Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo
Lecture#21: Concurrency Control – Part 1
(R&G ch. 17)

Last Class

• Introduction to Transactions
• ACID
• Concurrency Control
• Crash Recovery

For **Isolation** property, serial execution of transactions is safe but slow
– We want to find schedules equivalent to serial execution but allow interleaving.
• The way the DBMS does this is with its **concurrency control** protocol.

Today’s Class

• Serializability
• Two-Phase Locking
• Deadlocks
• Lock Granularities
Formal Properties of Schedules

- **Serializable Schedule**: A schedule that is *equivalent* to some serial execution of the transactions.
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Example

- Consider two txns:
  - T1 transfers $100 from B’s account to A’s
  - T2 credits both accounts with 6% interest.

- Assume at first A and B each have $1000.

```
BEGIN
A=A+100
A=A*1.06
COMMIT
T1
BEGIN
A=A+100
A=A*1.06
COMMIT
T2
```

Interleaving Example (Good)

```
BEGIN
A=A+100
B=B–100
COMMIT
T1
BEGIN
A=A*1.06
B=B*1.06
COMMIT
T2
```

Interleaving Example (Bad)

```
BEGIN
A=A+100
B=B–100
COMMIT
T1
BEGIN
A=A*1.06
B=B*1.06
COMMIT
T2
```

\[
\equiv
\]

\[
A=1166, B=954
\]

\[
A=1166, B=954
\]

A=116, B=954

or

A=1160, B=960

The bank lost $6!
Formal Properties of Schedules

- There are different levels of serializability:
  - **Conflict Serializability**
  - **View Serializability**

All DBMSs support this. This is harder but allows for more concurrency. Nobody does this.

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

- 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.

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

Conflict Serializability Intuition

- 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.
Conflict Serializability Intuition

Schedule

T1 | T2
---|---
BEGIN R(A) W(A) | BEGIN R(A) W(A)
R(B) W(B) COMMIT | R(B) W(B) COMMIT
R(B) | W(A)
W(A) | W(B)
COMMIT | COMMIT

Schedule

T1 | T2
---|---
BEGIN R(A) W(A) | BEGIN R(A) W(A)
R(B) | R(A)
W(B) | W(A)
COMMIT | COMMIT
R(B) | R(B)
W(B) | W(B)
COMMIT | COMMIT

Schedule

T1 | T2
---|---
BEGIN R(A) W(A) | BEGIN R(A) W(A)
R(B) | R(A)
W(B) | W(A)
COMMIT | COMMIT
R(B) | R(B)
W(B) | W(B)
COMMIT | COMMIT

Schedule

T1 | T2
---|---
BEGIN R(A) W(A) | BEGIN R(A) W(A)
R(B) | R(A)
W(B) | W(A)
COMMIT | COMMIT
R(B) | R(B)
W(B) | W(B)
COMMIT | COMMIT
Conflict Serializability Intuition

Schedule

T1
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

T2
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

Schedule

T1
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT

T2
BEGIN
R(A)
W(A)
R(B)
W(B)
COMMIT
Conflict Serializability Intuition

Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN R(A)</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

Serial Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEGIN R(A)</td>
<td>BEGIN R(A)</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

≡

Serializability

- **Q:** Are there any faster algorithms to figure this out other than transposing operations?

Dependency Graphs

- One node per txn.
- 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.
- Also known as a “precedence graph”
**Dependency Graphs**

- **Theorem:** A schedule is *conflict serializable* if and only if its dependency graph is acyclic.

---

**Example #1**

Schedule:

```
T1
BEGIN
R(A)
R(B)
W(A)
W(B)
COMMIT
```

```
T2
BEGIN
R(A)
R(B)
W(A)
W(B)
COMMIT
```

Dependency Graph:

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

---

**Example #2 – Lost Update**

Schedule:

```
T1
BEGIN
R(A)
W(A)
A = A - 1
COMMIT
```

```
T2
BEGIN
R(A)
W(A)
COMMIT
```

Dependency Graph:

- The cycle in the graph reveals the problem. The value of A depends on T2, and T2 depends on T1.
Example #2 – Lost Update

Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>R(A)</td>
<td>A = A - 1</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>A = A - 1</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Dependency Graph

T1 ———> A ———> T2

Example #3 – Threesome

• Q: Is this equivalent to a serial execution?
• A: Yes (T2, T1, T3)
  – Notice that T3 should go after T2, although it starts before it!

