Technische Hochschule Rosenheim

Chapter 8 – Transaction management

Databases lectures

Prof. Dr Kai Höfig

Chapter 8: Transactions

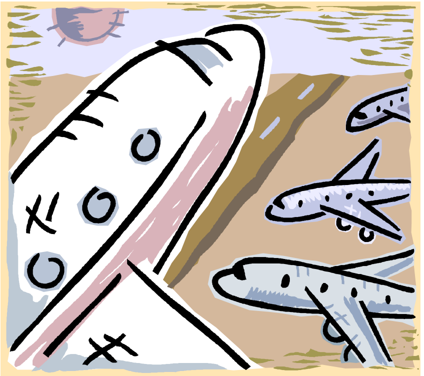
In this chapter, we will address the following questions

- Transactions:
  - What happens if multiple users want to access a database at the same time?
  - What is a transaction and what do I need it for?
  - What are the ACID criteria?
  - What are isolation levels, which levels are there, what is serialisability?
  - How do I achieve serialisability? Do I always want that?
  - What are locking and locking protocols such as 2PL?
  - How do I implement all this in SQL?

Literature: CompleteBook Chap 6.6, Chap 7; Beaver book Chap 12

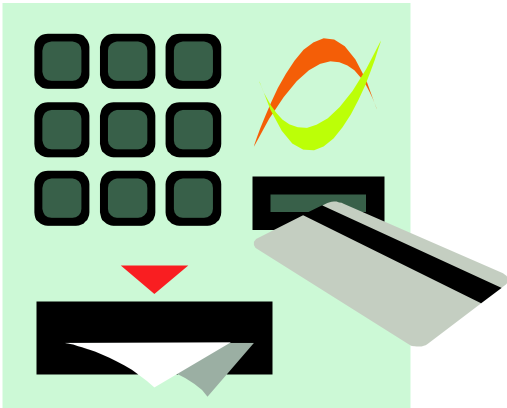
Chapter 9: Transactions, integrity and triggers

Example scenarios for transactions



- Seat reservation for flights simultaneously from many travel agencies
  → seat could be sold multiple times if multiple travel agencies identify the seat as available

- Overlapping account operations at a bank
  → accounts could contain incorrect balances if multiple transfers overlap



- Statistical database operations
  → results are corrupted if data is modified during the calculation

Image source: http://office.microsoft.com/de-de/images/

The term "transaction"

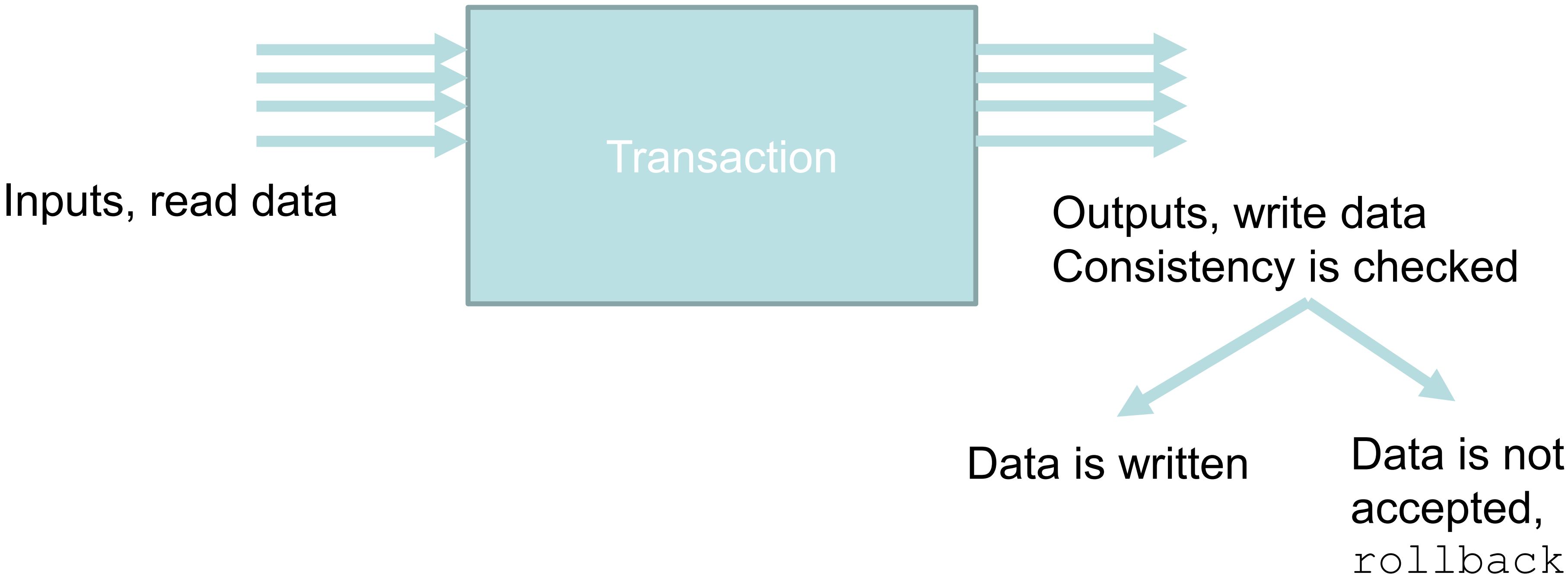
- Definition of transaction
  A transaction is a sequence of operations (actions) that transfer the database from a consistent state into a (possibly modified) consistent state, whereby the ACID principle must be followed.

