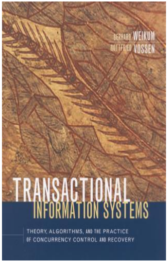
Transactional Information Systems:

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

Gerhard Weikum and Gottfried Vossen

ISBN 1-55860-508-8



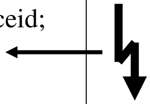
"Teamwork is essential. It allows you to blame someone else."(Anonymous)

Part III: Recovery

- 11 Transaction Recovery
- 12 Crash Recovery: Notion of Correctness
- 13 Page-Model Crash Recovery Algorithms
- 14 Object-Model Crash Recovery Algorithms
- 15 Special Issues of Recovery
- 16 Media Recovery
- 17 Application Recovery

Recall: Funds Transfer Example

```
void main ( ) {
  /* read user input */
  scanf (“%d %d %d”, &sourceid, &targetid, &amount);
  /* subtract amount from source account */
  EXEC SQL Update Account
    Set Balance = Balance - :amount Where Account_Id = :sourceid;
  /* add amount to target account */
  EXEC SQL Update Account
    Set Balance = Balance + :amount Where Account_Id = :targetid;
  EXEC SQL Commit Work; }
```



Observation: *failures may cause inconsistencies,*
require recovery for “atomicity” and “durability”

Also Recall: Dirty Read Problem

P1	Time	P2
r (x)	1	
x := x + 100	2	
w (x)	3	
	4	r (x)
	5	x := x - 100
failure & rollback	6	
	7	w (x)

↑ cannot rely on validity of previously read data

Observation: transaction rollbacks could affect concurrent transactions

Chapter 11: Transaction Recovery

- **11.2 Expanded Schedules**
- 11.3 Page-Model Correctness Criteria
- 11.4 Sufficient Syntactic Conditions
- 11.5 Further Relationships Among Criteria
- 11.6 Extending Page-Model CC Algorithms
- 11.7 Object-Model Correctness Criteria
- 11.8 Extending Object-Model CC Algorithms
- 11.9 Lessons Learned

"And if you find a new way, you can do it today.
You can make it all true. And you can make it undo."(Cat Stevens)

Expanded Schedules with Explicit Undo Steps

Dirty-read problem:

$s = r_1(x)\ w_1(x)\ r_2(x)\ \mathbf{a_1}\ w_2(x)\ c_2$

Approach:

- schedules with aborts are expanded by making the undo operations that implement the rollback explicit
- expanded schedules are analyzed by means of serializability arguments

Dirty-read in expanded schedule:

$s' = r_1(x)\ w_1(x)\ r_2(x)\ \mathbf{w_1^{-1}(x)}\ \mathbf{c_1}\ w_2(x)\ c_2 \quad \rightarrow \notin \text{CSR}$

Examples

$s = r_1(x)\ w_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ \mathbf{a_1}\ r_2(z)\ w_2(z)\ c_2$

Expansion?

How to handle active trasactions, as in

$s = w_1(x)\ w_2(x)\ w_2(y)\ w_1(x)$?

Formal Definition of Expanded Schedules

Definition 11.1 (Expansion of a Schedule):
For a schedule s the **expansion** of s, exp(s), is defined as follows:

- steps of exp(s):
  - $t_i \in \text{commit}(s) \Rightarrow op(t_i) \subseteq op(exp(s))$
  - $t_i \in \text{abort}(s) \Rightarrow (op(t_i) - \{a_i\}) \cup \{c_i\} \cup \{w_i^{-1}(x) \mid w_i(x) \in t_i\} \subseteq op(exp(s))$
  - $t_i \in \text{active}(s) \Rightarrow op(t_i) \cup \{c_i\} \cup \{w_i^{-1}(x) \mid w_i(x) \in t_i\} \subseteq op(exp(s))$
- step ordering in exp(s):
  - all steps from $op(s) \cap op(exp(s))$ occur in exp(s) in the same order as in s
  - all inverse steps of an aborted transaction occur in exp(s) after the original steps in s and before the commit of this transaction
  - all inverse steps of active transactions occur in exp(s) after the original steps of s and before the commits of these transactions
  - the ordering of inverse steps is the reverse of the ordering of the corresponding original steps