Example #4 – Inconsistent Analysis

Schedule

<table>
<thead>
<tr>
<th>TIME</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>R(A)</td>
<td>A = A - 10</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>A = A - 10</td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td>COMMIT</td>
</tr>
<tr>
<td>W(A)</td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

Dependency Graph

T1 ———> T2

T3

T1 ———> B

T2 ———> A

T3
Example #4 – Inconsistent Analysis

Schedule

\[
\begin{array}{ll}
T1 & T2 \\
\text{BEGIN} & \text{BEGIN} \\
R(A) & R(A) \\
A = A - 10 & \text{sum} = A \\
W(A) & R(B) \\
\text{sum} += B & \text{sum} += B \\
\text{COMMIT} & \text{ECHO}(\text{sum}) \\
R(B) & \text{ECHO}(\text{cnt}) \\
B = B + 10 & \text{cnt}++ \\
W(B) & \text{cnt}++ \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

Dependency Graph

T2 counts the number of active accounts.

Is it possible to create a schedule similar to this that is “correct” but still not conflict serializable?

View Serializability

- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are view equivalent if:
  - If T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
  - If T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2.
  - If T1 writes final value of A in S1, then T1 also writes final value of A in S2.
**View Serializability**

**Schedule**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>BEGIN</td>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td></td>
<td>COMIT</td>
<td>COMIT</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>

**Dependency Graph**

```
A
    /|
   / \
  T1 - A - T2
     |   |
     |   A
     |   |
     |   T3
```

**Serializability**

- **View Serializability** allows (slightly) more schedules than **Conflict Serializability** does.
  - But 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)

**In practice, Conflict Serializability is what gets used, because it can be enforced efficiently.**

**To allow more concurrency, some special cases get handled separately at the application level.**
Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

Executing with Locks

Two-Phase Locking

- Phase 1: Growing
  - Each txn requests the locks that it needs from the DBMS’s lock manager.
  - The lock manager grants/denies lock requests.
- Phase 2: Shrinking
  - The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.
Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

Executive with 2PL

- T1
  - BEGIN X-LOCK(A)
  - R(A)
  - W(A)
  - BEGIN X-LOCK(A)
  - R(A)
  - UNLOCK(A)
  - COMMIT
  - W(A)
  - UNLOCK(A)
  - COMMIT

- T2
  - Lock Manager
  - Granted (T1→A)
  - Denied!

- Lock Manager
  - Granted (T1→A)
  - Denied!
  - Released (T2→A)
  - Released (T1→A)
  - Granted (T2→A)
Two-Phase Locking

