Databases

- **Data Models**
  - Conceptual representation of the data
- **Data Retrieval**
  - How to ask questions of the database
  - How to answer those questions
- **Data Storage**
  - How/where to store data, how to access it
- **Data Integrity**
  - Manage crashes, concurrency
  - Manage semantic inconsistencies
Q3
2 Points

For the index in Figure 14.3, list and count the number of comparisons that must be done to find the record with ID 98345.

Compare with 10101, 32343, and 76766 in the index, follow the middle pointer, and then do three more comparisons (76766, 83821, and 98345). So 6 in total.

Q4
2 Points

For the index in Figure 14.4, how many comparisons are needed to find the record with: "dept = finance" and ID = 76543. List the comparisons that will be made.

Compare four (Biology, CS, EE, Finance) in the index
Compare the two after going to the relation 12121 and 76543
For a total of 6 again
Q5
2 Points

Calculate the total size of the index (Figure 14.5) if: the number of blocks in the relation is 2,000,000, and each block in the inner or outer index can store 500 pointer entries. As an example, for the example discussed in Figures 14.6 and 14.7, the innermost index requires 10,000 blocks, and the outer index requires another 100 blocks, for a total of 10,100 index blocks.

2 mil / 500 = 4000 inner index blocks
8 outer index blocks

Q6
2 Points

Suppose you have to create a B+-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 characters) and the average name is itself large (say 10 characters). Explain how “prefix compression” can be used to maximize the average fanout of nonleaf nodes. Prefix compression is briefly discussed at the end of 14.4.3 (7th edition). Specifically discuss how many more pointers we may be able to fit into the nonleaf nodes (approximately).
Consider a similar exercise of counting the number of "comparisons" for a B+-tree as questions 3 and 4, but let's say that each interior and leaf node has exactly 400 search keys and pointers. Assume the height of the index is 4 (so 3 interior levels and one leaf level). Count the average number of comparisons for a search query (approximate is fine). Discuss how you can use "binary search" within each block to improve this number, and estimate how much improvement that would be.

(This "CPU" cost doesn't matter if the primary goal is to minimize how many blocks are moved from disk to memory -- however, with increasingly large amounts of memory available, CPU can often be the bottleneck and so the number of comparisons also matters).

w/ just linear search: 200*4
w/ binary search: \( \log_2(400) \times 4 = 9 \times 4 = 36 \)

8.1: Consider searching the above B+-tree Index for "Gretzky". After following Ptr 1 to Node 2, which is the next pointer that the search algorithm should follow?

8.2: Consider inserting a tuple with key "Glass". Which leaf node will it be inserted in?

8.3: Consider deleting the tuple with the key "Gold". After deleting the key from Node 7, how would Node 2 be affected?

8.4: Consider inserting a new key "Jones". This will result in a split of Node 7, and that will in turn result in a split of Node 2. What is the final set of keys in Node 1 (root)?
• 8.4: Consider inserting a new key "Jones". This will result in a split of Node 7, and that will in turn result in a split of Node 2. What is the final set of keys in Node 1 (root)?

• 9.1: Consider deleting the tuple with the key "60". After deleting the key from Node 8, how would Node 3 be affected (per the algorithm in the book)?

• 9.2: Consider inserting a new key "30". This will result in a split of Node 5. What is the final set of keys in Node 2?

• 9.3: Which of the following sequences of inserts will result in addition of a new entry to the root?
  * ( ) 49, 50, 51
  * (X) 30, 20, 21
  * ( ) 56, 57, 58, 59
  * ( ) 24, 30, 26, 40

  9.1: underfull, borrows 55, not at all
  9.2: 25, 30, 36
• 10.1: Consider deleting the tuple with the key "30". After deleting the key from Node 4, how would Node 1 be affected?

• 10.2: Consider deleting the tuple with the key "30", followed by the tuple with key "32". What would the Node 1 look like after those two? Pick the answer that fits the best.
  - leaves must have at least \( \lceil (n - 1)/2 \rceil \) values (keys),

• 10.3: Consider inserting a new key "22". What is the final set of keys in Node 1 (root)? As above, pick the answer that best fits

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td>lock-X(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock(A)</td>
<td>commit</td>
</tr>
<tr>
<td>&lt;fail&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10.1: not all
10.2: 17, 26 after we borrow 26 from node 3
10.3: 17, 23, 30

Back to locking: 2 Phase Locking

• Guarantees conflict-serializability
• Does not guarantee
  - recoverability
  - cascade-less schedules
2 Phase Locking

