3. Transaction theory

6 Serializability
3. Transaction theory

6. Serializability

7. View Serializability
3. Transaction theory

6. Serializability

7. View Serializability

8. Conflict Serializability
3. Transaction theory

6 Serializability

7 View Serializability

8 Conflict Serializability

9 Conflict Graph
3. Transaction theory

6 Serializability

7 View Serializability

8 Conflict Serializability

9 Conflict Graph

10 Closure Properties
3. Transaction theory

6. Serializability

7. View Serializability

8. Conflict Serializability

9. Conflict Graph

10. Closure Properties

11. Transaction Abort and Fault Tolerance
Serializability

- Introduction to the topic
- Formalization of schedules
- Serializability concepts
- Comparison of serializability concepts
Introduction to Serializability

\[ T_1 : \text{read}(A); \ A := A - 10; \ \text{write}(A); \ \text{read}(B); \]
\[ B := B + 10; \ \text{write}(B); \]
\[ T_2 : \text{read}(B); \ B := B - 20; \ \text{write}(B); \ \text{read}(C); \]
\[ C := C + 20; \ \text{write}(C); \]

- Execution alternatives for two transactions:
  - Serial, e.g. \( T_1 \) before \( T_2 \)
  - Interleaved, e.g. alternating steps of \( T_1 \) and \( T_2 \)
Interleaved execution: Example

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Execution 2</th>
<th>Execution 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_2$</td>
<td>$T_2$</td>
</tr>
<tr>
<td>read($A$)</td>
<td>read($A$)</td>
<td>read($A$)</td>
</tr>
<tr>
<td>$A − 10$</td>
<td>$A − 10$</td>
<td>$A − 10$</td>
</tr>
<tr>
<td>write($A$)</td>
<td>write($B$)</td>
<td>write($A$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>$B − 20$</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$B + 10$</td>
<td>$B + 10$</td>
<td>write($B$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($B$)</td>
<td>write($B$)</td>
</tr>
<tr>
<td>read($B$)</td>
<td>read($B$)</td>
<td>read($B$)</td>
</tr>
<tr>
<td>$B − 20$</td>
<td>$B + 10$</td>
<td>write($C$)</td>
</tr>
<tr>
<td>write($B$)</td>
<td>write($C$)</td>
<td>write($C$)</td>
</tr>
<tr>
<td>read($C$)</td>
<td>read($B$)</td>
<td>read($C$)</td>
</tr>
<tr>
<td>$C + 20$</td>
<td>$C + 20$</td>
<td>write($B$)</td>
</tr>
<tr>
<td>write($C$)</td>
<td>write($C$)</td>
<td>write($C$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Results of different executions

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A + B + C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Value</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>After execution 1</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>After execution 2</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>After execution 3</td>
<td>0</td>
<td>20</td>
<td>30</td>
<td>50</td>
</tr>
</tbody>
</table>
Simplified model

*Lock-Unlock-Model*

\[ T_1 : \text{lock } A; \text{ unlock } A; \text{ lock } B; \text{ unlock } B; \]
\[ T_2 : \text{lock } B; \text{ unlock } B; \text{ lock } C; \text{ unlock } C; \]
Interleaved Executions with Lock/Unlock: Example

<table>
<thead>
<tr>
<th>Execution 1</th>
<th>Execution 2</th>
<th>Illegal execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_1$ $T_2$</td>
</tr>
<tr>
<td>lock $A$</td>
<td>lock $A$</td>
<td>lock $A$</td>
</tr>
<tr>
<td>unlock $A$</td>
<td>unlock $A$</td>
<td>unlock $A$</td>
</tr>
<tr>
<td>lock $B$</td>
<td>lock $B$</td>
<td>lock $B$</td>
</tr>
<tr>
<td>unlock $B$</td>
<td>unlock $B$</td>
<td>unlock $B$</td>
</tr>
<tr>
<td>read $C$</td>
<td>lock $B$</td>
<td>unlock $B$</td>
</tr>
<tr>
<td>unlock $B$</td>
<td>unlock $B$</td>
<td>unlock $B$</td>
</tr>
<tr>
<td>unlock $C$</td>
<td>lock $C$</td>
<td>lock $B$</td>
</tr>
<tr>
<td>unlock $B$</td>
<td>unlock $C$</td>
<td>unlock $B$</td>
</tr>
<tr>
<td>unlock $C$</td>
<td>unlock $C$</td>
<td>unlock $C$</td>
</tr>
</tbody>
</table>
Serializability

