Concurrent approach to a database

Motivation example:

Bank transfer 100,- Kč from account "A" to account "B" and concurrent withdrawal of 200 Kč from account "B".

Transction	Variable A	Variable B	Account A balance	Account B balance
			1000,-	1000,-
T1: read A	T1: 1000			
T1: read B		T1: 1000		
T1: subtract 100 from A	T1: 900			
T1: add 100 to B		T1: 1100		
T1: write A			900,-	
T2: read B		T2: 1000		
T1: write B				1100,-
T2: subtract 200 from B		T2: 800		
T2: Write B				800,-
Resulting balance			900,-	800,-
Expected balance			900,-	900,-

Concurrent transaction may violate DB consistency even if each pf the transaction (if executed alone) would not violate DB consistency.

Concurrent approach to a database

Transactio:

ACID property:

Atomicity:	atomicity – either complete or nothing
Consistency	transaction must be correct w.r.t. sustaining invariants – integrity constrains
Isolation	(isolation = serializability). Even if being executed concurrently, the result is the same as if execited serially
Durability	Data modification carried out by a successfully completed transaction are persistent (durable) even in case of an accident (failure/accident recovery).

Concurrent approach to a database

Serializable execution of transactions:

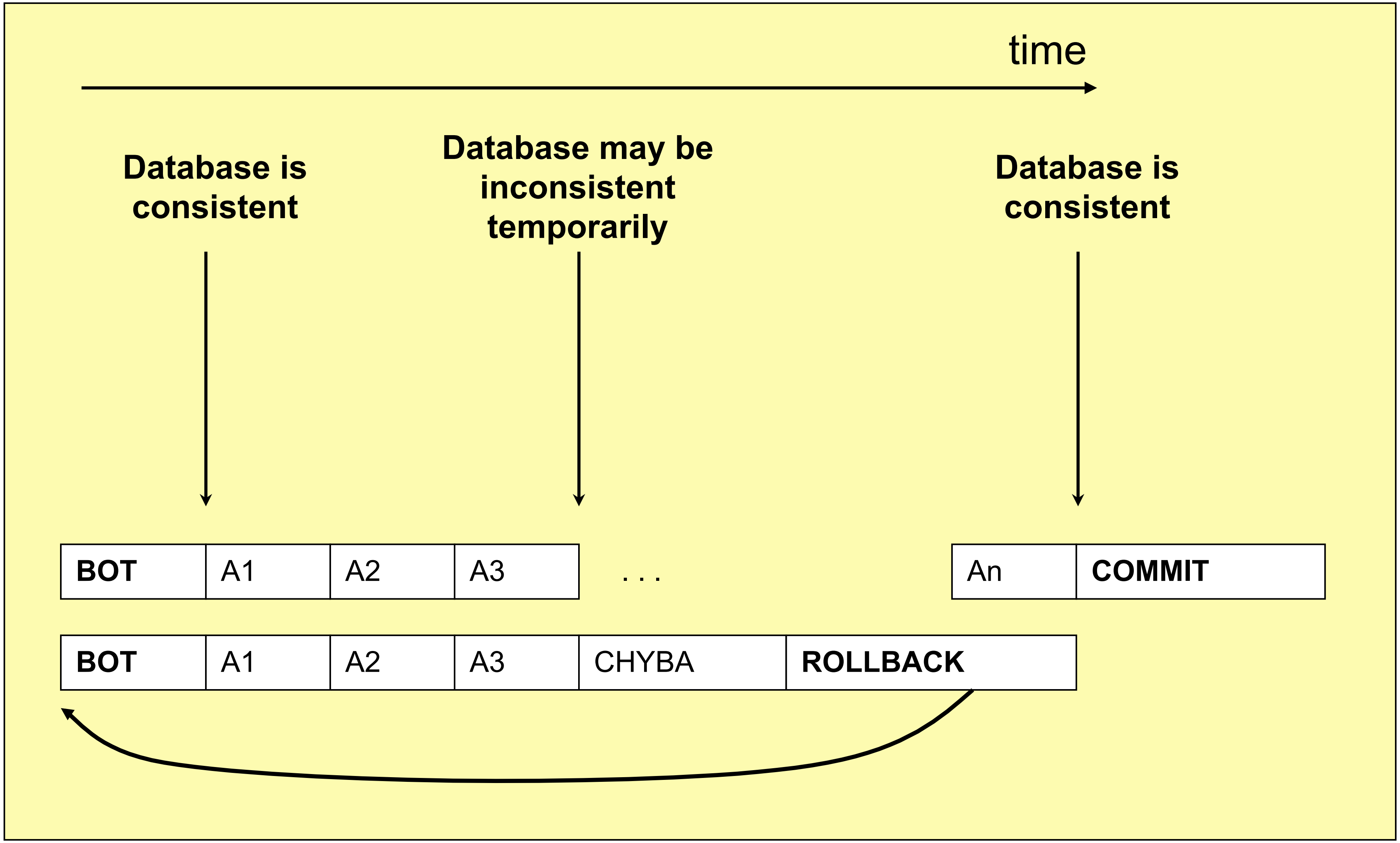
- Multiple transactions running „in parallel“ (higher throughput of the system)
- Result equivalent to a serial execution.

