Transaction Management

Transaction ?

Recovery process?

Failure?

Withdraw Money

Enter PWD

Enter account Info

Withdraw 5000

Failure

Balance updated ?

© Can Stock Photo - csp3931903

Withdraw 5000

Transaction

Enter PWD

DB

Enter account Info

Withdraw 5000

Outline

- Transaction basics
- Concurrency control
- Recovery management
- Transaction design issues
- Workflow management

Transaction Definition

- Supports daily operations of an organization
- Become more important with the growth of the internet

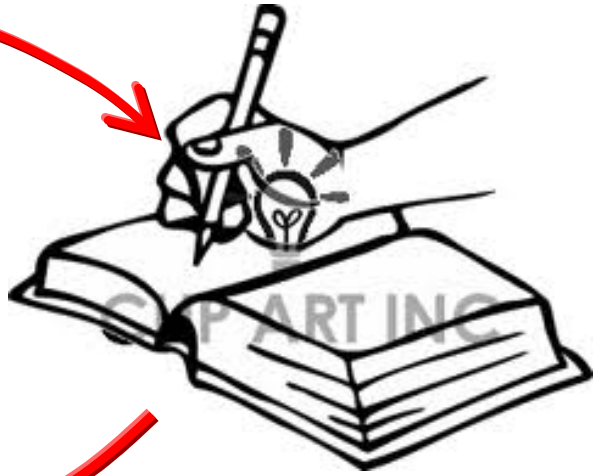
Transaction Definition

- Collection of database operations
- Reliably and efficiently processed as **one unit of work**
- No lost data
  - **Interference among multiple users**
  - **Failures** (operating system, program, disk, …)

Library Example

10 copies of
a database book

1.borrow



3.return

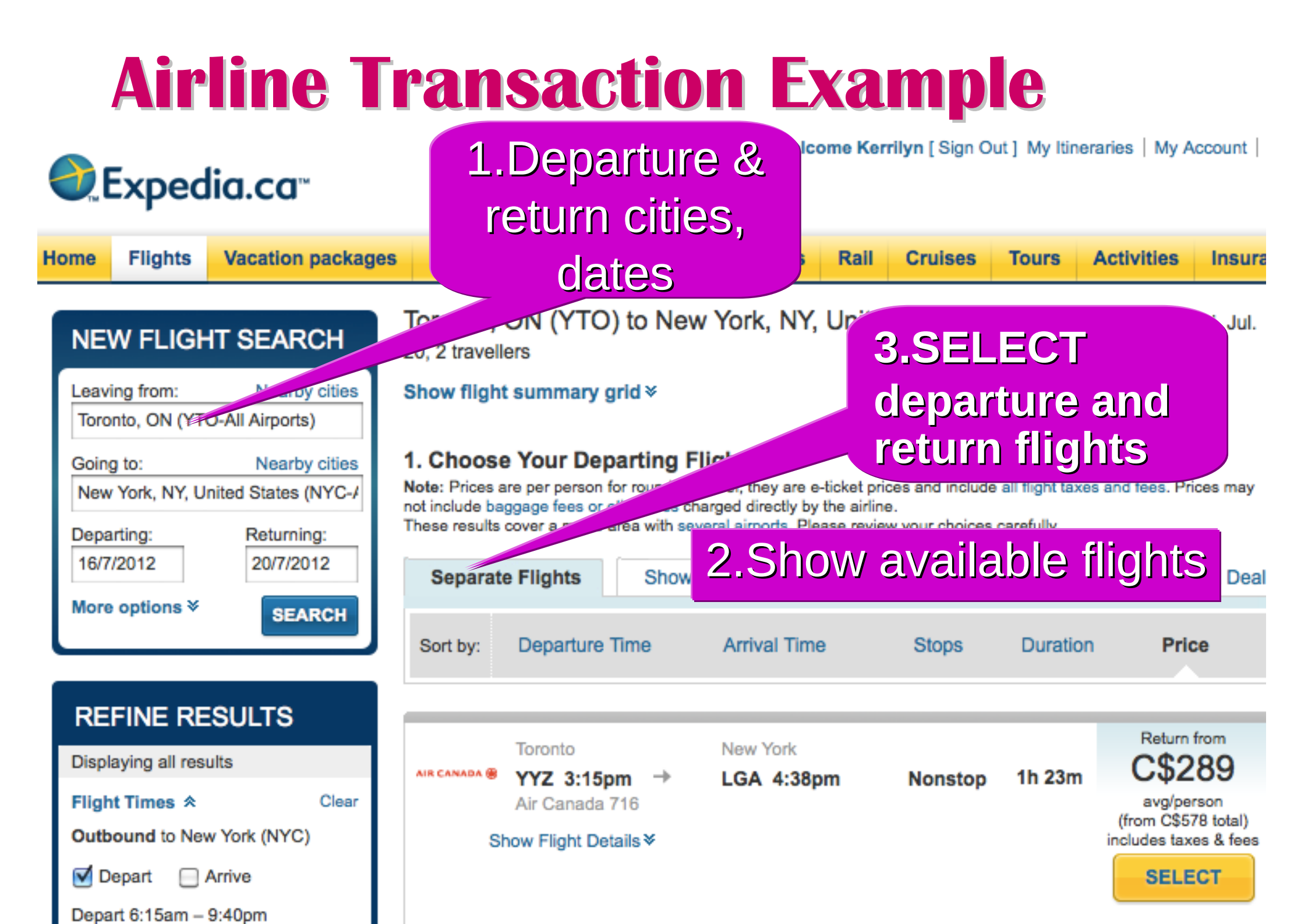
2.Update

4.Update all

Library Example

- Borrow for read only
  - concurrently
- Borrow for update
  - one person at a time

Airline Transaction Example



New SQL statements:

- START TRANSACTION
  - defines beginning of transaction statements
  - some DBMSs omit
- COMMIT: end of transaction
- ROLLBACK
  - Undo command
  - If any error occurs, all changes are deleted from the database
- On Error: part of exception handling

Airline Transaction Example

START TRANSACTION

Display greeting

Get reservation preferences from user

SELECT departure and return flight records

If reservation is acceptable then

UPDATE seats remaining of departure flight record

UPDATE seats remaining of return flight record

INSERT reservation record

Print ticket if requested

End If

On Error: ROLLBACK

COMMIT

ATM Transaction Example

START TRANSACTION

Display greeting

Get account number,pin,type,amount

SELECT account number,type,balance

If **balance is sufficient** then

UPDATE account by posting debit

INSERT history record

Display message and **dispense cash**

UPDATE history record

Print receipt if requested

End If

On Error: **ROLLBACK**

COMMIT

If fail?

Transaction Properties (ACID)

- Atomic
- Consistent
- Isolated
- Durable

Transaction Properties (ACID)

- **Atomic: all or nothing**
  - **all changes are made or none are made**
- **Consistent: database must be consistent before and after a transaction**
  - by the time any transaction ends, each and every reference in the database must be valid.

example

Transaction Properties (ACID)

- Enforce referential integrity:
  - If a transaction consisted of an attempt to delete a record referenced by another, each of the followings would maintain consistency:
    - **abort** the transaction, rolling back to the consistent, prior state; or,
    - **delete ALL records** that reference the deleted record (*cascade delete*); or,
    - **nullify the relevant fields** in **ALL records** that point to the deleted record.

propagation constraints

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacSSN	OffDays
1111	IS320	SUMMER	2006	BLM302	10:30:00		MW
1234	IS320	FALL	2005	BLM302			MW
2222	IS460	SUMMER	2005	BLM412			TH
3333	IS320	SPRING	2006	BLM214			W
4321	IS320	FALL	2005	BLM214			TH
4444	IS320	WINTER	2006	BLM302	1	543-21-0987	TTH
5555	FIN300	WINTER	2006	BLM207	0	765-43-2109	MW
5678	IS480	WINTER	200				
5679	IS480	SPRING	200				
6666	FIN450	WINTER	200				
7777	FIN480	SPRING	200				
8888	IS320	SUMMER	200				
9876	IS460	SPRING	200				

Offering

Transaction: delete IS320

CourseNo	CrsDesc	CrsUnits
FIN300	FUN ENTALS OF FINANCE	4
FIN450	PRI IPLES OF INVESTMENTS	4
FIN480	CO PORATE FINANCE	4
IS320	FUNDAMENTALS OF BUSINESS PROGRAMMING	4
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

Course

Transaction Properties(ACID)

- **Isolated: no unwanted interference from other *users***
  - requirement that other operations cannot access data that has been modified during a transaction that has not yet completed
  - In case of concurrent transactions

Transaction Properties(ACID)

- **Durable:**
  - the ability of the DBMS to **recover the committed transaction** updates against any kind of system failure (HW or SW)
    - the data changes will survive system failure, and
    - all integrity constraints have been satisfied so the DBMS won't need to reverse the transaction.
  - implemented by writing transactions into a transaction log

Transaction Processing Services

- Concurrency control
- Recovery management
- Service characteristics
  - Transparent
    - Application programmer does not write code except SQL statements
  - Consume significant resources
  - Significant cost component
  - Transaction design important

Concurrency Control

- Problem definition
- Concurrency control problems
- Concurrency control tools

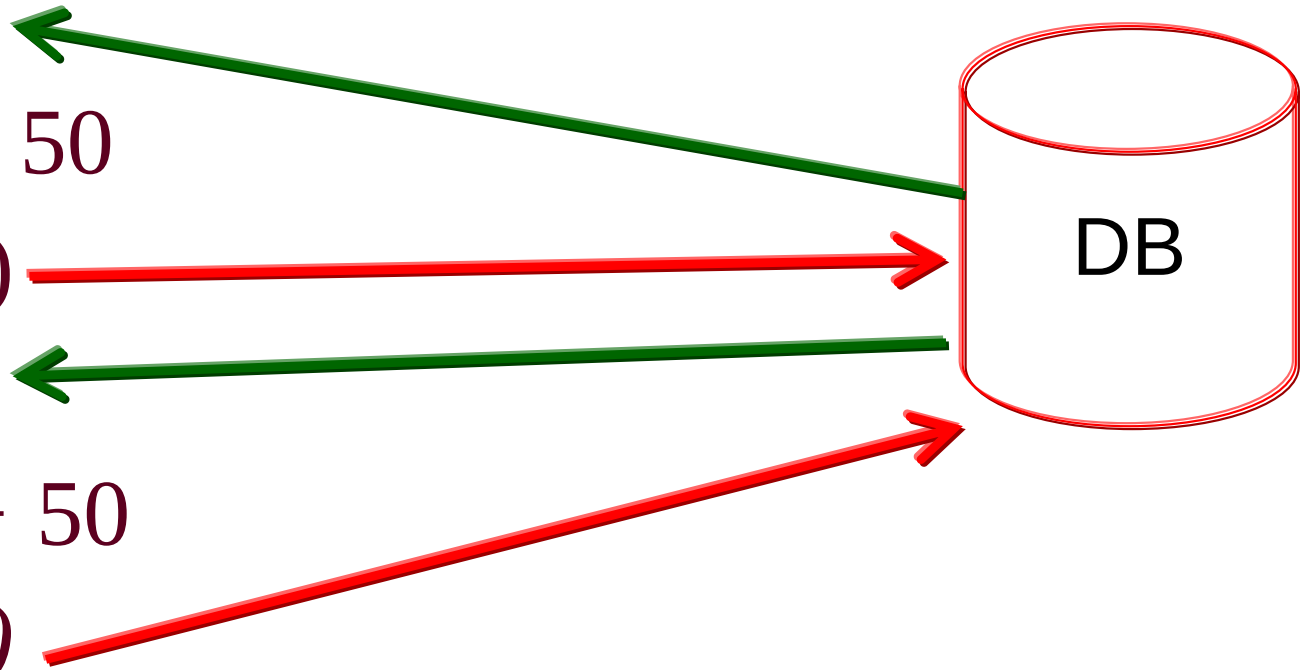
Concurrency Control Problem

- **Objective:**
  - Maximize the amount of work performed
  - Throughput: number of transactions processed per unit time(min)
    - Large organizations: thousands of transactions per minute
  - Generating schedules with the *Serializability* property.
    - **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions

