

CA420 Databases II

Serialisability

A Concurrency Problem – *Inconsistent Retrieval*

Problem:

- accounts 13 and 86 each have a balance of €200, and
- T_3 reads the sum of the balances, at the same time as
- T_4 transfers €100 from account 13 to account 86

13	86	Read ₄ (Accounts[13]); -- reads 200	13	86
€200	€200	Write ₄ (Accounts[13], 100);	€100	€300
		Read ₃ (Accounts[13]); -- reads <u>100</u>		
		Read ₃ (Accounts[86]); -- reads <u>200</u>		
		Read ₄ (Accounts[86]); -- reads 200		
		Write ₄ (Accounts[86], 300);		
		Commit ₄ ;		
		Commit ₃ ;		

- but T_3 reads €300, not €400
- this is known as *inconsistent retrieval*

A Concurrency Problem – *Lost Updates*

```
Procedure Deposit begin
  Start;
  input(account#, amount);
  temp := Read(Accounts[account#]);
  temp := temp + amount;
  Write(Accounts[account#], temp);
  Commit;
End

Read1(Accounts[13]); -- reads 1000
Read2(Accounts[13]); -- reads 1000
Write2(Accounts[13], 1100);
Commit2;
Write1(Accounts[13], 1100);
Commit1;
```

Deposited €100 twice, yet final balance is only €1100!

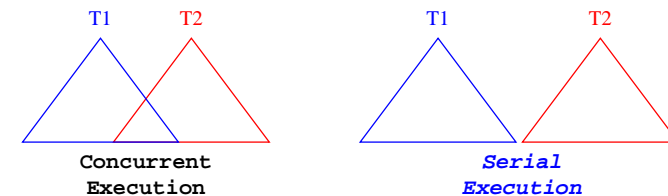
This is known as the *lost update problem*

(Note: the execution is ST, hence also ACA and RC)

Serial Executions

An execution is *serial* if:

- for any pair of transactions, *all* the operations of one transaction execute before *any* of the operations of the other

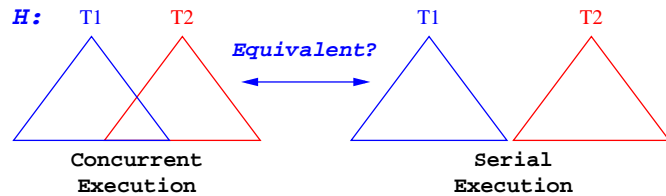


- serial executions are assumed to be *correct*
- moreover, *any* serial execution is assumed to be correct

Serialisability

Approach to defining correctness (serialisability):

- serial executions are correct,
- so all executions that are *equivalent to serial executions* are also correct



In other words:

- if a scheduler generates a history H ,
- and we can prove that that history has the *same effect* as a serial history,
- then H (and the scheduler) can be said to be “correct”

Serialisability – Simple Example, 1

Consider:

- $r_1[x] \rightarrow w_1[y] \rightarrow c_1$ and
- $r_2[x] \rightarrow w_2[x] \rightarrow c_2$

There only only two possible serial executions:

- $r_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2$
- $r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow r_1[x] \rightarrow w_1[y] \rightarrow c_1$

Now, consider:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_1[y] \rightarrow c_1$

It can be argued that this is equivalent to:

- $r_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2$
- so this interleaving might be considered correct

Serialisability – Simple Example, 1, Continued

Consider:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_1[y] \rightarrow c_1$
- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow c_2$
- $r_1[x] \rightarrow r_2[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_2[x] \rightarrow c_2$
- $r_1[x] \rightarrow w_1[y] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2$

- each execution has the same effect, so they are “equivalent”
- the latter execution is serial, so it is “correct”
- therefore, the first execution can also be considered “correct”

Serialisability – Simple Example, 2

Consider:

- $r_1[x] \rightarrow w_1[x] \rightarrow c_1$ and
- $r_2[x] \rightarrow w_2[x] \rightarrow c_2$

There only only two possible serial executions:

