CSE 544
Principles of Database Management Systems

Alvin Cheung
Fall 2015

Lecture 11 – Transactions: Concurrency Control
Announcements

• HW2 due tonight

• HW3 posted
  – Due in two weeks
  – Check website for OH

• Next couple of lectures we will talk about transactions
Where We Are

• Data models
  – Relational
  – IMS / Codasyl
  – Unstructured

• Query processing
  – Algorithms for relational operators
  – Indexing and physical design

• Dealing with the real world
  – Data warehousing
  – Transaction processing
References

• **Concurrency control and recovery.**

• **Database management systems.**
  Ramakrishnan and Gehrke.
  Third Ed. **Chapters 16 and 17.**
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Motivating Example

UPDATE Budget
SET money=money-100
WHERE pid = 1

UPDATE Budget
SET money=money+60
WHERE pid = 2

UPDATE Budget
SET money=money+40
WHERE pid = 3

SELECT sum(money)
FROM Budget

Would like to treat each group of instructions as a unit
Definition

• **A transaction** = one or more operations, single real-world transition

• **Examples**
  – Transfer money between accounts
  – Purchase a group of products
  – Register for a class (either waitlist or allocated)
  – What else?
Transactions

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

• Fact: Turing awards to database researchers:
  – Charles Bachman 1973 for CODASYL
  – Edgar Codd 1981 for inventing relational dbms
  – Jim Gray 1998 for inventing transactions
  – Michael Stonebraker 2015 for postgres
START TRANSACTION

UPDATE Budget SET money = money - 100
WHERE pid = 1

UPDATE Budget SET money = money + 60
WHERE pid = 2

UPDATE Budget SET money = money + 40
WHERE pid = 3

COMMIT
ROLLBACK

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

• This causes the system to “abort” the transaction
  – Database returns to a state without any of the changes made by 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, admission control, etc
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs
- **Consistency**: A transaction as a whole does not violate integrity constraints
- **Isolation**: Transactions appear to execute one after the other in sequence
- **Durability**: If a transaction commits, its changes will survive failures

• Q: Benefits & drawbacks of providing ACID transactions?
What Could Go Wrong?

- Why is it hard to provide ACID properties?
  - Concurrent operations
    - Isolation problems
    - We saw one example earlier
  - Failures can occur at any time
    - Atomicity and durability problems
    - Next lecture
- Transaction may need to abort
In a World Without Transactions

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

  DELETE Product
  WHERE price <=0.99

Client 2: SELECT count(*)
  FROM Product

  SELECT count(*)
  FROM SmallProduct

What could go wrong? Inconsistent reads
Different Types of Problems

Client 1:
```
UPDATE Product
SET Price = Price - 1.99
WHERE pname = 'Gizmo'
```

Client 2:
```
UPDATE Product
SET Price = Price*0.5
WHERE pname='Gizmo'
```

What could go wrong? Lost update
Different Types of Problems

Client 1: UPDATE SET Account.amount = 1000000
      WHERE Account.number = 1001

Client 2: SELECT Account.amount
          FROM Account
          WHERE Account.number = 1001

What could go wrong? Dirty reads

Aborted by system
Types of Problems: Summary

• Concurrent execution problems
  – Write-read conflict: dirty read (includes inconsistent read)
    • A transaction reads a value written by another transaction that has not yet committed
  – Read-write conflict: unrepeateable read
    • A transaction reads the value of the same object twice. Another transaction modifies that value in between the two reads
  – Write-write conflict: lost update
    • Two transactions update the value of the same object. The second one to write the value overwrite the first change

• Failure problems
  – DBMS can crash in the middle of a series of updates
  – Can leave the database in an inconsistent state
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Schedules

- Given multiple transactions

- A schedule is a sequence of interleaved actions from all transactions
Example 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>WRITE(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>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>READ(A, t)</td>
<td>WRITE(A, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
<td>WRITE(B, t)</td>
</tr>
<tr>
<td></td>
<td>READ(B, t)</td>
<td>READ(A, s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
<td>WRITE(A, s)</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, s)</td>
<td>WRITE(B, s)</td>
</tr>
</tbody>
</table>
Serializable Schedule

- A schedule is *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></td>
<td></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></td>
<td>READ(B, t)</td>
</tr>
<tr>
<td></td>
<td>t := t+100</td>
</tr>
<tr>
<td></td>
<td>WRITE(B, t)</td>
</tr>
</tbody>
</table>
Notation

\[ T_1: r_1(A); w_1(A); r_1(B); w_1(B) \]
\[ T_2: r_2(A); w_2(A); r_2(B); w_2(B) \]
Serializable Execution

- **Serializability**: interleaved execution has *same effect as some serial execution*

