Concurrent control

- **Concurrency control** is the process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing the database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for concurrency control

- Three examples of potential problems caused by concurrency:
  - Lost updates
  - Uncommitted data
  - Inconsistent retrievals
Lost update problem

- Occurs when a successfully completed update is overwritten by another transaction
- Example:
  - \( T_1 \) withdraws $10 from an account with initial \( \text{bal}_o = $100 \)
  - \( T_2 \) deposits $100 into same account

The final balance should be $190, but incorrectly overlapping transactions may lead to a final balance of only $90

Lost update problem (illustr.)

<table>
<thead>
<tr>
<th>Time</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
<th>( \text{bal}_o )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_0 )</td>
<td>begin transaction</td>
<td>begin transaction</td>
<td>100</td>
</tr>
<tr>
<td>( t_1 )</td>
<td>read(( \text{bal}_o ) )</td>
<td>read(( \text{bal}_o ) )</td>
<td>100</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>( \text{bal}_o ) = 100</td>
<td>( \text{bal}_o ) = 108</td>
<td>100</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>write(( \text{bal}_o ) )</td>
<td>write(( \text{bal}_o ) )</td>
<td>200</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>commit(( \text{bal}_o ) )</td>
<td>commit(( \text{bal}_o ) )</td>
<td>80</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>commit</td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>

- Result: \( T_2 \)'s update is lost

Uncommitted data problem

- Occurs when one transaction accesses the intermediate results of another transaction before they are committed – and the second transaction is then rolled back.
- Example:
  - \( T_4 \) updates \( \text{bal}_o \) to $200, but there is an error in a later UPDATE, so \( \text{bal}_o \) is rolled back to original value of $100
  - After the \( \text{bal}_o \) update, but before the rollback, \( T_3 \) reads the new value of \( \text{bal}_o \) ($200) and uses this as the basis of a $10 reduction, giving a final new balance of $190, not $90
Uncommitted data problem (illustr.)

<table>
<thead>
<tr>
<th>Time</th>
<th>T₁</th>
<th>T₂</th>
<th>bal₁</th>
<th>bal₂</th>
<th>bal₃</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>begin_transaction</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₂</td>
<td>read(bal₁)</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₃</td>
<td>begin_transaction</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₄</td>
<td>read(bal₂)</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₅</td>
<td>bal₂ = bal₂ + 10</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t₆</td>
<td>commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>210</td>
</tr>
</tbody>
</table>

Inconsistent retrievals problem

- Occurs when a transaction reads several values, but a different transaction updates some of them in the midst of this process
  - Some data are read before they are changed and others after they are changed — yielding inconsistent results
- Example:
  - T₆ is totaling balances of account x ($100), account y ($50), and account z ($25)
  - Meanwhile, T₅ transfers $10 from bal₁ to bal₃ after x is totaled but before z is, so T₆ has a result that’s $10 too high

Inconsistent retrievals problem (illustr.)
Scheduler

- A **serializable schedule** is a schedule of a transaction's operations in which the interleaved execution of all active transactions yields the same results as if those transactions were executed in serial order.
  - Waiting for one transaction to finish before starting any others may be inefficient.
  - The built-in scheduler ensures efficient use of the DBMS and CPU by interleaving operations when possible.
    - If transactions access unrelated data, then there is no conflict among individual operations between transactions.

Scheduling

- Methods for scheduling conflicting operations in concurrent transactions:
  - Locking methods
  - Timestamping

Lock terminology

- **Lock granularity** refers to the size of the locked resource:
  - database-level…table-level…page-level…row-level
  - slow data access → significant overhead
- An **exclusive lock** prohibits other users from reading the locked resource.
- A **shared lock** allows other users to read the locked resource, but they cannot update it.
Optimistic vs. Pessimistic locking

- **Optimistic locking** assumes that no transaction conflict(s) will occur:
  - DBMS processes a transaction to a temporary file; checks whether conflict occurred:
    - If not, the transaction is finished
    - If so, the transaction is repeated until there is no conflict
- **Pessimistic locking** assumes that conflict(s) will occur:
  - Locks are issued before a transaction is processed, and then the locks are released
- Optimistic locking is acceptable for applications with few update operations

Two-phase locking

- **Two-phase locking** (2PL) guarantees serializability:
  - one of the most common techniques used to achieve this
  - Transactions are allowed to obtain as many locks as necessary (growing phase)
  - Once the first lock is released (shrinking phase), no additional locks can be obtained
  - Two-phase locking doesn't prevent deadlocks

Deadlock

- An impasse that may result when two (or more) transactions are waiting for locks held by the other to be released
Controlling deadlocks

- Three basic techniques:
  - **Deadlock prevention**
    - Abort a transaction if possibility of deadlock
    - Reschedule transaction for later execution
  - **Deadlock detection**
    - DBMS periodically tests database for deadlocks
    - If found, one transaction ("victim") is rolled back
  - **Deadlock avoidance**
    - Transactions obtain all needed locks before execution

Timestamping

- **Timestamp**: A unique identifier created by DBMS that indicates the relative starting time of a transaction
- Transactions ordered globally so that older transactions (transactions with smaller timestamps) get priority in the event of conflict
- Conflict is resolved by rolling back and restarting the associated transaction - no locks, so no deadlocks
- Demands a lot of system resources (both memory and processing overhead)