Today...

- Last Two Sessions:
  - DBMS Internals- Part XI
    - Transaction Management
  - Student Presentations of P3

- Today’s Session:
  - Transaction Management (Cont’d)

- Announcement:
  - PS4 is due on Tuesday, April 15th
Lock Conversions

- A transaction may need to change the lock it already acquires on an object
  - From Shared to Exclusive
    - This is referred to as *lock upgrade*
  - From Exclusive to Shared
    - This is referred to as *lock downgrade*

- For example, an SQL update statement might acquire a Shared lock *on each row*, \( R \), in a table and if \( R \) satisfies the condition (in the WHERE clause), an Exclusive lock must be obtained for \( R \)
Lock Upgrades

- A lock upgrade request from a transaction \( T \) on object \( O \) must be handled specially by:
  - Granting an Exclusive lock to \( T \) immediately if no other transaction holds a Shared lock on \( O \)
  - Otherwise, queuing \( T \) at the front of \( O \)'s queue (i.e., \( T \) is favored)

- \( T \) is favored because it already holds a Shared lock on \( O \)
  - Queuing \( T \) in front of another transaction \( T' \) that holds no lock on \( O \), but requested an Exclusive lock on \( O \) averts a deadlock!
  - However, if \( T \) and \( T' \) hold a Shared lock on \( O \), and both request upgrades to an Exclusive lock, a deadlock will arise regardless!
Lock Downgrades

- Lock upgrades can be entirely avoided by obtaining Exclusive locks *initially*, and downgrade them to Shared locks once needed

- **Would this violate any 2PL requirement?**
  - On the surface yes; since the transaction (say, $T$) may need to upgrade later
  - This is, however, a special case as $T$ conservatively obtained an Exclusive lock, and did nothing but read the object that it downgraded
  - 2PL can be safely extended to allow lock downgrades in the growing phase, *provided that the transaction has not modified the object*
  - This might reduce concurrency (due to obtaining some unnecessary Exclusive locks) but improve throughput (due to reducing deadlocks)!
Outline

- Lock Conversions
- Dealing with Deadlocks
- Dynamic Databases and the Phantom Problem
- Concurrency Control in B+ Trees
Deadlock Detection

- The lock manager maintains a structure called a *waits-for graph* to *periodically* detect deadlocks.

- In a waits-for graph:
  - The nodes correspond to active transactions.
  - There is an edge from Ti to Tj *if and only if* Ti is waiting for Tj to release a lock.

- The lock manager *adds* and *removes* edges to and from a waits-for graph when it *queues* and *grants* lock requests, respectively.

- A deadlock is detected when a *cycle* in the waits-for graph is found.
The following schedule is free of deadlocks:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td></td>
</tr>
<tr>
<td>S(B)</td>
<td></td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X(C)</td>
<td>S(C)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(C)</td>
</tr>
</tbody>
</table>

A schedule *without* a deadlock

The Corresponding *Waits-For Graph*

No cycles; hence, no deadlocks!

*The nodes correspond to active transactions and there is an edge from Ti to Tj *if and only if* Ti is waiting for Tj to release a lock*
Deadlock Detection (Cont’d)

- The following schedule is **NOT** free of deadlocks:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td>X(C)</td>
<td>R(C)</td>
<td>S(C)</td>
<td>W(B)</td>
</tr>
<tr>
<td>X(A)</td>
<td>X(B)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  A schedule *with* a deadlock

  The Corresponding *Waits-For Graph*

  *The nodes correspond to active transactions and there is an edge from Ti to Tj *if and only if* Ti is waiting for Tj to release a lock*
The following schedule is **NOT** free of deadlocks:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>R(A)</td>
<td>X(B)</td>
<td>S(C)</td>
</tr>
<tr>
<td>S(B)</td>
<td>W(B)</td>
<td>X(C)</td>
<td>R(C)</td>
</tr>
<tr>
<td>X(A)</td>
<td>X(B)</td>
<td>X(A)</td>
<td>X(B)</td>
</tr>
</tbody>
</table>

*The nodes correspond to active transactions and there is an edge from Ti to Tj *if and only if* Ti is waiting for Tj to release a lock*
Resolving Deadlocks

- A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks
  - This allows some of the waiting transactions to proceed

- The choice of which transaction to abort can be made using different criteria:
  - The one with the fewest locks
  - Or the one that has done the least work
  - Or the one that is farthest from completion (*more accurate*)

- **Caveat**: a transaction that was aborted in the past, should be *favored* subsequently and not aborted upon a deadlock detection!
Deadlock Prevention

- Studies indicate that deadlocks are relatively infrequent and *detection-based schemes* work well in practice.

- However, if there is a high level of *contention* for locks, *prevention-based schemes* could perform better.

