

Information Systems

Chapter 6:

Transaction Processing & Recovery

Rita Schindler | TU Ilmenau, Germany

www.tu-ilmenau.de/dbis

Transaction Processing

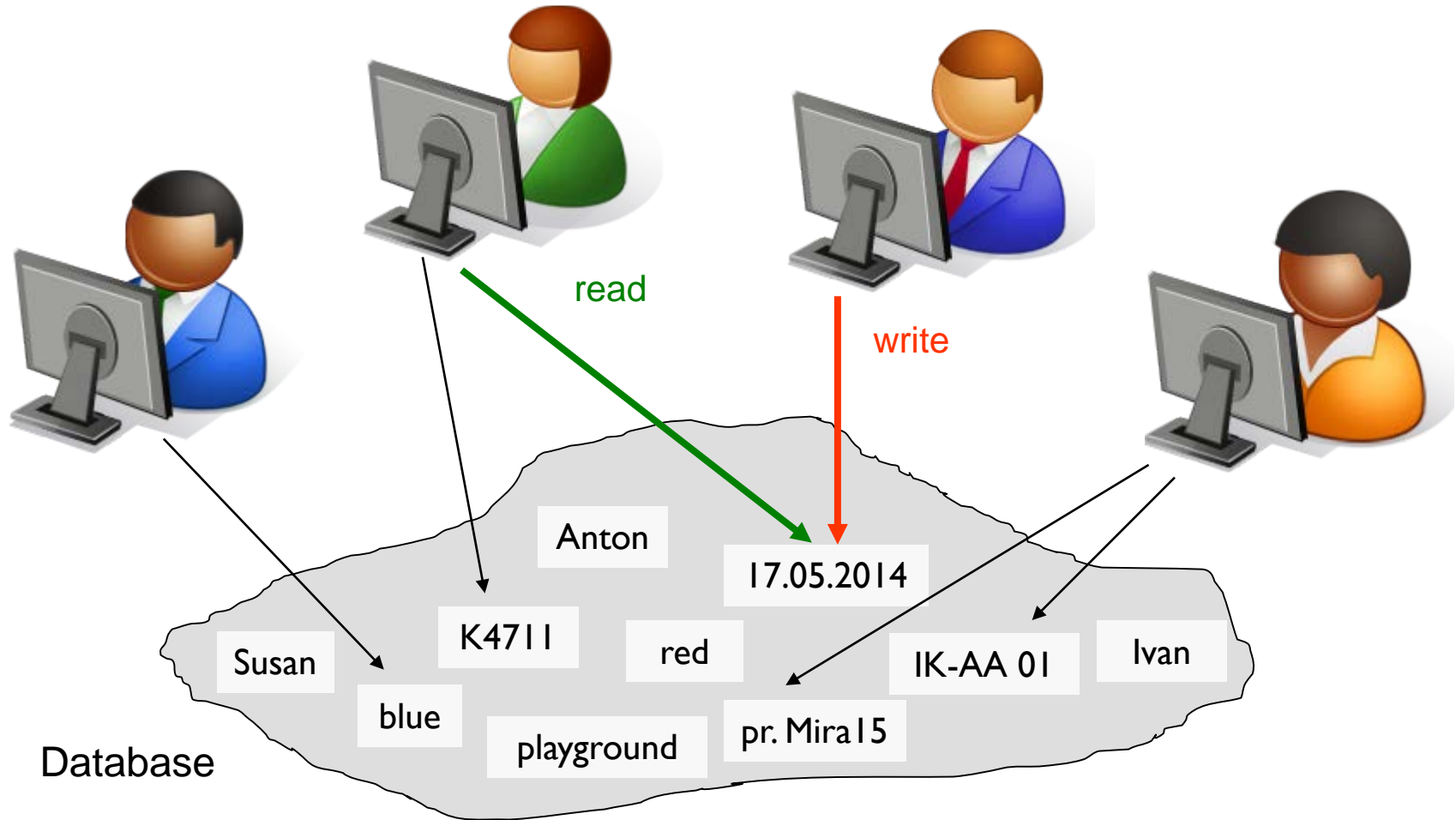
- ▶ Basic Concepts
- ▶ Concurrency Problems
- ▶ Serializability
- ▶ Concurrency Control in Oracle

Transaction Processing: Basic Concepts

A **transaction** is a unit of a program execution that accesses and possibly modifies various data objects (tuples, relations).