- Aspects:
  - Semantic integrity: correct (consistent) DB state after the end of the transaction
  - Processing integrity: avoid errors caused by "simultaneous" access of multiple users to the same data

Image source: http://office.microsoft.com/de-de/images/

Consistency of a transaction

- Operations of a transaction can either read data (e.g. by `SELECT`) or write data (e.g. by `INSERT`, `UPDATE`, `DELETE`).

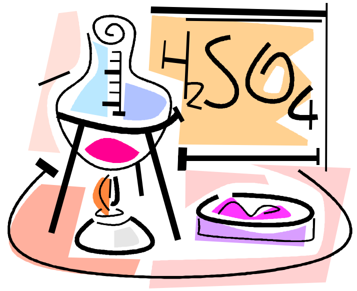


Two laws of concurrency

- Concurrent or simultaneous execution of tasks should not cause programmes to run incorrectly. (Isolation in ACID)
  - If all data is contained in a central source, if it is available from a central computing unit, and if the applications only require a very short time, then the problem of concurrency is easily solved by sequential execution.

- The concurrent execution of tasks should not be significantly slower than sequential execution.

ACID properties



Atomicity:

Transaction is either executed entirely or not at all

Consistency (also integrity preservation):

Database is in a consistent state both before the start of and after a transaction is completed

Isolation:

Users working with a database should have the impression that they are the only ones working with this database

Durability:

After a transaction has been successfully completed, the result of that transaction must be stored "permanently" in the database

Image source: http://office.microsoft.com/de-de/images/

Commands for transaction control

- **BOT** (Begin-of-Transaction)
  - beginning of a transaction: implicit in SQL!

- `commit`:
  - the transaction should be completed successfully (and the next one should be started implicitly)

- `abort`:
  - the transaction should be aborted (and the next one should be started implicitly)

Image source: http://office.microsoft.com/de-de/images/

Integrity violation in transactions

- Example of a transaction T:
  - Table `ACCOUNTS(AccountNo, Balance)`
  - Transfer of an amount of `€200` from account `K1` to another account `K2`
  - Condition: the total of the balances of all accounts remains constant

- Implementation of transaction T in SQL
  - as a sequence of (two) elementary changes:

```
update ACCOUNTS set Balance = Balance - 200 where AccountNo
= K1
update ACCOUNTS set Balance = Balance + 200 where AccountNo
= K2
```

➔ Condition is not necessarily fulfilled between the individual change steps!

Simplified model for transactions

- Representation of database changes in a transaction
  - `read (A, x)`: assigns the value of DB object A to the variable x
  - `write(x, A)`: stores the value of the variable x in DB object A

- Example of our transaction T:

```
read(BalanceAccountK1,x); x := x - 200;
write(x,BalanceAccountK1);
read(BalanceAccountK2,y); y := y + 200;
write(y,BalanceAccountK2);
commit
```

- Example of another transaction S that transfers `€100` from `K1` to `K3`:

```
read(BalanceAccountK1,u); u := u - 100;
write(u,BalanceAccountK1);
read(BalanceAccountK3,v); v := v + 100;
write(v,BalanceAccountK3);
commit
```

Execution variants for two transactions S, T

- Serial execution of S before T:

S	T
read(BalanceAccountK1,u);	
u := u - 100;	
write(u,BalanceAccountK1);	
read(BalanceAccountK3,v);	
v := v + 100;	
write(v,BalanceAccountK3);	
commit	
	read(BalanceAccountK1,x);
	x := x - 200;
	write(x,BalanceAccountK1);
	read(BalanceAccountK2,y);
	y := y + 200;
	write(y,BalanceAccountK2);

- "Mixed" execution such as alternating steps of S and T:

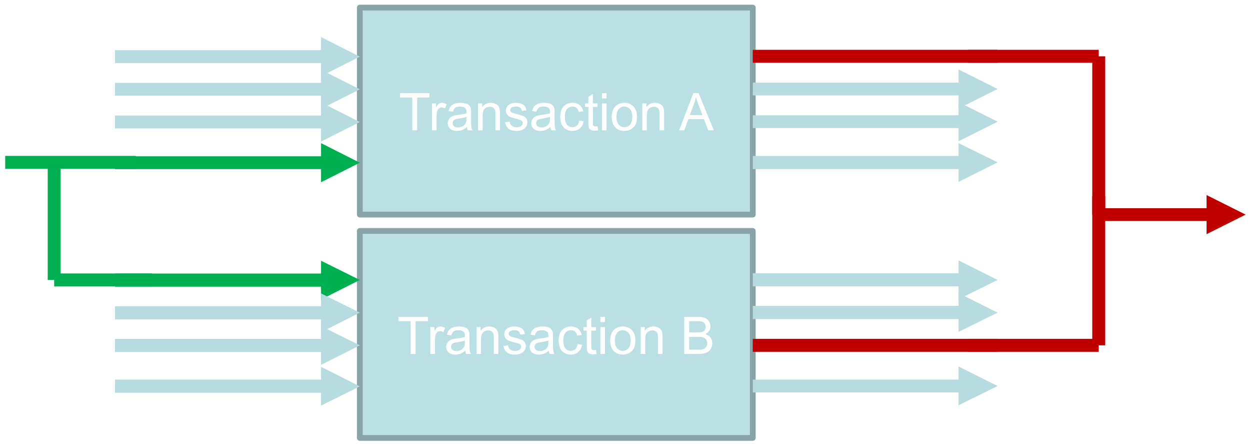
S	T
read(BalanceAccountK1,u);	
	read(BalanceAccountK1,x);
u := u - 100;	
	x := x - 200;
write(u,BalanceAccountK1);	
	write(x,BalanceAccountK1);
read(BalanceAccountK3,v);	
	read(BalanceAccountK2,y);
v := v + 100;	
	y := y + 200;
write(v,BalanceAccountK3);	
	write(y,BalanceAccountK2);
Commit	

Chapter 9: Transactions, integrity and triggers

Concurrency

- If two transactions have simultaneous **read-only** access to an object, then the consistency cannot be violated because the state of the object does not change.