Example of Fund Transfer

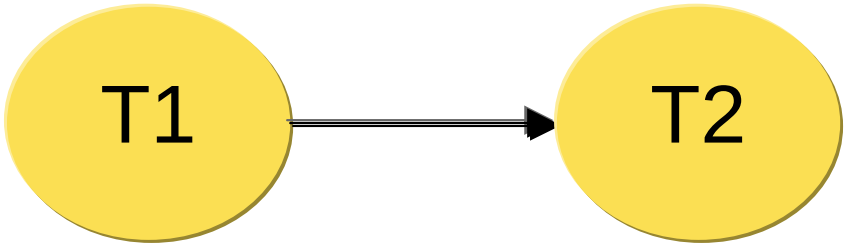
Transaction to transfer \$50 from account *A* to account *B*:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)



Serial Schedule

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)



Serializability

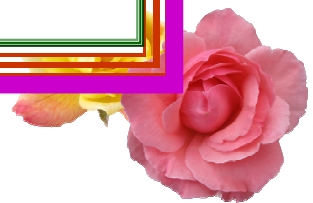
S1

T_1	T_2
read(A) $A := A - 50$ write (A)	
read(B) $B := B + 50$ write(B)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
	read(B) $B := B + temp$ write(B)

S2

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	
	read(B) $B := B + temp$ write(B)

Effects are the same



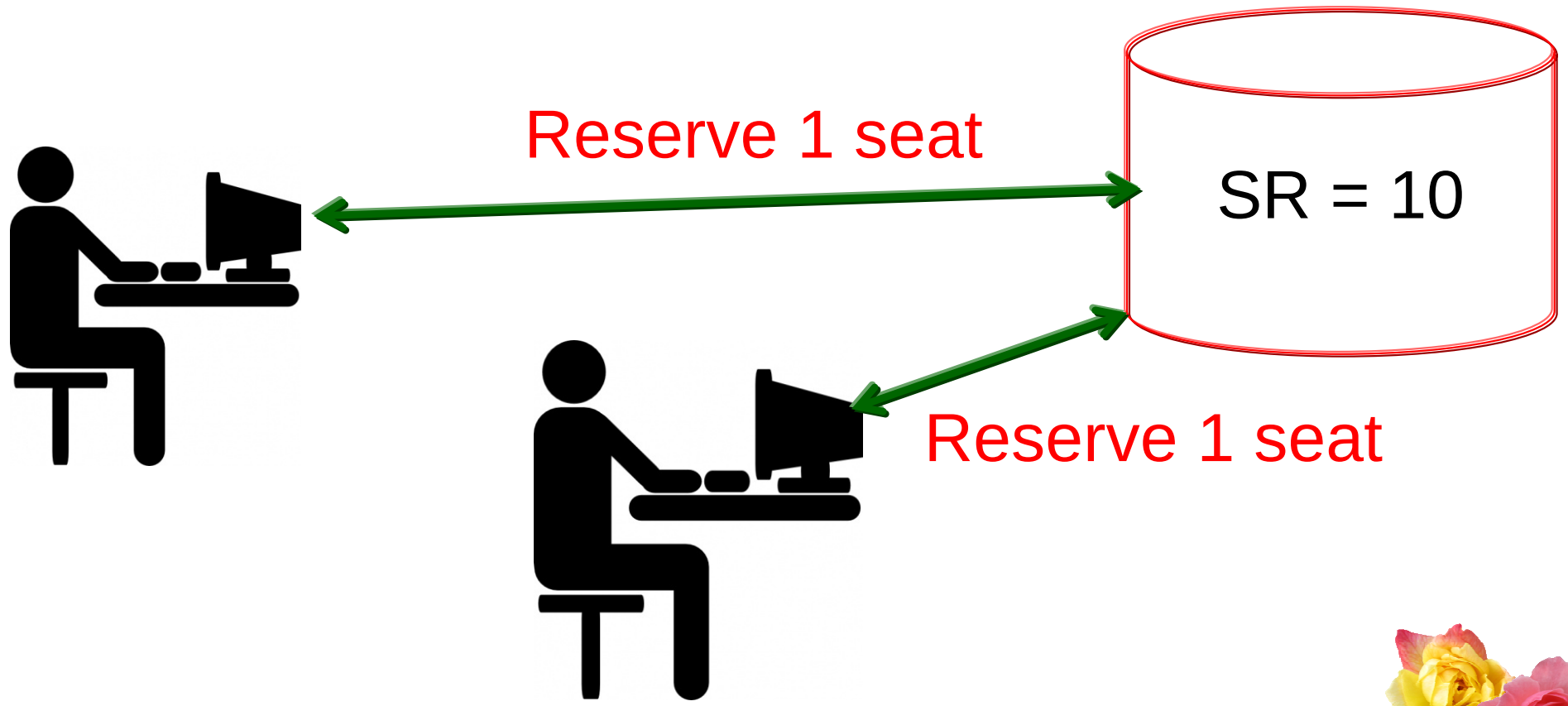
Concurrency Control Problem

- **Constraint:**
  - No interference:
    - **effect is the same as serial effect (1-by-1 execution)**
  - Interference occurs on commonly manipulated data known as hot spots
    - **Example: seats remaining in a flight record**
  - concurrency control must be applied to the entire database, not just the hot spots

Lost Update Problem

Two users trying to reserve the same flight:

need to change seats remaining (SR) field



Lost Update Problem

Two users changing the same part of the database

Transaction A	Time	Transaction B
Read SR (10)	T_1	
	T_2	Read SR (10)
If SR > 0 then SR = SR -1	T_3	
	T_4	If SR > 0 then SR = SR -1
Write SR (9)	T_5	
	T_6	Write SR (9)

SR: Seat Reserved column name *Write-Write problem*

Uncommitted Dependency Problem

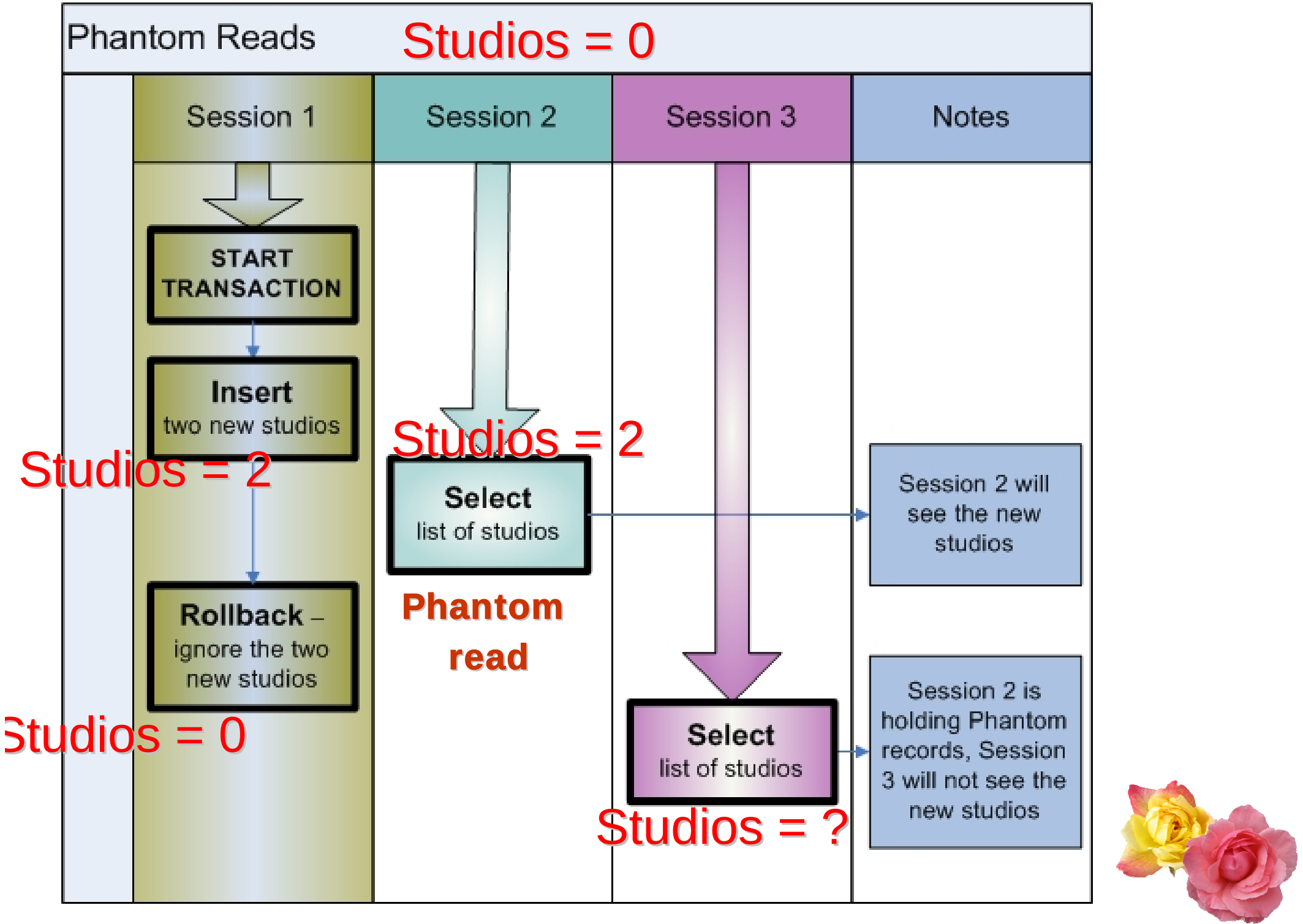
Transaction A	Time	Transaction B
Read SR (10)	T_1	
SR = SR - 1	T_2	
Write SR (9)	T_3	
	T_4	Read SR (9)
ROLLBACK ★	T_5	