- **Schedule** of two transactions (Figure 1)
  \[
  r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_1[B] \rightarrow c_1 \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0
  \]

- **Serializable schedule**: equiv. to serial schedule
  \[
  r_0[A] \rightarrow w_0[A] \rightarrow r_1[A] \rightarrow r_0[B] \rightarrow w_0[B] \rightarrow c_0 \rightarrow r_1[B] \rightarrow c_1
  \]
Ignoring Details

• Sometimes transactions’ actions can commute accidentally because of specific updates
  – Fact: 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
Conflict Serializability

Conflicts: (aka bad things happen if swapped)

Two actions by same transaction $T_i$:

Two writes by $T_i$, $T_j$ to same element

Read/write by $T_i$, $T_j$ to same element
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

• Fact: if the graph has no cycles, then it is conflict serializable!
Example 1

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
Conflict Serializability

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

\[
\begin{align*}
&\text{w}_1(Y); \text{w}_2(Y); \text{w}_2(X); \text{w}_1(X); \text{w}_3(X); \\
&\text{w}_1(Y); \text{w}_1(X); \text{w}_2(Y); \text{w}_2(X); \text{w}_3(X);
\end{align*}
\]

Equivalent, but can’t swap

Lost write
Scheduler

• The scheduler is the module that schedules the transaction’s actions, ensuring serializability
• How? We discuss three techniques in class:
  – Locks
  – Timestamps
  – Validation
Outline

• Transactions motivation, definition, properties

• Concurrency control and locking

• Optimistic concurrency control
Locking Scheduler

Simple idea:
- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If 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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{L}_1(A); \text{READ}(A, t)</td>
<td>\text{L}_2(A); \text{READ}(A,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>\text{WRITE}(A, t); U_1(A); \text{L}_1(B)</td>
<td>\text{WRITE}(A,s); U_2(A); \text{L}_2(B); \text{DENIED}...</td>
</tr>
<tr>
<td>\text{READ}(B, t)</td>
<td>...\text{GRANTED}; \text{READ}(B,s)</td>
</tr>
<tr>
<td>t := t+100</td>
<td>s := s*2</td>
</tr>
<tr>
<td>\text{WRITE}(B,t); U_1(B);</td>
<td>\text{WRITE}(B,s); U_2(B);</td>
</tr>
</tbody>
</table>

Scheduler has ensured a conflict-serializable schedule
Is this enough?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); \text{READ}(A, t) )</td>
<td>( L_2(A); \text{READ}(A,s) )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s^2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A, t); \ U_1(A); )</td>
<td>( \text{WRITE}(A,s); \ U_2(A); )</td>
</tr>
<tr>
<td>( L_1(B); \text{READ}(B, t) )</td>
<td>( L_2(B); \text{READ}(B,s) )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s^2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B,t); \ U_1(B); )</td>
<td>( \text{WRITE}(B,s); \ U_2(B); )</td>
</tr>
</tbody>
</table>

Locks did not enforce conflict-serializability !!!

---

39
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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); L_1(B); )</td>
<td>( L_2(A); )</td>
</tr>
<tr>
<td>( \text{READ}(A, t) )</td>
<td>( \text{READ}(A, s) )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(A, t); U_1(A) )</td>
<td>( \text{WRITE}(A, s); )</td>
</tr>
<tr>
<td>( \text{READ}(B, t) )</td>
<td>( \text{L}_2(B); \text{DENIED} \ldots )</td>
</tr>
<tr>
<td>( t := t+100 )</td>
<td>( s := s*2 )</td>
</tr>
<tr>
<td>( \text{WRITE}(B, t); U_1(B) )</td>
<td>( \text{WRITE}(B, s); U_2(A); \text{U}_2(B) )</td>
</tr>
</tbody>
</table>

Now it is conflict-serializable
Example with Multiple Transactions

Equivalent to each transaction executing entirely the moment it enters shrinking phase
What about Aborts?

- 2PL enforces conflict-serializable schedules
- But what if a transaction releases its locks and then aborts?
**Example with Abort**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L₁(A); L₁(B); 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);</td>
</tr>
<tr>
<td></td>
<td>L₂(B); DENIED…</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); U₁(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>…GRANTED; READ(B,s)</td>
</tr>
<tr>
<td></td>
<td>s := s*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B,s); U₂(A); U₂(B);</td>
</tr>
<tr>
<td>Abort</td>
<td>Commit</td>
</tr>
</tbody>
</table>
Strict 2PL

• Strict 2PL: All locks held by a transaction are released when the transaction is completed
  – Also called “long-duration locks”

• Ensures that schedules are recoverable
  – Transactions commit only after all transactions whose changes they read also commit

• Avoids cascading rollbacks