- Deadlocks can be averted by giving each transaction a *priority* and ensuring that lower-priority transactions are not allowed to wait for higher-priority ones (or vice versa).
Deadlock Prevention (*Cont’d*)

- One way to assign priorities is to give each transaction a *timestamp* when it is started
  - Thus, the lower the timestamp, the higher is the transaction’s priority

- If a transaction \( T_i \) requests a lock and a transaction \( T_j \) holds a conflicting lock, the lock manager can use one of the following policies:
  - **Wound-Wait**: If \( T_i \) has higher priority, \( T_j \) is aborted; otherwise, \( T_i \) waits
  - **Wait-Die**: If \( T_i \) has higher priority, it is allowed to wait; otherwise, it is aborted
Reissuing Timestamps

- A subtle point is that we must ensure that no transaction is perennially aborted because it never had a sufficiently high priority.

- To avoid that, when a transaction is aborted and restarted, it should be given the same timestamp it had originally.
  - This policy is referred to as reissuing timestamps.

- Reissuing timestamps ensures that each transaction will eventually become the oldest and accordingly get all the locks it requires!
Outline

- Lock Conversions
- Dealing with Deadlocks
- Dynamic Databases and the Phantom Problem
- Concurrency Control in B+ Trees
Dynamic Databases

- Thus far, we have assumed *static databases* in a sense that they do not *grow* and *shrink*.

- We now relax that condition and assume *dynamic databases* (i.e., databases that grow and shrink).

- To study locking protocols for dynamic databases, we consider the following:
  - A Sailors relation S
  - A transaction **T1** which scans S to find the oldest Sailor for each of the rating levels 1 and 2
  - A transaction **T2** which inserts a new Sailor with rating 1 and age 96
A Possible Scenario

- Assume a scenario whereby the actions of $T_1$ and $T_2$ are interleaved as follows:
  - $T_1$ identifies all pages containing Sailors with rating 1 (say, pages $A$ and $B$)
  - $T_1$ finds the age of the oldest Sailor with rating 1 (say, 71)
  - $T_2$ inserts a new Sailor with rating 1 and age 96 (perhaps into page $C$ which does not contain any Sailor with rating 1)
  - $T_2$ locates the page containing the oldest Sailor with rating 2 (say, page $D$) and deletes this Sailor (whose age is, say, 80)
  - $T_2$ commits
  - $T_1$ identifies all pages containing Sailors with rating 2 (say, pages $D$ and $E$), and finds the age of the oldest such Sailor (which is, say, 63)
  - $T_1$ commits
A Possible Scenario (Cont’d)

- We can apply strict 2PL to the given interleaved actions of **T1** and **T2** as follows (S = Shared; X = Exclusive):

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td>R(C)</td>
</tr>
<tr>
<td>R(B)</td>
<td>W(C)</td>
</tr>
<tr>
<td></td>
<td>R(D)</td>
</tr>
<tr>
<td></td>
<td>W(D)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(A)</td>
<td>S(B)</td>
</tr>
<tr>
<td>R(A)</td>
<td>S(B)</td>
</tr>
<tr>
<td>R(B)</td>
<td>R(B)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>E(C)</td>
<td>E(D)</td>
</tr>
<tr>
<td>R(C)</td>
<td>R(D)</td>
</tr>
<tr>
<td>W(C)</td>
<td>W(D)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(D)</td>
<td>S(E)</td>
</tr>
<tr>
<td>R(D)</td>
<td>R(E)</td>
</tr>
<tr>
<td>S(E)</td>
<td>R(E)</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
A Possible Scenario (Cont’d)

- We can apply strict 2PL to the given interleaved actions of \( T1 \) and \( T2 \) as follows (\( S = \) Shared; \( X = \) Exclusive):

\[
\begin{array}{c|c|c}
\hline
T1 & T2 & \\
R(A) & R(A) & S(A) \\
R(B) & R(B) & R(A) \\
R(C) & R(C) & R(D) \\
\text{W(C)} & \text{S(B)} & \text{W(C)} \\
\text{R(D)} & \text{E(D)} & \text{R(D)} \\
\text{W(D)} & \text{E(D)} & \text{W(D)} \\
\text{Commit} & \text{Commit} & \text{Commit} \\
\hline
\end{array}
\]

- A tuple with rating 1 and age 71 is returned
- A tuple with rating 1 and age 96 is inserted
- A tuple with rating 2 and age 80 is deleted
- A tuple with rating 2 and age 63 is returned
A Possible Scenario (Cont’d)

