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.

Last Class

Today’s Class

• Serializability
• Two-Phase Locking
• Deadlocks
• Lock Granularities
• Locking in B+Trees
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!)

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.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BEGIN</strong></td>
<td><strong>BEGIN</strong></td>
</tr>
<tr>
<td>A=A+100</td>
<td>A=A*1.06</td>
</tr>
<tr>
<td>B=B–100</td>
<td>B=B*1.06</td>
</tr>
<tr>
<td><strong>COMMIT</strong></td>
<td><strong>COMMIT</strong></td>
</tr>
</tbody>
</table>

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

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

\[
\begin{aligned}
A &= 1166, B &= 954 \\
A &= 1166, B &= 954 \\
\end{aligned}
\]

Interleaving Example (Bad)

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

\[
\begin{aligned}
A &= 1166, B &= 954 \\
A &= 1166, B &= 960 \\
\text{or} \\
A &= 1160, B &= 960 \\
\end{aligned}
\]

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 \textit{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 \textit{conflict serializable} if:
  - S is conflict equivalent to some serial schedule.

Conflict Serializability Intuition

- A schedule S is \textit{conflict serializable} if:
  - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.
Serializability

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

Dependency Graphs

- One node pertxn.
- 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”

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

Example #1

- Schedule
- 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

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

T1

T2

Example #3 – Threesome

Schedule

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

T1

T2

Example #3 – Threesome

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

T1

T2

T3

Example #3 – Threesome

```
BEGIN
R(A)
W(A)
```

T3

```
BEGIN
R(B)
W(B)
```

T1

T2

Example #4 – Inconsistent Analysis

Schedule

```
BEGIN
R(A)
A = A - 10
W(A)
```

T1

```
BEGIN
R(B)
B = B + 10
W(B)
```

T2

Dependency Graph

```
T1
```

```
T2
```

```
T3
```

Example #4 – Inconsistent Analysis

```
BEGIN
R(A)
A = A - 10
W(A)
```

T1

```
BEGIN
R(B)
B = B + 10
W(B)
```

T2

```
ECHO(sum)
```

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

Schedule

T1
BEGIN R(A)
A = A-10
W(A)
if(A>0): cnt++
R(B)
if(B>0): cnt++
W(B)
COMMIT

T2
BEGIN
R(A)
B = B+10
W(B)
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.

Schedule

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

T2
BEGIN
R(A)
W(A)
COMMIT

T3
BEGIN
W(A)
COMMIT

Schedule

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

T2
BEGIN
R(A)
W(A)
COMMIT

T3
BEGIN
W(A)
COMMIT

VIEW

Schedule

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

T2
BEGIN
R(A)
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)

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**
  - **Serial**
  - **Conflict Serializable**
  - **View Serializable**

Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities
- Locking in B+Trees
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.

Two-Phase Locking

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

Transaction Lifetime

Growing Phase

Shrinking Phase

Transaction Lifetime

Growing Phase

Shrinking Phase

2PL Violation!
Executing with 2PL

2PL – Cascading Aborts

2PL Observations

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.

- 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

- 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.

