Announcements (April 26)

- Homework #4 due this Thursday (April 28)
  - Sample solution will be available on Thursday
- Project demo period: April 28 – May 1
  - Remember to email me to sign up for a 30-minute slot
- Final exam on Monday, May 2, 2-5pm
  - 3 hours—no time pressure!
  - Open book, open notes
  - Comprehensive, but with emphasis on the second half of the course and materials exercised in homework
- Sample final (from last year) available
  - Solution will be available on Thursday

Transactions

- Transaction: a sequence of operations with ACID properties
  - 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

```
T_1: read(A); write(A); read(B); write(B); commit;
T_2: read(A); write(A); read(C); write(C); commit;
```

Good versus bad schedules

<table>
<thead>
<tr>
<th>T_1</th>
<th>T_2</th>
<th>T_1</th>
<th>T_2</th>
<th>T_1</th>
<th>T_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
<td>w(B)</td>
</tr>
<tr>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
<td>r(C)</td>
</tr>
<tr>
<td>w(C)</td>
<td>w(C)</td>
<td>w(C)</td>
<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
  - \( r/w(X) \) and \( r/w(Y) \) do not

- 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.

### Compatibility Matrix

<table>
<thead>
<tr>
<th>Mode of lock(s) currently held by other transactions</th>
<th>Mode of 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

Add 1 to both $A$ and $B$ (preserve $A = B$)
- Read 100
- Write 100 + 1
- $T_1$ locks $X(A)$
- $T_1$ reads $A$
- $T_1$ writes $A$
- $T_1$ unlocks $A$

Multiply both $A$ and $B$ by 2 (preserves $A = B$)
- Read 101
- Write 101 + 2
- $T_2$ locks $X(A)$
- $T_2$ reads $A$
- $T_2$ writes $A$
- $T_2$ unlocks $A$

Possible schedule under locking:
- $T_1$ locks $X(B)$
- $T_1$ reads $B$
- $T_1$ writes $B$
- $T_1$ unlocks $B$

But still not conflict-serializable:
- $T_2$ locks $X(B)$
- $T_2$ reads $B$
- $T_2$ writes $B$
- $T_2$ unlocks $B$

Two-phase locking (2PL)

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

$T_1$ locks $X(A)$
- Read $A$
- Write $A$
- $T_1$ unlocks $A$

2PL guarantees a conflict-serializable schedule:
- $T_2$ locks $X(A)$
- Read $A$
- Write $A$
- $T_2$ unlocks $A$

 cannot obtain the lock on $B$ until $T_2$ unlocks.
Problem of 2PL

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(A)</td>
<td>r(A)</td>
</tr>
<tr>
<td>w(A)</td>
<td>w(A)</td>
</tr>
<tr>
<td>r(B)</td>
<td>r(B)</td>
</tr>
<tr>
<td>w(B)</td>
<td>w(B)</td>
</tr>
</tbody>
</table>

- $T_2$ 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

$T_1$ 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

![Diagram]

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 descendent in S mode
  - IX: intend to lock some descendent in X mode
  - SIX (= S + IX): lock the entire subtree in S mode; intend to lock descendent in X mode
Multiple-granularity locking protocol

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

- $T_1$ reads all of $R$
  - $T_1$ gets an S lock on $R$
- $T_2$ scans $R$ and updates a few rows
  - $T_2$ gets an SIX lock on $R$, and then occasionally gets an X lock for some rows
- $T_3$ uses an index to read only part of $R$
  - $T_3$ gets an IS lock on $R$, and then repeatedly gets an S lock on rows it needs to access

Phantom problem revisited

- Lock everything read by a transaction $\rightarrow$ reads are repeatable, but may see phantoms
- Example: different average
  - $T_1$: `SELECT AVG(GPA) FROM Student WHERE age = 10;`
  - $T_2$: `INSERT INTO Student VALUES(789, 'Nelson', 10, 1.0);`
  - `COMMIT;`
  - $T_3$: `SELECT AVG(GPA) FROM Student WHERE age = 10;`
  - $T_2$: `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)