Today’s Class

- Serializability: concepts and algorithms
- Locking-based Concurrency Control:
  - 2PL
  - Strict 2PL
- Deadlocks

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.

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

- 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.

\[ \equiv \]

\begin{align*}
\text{T1} & \quad \text{T2} \\
\text{BEGIN} & \quad \text{BEGIN} \\
R(A) & \quad R(A) \\
W(A) & \quad W(A) \\
R(B) & \quad R(B) \\
W(B) & \quad W(B) \\
\text{COMMIT} & \quad \text{COMMIT} \\
\text{R(B)} & \quad \text{R(A)} \\
\text{W(A)} & \quad \text{W(A)} \\
\text{R(A)} & \quad \text{R(B)} \\
\text{W(B)} & \quad \text{W(B)} \\
\text{COMMIT} & \quad \text{COMMIT} \end{align*}
Serializability

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

Dependency Graphs

• One node per txn.
• 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.
• 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)
W(A)
R(B)
W(B)
COMMIT
```

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

Dependency Graph

```
A
```

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)
A = A - 1
W(A)
COMMITS

T2

Dependency Graph

A

T1

T2

Example #3 – Threesome

Schedule

T1

BEGIN
R(A)
W(A)

T2

BEGIN
R(A)
W(A)

T3

BEGIN
R(A)
W(A)

Dependency Graph

T1

T2

T3

A

B

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

T1

BEGIN
R(A)
A = A - 1
W(A)

T2

BEGIN
R(A)
A = A - 1
W(A)

Dependency Graph

T1

T2

B

A

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

Need an algorithm for generating serial schedule from an acyclic dependency graph.
    Topological Sorting
Example #4 – Inconsistent Analysis

Schedule

T1
BEGIN
R(A)
A = A - 10
W(A)
COMMIT

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

T2
BEGIN
R(A)
if(A>0): cnt++
R(B)
if(B>0): cnt++
ECHO(cnt)
COMMIT

Dependency Graph

T1
A
T2
B

T2 counts the number of active accounts.

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

T1
BEGIN
R(A)
W(A)
COMMIT

T2
BEGIN
W(A)
COMMIT

T3
BEGIN
W(A)
COMMIT

Dependency Graph

T1
A
T2
A
T3
A

View Serializability

Schedule

T1
BEGIN
R(A)
W(A)
COMMIT

T2
BEGIN
W(A)
COMMIT

T3
BEGIN
W(A)
COMMIT

VIEW

Allows all conflict serializable schedules + “blind writes”
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, such as for travel reservations, etc.

Schedules

All Schedules

- View Serializable
  - Conflict Serializable
    - Serial

Today’s Class

- Serializability: concepts and algorithms
- Locking-based Concurrency Control:
  - 2PL
    - Strict 2PL
- Deadlocks
Executing without Locks

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

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

Executing with Locks

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

Lock Types

- Basic Types:
  - S-LOCK – Shared Locks (reads)
  - X-LOCK – Exclusive Locks (writes)

• Q: If a txn only needs to read ‘A’, should it still get a lock?
• A: Yes, but you can get a shared lock.
Executing with Locks

```plaintext
CMU SCS

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

BEGIN
X-LOCK(A)
W(A)

BEGIN
S-LOCK(A)
R(A)
UNLOCK(A)
COMMIT

COMMIT
T1
T2

Transaction Manager

Granted (T1→A)
Released (T1→A)
 Granted (T2→A)
Released (T2→A)

Granted (T1→A)
Released (T1→A)

Faloutos/Pavlo
CMU SCS 15-415/615
40

Concurency Control

• We need to use a well-defined protocol that ensures that txns execute correctly.

Faloutos/Pavlo
CMU SCS 15-415/615
41

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.

Faloutos/Pavlo
CMU SCS 15-415/615
42

Two-Phase Locking

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

Transaction Lifetime

Faloutos/Pavlo
CMU SCS 15-415/615
43
Two-Phase Locking

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

Transaction Lifetime

Growing Phase  Shrinking Phase

Lock Management

- Lock and unlock requests handled by the DBMS’s lock manager (LM).
- LM contains an entry for each currently held lock:
  - Pointer to a list of txns holding the lock.
  - The type of lock held (shared or exclusive).
  - Pointer to queue of lock requests.