Example 11.2:

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

$\Rightarrow\ exp(s) = w_1(x)\ w_2(x)\ w_2(y)\ w_1(y)\ w_1^{-1}(y)\ w_2^{-1}(y)\ w_2^{-1}(x)\ w_1^{-1}(x)\ c_2\ c_1$

Chapter 11: Transaction Recovery

Expanded Conflict Serializability (XCSR)

Definition 11.2 (Expanded Conflict Serializability):
A schedule s is **expanded conflict serializable** if its expansion, exp(s), is conflict serializable.
XCSR denotes the class of expanded conflict serializable schedules.

Example 11.4:

- $s = r_1(x)\ w_1(x)\ r_2(x)\ a_1\ c_2$
  $\Rightarrow$ $exp(s) = r_1(x)\ w_1(x)\ r_2(x)\ w_1^{-1}(x)\ c_1\ c_2$ $\notin$ XCSR
  $s' = r_1(x)\ w_1(x)\ a_1\ r_2(x)\ c_2$
  $\Rightarrow$ $exp(s') = r_1(x)\ w_1(x)\ w_1^{-1}(x)\ c_1\ r_2(x)\ c_2$ $\in$ XCSR

Lemma 11.1:

- XCSR $\subset$ CSR

Example 11.5:

- $s = w_1(x)\ w_2(x)\ a_2\ a_1$
  $\Rightarrow$ $exp(s) = w_1(x)\ w_2(x)\ w_2^{-1}(x)\ c_2\ w_1^{-1}(x)\ c_1$ $\notin$ XCSR

Reducibility (RED)

Definition 11.3 (Reducibility):
A schedule s is **reducible** if its expansion, exp(s), can be transformed into a serial history by finitely many applications of the following rules:

- **commutativity rule (CR):**
  if p, q $\in$ op(exp(s)) s.t. $p < q$ and (p, q) $\notin$ conf(exp(s)) and if there is no step o $\in$ op(exp(s)) with $p < o < q$, then the order of p and q can be reversed.
- **undo rule (UR):**
  if p, q $\in$ op(exp(s)) are inverses of each other (i.e., of the form $p=w_i(x)$ and $q=w_i^{-1}(x)$) and if there is no other step o in between p and q, then the pair of steps p and q can be removed from exp(s).
- **null rule (NR):**
  if p $\in$ op(exp(s)) has the form $p=r_i(x)$ s.t. $t_i \in$ active(s) $\cup$ abort(s), then p can be removed from exp(s).
- **ordering rule (OR):**
  two commutative, unordered operations can be arbitrarily ordered.

Examples in RED and outside RED

Example 11.6:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ a_2\ a_1$

$\Rightarrow exp(s) = r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ w_2^{-1}(x)\ c_2\ w_1^{-1}(x)\ c_1 \quad \in RED$

$\sim r_1(x)\ w_1(x)\ r_2(x)\ c_2\ w_1^{-1}(x)\ c_1$ by UR

$\sim w_1(x)\ c_2\ w_1^{-1}(x)\ c_1$ by NR

$\sim w_1(x)\ w_1^{-1}(x)\ c_2\ c_1$ by CR

$\sim c_2\ c_1$ by UR

Example 11.7:

$s = w_1(x)\ r_2(x)\ c_1\ c_2$

s is in RED, since reduction yields s' = $w_1(x)\ c_1\ r_2(x)\ c_2$

Example 11.8:

$s = w_1(x)\ w_2(x)\ c_2\ c_1$ with prefix s' = $w_1(x)\ w_2(x)\ c_2$

s is in RED, but s' is not

Prefix-Reducibility (PRED)

Definition 11.9 (Prefix Reducibility):
A schedule s is **prefix reducible** if each of its prefixes is reducible.
PRED denotes the class of all prefix-reducible schedules.

