Concurrency Control

Outline

- Concurrency control
  - Lock-Based Protocols (Two-Phase Locking)
  - Timestamp-Based Protocols
  - Multiversion Protocols (Snapshot Isolation)

Concurrency Control

- **Concurrency-control schemes** allow us to create, for a set of concurrent transactions, only acceptable schedules.
  - Isolation property may be weakened
  - They introduce an overhead

  Tradeoff: amount of concurrency vs. amount of overhead

- **Goal**: Ideally, to reach the largest concurrency level whereas ensuring that executed schedule is:

  serializable + recoverable + cascadeless

- A variety of concurrency-control schemes exists
  No one is clearly the best
  - Most used ones: two-phase locking and snapshot isolation

Lock-Based Protocols

- Access to data items in a mutually exclusive way
  - For a transaction to access an item, it needs to hold a **lock** on it
  - **Idea**: Hold the lock *long* enough to ensure serializability, but *short* enough to enhance concurrency

- Transactions can hold 2 **types of lock** on data :
  a) **Shared** (S): Data item can only be *read* (no writing allowed).
     - **Several transactions** can have an S-lock on the same data item
  b) **Exclusive** (X): Data item can be both *read and written*.
     - **Only one transaction** can have an X-lock on a data item at the same time
  - If a lock holds on a data point *p*, another transaction may be granted a lock on it if the requested lock type is compatible:

	S	X
S	true	false
X	false	false

Compatibility table

Lock-Based Protocols

- E.g., bank transaction:

T_1: **lock-X**(B);
read(B);
B := B – 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A);

T_2: **lock-S**(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display(A+B);

T_1	T_2	concurrency-control
lock-X(B)		
		grant-X(B, T_1)
read(B)		
B := B – 50		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display(A + B)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
A := A + 50		
write(A)		
unlock(A)		

- Simply locking is **not sufficient** to guarantee *serializability*
  - Unlock of B is too early
  - T_2 might display an inconsistent A+B value

Lock-Based Protocols

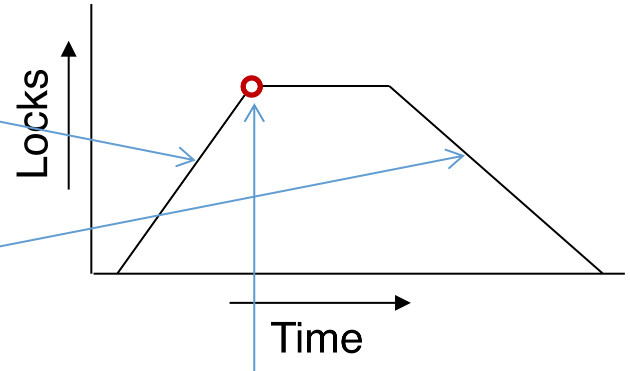
- E.g., bank transaction:

T_3:	**lock-X**(B);	T_4:	**lock-S**(A);
	read(B);		**read**(A);
	$B := B - 50$;		**lock-S**(B);
	write(B);		**read**(B);
	lock-X(A);		**display**($A+B$);
	read(A);		**unlock**(A);
	$A := A + 50$;		**unlock**(B);
	write(A);		
	unlock(B);		
	unlock(A);		

- Now, T_4 won't print inconsistent results!

Two-Phase Locking Protocol

- Protocol that ensures *conflict-serializable schedules*
    - Two phases to issue lock/unlock requests:
        1. **Growing Phase**
            - Transaction can obtain locks
            - Transaction **cannot** release locks
        2. **Shrinking Phase** (after 1st lock release)
            - Transaction **cannot** obtain locks
            - Transaction can release locks



- **Lock point**: point where the last lock was acquired
    - Transactions can be *ordered in terms of their lock points*
    - This is a serializability ordering for the transactions!

Two-Phase Locking Protocol

- Susceptible of *deadlocks*.
  - **Deadlock**: no transaction can finish, as they mutually lock each other out
    - *Necessary evil*: deadlocks are preferable to inconsistent states
    - They can be handled by rolling back one of the transactions
- Susceptible of *starvation*.
- Susceptible of *cascading rollbacks*.

