1. We are considering the following classes of schedules (page numbers refer to the course book Ramakrishnan & Gehrke):

conflict serializable (CSER): schedule is conflict equivalent with some serial schedule (two schedules are conflict equivalent if they have the same operations of the same transactions and the order of conflicting pairs of operations is the same in them) (p 550)

recoverable (REC): transactions commit only after all those transactions whose changes they read, commit (p 550)

avoiding cascading aborts (ACA): transactions read only changes by committed transactions (p 530)

strict (STR): other transactions don’t read the value written by some transaction \( T \) nor overwrite it until \( T \) is committed or aborted (p 552)

The classes are not disjoint: \( \text{STR} \subset \text{ACA} \subset \text{REC} \); see also the diagram on p. 576. If the schedule does not contain commit or abort operations, a commit is assumed after all operations in the schedule. Notation \( +C \) means that the schedule is in class \( C \); notation \( -C \) means the schedule is not in class \( C \).

(a) \( T1:R(X), T2:R(X), T1:W(X), T2:W(X) \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, +\text{ACA}, -\text{STR}\)

(b) \( T1:W(X), T2:R(Y), T1:R(Y), T2:R(X) \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(+\text{CSER}, +\text{REC}, -\text{ACA}, -\text{STR}\); we don’t know if the schedule is recoverable or not, because we don’t know in which order \( T1 \) and \( T2 \) commit (if they commit at all).

(c) \( T1:R(X), T2:R(Y), T3:W(X), T2:R(X), T1:R(Y) \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2} \\
\text{T3}
\end{array}
\]

\(+\text{CSER}, +\text{REC}, -\text{ACA}, -\text{STR}\); we don’t know if the schedule is recoverable or not, because we don’t know in which order \( T2 \) and \( T3 \) commit (if they commit at all).

(d) \( T1:R(X), T1:R(Y), T1:W(X), T2:R(Y), T3:W(Y), T1:W(X), T2:R(Y) \)
\[
\begin{array}{c}
\text{T1} \\
\text{T3} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, -\text{ACA}, -\text{STR}\); we don’t know if the schedule is recoverable or not, because we don’t know in which order \( T2 \) and \( T3 \) commit (if they commit at all).

(e) \( T1:R(X), T2:W(X), T1:W(X), T2:ABORT, T1:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, +\text{ACA}, -\text{STR}\)

(f) \( T1:R(X), T2:W(X), T1:W(X), T2:COMMIT, T1:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, +\text{ACA}, -\text{STR}\)

(g) \( T1:W(X), T2:R(X), T1:W(X), T2:ABORT, T1:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, -\text{REC}, -\text{ACA}, -\text{STR}\)

(h) \( T1:W(X), T2:R(X), T1:W(X), T2:COMMIT, T1:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, -\text{REC}, -\text{ACA}, -\text{STR}\)

(i) \( T1:W(X), T2:R(X), T1:W(X), T2:COMMIT, T1:ABORT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2}
\end{array}
\]

\(-\text{CSER}, -\text{REC}, -\text{ACA}, -\text{STR}\)

(j) \( T2:R(X), T3:W(X), T3:COMMIT, T1:W(Y), T1:COMMIT, T2:R(Y), T2:W(Z), T2:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2} \\
\text{T3}
\end{array}
\]

\(+\text{CSER}, +\text{REC}, +\text{ACA}, +\text{STR}\)

(k) \( T1:R(X), T2:W(X), T2:COMMIT, T1:W(X), T1:COMMIT, T3:R(X), T3:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2} \\
\text{T3}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, +\text{ACA}, +\text{STR}\)

(l) \( T1:R(X), T2:W(X), T1:W(X), T3:R(X), T1:COMMIT, T2:COMMIT, T3:COMMIT \)
\[
\begin{array}{c}
\text{T1} \\
\text{T2} \\
\text{T3}
\end{array}
\]

\(-\text{CSER}, +\text{REC}, -\text{ACA}, -\text{STR}\)

Note that the serializability graphs differ from the course book R & G in those schedules that have abort operations in them (i), (g) ja (i). Instead all graphs are according to the Elmasri & Navathe book. In R & G the nodes are only the committed transactions, which means that for example schedule (i) should be serializable, which is in contradiction with the footnote in p. 525. It is better to use the definition in E & N here.
2. Strict 2PL does not allow any of the schedules. Attempting schedule (a) leads to a deadlock (both transactions wait for each other to release its locks):

\[ T_1 \rightarrow T_2 \]

2PL allows only schedule (b): transaction \( T_1 \) can release its locks after the operation \( T_1: R(Y) \).