Theorem 11.1:
- PRED $\subset$ RED (Lemma 11.2)
- XCSR $\subset$ RED
- XCSR and PRED are incomparable

Activity: Why Histories are [not] in PRED?

1)	$w_1(x)\ r_2(x)\ a_1\ a_2$	$\in$ PRED
2)	$w_1(x)\ r_2(x)\ a_1\ c_2$	$\notin$ PRED
3)	$w_1(x)\ r_2(x)\ c_2\ c_1$	$\notin$ PRED
4)	$w_1(x)\ r_2(x)\ c_2\ a_1$	$\notin$ PRED
5)	$w_1(x)\ r_2(x)\ a_2\ a_1$	$\in$ PRED
6)	$w_1(x)\ r_2(x)\ a_2\ c_1$	$\in$ PRED
7)	$w_1(x)\ r_2(x)\ c_1\ c_2$	$\in$ PRED
8)	$w_1(x)\ r_2(x)\ c_1\ a_2$	$\in$ PRED
9)	$w_1(x)\ w_2(x)\ a_1\ a_2$	$\notin$ PRED
10)	$w_1(x)\ w_2(x)\ a_1\ c_2$	$\notin$ PRED
11)	$w_1(x)\ w_2(x)\ c_2\ c_1$	$\notin$ PRED
12)	$w_1(x)\ w_2(x)\ c_2\ a_1$	$\notin$ PRED
13)	$w_1(x)\ w_2(x)\ a_2\ a_1$	$\in$ PRED
14)	$w_1(x)\ w_2(x)\ a_2\ c_1$	$\in$ PRED
15)	$w_1(x)\ w_2(x)\ c_1\ c_2$	$\in$ PRED
16)	$w_1(x)\ w_2(x)\ c_1\ a_2$	$\in$ PRED

Chapter 11: Transaction Recovery

- 11.2 Expanded Schedules
- 11.3 Page-Model Correctness Criteria
- **11.4 Sufficient Syntactic Conditions**
- 11.5 Further Relationships Among Criteria
- 11.6 Extending Page-Model CC Algorithms
- 11.7 Object-Model Correctness Criteria
- 11.8 Extending Object-Model CC Algorithms
- 11.9 Lessons Learned

Example

Consider

$s = w_1(x)\ r_2(x)\ c_2\ a_1$

s is not acceptable (why?),

yet an SR scheduler would consider it valid (why?).

Sufficient Condition: Recoverability

Definition 11.5 (Recoverability):
A schedule s is **recoverable** if the following holds for all t_i, $t_j \in$ trans(s):
if t_i reads from t_j in s and $c_i \in$ op(s), then $c_j < c_i$.
RC denotes the class of all recoverable schedules.

Example 11.10:

$s_1 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ c_2\ w_1(z)\ c_1$ $\notin$ RC

$s_2 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ w_1(z)\ c_1\ c_2$ $\in$ RC

Sufficient Condition: Avoidance of Cascading Aborts

Definition 11.20 (Avoiding Cascading Aborts):
A schedule s **avoids cascading aborts** if the following holds for all $t_i, t_j \in$ trans(s):
if t_i reads x from t_j in s, then $c_j < r_i(x)$.
ACA denotes the class of all schedules that avoid cascading aborts.

Examples 11.10 and 11.11:

$s_2 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ w_1(z)\ c_1\ c_2$ $\notin$ ACA

$s_3 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ c_2$ $\in$ ACA

$s = w_0(x, 1)\ c_0\ w_1(x, 2)\ w_2(x, 3)\ c_2\ a_1$ $\in$ ACA

Sufficient Condition: Strictness

Definition 11.7 (Strictness):
A schedule s is **strict** if the following holds for all t_i, $t_j \in$ trans(s):
for all $p_i(x) \in op(t_i)$, p=r or p=w, if $w_j(x) < p_i(x)$ then $a_j < p_i(x)$ or $c_j < p_i(x)$.
ST denotes the class of all strict schedules.

