Lecture2and3 W4-5

Friday, 25 October 2024 16:59

Lecture2a... 3

Examenul final are 2p de întrebări de tip grilă cu un singur răspuns corect. Pentru fiecare din cele 3 răspunsuri posibile. Poți da examenul o singură dată în sesiune

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

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

Lost Update Problem

P1	Time	P2
r (x)	/* x = 100 */	
	1	
	2	r (x)
x := x+100	4	x := x+200
w (x)	5	
	/* x = 200 */	
	6	w (x)
	/* x = 300 */	

update "lost" => we lose the consistency from ACID

Observation: problem is the interleaving $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$

Time →

index 1 => transaction 1
index 2 => transaction 2

r(x) -> read information x from DB
W(x) -> write information x from DB
a(x) -> abort operation

A transaction is r(x)+w(x)

Inconsistent Read Problem

P1	Time	P2
	1	r (x)
	2	x := x – 10
	3	w (x)
sum := 0	4	
r (x)	5	
r (y)	6	
sum := sum +x	7	
sum := sum + y	8	
	9	r (y)
	10	y := y + 10
	11	w (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

the scheduler we want to model should not interleave operations.

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

Usually we will consider that all transactions commit,
In this slide we assume that operation $w_1(x)$ aborts and should be rolled back

$r_1(x)\ w_1(x)\ r_2(x)\ a_1(x)\ w_2(x)$

$a_1(x)$ should happen before $r_2(x)$

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

We want to order operations from the transactions.

The scheduler is an algorithm
We input transactions into the scheduler
The output is a schedule, an order of operations, NOT transactions

t_1	t_2	ex schedule:	or Schedule	Sch =
$r_1(x)$	$r_2(z)$	$r_1(x)$	$r_2(x)$	$r_1(y)$
$r_1(y)$	$r_2(x)$	$r_2(z)$	$r_1(z)$	$r_1(x)$
$w_1(x)$	$w_2(y)$	$r_2(x)$	$r_1(y)$	but not
$w_2(x)$		$r_1(x)$		
input data		output data		

ex: $r_1(x), r_1(y)\ldots$

because it does not respect the order of operations inside their respective transaction.

Schedules and Histories

Definition 3.1 (Schedules and histories):
Let $T=\{t_1, \ldots, t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t=(op, <)$ with op_i denoting the operations of t and $<$ their ordering.
(i) A **history** for T is a pair $s=(op(s), <_s)$ s.t. all operation in the history belong only to the transactions of T
(a) $op(s) \subseteq \bigcup_{i=1..n} op_i \cup \bigcup_{i=1..n} \{a_i, c_i\}$
(b) for all i, $1 \le i \le n$: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$ → if a transaction commits, then it cannot abort, and vice versa
(c) $\bigcup_{i=1..n} <_i \subset <_s$
(d) for all i, $1 \le i \le n$, and all $p \in op_i$: $p <_s c_i$ or $p <_s a_i$ → commit/abort are the last operation of each transaction
(e) for all $p, q \in op(s)$ 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. Partially order

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.

We have 2 criteria for a scheduler

1) Correctness: we pick only from schedules which order the operations in such a way so that the order of the operations respect the ACID properties of the transaction they are a part of.
(c)
2) Efficiency: from the correct schedules we pick the schedule which is the most parallelized.

History = Complete Schedule
Partial Schedule = Schedule

Schedules and Histories (2)

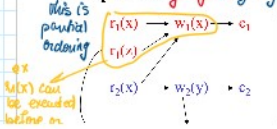
So, a history (for partially ordered transactions):

a) has to contain all operations from all transactions;
b) needs a distinct termination operation for every transaction;
c) preserves all orders within the transactions;
d) has the termination steps as the final steps in each transaction;
e) orders conflicting operations.

Because of a) and b) a history is also called a complete schedule.