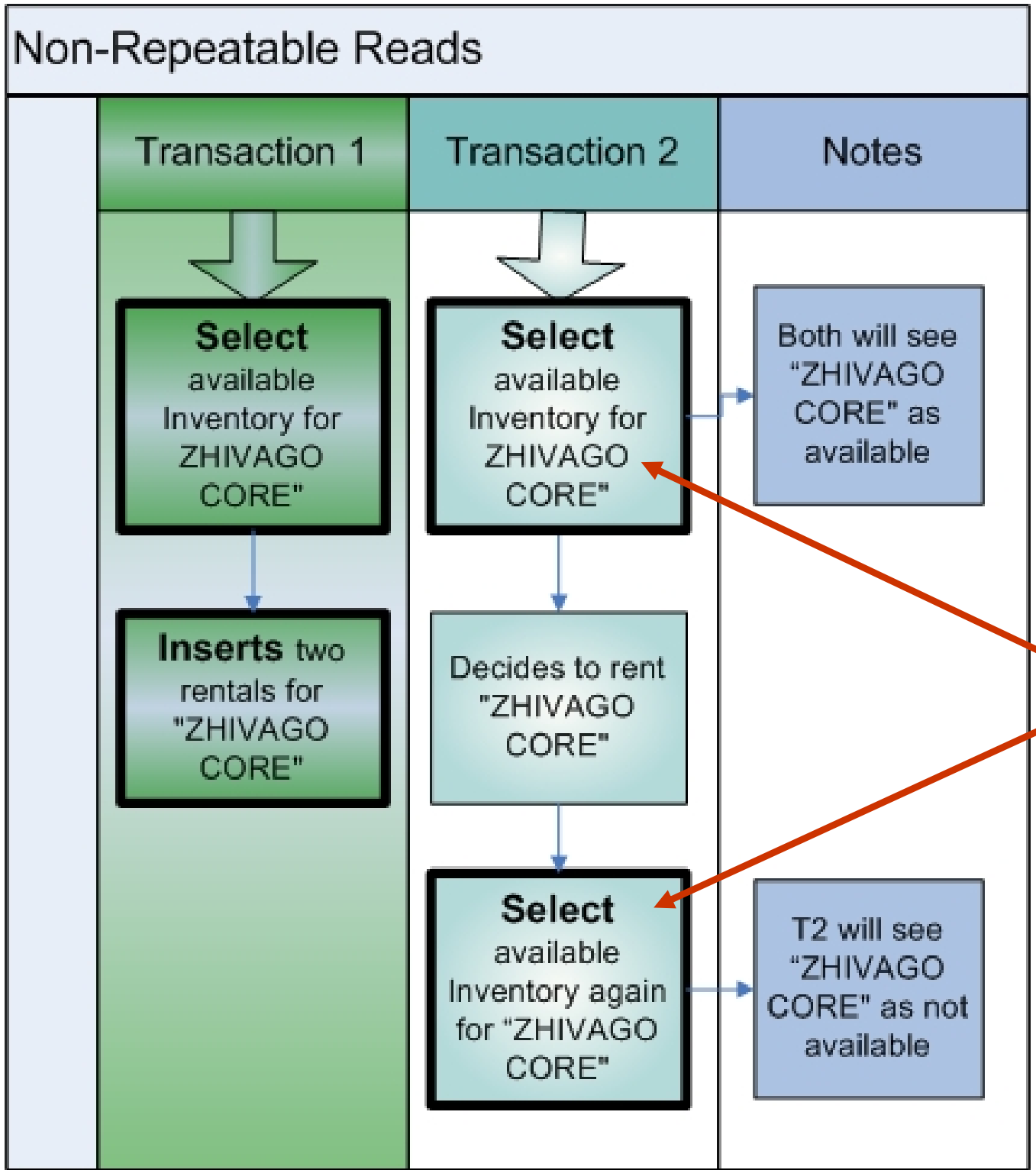
failure

the write- read problem

Inconsistent Retrieval Problems

- Interference causes inconsistency among multiple retrievals of a subset of data
- Incorrect summary
- Phantom read
- Non repeatable read


Phantom Reads
Studios = 0
Session 1
Session 2
Session 3
Notes
START TRANSACTION
Insert two new studios
Studios = 2
Rollback – ignore the two new studios
Studios = 0
Studios = 2
Select list of studios
Phantom read
Session 2 will see the new studios
Select list of studios
Studios = ?
Session 2 is holding Phantom records, Session 3 will not see the new studios

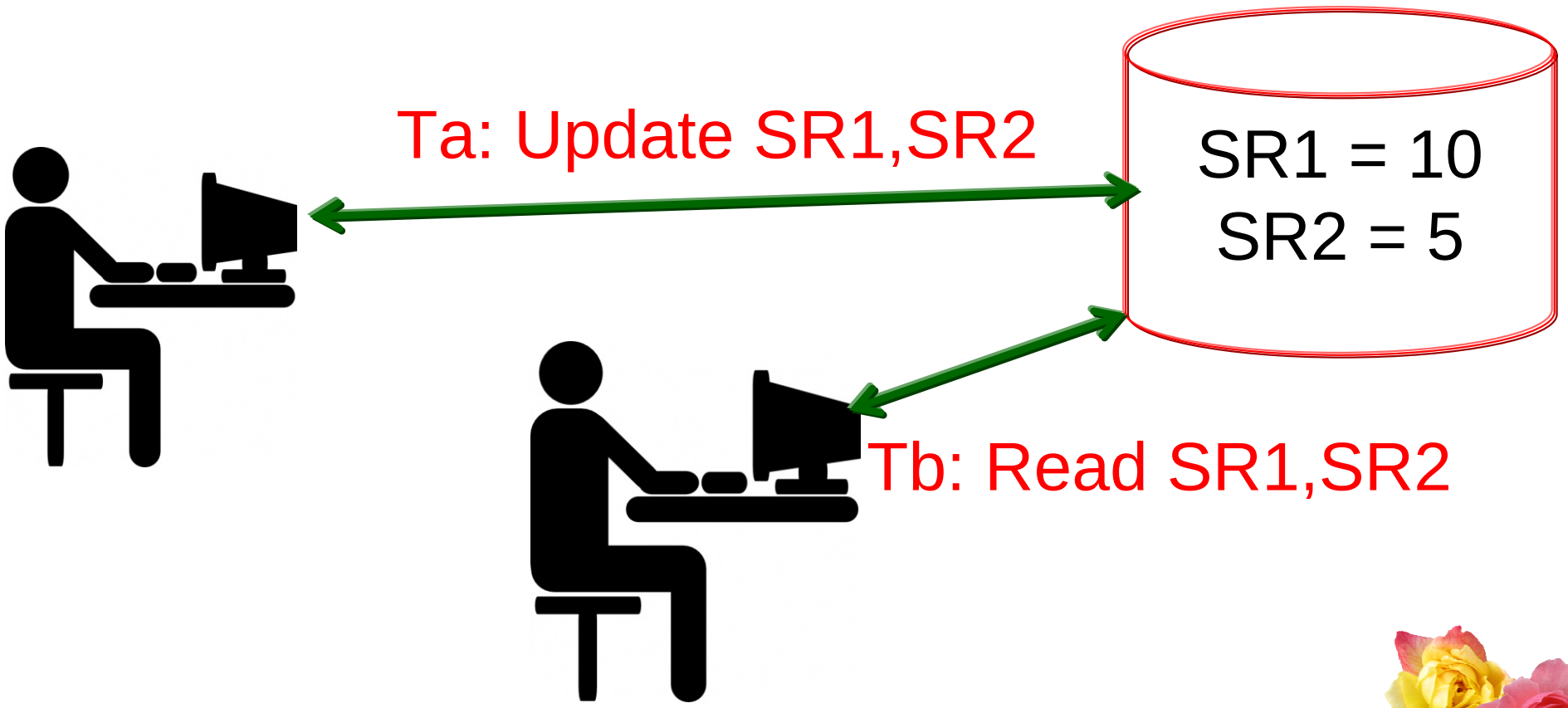


Non-repeatable read

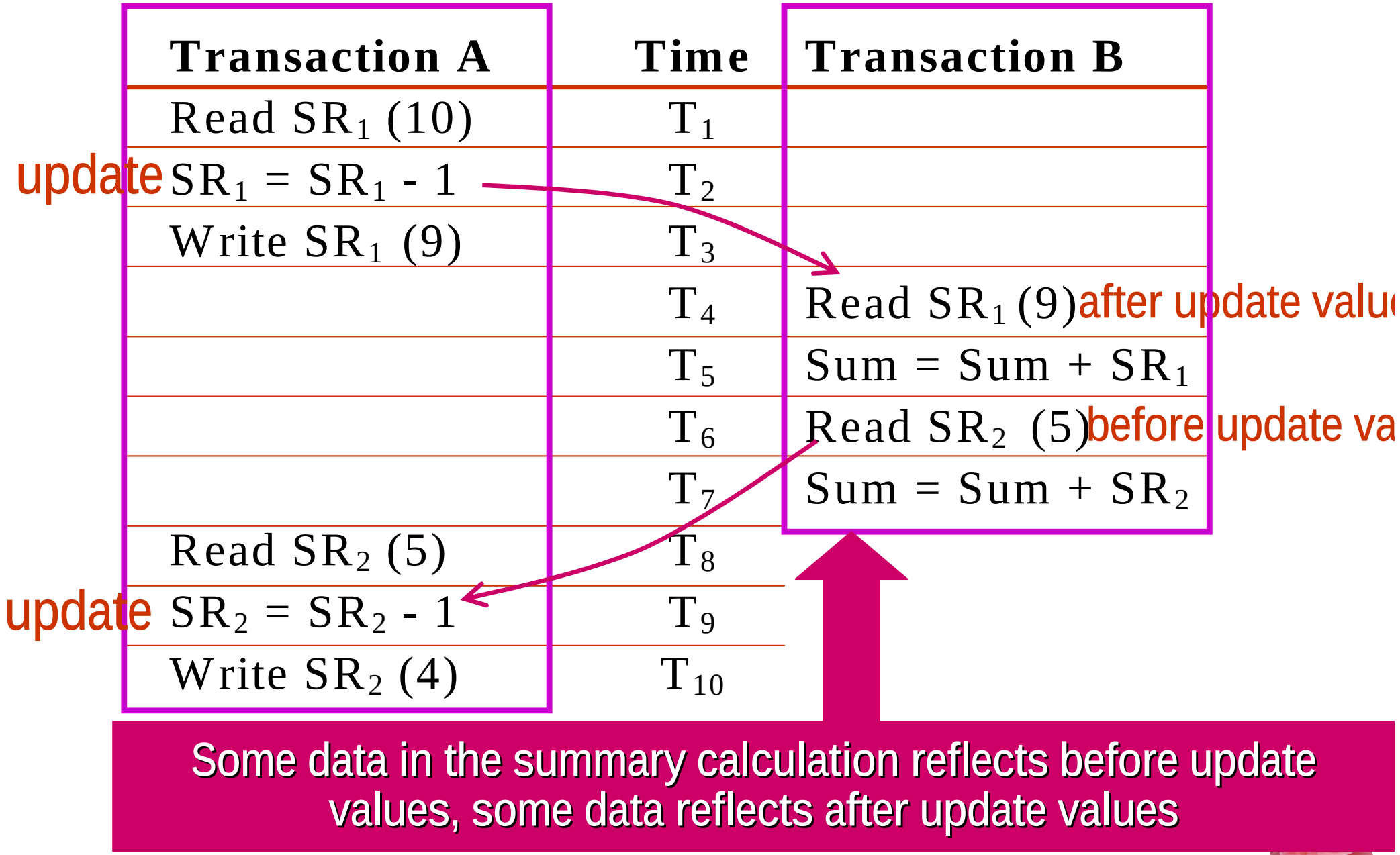
2 reads get diff results

Incorrect Summary Problem

Interference causes inconsistency



Incorrect Summary Problem



HOW TO
solve concurrency problems ?

