Review

• DBMSs support ACID Transaction semantics.
• Concurrency control and Crash Recovery are key components

Review

• For Isolation property, serial execution of transactions is safe but slow
  – Try to find schedules equivalent to serial execution
• One solution for “conflict serializable” schedules is Two Phase Locking (2PL)
Outline

- Serializability - concepts and algorithms
- One solution: Locking
  - 2PL
  - variations
- Deadlocks

Conflicting Operations

- We need a formal notion of equivalence that can be implemented efficiently…
  - Base it on the notion of “conflicting” operations

- Definition: Two operations conflict if:
  - They are by different transactions,
  - they are on the same object,
  - and at least one of them is a write.

Conflict Serializable Schedules

- Definition: Two schedules are conflict equivalent iff:
  - They involve the same actions of the same transactions, and
  - every pair of conflicting actions is ordered the same way

- Definition: Schedule S is conflict serializable if:
  - S is conflict equivalent to some serial schedule.

- Note, some “serializable” schedules are NOT conflict serializable (see example #4’, later)
Conflict Serializability – Intuition

• A schedule $S$ is conflict serializable if:
  – You are able to transform $S$ into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

• Example:

\[
\begin{array}{c}
R(A) \ W(A) \\
R(B) \ W(B)
\end{array}
\]

\[
\begin{array}{c}
R(A) \ W(A) \\
R(B) \ W(B)
\end{array}
\]

\[
\begin{array}{c}
R(A) \ W(A) \\
R(B) \ W(B)
\end{array}
\]

Conflict Serializability (Continued)

• Here’s another example:

\[
\begin{array}{c}
R(A) \ W(A)
\end{array}
\]

• Serializable or not????

Conflict Serializability (Continued)

• Here’s another example:

\[
\begin{array}{c}
R(A) \ W(A)
\end{array}
\]

• Serializable or not???? NOT!
Serializability

- Q: any faster algorithm? (faster than transposing ops?)

Dependency Graph

- One node per Xact
- Edge from $T_i$ to $T_j$ if:
  - An operation $O_i$ of $T_i$ conflicts with an operation $O_j$ of $T_j$ and
  - $O_i$ appears earlier in the schedule than $O_j$.

Dependency Graph

- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic.

('dependency graph': a.k.a. 'precedence graph')
Example #1

- A schedule that is not conflict serializable:

  T1: R(A), W(A), R(B), W(B)
  T2: R(A), W(A), R(B), W(B)

  A
  \[ \text{dependency graph} \]
  B

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Example #2 (Lost update)

T1

- Read(N)
- N = N - 1
- Write(N)

T2

- Read(N)
- N = N - 1
- Write(N)
Example #2 (Lost update)

T1
- Read(N)
- N = N - 1
- Write(N)

T2
- Read(N)
- N = N - 1
- Write(N)

R/W

Example #3

T1
- Read(A)
- write(A)

T2
- Read(A)
- write(A)

T3
- Read(B)
- write(B)
- Read(B)
- write(B)
Example #3

A: T2, T1, T3
(Notice that T3 should go after T2, although it starts before it!)

Q: algo for generating serial execution from (acyclic) dependency graph?

A: Topological sorting
Example #4 (Inconsistent Analysis)

T1    T2
R (A)  
A = A-10
W (A)  dependency graph?

R(A)  
Sum = A
R (B)  
Sum += B

R(B)  
B = B+10
W(B)

Example #4 (Inconsistent Analysis)

.create a 'correct' schedule that is not conflict-serializable

T1    T2
R (A)  
A = A-10
W (A)  

R(A)  
Sum = A
R (B)  
Sum += B

R(B)  
B = B+10
W(B)

Example #4’ (Inconsistent Analysis)

T1    T2
R (A)  
A = A-10
W (A)  A: T2 asks for the count of my active accounts

R(A)  
if (A>0), count=1
R (B)  
if (B>0), count++

R(B)  
B = B+10
W(B)
**An Aside: View Serializability**

- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are **view equivalent** if:
  1. If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  2. If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  3. If Ti writes final value of A in S1, then Ti also writes final value of A in S2

