Transactional Information Systems:

Theory, Algorithms, and the Practice of Concurrency Control and Recovery

Gerhard Weikum and Gottfried Vossen

© 2002 Morgan Kaufmann
ISBN 1-55860-508-8

“Teamwork is essential. It allows you to blame someone else.” (Anonymous)
Part II: Concurrency Control

- 3 Concurrency Control: Notions of Correctness for the Page Model
- 4 Concurrency Control Algorithms
- 5 Multiversion Concurrency Control
- 6 Concurrency Control on Objects: Notions of Correctness
- 7 Concurrency Control Algorithms on Objects
- 8 Concurrency Control on Relational Databases
- 9 Concurrency Control on Search Structures
- 10 Implementation and Pragmatic Issues
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- 3.4 Correctness of Histories and Schedules
- 3.5 Herbrand Semantics of Schedules
- 3.6 Final-State Serializability
- 3.7 View Serializability
- 3.8 Conflict Serializability
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned

“Nothing is as practical as a good theory.” (Albert Einstein)
### Lost Update Problem

<table>
<thead>
<tr>
<th>P1</th>
<th>Time</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r (x)</code></td>
<td>1</td>
<td><code>r (x)</code></td>
</tr>
<tr>
<td><code>x := x + 100</code></td>
<td>2</td>
<td><code>x := x + 200</code></td>
</tr>
<tr>
<td><code>w (x)</code></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td><code>w (x)</code></td>
</tr>
</tbody>
</table>

/* x = 100 */

/* x = 200 */

/* x = 300 */

Observation: problem is the interleaving \( r_1(x) \) \( r_2(x) \) \( w_1(x) \) \( w_2(x) \)
Inconsistent Read Problem

<table>
<thead>
<tr>
<th>P1</th>
<th>Time</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>r (x)</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>x := x – 10</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>w (x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r (y)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>y := y + 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w (y)</td>
</tr>
</tbody>
</table>

sum := 0
r (x)
r (y)
sum := sum + x
sum := sum + y

“sees” wrong sum

Observations:
Problem is the interleaving \( r_2(x) \) \( w_2(x) \) \( r_1(x) \) \( r_1(y) \) \( r_2(y) \) \( w_2(y) \)
No problem with sequential execution
Dirty Read Problem

<table>
<thead>
<tr>
<th>P1</th>
<th>Time</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r (x)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x := x + 100</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>w (x)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>r (x)</td>
</tr>
<tr>
<td>failure &amp; rollback</td>
<td>5</td>
<td>x := x - 100</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>w (x)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

cannot rely on validity of previously read data

Observation: transaction rollbacks could affect concurrent transactions
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
  - 3.4 Correctness of Histories and Schedules
  - 3.5 Herbrand Semantics of Schedules
  - 3.6 Final-State Serializability
  - 3.7 View Serializability
  - 3.8 Conflict Serializability
  - 3.9 Commit Serializability
  - 3.10 An Alternative Criterion: Interleaving Specifications
  - 3.11 Lessons Learned
Partial Order

(a,a) in R (reflexive)

(a,b) in R AND (b,a) in R \rightarrow a=b (antisymmetric)

(a,b) in R, (b,c) in R \rightarrow (a,c) in R (transitive)

A prefix of a partial order <s is essentially obtained by omitting pieces from the end of a reachability chain. More precisely;

if \( s = (\text{op}(s), <s) \), then a prefix of \( s \) has the form \( s' = (\text{op}(s'), <s') \) such that:

1. \( \text{op}(s') \subseteq \text{op}(s) \)
2. \( <s' \subseteq <s \)
3. \( (\forall p \text{ in } \text{op}(s')) (\forall q \text{ in } \text{op}(s)) q <s p \rightarrow q \text{ in } \text{op}(s') \)
4. \( (\forall p, q \text{ in } \text{op}(s'), p <s q \rightarrow p <s' q \)
Schedules and Histories

Definition 3.1 (Schedules and histories):
Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(\text{op}_i, <_i)$ with $\text{op}_i$ denoting the operations of $t_i$ and $<_i$ their ordering.

(i) A **history** for $T$ is a pair $s=(\text{op}(s),<_s)$ s.t.
   (a) $\text{op}(s) \subseteq \bigcup_{i=1..n} \text{op}_i \cup \bigcup_{i=1..n} \{a_i, c_i\}$
   (b) for all $i$, $1 \leq i \leq n$: $c_i \in \text{op}(s) \iff a_i \notin \text{op}(s)$
   (c) $\bigcup_{i=1..n} <_i \subseteq <_s$
   (d) for all $i$, $1 \leq i \leq n$, and all $p \in \text{op}_i$: $p <_s c_i$ or $p <_s a_i$
   (e) for all $p, q \in \text{op}(s)$ s.t. at least one of them is a write and both access the same data item: $p <_s q$ or $q <_s p$

(ii) A **schedule** is a prefix of a history.

Definition 3.2 (Serial history):
A history $s$ is **serial** if for any two transactions $t_i$ and $t_j$ in $s$, where $i \neq j$, all operations from $t_i$ are ordered in $s$ before all operations from $t_j$ or vice versa.
Example Schedules and Notation

Example 3.4:

\[ r_1(x) \xrightarrow{} w_1(x) \xrightarrow{} c_1 \]
\[ r_1(z) \]
\[ r_2(x) \xrightarrow{} w_2(y) \xrightarrow{} c_2 \]
\[ r_3(z) \xrightarrow{} w_3(y) \xrightarrow{} c_3 \]
\[ w_3(z) \]

\[ \text{trans}(s) := \{ t_i \mid s \text{ contains step of } t_i \} \]
\[ \text{commit}(s) := \{ t_i \in \text{trans}(s) \mid c_i \in s \} \]
\[ \text{abort}(s) := \{ t_i \in \text{trans}(s) \mid a_i \in s \} \]
\[ \text{active}(s) := \text{trans}(s) - (\text{commit}(s) \cup \text{abort}(s)) \]

Example 3.6:

\[ r_1(x) r_2(z) r_3(x) w_2(x) w_1(x) r_3(y) r_1(y) w_1(y) w_2(z) w_3(z) c_1 a_3 \]
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- **3.4 Correctness of Histories and Schedules**
- 3.5 Herbrand Semantics of Schedules
- 3.6 Final-State Serializability
- 3.7 View Serializability
- 3.8 Conflict Serializability
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned
Correctness of Schedules

1. Define equivalence relation $\approx$ on set $S$ of all schedules.
2. “Good” schedules are those in the equivalence classes of serial schedules.

