1. (Ex 16.2)

16.2.1. \( T_2 : R(Y), W(Y) \)

without concurrency control, \( T_2 \) could interfere with \( T_1 \)

\( T_1 : R(x) \ W(x) \ \ R(Y) \ W(Y) \)

\( T_2 : \quad R(Y) \ W(Y) \) \( \} \) lost update.

16.2.2. with strict 2PL locking, either \( T_1 \) or \( T_2 \) will

obtain exclusive lock on \( Y \) first.

\( T_1 \) obtains first \( \quad T_2 \) obtains first

\( T_1 : \quad T_2 \)

\( X(x) \)
\( R(x) \)
\( X(Y) \)
\( R(Y) \)
\( W(Y) \)
\( \text{commit} \)

\( X(Y) \)
\( R(Y) \)
\( W(Y) \)
\( \text{commit} \)

\( T_1 : \quad T_2 \)

\( X(x) \)
\( R(x) \)
\( W(x) \)

\( X(Y) \)
\( R(Y) \)
\( W(Y) \)
\( \text{commit} \)
2. (Ex. 14.2)

 Serializable: unknown.
 If both T1 and T2 commit ⇒ NO
 If T1 aborts, T2 commit ⇒ YES (T1(abort)→T2)

 Conflict Serializable: unknown.
 If both commit ⇒ precede graph is

 \[
 \begin{align*}
 T_1 & \rightarrow \ T_2 \\
 \end{align*}
 \]
 ⇒ cycle ⇒ NO
 If either transaction aborts ⇒ YES.

 View Serializable: Unknown
 If both T1 and T2 commit ⇒ NO
 If T1 abort, T2 commit ⇒ YES (T1(abort)→T2)

 Recoverable: YES
 Avoid cascading abort: YES
 Strict: NO.
 Since x written by T1 is overwritten by T2 before T1 commits/aborts

 2.

 Serializable: YES
 (T1 → T2)

 Conflict Serializable: YES
 Even both commit, graph is a cycle (T1 → T2)

 View Serializable: YES
 (T1 → T2)

 Recoverable: Unknown
 If T2 commits after T1 commits ⇒ YES
 If T2 commits before T1 ⇒ NO

 Avoid cascading abort: NO.
 Since T2 read x written by T1 before T1 commit/abort.

 Strict: NO.
 Since x written by T1 is read by T2 before T2 commits.
5) Serializable: YES
   \((T_1(abort) \Rightarrow T_2 \equiv T_2)\)

   - Conflict - Serializable: YES
     - Graph has only 1 node => acyclic

   - View Serializable: YES
     \((T_1 \Rightarrow T_2) \equiv T_2\)

   - Recoverable: YES

   - Avoid cascading abort: YES

   - Strict: No
     - Since x written by T_2 is overwritten by T_1 before T_2 aborts.

9) Serializable: YES
   \((T_1(abort) \Rightarrow T_2 \equiv T_2)\)

   - Conflict - Serializable: YES
     - Graph has only 1 node => acyclic

   - View Serializable: YES
     \((T_1(abort) \Rightarrow T_2 \equiv T_2)\)

   - Recoverable: No
     - Since T_2 read x from written by T_1 while T_2 commits before T_1 abort.

   - Avoid cascading abort: No
     - Since T_2 commits before T_1 abort while T_2 read x written by T_1.

   - Strict: No
     - Since T_2 read x from T_1 before T_1 commits.
Due to conflict between $T_1$ and $T_2$

Conflict Serializable: No

View Serializable: No. Since $T_3$ reads $X$ written by $T_1$.

Since $T_3$ reads $X$ written by $T_1$ =>
$T_3$ must follow $T_1$ in serial schedule $T_1 \rightarrow T_3$.

$T_2$ can't be at the middle since $T_3$ will read $X$ from $T_2$.
$T_2$ can't be at the end since $T_2$ will be last write of $X$.
$T_2$ can't be at front since $T_2$ will not read $X$ from initial value.

=> Can't find equivalent serial schedule.

Serializable: No

Due to conflict between $T_1$ and $T_2$

Conflict Serializable: No

Graph:

View Serializable: No. Explain as the case of (11)

Recoverable: YES

$T_3$ read $X$ from $T_1$, $T_2$ read and commits after $T_1$, $T_2$ commits.

Avoid cascading abort: No

Since $T_3$ reads $X$ written by $T_1$ before $T_2$ commits.

Strict: No

Since $X$ written by $T_2$ $T_3$ overwrites $X$ written by $T_2$ before $T_2$ commits.
3. (Ex. 17.8.2)

If the second query is

\[ \text{Salary} = 120 \]

Replace \((\text{Salary} = 120)\) where \(\text{EMP.ename} = 'Santa'\).

Then it could conflict with the presented query.

For example, suppose the EMP has a 'Santa' tuple

<table>
<thead>
<tr>
<th>eid</th>
<th>ename</th>
<th>age</th>
<th>salary</th>
<th>hired</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>'Santa'</td>
<td>30</td>
<td>120</td>
<td>20</td>
</tr>
</tbody>
</table>

If we execute query 1 and query 2 serially, the result will be

\[
\begin{array}{c}
Q_1 \\ R(\text{EMP}) \\
W(\text{EMP}) \\
Q_2 \\
\end{array} \Rightarrow \begin{array}{c}
\text{ename} \\
'Santa' \\
\text{salary} \\
120 \\
\end{array}
\]

\[
\begin{array}{c}
Q_1 \\
R(\text{EMP}) \\
W(\text{EMP}) \\
Q_2 \\
\end{array} \Rightarrow \begin{array}{c}
\text{ename} \\
'Santa' \\
\text{salary} \\
120 \\
\end{array}
\]

If we execute the 2 queries concurrently, it may be

\[
\begin{array}{c}
Q_1 \\
R(\text{EMP}) \\
R(\text{EMP}) \\
W(\text{EMP}) \\
Q_2 \\
\end{array} \Rightarrow \begin{array}{c}
\text{ename} \\
'Santa' \\
\text{salary} \\
120 \\
\end{array}
\]

\[
\begin{array}{c}
Q_1 \\
R(\text{EMP}) \\
R(\text{EMP}) \\
W(\text{EMP}) \\
Q_2 \\
\end{array} \Rightarrow \text{different from both serial schedule!}
\]

Using locking tuples, then \(Q_2\) can't read until \(Q_1\) releases the exclusive lock or \(Q_2\) will hold the exclusive lock until it is done and releases to \(Q_1\). Thus, the result will be similar to one of the 2 serial schedule.