- $r_1[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2$
- $r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow r_1[x] \rightarrow w_1[x] \rightarrow c_1$

Now, consider:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_1[x] \rightarrow c_1$
- this does not appear to be equivalent to either of the two serial executions above
- so perhaps this latter execution cannot be considered “correct”

Conflicting Operations

Two operations *conflict* if their order of execution matters:

- $r_i[x] \rightarrow w_j[x]$
- $w_j[x] \rightarrow r_i[x]$
- T_i reads a different value for x in each case, so the order matters

For *reads* and *writes*, two operations *conflict* iff:

- they both operate on the same data item, and
- at least one of them is a write

The definition can easily be extended to include further operations (such as *increment* and *decrement*)

Complete Histories (or “Schedules” or “Executions”)

For $T = \{T_1, T_2, \dots, T_n\}$, a *complete history* H over T is partial ordering such that:

1. $H = \bigcup_{i=1}^n T_i$
2. $\langle_H \supseteq \bigcup_{i=1}^n \langle_i$, and
3. for any two conflicting operations $p, q \in H$, either:
 - $p \langle_H q \in \langle_H$, or
 - $q \langle_H p \in \langle_H$

Or, in English:

1. H contains all and only of the operations of $\{T_1, T_2, \dots, T_n\}$
2. \langle_H honours the order of each transactions' operations (\langle_i)
3. \langle_H orders each pair of conflicting operations

Conflict Matrices

A *conflict matrix* describes which pairs of operations conflict:

- that is, pairs for which the order of execution matters

Conflict matrix for read and write operations:

	$r[x]$	$w[x]$
$r[x]$		×
$w[x]$	×	×

A conflict matrix may also include other operations:

	$r[x]$	$w[x]$	$incr[x]$	$decr[x]$
$r[x]$				
$w[x]$				
$incr[x]$				
$decr[x]$				

Histories

In a complete history, all transactions either commit or abort; there are no partially-executed (or “active”) transactions

During transaction processing, some transactions may have committed, some aborted, and some be active

A *history* is just a prefix of a complete history and may include active transactions

It models the active state of transaction processing, from the perspective of a scheduler

Complete Histories and Histories

A complete history:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow w_1[x] \rightarrow c_1$

A history:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2$

In this case, the second example is a *prefix* of the first

(Note, however, that in general these are partial orders or DAGs, not always total orders or lists as here)

Committed Projections

The *committed projection* of a history H , denoted $C(H)$, is simply:

- H with all the operations of transactions T_i such that $c_i \notin H$ removed

That is, $C(H)$ is just H with:

- all the operations of active transactions removed, and
- all the operations of aborted transactions removed

When a transaction successfully commits, the DBMS effectively “promises” that its execution was “correct”

The committed projection of H includes all and only transactions for which such “promises” have been made in H

Example Histories

$$\begin{array}{l} T_1 = r_1[x] \rightarrow w_1[x] \rightarrow c_1 \\ T_3 = r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_1 \\ T_4 = r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_1 \end{array}$$

$$\begin{array}{l} H_1 = r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_1 \\ \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ \quad \quad \quad r_3[x] \rightarrow w_3[y] \rightarrow w_3[x] \rightarrow c_1 \\ \quad \quad \quad \uparrow \\ \quad \quad \quad w_4[x] \rightarrow w_4[y] \rightarrow w_4[z] \rightarrow c_1 \\ \quad \quad \quad \uparrow \\ \quad \quad \quad r_1[x] \rightarrow w_1[x] \rightarrow c_1 \end{array}$$

$$\begin{array}{l} H_2 = r_4[y] \rightarrow w_4[x] \rightarrow w_4[y] \\ \quad \quad \quad \uparrow \quad \quad \quad \uparrow \\ \quad \quad \quad r_3[x] \rightarrow w_3[y] \\ \quad \quad \quad \uparrow \\ \quad \quad \quad w_4[x] \rightarrow w_4[y] \\ \quad \quad \quad \uparrow \\ \quad \quad \quad r_1[x] \rightarrow w_1[x] \rightarrow c_1 \end{array}$$