Executing with 2PL

LOCK (A)  
COMMIT  

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

T2  
BEGIN  
X-LOCK (A)  
R(A)  
W(A)  
UNLOCK (A)  
COMMIT

Lock Management

- When lock request arrives see if any other txn holds a conflicting lock.
  - If not, create an entry and grant the lock
  - Else, put the requestor on the wait queue
- All lock operations must be atomic.
- Lock upgrade: The txn that holds a shared lock upgrade to hold an exclusive lock.
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></td>
</tr>
<tr>
<td>R(A)</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>W(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>ABORT</td>
<td></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 \textit{strict} if a value written by a txn is not read or overwritten by other txns until that txn finishes.
- Advantages:
  - Recoverable.
  - Do not require cascading aborts.
  - Aborted txns can be undone by just restoring original values of modified tuples.

Examples

- **T1**: Move $50 from Christos’ account to his bookie’s account.
- **T2**: Compute the total amount in all accounts and return it to the application.
- Legend:
  - \( A \rightarrow \) Christos’ account.
  - \( B \rightarrow \) 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

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>A=100, B=100</td>
<td></td>
</tr>
<tr>
<td>T1 Output</td>
<td>200</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>(X-\text{LOCK}(A))</td>
<td>(S-\text{LOCK}(A))</td>
</tr>
<tr>
<td>(R(A))</td>
<td>(R(A))</td>
<td></td>
</tr>
<tr>
<td>(A=A-50)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(W(A))</td>
<td>(W(A))</td>
<td></td>
</tr>
<tr>
<td>(X-\text{LOCK}(B))</td>
<td>(S-\text{LOCK}(B))</td>
<td></td>
</tr>
<tr>
<td>(R(B))</td>
<td>(R(B))</td>
<td></td>
</tr>
<tr>
<td>(B=B+50)</td>
<td>(ECHO(A+B))</td>
<td></td>
</tr>
<tr>
<td>(W(B))</td>
<td>(UNLOCK(A))</td>
<td></td>
</tr>
<tr>
<td>(UNLOCK(B))</td>
<td>(UNLOCK(B))</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>COMMIT</td>
<td></td>
</tr>
</tbody>
</table>

### Strict Two-Phase Locking

**Q:** Why is avoiding “dirty reads” important?

**A:** If a txn aborts, all actions must be undone. Any txn that read modified data must also be aborted.

**Good:**
- Avoids “dirty reads” etc

**Bad:**
- Limits concurrency even more
- And still may lead to deadlocks
Schedules

- All Schedules
- View Serializable
- Conflict Serializable
- Avoid Cascading Abort
- Strict 2PL
- Serial

Two-Phase Locking

- 2PL seems to work well.
- Is that enough? Can we just go home now?

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?

Shit Just Got Real

- TIME
- Lock Manager

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

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

- Granted (T1→A)
- Granted (T2→B)
- Denied!
- Denied!
Today’s Class

• Serializability: concepts and algorithms
• One solution: Locking
  – 2PL
  – variations
• Deadlocks:
  – Detection
  – Prevention

Deadlock Detection

• The DBMS creates a *waits-for* graph:
  – Nodes are transactions
  – Edge from Ti to Tj if Ti is waiting for Tj to release a lock
• The system periodically check for cycles in *waits-for* graph.

Deadlock Detection

• Schedule
• *Waits-for* Graph

• 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.

- **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.

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

Today’s Class

- Serializability: concepts and algorithms
- One solution: Locking
  - 2PL
  - variations
- Deadlocks:
  - Detection
  - Prevention
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 $\rightarrow$ higher priority (e.g., $T_1 > T_2$)
- Two different prevention policies:
  - **Wait-Die:** If $T_1$ has higher priority, $T_1$ waits for $T_2$; otherwise $T_1$ aborts (“old wait for young”)
  - **Wound-Wait:** If $T_1$ has higher priority, $T_2$ aborts; otherwise $T_1$ waits (“young wait for old”)

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?
Performance Problems

- Executing more txns can increase the throughput.
- But there is a tipping point where adding more txns actually makes performance worse.

Lock Thrashing

- When a txn holds a lock, other txns have to wait for it to finish.
- If you have a lot of txns with a lot of locks, then you will have a lot of waiting.
- A lot of waiting means txns take longer and hold their locks longer…

Lock Thrashing

No Locks

With Locks

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.
LOCK TABLE

- Explicitly locks a table.
- Not part of the SQL standard.
  - Postgres Modes: **SHARE, EXCLUSIVE**
  - MySQL Modes: **READ, WRITE**

SELECT...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
- **Big Assumption**: The database is fixed.
  - That is, objects are not inserted or deleted.