- If two transactions have **write** access to the same object, this can lead to a violation of the *isolation* principle.

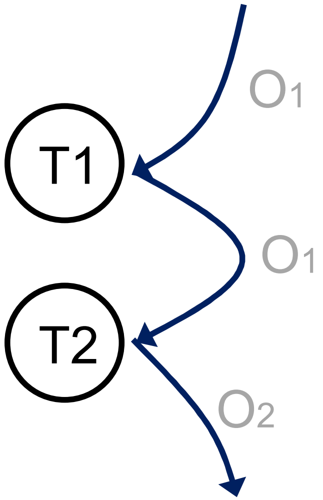


Static vs. dynamic allocation

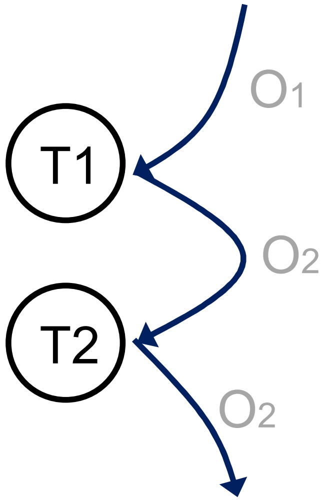
- **Static allocation** means that a transaction explicitly specifies **at design time** which objects it has read and write access to.
  - e.g. an account booking has write access to any account, so the entire account table must be allocated → very pessimistic
- A transaction manager can then easily check if concurrent execution may cause conflicts with another transaction, and then execute the transactions sequentially.
- With **dynamic allocation**, objects are allocated **at runtime**.
  - e.g. an account booking has access to an account 12345 and only this account is allocated → high concurrency, but difficult to calculate.

Dependency graph

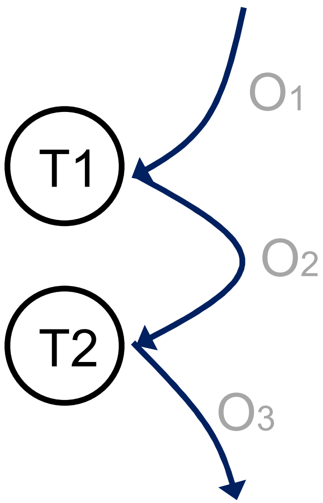
…
T1 **read** O_1
…
T2 **write** O_2
…



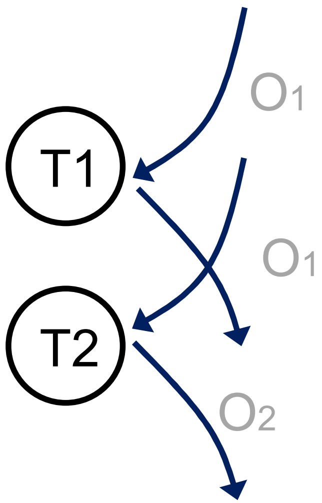
…
T1 **write** O_1
…
T2 **read** O_2
…



…
T1 **write** O_1
…
T2 **write** O_2
…



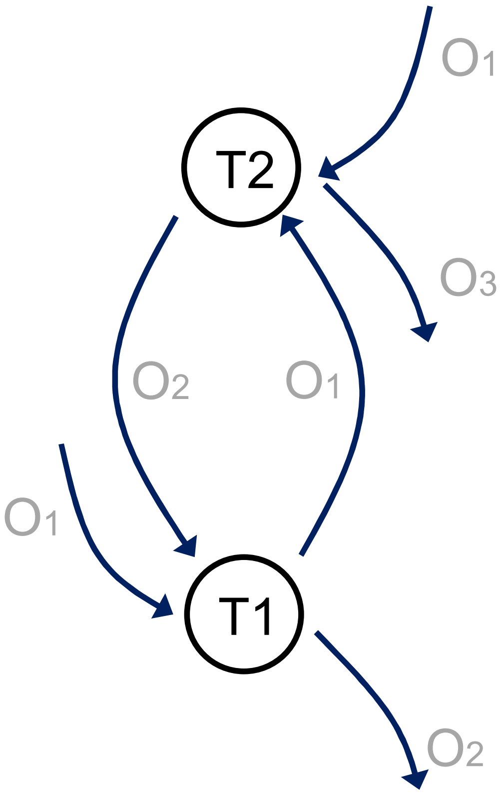
…
T1 **read** O_1
…
T2 **read** O_2
…



- Write accesses to data create dependencies in the concurrent execution of different transactions.
- No cycles means sequential execution is possible, isolation is guaranteed

write → write dependency and lost update

…
T2 read O_1
…
T1 write O_2
…
T2 write O_3
…

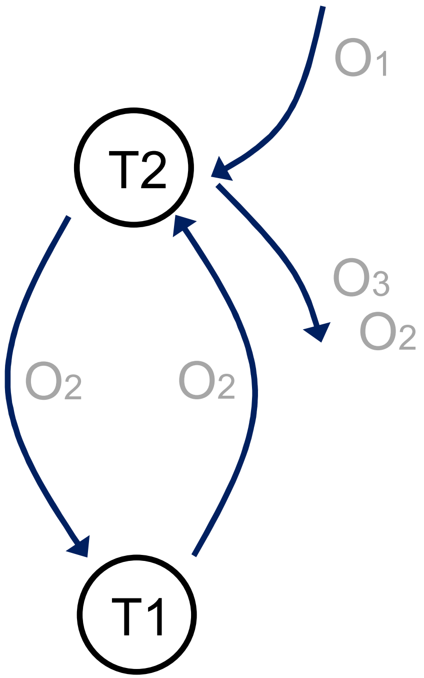


T1	T2	A
`read(A, x);`		10
	`read(A, y);`	10
`x := x + 1;`		10
	`y := y + 1;`	10
`write(x, A);`		11
	`write(y, A);`	11

