Last Class

- Introduction to Transactions
- ACID
- Concurrency Control (2PL)
- Crash Recovery (WAL)

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

Formal Properties of Schedules

- **Serial Schedule**: A schedule that does not interleave the actions of different transactions.
- **Equivalent Schedules**: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.*

(*) no matter what the arithmetic operations are!

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.

Example

• Assume at first A and B each have $1000.
• Q: What are the possible outcomes of running T1 and T2?

Example

• Q: What are the possible outcomes of running T1 and T2 together?
• A: Many! But A+B should be:
  $2000*1.06=$2120
• There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running **serially** in some order.
Example

- Legal outcomes:
  - A=1166, B=954 → $2120
  - A=1160, B=960 → $2120
- The outcome depends on whether T1 executes before T2 or vice versa.

Serial Execution Example

Interleaving Example (Good)
Interleaving Example (Bad)

Schedule

\[
\begin{array}{c|c}
\text{T1} & \text{T2} \\
\hline
\text{BEGIN} & \text{BEGIN} \\
A=a+100 & a^*1.06 \\
\text{B=b-100} & B=b^*1.06 \\
\text{COMMIT} & \text{COMMIT} \\
\end{array}
\]

\[A=1166, \; B=954 \] or \[A=1160, \; B=960\]

The bank lost $6!

Formal Properties of Schedules

- There are different levels of serializability:
  - **Conflict Serializability**
    - All DBMSs support this.
  - **View Serializability**
    - 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.
  – Note that some serializable schedules are NOT conflict serializable.

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.
Conflict Serializability Intuition

Schedule

Serial Schedule

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**

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 = 1
W(A)
COMMIT
```

```
T2
BEGIN
R(A)
A = A - 1
W(A)
COMMIT
```

**Dependency Graph**

A

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

• Need an algorithm for generating serial schedule from an acyclic dependency graph.
  – **Topological Sorting**

Example #4 – Inconsistent Analysis

Is it possible to create a schedule similar to this that is “correct” but still not conflict serializable?
Example #4 – Inconsistent Analysis

Schedule

\[
\text{BEGIN R(A) A = A - 10 W(A)}
\]

\[
\text{if (A>0) \{ cnt++ \}}
\]

\[
\text{if (B>0) \{ cnt++ \}}
\]

\[
\text{R(B) B = B + 10 W(B)}
\]

\[
\text{COMMIT}
\]

Dependency Graph

A

T1

T2

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

Schedule

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

Schedule

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

\[
\text{BEGIN R(A) W(A) COMMIT}
\]

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)

Serializability

- 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

Two-Phase Locking

• Phase 1: Growing
  – Eachtxn 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.

Executing with 2PL

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

<table>
<thead>
<tr>
<th>Lock Manager</th>
<th>Granted (T1→A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denied!</td>
<td></td>
</tr>
<tr>
<td>Released (T1→A)</td>
<td>Granted (T2→A)</td>
</tr>
<tr>
<td>Released (T2→A)</td>
<td></td>
</tr>
</tbody>
</table>
Lock Types

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

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Exclusive</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

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.

- 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

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

This is all wasted work!
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.

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 → Christos’ account.
  - B → The bookie’s account.

Non-2PL Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Initial State</th>
<th>T2 Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>BEGIN X-LOCK(A)</td>
<td>A=100, B=100</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>S-LOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A=A-50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X-LOCK(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B=B+50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S-LOCK(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNLOCK(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COMMIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ECHO(A+B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2PL Example

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

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

Initial State
A=100, B=100

T2 Output
200

Strict 2PL Example

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

Initial State
A=100, B=100

T2 Output
200

Schedules

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

Initial State
A=100, B=100

T2 Output
200

CMU SCS
2PL Example

CMU SCS
Strict 2PL Example

CMU SCS
Schedules
Two-Phase Locking

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

Shit Just Got Real

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?
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.

```
BEGIN
S-LOCK(A)
BEGIN
S-LOCK(D)
BEGIN
S-LOCK(B)
BEGIN
X-LOCK(B)
X-LOCK(8)
BEGIN
S-LOCK(A)
```

Schedule

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)

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 → 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”)

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…

No Locks

<table>
<thead>
<tr>
<th># of Concurrent Txs</th>
<th>Throughput (txn/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>200</td>
<td>2.5</td>
</tr>
<tr>
<td>400</td>
<td>3.5</td>
</tr>
<tr>
<td>600</td>
<td>4.0</td>
</tr>
<tr>
<td>800</td>
<td>4.5</td>
</tr>
<tr>
<td>1000</td>
<td>5.0</td>
</tr>
<tr>
<td>1200</td>
<td>5.5</td>
</tr>
</tbody>
</table>

With Locks

<table>
<thead>
<tr>
<th># of Concurrent Txs</th>
<th>Throughput (txn/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>200</td>
<td>1.0</td>
</tr>
<tr>
<td>400</td>
<td>1.5</td>
</tr>
<tr>
<td>600</td>
<td>2.0</td>
</tr>
<tr>
<td>800</td>
<td>2.5</td>
</tr>
<tr>
<td>1000</td>
<td>3.0</td>
</tr>
<tr>
<td>1200</td>
<td>3.5</td>
</tr>
</tbody>
</table>
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: `SHARED, 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`
Locking Demo

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.