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

<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>S-LOCK(A)</td>
</tr>
<tr>
<td>R(A)</td>
<td>=</td>
</tr>
<tr>
<td>A=A-50</td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td>UNLOCK(A)</td>
</tr>
<tr>
<td>UNLOCK(A)</td>
<td>S-LOCK(B)</td>
</tr>
<tr>
<td>X-LOCK(B)</td>
<td>=</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
<tr>
<td>B=B+50</td>
<td>UNLOCK(B)</td>
</tr>
<tr>
<td>W(B)</td>
<td>ECHO(A+B)</td>
</tr>
<tr>
<td>UNLOCK(B)</td>
<td>COMMIT</td>
</tr>
</tbody>
</table>
```
Strict Two-Phase Locking

- Txns hold all of their locks until commit.
- Good:
  - Avoids “dirty reads” etc
- Bad:
  - Limits concurrency even more
  - And still may lead to deadlocks
Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities
- Locking in B+Trees

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

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

T1 T2
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)

Lock Manager

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

TIME

Lock Manager
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 Handling

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

- 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:** 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 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 Handling

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

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
- Locking in B+Trees

---

**Lock Granularities**

- When we say that a txn acquires a “lock”, what does that actually mean?
  - On a field? Record? 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 offshore bank account.
• T2: Increase all account balances by 1%.
• Q: What locks should they obtain?

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.

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></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IS</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td><strong>IX</strong></td>
<td>✔</td>
<td>✔</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>S</strong></td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><strong>SIX</strong></td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>X</td>
</tr>
</tbody>
</table>

Multiple Granularity Protocol

- **Stronger** Privileges
- **Weaker** Privileges

T1 Holds

T2 Wants
Locking Protocol

- Each txn obtains appropriate lock at highest level of the database hierarchy.
- To get S or IS lock on a node, the txn must hold at least IS on parent node.
  - What if txn holds SIX on parent? S on parent?
- To get X, IX, or SIX on a node, must hold at least IX on parent node.

Example – Two-level Hierarchy

<table>
<thead>
<tr>
<th>T2 Wants</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>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>X</td>
<td>X</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 – Threesome

- Assume three txns execute at same time:
  - T1: Scan R and update a few tuples.
  - T2: Scan a portion of tuples in R.
  - T3: Scan all tuples in R.
Example – Threesome

• **T1:** Get an **SIX** lock on \(R\), then get **X** lock on tuples that are updated.
• **T2:** Get an **IS** lock on \(R\), and repeatedly get an **S** lock on tuples of \(R\).
• **T3:** Two choices:
  – T3 gets an **S** lock on \(R\).
  – OR, T3 could behave like T2; can use **lock escalation** to decide which.

Lock Escalation

• Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired.
• Reduces the number of requests that the lock manager has to process.

Multiple Lock Granularities

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

Today’s Class

• Serializability
• Two-Phase Locking
• Deadlocks
• Lock Granularities
• Locking in B+Trees
Locking in B+Trees

- **Q:** What about locking indexes?
- **A:** They are not quite like other database elements so we can treat them differently:
  - It’s okay to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained.

Example

- **T1** wants to insert in **H**
- **T2** wants to insert in **I**
- **Q:** Why not plain 2PL?
- **A:** Because txns have to hold on to their locks for too long!

Lock Crabbing

- Improves concurrency for B+Trees.
- Get lock for parent; get lock for child; release lock for parent if “safe”.
- **Safe Nodes:** Any node that won’t split or merge when updated.
  - Not full (on insertion)
  - More than half-full (on deletion)

Lock Crabbing

- **Search:** Start at root and go down; repeatedly,
  - S lock child
  - then unlock parent
- **Insert/Delete:** Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
  - If child is safe, release all locks on ancestors.
### Example #1 – Search 38

It’s safe to release the lock on A.

### Example #2 – Delete 38

We may need to coalesce B, so we can’t release the lock on A. We know that C will not need to merge with F, so it’s safe to release A+B.

### Example #3 – Insert 45

We know that if C needs to split, B has room so it’s safe to release A.

### Example #4 – Insert 25

We need to split H so we need to keep the lock on its parent node.
Problems

• **Q:** What was the first step that all of the update examples did on the B+Tree?

![Diagram showing update examples on a B+Tree](image)

• **A:** Locking the root every time becomes a bottleneck with higher concurrency.

• **Can we do better?**

Better Tree Locking Algorithm

• **Main Idea:**
  – Assume that the leaf is ‘safe’, and use S-locks & crabbing to reach it, and verify.
  – If leaf is not safe, then do previous algorithm.


Example #2 – Delete 38

![Diagram showing an example of deleting 38 from a B+Tree](image)

D will not need to coalesce, so we’re safe!
**Example #4 – Insert 25**

We need to split H so we have to restart and re-execute like before.

**Better Tree Locking Algorithm**

- **Search**: Same as before.
- **Insert/Delete**:
  - Set locks as if for search, get to leaf, and set X lock on leaf.
  - If leaf is not safe, release all locks, and restart txn using previous Insert/Delete protocol.
- Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful.

**Additional Points**

- **Q**: Which order to release locks in multiple-granularity locking?
  - **A**: From the bottom up
- **Q**: Which order to release locks in tree-locking?
  - **A**: As early as possible to maximize concurrency.

**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