Deadlock

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Two-Phase Locking Protocol

- Susceptible of *deadlocks*.
- Susceptible of *starvation*.
  - **Starvation**: when a transaction T_1 requests an **X-lock** and needs to wait for a previous transaction's **S-lock** to be released, new **S-lock** requests of other transactions could advance T_1 and block its access to the data
  - Possible solution: use two conditions to grant a lock on a data point *p*:
    1. There is no other transaction holding an *incompatible* lock on *p*
    2. There is no other transaction waiting for a lock on *p*

Remember compatibility table

- Susceptible of *cascading rollbacks*.

Two-Phase Locking Protocol

- Susceptible of *deadlocks*.
- Susceptible of *starvation*.
- Susceptible of *cascading rollbacks*.

  Possible solutions:

  - **Strict two-phase locking**: a transaction must hold *all its exclusive locks* until it commits/aborts.
    - Ensures recoverability and avoids cascading rollbacks
  - **Rigorous two-phase locking**: a transaction must hold *all locks* until it commits/aborts.
    - Transactions can be serialized in the order in which they commit.

Two-Phase Locking Protocol

- A transaction might need different lock types for the same data item in different moments.
- 2PL can sue **lock conversions**
  - **Upgrade** from a lock-S to a lock-X
    - Only during growing phase
  - **Downgrade** from a lock-X to a lock-S
    - Only during shrinking phase

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Strict/rigorous two-phase locking, with **lock conversions**, are usually implemented in commercial DBMS

Implementing 2PL

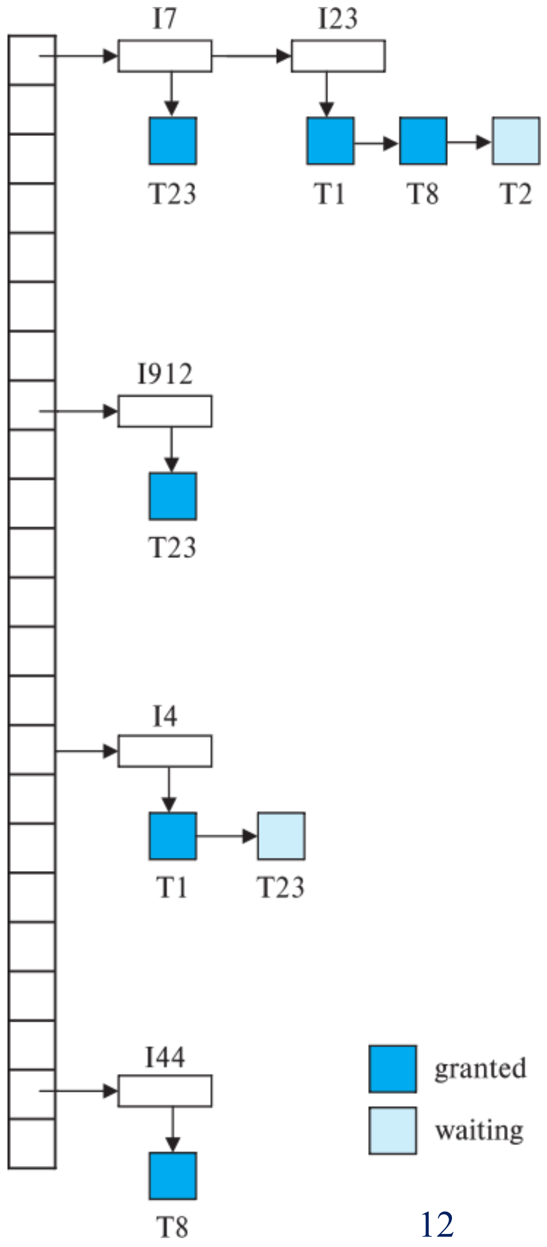
Automatic **generation of lock/unlock** requests