Example Schedules and Notation

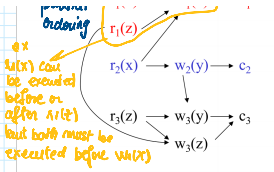
Example 3.4: History (everything is commited or aborted)



trans(s):= $\{t_i \mid s \text{ contains step of } t_i\}$
commit(s):= $\{t_i \in trans(s) \mid c_i \in s\}$
abort(s):=

Exercises:
$s = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)$
$trans(s) = \{t_1, t_2, t_3\}$
$commit(s) = \{t_1, t_2\}$
$abort(s) = \{t_3\}$
$active(s) = \{t_2\}$

$t_1 = r_1(x)$, $t_2 = r_2(x)$
$s' = r_1(x)\ r_2(x)$



$r_1(x)$ can be executed before or after $r_3(z)$ but both must be executed before $w_3(x)$

{t_i | s contains step of t_i}
commit(s):=
{$t_i \in$ trans(s) | $c_i \in$ s}
abort(s):=
{$t_i \in$ trans(s) | $a_i \in$ s}
active(s):=
trans(s) – (commit(s) ∪ abort(s))

Example 3.6: Schedule (there are still running transactions)
This is totally ordered
$r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ a_3$
Time

active (s) = {t_2, t_3}

$t_1 = r_1(x)\ r_1(z)\ w_1(x)\ w_1(y)\ c_1$
$s_1 = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ a_3$
$s_2 = r_2(x)\ r_1(x)\ w_1(x)\ w_1(y)\ c_1\ r_2(z)\ w_2(x)\ w_2(z)\ c_2\ r_3(x)\ r_3(y)\ w_3(z)\ a_3$
$s_3 = r_1(x)\ w_1(x)\ w_1(y)\ c_1\ r_2(x)\ w_2(x)\ w_2(z)\ c_2\ r_3(x)\ r_3(y)\ w_3(z)\ a_3$
} executed in parallel

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

Correctness of Schedules

1. Define equivalence relation ≈ 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.

If person A and person B share a bank account with 10 lei and both people want to retrieve 10 lei, at the same time one of them is going to get an error
A → −10 lei ⇒ B − error
B → −10 lei ⇒ A − error
both of these situations is correct, even though it is different

So, there are cases where equal states at the end means equivalent, but there are always other cases!

x=0, y=1 ⋮ x=100, y=101
x=0, y=1 ⋮ x=100, y=101

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

Herbrand Semantics of Schedules

Definition 3.3 (Herbrand Semantics of Steps):
For schedule s the **Herbrand semantics H_s** of steps $r_i(x)$, $w_i(x) \in op(s)$ 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 $r_i(y_j)$, $1 \le j \le m$, 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, $1 \le i \le n$,
the **Herbrand universe HU** is the smallest set of symbols s.t.
(i) $f_{ix}() \in HU$ for each $x \in D$ where f_{ix} is a constant, and
(ii) if $w_i(x) \in op_i$ 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
$H[s]: D \to HU$ defined by $H[s](x) := H_s[w(x)]$,
where $w_i(x)$ is the last operation from s writing x, for each $x \in D$.

Transaction Processing Systems 3-15

Herbrand Semantic
= output (x,y) of an operation H[Wo(x)]
or a schedule H[s](x)
H[s](y)

Herbrand Semantics: Example

$s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$

T_0: $w_0(x)$ $w_0(y)$ c_0 — T_1: $r_1(x)$ $w_1(y)$ c_1 — T_2: $r_2(y)$ $w_2(x)$ c_2

initialization transaction

we introduce these to initialize the data in the database

the Herbrand semantics of a read of data y depends on the last operation which modified it, that is $w_0(y)$

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

$H[s](x) = H_s[w_2(x)] = f_{2x}(f_{0y}())$ ⇒ the result of running the schedule for data x is actually the result of the last operation which writes x
$H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}())$

transaction 2 is the data where output we want
transaction 1 is the data where output we want

