COMMIT T>: mark T as completed
  - <ABORT T>: mark T as completed
  - <T,X,v>: if T is not completed
    - then write X=v to disk
    - else ignore
  - <START T>: ignore
Recovery with Undo Log

...<T6,X6,v6>
...<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMITE T5>
<T3,X3,v3>
<T2,X2,v2>

Question 1 in class: Which updates are undone?

Question 2 in class: How far back do we need to read in the log?
Recovery with Undo Log

• Note: all undo commands are *idempotent*
  – If we perform them a second time, no harm is done
  – E.g. if there is a system crash during recovery, simply restart recovery from scratch
Recovery with Undo Log

When do we stop reading the log?

- We cannot stop until we reach the beginning of the log file
- This is impractical

Instead: use checkpointing
Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions
Undo Recovery with Checkpointing

During recovery, Can stop at first <CKPT>

... 
... 
<T9,X9,v9>
... 
... (all completed)
<CKPT>
<START T2>
<START T3
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

other transactions

transactions T2,T3,T4,T5
Nonquiescent Checkpointing

• Problem with checkpointing: database freezes during checkpoint
• Would like to checkpoint while database is operational
• Idea: nonquiescent checkpointing

Quiescent = being quiet, still, or at rest; inactive
Non-quiescent = allowing transactions to be active
Nonquiescent Checkpointing

• Write a $<\text{START CKPT}(T_1,\ldots,T_k)>$ where $T_1,\ldots,T_k$ are all active transactions
• Continue normal operation
• When all of $T_1,\ldots,T_k$ have completed, write $<\text{END CKPT}>$
Undo Recovery with Nonquiescent Checkpointing

During recovery, Can stop at first <CKPT>

Q: do we really need <END CKPT> ?

\[
\begin{align*}
\ldots & \ldots \\
\ldots & \ldots \\
\ldots & \ldots \\
\ldots & \ldots \\
<\text{START CKPT T4, T5, T6}> \\
\ldots & \ldots \\
\ldots & \ldots \\
<\text{END CKPT}> \\
\ldots & \ldots \\
\ldots & \ldots \\
\ldots & \ldots \\
\end{align*}
\]

\{ earlier transactions plus T4, T5, T6 \}

\{ T4, T5, T6, plus later transactions \}

\{ later transactions \}
Redo Logging

Log records

- \(<\text{START } T>\) = transaction T has begun
- \(<\text{COMMIT } T>\) = T has committed
- \(<\text{ABORT } T>\) = T has aborted
- \(<T,X,v>\) = T has updated element X, and its new value is v
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,16&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Redo-Logging Rules

R1: If T modifies X, then both \(<T,X,v>\) and \(<\text{COMMIT T}>\) must be written to disk before OUTPUT(X)

• Hence: OUTPUTs are done \textit{late}
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,16&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Recovery with Redo Log

After system’s crash, run recovery manager

• Step 1. Decide for each transaction T whether it is completed or not
  – <START T>….<COMMIT T>…. = yes
  – <START T>….<ABORT T>…… = yes
  – <START T>.......................... = no

• Step 2. Read log from the beginning, redo all updates of committed transactions
Recovery with Redo Log

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
...
...
Nonquiescent Checkpointing

- Write a `<START CKPT(T₁,…,Tₖ)>` where T₁,…,Tₖ are all active transactions
- Flush to disk all blocks of committed transactions (*dirty blocks*), while continuing normal operation
- When all blocks have been written, write `<END CKPT>`
Redo Recovery with Nonquiescent Checkpointing

Step 1: look for
The last
<END CKPT>

All OUTPUTs of T1 are known to be on disk

Step 2: redo from the earliest start of T4, T5, T6
ignoring transactions committed earlier

Cannot use
Comparison Undo/Redo

• Undo logging:
  – OUTPUT must be done early
  – If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don’t need to redo) – inefficient

• Redo logging
  – OUTPUT must be done late
  – If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk, no need to undo)
    – inflexible

• Would like more flexibility on when to OUTPUT: undo/redo logging (next)
Undo/Redo Logging

Log records, only one change

• \(<T,X,u,v>= T\) has updated element \(X\), its \textit{old} value was \(u\), and its \textit{new} value is \(v\)
Undo/Redo-Logging Rule

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

Note: we are free to OUTPUT early or late relative to <COMMIT T>
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>START T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8,16&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Can OUTPUT whenever we want: before/after COMMIT$^61$
Recovery with Undo/Redo Log

After system’s crash, run recovery manager
• Redo all committed transaction, top-down
• Undo all uncommitted transactions, bottom-up
Recovery with Undo/Redo Log

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

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
...
...
Concurrency Control

• Multiple transactions are running concurrently $T_1, T_2, \ldots$
• They read/write some common elements $A_1, A_2, \ldots$
• How can we prevent unwanted interference?