- A cycle in the dependency graph caused by a **write** → **write** dependency ***can*** result in a so-called **lost update**.

write → read dependency and dirty read

…
T2 **write** O_2
…
T1 **read** O_2
…
T2 **write** O_3
…



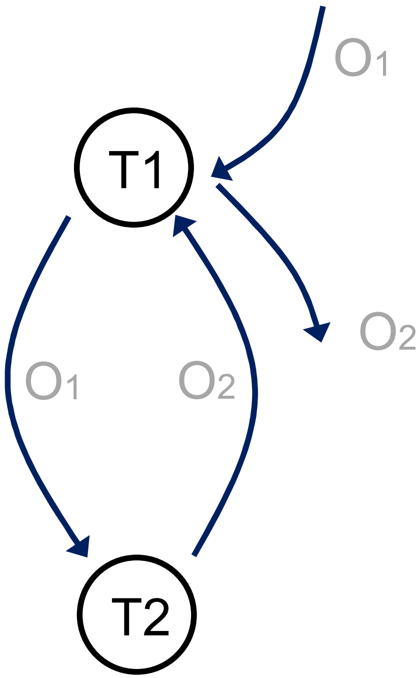
T1	T2
`read(A, x);`	
`x := x / 100;`	
`write(x, A);`	
	`read(A, x);`
	`read(B, y);`
	`y := y + x;`
	`write(y, B);`
	`commit;`
`abort;`	

O2

- A cycle in the dependency graph caused by a **write** → **read** dependency ***can*** result in a so-called **dirty read**.

read → **write** dependency and unrepeatable read

…
T1 **read** O_1
…
T2 **write** O_2
…
T1 **read** O_2
…



T1	T2
read(A, x);	
	read(A, y);
	y := y / 2;
	write(y, A);
	read(C, z);
	z := z + y;
	write(z, C);
	commit;
read(B, y);	
x := x + y;	
read(C, z);	
x := x + z;	
commit;	

O_2

- A cycle in the dependency graph caused by a **read** → **write** dependency ***can*** result in a so-called **unrepeatable read**.

Phantom problem

- Not only tuples in the database can be objects, but also indexes.

- Example of a special case of unrepeatable read on an index. Here, the index is changed during the execution of T1 and therefore the calculation of the bonus at this point in time is no longer correct.

T1	T2
select count (*) **into** X **from** Customer	
	insert **into** Customer **values** (‘Meier‘, 0, …)
	commit;
update Customer **set** Bonus = Bonus +10000/X;	
commit;	

Summary: problems with multi-user operations

- Lost update: lost changes
- Dirty read: dependencies on unreleased data
- Phantom problem: does the data set exist or not?
- Unrepeatable read: inconsistent reading

Chapter 9: Transactions, integrity and triggers

Examples of interleaved executions

- Two transactions
  - T_1 : `read(A,x); x:=x-10; write(x,A); read(B,y); y:=y+10; write(y,B);`
  - T_2 : `read(B,y); y:=y-20; write(y,B); read(C,z); z:=z+20; write(z,C);`
- Examples of interleaved executions

Execution 1	
T1	**T2**
read(A,x);	
x := x-10;	
write(x,A);	
read(B,y);	
y := y+10;	
write(y,B);	
	read(B,y);
	y := y-20;
	write(y,B);
	read(C,z);
	z := z+20;
	write(z,C);

Beispiele für verschränkte Ausführungen

- Zwei Transaktionen
  - T_1: `read(A,x); x:=x-10; write(x,A); read(B,y); y:=y+10; write(y,B);`
  - T_2: `read(B,y); y:=y-20; write(y,B); read(C,z); z:=z+20; write(z,C);`

- Beispiele für verschränkte Ausführungen

Ausführung 1		Ausführung 2		Ausführung 3	
T1	**T2**	**T1**	**T2**	**T1**	**T2**
read(A,x);		**read**(A,x);		**read**(A,x);	
x := x-10;			**read**(B,y);	x := x-10;	
write(x,A);		x := x-10;			**read**(B,y);
read(B,y);			y := y-20;	**write**(x,A);	
y := y+10;		**write**(x,A);			y := y-20;
write(y,B);			**write**(y,B);	**read**(B,y);	
	read(B,y);	**read**(B,y);			**write**(y,B);
	y := y-20;		**read**(C,z);	y := y+10;	
	write(y,B);	y := y+10;			**read**(C,z);
	read(C,z);		z := z+20;	**write**(y,B);	
	z := z+20;	**write**(y,B);			z := z+20;
	write(z,C);		**write**(z,C);		**write**(z,C);

Serialisability (1)

- Effect of the different executions

	A	B	C	A+B+C
initial value	10	10	10	30
after Execution 1	0	0	30	30
after Execution 2	0	0	30	30
after Execution 3	0	20	30	50

- Definition of serialisability
  An interleaved execution of multiple transactions is called serialisable if its effect is identical to the effect of a (randomly selected) serial execution of these transactions.