An interleaved execution of a set of transactions is called **serializable**, if the result of the interleaved execution is identical with the result of a (randomly chosen) serial execution of the same set of transactions.
The concept of schedules
The Read-Write Model

- *Transaction T* is a finite sequence of operations (steps) $p_i$ of the form $r(x_k)$ or $w(x_k)$:

  \[ T = p_1 p_2 p_3 \ldots p_n \text{ with } p_i \in \{ r(x_k), w(x_k) \} \]

- A *complete transaction T* has as last step either an *Abort* $a$ or a *Commit* $c$:

  \[ T = p_1 \ldots p_n a \]

  or

  \[ T = p_1 \ldots p_n c. \]
Interleaved transactions

**SHUFFLE**(*T*): set of all possible interleaved executions of the steps of all transactions *T*$_i$ in the set *T*

- All steps of the transaction *T*$_i$ are included only once
- The relative order of the steps of a transaction is preserved

\[
T_1 := r_1(x)w_1(x) \\
T_2 := r_2(x)r_2(y)w_2(y)
\]

\[
\text{SHUFFLE}(T) = \{ r_1(x)w_1(x)r_2(x)r_2(y)w_2(y), \\
r_2(x)r_1(x)w_1(x)r_2(y)w_2(y), \\
\ldots \}
\]
Schedule

A complete schedule is a \textsc{Shuffle}-Element of a set of complete transactions.

A schedule is a prefix of a complete schedule.

\[
\underbrace{r_1(x)r_2(x)w_1(x)\ r_2(y)\ a_1\ w_2(y)\ c_2}_{\text{a schedule}}
\underbrace{a\ \text{complete schedule}}_{\text{a complete schedule}}
\]
Serial schedule

- A serial schedule $s$ for $T$ is a complete schedule of the following form:

$$s := T_{\rho(1)} \cdots T_{\rho(n)}$$

for a permutation $\rho$ of $\{1, \ldots, n\}$

- Resulting serial schedules for two transactions: $T_1 := r_1(x)w_1(x)c_1$ and $T_2 := r_2(x)w_2(x)c_2$:

$$s_1 := \underbrace{T_1}_{T_1} \underbrace{T_2}_{T_2}$$

$$s_2 := \underbrace{T_2}_{T_2} \underbrace{T_1}_{T_1}$$
Correctness criteria

A schedule $s$ is **correct**, if its effect (the result of the execution of the schedule) is equivalent to the effect of a (randomly chosen) serial schedule $s'$ regarding the same set of transactions ($s \approx s'$).

If a schedule $s$ is equivalent to a serial schedule $s'$, then $s$ is **serializable** (to $s'$).

- Open: How to define "being equivalent"?
**View serializability**

- Idea: Effect of a transaction only depends on what values were seen by the transaction.
- It reads the value that was written last
- Special treatment necessary for the initialization and for the final result of a schedule
View serializability: Preparations

- Artificial additional transactions:
  1. Initial transaction $T_0$: initially writes all involved objects
  2. Terminal transaction $T_∞$: reads all involved objects in the end
Reads-from-relation

- If $\rightarrow_s$ is the relation "temporally before in the schedule $s$", then "$r_j(x)$ reads $x$ from $T_i$" if and only if:
  - $w_i(x) \rightarrow_s r_j(x)$ and
  - $\forall k (w_i(x) \rightarrow_s w_k(x) \land w_k(x) \rightarrow_s r_j(x))$.

- Reads-from-relation $RF(s)$ for a schedule $s$:
  $RF(s) := \{ (T_i, x, T_j) | r_j(x) \text{ reads } x \text{ from } T_i \}$
View equivalence

Two schedules $s$ and $s'$ are **view equivalent**, if:

1. $\text{op}(s) = \text{op}(s')$
   - $\text{op}(s)$: the set of all steps occurring in $s$ including $a$ and $c$ (Sets must be identical for both schedules)

2. $\text{RF}(s) = \text{RF}(s')$
   - Schedules $s$ and $s'$ have the same ”Reads-from-relation”
View equivalence: Example I

