Announcement

- Project milestone #2 due today (April 14)
- Homework #4 due in 9 days (April 23)
- Recitation session this Friday (April 18)
  - Homework #4 Q&A
- Project demo period in two weeks (April 28-May 3)
  - Sign-up sheet will be available next Monday
- Final exam in 17 days (May 1, 2-5pm)

Review

- ACID
  - Atomicity: TX's are either completely done or not done at all
  - Consistency: TX's should leave the database in a consistent state
  - Isolation: TX's must behave as if they are executed in isolation
  - Durability: Effects of committed TX's are resilient against failures
- SQL transactions
  - Begins implicitly
  - SELECT ..;
  - UPDATE ..;
  - ROLLBACK | COMMIT;
Concurrency control

Goal: ensure the "I" (isolation) in ACID

\[
\begin{align*}
T_1: & \text{read}(A); & \text{write}(A); & \text{commit;} \\
T_2: & \text{read}(A); & \text{write}(A); & \text{commit;}
\end{align*}
\]

Good versus bad schedules

<table>
<thead>
<tr>
<th>Good!</th>
<th>Bad!</th>
<th>Good?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>T_2</td>
<td>T_1</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>Read 400</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>w(B)</td>
<td>400 - 100</td>
<td>400 - 50</td>
</tr>
<tr>
<td>r(A)</td>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(A)</td>
<td>r(C)</td>
<td>r(C)</td>
</tr>
<tr>
<td>r(C)</td>
<td>w(B)</td>
<td>w(B)</td>
</tr>
<tr>
<td>w(C)</td>
<td>w(C)</td>
<td>w(C)</td>
</tr>
</tbody>
</table>

Serial schedule

- Execute transactions in order, with no interleaving of operations
  - \( T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C) \)
  - \( T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B) \)

- Isolation achieved by definition!

- Problem: no concurrency at all
- Question: how to reorder operations to allow more concurrency
Conflicting operations

- Two operations on the same data item conflict if at least one of the operations is a write:
  - \( r(X) \) and \( w(X) \) conflict
  - \( w(X) \) and \( r(X) \) conflict
  - \( w(X) \) and \( w(X) \) conflict
  - \( r(X) \) and \( r(X) \) do not conflict
  - \( r/w(X) \) and \( r/w(Y) \) do not conflict

- Order of conflicting operations matters:
  - E.g., if \( T_1.r(A) \) precedes \( T_2.w(A) \), then conceptually, \( T_1 \) should precede \( T_2 \).

Precedence graph

- A node for each transaction.
- A directed edge from \( T_i \) to \( T_j \) if an operation of \( T_i \) precedes and conflicts with an operation of \( T_j \) in the schedule.

Conflict-serializable schedule

- A schedule is conflict-serializable iff its precedence graph has no cycles.
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is "good"):
  - In that serial schedule, transactions are executed in the topological order of the precedence graph.
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions.
**Locking**

*Rules*
- If a transaction wants to read an object, it must first request a shared lock (S mode) on that object.
- If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object.
- Allow one exclusive lock, or multiple shared locks.

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of the lock requested</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
<td>Yes</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>No</td>
</tr>
<tr>
<td>X</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Basic locking is not enough**

<table>
<thead>
<tr>
<th>Possible schedule under locking</th>
<th>But still not conflict-serializable!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add 1 to both A and B (preserve (A = B))</td>
<td>(A \neq B)!</td>
</tr>
<tr>
<td>Lock-X(A)</td>
<td>Read 101</td>
</tr>
<tr>
<td>Write 100+1</td>
<td>unlock(A)</td>
</tr>
<tr>
<td>Lock-X(B)</td>
<td>Read 100</td>
</tr>
<tr>
<td>Write 200+1</td>
<td>unlock(B)</td>
</tr>
</tbody>
</table>

Two-phase locking (2PL)

*All lock requests precede all unlock requests*
- Phase 1: obtain locks, phase 2: release locks

<table>
<thead>
<tr>
<th>(T_1)</th>
<th>(T_2)</th>
<th>(T_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-X(A)</td>
<td>Read 101</td>
<td>Write 101 * 2</td>
</tr>
<tr>
<td>Write 100+1</td>
<td>unlock(A)</td>
<td>lock-X(A)</td>
</tr>
<tr>
<td>Lock-X(B)</td>
<td>Read 100</td>
<td>Write 100 * 2</td>
</tr>
<tr>
<td>Write 200+1</td>
<td>unlock(B)</td>
<td>lock-X(B)</td>
</tr>
</tbody>
</table>

2PL guarantees a conflict-serializable schedule.
Problem of 2PL

- $T_2$ has read uncommitted data written by $T_1$
- If $T_1$ aborts, then $T_2$ must abort as well
- Cascading aborts possible if other transactions have read data written by $T_2$
- Even worse, what if $T_2$ commits before $T_1$?
  - Schedule is not recoverable if the system crashes right after $T_2$ commits

Strict 2PL

- Only release locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts
- Used in most commercial DBMS (except Oracle)

Deadlocks

Deadlock: cycle in the wait-for graph

- $T_1$ is waiting for $T_2$
- $T_2$ is waiting for $T_1$
- Deadlock!
Dealing with deadlocks

- Impose an order for locking objects
  - Must know in advance which objects a transaction will access
- Timeout
  - If a transaction has been blocked for too long, just abort
- Prevention
  - Idea: abort more often, so blocking is less likely
  - Suppose T is waiting for T'
    - Wait/die scheme: Abort T if it has a lower priority; otherwise T waits
    - Wound/wait scheme: Abort T' if it has a lower priority; otherwise T waits
- Detection using wait-for graph
  - Idea: deadlock is rare, so only deal with it when it becomes an issue
  - When do we detect deadlocks?
  - Which transactions do we abort in case of deadlock?

