Concurrency Control
(R&G ch. 17)

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

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:
  \[
  \begin{align*}
  &R(A) \ W(A) \quad R(B) \ W(B) \\
  &R(A) \ W(A) \quad R(B) \ W(B) \\
  &R(A) \ W(A) \ R(B) \ W(B) \\
  &\equiv \\
  &R(A) \ W(A) \ R(B) \ W(B)
  \end{align*}
  \]

Conflict Serializability (Continued)

- Here’s another example:
  \[
  \begin{align*}
  &R(A) \quad W(A) \\
  &R(A) \ W(A)
  \end{align*}
  \]

  • Serializable or not???

Conflict Serializability (Continued)

- Here’s another example:
  \[
  \begin{align*}
  &R(A) \quad W(A) \\
  &R(A) \ W(A)
  \end{align*}
  \]

  • Serializable or not???

  NOT!

Serializable

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

Dependency Graph

- One node per Xact
- Edge from Ti to Tj if:
  - An operation Oi of Ti conflicts with an operation Oj of Tj and
  - Oi appears earlier in the schedule than Oj.

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:

<table>
<thead>
<tr>
<th>T1:</th>
<th>R(A), W(A), R(B), W(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2:</td>
<td>R(A), W(A), R(B), W(B)</td>
</tr>
</tbody>
</table>

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

Example #2 (Lost update)

<table>
<thead>
<tr>
<th>T1:</th>
<th>Read(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N = N - 1</td>
</tr>
<tr>
<td></td>
<td>Write(N)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2:</th>
<th>Read(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N = N - 1</td>
</tr>
<tr>
<td></td>
<td>Write(N)</td>
</tr>
</tbody>
</table>

Example #3

<table>
<thead>
<tr>
<th>T1:</th>
<th>Read(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2:</th>
<th>Read(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write(A)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T3:</th>
<th>Read(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
</tbody>
</table>
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
R (A)
A = A-10
W (A)
R(A)
Sum = A
R (B)
Sum += B
R(B)
B = B+10
W(B)

T2
create a ‘correct’ schedule that is not conflict-serializable

Example #4’ (Inconsistent Analysis)

T1
R (A)
A = A-10
W (A)
R(A)
if (A>0), count=1
R (B)
if (B>0), count++
R(B)
B = B+10
W(B)

T2
A: T2 asks for
the count
of my active accounts

dependency
graph?
### An Aside: View Serializability

- Alternative (weaker) notion of serializability.
- Schedules $S_1$ and $S_2$ are **view equivalent** if:
  1. If $T_i$ reads initial value of $A$ in $S_1$, then $T_i$ also reads initial value of $A$ in $S_2$
  2. If $T_i$ reads value of $A$ written by $T_j$ in $S_1$, then $T_i$ also reads value of $A$ written by $T_j$ in $S_2$
  3. If $T_i$ writes final value of $A$ in $S_1$, then $T_i$ also writes final value of $A$ in $S_2$

### View Serializability

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

### 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”.

Lock Compatibility Matrix

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>X</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

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.

2PL

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

| T1 | R(A), W(A), R(B), W(B), Abort |
| T2 | R(A), W(A) |

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

Strict 2PL

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

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

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

Venn Diagram for Schedules
Q: Which schedules does Strict 2PL allow?

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

Example: Output = ?

<table>
<thead>
<tr>
<th>Lock_X(A)</th>
<th>Lock_S(B)</th>
<th>Read(B)</th>
<th>Lock_S(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>A = A-50</td>
<td>Write(A)</td>
<td>Lock_X(B)</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)               
T3: S(D), S(C), X(C)  
T4: X(A)

lock mgr:  
grant  
grant  
wait  
wait
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?

Deadlock handling

- 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??

Deadlock prevention

- 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