Given two complete schedules $s_1$ and $s_2$ consisting of two transactions $T_1$ and $T_2$ with:

$$s_1 := r_1(x)r_2(y)w_1(y)w_2(y)c_1c_2$$
$$s_2 := r_1(x)w_1(y)r_2(y)w_2(y)c_2c_1$$
View equivalence: Example II

For calculating the reads-from-relations $RF(s_1)$ and $RF(s_2)$, $s_1$ and $s_2$ are expanded by the initial transaction $T_0$ and the terminal transaction $T_\infty$ in the following way:

$$s_1 := \underbrace{w_0(x)w_0(y)c_0}_{T_0} \underbrace{r_1(x)r_2(y)w_1(y)w_2(y)c_1c_2}_{T_\infty} \underbrace{r_\infty(x), r_\infty(y)c_\infty}_{T_\infty}$$

$$s_2 := \underbrace{w_0(x)w_0(y)c_0}_{T_0} \underbrace{r_1(x)w_1(y)r_2(y)w_2(y)c_2c_1}_{T_\infty} \underbrace{r_\infty(x), r_\infty(y)c_\infty}_{T_\infty}$$
View equivalence: Example III

- Resulting \textit{reads-from-relations}:

\[
RF(s_1) := \{(T_0, x, T_1), (T_0, y, T_2), (T_0, x, T_{\infty}), (T_2, y, T_{\infty})\}
\]

\[
RF(s_2) := \{(T_0, x, T_1), (T_1, y, T_2), (T_0, x, T_{\infty}), (T_2, y, T_{\infty})\}
\]

- \textit{View equivalence} of \(s_1\) and \(s_2\): Comparison of \textit{reads-from-relations}

- Since \(RF(s_1) \neq RF(s_2)\), \(s_1\) and \(s_2\) are \textit{not} view equivalent
View Serializability

A schedule $s$ is view serializable, only if $s$ is view equivalent to a serial schedule.

- Set of all view serializable schedules: $VSR$ (view serializability)
- For $n$ transactions, there exist $n!$ serial schedules
- Problems:
  - Exponential complexity for testing
  - Testing requires complete schedules (blind writes and transaction aborts)
View Serializability: Example

- Obviously, schedule $s_2$ is view serializable, because $s_2$ is serial.
- Possible serial schedules $s'$ and $s''$ for $s_1$:
  - $s' = T_1 T_2 = r_1(x) w_1(y) c_1 r_2(y) w_2(y) c_2$
    \[
    RF(s') := \{(T_0, x, T_1), (T_1, y, T_2), (T_0, x, T_\infty), (T_2, y, T_\infty)\}
    \]
    $\rightarrow RF(s') \neq RF(s_1)$
  - $s'' = T_2 T_1 = r_2(y) w_2(y) c_2 r_1(x) w_1(y) c_1$
    \[
    RF(s'') := \{(T_0, y, T_2), (T_0, x, T_1), (T_0, x, T_\infty), (T_1, y, T_\infty)\}
    \]
    $\rightarrow RF(s'') \neq RF(s_1)$

$RF(s') \neq RF(s_1)$ and $RF(s'') \neq RF(s_1)$: Schedule $s_1$ is not serializable!
Conflict Serializability

- Idea: Only the relative order of operations and not the actually read value is of importance in conflict situations.
- It is not necessary to know which transaction has most recently written a value, only whether the value has been written before or after a transaction.
## Conflicts

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $A$</td>
<td>read $A$</td>
</tr>
</tbody>
</table>

*order independent*

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read $A$</td>
<td>write $A$</td>
</tr>
</tbody>
</table>

*order dependent*

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>write $A$</td>
<td>read $A$</td>
</tr>
</tbody>
</table>

*order dependent*

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>write $A$</td>
<td>write $A$</td>
</tr>
</tbody>
</table>

*order dependent*
Conflict Serializability

- Conflict relation $C$ of $s$:
  \[ C(s) := \{ (p, q) \mid p, q \text{ are in conflict and } p \rightarrow_s q \} \]

- Conflict matrix:

\[
\begin{array}{c|cc}
 & r_i(x) & w_i(x) \\
\hline
r_j(x) & \sqrt{} & - \\
w_j(x) & - & - \\
\end{array}
\]
Adjusted conflict relation

