Transactional Guarantees

- Concurrency Control
- Deadlocks
- Weakening Guarantees
- Recovery

XKCD and SQL injection attacks

Just for fun, not part of the course material, DB jokes in popular media are so rare.....
Q9:

What type of histogram is this?

- equi-depth
- equi-width
- neither

Q1.2
1 Point

What is the number of tuples resulting from a range query of 6 <= value < 21?

111

Explanation
35 + 26 + 50 = 111

Q1.3
1 Point

What about 9 <= value < 24?

106

Explanation
3/5 * 35 + 26 + 50 + 3/5 * 15 = 106

Q1.4
1 Point

What is the number of tuples resulting from a range query of 19 <= value <= 24?

25
Q4 External sorting
3 Points
Assume you need to sort 2000 tuples. 5 tuples fit into a block, and the system has 100 blocks of memory available. Count all reads and writes.

Q4.1
1 Point
How many runs are created?
4
Explanation
Runs are size of memory \((2000/5)/100 = 4\)

How many blocks is each?
100
Explanation
500

Q4.2
1 Point
How many seeks for run creation?
8
Explanation
4 runs, 2 seeks each

How many blocks transfers for run creation?
800
Explanation
Each block has to be read once and written once, so \(2 \times 400 = 800\)

Q4.3
1 Point
How many merge steps (phases) are needed?
1
Explanation
Just one step. Only 4 runs, can easily have the minimum of 1 output block, and 1 input block for each run.
External merge

2000 tuples, 50 tuples/block => 400 blocks  
M=100 means each run can be max 100 blocks => 4 runs

- runs 100 blocks each
- each input buffer 10 blocks ((100-60)/4)

10 blocks left for each of 4 runs
- each run will have to be filled 100/10=10 times
- 4 * 10 = 40 seeks for input to the runs
- 40 (for input) + 7 (for output) = 47

Q9 5

Q5

6 Points

Consider an equi-join on attribute B of relations R and S.  
Assume:

- \(b_r\) and \(b_s\) are 1000 and 2000, respectively
- we have an B+tree index on relation S on attribute B, of height 3.
- leaf nodes hold ptrs to 500 records
- B is a key in R, but not in S.
- 100 tuples in R each have 4 matches in S
- Each block of R or S holds 50 tuples.
- The below totals should reflect both identifying the matches, and returning the corresponding tuples.

Q5.1

1 Point

Compute blocks transferred assuming the index is a primary.

R blocks transferred:

| 1000 |

Explanation

Must read through entire relation R to find matches, no index.

Q5.2

1 Point

Compute blocks transferred assuming the index is a primary.

Index blocks:

| 15000 |

Explanation

1000 blocks * 50 tuples per block means 50,000 probes of S's index, each of which costs 3 block transfers.
Q5.3 1 Point
Compute blocks transferred assuming the index is a **primary**.

**S blocks transferred:**

 Enter the number of blocks transferred.

**Explanation**
Need to read 400 tuples from S, though this is a primary, the entire 400 are not consecutive. Instead, the four tuples that match each specific tuple of the 100 that have a match, are on the same page.

For example, the tuples in R that have a (actually 4) matches are tuples 100, 200, etc., the 4 tuples of S that match B=100 are located on a single page. The 4 matching B=200 are on a single (but different than the previous) page. The 4 matching B=300 are on yet another page. So there are 100 distinct pages of S that have matches.

Q5.4 1 Point
Compute blocks transferred assuming the index is a **secondary**.

**R blocks transferred:**

 Enter the number of blocks transferred.

**Explanation**
no change

Q5.5 1 Point
Compute blocks transferred assuming the index is a **secondary**.

**Index blocks transferred:**

 Enter the number of index blocks transferred.

**Explanation**
no change

Q5.6 1 Point
Compute blocks transferred assuming the index is a **secondary**.

**S blocks transferred:**

 Enter the number of blocks transferred.