Cura 5 exercitii

Transaction Processing Systems 3-16

Example:
Let the schedule
$S = W_0(x)\ W_0(y)\ W_0(z)\ C_0\ \ W_1(y)\ R_2(z)\ R_2(z)\ R_3(y)\ \ W_2(y)\ R_1(z)\ W_2(x)\ W_2(z)\ R_3(z)\ W_3(z)\ C_1\ C_2\ C_3$

$H[S](x) = H_S[W_2(x)] = f_{2x}(H_S[R_2(x)], H_S[R_3(z)]) = f_{2x}(H_S[W_0(x)], H_S[W_0(z)]) = f_{2x}(f_{0x}(), f_{0z}())$
keep the order of the operation like in the schedule

$H[S](y) = H_S[W_2(y)] = f_{2y}(H_S[R_2(x)], H_S[R_2(z)]) = f_{2y}(H_S[W_0(x)], H_S[W_0(z)]) = f_{2y}(f_{0x}(), f_{0z}())$

$H[S](z) = H_S[W_3(z)] = f_{3z}(H_S[R_3(y)], H_S[R_3(z)]) = f_{3z}(H_S[W_1(y)], H_S[W_0(z)]) = f_{3z}(f_{1y}(H_S[R_2(x)], H_S[R_2(z)]))$
$= f_{3z}(f_{1y}(), f_{2z}(H_S[W_0(x)], H_S[W_0(z)]))$
$= f_{3z}(f_{1y}(), f_{2z}(f_{0x}(), f_{0z}()))$

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

Transaction Processing Systems 3-18

Ex.1.

Let's say we have the schedule $s = r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$

We augment the previous schedule with a virtual transaction $t_0 = w_0(x)\ w_0(y)\ c_0$ which just writes some initial values for the data x and y in the dataset; the transaction t_0 initializes the dataset. So the new schedule becomes:

$s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$

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

Observation: $r_1 - r_1(y)\ w_1(x)$
bookId=1 costs 100 lei
y = Select * from accounts where accountNo=a
if (y>100) {
// x = quantity-1
Update books set quantity = quantity-1 where bookId=1
}

Final-State Equivalence

Definition 3.6 (Final State Equivalence):
Schedules s and s' are called **final state equivalent**, denoted $s \approx_f s'$, if $op(s) = op(s')$ and $H[s] = H[s']$.

Example a:
$s = r_2(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_2(z)\ r_2(x)\ w_2(x)\ w_1(x)$
$s' = r_3(z)\ w_2(z)\ r_2(y)\ r_2(x)\ w_2(x)\ r_1(x)\ w_1(y)\ w_1(x)$
serial
$H[s](x) = H_s[w_1(x)] = f_{1x}(f_{0x}(), f_{0y}()) = H_{s'}[w_1(x)] = H[s'](x)$
$H[s](y) = H_s[w_1(y)] = f_{1y}(f_{0x}(), f_{0y}()) = H_{s'}[w_1(y)] = H[s'](y)$
$H[s](z) = H_s[w_3(z)] = f_{3z}(f_{0x}(), f_{0y}()) = H_{s'}[w_3(z)] = H[s'](z)$
$\Rightarrow s \approx_f s'$

incorrect because Lost update
Example b: interleaving
$s = r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$
$s' = r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$
serial
$H[s](y) = H_s[w_2(y)] = f_{2y}(f_{0y}())$
$H[s'](y) = H_{s'}[w_2(y)] = f_{2y}(f_{1y}(f_{0x}(), f_{0y}()))$
Since the Herbrand semantic of s and s' are different, s and s' are not final state equivalent
$\Rightarrow \neg(s \approx_f s')$

Ex.2. Let's say we have the following two schedules:
$s = w_0(x)\ w_0(y)\ w_0(z)\ r_1(x)\ r_2(y)\ w_2(z)\ r_1(z)\ w_2(z)\ r_1(y)\ w_1(x)$
$s' = w_0(x)\ w_0(y)\ w_0(z)\ r_2(y)\ w_2(z)\ r_1(y)\ r_1(x)\ w_2(z)\ r_1(z)\ w_1(y)\ w_1(x)$