- \( \text{conf}(s) \) is an "adjusted" conflict relation, which includes no aborted transactions
  \[
  \text{conf}(s) := C(s) - \{ (p, q) \mid (p \in t' \lor q \in t') \land t' \in \text{aborted}(s) \}
  \]
- \( \text{aborted}(s) \): Set of aborted transactions in schedule \( s \)
Adjusted conflict relation: Example

- Given schedule \( s \):
  \[
s = r_1(x)w_1(x)r_2(x)r_3(y)w_2(y)c_2a_1c_3
  \]
- Conflict relation for \( s \):
  \[
  C(s) := \{(w_1(x), r_2(x)), (r_3(y), w_2(y))\}
  \]
- Removing aborted transaction \( T_1 \) from \( s \):
  \[
  \text{conf}(s) := \{(r_3(y), w_2(y))\}
  \]
Conflict equivalence

Two schedules $s$ and $s'$ are \textit{conflict equivalent} ($s \simeq_c s'$) if:

1. $op(s) = op(s')$
2. $conf(s) = conf(s')$
Conflict serializability

A schedule $s$ is **conflict serializable**, if and only if $s$ is conflict equivalent to a serial schedule.

- Class of all conflict serializable schedules: **CSR** (conflict serializability)
Conflict serializability: Example I

- Given two schedules \( s \) and \( s' \):
  \[
  s = r_1(x) r_1(y) w_2(x) w_1(y) r_2(z) w_1(x) w_2(y) \\
  s' = r_1(y) r_1(x) w_1(y) w_2(x) w_1(x) r_2(z) w_2(y)
  \]

- Question:
  Are schedules \( s \) and \( s' \) conflict equivalent?

- Step 1:
  \( op(s) = op(s') \) applies, since all database operations occurring in \( s \) occur in \( s' \) as well; also applies vice versa
**Conflict serializability: Example II**

- **Step 2: Adjusted conflict relations**

\[
\text{conf}(s) = \{ (r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)), (w_1(y), w_2(y)) \}
\]

\[
\text{conf}(s') = \{ (r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)), (w_1(y), w_2(y)) \}
\]

- \(\text{conf}(s) = \text{conf}(s')\) applies; so the conflict relations are equal and therefore \(s\) and \(s'\) are conflict equivalent
Conflict serializability: Example III

- Test for conflict serializability by comparison with serial schedules
- Given schedule $s$:

\[ s = r_1(x)r_1(y)w_2(x)w_1(y)r_2(z)w_1(x)w_2(y) \]
Conflict serializability: Example IV

- Adjusted conflict relation for $s$:

$$\text{conf}(s) = \{(r_1(x), w_2(x)), (w_2(x), w_1(x)), (r_1(y), w_2(y)), (w_1(y), w_2(y))\}$$

- Possible serial schedule $s_1$:

$$s_1 = T_1 T_2 = r_1(x) r_1(y) w_1(y) w_1(x) c_1 w_2(x) r_2(z) w_2(y) c_2$$

- Conflict relation of $s_1$ is *not* equal to the conflict relation of $s$:

$$\text{conf}(s_1) = \{(r_1(x), w_2(x)), (w_1(x), w_2(x)), (r_1(y), w_2(y)), (w_1(y), w_2(y))\}$$
Conflict serializability: Example V

- Possible serial schedule $s_2$ as a candidate:
  
  $$s_2 = T_2 T_1 = w_2(x) r_2(z) w_2(y) c_2 r_1(x) r_1(y) w_1(y) w_1(x) c_1$$

- Conflict relation of $s_2$ is not equal to the conflict relation of $s$:
  
  $$\text{conf}(s_2) = \{(w_2(x), r_1(x)), (w_2(y), r_1(y)), (w_2(y), w_1(y)), (w_2(x), w_1(x))\}$$

- Therefore: $s \notin \text{CSR}$, which means that schedule $s$ is not conflict serializable
Conflict serializability: Example VI

Schedule:

\[ s_3 = r_1(x)r_2(x)w_2(y)c_2w_1(x)c_1 \]

is obviously conflict serializable, since only one conflict occurs
Graph-based Test

- Conflict graph $G(s) = (V, E)$ for schedule $s$:
  1. Set of vertices $V$ includes all transactions occurring in $s$
  2. Set of edges $E$ includes all directed edges between two conflicting transactions:
     
     $$(t, t') \in E \iff t \neq t' \land (\exists p \in t)(\exists q \in t') \text{ with } (p, q) \in \text{conf}(s)$$
Chronological sequence of three transactions