Serializability - methods:

- locking on various granularity levels:
  - locking of the complete DB (=> serial execution)
  - table locking
  - row locking
- time stamps
- MVCC (multiversion concurrency control)
- predicate locks

Concurrent approach to a database

Transaction = sequence of **read** / **write** actions on DB objects
(insert a delete not taken into account, yet).



time

Database is consistent

Database may be inconsistent temporarily

Database is consistent

BOT | A1 | A2 | A3 | . . . | An | **COMMIT**

BOT | A1 | A2 | A3 | CHYBA | **ROLLBACK**

Concurrent approach to a database

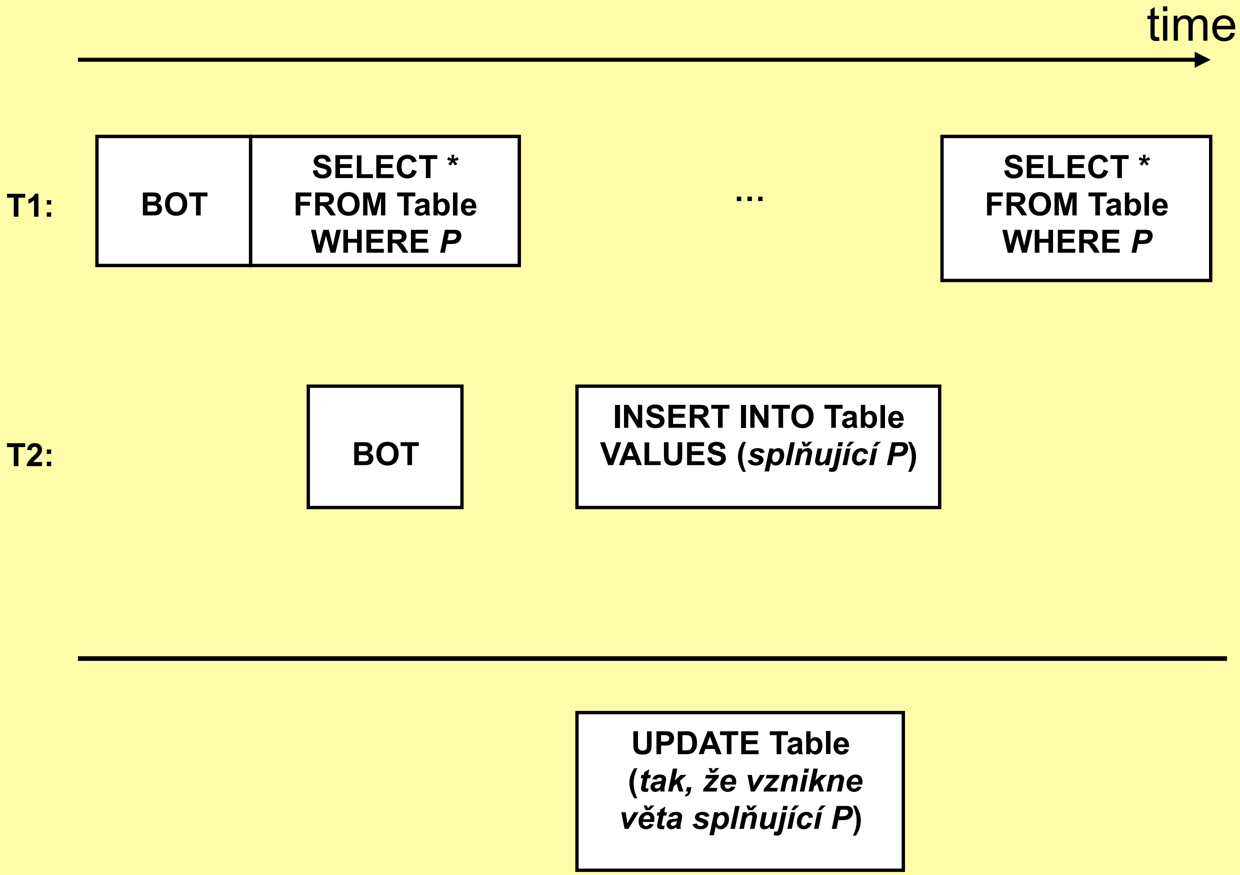
- Multiple **read**s of the same object **cannot** violate the consistency.
- Multiple **write**s of the same object within one transaction need not be taken into account (transakce is correct – see “C” in ACID).
- Only **read**s **a** **write**s executed within different transactions **may** violate the consistency.