Goals: Understand the basic properties of a transaction and learn the concepts underlying transaction processing as well as the concurrent executions of transactions.

Transactions

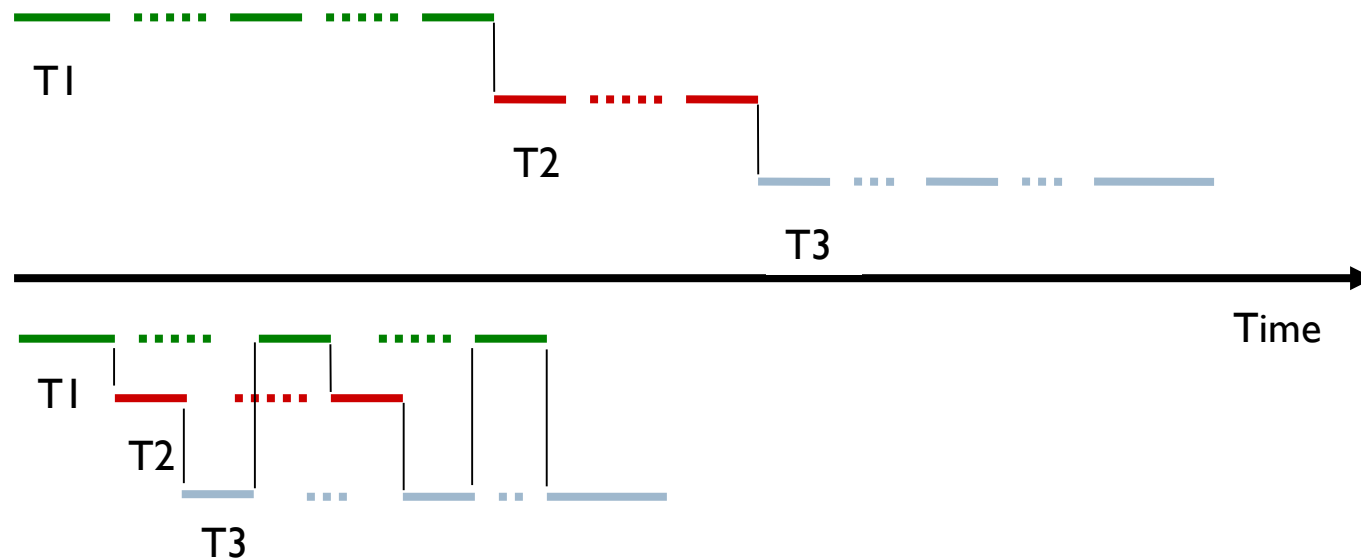


Transaction Processing (I)

- ▶ critical situations
 - ▶ many users want to access the data (read) AND make changes (write)
 - ▶ (e.g. flight booking systems, seat reservations, ...)
- ▶ not critical
 - ▶ many users only want to access the data (read)
 - ▶ (e.g. statistical databases)
- ▶ The steps of further processing between reading and writing are not important for transaction processing
- ▶ SQL: **SELECT ...** , **UPDATE ...** bzw. **INSERT ...**

Transaction Processing (2)

- ▶ many transactions at the same time
- ▶ sequential execution is too slow
- ▶ interleaving transactions to exploit waiting times for slow resources (e.g. secondary storage) and interactive actions (e.g. input)



Example: Airline Reservations

Flights (FNo , Fdate , SeatNo , Status)

```
SELECT SeatNo
FROM Flights
WHERE FNo = 123
      AND FDate = '2015-05-02'
      AND Status = 'available' ;
```

```
UPDATE Flights
SET Status = 'occupied'
WHERE FNo = 123
      AND FDate = '2015-05-02'
      AND SeatNo = '22A' ;
```

ACID

DBMS has to maintain the following properties of transactions:

- ▶ **A**tomicity: A transaction is an atomic unit of processing, and it either has to be performed in its entirety or not at all.
- ▶ **C**onsistency: A successful execution of a transaction must take a consistent database state to a (new) consistent database state. (→ integrity constraints)
- ▶ **I**solation: A transaction must not make its modifications visible to other transactions until it is committed, i.e., each transaction is unaware of other transactions executing concurrently in the system. (→ concurrency control)
- ▶ **D**urability: Once a transaction has committed its changes, these changes must never get lost due to subsequent (system) failures. (→ recovery)