- One possible serial execution of \textbf{T1} and \textbf{T2} is as follows (\textbf{S} = Shared; \textbf{X} = Exclusive):

\begin{tabular}{|c|c|}
\hline
\textbf{T1} & \textbf{T2} \\
\hline
R(A) & S(A) \\
R(B) & R(A) \\
R(D) & S(B) \\
R(E) & R(B) \\
Commit & S(D) \\
R(C) & S(D) \\
W(C) & R(D) \\
R(D) & S(E) \\
W(D) & R(E) \\
Commit & Commit \\
\hline
\end{tabular}

- A tuple with rating 1 and age 71 is returned
- A tuple with rating 2 and age 80 is returned
- A tuple with rating 1 and age 96 is inserted
- A tuple with rating 2 and age 80 is deleted
A Possible Scenario (Cont’d)

- Another possible *serial execution* of T1 and T2 is as follows *(S = Shared; X = Exclusive):*

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A) R(B) R(D) R(E) Commit</td>
<td>R(C) W(C) R(D) W(D) Commit</td>
</tr>
</tbody>
</table>

A tuple with rating 1 and age 96 is returned

A tuple with rating 1 and age 96 is inserted

A tuple with rating 2 and age 80 is deleted

A tuple with rating 2 and age 63 is returned
A Possible Scenario: Revisit

- We can apply strict 2PL to the given interleaved actions of \textbf{T1} and \textbf{T2} as follows (\textbf{S} = Shared; \textbf{X} = Exclusive):

\begin{tabular}{|c|c|}
\hline
\textbf{T1} & \textbf{T2} \\
\hline
R(A) & S(A) \\
R(B) & R(A) \\
R(C) & S(B) \\
W(C) & R(B) \\
R(D) & E(C) \\
W(D) & R(D) \\
Commit & W(C) \\
R(E) & E(D) \\
Commit & W(D) \\
\hline
\end{tabular}

- A tuple with rating 1 and age 71 is returned
- A tuple with rating 1 and age 96 is inserted
- A tuple with rating 2 and age 80 is deleted
- A tuple with rating 2 and age 63 is returned

This schedule is not identical to any serial execution of T1 and T2!
The Phantom Problem

- The problem is that \textbf{T1} assumes that it has locked “all” the pages which contain Sailors records with rating 1.

- This assumption is violated when \textbf{T2} inserts a new Sailor record with rating 1 on a different page.

- Hence, locking pages at any given time does not prevent new \textit{phantom} records from being added on other pages! This is commonly known as the “\textit{Phantom Problem}.”

- The Phantom Problem is caused, not because of a flaw in the Strict 2PL protocol, but because of \textbf{T1’s} unrealistic assumptions.
How Can We Solve the Phantom Problem?

- If there is *no index* on rating and all pages in Sailors must be scanned, \textbf{T1} should somehow ensure that no *new* pages are inserted to the Sailors relation
  - This has to do with the *locking granularity*

- If there is an *index* on rating, \textbf{T1} can lock the index entries and the data pages which involve the targeted ratings, and accordingly prevent new insertions
  - This technique is known as *index locking*
Outline

- Lock Conversions
- Dealing with Deadlocks
- Dynamic Databases and the Phantom Problem
- Concurrency Control in B+ Trees
Concurrency Control in B+ Trees

- We focus on applying concurrency control on B+ trees for:
  - Searches
  - Insertions/deletions

- Three observations provide the necessary insights to apply a locking protocol for B+ trees:
  1. The higher levels of a B+ tree only direct searches
  2. Searches never go back up a B+ tree when they proceed along paths to desired leaves
  3. Insertions/deletions can cause splits/merges, which might propagate all the way up, from leaves to the root of a B+ tree
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

Example: Search for data entry 38*

```
20
35
```

Obtain a Shared Lock
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*

```
<table>
<thead>
<tr>
<th>20*</th>
<th>22*</th>
</tr>
</thead>
<tbody>
<tr>
<td>23*</td>
<td>24*</td>
</tr>
<tr>
<td>35*</td>
<td>36*</td>
</tr>
<tr>
<td>38*</td>
<td>41*</td>
</tr>
<tr>
<td>44*</td>
<td></td>
</tr>
</tbody>
</table>
```

- Obtain a Shared Lock
- Release the Shared Lock
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*

```
  20
   
  20*
  22*
  23*
  24*
  35*
  36*
  38*
  41*
  44*

Obtain a Shared Lock
```
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*

```
  20
  /  \
23   35
  \
 22   36
```

Obtain a Shared Lock

```
  20*
  /  \
23*  35*
```

Release the Shared Lock

```
  23*
  /  \
24*  38*
```

Obtain a Shared Lock

```
  35*
```

Obtain a Shared Lock

```
  38*  41*
```

Obtain a Shared Lock

```
  44*  44*
```
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.

- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.

