Today...

- **Last Two Sessions:**
  - DBMS Internals - Part XI
    - Transaction Management

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

- **Announcements:**
  - PS5 (the “last” assignment) is now posted. It is due on Thursday, April 23rd
  - The final exam is on Monday April 27th, from 8:30AM to 11:30AM in room 1190 (*all materials are included- open book, open notes*)
Outline

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

- Locking comes with delays mainly from *blocking*

- Usually, the first few transactions are unlikely to conflict
  - Throughput can rise in proportion to the number of active transactions

- As more transactions are executed concurrently, the likelihood of blocking increases
  - Throughput will increase more slowly with the number of active transactions

- There comes a point when adding another active transaction will actually decrease throughput
  - When the system *thrashes*!
If a database begins to *thrash*, the DBA should reduce the number of active transactions.

Empirically, thrashing is seen to occur when 30% of active transactions are blocked!
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 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 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*. 
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>W(B)</td>
</tr>
<tr>
<td>S(B)</td>
<td>X(C)</td>
<td>S(C)</td>
<td>R(C)</td>
</tr>
</tbody>
</table>

No cycles; hence, no deadlocks!

A schedule *without* 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></th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S(A)</td>
<td>R(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X(B)</td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X(B)</td>
<td>S(C)</td>
<td>R(C)</td>
</tr>
<tr>
<td></td>
<td>S(C)</td>
<td>R(C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(C)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X(B)</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></td>
<td></td>
</tr>
<tr>
<td>S(B)</td>
<td></td>
<td>X(B)</td>
<td>W(B)</td>
</tr>
<tr>
<td></td>
<td>S(C)</td>
<td>R(C)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>X(C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></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.

*Cycle detected; hence, a deadlock!*

The Corresponding **Waits-For Graph**

*A schedule with a deadlock*
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*

- 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 $T_1$ which *only* scans $S$ to find the oldest Sailor for specific rating levels
  - A transaction $T_2$ which updates Sailor while $T_1$ is running
A Possible Scenario

Assume a scenario whereby the actions of T1 and T2 are interleaved as follows:

- **T1** identifies all pages containing Sailors with rating 1 (say, pages A and B)
- **T1** finds the age of the oldest Sailor with rating 1 (say, 71)
- **T2** inserts a new Sailor with rating 1 and age 96 (perhaps into page C which does not contain any Sailor with rating 1)
- **T2** locates the page containing the oldest Sailor with rating 2 (say, page D) and deletes this Sailor (whose age is, say, 80)
- **T2** commits
- **T1** 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)
- **T1** commits
A Possible Scenario (*Cont’d*)

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

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

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

  A tuple with rating 1 and age 71 is returned

  A tuple with rating 2 and age 63 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)**

- One 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)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>R(D)</td>
<td></td>
</tr>
<tr>
<td>R(E)</td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td></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>

A tuple with rating 1 and age 71 is returned

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

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)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>R(D)</td>
<td></td>
</tr>
<tr>
<td>R(E)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td></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></td>
<td>E(C)</td>
</tr>
<tr>
<td></td>
<td>R(C)</td>
</tr>
<tr>
<td></td>
<td>W(C)</td>
</tr>
<tr>
<td></td>
<td>E(D)</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>

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 **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></td>
</tr>
<tr>
<td>R(B)</td>
<td></td>
</tr>
<tr>
<td>R(C)</td>
<td></td>
</tr>
<tr>
<td>W(C)</td>
<td></td>
</tr>
<tr>
<td>R(D)</td>
<td></td>
</tr>
<tr>
<td>W(D)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
<tr>
<td>R(D)</td>
<td></td>
</tr>
<tr>
<td>R(E)</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>

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

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

A tuple with rating 1 and age 71 is returned

A tuple with rating 2 and age 63 is returned

A tuple with rating 1 and age 96 is inserted

A tuple with rating 2 and age 80 is deleted

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, \textit{T1} should somehow ensure that no new pages are inserted to the Sailors relation
  - This has to do with the \textit{locking granularity}