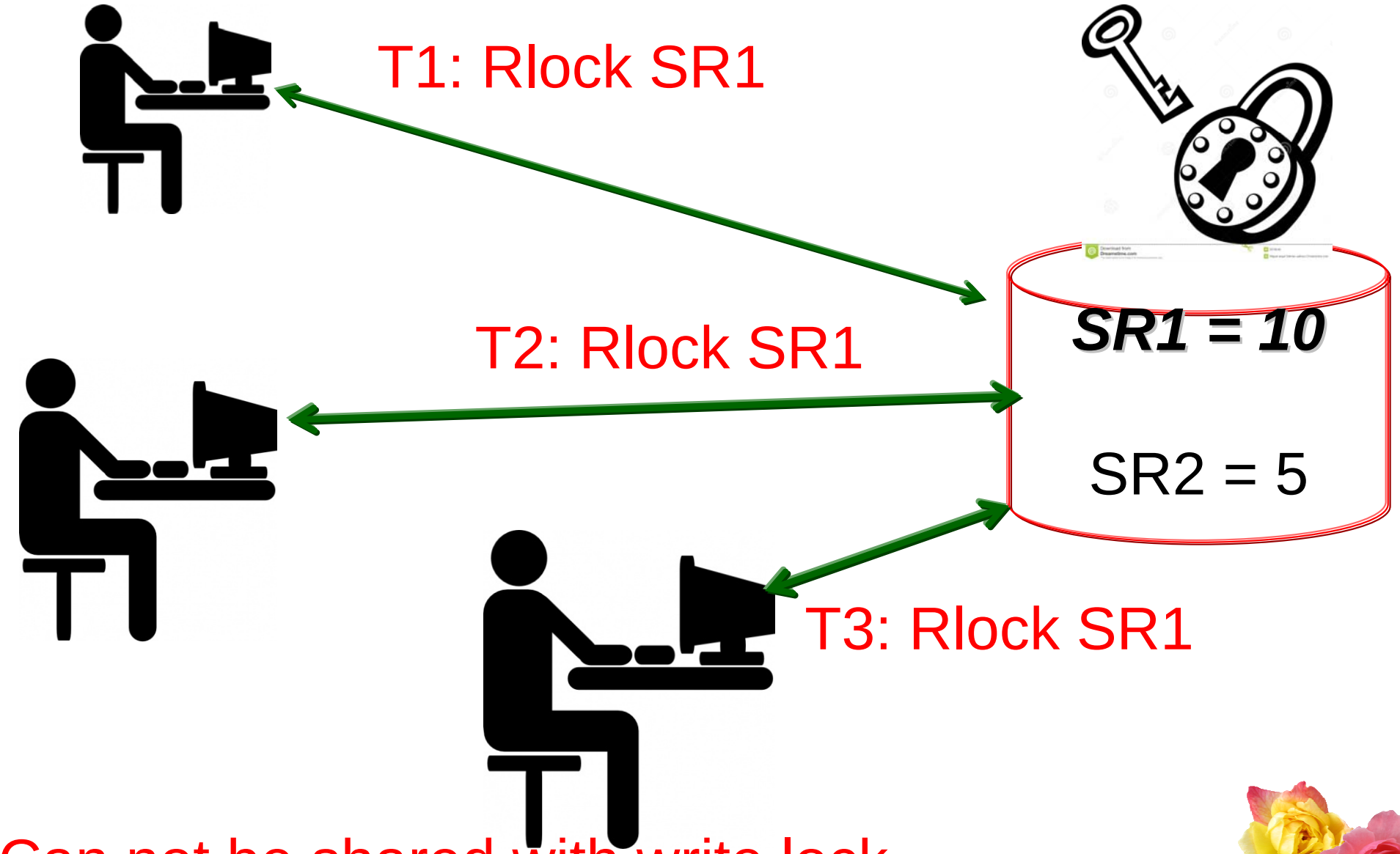
Locking Fundamentals

- Fundamental tool of concurrency control
  - Locks can be used to prevent interference problems
- **Obtain** lock before accessing an item
  - release lock when finish
- **Wait** if a conflicting lock is held
- Concurrency control manager maintains the lock table

Locking data

- **Shared lock:** a read lock
- **Exclusive lock:** a write lock
  - Not allow to share Xlock

Shared lock: a read lock



Can not be shared with write lock

Locking Fundamentals

- Conflicting locks
  - **Shared lock:** conflicts with exclusive locks:
    - **Write lock**
  - **Exclusive lock:** conflicts with all other kinds of locks:
    - **Write lock**
    - **Read lock**

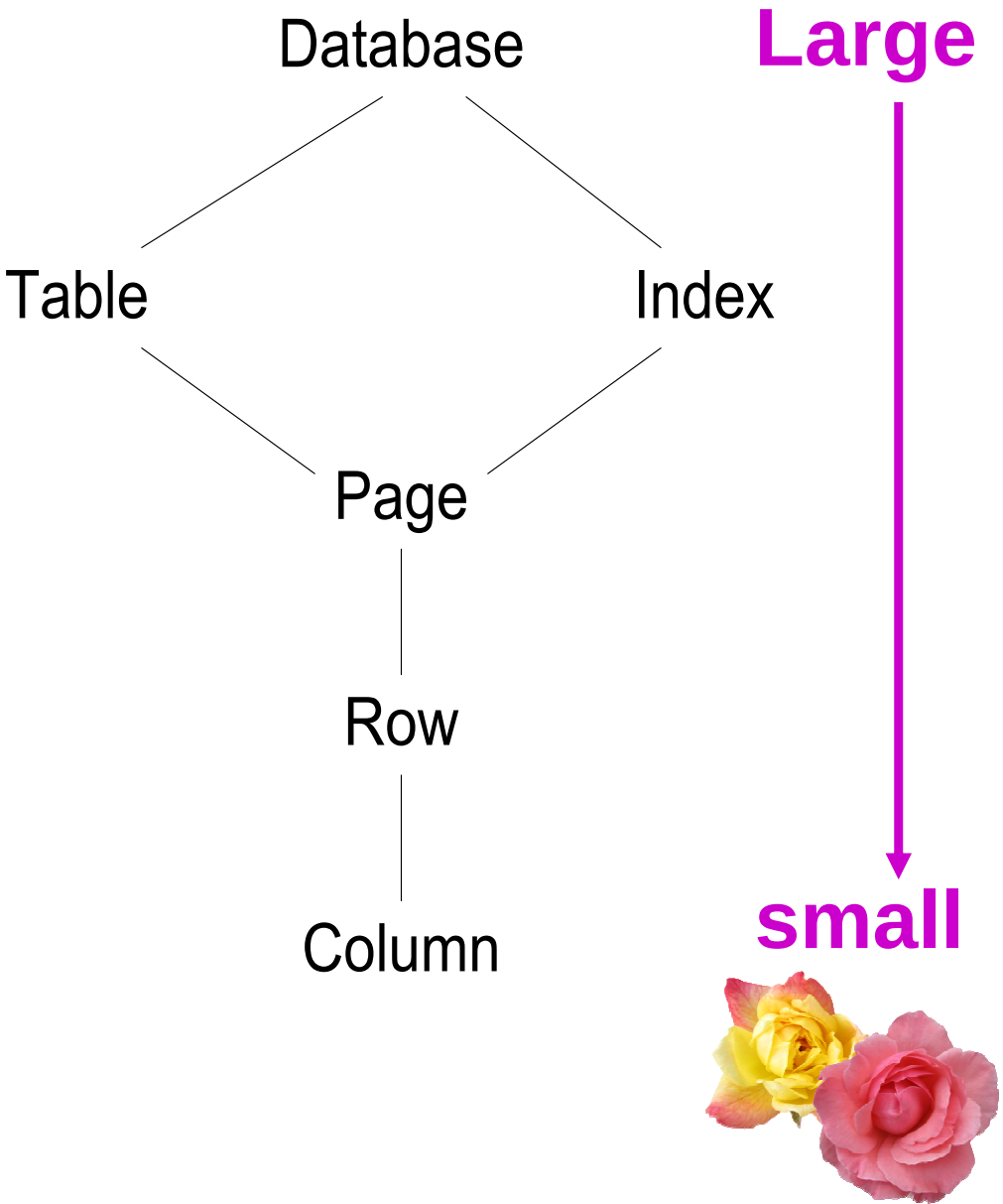
Locking Fundamentals

- Lock usage:
  - Obtain appropriate kind of lock before accessing database item
  - Wait if another transaction holds a conflicting lock
- Lock table:
  - Details about locks held by transactions
  - Concurrency control manager maintains
  - Lock: insert a lock record
  - Release (unlock): delete a lock record

Locking Granularity

(size of database item locked)

- Tradeoff:
  - Overhead (#of locks) versus waiting



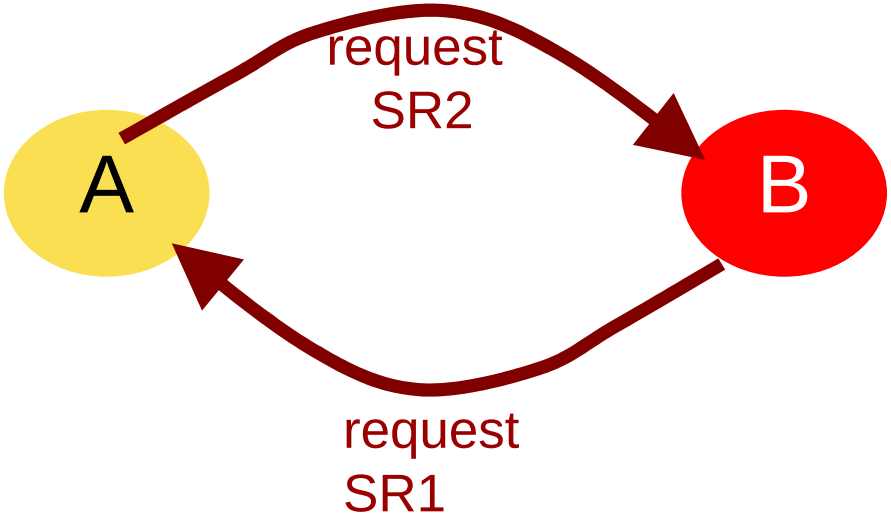
Locking Granularity?

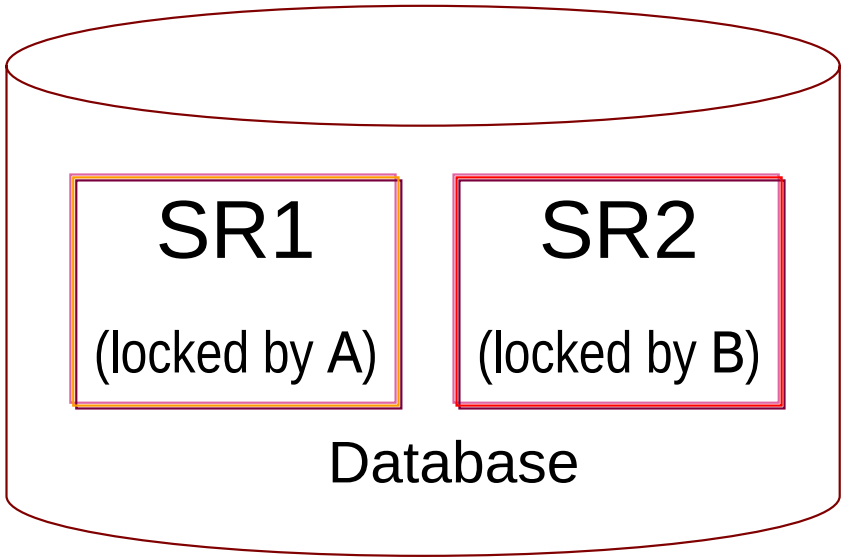
- **Intent locks** (intent-for-update locks)
  - Permit more concurrency control on large items (e.g. table)
  - Purpose
    - **To prevent other transactions from modifying the higher-level resource in a way that would invalidate the lock at the lower level.**
      - **Obtain intent lock on large item before shared/exclusive locks on smaller items**
    - **To improve the efficiency of the Database Engine in detecting lock conflicts at the higher level of granularity.**