**Explanation**
secondary means that each tuple assumed on distinct page, so 400 pages
Lock-based Protocols

- Transactions must *acquire* locks before using data
  - locking usually handled by transaction statements
- Two types:
  - *Shared* (S) locks (*read locks*)
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (*write locks*)
    - Obtained for updating a data item

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- Assume T1 requests lock held by T2:

<table>
<thead>
<tr>
<th>Held lock</th>
<th>Lock wanted</th>
<th>Allow?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Shared</td>
<td>YES</td>
</tr>
<tr>
<td>Shared</td>
<td>Exclusive</td>
<td>NO</td>
</tr>
<tr>
<td>Exclusive</td>
<td>-</td>
<td>NO</td>
</tr>
</tbody>
</table>

- If *compatible*, grant the lock, otherwise T1 waits in a *queue*.
Lock instructions

Potential schedule

T1 T2

lock-X(B)
read(B)
B ← B-50
write(B)
unlock(B)

lock-S(A)
read(A)
A ← A + 50
write(A)
unlock(A)

lock-S(B)
read(B)
unlock(B)
display(A+B)

T1 T2

lock-S(A)
read(A)
unlock(A)

Good!

lock-S(B)
read(B)
unlock(B)
display(A+B)

lock-X(B)
read(B)
B ← B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)

Good!
**Lock instructions**

*Potential schedule*

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
</table>
| lock-X(B)  
read(B)  
B $\leftarrow$ B-50  
write(B)  
unlock(B) | lock-S(A)  
read(A)  
unlock(A) |
| lock-X(A)  
read(A)  
A $\leftarrow$ A + 50  
write(A)  
unlock(A) | lock-S(B)  
read(B)  
unlock(B)  
display(A+B) |

*Not good!*

---

**2-Phase Locking Protocol (2PL)**

- **Phase 1: Growing phase**
  - Transaction may obtain locks
  - But may not release them

- **Phase 2: Shrinking phase**
  - Only release locks

- **2PL guarantees conflict-serializability**
  - *lock-point*: the time at which a transaction acquired last lock
  - if *lock-point* ($T_1$) < *lock-point* ($T_2$), there can’t be an edge from $T_2$ to $T_1$ in the precedence graph

T1

| lock-X(B)  
read(B)  
B $\leftarrow$ B-50  
write(B)  
unlock(B) | lock-X(A)  
read(A)  
A $\leftarrow$ A + 50  
write(A)  
unlock(A) |

*not allowed*
Lockpoints Intuition
(pseudo-proof by contradiction)

T1
lock(x)
write(x)
unlock(x)

T2
lock(x)
read(x)

T1’s lockpoint must be before T2’s, because T1 already in shrinking phase, T2 still growing phase

But if we also have an edge from T2 to T1: T2’s lockpoint must also be before T1’s. Contradiction!
Cycle requires edge and it’s reverse, but we just showed this can’t happen, so… conflict-serializable.

Back to locking: 2 Phase Locking

- Guarantees conflict-serializability
- Does not guarantee
  - recoverability
  - cascade-less schedules

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

<action aborts>
### Strict 2PL

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

<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>commit</td>
<td><strong>commit</strong></td>
<td><strong>commit</strong></td>
</tr>
</tbody>
</table>

### Rigorous 2PL

- **Beginning timestamp 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>write(A)</td>
<td>commit</td>
</tr>
<tr>
<td>unlock(A), unlock(B)</td>
<td></td>
</tr>
</tbody>
</table>

### Beginning timestamp order?
- $T_1 \rightarrow T_2$

### Commit order?
- $T_2 \rightarrow T_1$

Weird.

- 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 transaction proceeds

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

Rolling back transactions can be costly...

Deadlocks

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

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>
</tbody>
</table>

- But might not know locks in advance

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

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(W)</td>
<td>S(V)</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 (not always done)

- 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
Granularity Hierarchy

The highest level in the example hierarchy is the entire database. The levels below are of relation and tuple in that order. Can lock at any level in the hierarchy.