Transactional Guarantees

- Concurrency Control
- Deadlocks
- Weakening Guarantees
- Recovery

Q10 - 1

**Explanation**

Not conflict serializable because accesses to A require edge from T1 to T2, while accesses to B require edge from T2 to T1. Hence, cycle in precedence graph, not conflict serializable.

Not view serializable because T1 has initial read of A, T2 has initial read of B, no view serial schedule could have both of those.

However, the answer IS guaranteed to be the same as either T1,T2 or T2,T1 because the reads and writes do not clobber each other, and the updates are just constants, meaning their order does not matter. Hence, serializable.

Choose the most specific answer below:
- serializable
- conflict serializable
- view serializable
- not serializable
Q10 - 2

**Explanation**

Precedence graph would only have edge from T1 to T2.
No cycle, so conflict serializable. This is most specific as CS is subset of VS which is subset of S.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A := 100</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>A := 10</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B := 100</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B := 10</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Choose the most specific answer below:
- Serializable
- Conflict serializable
- View serializable

Q10 - 3

**Explanation**

Not CS (cycle in graph). Not VS because T2 has both last writes AND initial read of B (no serial sched would have both).
Could show not serializable by plugging in numbers and showing that equiv to neither T1,T2 or T2,T1.
Easier to note that T1's increment of B by 100 is overwritten by T2 (which reads B first and writes last, so T1's add to B is ignored. No serial schedule would do this.

**Edge in precedence graph if:**
1. $T_i$ executes $write(Q)$ before $T_j$ executes $read(Q)$.
2. $T_i$ executes $read(Q)$ before $T_j$ executes $write(Q)$.
3. $T_i$ executes $write(Q)$ before $T_j$ executes $write(Q)$.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
</tr>
<tr>
<td>A := 100</td>
<td></td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td>tmp := 0.1*B</td>
</tr>
<tr>
<td></td>
<td>B := tmp</td>
</tr>
<tr>
<td></td>
<td>read(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B := 100</td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write(B)</td>
</tr>
<tr>
<td></td>
<td>read(A)</td>
</tr>
<tr>
<td></td>
<td>A := tmp</td>
</tr>
<tr>
<td></td>
<td>write(A)</td>
</tr>
</tbody>
</table>

Choose the most specific answer below:
- Serializable
- Conflict serializable
- View serializable
- Not serializable
Q10 - 4

Assume that each write outputs a unique value computed using all prior reads by the same transactions.

Choose the most specific answer below:
- serializable
- conflict serializable
- view serializable
- not serializable

Explanation
A better way to put this might be that the "uniqueness" requirement implies that all reads from relationships must be the same as in any equivalent serial schedule. Otherwise, written values would change and consistency would presumably be violated.
T2's read of B is before T3's write, so T2 < T3 in the equivalent serial schedule. However, T2's write of A is after T3's read of A, meaning T3 < T2 in the equivalent serial schedule. Both of these cannot be true at the same time, so there can be no equivalent serial schedule.

Q10 - 5

The following schedule is conflict-serializable. List out at least four equivalent serial schedules for this schedule. You can do this by drawing the precedence graph, and then finding sequences of transactions where all the edges go from left to right. As an example: T1, T2, T3, T4, T5 is clearly not an equivalent serial schedule because there is an edge from T3 to T1 (due to A).

Explanation
Edges:
- 2 → 1
- 3 → 2
- 5 → 2
- 5 → 4
- 2 → 4
- 3,2,1 must be in that order
- 5,2,4 must be in that order

53214
53241
35214
35241
might be others?
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.

Intention Locks

- New lock mode, called intention locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: intention shared
    - The lower levels below may be locked in the shared mode
  - IX: intention exclusive
  - SIX: shared and intention-exclusive
    - The entire subtree is locked in the shared mode, but might also want exclusive locks on some nodes below
- Protocol:
  - Before acquiring a lock on a data item, all the ancestors must be locked as well, at least in the intention mode
  - Lock acquisition order is from the root down to the desired node.
Intention Locks

(1) Want to lock \(t1\) in shared mode, \(DB\) and then \(R1\) must be locked in at least IS mode (but IX, SIX, S, X are okay too), then \(t1\) in S mode.
(2) Want to lock \(t4\) in exclusive mode, \(DB\) and then \(R2\) must be locked in at least IX mode (SIX, X are okay too), then \(t4\) must be locked in X mode.