H_1 is a complete history over $\{T_1, T_3, T_4\}$, H_2 is a history (also a prefix of H_1)

Serialisability Approach

Review:

- defined transactions
- defined conflicting operations
- defined complete histories
- defined histories
- defined committed project of a history

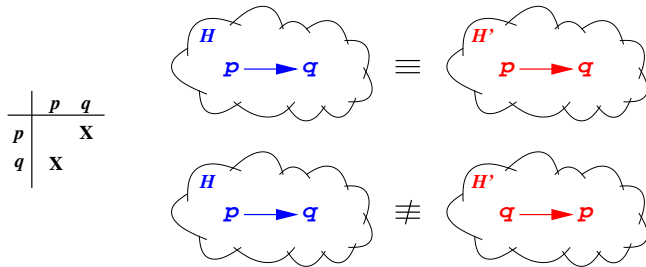
Approach:

- define equivalence of two histories
- define serial histories
- *a history is (conflict) serialisable iff it is equivalent to a serial one*

Equivalence of Histories

Two histories H and H' are *equivalent* (\equiv) iff:

- they are defined over the same set of transactions and have the same operations, and
- they order the conflicting operations of non-aborted transactions in the same way; that is:
 - for conflicting $p_i \in T_i$ and $q_j \in T_j$ where $a_i, a_j \notin H$:
 - if $p_i <_H q_j$ then $p_i <_{H'} q_j$



Serial Histories

A complete history is *serial* iff:

- for each pair of transactions $T_i, T_j \in H$, either:
 - all operations of T_i appear before any of those of T_j
 - or vice versa



That is, transactions' executions are not interleaved in any way

History Equivalence – Examples

$$H_1 = r_2[x] \rightarrow r_1[y] \rightarrow w_1[x] \rightarrow r_2[z] \rightarrow w_2[z] \rightarrow w_1[z]$$

$$H_2 = r_2[z] \rightarrow w_2[z] \rightarrow r_2[x] \rightarrow r_1[y] \rightarrow w_1[x] \rightarrow w_1[z]$$

$$H_3 = r_2[x] \rightarrow r_1[y] \rightarrow w_1[x] \rightarrow w_1[z] \rightarrow r_2[z] \rightarrow w_2[z]$$

In each case, these are the same operations from the same transactions

H_1 is equivalent to H_2 , but neither is equivalent to H_3

(Conflict) Serialisability

A history H is *serialisable* iff:

- its *committed projection* $C(H)$ is equivalent to a serial history

Serialisability is only concerned with committed transactions

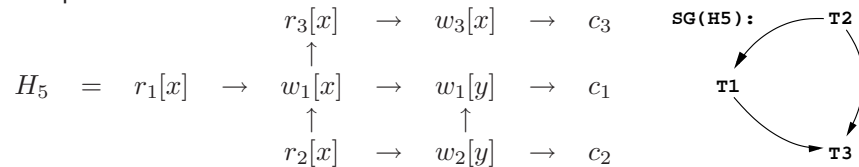
We now have a formal definition of serialisability, but can we tell automatically where a given execution is serialisable

Serialisation Graphs

The *serialisation graph* for a history H , denoted $SG(H)$, is a directed graph:

- whose nodes are the committed transactions of H , and
- which contains an edge $T_i \rightarrow T_j$ iff:
 - $\exists p \in T_i$, and $\exists q \in T_j$ and p and q conflict

Example:



The (Conflict) Serialisability Theorem

The serialisability theorem:

- a history H is serialisable iff its serialisation graph $SG(H)$ is acyclic

Note: this is both *if* and *only if*

Proof – show:

- a history H is serialisable *if* its serialisation graph $SG(H)$ is acyclic, and
- a history H 's serialisation graph $SG(H)$ is acyclic *if* H is serialisable