- When a transaction T_i makes a **read**(p),
  - it issues a **lock-S**(p) request
- When a transaction T_i makes a **write**(p),
  - if it holds a **lock-S**(p), it issues an **upgrade**(p) request;
  - Otherwise, it issues a **lock-X**(p) request
- All acquired locks are released when T_i commits/aborts

Implementing 2PL

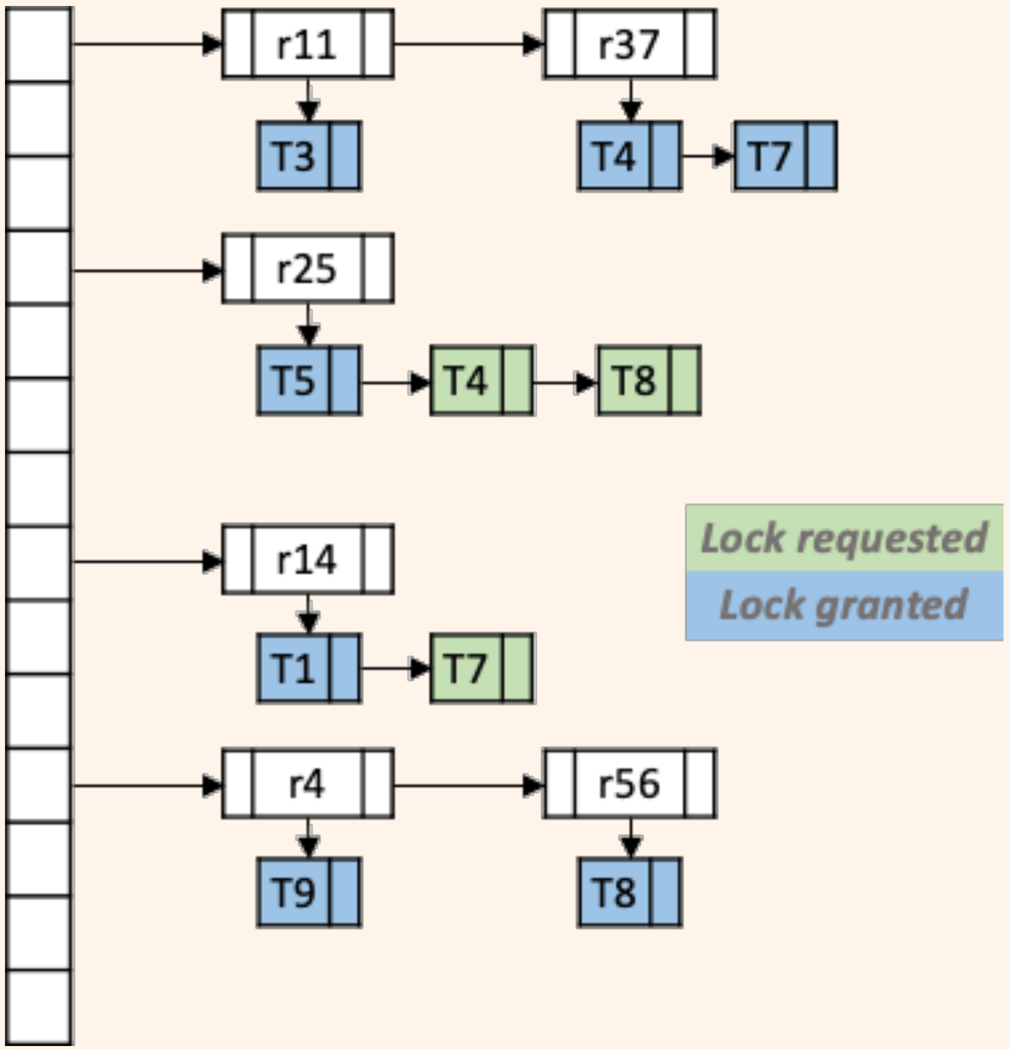
- **Lock manager**: receives lock/unlock requests and answers with a:
  A transaction waits until its request is answered
  - Lock grant
  - Order to roll back (deadlock)

- Uses a **lock table**: hash structure to save granted locks and pending requests
  - A queue for each data item (I_x)
  - **Lock request** for I_x is appended to its queue of requests, and granted if it is compatible with any previous lock
  - **Unlock request** leads to the deletion of the request, and checking if any request waits to be granted



Exercise

- What happens if T_9 requests a S-lock on item r_{37}?
- Can T_7 request a X-lock on item r_4?
- What happens if T_3 requests a X-lock on item r_8?
- What happens if T_5 requests a X-lock on item r_{56}?