Example 11.11 and 11.13:

$s_3 = w_1(x)\ w_1(y)\ r_2(u)\ w_2(x)\ w_1(z)\ c_1\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ c_2$ $\notin$ ST

$s_4 = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ c_2$ $\in$ ST

Sufficient Condition: Rigorousness

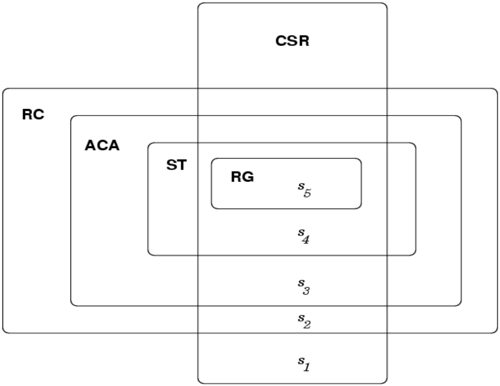
Definition 11.8 (Rigorousness):
A schedule s is **rigorous** if it is strict and the following holds for all $t_i, t_j \in$ trans(s):
if $r_j(x) < w_i(x)$ then $a_j < w_i(x)$ or $c_j < w_i(x)$.
RG denotes the class of all rigorous schedules.

Example 11.13 and 11.14:

$s_4 = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ w_3(u)\ c_3\ c_2$ $\notin$ RG

$s_5 = w_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ w_3(u)\ c_3$ $\in$ RG

Situation



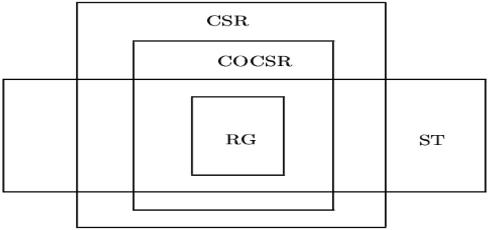
Relationships Among Schedule Classes

Theorems 11.2, 11.3, 11.4:

- $RG \subset ST \subset ACA \subset RC$
- $RG \subset COCSR$
- $CSR \cap ST \subset PRED \subset CSR \cap RC$

Proofs?

Situation



Log-Recoverability

Definition 11.9 (Log Recoverability):
A schedule s is **log recoverable** if the following properties hold:

- s is recoverable
- for all t_i, $t_j \in$ trans(s): if there is a ww conflict of the form $w_i(x) < w_j(x)$ in s, then
  - $a_i < w_j(x)$ or $c_i < c_j$ if t_j commits,
  - or $a_j < a_i$ if t_i aborts.

LRC denotes the class of all log recoverable schedules.

Relationship to PRED for wr and ww conflicts:

1)	$w_1(x)\ r_2(x)\ a_1\ a_2$	$\in$ PRED	1)	$w_1(x)\ w_2(x)\ a_1\ a_2$	$\notin$ PRED
2)	$w_1(x)\ r_2(x)\ a_1\ c_2$	$\notin$ PRED	2)	$w_1(x)\ w_2(x)\ a_1\ c_2$	$\notin$ PRED
3)	$w_1(x)\ r_2(x)\ c_2\ c_1$	$\notin$ PRED	3)	$w_1(x)\ w_2(x)\ c_2\ c_1$	$\notin$ PRED
4)	$w_1(x)\ r_2(x)\ c_2\ a_1$	$\notin$ PRED	4)	$w_1(x)\ w_2(x)\ c_2\ a_1$	$\notin$ PRED
5)	$w_1(x)\ r_2(x)\ a_2\ a_1$	$\in$ PRED	5)	$w_1(x)\ w_2(x)\ a_2\ a_1$	$\in$ PRED
6)	$w_1(x)\ r_2(x)\ a_2\ c_1$	$\in$ PRED	6)	$w_1(x)\ w_2(x)\ a_2\ c_1$	$\in$ PRED
7)	$w_1(x)\ r_2(x)\ c_1\ c_2$	$\in$ PRED	7)	$w_1(x)\ w_2(x)\ c_1\ c_2$	$\in$ PRED
8)	$w_1(x)\ r_2(x)\ c_1\ a_2$	$\in$ PRED	8)	$w_1(x)\ w_2(x)\ c_1\ a_2$	$\in$ PRED