Serialisability (2) – read/write model

- The read/write model
  Transaction T is a finite sequence of operations (steps) p_i of the form $r(x_i)$ or $w(x_i)$:

  $$T = p_1 p_2 p_3 \ldots p_n \text{ with } p_i \in \{r(x_i), w(x_i)\}$$

- The complete transaction T has as its last step either an abort a or a commit c:

  $$T = p_1 \ldots p_n a$$
  or
  $$T = p_1 \ldots p_n c.$$

Serialisability (3) - schedule

- A complete schedule is a sequence of DB operations where all operations belong to complete transactions and all operations of those transactions occur in the same relative order in the schedule as in the transaction.

- A schedule is a prefix of a complete schedule.

- Example:

$$\underbrace{\underbrace{r_1(x)\ r_2(x)\ w_1(x)}_{\text{Schedule}}\ r_2(y)\ a_1\ w_2(y)\ c_2}_{\text{Complete schedule}}$$

Serialisability (4) – serial schedule

- A serial schedule s for $T_1, \ldots, T_n$ is a complete schedule in the following form:

$$s := T_{\rho(1)}, \ldots, T_{\rho(n)} \text{ for a permutation } \rho \text{ of } \{1, \ldots, n\}$$

- Example: serial schedules for two transactions
$T_1 := r_1(x)\ w_1(x)\ c_1$ and $T_2 := r_2(x)\ w_2(x)\ c_2$:

$$s_1 := \underbrace{r_1(x)\ w_1(x)\ c_1}_{T_1}\ \underbrace{r_2(x)\ w_2(x)\ c_2}_{T_2}$$

$$s_2 := \underbrace{r_2(x)\ w_2(x)\ c_2}_{T_2}\ \underbrace{r_1(x)\ w_1(x)\ c_1}_{T_1}$$

Serialisability (5) – correctness criterion

- A schedule s is correct if the effect of the schedule s (result of executing the schedule) is equivalent to the effect of a (random) serial schedule s' with respect to the same set of transactions (in characters $s \approx s'$).

- If a schedule s is equivalent to a serial schedule s', then s is serialisable (to s').

- Question: how do we ensure serialisability with maximum parallelism?

→ Optimistic techniques (try and undo if necessary)

→ Pessimistic techniques (use locking protocols)

Chapter 9: Transactions, integrity and triggers

Locking protocols

- Ensure serialisability through exclusive access to objects (synchronisation of the accesses)
- Implementation via locking and locking protocols
- Locking protocol guarantees serialisability without additional tests!

Locking models (elementary locks)

- Write locks and read locks in the following notation:
  - $rl(x)$: read lock on an object x
  - $wl(x)$: write lock on an object x
  - read unlock $ru(x)$ and write unlock $wu(x)$, often summarised as $u(x)$ unlock

- Compatibility matrix for elementary locks

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	ok	-
$wl_j(x)$	-	-

Locking rules

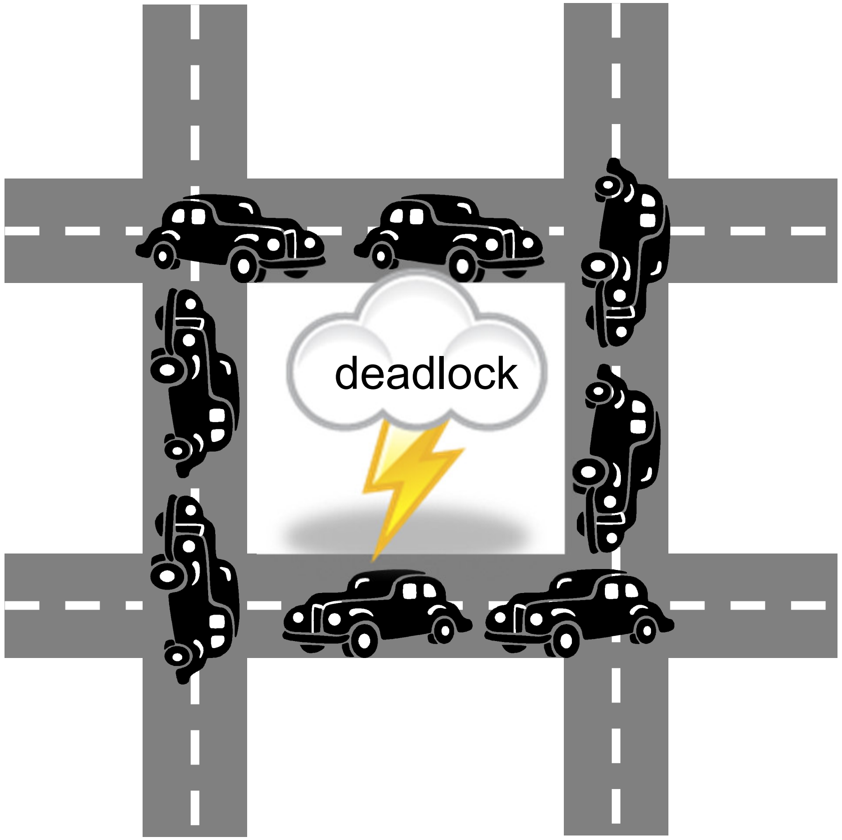
- Write access $w(x)$ is only possible after acquiring a write lock $wl(x)$
- Read accesses $r(x)$ are only permitted after $rl(x)$ or $wl(x)$
- Only lock objects that are not already locked by another transaction
- Locks of the same type are set a maximum of once, i.e. more precisely
  - after $rl(x)$, only $wl(x)$ is permitted, then no more lock on x
  - after $u(x)$ by T_i, T_i is not allowed to execute $rl(x)$ or $wl(x)$ again
- Before a **`commit`**, all locks must be released