Proof, Serialisability Theorem – “If” Part

Assume $SG(H)$ is acyclic, and show H is serialisable

- assume T_1, T_2, \dots, T_m are the committed transactions in H , hence these are the nodes of $SG(H)$
- let the sequence $T_{i_1}, T_{i_2}, \dots, T_{i_m}$ be any topological sort of $SG(H)$
- let H_s be the serial history $T_{i_1}, T_{i_2}, \dots, T_{i_m}$
- goal: *show $C(H) \equiv H_s$, and hence that H is serialisable*
- for any conflicting p and q from committed transactions T_p and T_q from H , respectively, such that $p <_H q$ we have:
 - $T_p \rightarrow T_q$ is in $SG(H)$
 - therefore, T_p before T_q in topological sort of $SG(H)$
 - therefore, T_p before T_q in serial history H_s

Therefore $C(H) \equiv H_s$ and H is serialisable, as required

Proof, Serialisability Theorem – “Only If” Part

Assume H is serialisable, show $SG(H)$ is acyclic

- whenever we have an edge $T_i \rightarrow T_j$ in $SG(H)$
- there exist conflicting operations p_i and q_j such that $p_i <_H q_j$
- because $C(H) \equiv H_s$, we have $p_i <_{H_s} q_j$
- therefore T_i appears before T_j in H_s
- now, suppose there were a cycle in $SG(H)$
 - assume that cycle is $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_k \rightarrow T_1$
- hence, T_1, T_2, \dots, T_k all appear before T_1 in the serial history H_s , which is an absurdity

Therefore, no such cycle can exist, and hence $SG(H)$ is acyclic, as required

Note on “Equivalent Serial Histories”

A serialisable history H is equivalent to *any* serial history corresponding to a topological sort of $SG(H)$

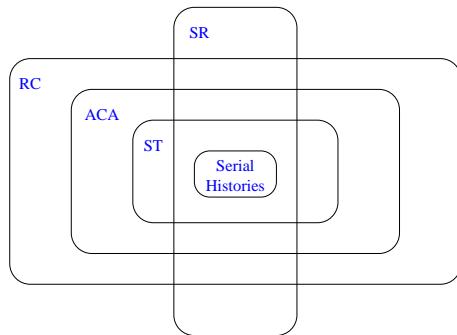
So H may be equivalent to more than one serial history

SR is orthogonal to RC, ACA and ST

SR is orthogonal to RC, ACA and ST:

- there exist executions which are conflict serialisable, but not RC, ACA or ST
- $w_1[x] \rightarrow w_1[y] \rightarrow w_2[x] \rightarrow r_2[y] \rightarrow c_2 \rightarrow c_1$
- and vice versa

Recoverability, Serialisability – Summary



RC: a txn should not commit before txns it read from have committed or aborted

ACA: a txn should not read from an as yet uncommitted txn

ST: a txn should not read or write an item that was previously written by an as yet uncommitted txn

SR: the committed project of an execution is equivalent to a serial execution

Some SR, RC, ACA and ST Examples

$H_1: r_1[x] \rightarrow w_2[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow c_2$

$H_2: r_1[x] \rightarrow w_2[x] \rightarrow r_1[x] \rightarrow c_1$

$H_3: r_1[y] \rightarrow w_2[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow c_2$

$H_4: r_1[x] \rightarrow w_2[x] \rightarrow w_3[y] \rightarrow c_2 \rightarrow c_3 \rightarrow r_1[y] \rightarrow c_3$

$H_5: r_1[x] \rightarrow w_2[x] \rightarrow r_3[x] \rightarrow w_1[y] \rightarrow w_3[y] \rightarrow a_2 \rightarrow c_1 \rightarrow c_3$

$H_6: w_1[x] \rightarrow w_2[x] \rightarrow w_3[x] \rightarrow w_4[x] \rightarrow c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4$

$H_7: r_1[x] \rightarrow w_2[x] \rightarrow w_3[x] \rightarrow w_1[x] \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$

$H_8: r_1[x] \rightarrow w_2[x] \rightarrow w_3[x] \rightarrow c_2 \rightarrow c_3 \rightarrow w_1[y] \rightarrow c_1$

A Serialisability Curiosity

The serialisation order may be *different* from the execution order!