Compatibility Matrix with Intention Lock Modes

- Locks from different transactions:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>IX</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SIX</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>X</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>
Example

- Assume:
  - $T_1$ wants *shared* lock on $t_2$
  - $T_2$ wants *exclusive* lock on $t_4$

Can $T_2$ access object $t_{2.2}$ in *X* mode?

What locks will $T_2$ get?
other
Concurrency Control Schemes

Snapshot Isolation

• Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  • Several others support this in addition to locking-based protocol

• A type of optimistic concurrency control

• Key idea:
  • For each object, maintain past “versions” of the data along with timestamps
    • Every update to an object causes a new version to be generated
Snapshot Isolation

- **Read queries:**
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
  - Never aborted

- **Update queries (transactions):**
  - Reads processed as above on a snapshot
  - Writes are done in private storage. However, *the writes are visible to the transaction that made them.*
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

---

**Snapshot Isolation**

- **A transaction T1 executing with Snapshot Isolation**
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits

**First-committer-wins rule:**
- Commits only if no other concurrent transaction has already written data that T1 intends to write (*overlapping writesets*)
- Another option: first-writer-wins rule

---

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>W(Y := 1)</td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>Start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(X) → 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(Y) → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(X := 2)</td>
<td></td>
<td>W(Z := 3)</td>
</tr>
<tr>
<td>W(Z := 3)</td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>R(Z) → 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(Y) → 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W(X := 3)</td>
<td></td>
<td>Commit-Req</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>Abort</td>
</tr>
</tbody>
</table>

*initial values zero*

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back
Snapshot Isolation

**Advantages:**
- Read queries do not block, and run very fast
- *As long as conflicts are rare*, update transactions don’t abort
- Overall better performance than locking-based protocols

\[
\begin{array}{ccc}
T_1 & & T_2 \\
w(x) & & w(y) \\
r(y) & & r(x) \\
commit & & commit
\end{array}
\]

\[x = y = 0\]

**Major disadvantage:**
- Not serializable!

**Also called write skew**

Snapshot Isolation: Multi-Version Implementation

- each write to Q creates a new version of Q
- reads are parameterized by transaction’s *timestamp*
  - satisfied by last write before that timestamp
- Implementing Snapshot Isolation w/ *multi-version consistency control* (MVCC)
  - transaction gets \(StartTS(T_i)\), \(CommitTS(T_i)\),
  - write by \(T_i\) saved with \(CommitTS(T_i)\)
  - read by \(T_i\) satisfied by last version w/ time < \(StartTS(T_i)\)
  - as a result:
    - transaction only see writes committed prior to start
    - i.e. a *snapshot*
Two approaches: *first-committer-wins*, and *first-updater-wins*.

$T_j$ is said to be concurrent with a transaction $T_i$ if either:

- $\text{StartTS}(T_j) \leq \text{StartTS}(T_i) \leq \text{CommitTS}(T_j)$, or
- $\text{StartTS}(T_i) \leq \text{StartTS}(T_j) \leq \text{CommitTS}(T_i)$

Under *first-committer-wins*, $T_i$ checks at commit time to see if any concurrent transaction has written an object that it is trying to write. If so, $T_i$ aborts.

Under *first-updater-wins*, $T_i$ checks at each write. Before writing $Q$, $T_i$:

- Attempts to acquire a write lock on $Q$. If the lock is acquired, $T_i$ aborts if a concurrent transaction $T_j$ has already written $Q$. Commit attempts always succeed.
- If the lock was not successful, $T_i$ waits to see if $T_j$ commits or aborts. If $T_j$ commits, $T_i$ aborts. If $T_j$ aborts:
  - $T_i$ repeats the check for a concurrent writer having updated $Q$. If found,
    - $T_i$ aborts.
  - else
    - $T_i$ commits

---

**Snapshot Isolation**

- **Advantages:**
  - Read query don't block at all, and run very fast
  - If conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols

- **Major disadvantage:**
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
Guarantees of Atomicity and Isolation

- Serializability
  - conflict, view
- Concurrency Control
  - locking
    - 2-phase locking, strict, rigorous
    - granularity, intention locks
- Other approaches to concurrency control
  - timestamp-based
  - optimistic
  - snapshot isolation
- Recovery