- Equivalence must be efficiently decidable.
- “Good” equivalence classes should be “sufficiently large”.

For the moment, disregard aborts: assume that all transactions are committed.
Activity

• What is an equivalence relation?

• List the three defining conditions!
Equivalence relation

A given binary relation ∼ on a set $X$ is said to be an equivalence relation if and only if it is reflexive, symmetric and transitive.
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- 3.4 Correctness of Histories and Schedules
- **3.5 Herbrand Semantics of Schedules**
- 3.6 Final-State Serializability
- 3.7 View Serializability
- 3.8 Conflict Serializability
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned
Herbrand Semantics of Schedules

Definition 3.3 (Herbrand Semantics of Steps):
For schedule $s$ the Herbrand semantics $H_s$ of steps $r_i(x), w_i(x) \in \text{op}(s)$ is:

(i) $H_s[r_i(x)] := H_s[w_j(x)]$ where $w_j(x)$ is the last write on $x$ in $s$ before $r_i(x)$.
(ii) $H_s[w_i(x)] := f_{ix}(H_s[r_i(y_1)], ..., H_s[r_i(y_m)])$ where the $r_i(y_j), 1 \leq j \leq m$, are all read operations of $t_i$ that occur in $s$ before $w_i(x)$ and $f_{ix}$ is an uninterpreted $m$-ary function symbol.

Definition 3.4 (Herbrand Universe):
For data items $D = \{x, y, z, ...\}$ and transactions $t_i, 1 \leq i \leq n$, the Herbrand universe $HU$ is the smallest set of symbols s.t.

(i) $f_{0x}( ) \in HU$ for each $x \in D$ where $f_{0x}$ is a constant, and
(ii) if $w_i(x) \in \text{op}_i$ for some $t_i$, there are $m$ read operations $r_i(y_1), ..., r_i(y_m)$ that precede $w_i(x)$ in $t_i$, and $v_1, ..., v_m \in HU$, then $f_{ix}(v_1, ..., v_m) \in HU$.

Definition 3.5 (Schedule Semantics):
The Herbrand semantics of a schedule $s$ is the mapping $H[s]: D \rightarrow HU$ defined by $H[s](x) := H_s[w_i(x)]$, where $w_i(x)$ is the last operation from $s$ writing $x$, for each $x \in D$. 
Herbrand Semantics: Example

\[ s = w_0(x) \, w_0(y) \, c_0 \, r_1(x) \, r_2(y) \, w_2(x) \, w_1(y) \, c_2 \, c_1 \]

\[
H_s[w_0(x)] = f_{0x}( )
\]
\[
H_s[w_0(y)] = f_{0y}( )
\]
\[
H_s[r_1(x)] = H_s[w_0(x)] = f_{0x}( )
\]
\[
H_s[r_2(y)] = H_s[w_0(y)] = f_{0y}( )
\]
\[
H_s[w_2(x)] = f_{2x}(H_s[r_2(y)]) = f_{2x}(f_{0y}( ))
\]
\[
H_s[w_1(y)] = f_{1y}(H_s[r_1(x)]) = f_{1y}(f_{0x}( ))
\]

\[
H[s](x) = H_s[w_2(x)] = f_{2x}(f_{0y}( ))
\]
\[
H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}( ))
\]
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- 3.4 Correctness of Histories and Schedules
- 3.5 Herbrand Semantics of Schedules

- **3.6 Final-State Serializability**
- 3.7 View Serializability
- 3.8 Conflict Serializability
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned
Definition 3.6 (Final State Equivalence):
Schedules s and s‘ are called **final state equivalent**, denoted \( s \approx_f s' \), if \( \text{op}(s) = \text{op}(s') \) and \( H[s] = H[s'] \).

Example a:

\[
\begin{align*}
s &= r_1(x) \ r_2(y) \ w_1(y) \ r_3(z) \ w_3(z) \ r_2(x) \ w_2(z) \ w_1(x) \\
s' &= r_3(z) \ w_3(z) \ r_2(y) \ r_2(x) \ w_2(z) \ r_1(x) \ w_1(y) \ w_1(x)
\end{align*}
\]

\[
\begin{align*}
H[s](x) &= H_s[w_1(x)] = f_{1x}(f_{0x}( )) = H_{s'}[w_1(x)] = H[s'](x) \\
H[s](y) &= H_s[w_1(y)] = f_{1y}(f_{0x}( )) = H_{s'}[w_1(y)] = H[s'](y) \\
H[s](z) &= H_s[w_2(z)] = f_{2z}(f_{0x}( ), f_{0y}( )) = H_{s'}[w_2(z)] = H[s'](z)
\end{align*}
\]

\( \Rightarrow s \approx_f s' \)

Example b:

\[
\begin{align*}
s &= r_1(x) \ r_2(y) \ w_1(y) \ w_2(y) \\
s' &= r_1(x) \ w_1(y) \ r_2(y) \ w_2(y)
\end{align*}
\]

\[
\begin{align*}
H[s](y) &= H_s[w_2(y)] = f_{2y}(f_{0y}( )) \\
H[s']'(y) &= H_{s'}[w_2(y)] = f_{2y}(f_{1y}(f_{0x}( )))
\end{align*}
\]

\( \Rightarrow \neg (s \approx_f s') \)
Definition 3.7 (Reads-from Relation; Useful, Alive, and Dead Steps):
Given a schedule \( s \), extended with an initial and a final transaction, \( t_0 \) and \( t_\infty \).

(i) \( r_j(x) \) reads \( x \) in \( s \) from \( w_i(x) \) if \( w_i(x) \) is the last write on \( x \) s.t. \( w_i(x) <_s r_j(x) \).