- 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 – Cascading Aborts

Schedule

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN</td>
</tr>
<tr>
<td>X-LOCK(A)</td>
<td>X-LOCK(A)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>R(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>ABORT</td>
</tr>
</tbody>
</table>
```

This is a permissible schedule in 2PL, but we have to abort T2 too.

This is all wasted work!

2PL Observations

- There are schedules that are serializable but would not be allowed by 2PL.
- Locking limits concurrency.
- May lead to deadlocks.
- May still have “dirty reads”
  - Solution: **Strict 2PL**

Strict Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
- Allows only conflict serializable schedules, but it is actually stronger than needed.

Release all locks at end of txn.
Strict Two-Phase Locking

- A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.
- Advantages:
  - Does not incur cascading aborts.
  - Aborted txns can be undone by just restoring original values of modified tuples.

Examples

- **T1**: Move $50 from Andy’s account to his bookie’s account.
- **T2**: Compute the total amount in all accounts and return it to the application.
- Legend:
  - **A** → Andy’s account.
  - **B** → The bookie’s account.

Non-2PL Example

Initial State

A = 100, B = 100

T2 Output

150

2PL Example

Initial State

A = 100, B = 100

T2 Output

200
Strict 2PL Example

Initial State
A=100, B=100

T2 Output
200

T1
BEGIN
X-LOCK(A)
R(A)
A=A-50
W(A)
X-LOCK(B)
R(B)
B=B+50
W(B)
UNLOCK(A)
UNLOCK(B)
COMMIT

T2
BEGIN
S-LOCK(A)
R(A)
S-LOCK(B)
R(B)
ECHO(A+B)
UNLOCK(A)
UNLOCK(B)
COMMIT

Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

It Just Got Real, Son

T1
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)

T2
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

Lock Manager

Granted (T1→A)
Denied!
Granted (T2→B)
Denied!
Deadlocks

- **Deadlock**: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock detection
  - Deadlock prevention

Deadlock Detection

- The DBMS creates a *waits-for* graph:
  - Nodes are transactions
  - Edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock
- The system periodically check for cycles in *waits-for* graph.
Deadlock Detection

- How often should we run the algorithm?
- How many txns are typically involved?
- What do we do when we find a deadlock?

Deadlock Handling

- Q: What do we do?
- A: Select a “victim” and rollback it back to break the deadlock.
Deadlock Handling

- **Q:** Which one do we choose?
- **A:** It depends…
  - By age (lowest timestamp)
  - By progress (least/most queries executed)
  - By the # of items already locked
  - By the # of txns that we have to rollback with it
- We also should consider the # of times a txn has been restarted in the past.

Deadlock Handling

- **Q:** How far do we rollback?
- **A:** It depends…
  - Completely
  - Minimally (i.e., just enough to release locks)

Deadlock Prevention

- When a txn tries to acquire a lock that is held by another txn, kill one of them to prevent a deadlock.
- No waits-for graph or detection algorithm.

Deadlock Prevention

- Assign priorities based on timestamps:
  - Older → higher priority (e.g., T1 > T2)
- Two different prevention policies:
  - **Wait-Die:** If T1 has higher priority, T1 waits for T2; otherwise T1 aborts (“old wait for young”)
  - **Wound-Wait:** If T1 has higher priority, T2 aborts; otherwise T1 waits (“young wait for old”)
**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?

---

**Today’s Class**

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities
Lock Granularities

• When we say that a txn acquires a “lock”, what does that actually mean?
  – On an Attribute? Tuple? Page? Table?
• Ideally, each txn should obtain fewest number of locks that is needed…

Database Lock Hierarchy

Example

• T1: Get the balance of Andy’s shady off-shore bank account.
• T2: Increase Christos’ bank account balance by 1%.

Example

• Q: What locks should they obtain?
• A: Multiple
  – Exclusive + Shared for leafs of lock tree.
  – Special Intention locks for higher levels
Intention Locks

- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.
- If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

Intention Locks

- Intention-Shared (IS): Indicates explicit locking at a lower level with shared locks.
- Intention-Exclusive (IX): Indicates locking at lower level with exclusive or shared locks.

Intention Locks

- Shared+Intention-Exclusive (SIX): The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

Compatibility Matrix

<table>
<thead>
<tr>
<th>T1 Holds</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>IX</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S</td>
<td>✔</td>
<td>X</td>
<td>✔</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SIX</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Example – Two-level Hierarchy

Read Andy’s record in R.

Tuple 1

Table R

Tuple 2

Tuple n

Write

Update Christos’ record in R.

Tuple 1

Tuple 2

Tuple n

Read

Update Christos’ record in R.
Example – Two-level Hierarchy

<table>
<thead>
<tr>
<th>T1 Holds</th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Update Christos’ record in R.

Example – Threesome

- Assume three txns execute at same time:
  - T1: Scan R and update a few tuples.
  - T2: Read a single tuple in R.
  - T3: Scan all tuples in R.

Scan R and update a few tuples.
Example – Threesome

Scan \( R \) and update a few tuples.

Read a single tuple in \( R \).

Read all tuples in \( R \).
Example – Threesome

Scan all tuples in \( R \).

\[ \begin{align*}
T1 & \quad S \\
T2 & \quad \text{SIX} \\
T3 & \quad X \\
\end{align*} \]

Multiple Lock Granularities

- Useful in practice as each txn only needs a few locks.
- Intention locks help improve concurrency:
  - \textit{Intention-Shared (IS)}: Intent to get \textit{S} lock(s) at finer granularity.
  - \textit{Intention-Exclusive (IX)}: Intent to get \textit{X} lock(s) at finer granularity.
  - \textit{Shared+Intention-Exclusive (SIX)}: Like \textit{S} and \textit{IX} at the same time.

Locking in Practice

- You typically don’t set locks manually.
- Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.
- Also useful for doing major changes.

\textbf{LOCK TABLE}

\begin{itemize}
  \item Explicitly locks a table.
  \item Not part of the SQL standard.
  \begin{itemize}
    \item Postgres Modes: \texttt{SHARE, EXCLUSIVE}
    \item MySQL Modes: \texttt{READ, WRITE}
  \end{itemize}
\end{itemize}
**SELECT...FOR UPDATE**

```sql
SELECT * FROM <table>
WHERE <qualification> FOR UPDATE;
```

- Perform a select and then sets an exclusive lock on the matching tuples.
- Can also set shared locks:
  - Postgres: `FOR SHARE`
  - MySQL: `LOCK IN SHARE MODE`

---

**Concurrency Control Summary**

- Conflict Serializability ↔ Correctness
- Automatically correct interleavings:
  - Locks + protocol (2PL, S2PL ...)
  - Deadlock detection + handling
  - Deadlock prevention