Deadlocks

- Example

deadlock

	T1	T2	
	wl(x)		
		wl(y)	
wait →	wl(y)		
		wl(x)	← wait

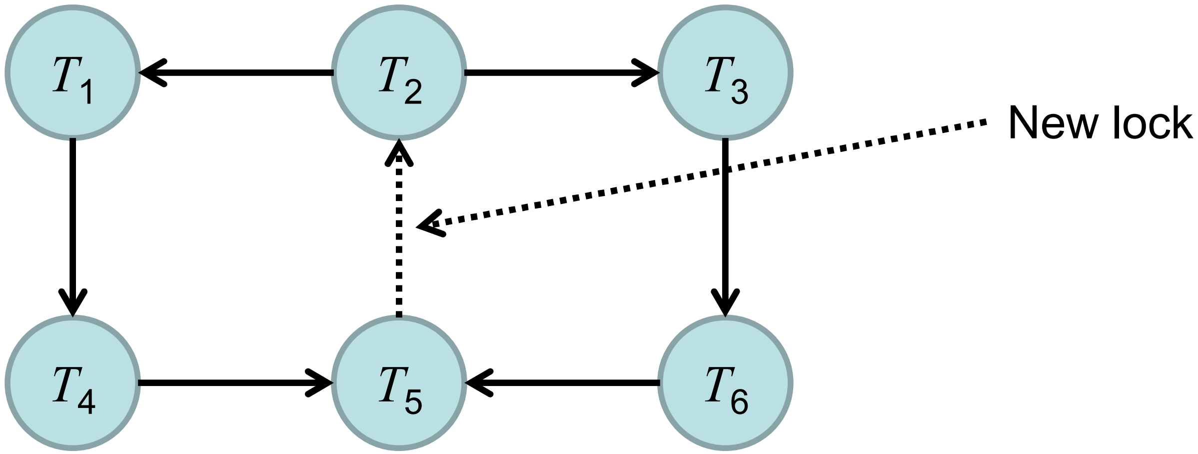


- Alternatives
  - deadlocks are detected and **resolved**
  - deadlocks are **avoided** from the outset

Source: http://minutillo.com/steve/weblog/2003/1/21/deadlock/

Deadlock detection and resolution

- Waiting chart



- Resolving by **terminating a transaction**, criteria:
  - Number of cycles initiated
  - Length of a transaction
  - Reset effort of a transaction
  - Importance of a transaction
  - . . .

Need for locking protocols

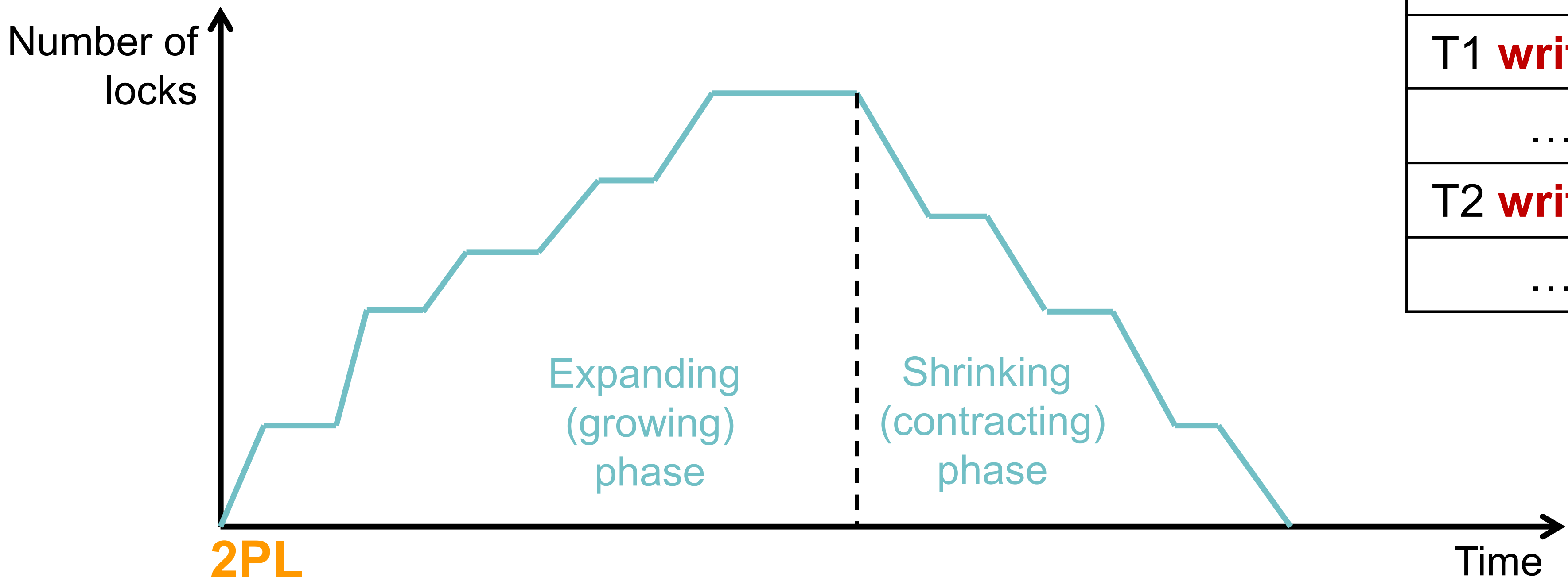
- Example: in this case, the locking (`wl`) takes place correctly, and after access (`w`), the locking is enabled once again (`u`). But this doesn't help, because T1 overwrites y and T2 overwrites x.

T1	T2
wl(x)	
w(x)	
u(x)	
	wl(x)
	w(x)
	u(x)
	wl(y)
	w(y)
	u(y)
wl(y)	
w(y)	
u(y)	

Two-phase locking protocol

- Two-phase locking (2PL) protocol

…
T2 **read** O_1
…
T1 **write** O_2
…
T2 **write** O_3
…



- If a new lock is not requested after a lock has been released, then cycles in the dependency graph cannot lead to isolation problems because the objects are blocked.