The SCHEDULER is responsible for that
Three Famous Anomalies

What can go wrong if we didn’t have concurrency control:

• Dirty reads
• Lost updates
• Inconsistent reads

Many other things may go wrong, but have no names
Dirty Reads

$T_1$: WRITE($A$)

$T_2$: READ($A$)

$T_1$: ABORT
Lost Update

\[
\begin{align*}
    &T_1: \text{READ}(A) \\
    &T_1: A := A + 5 \\
    &T_1: \text{WRITE}(A) \\
    &T_2: \text{READ}(A); \\
    &T_2: A := A \times 1.3 \\
    &T_2: \text{WRITE}(A);
\end{align*}
\]
Inconsistent Read

$T_1$: $A := 20; \ B := 20;$

$T_1$: WRITE(A)

$T_1$: WRITE(B)

$T_2$: READ(A);

$T_2$: READ(B);
Schedules

- Given multiple transactions
- A *schedule* is a sequence of interleaved actions from all transactions
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>
A Serial Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>WRITE(B,s)</td>
</tr>
</tbody>
</table>
Serializable Schedule

- A schedule is \textit{serializable} if it is equivalent to a serial schedule
A Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>

Notice: this is NOT a serial schedule
A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A, t)</td>
<td>READ(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t)</td>
<td>WRITE(A,s)</td>
</tr>
<tr>
<td></td>
<td>READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s)</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td></td>
</tr>
<tr>
<td>t := t+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
</tr>
</tbody>
</table>
Ignoring Details

• Sometimes transactions’ actions may commute accidentally because of specific updates
  – Serializability is undecidable!

• The scheduler shouldn’t look at the transactions’ details

• Assume worst case updates, only care about reads $r(A)$ and writes $w(A)$
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:

\[
\begin{align*}
&\text{r}_1(A); \ w_1(A); \ r_2(A); \ w_2(A); \ r_1(B); \ w_1(B); \ r_2(B); \ w_2(B) \\
\rightarrow
\end{align*}
\]

\[
\begin{align*}
&\text{r}_1(A); \ w_1(A); \ r_1(B); \ w_1(B); \ r_2(A); \ w_2(A); \ r_2(B); \ w_2(B)
\end{align*}
\]
Conflict Serializability

• Any conflict serializable schedule is also a serializable schedule (why ?)

• The converse is not true, even under the “worst case update” assumption

Equivalent, but can’t swap
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
Scheduler

• The scheduler is the module that schedules the transaction’s actions, ensuring serializability

• How? Three techniques:
  – 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

\[ l_i(A) = \text{transaction } T_i \text{ acquires lock for element } A \]

\[ u_i(A) = \text{transaction } T_i \text{ releases lock for element } A \]
Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td><strong>T2</strong></td>
</tr>
<tr>
<td>L₁(A); READ(A, t)</td>
<td>L₂(A); READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A); L₁(B)</td>
<td>WRITE(A, s); U₂(A);</td>
</tr>
<tr>
<td>READ(B, t)</td>
<td>L₂(B); DENIED...</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t); U₁(B);</td>
<td>WRITE(B, s); U₂(B);</td>
</tr>
</tbody>
</table>

...GRANTED; READ(B, s)

The scheduler has ensured a conflict-serializable schedule
## Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); READ(A, t)</td>
<td>L₂(A); READ(A, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(A, t); U₁(A);</td>
<td>WRITE(A, s); U₂(A);</td>
</tr>
<tr>
<td>L₁(B); READ(B, t)</td>
<td>L₂(B); READ(B, s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>WRITE(B, t); U₁(B);</td>
<td>WRITE(B, s); U₂(B);</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!!
Two Phase Locking (2PL)

The 2PL rule:

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

- This ensures conflict serializability! (why?)
Example: 2PL transactions

T1
L1(A); L1(B); READ(A, t)
t := t+100
WRITE(A, t); U1(A)

READ(B, t)
t := t+100
WRITE(B, t); U1(B); 

T2
L2(A); READ(A, s)
s := s*2
WRITE(A, s);
L2(B); DENIED...

...GRANTED; READ(B, s)
s := s*2
WRITE(B, s); U2(A); U2(B);

Now it is conflict-serializable
Deadlock

• 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 $T_{73}$ waits for a lock held by $T_1$ !!