<table>
<thead>
<tr>
<th>T1: R(A)</th>
<th>W(A)</th>
<th>T2: W(A)</th>
<th>T3: W(A)</th>
</tr>
</thead>
</table>

**View Serializability**

- Basically, allows all conflict serializable schedules + “blind writes”

<table>
<thead>
<tr>
<th>T1: R(A)</th>
<th>W(A)</th>
<th>T2: W(A)</th>
<th>T3: W(A)</th>
</tr>
</thead>
</table>

- T1: R(A), W(A) | T2: W(A) | T3: W(A) |

<table>
<thead>
<tr>
<th>A: 5</th>
<th>10</th>
<th>8</th>
<th>25</th>
</tr>
</thead>
</table>

| A: 5 | 8 | 10 | 25 |
Notes on Serializability
Definitions

• View Serializability allows (slightly) more schedules than Conflict Serializability does.
  – Problem is that it is difficult to enforce efficiently.
• Neither definition allows all schedules that you would consider “serializable”.
  – This is because they don’t understand the meanings of the operations or the data (recall example #4’)

Notes on Serializability
Definitions

• In practice, Conflict Serializability is what gets used, because it can be enforced efficiently.
  – To allow more concurrency, some special cases do get handled separately, such as for travel reservations, etc.

Outline

• Serializability - concepts and algorithms
  ➤ One solution: Locking
    – 2PL
    – variations
• Deadlocks
Two-Phase Locking (2PL)

- **Locking Protocol**
  - ‘S’ (shared) and ‘X’ (eXclusive) locks
  - A transaction can not request additional locks once it releases any locks.
  - Thus, there is a “growing phase” followed by a “shrinking phase”.

THEOREM: if all transactions obey 2PL -> all schedules are serializable

2PL

THEOREM: if all transactions obey 2PL -> all schedules are serializable

(if even one violates 2PL, non-serializability is possible - example?)
Two-Phase Locking (2PL), cont.

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to Cascading Aborts.

Strict 2PL

- Allows only conflict serializable schedules, but it is actually stronger than needed for that purpose.

2PL

- Problem: Cascading Aborts
- Example: rollback of T1 requires rollback of T2!

\[
\begin{align*}
\text{T1: } & \ R(A), W(A), & R(B), W(B), \text{ Abort} \\
\text{T2: } & \ R(A), W(A) \\
\end{align*}
\]

- Solution: Strict 2PL, i.e,
- keep all locks, until ‘commit’
Strict 2PL (continued)

- In effect, “shrinking phase” is delayed until
  - Transaction commits (commit log record on disk), or
  - Aborts (then locks can be released after rollback).

Next ...

- A few examples

Non-2PL, A = 1000, B = 2000, Output =?

<table>
<thead>
<tr>
<th>Action</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A := A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
<tr>
<td>PRINT(A+B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>
### 2PL, A = 1000, B = 2000, Output =?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th>Read(A)</th>
<th>Lock_S(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A := A - 50</td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>Unlock(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := B + 50</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Unlock(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Unlock(A)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRINT(A+B)</td>
</tr>
</tbody>
</table>

### Strict 2PL, A = 1000, B = 2000, Output =?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th>Read(A)</th>
<th>Lock_S(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A := A - 50</td>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>Unlock(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>B := B + 50</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>Unlock(B)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Unlock(A)</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRINT(A+B)</td>
</tr>
</tbody>
</table>

### Venn Diagram for Schedules

- **All Schedules**
- **View Serializable**
- **Conflict Serializable**
- **Avoid Cascading Abort**
- **Serial**
Q: Which schedules does Strict 2PL allow?

- All Schedules
- View Serializable
- Conflict Serializable
- Avoid Cascading
- Abort
- Serial

Lock Management

- Lock and unlock requests handled by the Lock Manager (LM).
- LM contains an entry for each currently held lock.
- Q: structure of a lock table entry?
Lock Management

- Lock and unlock requests handled by the Lock Manager (LM).
- LM contains an entry for each currently held lock.
- Lock table entry:
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests

Lock Management, cont.