Let's compute the Herbrand semantics for s
$H[s](x) = H_s[w_1(x)] = f_{1x}(H_s[r_1(x)], H_s[r_1(y)]) = f_{1x}(H_s[w_0(x)], H_s[w_0(y)]) = f_{1x}(f_{0x}(), f_{0y}())$
$H[s](y) = H_s[w_2(y)] = f_{2y}(H_s[r_2(y)], H_s[r_2(z)]) = f_{2y}(H_s[w_0(y)], H_s[w_0(z)]) = f_{2y}(f_{0y}(), f_{0z}())$
$H[s](z) = H_s[w_2(z)] = f_{2z}(H_s[r_2(y)], H_s[r_2(z)]) = f_{2z}(H_s[w_0(y)], H_s[w_0(z)]) = f_{2z}(f_{0y}(), f_{0z}())$

Let's compute the Herbrand semantics for s':
$H[s'](x) = H_{s'}[w_1(x)] = f_{1x}(H_{s'}[r_1(x)], H_{s'}[r_1(y)]) = f_{1x}(H_{s'}[w_0(x)], H_{s'}[w_2(z)]) = f_{1x}(f_{0x}(), f_{2z}(f_{0y}(), f_{0z}()))$
$H[s'](y) = H_{s'}[w_1(y)] = f_{1y}(H_{s'}[r_1(x)], H_{s'}[r_1(y)]) = f_{1y}(H_{s'}[w_0(x)], H_{s'}[w_2(z)]) = f_{1y}(f_{0x}(), f_{2z}(f_{0y}(), f_{0z}()))$
$H[s'](z) = H_{s'}[w_2(z)] = f_{2z}(H_{s'}[r_2(y)], H_{s'}[r_2(z)]) = f_{2z}(H_{s'}[w_0(y)], H_{s'}[w_0(z)]) = f_{2z}(f_{0y}(), f_{0z}())$

Because $H[s] = H[s'] \Rightarrow s \approx_f s'$ (i.e. s and s' are final-state equivalent)
Because s' is a serial schedule (i.e. $s' = t_2\ t_1\ t_3$) $\Rightarrow$ s is final-state serializable

Reads-from Relation

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_∞.
(i) **$r_i(x)$ reads x in s from $w_j(x)$** if $w_j(x) <_s r_i(x)$.
(ii) The **reads-from relation** of s is
$RF(s) := \{(t_i, x, t_j) \mid$ an $r_i(x)$ reads x from a $w_j(x)\}$.
(iii) Step p is **directly useful** for step q, denoted $p \to q$, if q reads from p, or p is a read step and q is a subsequent write step of the same transaction.
$\to^*$, the **"useful" relation**, denotes the reflexive and transitive closure of $\to$.

reflexivity $p \to p$
transitivity $p \to q, q \to r \Rightarrow p \to r$

Herbrand Semantic
directly useful
alive step
dead steps

Example 2.
Let there be two schedules:
$s = r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$
$s' = r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$

We augment these 2 schedules with transactions t_0 and t_∞.
$s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ r_\infty(x)\ r_\infty(y)\ c_\infty$

(ii) The **reads-from relation** of s is
RF(s) : = {(t_i, x, t_j) | an r_j(x) reads x from a w_i(x)}.
(iii) Step p is **directly useful** for step q, denoted p→q, if q reads from p, or p is a read step and q is a subsequent write step of the same transaction. >*, the "**useful**" **relation**, denotes the reflexive and transitive closure of >.
(iv) Step p is **alive** in s if it is useful for some step from t_∞ and **dead** otherwise.
(v) The **live-reads-from relation** of s is
LRF(s) : = {(t_i, x, t_j) an alive r_j(x) reads x from w_i(x)}

LRF(s) ⊆ RF(s)