Relationship Between LRC and PRED

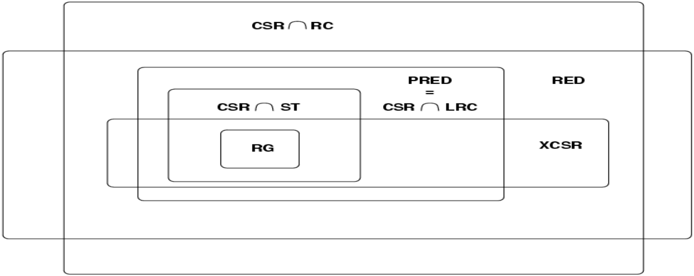
Theorem 11.5:

- PRED = CSR $\cap$ LRC

Proof sketch:

- Lemma 11.3: If s $\in$ CSR $\cap$ LRC, then all operations of uncommitted transactions can be eliminated using rules CR, UR, NR, and OR.
- PRED $\supseteq$ CSR $\cap$ LRC:
  Assume s $\in$ CSR $\cap$ LRC.
  After eliminating operations of uncommitted transactions by Lemma 11.31 (and preserving all conflict orders among committed transactions), s is still CSR and so is every prefix of s. Thus s is in PRED.
- PRED $\subseteq$ LRC:
  Assume s $\in$ PRED but $\notin$ LRC. Consider a conflict $w_i(x) < w_j(x)$. Since s $\notin$ LRC,
  either a) t_j commits but t_i does not commit or commits after t_j
  or b) t_i aborts but t_j does not abort or aborts after t_i.
  All cases lead to contradictions to the assumption that s is in PRED.
  Similarly, assuming that s does not satisfy the RC property for situations like $w_i(x) < r_j(x)\ c_j$, leads to a contradiction.
- PRED $\subseteq$ CSR

Situation



Chapter 11: Transaction Recovery

- 11.2 Expanded Schedules
- 11.3 Page-Model Correctness Criteria
- 11.4 Sufficient Syntactic Conditions
- 11.5 Further Relationships Among Criteria
- **11.6 Extending Page-Model CC Algorithms**
- 11.7 Object-Model Correctness Criteria
- 11.8 Extending Object-Model CC Algorithms
- 11.9 Lessons Learned

Extending 2PL for ST and RG

Theorem 11.6:
Gen(SS2PL) = RG

Theorem 11.7:
Gen(S2PL) $\subseteq$ CSR $\cap$ ST

Extending SGT for LRC

Approach:

- **defer commit** upon commit request of t_j
  if there is a ww or wr conflict from t_i to t_j and t_i is not yet committed
- **enforce cascading abort** for t_j upon abort request of t_i
  if there is a ww or wr conflict from t_i to t_j

ESGT algorithm:

- process w and r steps as usual and maintain serialization graph
  with explicit labeling of edges that correspond to ww or wr conflicts
- upon c_i test if t_i has a predecessor w.r.t. ww or wr edges in the graph;
  if no predecessor exists then perform c_i and resume waiting successors
- upon a_i test if t_i has successor w.r.t. ww or wr edges in the graph;
  if no successor exists then perform a_i,
  otherwise enforce aborts for all successors of t_i

Theorem 11.8:
$Gen(ESGT) \subseteq CSR \cap LRC$

Remark: similar approaches are feasible for other CC protocols (including non-strict 2PL)

Chapter 11: Transaction Recovery

- 11.2 Expanded Schedules
- 11.3 Page-Model Correctness Criteria
- 11.4 Sufficient Syntactic Conditions
- 11.5 Further Relationships Among Criteria
- 11.6 Extending Page-Model CC Algorithms
- **11.7 Object-Model Correctness Criteria**
- 11.8 Extending Object-Model CC Algorithms
- 11.9 Lessons Learned