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r(y)$</td>
<td>$r(y)$</td>
<td>$r(u)$</td>
</tr>
<tr>
<td></td>
<td>$w(y)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w(x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w(x)$</td>
<td>$w(x)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$w(z)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$w(x)$</td>
</tr>
</tbody>
</table>

$s = r_1(y)r_3(u)r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)$
Conflict graph

\[ G(s): \]

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1 \]
Properties of conflict graph $G(s)$

1. If $s$ is a serial schedule, the given conflict graph is an acyclic graph.
2. For every acyclic graph $G(s)$, a serial schedule $s'$ can be constructed in order to make $s$ conflict serializable to $s'$ (Test e.g. by topological sorting).
3. If a graph contains cycles, the respective schedule is not conflict serializable.
Topological Sorting

- (Recursive) iteration through the graph can detect the absence of cycles
  - Each node is visited once
  - Processing status per node and time stamp for entering and leaving is noted
  - Repeated reentry on arbitrary unprocessed nodes

- Topological sorting can be done in the same run
  - Times of leaving provides the order

- Sorted sequence defines a conflict-equivalent serial schedule

- See example of Saake/Sattler: Algorithmen und Datenstrukturen (the absent-minded professor ...)

Thomas Leich

Transaction Management

Last updated: 27.10.2017
Topological Sorting: Input

- panties
- trousers
- belt
- shirt
- tie
- jacket
- socks
- shoes
- watch
Topological Sorting: Result

- panties
  - trousers
    - belt
    - tie
      - jacket
  - 11/16
  - 12/15
  - 6/7

- watch
  - socks
    - shoes
  - 17/18
  - 13/14
  - 9/10
Topological Sorting: Alternative result

- panties
- trousers
- belt
- tie
- jacket
- shirt
- shoes
- socks
- watch

17/18
9/12
1/4
5/8
6/7
2/3
15/16
10/11
13/14
Conflict Graphs and Serializability

For every schedule $s$ applies:

$$G(s) \text{ acyclical} \iff s \in \text{CSR}$$
Problems during run-time

- Verification of serializability properties during run-time
  - Only incomplete schedules available during run-time
  - Observation of incomplete schedules is necessary
  - Transactions that have not performed a *commit* yet can still be aborted anytime
- How does this affect the accuracy of the verification?
Closure Properties

1. **Prefix Closure**
   If property $E$ applies to a schedule $s$, $E$ also applies to any prefix of $s$. If $E$ is fulfilled at the end of a schedule, $E$ must not have been violated before.

2. **Commit Closure**
   If $E$ applies to $s$, $E$ also applies to $CP(s)$ ("committed projection"). If $E$ applies to a set of transactions, it still applies if some of them are aborted.

3. **Prefix-commit Closure (PCC)**
   Prefix-commit closure is the conjunction.
Closure Properties

- **VSR**-Schedules are **not** prefix-commit closed
  - not prefix closed: blind writes can repair
  - not commit closed: last writes can be changed by Abort

- **CSR**-Schedules are prefix-commit closed
  - prefix closed: cycles in the graph remain cycles
  - commit closed: Aborts can not generate any cycles
CSR vs VSR: Example I

Does $\text{CSR} \subset \text{VSR}$ or $\text{VSR} \subset \text{CSR}$?

- Given: schedule $s$:

$$s = r_1(y)r_3(w)\overbrace{r_2(y)w_1(y)w_1(x)w_2(x)w_2(z)w_3(x)c_2c_1c_3}^{T_2 \rightarrow T_1}$$

$$T_1 \rightarrow T_2$$
CSR vs VSR: Example II

Schedule $s$ is not conflict serializable, since conflict graph $G(s)$ contains a cycle.
CSR vs VSR: Example III

Is $s$ view serializable?