Read-Write Model

Model used for representing database modifications of a transaction:

- ▶ **read**(A, x): assign value of database object A to variable x
- ▶ **write**(x, A): write value of variable x to database object A

Example of a Transaction T

read(A, x)

x := x - 200

write(x, A)

read(B, y)

y := y + 100

write(y, B)

Transaction Schedule reflects
chronological order of operations

Read-Write Model (2)

► other terms related to transactions

BOT – Begin of transaction

EOT – End of transaction
(SQL: COMMIT)

ABORT – Abort of transaction
and reset all actions

Isolation

- ▶ **Main focus** here: Maintaining isolation in the presence of multiple, concurrent user transactions
- ▶ **Goal:** “Synchronization” of transactions; allowing concurrency
 - ▶ (instead of insisting on a strict serial transaction execution, i.e., process complete T1, then T2, then T3 etc.)
 - ▶ increase the throughput of the system,
 - ▶ minimize response time for each transaction
- ▶ Problems that can occur for certain transaction schedules without appropriate concurrency control mechanisms:

Lost Update Problem

Time	Transaction T_1	Transaction T_2
1	read (A, x)	
2	x := x + 200	
3		read (A, y)
4		y := y + 100
5	write (x, A)	
6		write (y, A)
7		commit
8	commit	

- ▶ The update performed by T_1 gets lost;
- ▶ possible solution: T_1 locks / unlocks database object A
 $\Rightarrow T_2$ cannot read A while A is modified by T_1

Dirty Read Problem

Time	Transaction T_1	Transaction T_2
1	read (A, x)	
2	$x := x + 100$	
3	write (x, A)	
4		read (A, y)
5		write (y, B)
6	rollback	

- ▶ T_1 modifies a database object A, and then the transaction T_1 fails for some reason.
- ▶ Meanwhile the modified database object, however, has been accessed by another transaction T_2 .
- ▶ Thus T_2 has read data that “never existed”.

Nonrepeatable Read Problem

Time	Transaction T_1	Transaction T_2
1	read (A, y1)	
2		read (A, x1)
3		x1 := x1 - 100
4		write (x1, A)
5		read (C, x2)
6		x2 := x2 + 100
7		write (x2, C)
8		commit
9	read (B, y2)	
10	read (C, y3)	
11	sum := y1 + y2 + y3	
12	commit	

- ▶ In this schedule, the total computed by T_1 is wrong (off by 100).
⇒ T_1 must lock/unlock several database objects

Nonrepeatable Read Problem

T1	Time	T2	X	Y	Z	SUM
	0		100	200	300	600
SUM := 0	1					0
Read (X)	2					
Read (Y)	3					
SUM:=SUM+X	4					100
SUM:=SUM+Y	5					300
	6	Read (X)				
	7	X:=X-50	50			
	8	Write (X)	50			
	9	Read (Z)				
	10	Z:=Z+50			350	
	11	Write (Z)			350	
	12	COMMIT				
Read (Z)	13					
SUM:=SUM+Z	14					650
COMMIT	15					



Bank Manager



Serializability

- ▶ DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc.

A (possibly concurrent) schedule S is **serializable** if it is equivalent to a serial schedule S' , i.e., S has the same result database state as S' .

- ▶ How to ensure serializability of concurrent transactions?

Conflicts

Conflicts between operations of two transactions:

T_i	T_j
read (A, x)	read (A, y)

(order does not matter)

T_i	T_j
read (A, x)	write (y, A)

(order matters)

T_i	T_j
write (x, A)	read (A, y)

(order matters)

T_i	T_j
write (x, A)	write (y, A)

(order matters)

A schedule S is **serializable** with regard to the above conflicts **iff** S can be transformed into a serial schedule S' by a series of swaps of non-conflicting operations.

Checks for Serializability

- ▶ Checks for serializability are based on **precedence graph** that describes dependencies among concurrent transactions.
- ▶ If the graph has no cycle, then the transactions are serializable. → They can be executed concurrently without affecting each others transaction result.