(ii) The reads-from relation of \( s \) is
\[
RF(s) := \{ (t_i, x, t_j) \mid \text{an } r_j(x) \text{ reads } x \text{ from a } w_i(x) \}.
\]

(iii) Step \( p \) is directly useful for step \( q \), denoted \( p \rightarrow q \), if \( q \) reads from \( p \), or \( p \) is a read step and \( q \) is a subsequent write step of the same transaction. 
\( \rightarrow^* \), the “useful” relation, denotes the reflexive and transitive closure of \( \rightarrow \).

(iv) Step \( p \) is alive in \( s \) if it is useful for some step from \( t_\infty \), and dead otherwise.

(v) The live-reads-from relation of \( s \) is
\[
LRF(s) := \{ (t_i, x, t_j) \mid \text{an alive } r_j(x) \text{ reads } x \text{ from } w_i(x) \}.
\]

Example 3.7:
\[
s = r_1(x) \ r_2(y) \ w_1(y) \ w_2(y)
\]
\[
s' = r_1(x) \ w_1(y) \ r_2(y) \ w_2(y)
\]
\[
RF(s) = \{ (t_0, x, t_1), (t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty) \}
\]
\[
RF(s') = \{ (t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty) \}
\]
\[
LRF(s) = \{ (t_0, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty) \}
\]
\[
LRF(s') = \{ (t_0, x, t_1), (t_1, y, t_2), (t_0, x, t_\infty), (t_2, y, t_\infty) \}
\]
Directly Useful

\[ s' = w_0(x) w_0(y) r_1(x) w_1(y) r_2(y) w_2(y) r_\infty(x) r_\infty(y) \]

\[ \text{LRF}(s') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\} \]
Final-State Serializability

**Theorem 3.1:**
For schedules $s$ and $s'$ the following statements hold.
(i) $s \approx_ff s'$ if and only if $\operatorname{op}(s) = \operatorname{op}(s')$ and $\operatorname{LRF}(s) = \operatorname{LRF}(s')$.
(ii) For $s$ let the step graph $D(s) = (V, E)$ be a directed graph with vertices $V := \operatorname{op}(s)$ and edges $E := \{(p, q) \mid p \rightarrow q\}$, and the reduced step graph $D_1(s)$ be derived from $D(s)$ by removing all vertices that correspond to dead steps. Then $\operatorname{LRF}(s) = \operatorname{LRF}(s')$ if and only if $D_1(s) = D_1(s')$.

**Corollary 3.1:**
Final-state equivalence of two schedules $s$ and $s'$ can be decided in time that is polynomial in the length of the two schedules.

**Definition 3.8 (Final State Serializability):**
A schedule $s$ is **final state serializable** if there is a serial schedule $s'$ s.t. $s \approx_ff s'$. $\text{FSR}$ denotes the class of all final-state serializable histories.
Final State Equivalence example

\[ s = r_1(x) \ r_2(y) \ w_1(y) \ r_3(z) \ w_3(z) \ r_2(x) \ w_2(z) \ w_1(x) \ c_1 \ c_2 \ c_3 \]

\[ s' = r_3(z) \ w_3(z) \ r_2(y) \ r_2(x) \ w_2(z) \ r_1(x) \ w_1(y) \ w_1(x) \ c_1 \ c_2 \ c_3 \]

LRF
FSR: Example 3.9

$s = r_1(x) \ r_2(y) \ w_1(y) \ w_2(y)$

$D(s):$

$D(s{'})$: 

$D_1(s)$ different than $D_1(s{'})$
Critique of FSR

- Using semantics
  - Commutativity. Suppose we know that both t1 and t2 perform $X=X+1$ via $R(X)W(X)$ then $t1t2$ is the same as $t2t1$ yet the two are not final state equivalent.
  - Copiers. Suppose we know that t3 and t4 simply copy their arguments via $r(X)w(Y)$ and $r(Y)w(X)$. Then, $r1(X)r3(X)w3(Y)w2(X)r4(Y)w4(X)r5(X)w5(Z)w1(Z)$ is equivalent to $t5t1t3t2t4$ although it’s not final state serializable. key- r5 read X0.

- Not using integrity constraints that can lead to a liberal notion.
- Sometimes a history is final state equivalent to a serial history of a subset of the transactions.
- Final state equivalence may “switch” transactions order as in $r1(X)r2(X)w2(X)r3(Y)w3(Y)w1(Y)$ which is only final state equivalent to $t3t1t2$.
- Does not account for infinite schedules as in a set of transactions performing each $X=X+1$ and $Y=Y+1$, interleaving in “the middle”, maintaining each IC = $\{X=Y\}$.
- The “new” history may not reflect reality as usually we do not deal with straight-line programs.
Examples

\[ A_1 : R(x) \quad W(z) \]
\[ A_2 : \quad W(x) \]
\[ A_3 : \quad R(x)W(y) \]
\[ A_4 : \quad R(y)W(x) \]
\[ A_5 : \quad R(x)W(z) \]

Copier
A5A1A3A2A4

Equivalent to just A2
\[ X++ X++ \text{ Y}++X++Y++X++Y++X++Y++X++Y++\ldots \]

Does not account for infinite schedules as in a set of transactions performing each \(X=X+1\) and \(Y=Y+1\), interleaving in “the middle”, maintaining each IC = \{\(X=Y\}\).
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

• 3.2 Canonical Synchronization Problems
• 3.3 Syntax of Histories and Schedules
• 3.4 Correctness of Histories and Schedules
• 3.5 Herbrand Semantics of Schedules
• 3.6 Final-State Serializability
• 3.7 View Serializability
• 3.8 Conflict Serializability
• 3.9 Commit Serializability
• 3.10 An Alternative Criterion: Interleaving Specifications
• 3.11 Lessons Learned
Canonical Anomalies Reconsidered