- When lock request arrives see if any other xact holds a conflicting lock.
  - If not, create an entry and grant the lock
  - Else, put the requestor on the wait queue
- **Lock upgrade**: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Lock Management, cont.

- Two-phase locking is simple enough, right?
- We’re not done. There’s an important wrinkle …
Example: Output = ?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_S(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Lock_S(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
</tbody>
</table>

Example: Output = ?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_S(B)</td>
<td></td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Lock_S(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
</tr>
</tbody>
</table>

Outline

- Serializability - concepts and algorithms
- One solution: Locking
  - 2PL
  - variations
- Deadlocks
  - detection
  - prevention
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
- Many systems just punt and use Timeouts
  - What are the dangers with this approach?

Deadlock Detection

- Create a waits-for graph:
  - Nodes are transactions
  - Edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in waits-for graph

Deadlock Detection (Continued)

Example:

- T1: S(A), S(D), S(B)
- T2: X(B), X(C)
- T3: S(D), S(C), X(A)
- T4: X(B)

Diagram:

- T1 → T2
- T4 → T3
- T3 → T1
- T2 → T4
- T1 → T4
- T4 → T3
- T3 → T1
- T1 → T2
- T2 → T3
- T3 → T1
- T1 → T4
- T4 → T3
- T3 → T1
- T1 → T2
- T2 → T3
- T3 → T1
- T1 → T4
- T4 → T3
- T3 → T1
- T1 → T2
- T2 → T3
- T3 → T1
- T1 → T4
- T4 → T3
- T3 → T1
Another example

- is there a deadlock?
- if yes, which xacts are involved?

• now, is there a deadlock?
• if yes, which xacts are involved?

Deadlock detection

- how often should we run the algo?
- how many transactions are typically involved?
Deadlock handling

- Q: what to do?

Q0: what to do?
- A: select a 'victim' & 'rollback'
- Q1: which/how to choose?

Q1: which/how to choose?
- A1.1: by age
- A1.2: by progress
- A1.3: by # items locked already...
  - A1.4: by # xacts to rollback
- Q2: How far to rollback?
Deadlock handling

• Q2: How far to rollback?
  • A2.1: completely
  • A2.2: minimally
• Q3: Starvation??

Outline

• Serializability - concepts and algorithms
• One solution: Locking
  – 2PL
  – variations
• Deadlocks
  – detection
  – prevention
Deadlock Prevention

- Assign priorities based on timestamps (older -> higher priority)
- We only allow ‘old-wait-for-young’
- (or only allow ‘young-wait-for-old’)
- and rollback violators. Specifically:
  - Say Ti wants a lock that Tj holds - two policies:
    - **Wait-Die**: If Ti has higher priority, Ti waits for Tj;
      otherwise Ti aborts (i.e., old wait for young)
    - **Wound-wait**: If Ti has higher priority, Tj aborts;
      otherwise Ti waits (i.e., young wait for old)

Q: Why do these schemes guarantee no deadlocks?
A:

Q: When a transaction restarts, what is its (new) priority?
A:
Deadlock Prevention

- Q: Why do these schemes guarantee no deadlocks?
- A: only one ‘type’ of direction allowed.
- Q: When a transaction restarts, what is its (new) priority?
- A: its original timestamp. -- Why?

SQL statement

- usually, conc. control is transparent to the user, but
- LOCK <table-name> [EXCLUSIVE| SHARED]

Concurrency control - conclusions

- (conflict) serializability <-> correctness
- automatically correct interleavings:
  - locks + protocol (2PL, 2PLC, ...)
  - deadlock detection + handling
    - (or deadlock prevention)
Quiz:

• is there a serial schedule (= interleaving) that is not serializable?
• is there a serializable schedule that is not serial?
• can 2PL produce a non-serializable schedule? (assume no deadlocks)

Quiz - cont’d

• is there a serializable schedule that can not be produced by 2PL?
• a xact obeys 2PL - can it be involved in a non-serializable schedule?
• all xacts obey 2PL - can they end up in a deadlock?

Quiz - hints:

Q: 2PLC??
Quiz - hints:

- 2PL schedules
- Serializable schedules
- 2PLC
- Serial sch’s