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_1,x,t_∞), (t_2,y,t_∞)}
RF(s') = {(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_∞), (t_2,y,t_∞)}
LRF(s) = {(t_0,y,t_2), (t_0,x,t_∞), (t_2,y,t_∞)}
LRF(s') = {(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_∞), (t_2,y,t_∞)}

Final-State Serializability

Theorem 3.1:
For schedules s and s' the following statements hold.
(i) s ≈_f s' iff op(s)=op(s') and LRF(s)=LRF(s').
(ii) For s let the step graph D(s)=(V,E) be a directed graph with vertices V:=op(s) and edges E:={(p,q) p→q}, and the reduced step graph D_1(s) be derived from D(s) by removing all vertices that correspond to dead steps. Then LRF(s) = LRF(s') iff 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 ≈_f s'.
FSR denotes the class of all final-state serializable histories.

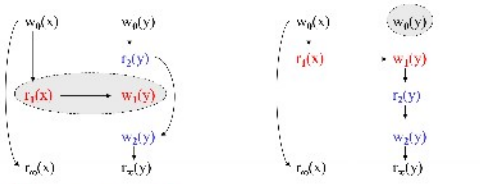
FSR: Example 3.9

Error: Lost Update

s= $r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$ — this operation is overwritten

s'= $r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$ — this is the correct schedules since is the serial one.

D(s): D(s'):



Some the 2 graphs presented there are different, the 2 schedules are not final state equivalent

dead steps

The final State Equivalence presents some problems because of the need to be checking the final state serial form against all possible final state serial forms possible for a specific schedule, thus it introduces a factorial component.

* This is why, even though it is specifically good at detecting the Lost Update Error, we are going to drop it. Drawbacks like Inconsistent Read anomaly occur, as the final state of the bad schedule is the same as of a serial permutation of the schedule. We discover that the bad schedule is final state serializable!

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

alive step → dead steps

We augument these 2 schedules with transactions t_0 and t_∞.

The read ∞ are always alive as they are useful to themselves

Let's define the read-from sets for these schedules:

those are alive

Let's compute the useful relations for schedule s:

this means that all the steps from this chain are alive steps

steps $w_0(x)$, $r_\infty(x)$ are alive even if $w_0(x)$ appeared on a dead chain

So, the steps $r_1(x)$, $w_1(y)$ are dead.

Let's compute the useful relations for schedule s'.

this means that all the steps from this chain are alive steps

this step may be alive if it appears on an alive chain

: this step is alive

So step $w_1(y)$ is dead.

So, the live-reads-from sets are:

Exemplu 1. Se da planificarea: s = $r_1(x)\ r_2(y)\ r_3(x)\ r_1(z)\ w_1(y)\ w_2(z)\ w_3(x)\ r_3(z)\ w_4(y)\ r_1(y)\ w_1(z)$ $w_3(y)$. Sa se calculeze multimile RF(s) si LRF(s).

Augumentam planificarea cu tranzactiile t_0 si t_∞:

RF(s) = { ... }

Calculam relatia "folositor" si "direct folositor":

is alive because is part of at least one alive chain

dead chain

dead chain

: operatii fpasi alive

: operatii fpasi alive

: operatii fpasi alive

(ii) this image is called the step Graph, and by eliminating the dead nodes we obtain the Reduced Step Graph

we can compare 2 schedules by looking at their Reduced Step Graph. if the 2 Reduced Steps Graphs are the same, then the 2 Schedules are Final State Equivalent

Prin urmare, operatiile urmatoare sunt dead-moarte: $r_1(x)$, $w_1(y)$, $w_4(z)$, $r_1(y)$, $w_1(z)$, $r_3(z)$. Restul operatiilor sunt alive/in viata.

Reamintim ca multimea LRF este:

LRF(s) = { ... }

Exemplu.2. Let's consider the schedule s = $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$. Decide whether s is final-state serializable (i.e. s belongs to FSR).

