Goals for Today

- Understand how to manage locks and deadlock when implementing 2PL or Strict 2PL
- Reason about the granularity of objects locked by transactions, the implications of new data on object locking
- Discuss Isolation levels in a DBMS

Basic Locking: Example (Take 2-abort)

\[
\begin{align*}
A &= 1000, B = 2000, Output = ? \\
\end{align*}
\]

Two-Phase Locking (2PL)

1) Protocol: each transaction must
   - Get a S (shared) or an X (exclusive) lock on object before reading
   - Get an X (exclusive) lock on object before writing

   Can upgrade a Shared lock to an Exclusive lock! (when okay?)

2) All lock requests precede all unlock requests!
   I.e., a transaction can not request additional locks once it releases any locks

\[
\begin{align*}
\text{Growing Phase} & \quad \text{Shrinking Phase} \\
\text{Time} & \quad 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ 20 \\
\end{align*}
\]

T1

- Lock_X(A) <granted>
- Read(A)
- A = A - 50
- Write(A)
- Lock_X(B) <granted>
- Unlock(A)
- Unlock(B)
- Read(B)
- B = B + 50
- Write(B)
- Unlock(B)
- PRINT(A+B)

T2

- Lock_S(A)
- Lock_S(B)
- Read(A)
- Read(B)
- Unlock(A)
- Unlock(B)
- ABORT!!

T2 has read uncommitted changes! It must also abort.
Avoiding Cascading Aborts: Strict 2PL

- Problem with 2PL: cascading aborts
- Another example: rollback of T1 requires rollback of T2

| T1: R(A), W(A), R(B), W(B), Abort |
| T2: R(A), W(A) |

- Solution: **Strict Two-phase Locking (Strict 2PL):**
  - Same as 2PL, except:
  - All locks held by a transaction are released only when the transaction completes

**Consequence:** a writer will block all other readers until the writer commits or aborts

---

View 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 T2 also writes final value of A in S2

| T1: R(A) | W(A)       |
| T2: W(A) |

---

Lock Management

- Lock/unlock requests are handled by the **Lock Manager**
  - Have table with entry for each currently held lock
- What object is being locked?
  - Possibilities: table(s), row(s), page(s)...
  - Too coarse-grained limits concurrency!
  - More on granularity later...

- **Lock table entry**
  - Object id of object being locked (e.g., table, row, page)
  - (Pointer to) list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - (Pointer to) queue of lock requests

---

Lock Management (cntd)

- When a lock request arrives
  - Check if any xact currently holds a conflicting lock on the object
  - If not, create an entry and grant the lock
  - Else, put the requesting xact on the wait queue

*Locking and unlocking have to be atomic operations!*

- Try Exercise 3
Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.

- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Deadlock Prevention

- Assign priorities based on timestamps

- Suppose Ti wants a lock that Tj holds
  Two possible policies:
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts (gets “wounded”); otherwise Ti waits

  In both, the older never aborts

- If a transaction re-starts, make sure it gets its original timestamp

  Why?

Deadlock Detection

- Alternative is to allow deadlocks to happen but to check for them and fix them if found.

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock

- Periodically check for cycles in the waits-for graph

- If cycle detected – find a transaction whose removal will break the cycle and kill it
Deadlock Detection (Cntd)

Example:

T1: \( S(A), S(D), S(B) \)
T2: \( X(B) \)
T3: \( S(D), S(C), X(A) \)
T4: \( X(B) \)

“Dynamic” Databases

- Relax assumption that database is a static set of objects
- With Insert and Delete possible, even Strict 2PL (on individual objects) will not assure serializability

The “Phantom” Problem – Example 1

- Consider T1 – “Find oldest sailor”
  T1 locks all Sailor records, finds oldest sailor \((age = 71)\)
  T2 inserts a new sailor; \((age = 96)\)
  T1 checks for the oldest sailor, finds oldest sailor \((age = 96)\)

- The sailor with age 96 is a “phantom tuple” from T1’s point of view --- first it’s not there, then it is

The “Phantom” Problem – Example 2

- Consider T3 – “Find oldest sailor for each rating”
  T3 locks all pages containing sailor records with \(rating = 1\)
  finds oldest sailor \((age = 71)\)
  T4 inserts a new sailor; \(rating = 1, age = 96\)
  T4 also deletes oldest sailor with \(rating = 2, age = 80\)
  commits

  T3 now locks all pages containing sailor records with \(rating = 2\), and finds oldest \((age = 63)\).

- T3 saw only part of T4’s effects!

No serial execution where T1’s result could happen!
No serial execution where T3’s result could happen!
The Problem

• How do you lock something that does not yet exist??

• T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.
  – Assumption only holds if no sailor records are added while they are executing!
  – Need some mechanism to enforce this assumption.

• Conflict serializability on reads and writes of individual objects guarantees serializability only if the set of objects is fixed

Solution: Index Key Value Locking

• If index on the rating field using Alternative (2)
  – T3 should lock the index page containing the data entries with rating = 1.
  – If there are no records with rating = 1, T3 must lock the index page where such a data entry would be, if it existed!

• If there is no suitable index
  – T3 must lock all pages,
  – and lock the file/table to prevent new pages from being added, to ensure that no records with rating = 1 are added or deleted.

Note: Index locking is a special case of predicate locking, which is difficult to implement in general

Isolation Levels in SQL Standard

• Idea: give users control over locking overhead incurred by their xacts
• Xacts can be specified with desired Isolation Level
  – Also, access mode like “read-only” only gets S locks

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- Does not get read locks, (not allowed to write objects)
- Write locks held to commit. Get read locks, but release those right away
- Strict 2PL. Locks before read & write, on individual objects
- Strict 2PL. Gets locks before read/write, including on sets of objects (index locks)