Time-stamp Based Concurrency Control

- No locks
- Transactions issued time-stamps when started
- Time-stamps determine the serializability order
- If T1 enters before T2, then T1 < T2 in serializability order
- Say \( \text{timestamp}(T1) < \text{timestamp}(T2) \)
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is aborted
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read, or wrote, that data item, then the write is rejected and T1 is aborted
  - Aborted transactions are restarted with a new timestamp
    - Possibility of starvation
    - Optimistic
**Timestamp-based CC**

- Example

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read($Y$)</td>
<td>read($Y$)</td>
<td>write($Y$)</td>
<td>write($X$)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td>read($X$)</td>
<td></td>
<td>write($Z$)</td>
<td>read($Z$)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td></td>
<td></td>
<td>write($Y$)</td>
<td></td>
</tr>
<tr>
<td>abort</td>
<td></td>
<td>abort</td>
<td></td>
<td>write($Z$)</td>
<td></td>
</tr>
</tbody>
</table>

$TS(T1) < TS(T2) < TS(T3) < TS(T4) < TS(T5)$

**Timestamp-based CC**

- The following set of instructions is not conflict-serializable:

<table>
<thead>
<tr>
<th></th>
<th>$T_3$</th>
<th>$T_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read($Q$)</td>
<td>write($Q$)</td>
<td></td>
</tr>
<tr>
<td>write($Q$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- As discussed before, not even view-serializable:
  - if $T_i$ reads initial value of $Q$ in $S$, must also in $S'$
  - if $T_i$ reads value written from $T_j$ in $S$, must also in $S'$
  - if $T_i$ performs final write to $Q$ in $S$, must also in $S'$
Timestamp-based CC

- Thomas’ Write Rule
  - *Ignore obsolete writes*

- Say \( \text{timestamp}(T1) < \text{timestamp}(T2) \)
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read, or wrote, that data item, then the write is *rejected* and T1 is aborted
    - *If a transaction with larger time-stamp already written that data item, then the write is ignored*

<table>
<thead>
<tr>
<th></th>
<th>T₃</th>
<th>T₄</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read(Q)</strong></td>
<td></td>
<td>write(Q)</td>
</tr>
<tr>
<td><strong>write(Q)</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ignored if \( T₃ < T₄ \)

---

Timestamp-based CC

- As discussed here, has a few issues
  - Starvation
  - Non-recoverable
  - Cascading rollbacks possible

- Most can be solved fairly easily
  - Read up

- We can always add more restrictions to ensure these things
  - The goal is to find the minimal set of restrictions to as to not hinder concurrency
Optimistic Concurrency Control

- Based on validation at transaction end (section 18.6)
- Intuition
  - Let the transactions execute as they wish
  - At the very end when they are about to commit, check if there might be any problems/conflicts etc
    - If no, let it commit
    - If yes, abort and restart
- Upshot:
  - *Hope not too many problems/aborts*
  - Very fast for read transactions, or when low contention

Validation-based (optimistic) Concurrency Control

- Each transaction $T_i$ has 3 timestamps
  - $\text{Start}(T_i)$: when $T_i$ starts execution
  - $\text{Validation}(T_i)$: when $T_i$ enters its validation phase
  - $\text{Finish}(T_i)$: when $T_i$ finishes its write phase
- Serializability order is validation order
  - $\text{TS}(T_i) = \text{Validation}(T_i)$
  - increases concurrency.
- Higher degree of concurrency if conflicts low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.
Validation-based (optimistic) Concurrency Control

- If for all $T_i$ with $TS(T_i) < TS(T_k)$ either one of the following condition holds:
  - $finish(T_i) < start(T_k)$ or
  - $start(T_k) < finish(T_i) < validation(T_k)$ and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_k$.

then validation succeeds and $T_k$ can be committed. Otherwise, validation fails and $T_k$ is aborted.

- Justification: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of $T_k$ do not affect reads of $T_i$ since they occur after $T_i$ has finished its reads.
  - the writes of $T_i$ do not affect reads of $T_k$ since $T_k$ does not read any item written by $T_i$.