RF(s) = { ... }

Let's compute the "useful" relationship:

$w_0(x) \rightarrow r_1(x) \rightarrow w_1(x)$: dead steps (except $w_0(x)$)

$w_0(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_\infty(x)$: alive steps

⇒ LRF(s) = { ... }

a) Let's consider the serial schedule s_{ser2} = $r_2(x)\ w_2(x)\ c_2\ r_1(x)\ w_1(x)\ c_1$

RF(s_{ser2}) = { ... }

Let's compute the "useful" relationship for this schedule:

$w_0(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x) \rightarrow w_1(x) \rightarrow r_\infty(x)$: alive steps

So, we have LRF(s) = LRF(s_{ser2}) = { ... }.

b) Let's consider the serial schedule s_{ser1} = $r_1(x)\ w_1(x)\ c_1\ r_2(x)\ w_2(x)\ c_2$

RF(s_{ser1}) = { ... }

Let's compute the "useful" relationship for this schedule:

$w_0(x) \rightarrow r_1(x) \rightarrow w_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_\infty(x)$: alive steps

So, we have LRF(s) = LRF(s_{ser1}) = { ... }.

So LRF(s) ≠ LRF(s_{ser2}) and LRF(s)≠LRF(s_{ser2}). So schedule s is not final-state serializable.

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

$RF(s_{act}) = \{(t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\}$

Let's compute the "useful" relationship for this schedule:

$w_0(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x) \rightarrow w_1(x) \rightarrow r_\infty(x)$: alive steps

So, we have $LRF(s_{act}) = RF(s_{act}) = \{(t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\}$.

So $LRF(s) \neq LRF(s_{12})$ and $LRF(s) \neq LRF(s_{21})$. So schedule s is not final-state serializable.

Canonical Anomalies Reconsidered

- **Lost update anomaly:**

  $L = r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$

  → history is not FSR

  $LRF(L) = \{(t_0,x,t_2), (t_2,x,t_\infty)\}$
  $LRF(t_1\ t_2) = \{(t_0,x,t_1), (t_1,x,t_2), (t_2,x,t_\infty)\}$
  $LRF(t_2\ t_1) \quad \{(t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\}$

- **Inconsistent read anomaly:**

  $I = r_3(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_3(y)\ c_1\ c_2$

  → history is FSR !

  $LRF(I) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
  $LRF(t_1\ t_2) \quad \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
  $LRF(t_2\ t_1) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_1), (t_2,y,t_\infty)\}$

Observation: (Herbrand) semantics of all read steps matters!

NEW EQUIVALENCE CRITERIA :

View Serializability

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

→ the final data should be the same (final state serializability)
+ we add an additional request:
the Herbrand Semantic of each step should be the same
= they read and write the same values at each step.

Theorem 3.2:
For schedules s and s' the following statements hold:
(i) $s \approx_v s'$ iff $op(s)=op(s')$ and $RF(s)=RF(s')$ → 2 schedules are view equivalent if the set of operations is the same AND the Read From (RF) set contains the same tuples!
(ii) $s \approx_v s'$ iff $D(s) \quad D(s')$ → 2 schedules are view equivalent if their STEP GRAPHS are the same!

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. $s \approx_v s'$.
VSR denotes the class of all view-serializable histories.

Inconsistent Read Reconsidered

- **Inconsistent read anomaly:**

  $I = r_2(x)\ w_3(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$

  → history is not VSR !

  this is not in the first ; this is not in the second

  $RF(I) = \{(t_0,x,t_3), (t_3,x,t_1), (t_0,y,t_1), (t_1,x,t_2), (t_2,y,t_\infty)\}$
  $RF(t_1\ t_2) = \{(t_0,x,t_1), (t_0,y,t_1), (t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
  $RF(t_2\ t_1) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_1), (t_2,y,t_1), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$

  $RF(I) \neq RF(t_1 t_2)$ and $RF(I) \neq RF(t_2 t_1)$
  ⇓ thus.