• Lost update anomaly:
  \[ L = r_1(x) \; r_2(x) \; w_1(x) \; w_2(x) \; c_1 \; c_2 \]

  \[ \rightarrow \text{history is not FSR} \]

  \[ \text{LRF}(L) = \{(t_0,x,t_2), (t_2,x,t_\infty)\} \]
  \[ \text{LRF}(t_1 \; t_2) = \{(t_0,x,t_1), (t_1,x,t_2), (t_2,x,t_\infty)\} \]
  \[ \text{LRF}(t_2 \; t_1) = \{(t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\} \]

• Inconsistent read anomaly:
  \[ I = r_2(x) \; w_2(x) \; r_1(x) \; r_1(y) \; r_2(y) \; w_2(y) \; c_1 \; c_2 \]

  \[ \rightarrow \text{history is FSR} ! \]

  \[ \text{LRF}(I) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\} \]
  \[ \text{LRF}(t_1 \; t_2) = \{(t_0,x,t_1), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\} \]
  \[ \text{LRF}(t_2 \; t_1) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\} \]

Observation: (Herbrand) semantics of all read steps matters!
**Definition 3.9 (View Equivalence):**
Schedules $s$ and $s'$ are **view equivalent**, denoted $s \cong_v s'$, if the following hold:
(i) $\text{op}(s) = \text{op}(s')$
(ii) $H[s] = H[s']$
(iii) $H_s[p] = H_{s'}[p]$ for all (read or write) steps

**Theorem 3.2:**
For schedules $s$ and $s'$ over the same set of transactions the following statements hold.
(i) $s \cong_v s'$ iff $\text{op}(s) = \text{op}(s')$ and $\text{RF}(s) = \text{RF}(s')$  \(\text{proof follows} \leftarrow\)
(ii) $s \cong_v s'$ iff $D(s) = D(s')$

**Corollary 3.2:**
View equivalence of two schedules $s$ and $s'$ can be decided in time that is polynomial in the length of the two schedules.

**Definition 3.10 (View Serializability):**
A schedule $s$ is **view serializable** if there exists a serial schedule $s'$ s.t. $s \cong_v s'$. VSR denotes the class of all view-serializable histories.
Theorem 3.2. ((i), $\leftarrow$)

- Consider $s$ and $s'$ such that $\text{op}(s)=\text{op}(s')$ and $\text{RF}(s)=\text{RF}(s')$. Let $w_{i1}(x_{u1}), \ldots, w_{im}(x_{um})$ be the write operations of $s$ in the order they were performed. $w_{ij}(x_{uj})$ is a write operation of transaction $i_j$ to variable $x_{uj}$.

- By induction, for $j=1, \ldots, m$ we claim the same value was written by each $w_{ij}$ in $s$ and $s'$.

- **Basis**, $j=1$. The first write operation can only depend on values read from $D_0$, the initial DB, or based on no input. Locate $w_{i1}(x_{u1})$ in $s'$. Because $\text{RF}(s)=\text{RF}(s')$, in $s'$ also the values read prior to $w_{i1}(x_{u1})$ by $t_{i1}$ are identical to those in $s$, and so the same value will be written.

- **Induction**, $j>1$. Consider the write operation $w_{ij}(x_{uj})$. The values in read operations on which it depends on have already been produced in steps $m<j$. By induction, the same values are produced in $s'$. As $\text{RF}(s)=\text{RF}(s')$, the same value will be produced in by $w_{ij}(x_{uj})$ in $s'$. 
Inconsistent Read Reconsidered

• Inconsistent read anomaly:
  \[ I = r_2(x) \, w_2(x) \, r_1(x) \, r_1(y) \, r_2(y) \, w_2(y) \, c_1 \, c_2 \]

→ history is not VSR!

\[
\begin{align*}
RF(I) &= \{(t_0,x,t_2), (t_2,x,t_1), (t_0,y,t_1), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\} \\
RF(t_1 \, t_2) &= \{(t_0,x,t_1), (t_0,y,t_1), (t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\} \\
RF(t_2 \, t_1) &= \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_1), (t_2,y,t_1), (t_2,x,t_\infty), (t_2,y,t_\infty)\}
\end{align*}
\]

**Observation:** *VSR properly captures our intuition*
Relationship Between VSR and FSR

Theorem 3.3:
VSR ⊂ FSR.

\[ s = w_1(x) r_2(x) r_2(y) w_1(y) c_1 c_2 \]
shows proper inclusion

s is in FSR but not in VSR

Theorem 3.4:
Let s be a history without dead steps. Then \( s \in \text{VSR} \) iff \( s \in \text{FSR} \).
On the Complexity of Testing VSR

Theorem 3.5:
The problem of deciding for a given schedule $s$ whether $s \in VSR$ holds is NP-complete.
Properties of VSR

**Definition 3.11 (Monotone Classes of Histories)**
Let $s$ be a schedule and $T \subseteq \text{trans}(s)$. $\Pi_T(s)$ denotes the projection of $s$ onto $T$. A class $E$ of histories is called **monotone** if the following holds:
- if $s$ is in $E$, then $\Pi_T(s)$ is in $E$ for each $T \subseteq \text{trans}(s)$.

VSR is not monotone.

**Example:**

$s = w_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \ w_3(x) \ w_3(y) \ c_3$

$\Pi_{\{t_1, t_2\}}(s) = w_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1$

$\rightarrow \in \ VSR$
$\rightarrow \notin \ VSR$
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- 3.4 Correctness of Histories and Schedules
- 3.5 Herbrand Semantics of Schedules
- 3.6 Final-State Serializability
- 3.7 View Serializability
- **3.8 Conflict Serializability**
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned
Conflict Serializability

Definition 3.12 (Conflicts and Conflict Relations): Let s be a schedule, t, t' ∈ trans(s), t ≠ t'.
(i) Two data operations p ∈ t and q ∈ t' are in conflict in s if they access the same data item and at least one of them is a write.
(ii) {(p, q)} | p, q are in conflict and p <s q} is the conflict relation of s.

Definition 3.13 (Conflict Equivalence): Schedules s and s' are conflict equivalent, denoted s ≈c s', if op(s) = op(s') and conf(s) = conf(s').

Definition 3.14 (Conflict Serializability): History s is conflict serializable if there is a serial history s' s.t. s ≈c s'. CSR denotes the class of all conflict serializable histories.