Schedules

- ▶ **Log** Graph $G_T = (A, P)$

- ▶ A – nodes are actions read (r) and write (w)
- ▶ P – edges, $P \subset A \times A$,
- ▶ actions w and r on the same object are ordered

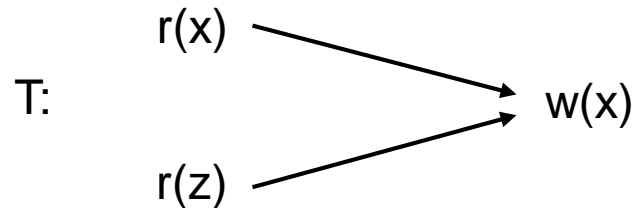
- ▶ **Schedule**

absolute order of execution of all actions of all active transactions during this time

- ▶ Correct ? \rightarrow serializability
- ▶ Control ? \rightarrow serialization graph

Example (I)

▶ Transaction log



▶ $G = (A, P)$

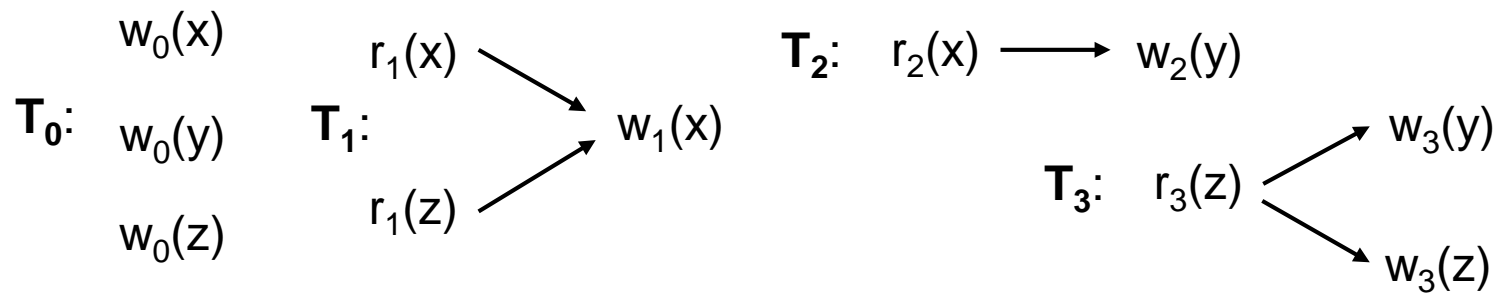
- ▶ $A = \{ r(x), r(z), w(x) \}$
- ▶ $P = \{ (r(x), w(x)), (r(z), w(x)) \}$

▶ (possible) Schedules

- ▶ $r(x), r(z), w(x)$
- ▶ $r(z), r(x), w(x)$

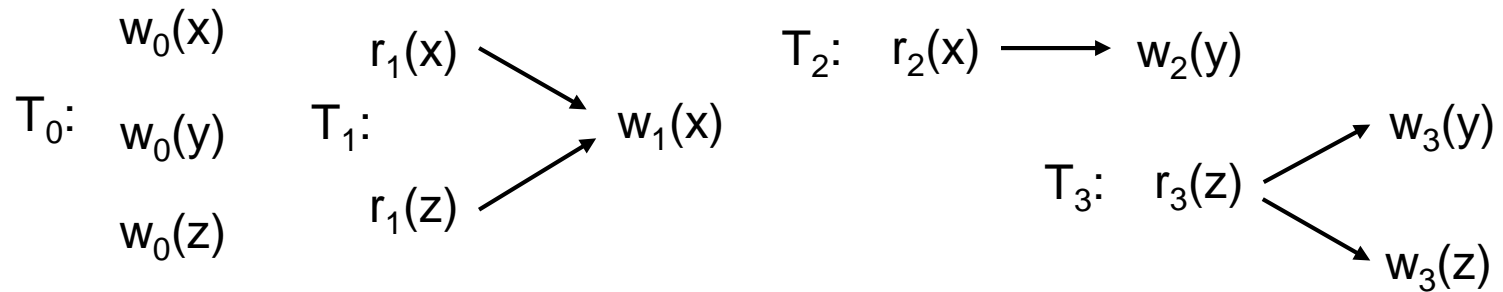
Example (2)