Observation: VSR properly captures our intuition

Relationship Between VSR and FSR

Theorem 3.3:
VSR $\subset$ FSR.

Theorem 3.4:
Let s be a history without dead steps. Then s $\in$ VSR iff s $\in$ 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 $\subseteq$ trans(s). Π_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 Π_T(s) is in E for each T $\subseteq$ trans(s).
VSR is not monotone.

Example:
s = $w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$ $\rightarrow \in$ VSR
$\Pi_{\{t1,\ t2\}}$(s) = $w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$ $\rightarrow \notin$ VSR

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

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 $s \approx_c s‘$, if op(s) = op(s‘) and conf(s) = conf(s‘).

Definition 3.14 (Conflict Serializability):
Schedule s is **conflict serializable** if there is a serial schedule s‘ s.t. $s \approx_c s‘$.
CSR denotes the class of all conflict serializable schedules.

Example a: $r_1(x)\ r_2(x)\ r_1(z)\ w_1(x)\ w_2(y)\ r_3(z)\ w_3(y)\ c_1\ c_2\ w_3(z)\ c_3$ → ∈ CSR
Example b: $r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$ → ∉ CSR

Properties of CSR

Theorem 3.8:
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 ∈ VSR, but s ∉ CSR.

Theorem 3.9:
(i) CSR is monotone.
(ii) $s \in CSR \Leftrightarrow \Pi_T(s) \in VSR$ for all $T \subseteq trans(s)$
(i.e., CSR is the largest monotone subset of VSR).

Conflict Graph

Definition 3.15 (Conflict Graph):
Let s be a schedule. The **conflict graph** G(s) = (V, E) is a directed graph with vertices V := commit(s) and edges E := {(t, t‘) | t ≠ t‘ and there are steps p ∈ t, q ∈ t‘ with (p, q) ∈ conf(s)}.

Theorem 3.10:
Let s be a schedule. Then s ∈ CSR iff G(s) is acyclic.

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:
$s = r_1(y)\ r_3(w)\ r_2(y)\ w_1(y)\ w_1(x)\ w_2(x)\ w_2(z)\ w_3(x)\ c_1\ c_3\ c_2$

G(s):

Proof of the Conflict-Graph Theorem

(i) Let s be a schedule in CSR. So there is a serial schedule s‘ with conf(s) = conf(s‘).
Now assume that G(s) has a cycle $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_k \rightarrow t_1$.
This implies that there are pairs $(p_1, q_2), (p_2, q_3), \ldots, (p_k, q_1)$ 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:

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 $\rightsquigarrow o_i(x)\ p_j(y)$ if $x \neq y$ or both o and p are reads

Example for transformations of schedules:

s $= w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ w_2(y)\ w_3(y)\ w_3(z)$

$\rightsquigarrow$[C2] $w_1(x)\ w_1(y)\ r_2(x)\ w_1(z)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

$\rightsquigarrow$[C2] $w_1(x)\ w_1(y)\ w_1(z)\ r_2(x)\ w_2(y)\ r_3(z)\ w_3(y)\ w_3(z)$

$= t_1\ t_2\ t_3$

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 ~* s‘.

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

Corollary 3.5:
Schedule s is commutativity-based reducible iff $s \in CSR$.

Order Preserving Conflict Serializability

Definition 3.18 (Order Preservation):
Schedule s is **order preserving conflict serializable** if it is conflict equivalent to a serial schedule s‘ and for all t, t‘ $\in$ 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):
Schedule s is **commit order preserving conflict serializable** if for all $t_i, t_j \in$ trans(s): if there are $p \in t_i$, $q \in t_j$ with $(p,q) \in$ conf(s) then $c_i <_s c_j$.
COCSR denotes the class of all schedules with this property.

Theorem 3.13:
COCSR $\subset$ CSR.