Optimistic Concurrency Control

- Serialization order?
  - $T_{25} < T_{26}$

- $T_{25}$ validates?
  - because first

- $T_{26}$ validates?
  - $T_{25}$ did not write

<table>
<thead>
<tr>
<th>$T_{25}$</th>
<th>$T_{26}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read(B)</code></td>
<td><code>read(B)</code></td>
</tr>
<tr>
<td><code>read(A)</code></td>
<td><code>B := B - 50</code></td>
</tr>
<tr>
<td><code>&lt;validate&gt;</code></td>
<td><code>A := A + 50</code></td>
</tr>
<tr>
<td><code>display(A + B)</code></td>
<td><code>&lt;validate&gt;</code></td>
</tr>
<tr>
<td><code>write(B)</code></td>
<td><code>write(A)</code></td>
</tr>
</tbody>
</table>

- finish($T_i$) < start($T_k$) or
- start($T_k$) < finish($T_i$) < validation($T_k$) and the set of data items written by $T_i$ does not intersect with the set of data items read by $T_k$. 
Optimistic Concurrency Control

The “Phantom” problem

- An interesting problem that comes up for dynamic databases
- Schema: accounts(acct_no, balance, zipcode, …)
- Transaction 1: Find the number of accounts in zipcode = 20742, and divide $1,000,000 by that number
- Transaction 2: Insert <acctX, …, 20742, …>
- Execution sequence:
  - T1 locks all tuples corresponding to "zipcode = 20742", finds the total number of accounts
  - T1 does the insert
  - T1 computes bonus = $1,000,000/num_accounts
  - T1 gives bonus to all matching accounts (including T2’s insert!)
    - update accounts set balance += 1000000/num_accounts where zipcode=20742
- Not serializable
- Root problem: locking granularity
  - needed to have lock on whole table
- Prevented by serializability, snapshot isolation
Weak Levels of Isolation in SQL

- SQL can be parameterized by isolation level:
  - **Read uncommitted**: allows *uncommitted writes* to be read
  - **Read committed**: only read committed data, repeated reads of same data might return different values as other transactions commit
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value
    - so read locks should be retained or caching used
    - transaction-local writes can change subsequent reads
    - Phantom problem not necessarily prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Serializable**: default, strongest (except for *linearizable*)
  - In many database systems, *read committed is the default*
    - has to be explicitly changed to serializable when required
      - *set isolation level serializable*
    - Oracle calls snapshot isolation "serializable"

Weak Isolation Levels: Read Uncommitted

\[
\begin{array}{c|c|c|c}
T_1 & T_2 & T_3 \\
\hline
x = 0 & & \\
start & & \\
write(X) = 1 & & \\
write(Y) = 1 & & \\
commit & & read(X) = 3 \\
\end{array}
\]

- Not serializable
- Doesn’t guarantee recoverable scheds
- Not free from cascading aborts
### Weak Isolation Levels: Read Committed

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(X) = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(Y) = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| start |
| write(X) = 2 |
| commit |

- Not serializable
- Guarantees recoverable scheds
- Free from cascading
- Stronger isolation than read uncommitted

### Weak Isolation Levels: Repeatable Reads

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>start</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(X) = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write(Y) = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| start |
| write(X) = 2 |
| commit |

- Not serializable
- Guarantees recoverable scheds
- Free from cascading aborts
- Still not serializable, but even stronger isolation
Weak Isolation Levels: Snapshot Iso

\[
\begin{array}{ccc}
T_1 & T_2 & T_3 \\
x = 0
\end{array}
\]

\[
\begin{align*}
&\text{start} \\
&\text{write}(X) = 1 \\
&\text{write}(Y) = 1
\end{align*}
\]

commit

\[
\begin{align*}
&\text{start} \\
&\text{read}(X) = 0 \\
&\text{write}(X) = 3
\end{align*}
\]

\[
\begin{align*}
&\text{commit} \\
&\text{read}(X) = 3
\end{align*}
\]

\[
\begin{align*}
&\text{start} \\
&\text{write}(X) = 2 \\
&\text{commit}
\end{align*}
\]

• Not serializable

• Guarantees recoverable scheds

• Free from cascading aborts

• Faster

abort

The Real World

• Shopping cart
  • add to cart
  • becomes unavailable before you check out
  • definitely < \textit{repeatable reads}

• Airline reservation w/ seat selections
  • multiple passengers reserving concurrently
  • all good until you hit the PAY button
    • then disappears
  • snapshot isolation?