Example a: r₁(x) r₂(x) r₁(z) w₁(x) w₂(y) r₃(z) w₃(y) c₁ c₂ w₃(z) c₃ → ∈ CSR
Example b: r₂(x) w₂(x) r₁(x) r₁(y) r₂(y) w₂(y) c₁ c₂ → ∉ CSR
Properties of CSR

Theorem 3.8: proof follows
CSR ⊂ VSR

Example: \( s = w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1 w_3(x) w_3(y) c_3 \)
s \(\in\) VSR, but s \(\notin\) CSR.

Theorem 3.9:
(i) CSR is monotone.
(ii) \( s \in CSR \iff \Pi_T(s) \in VSR \) for all \( T \subseteq \text{trans}(s) \)
    (i.e., CSR is the largest monotone subset of VSR).
Proof of Theorem 3.8

1. Suppose history $s$ is in CSR, $s'$ a conflict equivalent serial history.
2. Let $(t_i, x, t_j)$ be in $RF(s)$ but not in $RF(s')$.
3. Thus $(w_i(x), r_j(x))$ in $conf(s)$ and in $conf(s')$ (by 1).
4. However, $(t_k, x, r_j)$ in $RF(s')$, $(w_k(x), r_j(x))$ in $conf(s')$.
5. Operations $w_i(x)$ and $w_k(x)$ conflict in $s$. Two cases:
   1. $w_k(x)$ precedes $w_i(x)$ in $s$. So, $(w_k(x), w_i(x))$ in $conf(s)=conf(s')$. So, $w_k(x)$ precedes $w_i(x)$ in $s'$. This implies, by 4, that $r_j(x)$ precedes $w_i(x)$ in $s'$. But then $(r_j(x), w_i(x))$ is in $conf(s')=conf(s)$, contradicting 3.
   2. $w_i(x)$ precedes $w_k(x)$ in $s$. So, $r_j(x)$ precedes $w_k(x)$ in $s$ because of 2. So, $(r_j(x), w_k(x))$ in $conf(s)=conf(s')$. This contradicts 4.
6. We conclude $RF(s)=RF(s')$. So, $s$ is view serializable.
7. $w_1(x) w_2(x) w_2(y) c_2 w_1(y) c_1 w_3(x) w_3(y) c_3$ is view equivalent to $t_1 t_2 t_3$ but is not in CSR due to $w_1(x) w_2(x) w_2(y) w_1(y)$. 


Activity

• What is a directed graph?

• Think of ways to associate a graph with a schedule!
Definition 3.15 (Conflict Graph):
Let $s$ be a schedule. The conflict graph $G(s) = (V, E)$ is a directed graph with vertices $V := \text{commit}(s)$ and edges $E := \{(t, t') \mid t \neq t' \text{ and there are steps } p \in t, q \in t' \text{ with } (p, q) \in \text{conf}(s)\}$.

Theorem 3.10:
Let $s$ be a history. Then $s \in \text{CSR}$ iff $G(s)$ is acyclic. Proof follows.

Corollary 3.4:
Testing if a schedule is in CSR can be done in time polynomial to the schedule's number of transactions.

Example 3.12:
$s = r_1(y) \ r_3(w) \ r_2(y) \ w_1(y) \ w_1(x) \ w_2(x) \ w_2(z) \ w_3(x) \ \text{c}_1 \ \text{c}_3 \ \text{c}_2$

$G(s):$

```
t1 --> t2
     \      \\
     |      |
     v      v
 t3 --> t3
```
Activity

• What is a characterization (in a mathematical sense)?

• How do you prove a necessary and sufficient condition?

• What needs to be shown for the serializability theorem?
Proof of the Conflict-Graph Theorem

(i) Let \( s \) be a schedule in CSR. So there is a serial schedule \( s' \) with \( \text{conf}(s) = \text{conf}(s') \).

Now assume that \( G(s) \) has a cycle \( t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow t_1 \).

This implies that there are pairs \((p_1, q_2), (p_2, q_3), ..., (p_k, q_1)\)

with \( p_i \in t_i, q_i \in t_i, p_i <_s q_(i+1), \) and \( p_i \) in conflict with \( q_(i+1) \).

Because \( s' \approx_c s \), it also implies that \( p_i <_{s'} q_(i+1) \).

Because \( s' \) is serial, we obtain \( t_i <_{s'} t_(i+1) \) for \( i = 1, ..., k-1, \) and \( t_k <_{s'} t_1 \).

By transitivity we infer \( t_1 <_{s'} t_2 \) and \( t_2 <_{s'} t_1 \), which is impossible.

This contradiction shows that the initial assumption is wrong. So \( G(s) \) is acyclic.

(ii) Let \( G(s) \) be acyclic. So it must have at least one source node.

The following topological sort produces a total order \(<\) of transactions:

a) start with a source node (i.e., a node without incoming edges),

b) remove this node and all its outgoing edges,

c) iterate a) and b) until all nodes have been added to the sorted list.

The total transaction ordering order \(<\) preserves the edges in \( G(s) \);

therefore it yields a serial schedule \( s' \) for which \( s' \approx_c s \).
Commutativity and Ordering Rules

Commutativity rules: 2 directions!

C1: \( r_i(x) \ r_j(y) \sim r_j(y) \ r_i(x) \) if \( i \neq j \)
C2: \( r_i(x) \ w_j(y) \sim w_j(y) \ r_i(x) \) if \( i \neq j \) and \( x \neq y \)
C3: \( w_i(x) \ w_j(y) \sim w_j(y) \ w_i(x) \) if \( i \neq j \) and \( x \neq y \)

Ordering rule:

C4: \( o_i(x), \ p_j(y) \) unordered \( \sim \) \( o_i(x) \ p_j(y) \)
no conflict: if \( x \neq y \) or both \( o \) and \( p \) are reads

Example for transformations of schedules:

\[
\begin{align*}
s &= w_1(x) \ r_2(x) \ w_1(y) \ w_1(z) \ r_3(z) \ w_2(y) \ w_3(y) \ w_3(z) \\
\sim&>[C2] \quad w_1(x) \ w_1(y) \ r_2(x) \ w_1(z) \ w_2(y) \ r_3(z) \ w_3(y) \ w_3(z) \\
\sim&>[C2] \quad w_1(x) \ w_1(y) \ w_1(z) \ r_2(x) \ w_2(y) \ r_3(z) \ w_3(y) \ w_3(z) \\
= \quad t_1 \ t_2 \ t_3
\end{align*}
\]
Commutativity-based Reducibility

**Definition 3.16 (Commutativity Based Equivalence):**
Schedules s and s‘ s.t. op(s)=op(s‘) are **commutativity based equivalent**, denoted s ~* s‘, if s can be transformed into s‘ by applying rules C1, C2, C3, C4 finitely many times.