Outline

- Concurrency control
  - Lock-Based Protocols (Two-Phase Locking)
  - Timestamp-Based Protocols
  - Multiversion Protocols (Snapshot Isolation)

Timestamp-Based Protocols

- **Each transaction** T_i is given a timestamp TS(T_i) as it enters the system.
  - Use system clock or a logical counter

- Manage concurrent execution such that

  time-stamp order = **serializability order**

- **Each data point** p, is given **two** timestamps:
  - W-TS(p): largest timestamp of any transaction that executed a **write**(p) successfully.
  - R-TS(p): largest timestamp of any transaction that executed a **read**(p) successfully.

Timestamp-Ordering Protocol

- **Timestamp ordering (TSO)** protocol ensures that *conflicting operations* are executed in timestamp order.
- Transaction T_i can perform a **read**(p) only if no newer transaction has modified p
  - If TS(T_i) < W-TS(p): read is *rejected*, and T_i is rolled back
  - If TS(T_i) ≥ W-TS(p): read is *executed*, and update
    R-TS(p) := max[R-TS(p) ; TS(T_i)]

Rolled back transactions are assigned new TSs

- T_i can perform a **write**(p) only if no newer transaction has manipulated (read or write) p
  - If TS(T_i) < R-TS(p) *or* TS(T_i) < W-TS(p): write is *rejected*, and T_i is rolled back
  - Otherwise, write is *executed*, and update
    W-TS(p) := TS(T_i)

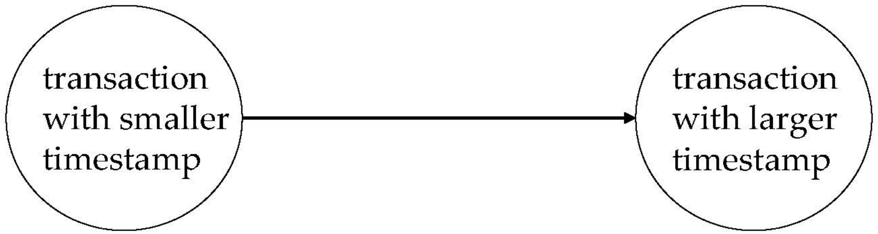
Exercise: TSO

- Is this schedule valid under TSO?
- Assume:
  - R-TS(A) = W-TS(A) = 0
  - R-TS(B) = W-TS(B) = 0
  - TS(T_{25}) = 25
  - TS(T_{26}) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Timestamp-Ordering Protocol

- TSO protocol guarantees serializability since all the arcs in the precedence graph are of the form:



- Free of deadlocks, as no transaction ever waits.
- Schedules may **not** be *cascade-free* and even **not** *recoverable*.

Outline

- Concurrency control
  - Lock-Based Protocols (Two-Phase Locking)
  - Timestamp-Based Protocols
  - Multiversion Protocols (Snapshot Isolation)

Multiversion Protocols

- Issue: Read-only transactions that consult large amounts of data might conflict with update transactions
  Poor performance

- Multiversion schemes maintain different **versions** of data points:
  - Each **write**(*p*) creates a new version of the data point, *p* .
  - A **read**(*p*) selects carefully the appropriate version of *p* , to ensure serializability.
    - Never waits, as the appropriate version is returned immediately.