Transactions

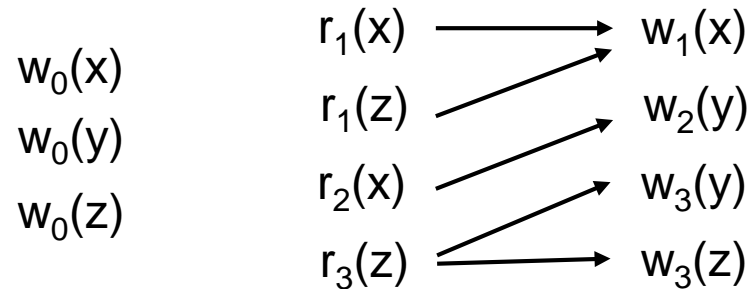


Example (2)

Transactions

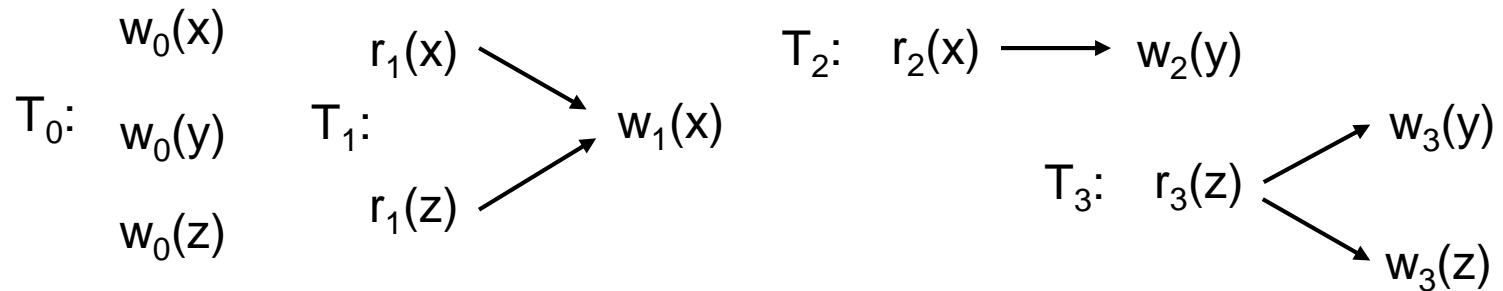


Log L

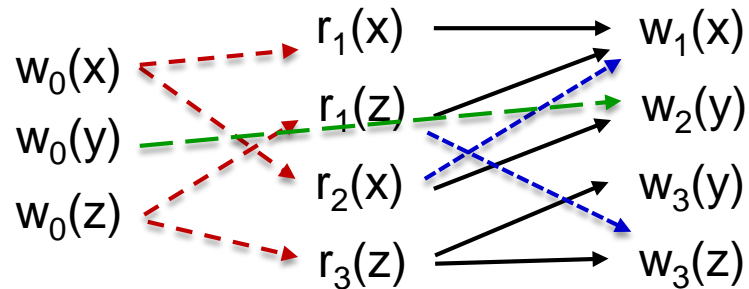


Example (2)




Transactions



Log L

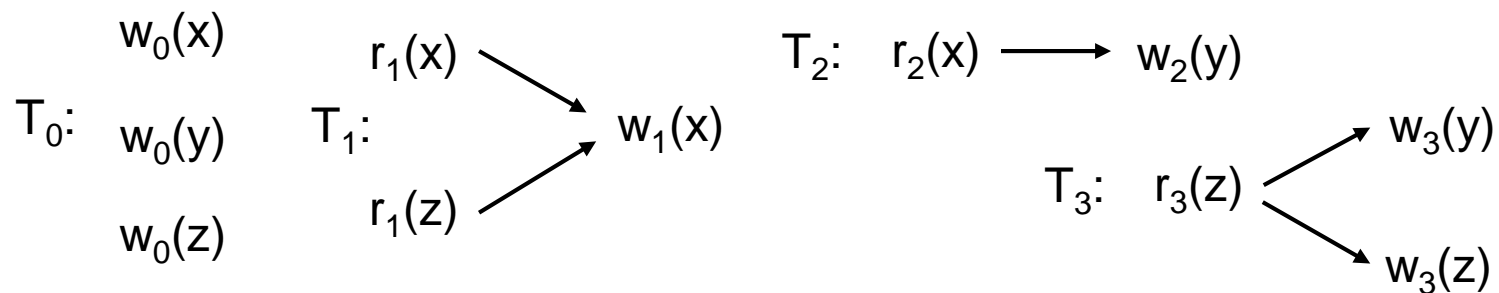


Possible conflicts:

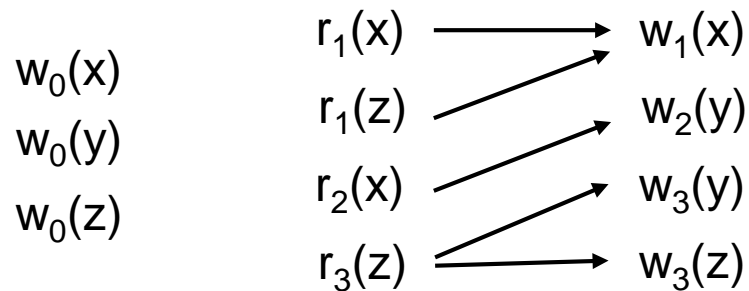
- ▶ write – read 
- ▶ read – write 
- ▶ write – write 

Example (2)

Transactions



Log L



Schedules

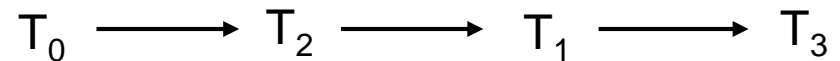
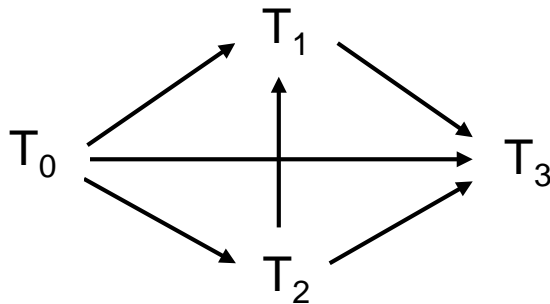
- $S1 = \{ w_0(x), w_0(y), w_0(z), r_1(x), r_1(z), r_2(x), r_3(z), w_1(x), w_2(y), w_3(y), w_3(z) \}$
- $S2 = \{ w_0(x), w_0(y), w_0(z), r_2(x), w_2(y), r_1(x), r_1(z), w_1(x), r_3(z), w_3(y), w_3(z) \}$

Example (2)

Schedules

- ▶ $L1 = \{ w_0(x), w_0(y), w_0(z), r_1(x), r_1(z), r_2(x), r_3(z), w_1(x), w_2(y), w_3(y), w_3(z) \}$
- ▶ $L2 = \{ w_0(x), w_0(y), w_0(z), r_2(x), w_2(y), r_1(x), r_1(z), w_1(x), r_3(z), w_3(y), w_3(z) \}$

Serialization Graphs



Example / Task

Given are the following schedules:

- ▶ $S1 = \{ r_1(x), w_1(x), r_2(x), r_3(z), w_3(x), r_1(z) \}$
- ▶ $S2 = \{ r_1(x), w_1(x), r_2(y), w_2(y), r_1(y), r_2(x) \}$

Are these schedules serializable?

Concurrency Control: Lock-Based Protocols

- ▶ One way to ensure serializability is to require that accesses to data objects must be done in a mutually exclusive manner.
- ▶ Allow transaction to access data object only if it is currently holding a **lock** on that object.
- ▶ Serializability can be guaranteed using locks in a certain fashion \Rightarrow Tests for serializability are redundant !

Types of locks that can be used in a transaction T:

- ▶ **slock(X)**
- ▶ **xlock(X)**
- ▶ **unlock(X)**

Types of Locks

Locks that can be used in a transaction T:

- ▶ **slock(X):**
 - ▶ **s**hared-lock (read-lock);
 - ▶ no other transaction than T can write data object X, but they can read X
- ▶ **xlock(X):**
 - ▶ **e**xclusive-lock;
 - ▶ T can read/write data object X;
 - ▶ no other transaction can read/write X
- ▶ **unlock(X):**
 - ▶ unlock data object X

Lock-Compatibility Matrix

requested	existing lock	
	slock	xlock
slock	OK	No
xlock	No	No

E.g., **xlock**(A) has to wait until all **slock**(A) have been released.