**Theorem 3.11:**
Let s and s‘ be schedules s.t. op(s)=op(s‘). Then $s \approx_c s'$ iff $s \sim^* s'$.

**Definition 3.17 (Commutativity Based Reducibility):**
History s is **commutativity-based reducible** if there is a serial history s‘ s.t. s ~* s‘.

**Corollary 3.5:**
History s is commutativity-based reducible iff $s \in \text{CSR}$. 
Proof of Theorem 3.11

(⇒) Suppose $s \approx_c s'$. We show how to transform $s'$ to $s$. Consider the last operation $o$ in $s$. Let $s' = o_1 \ldots o_k$. Operation $o$ is $o_j$. If $j = k$ proceed to (*). Otherwise exchange $o_j$ and $o_{j+1}$. This can be done because $o_{j+1}$ is not in the same transaction as $o_j$ and it cannot conflict with $o_{j+1}$ for otherwise there will be different conflict relations in $s$ and $s'$. We can perform such exchanges until $o$ becomes the last operation in the (modified) $s'$. (*)

Now, eliminate the same last operation from $s$ and the modified $s'$ and repeat the process until no operation remains in either schedule. The sequence of exchanges is the desired one.
Proof of Theorem 3.11

(⟺) Suppose \( s \sim^* s' \). By induction on the number of transformation steps. Basis: 0 steps. Then \( s = s' \). Induction: suppose the claim holds for less than \( n \) steps, consider \( n \) steps, \( n > 0 \). Consider the first transformation step. It does not change the order of steps within a transaction. Also, the 2 exchanged operations do not conflict (\( * \)). By hypothesis, the resulting schedules are conflict equivalent and thus, by (\( * \)), \( s \approx_c s' \).
Order Preserving Conflict Serializability

Definition 3.18 (Order Preservation):
History $s$ is order preserving conflict serializable if it is conflict equivalent to a serial history $s'$ and for all $t, t' \in \text{trans}(s)$: if $t$ completely precedes $t'$ in $s$, then the same holds in $s'$. OCSR denotes the class of all schedules with this property.

Theorem 3.12:
OCSR $\subset$ CSR.

Example 3.13:
$s = w_1(x) \, r_2(x) \, c_2 \, w_3(y) \, c_3 \, w_1(y) \, c_1$
$\rightarrow \in$ CSR
$\rightarrow \notin$ OCSR
Commit-order Preserving Conflict Serializability

**Definition 3.19 (Commit Order Preservation):**

History $s$ is **commit order preserving conflict serializable** if for all $t_i, t_j \in \text{trans}(s)$: if there are $p \in t_i, q \in t_j$ with $(p,q) \in \text{conf}(s)$ then $c_i <_s c_j$. COCSR denotes the class of all histories with this property.

**Theorem 3.13:**

COCSR $\subseteq$ CSR. Because a cycle in $\text{SG}(s)$ is impossible.

**Theorem 3.14:**

proof follows

History $s$ is in COCSR iff there is a serial schedule $s'$ s.t. $s \approx_c s'$ and for all $t_i, t_j \in \text{trans}(s)$: $t_i <_s t_j \iff c_i <_s c_j$.

**Theorem 3.15:**

proof follows

COCSR $\subset$ OCSR.

**Example:**

$s = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1$

$\rightarrow \in$ OCSR

$\rightarrow \notin$ COCSR
Theorem 3.14. (\(\rightarrow\))

- History \(s\) in COCSR and so is CSR, thus \(G(s)\) acyclic.
- Modify \(G(s)\): add edge from the first-to-commit “transaction node” to the second-to-commit one and so on.
- We claim the modified graph is still acyclic. Consider adding the new edges one at a time. Suppose at some point we have a cycle.
- But both edge types, original and new, from \(t_1\) to \(t_2\) mean \(t_1\) committed in \(s\) prior to \(t_2\). A cycle is impossible.
- Next, sort the modified graph. The result is the desired serial schedule \(s' \quad t <_s t' \quad \rightarrow \quad ct <_s ct'\)
Proof of Theorem 3.14 ($\iff$)

1. Suppose there exists a conflict equivalent schedule $s'$ satisfying $(*$) for all $ti, tj \in \text{trans}(s): ti <_{s'} tj \iff ci <_{s} cj$.

2. Suppose $s$ is not COCSR. There must be conflicting operations $p$ in $ti$, $q$ in $tj$ in $s$ such that $(p,q)$ in $\text{conf}(s)$ but $(*)$ $cj <_{s} ci$.

3. Transaction $ti$ must precede transaction $tj$ in $s'$ as $\text{conf}(s)=\text{conf}(s')$. By $(*$), $ci <_{s} cj$. This contradicts $(*$).