Aborts in Flat Object Schedules

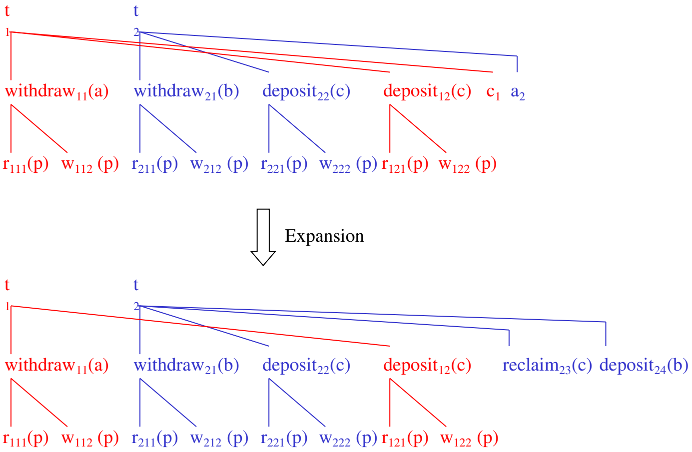
Definition 11.10 (Inverse operations):
An operation $f'(x_1', \ldots, x_{m'}', \uparrow y_1', \ldots, \uparrow y_{k'}')$ with input parameters x_1' through $x_{m'}'$ and output parameters y_1' through $y_{k'}'$ is the **inverse operation** of operation $f(x_1, \ldots, x_m, \uparrow y_1, \ldots, \uparrow y_k)$ if for all possible sequences α and ω of operations on a given interface, the return parameters in the sequence $\alpha\, f(\ldots)\, f'(\ldots)\, \omega$ are the same as in $\alpha\, \omega$.
$f'(\ldots)$ is also denoted as $f^{-1}(\ldots)$.

With the notion of inverse operations, the concepts of expanded schedules and PRED generalize to flat object schedules.

Examples 11.17 and 11.18:

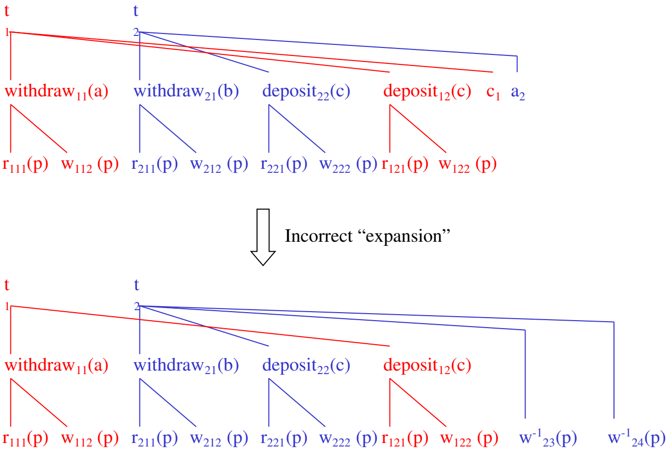
s_1 = $withdraw_1(a)\ withdraw_2(b)\ deposit_2(c)\ deposit_1(c)\ c_1\ a_2 \in$ PRED
$\Rightarrow exp(s_1) =$
$withdraw_1(a)\ withdraw_2(b)\ deposit_2(c)\ deposit_1(c)\ c_1\ reclaim_2(c)\ deposit_2(b)\ c_2$
s_2 = $insert_1(x)\ delete_2(x)\ insert_3(y)\ a_1\ a_2\ a_3 \notin$ PRED
$\Rightarrow exp(s_2) = insert_1(x)\ delete_2(x)\ insert_3(y)\ delete_1(x)\ c_1\ insert_2(x)\ c_2\ delete_3(y)\ c_3$

Example of Correctly Expanded Flat Object Schedule



tree-reducible

Example of Incorrectly Expanded Flat Object Schedule



not tree-reducible

Important observation:
Page-level undo is, in general, incorrect for object-model transactions.