- Several **variants**:
  - Multiversion Two-Phase Locking
  - Multiversion Timestamp Ordering
  - **Snapshot isolation**

Snapshot Isolation

Widely used in DBMS
(commercial/ open-source)

- Each transaction receives a "**snapshot**" (version) of the DB when it begins and operates locally
  - Transactions which modify the data need to be validated
  - After validation, allowed to commit and update the DB
    - Writing in the DB after commit needs to be an *atomic action*

- It uses timestamps too:
  - Each transaction T_i :
    - **StartTS**(T_i): when T_i started
    - **CommitTS**(T_i): when T_i requests validation
  - Each snapshot p_k for T_k:
    - **TS**(p_k) = **CommitTS**(T_k)

Snapshot Isolation

- When a transaction T_i reads a data point p, it uses the latest version p_k such that $TS(p_k) \leq StartTS(T_i)$
  - T_i doesn't see updates of transactions committed after T_i started

Initial state: $X_{(0)}$ = 100 € and $Y_{(0)}$ = 25 €	
T_1 #deposit 50€ in Y	***T_2*** # withdraw 50€ from X
R(X) # $X_{(0)}$=100 R(Y) # $Y_{(0)}$=25	
	R(X) # $X_{(0)}$=100 R(Y) # $Y_{(0)}$=25 W(X := 50) # $X_{(2)}$=50
W(Y := 75) # $Y_{(1)}$=75 R(X) # $X_{(0)}$=100 R(Y) # $Y_{(1)}$=75	
	R(Y) # $Y_{(0)}$=25
Final state: $X_{(2)}$ = 50 € and $Y_{(1)}$ = 75 €	

Concurrent updates not visible

Only own updates are visible

Snapshot Isolation

- T_i and T_j are **concurrent transactions** if:
  - StartTS(T_j) ≤ StartTS(T_i) ≤ CommitTS(T_j), **or**
  - StartTS(T_i) ≤ StartTS(T_j) ≤ CommitTS(T_i)

- **Validating** a transaction for committing needs care:
  - 2+ *concurrent transactions* might have modified the same data point *p*.
  - Which one persists? Two strategies:
    - **First committer wins**
    - First updater wins

Snapshot Isolation

- **First-committer wins** rule:

  T_i commits its copy p_i only if no other concurrent transaction T_k already wrote p with its own copy p_k

- Procedure

  1. Check if data item p was written by a concurrent transaction:
     $\exists k$: StartTS(T_i) $\leq$ TS(p_k) $\leq$ CommitTS(T_i) ?
  2. If so, T_i aborts
  3. If not, T_i commits and p_i writes to the DB

Initially: X=0 , Y=0 , Z=0

T_1	T_2	T_3
W(Y := 1)		
Commit		
	Start	
	R(X) # X=0	
	R(Y) # Y=1	
	W(Y := 2)	
		W(X := 2)
		W(Z := 3)
		Commit
	R(Z) # Z=0	
	R(Y) # Y=2	
	W(X := 3)	
	Commit-Req	
	Abort	

Previous updates are visible
Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T_2 is rolled back

Exercise SI

- Identify the values read by each operation of all the transactions

- Which commit request will be granted? Is there any that will be refused (+aborted)?

Before...: X=0 , Y=0 , Z=0

T_1	T_2	T_3
Start		
W(*Y* := 1)		
W(X := 2)		
	R(*X*)	
	R(*Z*)	
	W(*Y* := 2)	
		Start
		R(*Y*)
	W(*Z* := 3)	
		W(*Y* := 3)
		R(*Y*)
R(Y)		
		R(*Z*)
R(Z)		
		Commit-Req
	Commit-Req	
Commit-Req		

Snapshot Isolation

- Useful when large reads are common
  - Reads are *never* blocked, and don't interfere with updates
- Avoids:
  - *Dirty reads* (reads uncommitted data)
  - *Lost updates* (updates overwritten by another transaction)
  - *Non-repeatable reads* (two reads of T_i see different values)

- SI does not ensure serializability!!
  - E.g., a serializable schedule would end up with *A*=*B.*
    - SI swaps the values of *A* and *B*
    - Note the cycle in the precedence graph!
    - This problem is known as **write skew**
  - Extension: Serializable snapshot isolation (SSI)

Set A = 3 and B = 17

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Concurrency Control

Deadlock Handling

- **Deadlock** state refers to a *set of transactions* in which all them are waiting for another transaction in the set.
  - Solution: rolling back (partially) some of the transactions