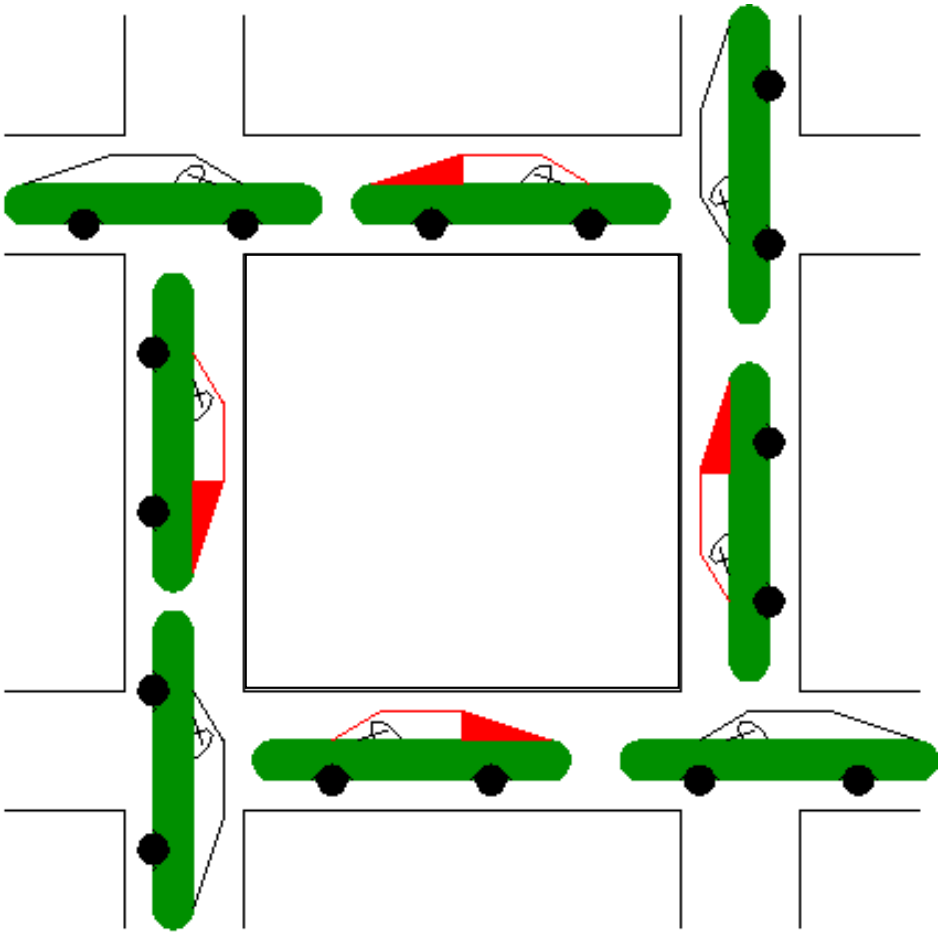
Deadlock (Mutual Waiting)

Transaction A	Time	Transaction B
XLock SR_1	T_1	
	T_2	XLock SR_2
XLock SR_2 (wait)	T_3	
	T_4	XLock SR_1 (wait)

Deadlock (Mutual Waiting)





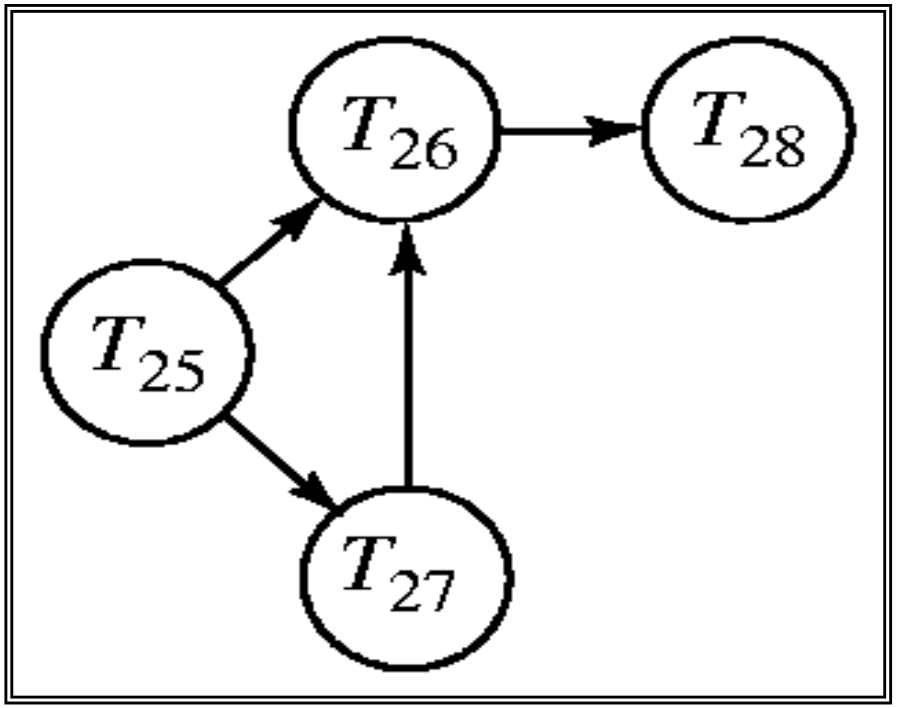


Both Ta,Tb want to access SR1,SR2

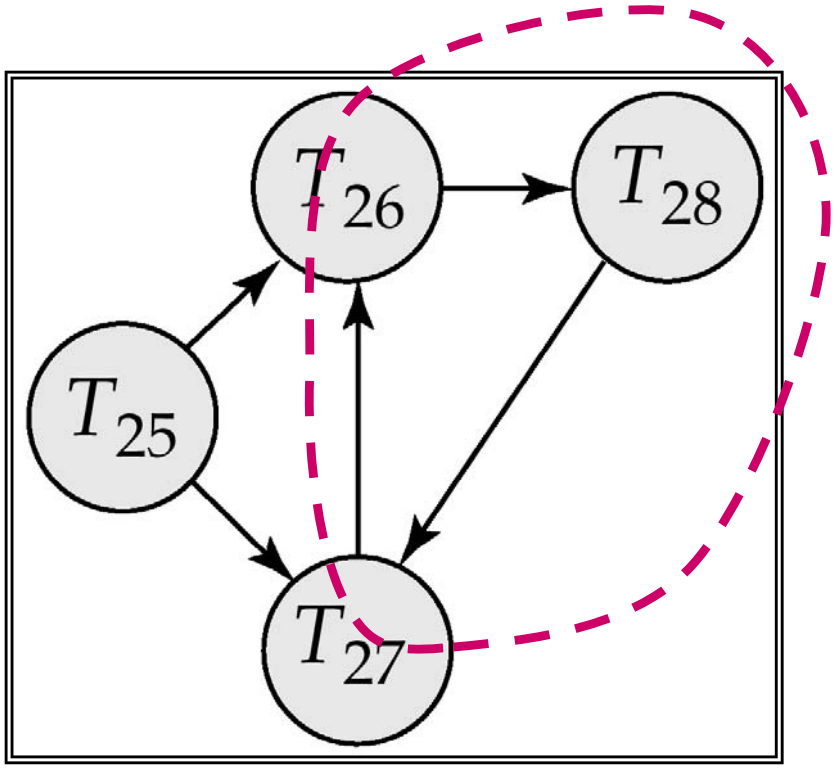
Deadlock Resolution

- **Detection**
  - Create a graph showing waiting dependencies
    - **Search the graph for cycles**
  - Empirically, most deadlocks involve 2 or 3 transactions
  - Used by enterprise DBMSs
- **Prevention:** Timeout
  - Waiting limit
  - Can abort transactions that are not deadlocked
  - Timeout interval is difficult to determine
  - Used by desktop DBMSs

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

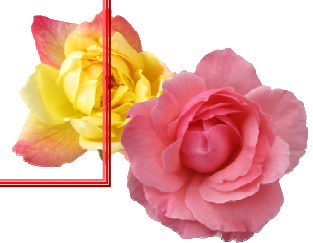
Two Phase Locking (2PL)

- Protocol to prevent ***lost update problems***

1. Locking phase
2. Unlocking phase

```
Lock A
....
Lock B
Lock C


....
Unlock B
Unlock A
Unlock C
```



Two Phase Locking (2PL)

- Protocol to prevent ***lost update problems***
- All transactions must follow conditions
  - **Obtain lock** before accessing item
  - **Wait** if a conflicting lock is held
  - Cannot obtain **new locks** after releasing locks (within a transaction)

Two distinct consecutive phases

during the transaction's execution:

- **Expanding phase** (Growing phase):
  - locks are acquired and no locks are released (the number of locks can only increase).
- **Shrinking phase**:
  - locks are released and no locks are acquired.

Guarantee *Serializability*

2PL

Lock..

…

Lock..

.

.

.

Unlock..

…

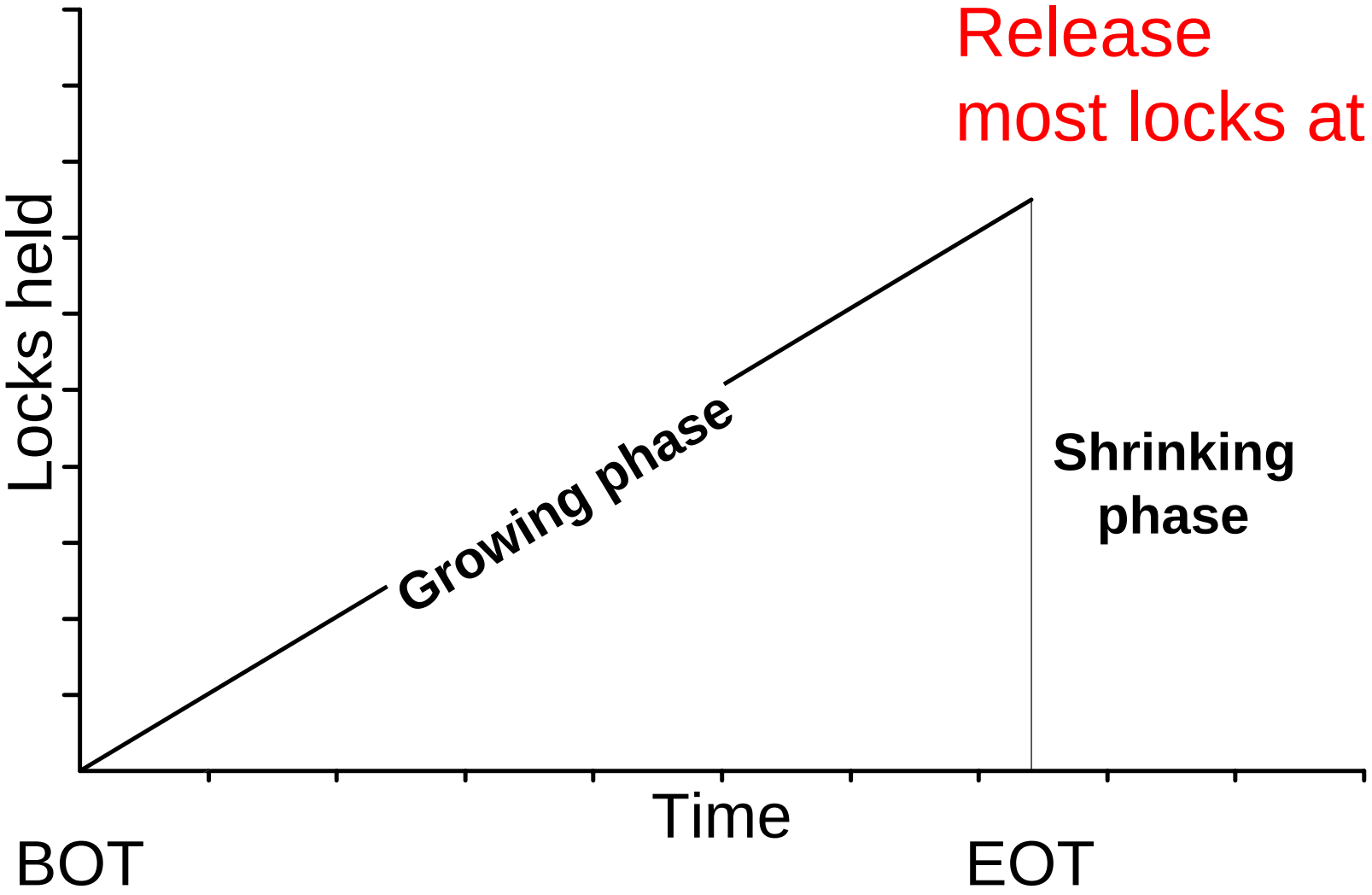
Unlock..