4. So, $s$ is COCSR.
Theorem 3.15 (Alternative Proof)

- Suppose $h$ is in COCSR.
- Consider two transactions $t_i$ and $t_j$ such that $c_i < c_j$.
- If they have conflicting operations do nothing.
- Otherwise, add a read at the beginning of $t_i$ and a write to the end of $t_j$ to a unique item $I_{ij}$. An extra conflict is created.
- The “commit property” that whenever there is a conflict, the commits adhere to it, is preserved.
- The new history $h'$ is also CSR by the “commit property”.
- Consider a serialization order $s'$ for $h'$.
- If $c_i$ precedes $c_j$ in $s$ then $t_i$ precedes $t_j$ in $s'$ (because of the extra conflicts).
- In particular, if all operations of $t_i$ precede all operations of $t_j$ in $s$, then $c_i$ precedes $c_j$ in $s$, then $t_i$ precedes $t_j$ in $s'$ and so $s$ is OCSR.
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

• 3.2 Canonical Synchronization Problems
• 3.3 Syntax of Histories and Schedules
• 3.4 Correctness of Histories and Schedules
• 3.5 Herbrand Semantics of Schedules
• 3.6 Final-State Serializability
• 3.7 View Serializability
• 3.8 Conflict Serializability

• 3.9 Commit Serializability
• 3.10 An Alternative Criterion: Interleaving Specifications
• 3.11 Lessons Learned
Commit Serializability

Definition 3.20 (Closure Properties of Schedule Classes):
Let E be a class of schedules.
For schedule s let CP(s) denote the projection $\Pi_{commit(s)}(s)$.
E is **prefix-closed** if the following holds: $s \in E \iff p \in E$ for each prefix of s.
E is **commit-closed** if the following holds: $s \in E \Rightarrow CP(s) \in E$.

Theorem 3.16:
CSR is prefix-commit-closed, i.e., prefix-closed and commit-closed.

Definition 3.21 (Commit Serializability):
Schedule s is **commit-\(\Theta\)-serializable** if CP(p) is \(\Theta\)-serializable for each prefix p of s, where \(\Theta\) can be FSR, VSR, or CSR.
The resulting classes of commit-\(\Theta\)-serializable schedules are denoted CMFSR, CMVSR, and CMCSR.

Theorem 3.17:
(i) CMFSR, CMVSR, CMCSR are prefix-commit-closed.
(ii) CMCSR $\subset$ CMVSR $\subset$ CMFSR
VSR is not prefix commit closed

\[ s = w_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \ w_3(x) \ w_3(y) \]

\( s \) is in VSR.

Consider prefix \( s' = \text{CP}(s') = w_1(x) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \)

\( s' \) is view equivalent neither to \( t_1 \ t_2 \) nor to \( t_2 \ t_1 \) because \( t_\infty \) reads a different final state.

So, VSR is not prefix commit closed.
FSR is not prefix commit closed

\[ s = w_1(x) \; w_2(x) \; w_2(y) \; c_2 \; w_1(y) \; c_1 \; w_3(x) \; w_3(y) \]

s is in VSR, same as t_1 \; t_2 \; t_3, and so in FSR.
Consider prefix \( s' = CP(s') = w_1(x) \; w_2(x) \; w_2(y) \; c_2 \; w_1(y) \; c_1 \)
s’ is final state equivalent neither to t_1t_2 nor to t_2t_1.
So, FSR is not prefix commit closed.
Swap $s_3$ and $s_5$
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

• 3.2 Canonical Synchronization Problems
• 3.3 Syntax of Histories and Schedules
• 3.4 Correctness of Histories and Schedules
• 3.5 Herbrand Semantics of Schedules
• 3.6 Final-State Serializability
• 3.7 View Serializability
• 3.8 Conflict Serializability
• 3.9 Commit Serializability
• 3.10 An Alternative Criterion: Interleaving Specifications
• 3.11 Lessons Learned
Interleaving Specifications: Motivation

**Example:** all transactions known in advance

- Transfer transactions on checking accounts a and b and savings account c:
  - $t_1 = r_1(a) \ w_1(a) \ r_1(c) \ w_1(c)$
  - $t_2 = r_2(b) \ w_2(b) \ r_2(c) \ w_2(c)$
- Balance transaction:
  - $t_3 = r_3(a) \ r_3(b) \ r_3(c)$
- Audit transaction:
  - $t_4 = r_4(a) \ r_4(b) \ r_4(c) \ w_4(z)$

Possible schedules:

1. $r_1(a) \ w_1(a) \ r_2(b) \ w_2(b) \ r_2(c) \ w_2(c) \ r_1(c) \ w_1(c)$ → $\in$ CSR
2. $r_1(a) \ w_1(a) \ r_3(a) \ r_3(b) \ r_3(c) \ r_1(c) \ w_1(c)$ → $\notin$ CSR
3. $r_1(a) \ w_1(a) \ r_2(b) \ w_2(b) \ r_1(c) \ r_2(c) \ w_2(c) \ w_1(c)$ → $\notin$ CSR
4. $r_1(a) \ w_1(a) \ r_4(a) \ r_4(b) \ r_4(c) \ w_4(z) \ r_1(c) \ w_1(c)$ → $\notin$ CSR

**Observations:** application may tolerate non-CSR schedules

*a priori knowledge of all transactions impractical*
Indivisible Units

Definition 3.22 (Indivisible Units):
Let $T=\{t_1, ..., t_n\}$ be a set of transactions. For $t_i, t_j \in T$, $t_i \neq t_j$, an indivisible unit of $t_i$ relative to $t_j$ is a sequence of consecutive steps of $t_i$ s.t. no operations of $t_j$ are allowed to interleave with this sequence.

$IU(t_i, t_j)$ denotes the ordered sequence of indivisible units of $t_i$ relative to $t_j$.
$IU_k(t_i, t_j)$ denotes the $k^{th}$ element of $IU(t_i, t_j)$.

Example 3.14:
$t_1 = r_1(x) w_1(x) w_1(z) r_1(y)$
$t_2 = r_2(y) w_2(y) r_2(x)$
$t_3 = w_3(x) w_3(y) w_3(z)$

$IU(t_1, t_2) = < [r_1(x) w_1(x)], [w_1(z) r_1(y)] >$
$IU(t_1, t_3) = < [r_1(x) w_1(x)], [w_1(z)], [r_1(y)] >$
$IU(t_2, t_1) = < [r_2(y)], [w_2(y) r_2(x)] >$
$IU(t_2, t_3) = < [r_2(y) w_2(y)], [r_2(x)] >$
$IU(t_3, t_1) = < [w_3(x) w_3(y)], [w_3(z)] >$
$IU(t_3, t_2) = < [w_3(x) w_3(y)], [w_3(z)] >$

Example 3.15:
$s_1 = r_2(y) r_1(x) w_1(x) w_2(y) r_2(x) w_1(z) w_3(x) w_3(y) r_1(y) w_3(z) \rightarrow$ respects all IUs

$s_2 = r_1(x) r_2(y) w_2(y) w_1(x) r_2(x) w_1(z) r_1(y) \rightarrow$ violates $IU_1(t_1, t_2)$ and $IU_2(t_2, t_1)$

but is conflict equivalent to an allowed schedule
Relatively Serializable Schedules

**Definition 3.23 (Dependence of Steps):**
Step q directly **depends on** step p in schedule s, denoted p~>q, if p <s q and either p, q belong to the same transaction t and p <t q or p and q are in conflict. ~>* denotes the reflexive and transitive closure of ~>.