Using locks in a transaction (lock requirements, LR):

- ▶ before each **read**(X) there is either a **xlock**(X) or a **slock**(X) and no **unlock**(X) in between
- ▶ before each **write**(X) there is a **xlock**(X) and no **unlock**(X) in between
- ▶ a **slock**(X) can be tightened using a **xlock**(X)
- ▶ after a **xlock**(X) or a **slock**(X) sometime an **unlock**(X) must occur

Lock-Compatibility Matrix (2)

But: “Simply setting locks/unlocks is not sufficient!”

That means, replace each

read(X)

→ **slock(X); read(X); unlock(X);**

write(X)

→ **xlock(X); write(X); unlock(X);**

Two-Phase Locking Protocol (TPLP)

A transaction T satisfies the TPLP iff

- after the first **unlock(X)** no locks **xlock(X)** or **slock(X)** occur
- That is, first T obtains locks, but may not release any lock (growing phase)
and then T may release locks, but may not obtain new locks (shrinking phase)

Strict Two-Phase Locking Protocol:

- ▶ All unlocks at the end of the transaction $T \Rightarrow$ no dirty reads are possible, i.e., no other transaction can read the (modified) data objects in case of a rollback of T .

Example

T1	Time	T2
	0	
SUM := 0	1	
SLOCK(Y)	2	
Read (Y)	3	
SUM:=SUM+Y	4	
	5	XLOCK(X)
	6	XLOCK(Z)
	7	Read (X)
	8	Read (Z)
	9	X:=X-50
	10	Z:=Z+50
	11	Write (Z)
	12	UNLOCK(Z)

T1	Time	T2
SLOCK(Z)	13	
Read (Z)	14	
SUM:=SUM+Z	15	
	16	Write (X)
	17	UNLOCK(X)
SLOCK(X)	18	
Read (X)	19	
SUM:=SUM+X	20	
UNLOCK(X)	21	
UNLOCK(Y)	22	
UNLOCK(Z)	23	
	24	COMMIT
COMMIT	25	

Two-Phase Locking Protocol (TPLP)

- ▶ The interleaved execution of transactions that satisfies the TPLP is correct. A TPLP-Scheduler generates a serializable schedule.
- ▶ Deadlocks
 - ▶ Timestamps → abort
 - ▶ Graph (of delay): cycle → abort

T1	T2
...	...
slock (x)	...
...	slock (y)
xlock (y)	...
	xlock (x)

Concurrency Control in Oracle

In Oracle the user can specify the following locks on relations and tuples using the command

lock table in \langle mode \rangle mode ;

mode	\triangle	tuple level	relation level
row share	\triangle	slock	intended slock
row exclusive	\triangle	xlock	intended xlock
share	\triangle	—	slock
share row exclusive	\triangle	—	sixlock
exclusive	\triangle	—	xlock

Concurrency Control in Oracle (2)

The following locks are performed automatically by the scheduler:

select	→	no lock
insert/update/delete	→	xlock /row exclusive
select ... for update	→	slock /row share
commit	→	releases all locks

Isolation Levels in Oracle

Oracle furthermore provides **isolation levels** that can be specified before a transaction by using the command

set transaction isolation level <level> ;

- ▶ **read committed** (default): each query executed by a transaction sees the data that was committed before the query (not before the whole transaction!)
(→ statement level read consistency)

Isolation Level: read committed

T_1	T_2
select A from R ; → old value	update R set A = new ; commit ;
select A from R ; → old value	
select A from R ; → new value	

- ▶ non-repeatable reads are possible (same select statement in a transaction gives different results at different times);
- ▶ dirty-reads are not possible

Isolation Levels in Oracle (2)

- ▶ **serializable**: serializable transactions see only those changes that were committed at the time the transaction began, plus own changes.
- ▶ Oracle generates an error when such a transaction tries to update or delete data modified by a transaction that commits after the serializable transaction began.

Isolation Level: serializable

T_1	T_2
set transaction isolation level serializable ; update R set A = new where B = 1 ; → error	set transaction ... update R set A = new where B = 1 ; commit ;

- ▶ Dirty-reads and non-repeatable reads are not possible.
- ▶ Furthermore, this mode guarantees serializability (but does not provide much parallelism).