- If there is an index on rating, \textit{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 \textit{index locking}
Outline

- Lock Conversions
- Dealing with Deadlocks
- Dynamic Databases and the Phantom Problem
- Concurrency Control in B+ Trees
Concurreny 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 leafs
  3. Insertions/deletions can cause splits/merges, which might propagate all the way up, from leafs 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

```
20
```

```
22
```

```
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*
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*
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*.

In the diagram:
- Obtain a Shared Lock at node 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*
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*
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*
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

  23

  38

  44

  20*

  22*

  23*

  24*

  35*

  36*

  38*

  41*

  44*
Lock-Coupling: An Example

- Insert data entry 45*: Obtain a Shared Lock

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

- Insert data entry 45*: 

  ![Diagram showing lock coupling example]

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

- Insert data entry 45*: 

Obtain a Shared Lock
Lock-Coupling: An Example

- Insert data entry **45***:

  ![Diagram showing lock-coupling example]

  - **Obtain a Shared Lock**
  - **Keep the Shared Lock** Since the Child is Full
  - **Obtain a Shared Lock**
Lock-Coupling: An Example

- Insert data entry 45*:

![Diagram showing lock coupling example with data entries and arrows indicating the coupling process.](attachment:lock-coupling-diagram.png)
Lock-Coupling: An Example

- Insert data entry 45*: 

  ![Diagram showing the process of inserting a data entry and releasing locks]

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

- Insert data entry 45*: 

  ![Diagram showing the process of Lock-Coupling]

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

- Insert data entry 45*: Insert 45* and Release the Lock
Lock-Coupling: Another Example

- Insert data entry 25*: 
  - Obtain a Shared Lock
Lock-Coupling: Another Example

- Insert data entry 25*:

```
<table>
<thead>
<tr>
<th>20</th>
<th>35</th>
</tr>
</thead>
</table>
```

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

20

35

23

38

44

20*

22*

23*

24*

35*

36*

38*

41*

44*
Lock-Coupling: Another Example

- Insert data entry 25*: 

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

- Insert data entry 25*:

  ![Diagram showing the process of inserting data entry 25* and obtaining a shared lock.]

  - Obtain a Shared Lock
  - Release the Shared Lock Since the Child is Not Full

- Data entries:
  - 20
  - 22
  - 23
  - 24
  - 35
  - 36
  - 38
  - 41
  - 44

- Child nodes:
  - 20*
  - 22*
  - 23*
  - 24*
  - 35*
  - 36*
  - 38*
  - 41*
  - 44*
Lock-Coupling: Another Example

- Insert data entry 25*: 

```
25
30
35
40
```

```
22
23
24
35
36
41
44
```

Obtain an Exclusive Lock
Lock-Coupling: Another Example

- Insert data entry 25*:

```
20
```

Request an Upgrade on the Lock
Since the Child is Full

```
23
```

Obtain an Exclusive Lock

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

\[ \text{A DEADLOCK WILL ARISE!} \]

- Obtain an Exclusive Lock
Lock-Coupling: Another Example

- Insert data entry 25*: 

  Otherwise...

  Insert 25* and Propagate

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

- There are several lock-based concurrency control schemes (e.g., 2PL & Strict 2PL)
  - The lock manager keeps track of the locks issued

- Deadlocks can arise, but they can either be detected and resolved, or initially prevented

- With dynamic databases, naïve locking strategies may expose the phantom problem
  - Resolving this problem has to do with the locking granularity
Summary

- *Index locking* is common, and affects performance significantly
  - Needed when accessing records via an index
  - Needed for *locking logical sets of records* (index locking/predicate locking)

- Tree-structured Indexes:
  - A straightforward use of 2PL is very inefficient
  - Bayer-Schkolnich illustrates a high potential for performance improvement
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