Deadlocks and the snowball effect

- During the step-by-step expanding phase (growing phase) of the two-phase locking protocol, deadlocks can occur due to cycles in the dependency graph because then transactions wait for each other for their locks to be released.
  - T1 waits for T2 for the release of O,
    T2 waits for T1 for the release of U

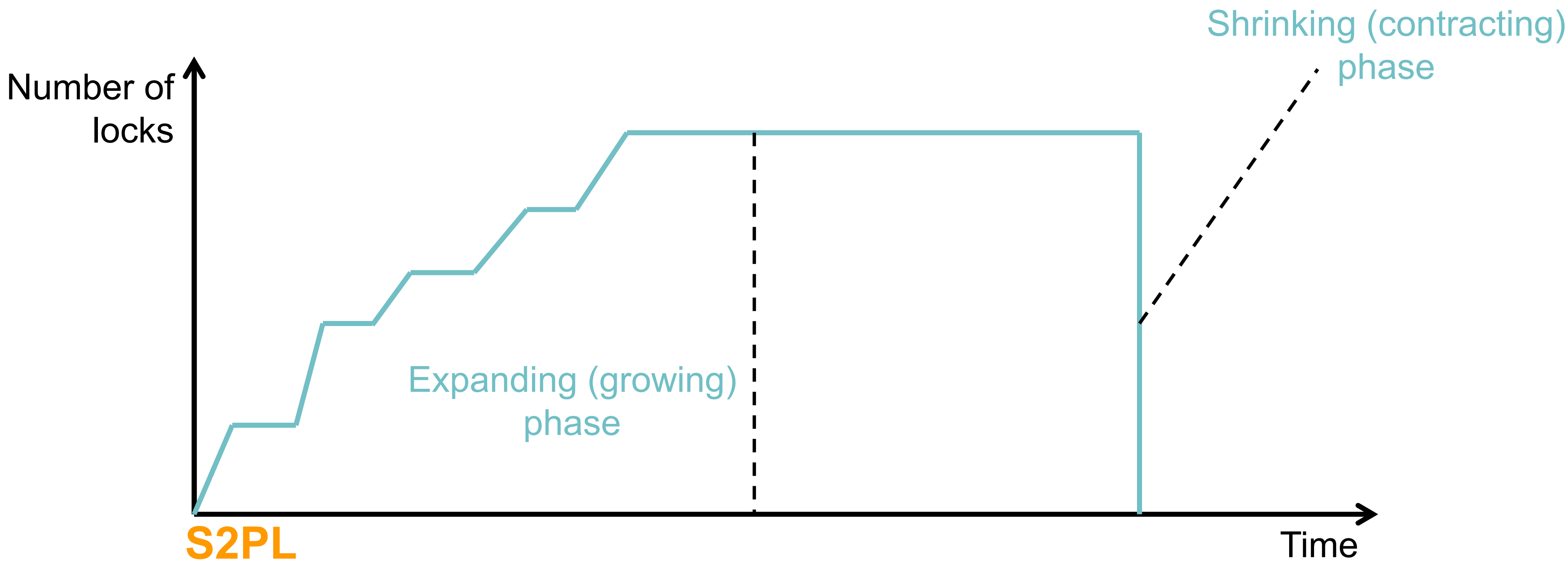
…
T2 **read** O
T1 **read** U
T1 **write** O
T2 **write** U
…

- During the step-by-step release of locks in the shrinking phase (contracting phase) before the end of the transaction, cascading resets can occur if a transaction is reversed but the releases have already been worked on.
  - T2 changes O, releases O, T1 reads O, T2 is reversed, T1 is thus invalid.

…
T2 **write** O2
…
T1 **read** O2
…
T2 **abort**
…

Strict two-phase locking protocol

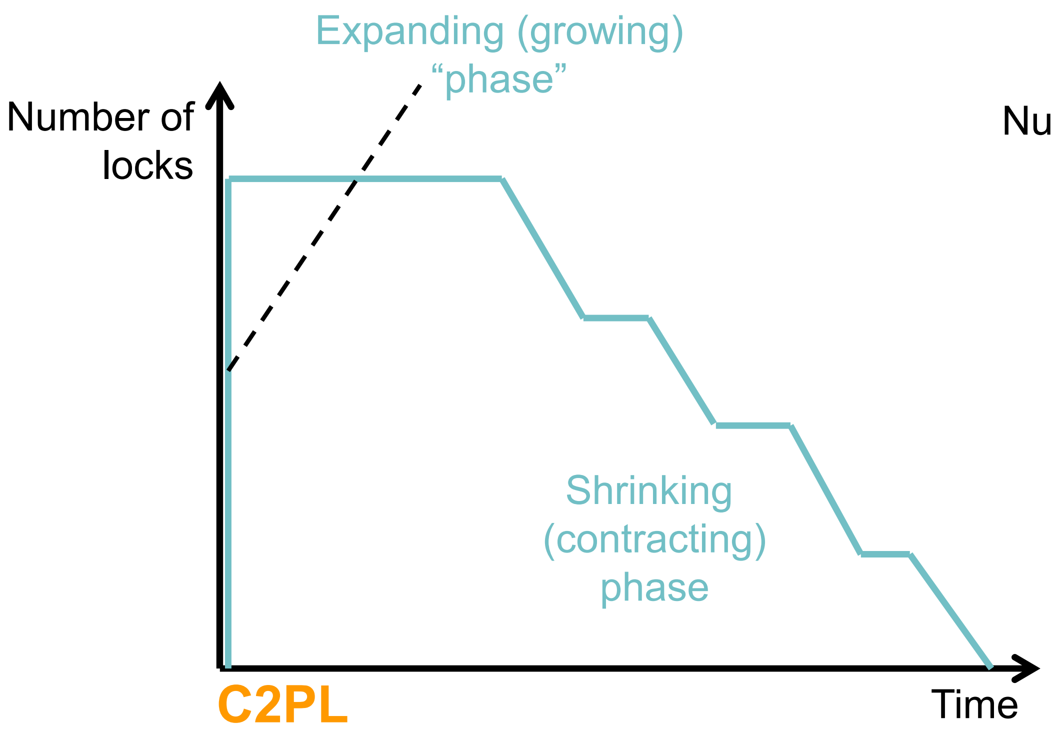
- Strict two-phase locking (S2PL) protocol