- Example: Search for data entry 38*

```
  20  
  /  
35   
/     
23    23* 24* 35* 36* 38* 41* 44* 
```
A Locking Strategy for Searches

- A search should obtain Shared locks on nodes, starting at the root and proceeding along the path to the desired leaf.
- Since searches never go back up the tree, a lock on a node can be released as soon as a lock on a child node is obtained.
- Example: Search for data entry 38*

Keep Locked Until the Result is Returned
Towards A Locking Strategy for Insertions/Deletions

- A conservative strategy for an insertion/deletion would be to obtain Exclusive locks on all the nodes along the path to the desired leaf
  - This is because splits/merges can propagate all the way up to the root

- However, once a child is locked, its lock will be needed only if a split/merge propagates back to it

- **When won’t a split propagate back to a node?**
  - When the node’s child is *not full*

- **When won’t a merge propagate back to a node?**
  - When the node’s child is *more than half-empty*
Lock-Coupling: A Locking Strategy for Insertions/Deletions (Cont’d)

- A strategy, known as *lock-coupling*, for insertions/deletions can be pursued as follows:
  - Start at the root and go down, obtaining Shared locks as needed (an Exclusive lock is only obtained for the desired leaf node)
  - Once a child is locked, check if it is safe
  - If the child is safe, release all locks on ancestors

- A node is safe when changes will not propagate up beyond it
  - A safe node for an insertion is the one that is not full
  - A safe node for a deletion is the one that is more than half-empty
Lock-Coupling: An Example

- Insert data entry 45*: Obtain a Shared Lock

```
  20
 /       \
35      38 44
/      /   \
23   24   35 36
/  /  /  \\
20 22 23 24
```

```
20 22 23 24 35 36 38 41 44
```
Lock-Coupling: An Example

- Insert data entry 45*:

Obtain a Shared Lock
Lock-Coupling: An Example

- Insert data entry 45*: 
  - Release the Shared Lock Since the Child is Not Full
  - Obtain a Shared Lock
  - Obtain a Shared Lock
Lock-Coupling: An Example

- Insert data entry 45*:

- Obtain a Shared Lock
Lock-Coupling: An Example

- Insert data entry 45*:

Keep the Shared Lock
Since the Child is Full

Obtain a Shared Lock
Lock-Coupling: An Example

- Insert data entry 45*: 

- Obtain an Exclusive Lock
Lock-Coupling: An Example

- Insert data entry **45**:

```
<table>
<thead>
<tr>
<th>20</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>38</td>
</tr>
<tr>
<td>44</td>
<td>44</td>
</tr>
</tbody>
</table>
```

- Obtain an Exclusive Lock
- Release the Shared Lock
- Since the Child is **Not Full**
Lock-Coupling: An Example

- Insert data entry **45***:

![Diagram showing lock-coupling example](image-url)

- Release the Shared Lock
  Since the Child is **Not Locked**

- Obtain an Exclusive Lock
Lock-Coupling: An Example

- Insert data entry 45*: 
  
  ![Diagram showing the insertion of 45* and release of the lock]
Lock-Coupling: Another Example

- Insert data entry $25^*$:

![Diagram showing the process of inserting data entry 25* and obtaining a shared lock.]
Lock-Coupling: Another Example

- Insert data entry 25*:

  Obtain a Shared Lock
Lock-Coupling: Another Example

- Insert data entry 25*

  - Release the Shared Lock Since the Child is Not Full
  - Obtain a Shared Lock
Lock-Coupling: Another Example

- Insert data entry 25*: Obtain a Shared Lock

Diagram:

- Node 20
- Node 35
- Node 38
- Node 44
- Node 23
- Node 36
- Node 41
- Node 44*
Lock-Coupling: Another Example

- Insert data entry 25*:

  ![Diagram showing lock coupling process]

  - Obtain a Shared Lock
  - Release the Shared Lock Since the Child is Not Full
Lock-Coupling: Another Example

- Insert data entry 25*:

  ![Diagram of lock-coupling example]

  - Obtain an Exclusive Lock
Lock-Coupling: Another Example

- Insert data entry 25*: Request an Upgrade on the Lock
  Since the Child is Full

- Obtain an Exclusive Lock
Lock-Coupling: Another Example

- Insert data entry 25*:

What if another transaction has a Shared lock on this node and wants to access the locked child node?

Obtain an Exclusive Lock
Lock-Coupling: Another Example

- Insert data entry 25*:

A DEADLOCK Will Arise!
Lock-Coupling: Another Example

- Insert data entry **25***:

Otherwise...

- Insert 25*** and Propagate

- **20*** **22*** **23*** **24*** **35*** **36*** **38*** **41*** **44***
Next Class

Queries

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

Transaction Manager
Lock Manager

DB

Recovery Manager

Continue...