Summary

- ▶ Notion of transaction
- ▶ ACID
- ▶ Isolation is achieved by concurrency control
- ▶ Serializability and conflict serializability
- ▶ 2 phase lock protocols
- ▶ Relaxing Isolation through isolation levels

Example / Task

A delivery store is based on a database with the following schema:

Store (PartNo , Place , Stock)

Delivery (DeliveryNo , PartNo , Amount , Price)

Reorder (PartNo , Amount , Producer)

The store delivers parts and reorders them from the producer.

The following updates will be executed on the relations:

```
INSERT INTO Delivery VALUES ( 739 , 4713 , 5 , 225.00 ) ;
```

```
UPDATE Store SET Stock = Stock – 5 WHERE PartNo = 4713 ;
```

```
INSERT INTO Reorder VALUES ( 4713 , 5 , 'Siemens' ) ;
```

Where must be placed a COMMIT to indicate an end of a transaction?

Which problems of consistency can occur?

Serial and Serializable Schedules (I)

T1	T2
Read (A, t)	Read (A, s)
$t := t + 100$	$S := s * 2$
Write (t, A)	Write (s, A)
Read (B, t)	Read (B, s)
$t := t + 100$	$s := s * 2$
Write (t, B)	Write (s, B)

Serial and Serializable Schedules (2)

Time	T1	T2	A	B
0			25	25
1	Read (A, t)			
2	$t := t + 100$			
3	Write (t, A)		125	
4	Read (B, t)			
5	$t := t + 100$			
6	Write (t, B)			125
7		Read (A, s)		
8		$S := s * 2$		
9		Write (s, A)	250	
10		Read (B, s)		
11		$s := s * 2$		
12		Write (s, B)		250

Serial and Serializable Schedules (3)

Time	T1	T2	A	B
0			25	25
1		Read (A, s)		
2		$S := s * 2$		
3		Write (s, A)	50	
4		Read (B, s)		
5		$s := s * 2$		
6		Write (s , B)		50
7	Read (A, t)			
8	$t := t + 100$			
9	Write (t, A)		150	
10	Read (B, t)			
11	$t := t + 100$			
12	Write (t , B)			150

Serial and Serializable Schedules (4)

Time	T1	T2	A	B
0			25	25
1	Read (A, t)			
2	$t := t + 100$			
3	Write (t, A)		125	
4		Read (A, s)		
5		$S := s * 2$		
6		Write (s, A)	250	
7	Read (B, t)			
8	$t := t + 100$			
9	Write (t, B)			125
10		Read (B, s)		
11		$s := s * 2$		
12		Write (s, B)		250

Serial and Serializable Schedules (5)

Time	T1	T2	A	B
0			25	25
1	Read (A, t)			
2	$t := t + 100$			
3	Write (t, A)		125	
4		Read (A, s)		
5		$S := s * 2$		
6		Write (s, A)	250	
7		Read (B, s)		
8		$s := s * 2$		
9		Write (s, B)		50
10	Read (B, t)			
11	$t := t + 100$			
12	Write (t, B)			150

Serial and Serializable Schedules (6)

Time	T1	T2	A	B
0			25	25
1	Read (A, t)			
2	$t := t + 100$			
3	Write (t, A)		125	
4		Read (A, s)		
5		$S := s + 200$		
6		Write (s, A)	325	
7		Read (B, s)		
8		$s := s + 200$		
9		Write (s , B)		225
10	Read (B, t)			
11	$t := t + 100$			
12	Write (t , B)			325

Task

A transaction T1, executed by an airline-reservation system, performs the following steps:

- ▶ The customer is queried for a desired flight time and cities. Information about the desired flights is located in database elements A and B, which the system retrieves from disk.
- ▶ The customer is told about the options, and selects a flight whose data, including the number of reservations for that flight is in B. A reservation on that flight is made for the customer.
- ▶ The customer selects a seat for the flight; seat data for the flight is in database element C.
- ▶ The system gets the customer's credit-card number and appends the bill for the flight to a list of bills in database element D.
- ▶ The customer's phone and flight data added to another list on database element E for fax to be sent confirming the flight.

Express transaction T1 as a sequence of r and w actions.