T1	T2
lock(A)	
read(A)	
	lock(A)
write(A)	
lock(B)	
unlock(A)	
	read(A)
	write(A)
	lock(B)
	unlock(A)
read(B)	
write(B)	
unlock(B)	
	read(B)
	write(B)
	unlock(B)

wait A

wait B

2PL Implementation

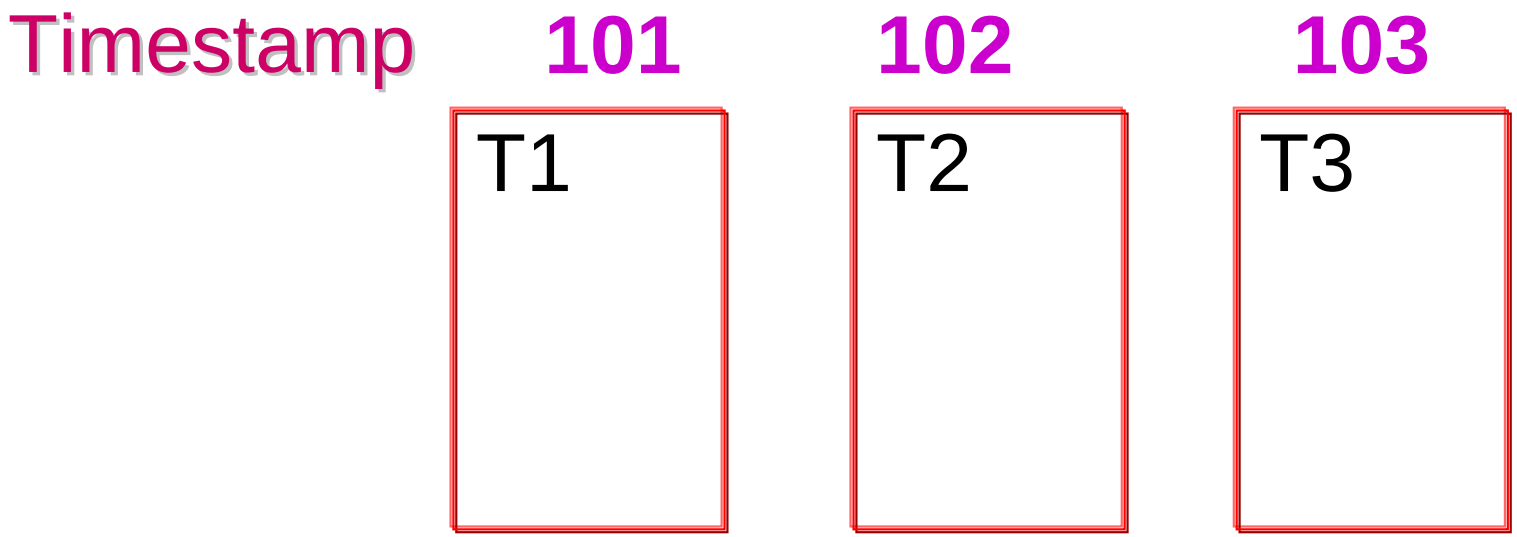


Optimistic Approaches

- **Assumes conflicts are rare**
- **No locks**
- **Check for conflicts** (version/time control)
  - **After each read and write**
  - **At the end of transaction**
- **Evaluation**
  - **Less overhead:** *no locks, deadlocks, …*
  - **More variability:** *penalty for conflict is rollback*
  - **Locking penalty:** *waiting (less penalty than rollback)*

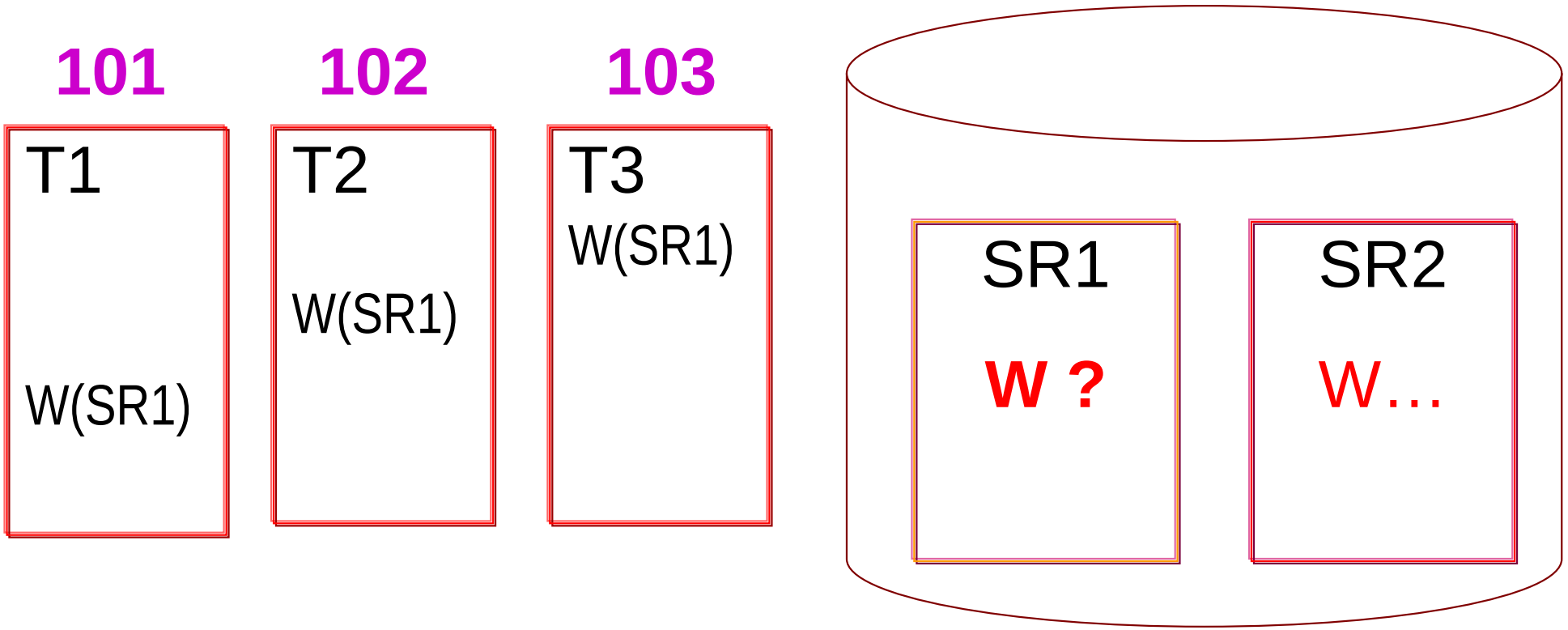
Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.**($T_{before} < T_{after}$)**
- time-stamps determine the serializability order.



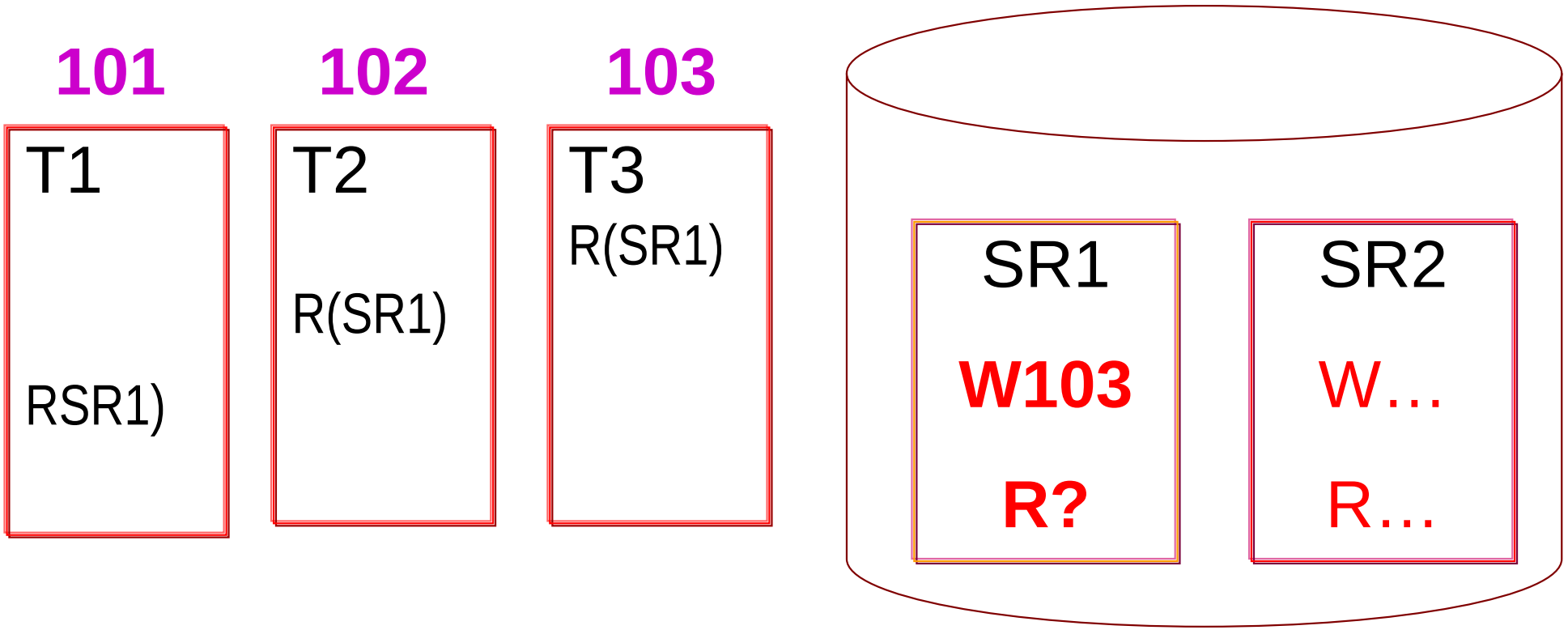
Timestamp-Based Protocols

- to assure serializability, the protocol maintains **for each data *Q* two timestamp values:**
  - **W-timestamp(*Q*)**
    - largest time-stamp of any T that executed write(*Q*) successfully.
  - **R-timestamp(*Q*)**
    - largest time-stamp of any T that executed read(*Q*) successfully.



W-timestamp(*Q*)

largest time-stamp of any *T* that executed write(*Q*) successfully.



R-timestamp(*Q*)

largest time-stamp of any ***T*** that executed read(*Q*) successfully.

Timestamp

read/write in the order of timestamps

- T1, read data(A)
  - If T2 written A, abort transaction T1-rollback
- T1, write data(A)
  - If T2 read A or
  - T2 written A
    - Abort transaction T1
  - Other cases, T1 can write A
- After read/write, update timestamp A

Timestamp rules

- **Maintain the serial order**
  - **T1 must write data before T2 read, write**
  - **T1 can not read data that written by T2.**

Write, Write
Read, Write

Example : *Timestamp approach*

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read(*X*)
read(*Y*)	read(*Y*)			
		write(*Y*)		
		write(*Z*)		
				read(*Z*)
	read(*X*)			
read(*X*)				
		write(*Z*)		
		abort		
				write(*Y*)
				write(*Z*)

Xw ,Xr
Yw ,Yr
Zw ,Zr

Locking strategies by table type.