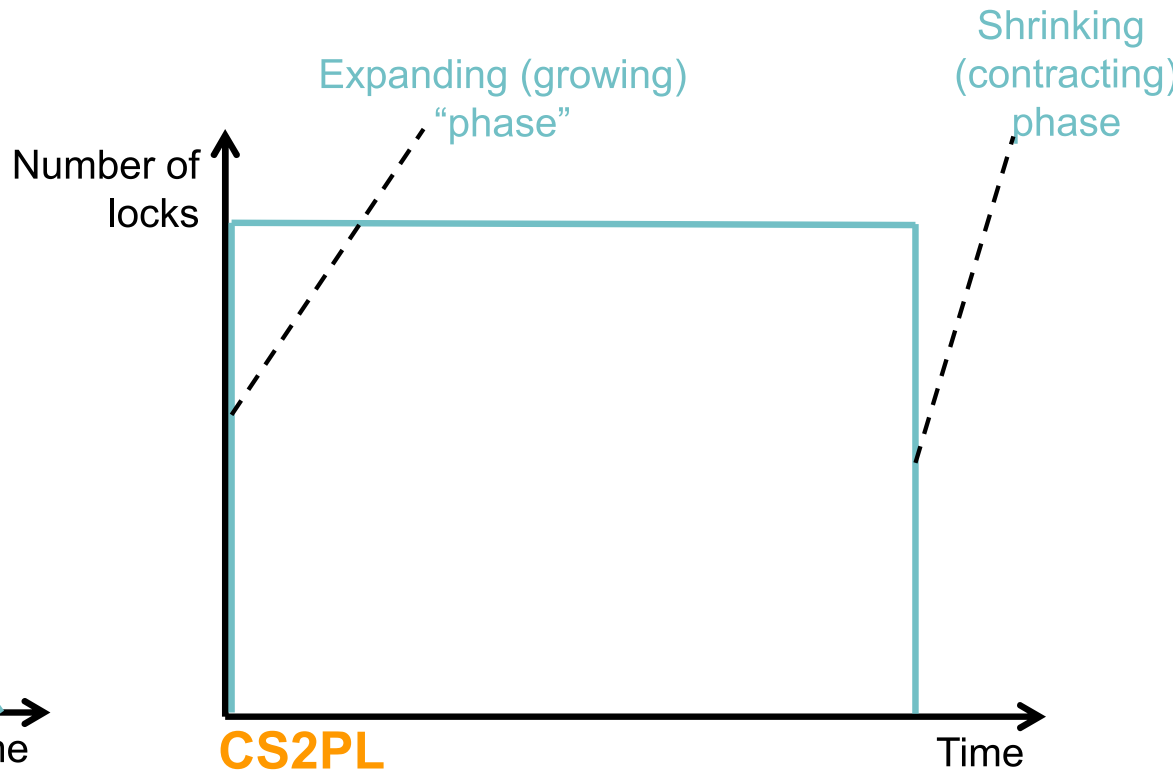


- Avoids cascading terminations by release after completion of the transaction, deadlocks are possible

Conservative two-phase locking protocol

- Conservative two-phase locking (conservative 2PL, C2PL) protocol
- Conservative strict two-phase locking (CS2PL) protocol





- Avoids deadlocks, but CS2PL usually results in sequential execution. Predicting all locks is often impossible (cf. static allocation)

Additional locking levels

- More complex locks are possible to increase parallelism
  - Shared (Read) (S)
  - Update Lock (U)
  - Exclusive Lock (X)
- Often still hierarchical locks
  - Intent Shared (S)
  - Intent Exclusive (IX)
  - Shared with Intent Exclusive (SIX)
- And a few more...

	Existing granted mode					
Requested mode	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

Lock compatibility in MS SQL Server [www.microsoft.com]

Chapter 9: Transactions, integrity and triggers

Isolation levels in SQL

- Increasing performance: loosening of serialisability

```
set transaction [ { read only | read write }, ]
[isolation level { read uncommitted |
                   read committed |
                   repeatable read |
                   serializable }, ]
...
```

- Standard

```
set transaction read write,
      isolation level serializable
```

Meaning of the isolation levels (1)

- **`read uncommitted`**
  - weakest level: access to data that is not written, only for **`read only`** transactions
  - statistical and similar transactions (approximate overview, incorrect values)
  - no locks → can be executed efficiently, other transactions are NOT hindered
- **`read committed`**
  - only reads final written values, but *unrepeatable read* possible
- **`repeatable read`**
  - no *unrepeatable read*, but *phantom problem* can occur
- **`serializable`**
  - guaranteed serialisability

Meaning of the isolation levels (2)

- Occurring () and avoided () problems per isolation level

Isolation level	Dirty read	Unrepeatable read	Lost update	Phantom read
read uncommitted				
read committed				
repeatable read				
serializable