Could be avoided, by ordering all elements (see book); or deadlock detection plus 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 $A := A + \text{something}$)
  - Increment operations commute
- **READ CHAPTER 17 in Ramakrishnan or 18.4 in Ullman**
The Locking Scheduler

Taks 1:
   add lock/unlock requests to transactions
   • Examine all READ(A) or WRITE(A) actions
   • Add appropriate lock requests
   • Ensure 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
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B-trees (the indexes of choice in databases)
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)

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

- Few transactions
  - No lock contention
  - High throughput

- More transactions
  - Some lock contention
  - Higher throughput (because more transactions)

- Even more transactions
  - A lot of lock contention
  - Lower throughput (thrashing)

See Ramakrishnan, page 534
Other Concurrency Control Methods

• Timestamps
  – Variation: snapshot isolation (Oracle)

• Validation
Timestamps

Every transaction receives a unique timestamp $\text{TS}(T)$

Could be:

- The system’s clock
- A unique counter, incremented by the scheduler
Timestaps

Main invariant:

The timestamp order defines the serialization order of the transaction
Timestamps

Associate to each element $X$:
- $RT(X) =$ the highest timestamp of any transaction that read $X$
- $WT(X) =$ the highest timestamp of any transaction that wrote $X$
- $C(X) =$ the commit bit: says if the transaction with highest timestamp that wrote $X$ committed

These are associated to each page $X$ in the buffer pool
Main Idea

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

In each of these cases

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

Check that $TS(U) < TS(T)$

When $T$ wants to read $X$, $r_T(X)$, how do we know $U$, and $TS(U)$?
Details

Read too late:

- T wants to read X, and $\text{TS}(T) < \text{WT}(X)$

\[
\text{START}(T) \ldots \text{START}(U) \ldots w_U(X) \ldots r_T(X)
\]

Need to rollback T!
Details

Write too late:

- T wants to write X, and
  \[ WT(X) < TS(T) < RT(X) \]

\[ \text{START}(T) \ldots \text{START}(U) \ldots r_U(X) \ldots w_T(X) \]

Need to rollback T!

Why do we check \( WT(X) < TS(T) \) ????
Details

Write too late, but we can still handle it:

- T wants to write X, and
  \[ TS(T) < RT(X) \text{ but } WT(X) > TS(T) \]

\[
\text{START}(T) \ldots \text{START}(V) \ldots w_V(X) \ldots w_T(X)
\]

Don’t write X at all!
(but see later…)
More Problems

Read dirty data:

• T wants to read X, and WT(X) < TS(T)
• Seems OK, but…

If C(X)=1, then T needs to wait for it to become 0
More Problems

Write dirty data:
- 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)=1, then T needs to wait for it to become 0
Timestamp-based Scheduling

When a transaction $T$ requests $r(X)$ or $w(X)$, the scheduler examines $RT(X)$, $WT(X)$, $C(X)$, and decides one of:

- To grant the request, or
- To rollback $T$ (and restart with later timestamp)
- To delay $T$ until $C(X) = 0$
Timestamp-based Scheduling

RULES:

• There are 4 long rules in the textbook, on page 974
• You should be able to understand them, or even derive them yourself, based on the previous slides
• Make sure you understand them!

READING ASSIGNMENT: 18.8.4
Multiversion Timestamp

• When transaction T requests r(X) but WT(X) > TS(T), then T must rollback.
• Idea: keep multiple versions of X: \( X_t, X_{t-1}, X_{t-2}, \ldots \)
  \[
  \text{TS}(X_t) > \text{TS}(X_{t-1}) > \text{TS}(X_{t-2}) > \ldots
  \]
• 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 a version $X_t$ such that $t < TS(T)$ and $t$ is the largest such
- $WT(X_t) = t$ and it never changes
- $RD(X_t)$ must also be maintained, to reject certain writes (why ?)
- When can we delete $X_t$: if we have a later version $X_{t1}$ and all active transactions $T$ have $TS(T) > t1$
Tradeoffs

- **Locks:**
  - Great when there are many conflicts
  - Poor when there are few conflicts

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

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

START(U)  VAL(U)  FIN(U)

U: Read phase  Validate  Write phase

T: Read phase  Validate ?

START(T)

IF RS(T) $\cap$ 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

U: Read phase | Validate | Write phase

T: Read phase | Validate | Write phase ?

START(U) | VAL(U) | FIN(U)

START(T) | VAL(T)

IF $WS(T) \cap WS(U)$ and $FIN(U) > VAL(T)$
(U has validated and U has not finished before T validates)

Then ROLLBACK(T)