Consider:

- $r_1[x] \rightarrow r_2[x] \rightarrow w_2[x] \rightarrow c_2 \rightarrow r_3[y] \rightarrow w_3[y] \rightarrow c_3 \rightarrow w_1[y] \rightarrow c_1$
- T_2 commits before T_3 begins
- T_2 and T_3 are executed serially
- but what is the serialisation order?

RC, ACA, ST and SR are Prefix-Commit Closed

RC, ACA, ST and SR are all prefix-commit closed

- RC, ACA, ST:
 - incorrect prefixes stay incorrect
 - $w_1[x] \rightarrow w_1[y] \rightarrow w_2[x] \rightarrow r_2[y] \rightarrow c_2 \rightarrow c_1$
- SR:
 - assume H' is a prefix of H
 - every edge in $SG(C(H'))$ is also in $SG(C(H))$
 - so any cycle in $SG(C(H'))$ is also in $SG(C(H))$
 - again incorrect prefixes stay incorrect

Prefix-Commit Closed Correctness Criteria

A correctness criteria P is prefix-commit closed if:

- whenever P is true of H ,
 P is also true of $C(H')$, for any prefix H' of H

Practical correctness criteria *must* be prefix-commit closed:

- assume H is correct and $C(H')$ is incorrect
- consider that the system might have failed in state H'
- upon restart, the stable database would be returned to the state $C(H')$
 - so $C(H')$ had better be correct!

View Equivalence – Intuition

Intuition:

- two histories are equivalent if they have the same effect on the database

But when do two histories have the “same effect”?

In two histories:

- if each transaction *reads* the same values:
 - its *writes* will be the same
 - its *outputs* will be the same
- if the *final write* on each data item is the same:
 - the stable database will be in the same state
 - $w_1[x] \rightarrow w_2[x] \rightarrow w_3[x] \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$
 - $w_2[x] \rightarrow w_1[x] \rightarrow w_3[x] \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$

View Equivalence – Definition

Two histories H and H' are *view equivalent* iff:

1. they are over the same set of transactions, and have the same operations
 2. for any T_i and T_j such that $a_i, a_j \notin H$, and for any x :
 - (a) if T_i reads x from T_j in H
then T_i reads x from T_j in H'
 - (b) for each x , if $w_i[x]$ is the *final write* of x in H
then it is also the final write of x in H'
2. or, in English:
- (a) the *reads-from* relationships are the same
 - (b) the *final writes* are the same

View Serialisability (VSR) – Example 1

Example:

- $w_1[x] \rightarrow w_2[x] \rightarrow w_2[y] \rightarrow c_2 \rightarrow w_1[y] \rightarrow c_1 \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow c_3$
- view equivalent to T_1, T_2, T_3
- yet prefix:
 - $w_1[x] \rightarrow w_2[x] \rightarrow w_2[y] \rightarrow c_2 \rightarrow w_1[y] \rightarrow c_1$
 - is view equivalent to neither T_1, T_2 nor T_2, T_1

Hence, this example is not view serialisable

View Serialisability (VSR)

View serialisability:

A history H is *view serialisable* iff:

- for any prefix H' of H :
 - $C(H')$ is view equivalent to some serial history

Notes:

- without “for any prefix H' ”, VSR would not be prefix-commit closed
- the equivalent serial history need not be the same for every prefix
- $SR \subset VSR$
exercise: prove this!

VSR Example – 2

Example:

$w_1[x] \rightarrow w_2[x] \rightarrow w_2[y] \rightarrow c_2 \rightarrow w_1[y] \rightarrow w_3[x] \rightarrow w_3[y] \rightarrow c_3 \rightarrow w_1[z] \rightarrow c_1$

This example is VSR:

- to c_1 , it is view equivalent to T_1, T_2, T_3
- to c_3 , it is view equivalent to T_2, T_3
- to c_2 , it is view equivalent to T_2
- however, it is not conflict serialisable

However:

- scheduling *all* and *only* VSR schedules is NP-Complete
- therefore, most practical schedulers are conflict based

Summary