3. (a) During Analysis transactions \( T_1 \) and \( T_3 \) are collected to the list of noncommitted transactions and \( T_2 \) to the list of committed transactions.
(b) During Redo the operations of committed transactions (\( T_2 \)) are redone. However it is not necessary to redo the operation with LSN 103, because pageLSN(\( P_3 \)) \( \geq 103 \), i.e. the change has been already written to disk.
(c) During Undo the operations of noncommitted transactions (\( T_1 \) and \( T_3 \)) are undone in reverse order. The operation with LSN 107 is undone, because pageLSN(\( P_3 \)) \( \geq 107 \) and the operation with LSN 101 is undone because pageLSN(\( P_5 \)) \( \geq 101 \).

4. When \( T_2 \) is aborted, the log is scanned backwards undoing all changes made by \( T_2 \). Compensation records for operations 108, 105 and 101 are written to the log and the operations are undone. Finally an end record for \( T_2 \) is written to the log.

5. In the Analysis phase the transaction table and the dirty page table are initially empty (see the last checkpoint). When the log has been scanned forward starting from the last checkpoint, in the end the transactions \( T_1 \) and \( T_2 \) (both active) are in the transaction table, lastLSN(\( T_1 \)) = 109, lastLSN(\( T_2 \)) = 112, page \( p \) is in the dirty page table and recLSN(\( p \)) = 107.

In the Redo phase every operation in the log after the earliest recLSN value are redone. It is not necessary to redo operations 107 and 108, because pageLSN(\( p \)) = 108. The operations 109, 110 and 112 are redone. During this phase no new records are written to the log.

In the Undo phase the transactions in the transaction table are aborted. The aborting of \( T_2 \) has already started; an abort record for \( T_1 \) is written to the log:

113: <abort, T1>

Next, every operation the transactions have executed is undone. Initially ToUndo = \{lastLSN(\( T_1 \)), lastLSN(\( T_2 \))\} = \{109, 112\}. The following is repeated until ToUndo = \( \emptyset \): Choose the largest value \( u \) in the set ToUndo, let ToUndo := ToUndo \( \setminus \{ u \} \) and process the log record \( u \).

The first operation to be undone is 112. It is itself an undo operation, so the only thing to do is to insert the undoNextLSN value (108) in the log record into the set ToUndo. Now ToUndo = \{108, 109\}.

The next operation is 109. It is an insert operation, so a compensation log record is written:

114: <CLR, insert, T1, p, j, y, 107>

and the record \( j \) is deleted from page \( p \). The prevLSN value (107) is inserted into the set ToUndo.

Now ToUndo = \{107, 108\}. The next operation is 108. It is an insert operation, so a compensation log record is written:

115: <CLR, insert, T2, p, k, z, 106>

and the record \( k \) is deleted from page \( p \). The prevLSN value (106) is inserted into the set ToUndo.

Now ToUndo = \{106, 107\}. The next operation is 107. It is an insert operation, so a compensation log record is written:

116: <CLR, insert, T1, p, i, x, 105>

and the record \( i \) is deleted from page \( p \). The prevLSN value (105) is inserted into the set ToUndo.

Now ToUndo = \{105, 106\}. The next operation is 106. It is a start operation, so a record marking the end of abortion for \( T_2 \) is written to the log:

117: <end, T2>

and \( T_2 \) is removed from the transaction table. The next operation is 105. It is a start operation, so a record marking the end of abortion for \( T_1 \) is written to the log:

118: <end, T1>

and \( T_1 \) is removed from the transaction table. Now ToUndo = \( \emptyset \), and the Undo phase ends (as does the whole recovery).

6. —