Principles of Database Management Systems

9: More on Transaction Processing
Pekka Kilpeläinen
(Partially based on Stanford CS245 slide originals by Hector Garcia-Molina, Jeff Ullman and Jennifer Widom)

Topics to discuss: [from Chapter 10]

• “Dirty reads” and cascading rollback
  - not solved by recovery (logging) or serializability (locking) techniques alone
  - How to avoid
• Transaction isolation in practice
• Deadlocks
  - Detection
  - Prevention

Problem of Uncommitted Data

• Recall:
  - **logging methods** provide reconstructing the DB to reflect the result of committed transactions
  - **locking** (or validation) methods provide serializability of transactions
• Problem yet uncovered: “dirty data”
  - data written by an uncommitted transaction (to buffers or to disk)
  - may lead to inconsistency, if read by others

Example (“dirty read”)

Assume 2PL with X-locks

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l(A); r l(A); A:=A+100; w l(A); l(B); u l(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>l(A); r l(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A:=A*2; w l(A)</td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>l(B); [Denied]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r l(B); Abort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u l(B); u l(A); r l(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:=B*2; w l(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u l(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constraint: A=B

Constraint violation: 250 50

Assume 2PL with X-locks

What’s the Problem?

• 2PL (as above) ensures correctness if transactions complete – But data written by an incomplete (and later aborted) transaction T may correspond to an inconsistent DB
• May require **cascading rollback**:
  - Transactions U that have read data written by an aborted T are rolled back (cancelling their effects on DB using log)
  - Transactions V that have read data written by U are rolled back ...
  - Expensive, and should be avoided

How to avoid cascading rollbacks?

.release write locks only after commit/abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>W l(A)</td>
<td></td>
</tr>
<tr>
<td>u l(A)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>Tj</td>
<td></td>
</tr>
</tbody>
</table>

• Clearly prevents Tj from reading dirty data, but ...

• If \(<T_i, \text{COMMIT}>\) not found on disk, recovery from a system failure cancels updates by \(T_i\)
  \(\rightarrow\) data read by \(T_j\) becomes dirty
  - Two solutions:
    • **Strict locking**
      - release \(T_i\)'s write locks only after its COMMIT/ABORT record flushed to disk
    • **Group commit**
      - release write locks only after commit/abort
      - flush log records to disk in the order they were created
        (immediate flushing of COMMIT-records not required)

### Recovery with group commit

**LOG:**

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Action</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;T_1, A, 10, 20&gt;)</td>
<td>(\rightarrow)</td>
<td></td>
</tr>
<tr>
<td>(&lt;T_1, \text{COMMIT}&gt;)</td>
<td>(\rightarrow)</td>
<td></td>
</tr>
<tr>
<td>(&lt;T_2, A, 20, 30&gt;)</td>
<td>(\rightarrow)</td>
<td></td>
</tr>
<tr>
<td>(&lt;T_2, \text{COMMIT}&gt;)</td>
<td>(\rightarrow)</td>
<td></td>
</tr>
</tbody>
</table>

If the log ends at...

1) both \(T_1\) and \(T_2\) cancelled; OK
2) \(T_2\) did not read uncommitted data; OK
   (and it is cancelled)
3) \(T_2\) did not read uncommitted data; OK

### Avoiding cascading rollbacks with validation

• **No change!**
  - Only committed transactions write (to buffer and to disk)
  \(\rightarrow\) there is no dirty data

### Transaction Isolation in Practise

• SQL2 (SQL-99) does not automatically require serializability

  • Transaction programmer may specify degree of concurrency control through "**isolation levels**":
    
    \[
    \text{set transaction isolation level [}
    \text{read uncommitted | read committed | repeatable read | serializable ]}
    \]

### SQL isolation levels
(from weakest to strongest):

• **read uncommitted**
  - no updates allowed; may read dirty data

• **read committed**
  - no dirty reads; repeated reading of some item may return different values

• **repeatable read**
  - adds stability of values to multiple reads of items; may encounter phantom tuples

• **serializable** (= what it says)

### Deadlocks

• **Detection**
  - Wait-for graph

• **Prevention**
  - **Timeout**
  - Wait-or-die
  - Wound-or-wait
Deadlock Detection

- Build a **wait-for graph** using lock table
  - Nodes: Active transactions;
  - Arcs: \((T,U)\) if \(T\) is waiting for a lock held by \(U\)
- Build incrementally or periodically
- When a cycle found, rollback culprit(s)

Timeout

- If transaction uses more than \(L\) sec.,
  cancel and roll it back!
- Simple scheme
- Hard to select \(L\)
- May lead to unnecessary rollbacks (of non-deadlocked transactions)

Timestamp-Based Deadlock Prevention

- Each transaction \(T_i\) is given a **timestamp** \(ts(T_i)\) when it starts
- If \(ts(T_i) < ts(T_j)\), transaction \(T_i\) is **older**, and transaction \(T_j\) is **younger**
- Two schemes
  - Wait-or-die
  - Wound-or-wait
  - Both give privilege to older transactions

Wait-or-die

- When \(T\) requesting a lock held by \(U\) ...
  - If \(T\) is older (i.e., \(ts(T) < ts(U)\)), \(T\) waits
  - If \(T\) is younger (i.e., \(ts(T) > ts(U)\)), \(T\) “dies” (is rolled back)

Example: (Wait-or-die)

```
T_1 (ts = 10)
   wait
T_2 (ts = 20)
   wait
 wait?
wait
T_3 (ts = 25)
```

Wound-or-wait

- When \(T\) requesting a lock held by \(U\) ...
  - If \(T\) is older (i.e., \(ts(T) < ts(U)\)), \(T\) “wounds” \(U\)
    - \(U\) is rolled back (releasing its locks), and \(T\) gets the lock it requested
  - If \(T\) is younger (i.e., \(ts(T) > ts(U)\)), \(T\) waits for \(U\) to release the lock

Example: (Wound-or-wait)
Example: (Wound-or-wait)

Why Do Wait-Die and Wound-Wait Work?

- Consider a deadlock, i.e., a cycle
  $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$
  in the wait-for graph
- With wait-or-die only older transactions wait,
  i.e., $ts(T_1) < ts(T_2) < \ldots < ts(T_1)$, which is impossible
- With wound-or-wait only younger transactions wait, which similarly cannot lead to a wait-for cycle

Timestamp-based vs. wait-for-graph based deadlock-management

- Timestamp methods prevent starvation:
  Restarted transactions retain their timestamp
  $\rightarrow$ each transaction becomes eventually oldest, and has a chance to finish
- Wait-die and wound-wait easier to implement than wait-for graphs (esp. for distributed databases)
- Wait-die and wound-wait may roll back transactions not involved in a deadlock

Summary

- Dirty reads and cascading rollback
- Transaction isolation in practice
- Deadlock
  - detection
  - prevention