Table Type	Examples	Suggested Locking Strategy
Live-High Volume	•Account	•**Optimistic** (first choice) •**Pessimistic** (second choice)
Live-Low Volume	•Customer •Insurance Policy	•**Pessimistic** (first choice) •**Optimistic** (second choice)
Log *(typically append only)*	•AccessLog •AccountHistory •TransactionRecord	•**Overly Optimistic**
Lookup / Reference *(typically read only)*	•State •PaymentType	•**Overly Optimistic**

Overly Optimistic is suitable for single user systems

Recovery Management

- Device characteristics and failure types
- Recovery tools
  - *transaction log, checkpointing*
- Recovery processes

Storage Device Basics

- Volatile: loses state after a shutdown *(RAM)*
- Nonvolatile: retains state after a shutdown *(Disk, tape, optical media)*
- Nonvolatile is more reliable than volatile but failures can cause loss of data
- Use multiple levels and redundant levels of nonvolatile storage for valuable data

Failure Types

- Local
  - Program detected and abnormal termination(/zero)
  - Affects only a single transaction
- Operating System
  - Affects all active transactions
  - Less common than local failures
- Device *(disk failure)*
  - Affects all active and past transactions
  - Least common

Transaction Log

- History of database changes
- Large storage overhead
- Operations
  - Undo: revert to previous state
  - Redo: reestablish a new state
- Fundamental tool of recovery management

Transaction Log Example

LSN	TransNo	Action	Time	Table	Row	Column	Old	New
1	101001	START	10:29					
2	101001	UPDATE	10:30	Acct	10001	AcctBal	100	200
3	101001	UPDATE	10:30	Acct	15147	AcctBal	500	400
4	101001	INSERT	10:32	Hist	25045	* *all columns*		<1002, 500, …>
5	101001	COMMIT	10:33					

Checkpoints

- Reduce restart work but adds overhead
  - Checkpoint log record
    (record working transaction at the checkpoint)
  - Write log buffers and database buffers (to disk)
- Checkpoint interval: time between checkpoints *(5 to 10 minutes in high volume environments)*
- All transaction processing stops during the checkpoint process

Other Recovery Tools

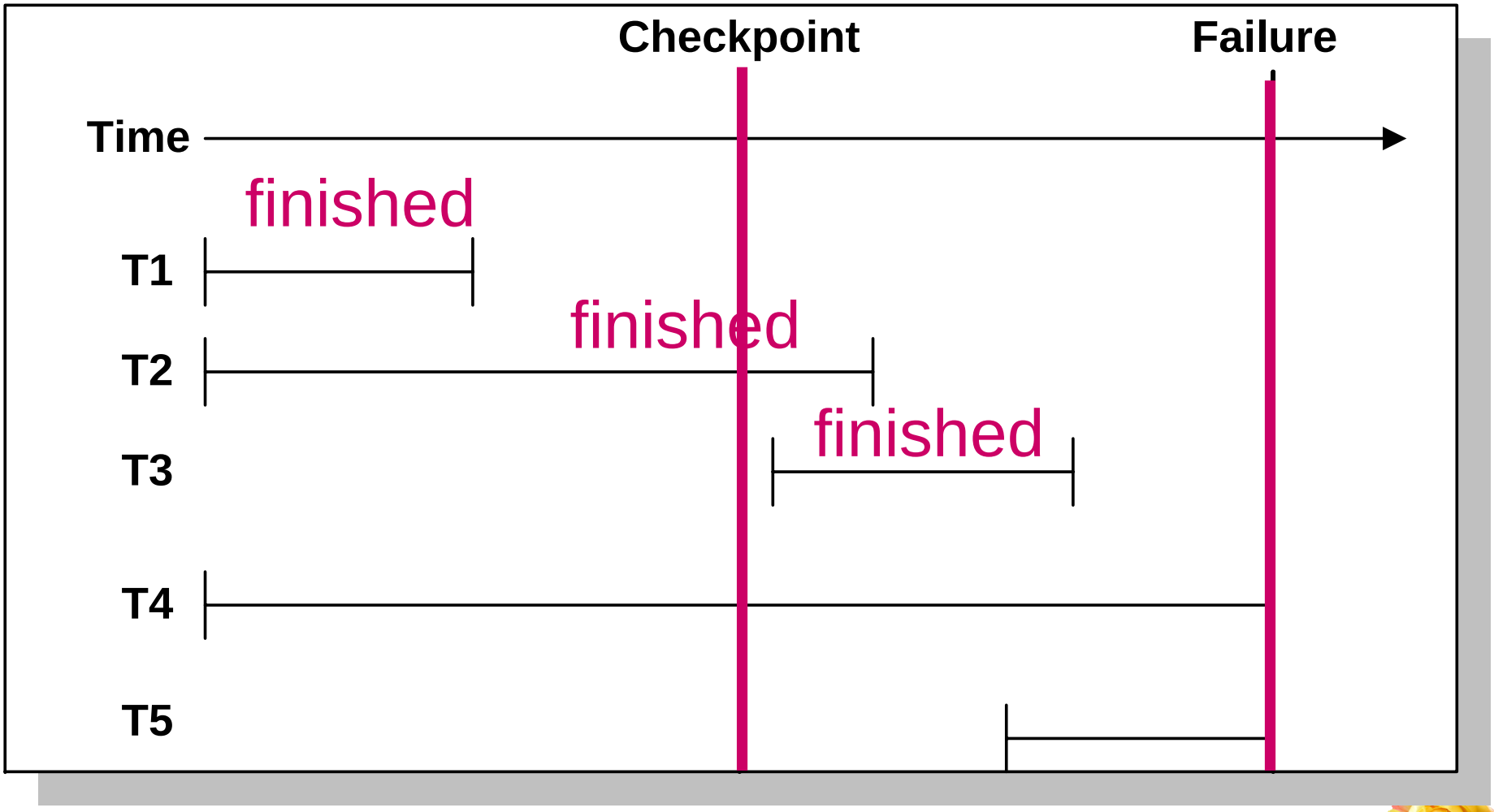
- Force writing to disk when
  - Checkpoint time
  - End of transaction
- Database backup
  - Complete, entire database
  - Incremental

Recovery from a Media Failure

- Restore database from the most recent backup
- **Redo** all committed transactions since the most recent backup
- **Restart** active transactions
  - Rollback and then restart

Assumes not log failure

Recovery Timeline



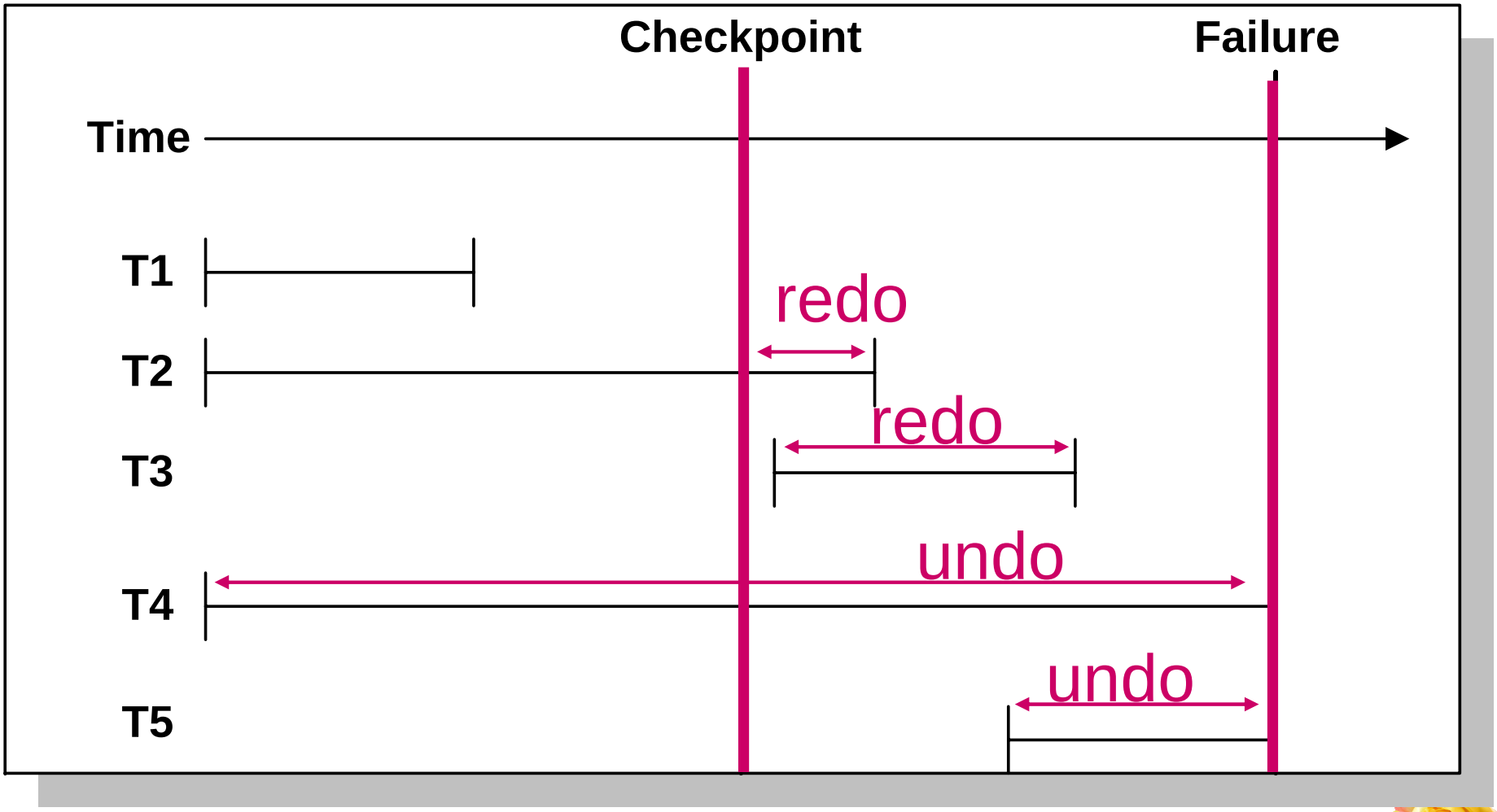
Recovery Processes

- Depend on timing of database writes
- Immediate update approach:
  - Before commit *(update database before committed)*
  - Log records written to disk first
  - Redo, undo
- Deferred update approach
  - After commit *(wait for commit before update database)*
  - Undo operations not needed

Recovery process

- **Redo process**
  - read log file and redo operations recorded in log file
- **Undo process**
  - send failure message to the computer that running the transaction
  - **rollback**