- Determining of the reads-from-relation $RF$

$$RF(s) = \{(T_0, y, T_1), (T_0, w, T_3), (T_0, y, T_2), (T_3, x, T_\infty), (T_1, y, T_\infty), (T_2, z, T_\infty), (T_0, w, T_\infty)\}$$

- Serial schedule $s' = T_2 \ T_1 \ T_3$:

$$s' = r_2(y)w_2(x)w_2(z)c_2r_1(y)w_1(y)w_1(x)c_1r_3(w)w_3(x)c_3$$

- Reads-from-relation for schedule $s'$:

$$RF(s') = \{(T_0, y, T_1), (T_0, w, T_3), (T_0, y, T_2), (T_3, x, T_\infty), (T_1, y, T_\infty), (T_2, z, T_\infty), (T_0, w, T_\infty)\}$$
CSR vs VSR: Example IV

- \( RF(s) = RF(s') \) applies, therefore, schedule \( s \) is view serializable
- Therefore: Conflict serializability is more restrictive than view serializability

\[ \neg (CSR \supset VSR) \]

- Generally:

\[ CSR \subset VSR \]

- Cause: blind writes can repair violations caused by conflicts
Fault Tolerance

In terms of fault tolerance, the following schedule $s$ is not acceptable:

$$s = r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$$

...though it is serializable in VSR and CSR!
Recoverability RC

- $s$ is *recoverable*, if the following condition is fulfilled:

$$(T_i \text{ reads from } T_j \text{ in } s) \land (c_i \in s) \Rightarrow (c_j \rightarrow_s c_i)$$
Recoverability: Example

\[ s_1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)c_2w_1(z)c_1 \]

In \( s_1 \), \( T_2 \) reads data object \( y \) from \( T_1 \) but \( c_2 \) comes before \( c_1 \) \( \rightsquigarrow \) \( s_1 \) is not recoverable

\[ s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)c_1c_2 \]

\( s_2 \) is recoverable

But: Problems when aborting \( T_1 \) instead of \( c_1 \) (dirty read)!
Avoiding cascading aborts

Schedule \( s \) avoids cascading aborts ACA, if the following condition is fulfilled:

\[
(T_i \text{ reads } x \text{ from } T_j \text{ in } s) \Rightarrow (c_j \rightarrow_s r_i(x))
\]

\( \Rightarrow \) A transaction may only read data that has last been written by an already committed transaction.
Avoiding cascading aborts: Example

- Schedule $s_2$ from the last example does not belong into the class ACA.
- However, $s_3$ avoids cascading aborts:

$$s_3 = w_1(x)w_1(y)r_2(u)w_2(x)w_1(z)c_1r_2(y)w_2(y)c_2$$

- Therefore: $s_3 \in \text{ACA}$
## Problems with Before-Images

<table>
<thead>
<tr>
<th>DB Content</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1 ) (initial value)</td>
<td>( w_1(x \leftarrow 2) \ [ BF_{x,T_1} = 1 ] )</td>
</tr>
<tr>
<td>( x = 2 )</td>
<td>( w_1(x \leftarrow 2) ) [ ( BF_{x,T_1} = 1 ) ]</td>
</tr>
<tr>
<td>( x = 3 )</td>
<td>( w_2(x \leftarrow 3) \ [ BF_{x,T_2} = 2 ] )</td>
</tr>
<tr>
<td>( x = 3 )</td>
<td>( a_1 ) rollback of ( w_1(x \leftarrow 2) ) with ( BF_x := 1 ). Overwriting of ( T_2 ) has to be maintained</td>
</tr>
<tr>
<td>( x = 3 )</td>
<td>( a_2 ) rollback of ( w_2(x \leftarrow 3) ) with ( BF_x := ?? )</td>
</tr>
</tbody>
</table>
Strictness ST

- Schedule \( s \) is **strict**, if the following applies:

\[
(w_j(x) \rightarrow_s p_i(x) \land j \neq i) \Rightarrow (a_j \rightarrow_s p_i(x) \lor c_j \rightarrow_s p_i(x), (p \in \{r, w\}))
\]

\( \leadsto \) No "written" object of an incomplete transaction may be read or overwritten
Strictness: Example

- \( s_3 \not\in \text{ST} \)
- \( s_4 \) is strict, therefore \( s_4 \in \text{ST} \):

\[
s_4 = w_1(x)w_1(y)r_2(u)w_1(z)c_1 w_2(x)r_2(y)w_2(y)c_2
\]
Relation between the concepts

- VSR
- CSR
- RC
- ACA
- ST
- serial

- Often using strict locking protocols
- Acceptable using cascading repair