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Two main methods:
  - **Prevention protocols** ensure that the system will never enter a deadlock state
    - Useful if deadlocks are usual
  - **Detection-recovery scheme**: the system is allowed to enter in a deadlock state, but it tries to recover from it
    - More efficient if deadlocks are unusual
    - Involves overheads (run-time cost and potential losses)

Deadlock Handling: Prevention

Deadlock prevention strategies:

- Place locks intelligently
  - Transactions lock all its data items before it begins
  - You always lock data items placed after other items previously locked (ordering)
    - Needs to know which data items are going to be used
- Rollback instead of wait if expecting a deadlock
- Lock timeouts

Deadlock Handling: Prevention

Deadlock prevention strategies:

- Place locks intelligently
- Rollback instead of wait if expecting a deadlock
  - When T_i requests a lock on *p*, already locked by T_j:
    - **wait-die**: T_i waits if it is older than T_j ; or rolls back otherwise
    - **wound-die**: T_i waits if it is younger than T_j ; or rolls backs otherwise
  - Might cause many unnecessary rollbacks
- Lock timeouts

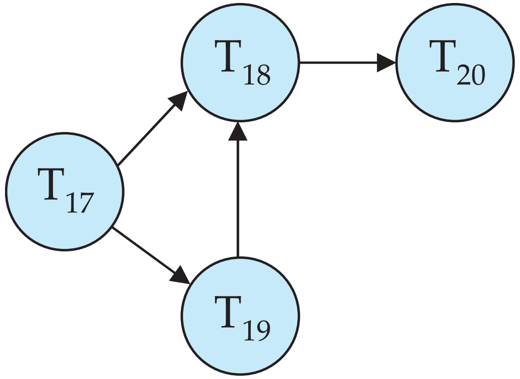
Deadlock Handling: Prevention

Deadlock prevention strategies:

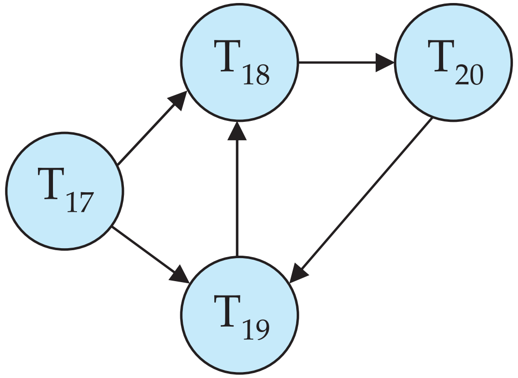
- Place locks intelligently
- Rollback instead of wait if expecting a deadlock
- **Lock timeouts** (less common, although easy to implement)
  - A transaction requests a lock and waits for **at most** a specific time.
    - If lock is not granted within that time, transaction *rolls itself back*
  - Might end up in *starvation*
  - Hard to decide **time** for timing out

Deadlock Handling: Detection-Recovery

- Periodically check for deadlocks and, if there is any, try to recover
- Use a ***wait-for* graph** to describe deadlocks:
  - *Vertices:* transactions
  - *Edges* ($T_i \rightarrow T_j$): if T_i is waiting for a lock held by T_j
  - *Deadlock state*: *iff* there is a cycle in the graph



Wait-for graph



Deadlocked wait-for graph

  - *Deadlock-detection*: look for cycles in the graph
    - Periodicity depends on deadlocks' frequency

Deadlock Handling: Detection-Recovery

- *Recovery*, in case of deadlock detected:
  - **Victim selection**: transaction(s) to roll back (the one that incurs in "minimum cost")
  - **Extent**: how far to roll back victim transaction?
    - **Total rollback**: Abort the transaction and restart it.
    - **Partial rollback**: Roll back only as far as necessary to break deadlock.
      - Release locks that other transactions are waiting for.
  - *Starvation* can happen if the victim is always the same!
    - Sol. 1: never select oldest transaction as the victim
    - Sol. 2: include no. rollbacks in cost function (*most common*)