Immediate Update Recovery



Immediate Update Recovery

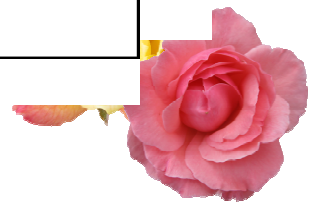
Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from checkpoint
T3	Started after CP; finished before failure	Redo forward from checkpoint
T4	Started before CP; not yet finished	Undo backwards from most recent log record
T5	Started after CP; not yet finished	Undo backwards from most recent log record

redo

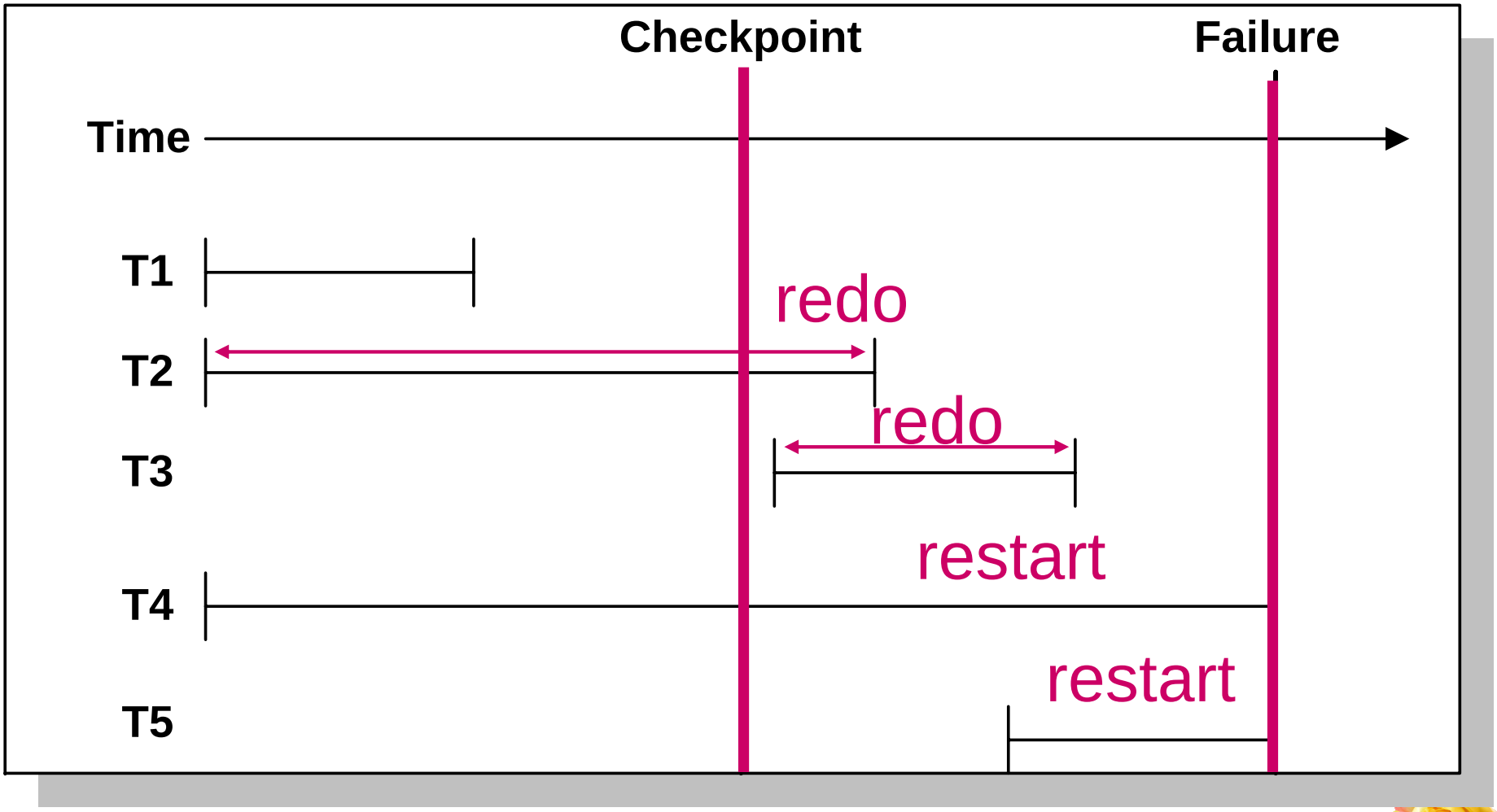
redo

undo

undo



Deferred Update Recovery



Restart T4 ,T5 without undo

Deferred Update Recovery

Class	Description	Restart Work
T1	Finished before CP	None
T2	Started before CP; finished before failure	Redo forward from first log record
T3	Started after CP; finished before failure	Redo forward from first log record
T4	Started before CP; not yet finished	None
T5	Started after CP; not yet finished	None

redo

redo

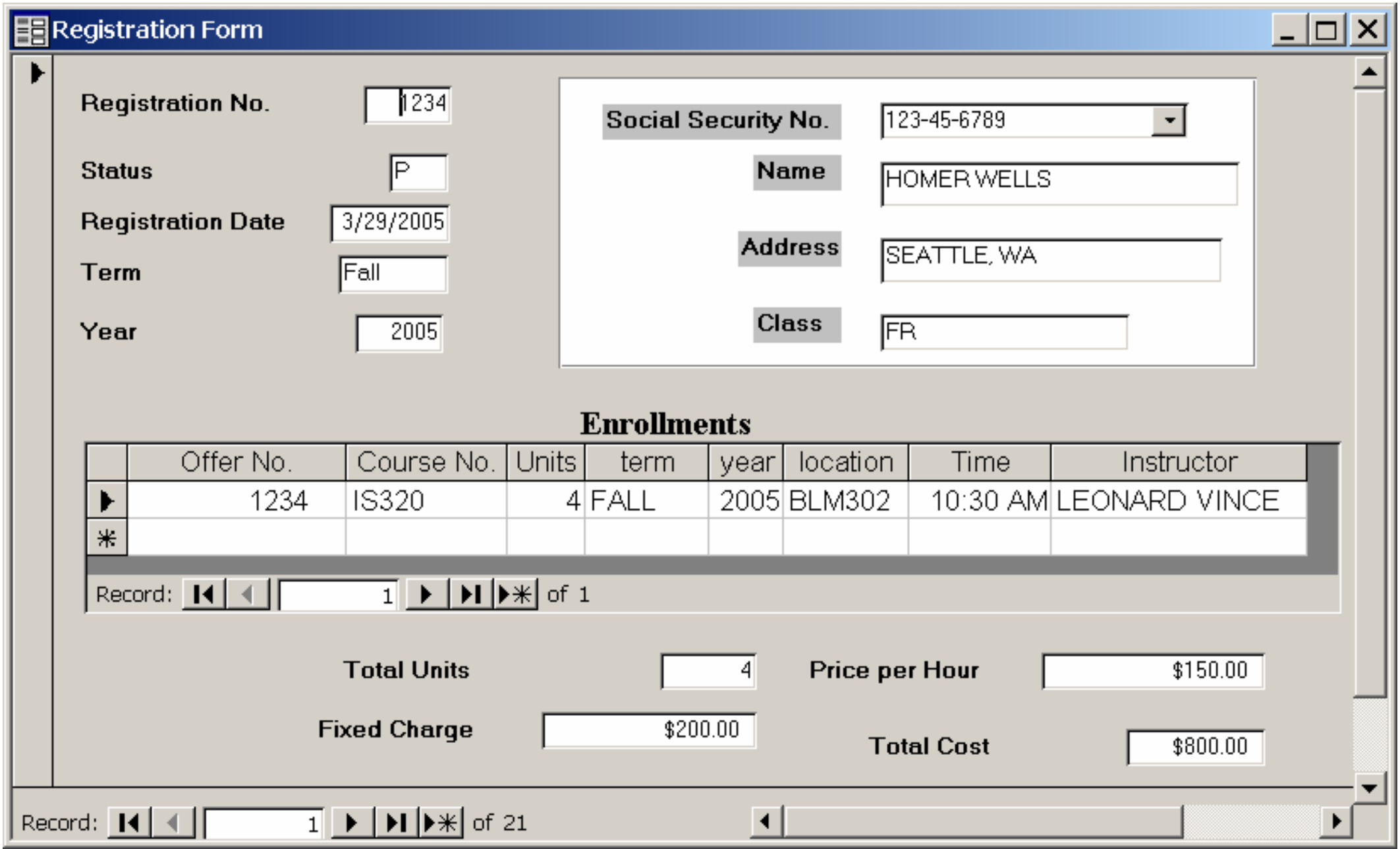
Transaction Design Issues

- Transaction boundary
- Isolation levels
- Deferred constraint checking
- Savepoints

Transaction Boundary Decisions

- Division of work into transactions
- Objective: minimize transaction duration
- Constraint: enforcement of important integrity constraints
- Transaction boundary decision can affect hot spots

Registration Form Example



Transaction Boundary Choices

- One transaction for the entire form
- One transaction for the main form and one transaction for all subform records
- One transaction for the main form and separate transactions for each subform record

Avoiding User Interaction Time

- Avoid to increase throughput
- Possible side effects: user confusion due to database changes
- Balance increase in throughput with occurrences of side effects
- Most situations increase in throughput more important than possible user confusion

Isolation Levels

- Degree to which a transaction is separated from the actions of other transactions
- Balance concurrency control overhead with interference problems
- Some transactions can tolerate uncommitted dependency and inconsistent retrieval problems
- Specify using the SET TRANSACTION statement

Uncommitted Dependency

- **or Dirty Read / Temporary Update**
- occurs when a transaction **is allowed to** retrieve or (worse) update a record that has been updated by another transaction but *has not yet been committed*

Uncommitted Dependency

Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7	read_item(Y);		
...	...	...	
	ROLLBACK		T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the temporary incorrect value of X

T2 read x updated by T1

T2 read x updated by T1

Losted update

Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7		COMMIT	T_2 loses update chance
8	read_item(Y);		
...	...	...	
	ROLLBACK		T_2 depends on uncommitted value and loses an update at time step 7

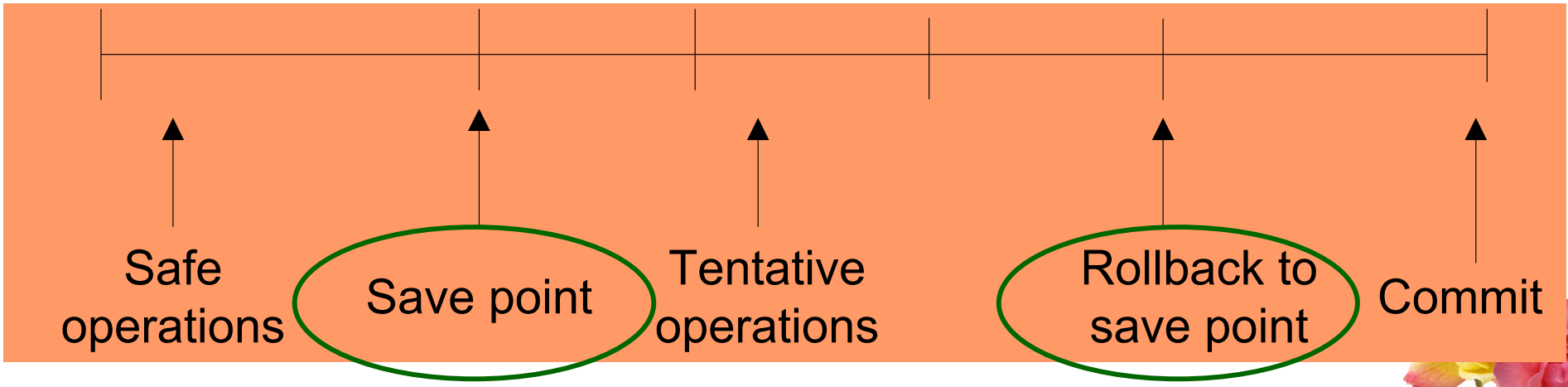
it also loses an update at time step 7, because the ROLLBACK in T1 causes data item X to be restored to its value before time step 1.

Integrity Constraint Timing

- Most constraints checked immediately
- Can defer constraint checking to EOT
- SQL
  - Constraint timing clause for constraints in a CREATE TABLE statement
  - SET CONSTRAINTS statement

Save Points

- Some transactions have tentative actions
- SAVEPOINT statement determines intermediate points
- ROLLBACK to specified save points



Summary

- Transaction: user-defined collection of work
- DBMSs support ACID properties
- Knowledge of concurrency control and recovery important for managing databases
- Transaction design issues are important
- Transaction processing is an important part of workflow management