**Definition 3.24 (Relatively Serial Schedule):**
s is **relatively serial** if for all transactions tᵢ, tⱼ: if q ∈ tⱼ is interleaved with some IUₖ(tᵢ, tⱼ), then there is no operation p ∈ IUₖ(tᵢ, tⱼ) s.t. p~>* q or q~>* p

**Example 3.16:**

\[ s₃ = r₁(x) \text{ } r₂(y) \text{ } w₁(x) \text{ } w₂(y) \text{ } w₃(x) \text{ } w₁(z) \text{ } w₃(y) \text{ } r₂(x) \text{ } r₁(y) \text{ } w₃(z) \]

**Definition 3.25 (Relatively Serializable Schedule):**
s is **relatively serializable** if it is conflict equivalent to a relatively serial schedule.

**Example 3.17:**

\[ s₄ = r₁(x) \text{ } r₂(y) \text{ } w₂(y) \text{ } w₁(x) \text{ } w₃(x) \text{ } r₂(x) \text{ } w₁(z) \text{ } w₃(y) \text{ } r₁(y) \text{ } w₃(z) \]
Relative Serialization Graph

Definition 3.26 (Push Forward and Pull Backward):
Let $IU_k(t_i, t_j)$ be an IU of $t_i$ relative to $t_j$. For an operation $p_i \in IU_k(t_i, t_j)$ let
(i) $F(p_i, t_j)$ be the last operation in $IU_k(t_i, t_j)$ and
(ii) $B(p_i, t_j)$ be the first operation in $IU_k(t_i, t_j)$.

Definition 3.27 (Relative Serialization Graph):
The relative serialization graph $RSG(s) = (V, E)$ of schedule $s$ is a graph with vertices $V := op(s)$ and edge set $E \subseteq V \times V$ containing four types of edges:
(i) for consecutive operations $p, q$ of the same transaction $(p, q) \in E$ (I-edge)
(ii) if $p \sim>* q$ for $p \in t_i, q \in t_j, t_i \neq t_j$, then $(p, q) \in E$ (D-edge)
(iii) if $(p, q)$ is a D-edge with $p \in t_i, q \in t_j$, then $(F(p, t_j), q) \in E$ (F-edge)
(iv) if $(p,q)$ is a D-edge with $p \in t_i, q \in t_j$, then $(p, B(q, t_i)) \in E$ (B-edge)

Theorem 3.18:
A schedule $s$ is relatively serializable iff $RSG(s)$ is acyclic.
**RSG Example**

**Example 3.19:**
\[ t_1 = w_1(x) \; r_1(z) \]
\[ t_2 = r_2(x) \; w_2(y) \]
\[ t_3 = r_3(z) \; r_3(y) \]

\[ IU(t_1, t_2) = < [w_1(x) \; r_1(z)] > \]
\[ IU(t_1, t_3) = < [w_1(x)], [r_1(z)] > \]
\[ IU(t_2, t_1) = < [r_2(x)], [w_2(y)] > \]
\[ IU(t_2, t_3) = < [r_2(x)], [w_2(y)] > \]
\[ IU(t_3, t_1) = < [r_3(z)], [r_3(y)] > \]
\[ IU(t_3, t_2) = < [r_3(z) \; r_3(y)] > \]

**RSG(s_5):**

\[ s_5 = w_1(x) \; r_2(x) \; r_3(z) \; w_2(y) \; r_3(y) \; r_1(z) \]
Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

- 3.2 Canonical Synchronization Problems
- 3.3 Syntax of Histories and Schedules
- 3.4 Correctness of Histories and Schedules
- 3.5 Herbrand Semantics of Schedules
- 3.6 Final-State Serializability
- 3.7 View Serializability
- 3.8 Conflict Serializability
- 3.9 Commit Serializability
- 3.10 An Alternative Criterion: Interleaving Specifications
- 3.11 Lessons Learned
Lessons Learned

• Equivalence to serial history is a natural correctness criterion

• CSR, albeit less general than VSR, is most appropriate for
  • complexity reasons
  • its monotonicity property
  • its generalizability to semantically rich operations

• OCSR and COCSR have additional beneficial properties
Proof of Theorem 3.14

(⇒) We prove by induction on n, the number of transactions in s that such s’ exists.

Basis (n=1): S itself is the required s’.

Induction (n>1):

By Theorem 3.13 s in CSR. So, there is a serial s’ such that s ≈_c s‘. Let t’ be the first transaction to commit in s. If t’ is also the first in s’, we eliminate t from s and s’ and by induction hypothesis we are done. So, suppose s’ = … t t’ …

Suppose there are conflicting operations p in t and q in t’. Then, (p,q) in conf(s’)=conf(s) and by COCSR c_t <_s c_t’. This is impossible as t’ is the first to commit in s.

So, there are no conflicting operations in t and t’. We exchange t t’ to t’ t to obtain a modified serial s”” such that conf(s)=conf(s’)=conf(s””). Repeating this process we obtain serial s”” such that conf(s””)=conf(s) and t’ is the 1st transaction in s””’. By induction, if we delete t’ from s”” to obtain s””” and from s to obtain s1, for all t_i, t_j ∈ trans(s1): t_i <_s”” t_j ⇔ c_i <_s1 c_j. Now, we add t’ at the beginning of s””” to obtain s””” and to s1 as before to obtain s and obtain the desired serial schedule.