Theorem 3.14:
Schedule s is in COCSR iff there is a serial schedule s‘ s.t. $s \approx_c s‘$ and for all $t_i, t_j \in$ trans(s): $t_i <_{s‘} t_j \Leftrightarrow c_i <_s c_j$.

Theorem 3.15:
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

Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

Commit Serializability

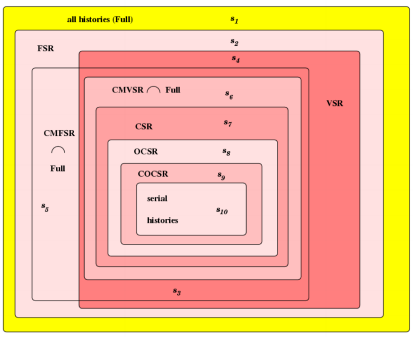
Definition 3.20 (Closure Properties of Schedule Classes):
Let E be a class of schedules.
For schedule s let CP(s) denote the projection $\Pi_{commit(s)}(s)$.
E is **prefix-closed** if the following holds: $s \in E \Leftrightarrow p \in E$ for each prefix of s.
E is **commit-closed** if the following 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-Θ-serializable** if CP(p) is Θ-serializable for each prefix p of s, where Θ can be FSR, VSR, or CSR.
The resulting classes of commit-Θ-serializable schedules are denoted CMFSR, CMVSR, and CMCSR.

Theorem 3.17:
(i) CMFSR, CMVSR, CMCSR are prefix-commit-closed.
(ii) CMCSR ⊂ CMVSR ⊂ CMFSR

Landscape of History Classes



Chapter 3: Concurrency Control – Notions of Correctness for the Page Model

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:

$r_1(a)\ w_1(a)\ r_2(b)\ w_2(b)\ r_2(c)\ w_2(c)\ r_1(c)\ w_1(c)$	$\rightarrow \in$ CSR	application-tolerable interleavings
$r_1(a)\ w_1(a)\ r_3(a)\ r_3(b)\ r_3(c)\ r_1(c)\ w_1(c)$	$\rightarrow \notin$ CSR	
$r_1(a)\ w_1(a)\ r_2(b)\ w_2(b)\ r_1(c)\ r_2(c)\ w_2(c)\ w_1(c)$	$\rightarrow \notin$ CSR	non-admissable interleavings
$r_1(a)\ w_1(a)\ r_4(a)\ r_4(b)\ r_4(c)\ w_4(z)\ r_1(c)\ w_1(c)$	$\rightarrow \notin$ 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 $p \rightsquigarrow q$, 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.
$\rightsquigarrow^*$ denotes the reflexive and transitive closure of $\rightsquigarrow$.

Definition 3.24 (Relatively Serial Schedule):
s is **relatively serial** if for all transactions t_i, t_j: if $q \in t_j$ is interleaved with some $IU_k(t_i, t_j)$, then there is no operation $p \in IU_k(t_i, t_j)$ s.t. $p \rightsquigarrow^* q$ or $q \rightsquigarrow^* p$

Example 3.16:

$s_3 = r_1(x)\ r_2(y)\ w_1(x)\ w_2(y)\ w_3(x)\ w_1(z)\ w_3(y)\ r_2(x)\ r_1(y)\ w_3(z)$

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

Example 3.17:

$s_4 = r_1(x)\ r_2(y)\ w_2(y)\ w_1(x)\ w_3(x)\ r_2(x)\ w_1(z)\ w_3(y)\ r_1(y)\ w_3(z)$

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 $p \rightsquigarrow^* q$ 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.

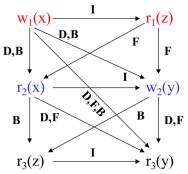
RSG Example

Example 3.19:
$t_1 = w_1(x)\ r_1(z)$
$t_2 = r_2(x)\ w_2(y)$
$t_3 = r_3(z)\ r_3(y)$

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

$s_5 = w_1(x)\ r_2(x)\ r_3(z)\ w_2(y)\ r_3(y)\ r_1(z)$

RSG(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**

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