- How to guarantee recoverability:
  - If T2 performs a dirty read from T1, then:
    - T2 can’t commit until T1 either commits or aborts
      - If T1 commits, T2 can proceed with committing
      - If T1 aborts, T2 must abort
  - So … cascades still happen

Strict 2PL

- Release *exclusive* locks only at the very end, atomically with commit or abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B)</td>
<td>lock-X(A)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(A)</td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>unlock(A), unlock(B)</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>&lt;action aborts&gt;</td>
<td>&lt;action aborts&gt;</td>
<td>&lt;action aborts&gt;</td>
</tr>
</tbody>
</table>

Strict 2PL will not allow that
### Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B) commit</td>
<td>lock-X(A) read(A) write(A) unlock(A) commit</td>
<td>lock-S(A) read(A) commit</td>
</tr>
</tbody>
</table>

- Release *exclusive* locks only at the very end
  - Atomically with commit or abort
- *Guarantees recoverable and cascade-less schedules*

### Rigorous 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A) lock-S(B) read(B) unlock(B) ... write(A) unlock(A) commit</td>
<td>lock-X(B) write(B) unlock(B) commit</td>
</tr>
</tbody>
</table>

Serialization order?
- T1 -> T2

Commit order?
- T2 -> T1

Weird.
Rigorous 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(A)</td>
<td>lock-X(B)</td>
</tr>
<tr>
<td>lock-S(B)</td>
<td>write(B)</td>
</tr>
<tr>
<td>read(B)</td>
<td>unlock(B)</td>
</tr>
<tr>
<td>...</td>
<td>commit</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

- Also hold *shared* locks until the end
  - serialization order == the commit order
- *More intuitive for users*

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are ignored

Rigorous 2PL:

- Release both exclusive *and read* locks only at the very end
  - Makes serialization order == commit order
  - More intuitive behavior for the users
Lock Conversion/Upgrading

- Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
- Doesn’t change any of the other properties of the protocol

Recap so far…

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc
- Lock-based protocols
  - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
  - Locks acquired during *growing phase*, released during *shrinking phase*
- Strict 2PL, Rigorous 2PL
More Locking Issues: Deadlocks

No action proceeds

Deadlock:
- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

\[
\begin{array}{c|c}
T1 & T2 \\
\hline
\text{lock-X(B)} & \text{lock-S(A)} \\
\text{read(B)} & \text{read(A)} \\
B \leftarrow B-50 & \text{lock-S(B)} \\
\text{write(B)} & \\
\hline
\end{array}
\]

Rolling back transactions can be costly...

Deadlocks

- 2PL does not prevent deadlock
  - Strict doesn’t either

\[
\begin{array}{c|c}
T1 & T2 \\
\hline
\text{lock-X(B)} & \text{lock-S(A)} \\
\text{read(B)} & \text{read(A)} \\
B \leftarrow B-50 & \text{lock-S(B)} \\
\text{write(B)} & \\
\hline
\end{array}
\]

Rolling back transactions can be costly...
Preventing deadlocks

- **Graph-based protocols**
  - Acquire locks only in a well-known order

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock-X(B)</td>
<td>lock-S(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(A)</td>
</tr>
<tr>
<td>B ← B-50</td>
<td>lock-S(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
</tr>
<tr>
<td>lock-X(A)</td>
<td></td>
</tr>
</tbody>
</table>

- But might not know locks in advance

Detecting existing deadlocks

- **Timeouts** (local information)
- **cycles in waits-for graph** (global information):
  - edge $T_i \rightarrow T_j$ when $T_i$ waiting for $T_j$ on locks

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(V)</td>
<td>X(V)</td>
<td>X(Z)</td>
<td>X(W)</td>
</tr>
<tr>
<td>S(W)</td>
<td>S(V)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suppose T4 requests lock-S(Z)....
Dealing with Deadlocks

- Deadlock detected, now what?
  - Will need to abort some transaction

- Victim selection
  - Use time-stamps; say T1 is older than T2
  - *wait-die scheme*:
    - T1 will wait for T2 if T2 has a lock T1 needs.
    - T2 immediately aborts if needs a lock held by T1
  - *wound-wait scheme*:
    - T1 will *wound* T2 (force it to abort) if T2 has a lock that T2 needs.
    - T2 waits for T1 if it needs a lock held by T1.

- Issues
  - Prefer to prefer transactions with the most work done
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing

Locking granularity

- Locking granularity
  - What are we taking locks on? Tables, tuples, attributes?

- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency

- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table?
    - Prefer to lock the whole table in that case