Perfect Commutativity

Definition 11.11 (Perfect Commutativity):
Given a set of operations for an object type, such that for each operation $f(x, p_1, ..., p_m)$ an appropriate inverse operation $f^{-1}(x, p_1', ..., p_{m'}')$ is included.
A commutativity table for these operations is called **perfect** if the following holds:

if $f(x, p_1, ..., p_m)$ and $g(x, q_1, ..., q_n)$ commute then
$f(x, p_1, ..., p_m)$ and $g^{-1}(x, q_1' ..., q_{n'}')$ commute,
$f^{-1}(x, p_1', ..., p_{m'}')$ and $g(x, q_1, ..., q_n)$ commute, and
$f^{-1}(x, p_1', ..., p_{m'}')$ and $g^{-1}(x, q_1' ..., q_{n'}')$ commute.

Definition 11.12 (Perfect Closure):
The **perfect closure** of a commutativity table for the operations of a given object type is the largest, perfect subset of the original commutativity table's commutative operation pairs.

Important observation:
For object types with perfect or perfectly closed commutativity tables,
***S2PL** does not need to acquire any additional locks for undo,*
*and therefore is **deadlock-free during rollback**.*

Examples of Commutativity Tables with Inverse Operations

for object type "page"

	$r_i(x)$	$w_i(x)$	$w_i^{-1}(x)$
$r_i(x)$	+	-	-
$w_i(x)$	-	-	-
$w_i^{-1}(x)$	-	-	-

perfect

for object type "set"

	insert	delete	test	$insert^{-1}$	$delete^{-1}$
insert	-	-	-	-	-
delete	-	-	-	-	-
test	-	-	+	-	-
$insert^{-1}$	-	-	-	+	-
$delete^{-1}$	-	-	-	-	+

not perfect

	insert	delete	test	$insert^{-1}$	$delete^{-1}$
insert	-	-	-	-	-
delete	-	-	-	-	-
test	-	-	+	-	-
$insert^{-1}$	-	-	-	-	-
$delete^{-1}$	-	-	-	-	-

perfectly closed

Chapter 11: Transaction Recovery

- 11.2 Expanded Schedules
- 11.3 Page-Model Correctness Criteria
- 11.4 Sufficient Syntactic Conditions
- 11.5 Further Relationships Among Criteria
- 11.6 Extending Page-Model CC Algorithms
- 11.7 Object-Model Correctness Criteria
- **11.8 Extending Object-Model CC Algorithms**
- 11.9 Lessons Learned

Complete and Partial Rollbacks in General Object-Model Schedules

Definition 11.15 (Terminated Subtransactions):
An object-model history has **terminated subtransactions** if each non-leaf node p_ω has either a child $c_{\omega\nu}$ or $a_{\omega\nu}$ that follows all other (ν-1) children of p_ω.
An object-model schedule with terminated subtransactions is a prefix of an object-model history with terminated subtransactions.

Definition 11.16 (Expanded Object Model Schedule):
For an object model schedule s with terminated subtransactions the **expansion** of s, exp(s), is an object-model history derived as follows:

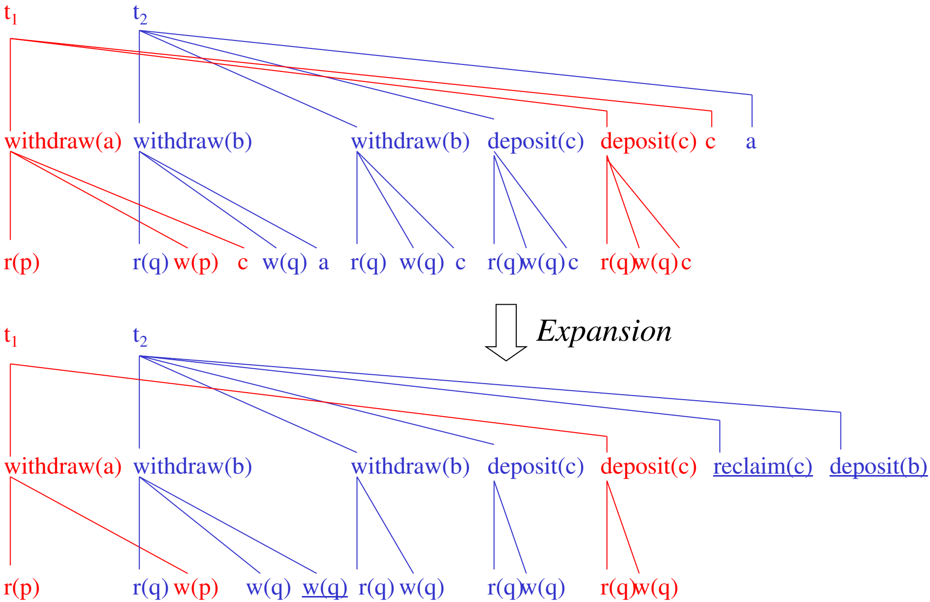
- All operations whose parent has a commit child are included in exp(s).
- For each operation whose parent p_ω has an abort child $a_{\omega\nu}$ an inverse operation is added for all of p's children that do themselves have a commit child, and a commit child is added to p.
  The inverse operations have the reverse order of the corresponding forward operations and placed in between the forward operations and the new commit child.
  All new children of p precede an operation q in exp(s) if the abort child of p preceded q in s.
- For each transaction in active(s) and each non-terminated subtransaction, inverse operations and a final commit child are added as children of the transaction roots, with ordering analagous to above.

Tree Prefix Reducibility for General Object-Model Schedules with Complete and Partial Rollbacks

Definition 11.17 (Extended Tree Reducibility):
An object model schedule s is **extended tree reducible** if its expansion, exp(s), can be transformed into a serial order of s's committed transaction roots by applying the following rules finitely many times:

1. the commutativity rule applied to adjacent leaves,
2. the tree-pruning rule for isolated subtrees,
3. the undo rule applied to adjacent leaves,
4. the null rule for read-only operations, and
5. the ordering rule applied to unordered leaves.

Example with Complete and Partial Rollbacks



Extending Layered Concurrency Control for Complete and Partial Rollbacks

Definition 11.14 (Strictness):
A flat object schedule s is strict if for each pair of L1 operations, p_j and q_i, from different transactions t_i and t_j such that p_j is an update operation, the order $p_j < q_i$ implies that $a_j < q_i$ or $c_j < q_i$.

Theorem 11.10:
A layered object-model schedule for which all level-to-level schedules are order-preserving conflict serializable and strict is extended tree reducible.

Theorem 11.12:
The layered S2PL protocol with perfect commutativity tables generates only schedules that are extended tree reducible.

Chapter 11: Transaction Recovery

- 11.2 Expanded Schedules
- 11.3 Page-Model Correctness Criteria
- 11.4 Sufficient Syntactic Conditions
- 11.5 Further Relationships Among Criteria
- 11.6 Extending Page-Model CC Algorithms
- 11.7 Object-Model Correctness Criteria
- 11.8 Extending Object-Model CC Algorithms
- **11.9 Lessons Learned**

Lessons Learned

- PRED captures correct schedules in the presence of aborts
  by means of intuitive transformation rules.
- Among the sufficient syntactic criteria, LRC, ACA, ST, and RG
  (all in conjunction with CSR), ST is the most practical one.
- Consequently, S2PL is the method of choice
  (and can be shown to guarantee PRED).
- PRED carries over to the object model, in combination
  with the transformation rules of tree-reducibility, leading to TPRED,
  and captures both complete and partial rollbacks of transactions.
- The most practical sufficient syntactic condition for
  layered schedules with perfect commutativity
  requires OCSR and ST for each level-to-level schedule,
  and can be implemented by layered S2PL.