Implementation of locking

- Do not rely on transactions themselves to lock/unlock explicitly
- DBMS inserts lock/unlock requests automatically

```
Transactions
Streams of operations

Lock table
Lock info for each object, including locks currently held and the request queue

Scheduler

Insert lock/unlock requests
Operations with lock/unlock requests

Serialized schedule with no lock/unlock operations
```

Multiple-granularity locks

- Hard to decide what granularity to lock
  - Trade-off between overhead and concurrency
- Granularities form a hierarchy
- Allow transactions to lock at different granularity, using intention locks
  - S, X: lock the entire subtree in S, X mode, respectively
  - IS: intend to lock some descendant in S mode
  - IX: intend to lock some descendant in X mode
  - SIX (= S + IX): lock the entire subtree in S mode; intend to lock descendant in X mode

```
Database
Tables
Pages
Rows
```
Multiple-granularity locking protocol

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of the lock requested</th>
<th>Grant the lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>S</td>
<td>Y</td>
</tr>
<tr>
<td>IS</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IX</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SIX</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- **Lock:** before locking an item, T must acquire intention locks on all ancestors of the item
  - To get S or IS, must hold IS or IX on parent
  - What if T holds S or SIX on parent?
  - To get X or IX or SIX, must hold IX or SIX on parent
- **Unlock:** release locks bottom-up
- 2PL must also be observed

### Examples
- **T1** scans R and update a few rows
  - T1 gets an SIX lock on R, then repeatedly gets an S lock on rows, and occasionally upgrade to X for some rows
- **T2** uses an index to read only part of R
  - T2 gets an IS lock on R, and then repeatedly gets an S lock on rows it needs to access
- **T3** reads all of R
  - T3 gets an S lock on R

### Phantom problem revisited
- **Reads** are repeatable, but may see phantoms
- Example: different average
  - T1:
    ```sql
    SELECT AVG(GPA)
    FROM Student WHERE age = 10;
    INSERT INTO Student
    VALUES(789, 'Nelson', 10, 1.0);
    COMMIT;
    SELECT AVG(GPA)
    FROM Student WHERE age = 10;
    COMMIT;
    ```
  - T2:
    ```sql
    SELECT AVG(GPA)
    FROM Student WHERE age = 10;
    COMMIT;
    ```

- How do you lock something that does not exist yet?
Solutions

- **Index locking**
  - Use the index on `Student(age)`
  - `T_2` locks the index block(s) with entries for `age = 10`
    - If there are no entries for `age = 10`, `T_2` must lock the index block where such entries _would be_, if they existed!

- **Predicate locking**
  - “Lock” the predicate `age = 10`
  - Reason with predicates to detect conflicts
  - Expensive to implement

Concurrency control without locking

- **Optimistic (validation-based)**

- **Timestamp-based**

- **Multi-version (Oracle, PostgreSQL)**

Optimistic concurrency control

- **Locking is pessimistic**
  - Use blocking to avoid conflicts
  - Overhead of locking even if contention is low

- **Optimistic concurrency control**
  - Assume that most transactions do not conflict
  - Let them execute as much as possible
  - If it turns out that they conflict, abort and restart
Sketch of protocol

- Read phase: transaction executes, reads from the database, and writes to a private space
- Validate phase: DBMS checks for conflicts with other transactions; if conflict is possible, abort and restart
  - Requires maintaining a list of objects read and written by each transaction
- Write phase: copy changes in the private space to the database

Pessimistic versus optimistic

- Overhead of locking versus overhead of validation and copying private space
- Blocking versus aborts and restarts
  - Locking has better throughput for environments with medium-to-high contention
  - Optimistic concurrency control is better when resource utilization is low enough

Timestamp-based

- Assign a timestamp to each transaction
  - Timestamp order is commit order
- Associate each database object with a read timestamp and a write timestamp
- When transaction reads/writes an object, check the object’s timestamp for conflict with a younger transaction; if so, abort and restart
- Problems
  - Even reads require writes (of object timestamps)
  - Ensuring recoverability is hard (plenty of dirty reads)
Multi-version concurrency control

- Maintain versions for each database object
  - Each write creates a new version
  - Each read is directed to an appropriate version
  - Conflicts are detected in a similar manner as timestamp concurrency control
- In addition to the problems inherited from timestamp concurrency control
  - Pro: Reads are never blocked
  - Con: Multiple versions need to be maintained
- Oracle and PostgreSQL use variants of this scheme

Summary

- Covered
  - Conflict-serializability
  - 2PL, strict 2PL
  - Deadlocks
  - Multiple-granularity locking
  - Predicate locking and tree locking
  - Overview of other concurrency-control methods
- Not covered
  - View-serializability
  - Concurrency control for search trees (not the same as multiple-granularity locking and tree locking)