Concurrent approach to a database

Lost update	**T1 WRITE**	**<o,1>**	Version 1 of object o will not sustain. As if T1 never run.
	T2 WRITE	**<o,2>**	
	T1 READ	<o,**2**>	
Dirty read	**T2 WRITE**	**<o,2>**	
	T1 READ	**<o,2>**	T1 read a temporary (not committed) value
	T2 ROLLBACK	<o,**1**>	
Unrepeatable read	**T1 READ**	**<o,1>**	
	T2 WRITE	**<o,2>**	
	T1 READ	<o,**2**>	unrepeatable read
Phantom problem	To be explained on one of next slides		

Concurrent approach to a database

Phantom problem



T1:

BOT

SELECT * FROM Table WHERE *P*

…

SELECT * FROM Table WHERE *P*

T2:

BOT

INSERT INTO Table VALUES (*splňující P*)

UPDATE Table (*tak, že vznikne věta splňující P*)

Concurrent approach to a database

LOST UPDATE - an example:

Transaction T_1 : withdrawal of the complete balance from account A.

Transakce T_2 : add 3% interests to account A.

An example of a transactional history (aka schedule):

Step	T_1	T_2
1.	BOT	
2.		BOT
3.	$a_1 := 0$	
4.		READ(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.	**WRITE(A, a_1)**	
7.		**WRITE(A, a_2)**
8.	COMMIT	
9.		COMMIT

Concurrent approach to a database

DIRTY READ – an example:

Transaction T_1: transfer 300,- Kč from account A to account B.

Transaction T_2: add 3% interests to account A, that is in an inconsistent status at the moment.

An example of a transactional history (aka schedule):

Step	**T_1**	**T_2**
1.	READ(A, a_1)	
	00	
	a_1)	
		READ(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.		WRITE(A, a_2)
7.	READ(B, b_1)	
	READ selhal, proto:	
8.	ROLLBACK	

ROLLBACK returns status to what it was at the beginning of transactionT1, This will cause that the whole effect of transaction T2 is lost.

T2 is to blame - it read a data object that was not confirmed (comitted) yet.

Concurrent approach to a database

UNREPEATABLE READ an example:

Transaction *T_1* transfers 300,- Kč from account A to account B.

Transaction *T_2* adds 3% interests to account A, that is in an inconsistent status at the moment.

An example of a transactional history (aka schedule):

Step	***T_1***	***T_2***
1.	**READ(A, a_1)**	
2.		READ(A, a_2)
3.		$a_2 := a_2 * 1.03$
4.		**WRITE(A, a_2)**
5.	$a_1 := a_1 – 300$	
6.	WRITE(A, a_1)	
7.	READ(E	
8	$b_1 := b_1$	
8.	WRITE(	

T_2 overwrote a data object, that transaction T_1 rad in and is going to work with it in the future -> T_1 will work with inconsistent data.

Variable a_1 does not reflect the status of the database. If we carried our READ(A,a_1) again, the contents of the variable a_1 would be different!

Concurrent approach to a database

PHANTOM PROBLEM – an example:

In the course of processing T_2, transaction T_1 introduces a new record to the database. Hence, the second SELECT will return different result.

An example of a transactional history (aka schedule):

Step	T_1	T_2
1.		SELECT sum(*StavUctu*) FROM *Ucty*
2.	INSERT INTO *Ucty* VALUES (*StavUctu*, *1000*)	
3.		SELECT sum(*StavUctu*) FROM *Ucty*

Concurrent approach to a database

Lost update	**T1 WRITE**	**<o,1>**	Version 1 of object o will not sustain. As if T1 never run.
	T2 WRITE	**<o,2>**	
	T1 READ	<o,2>	
Dirty read	**T2 WRITE**	**<o,2>**	
	T1 READ	**<o,2>**	T1 read a temporary (not committed) value
	T2 ROLLBACK	<o,1>	
Unrepeatable read	**T1 READ**	**<o,1>**	
	T2 WRITE	**<o,2>**	
	T1 READ	<o,2>	unrepeatable read
Phantom problem	T1 SELECT predicate	{ o1, o2}	
	T2 INSERT o3		
	T1 SELECT predicate	{ o1, o2}	

Transactional history (transaction schedule) – a sequence of actions belonging to several transactions that sustains the order in which the actions were executed.

History (**schedule**) is called **serial**, if all steps of one transaction were executed before all steps of the other transaction.

	Serialized hisotry		**Serial historie**	
Step	T_1	T_2	T_1	T_2
1	**BOT**		**BOT**	
2	READ(A)		READ(A)	
3		**BOT**	WRITE(A)	
4		READ(C)	READ(B)	
5	WRITE(A)		WRITE(B)	
6		WRITE(C)	**COMMIT**	
7	READ(B)			**BOT**
8	WRITE(B)			READ(C)
9	**COMMIT**			WRITE(C)
10		READ(A)		READ(A)
11		WRITE(A)		WRITE(A)
12		**COMMIT**		**COMMIT**

SERIALIZABILITY theory:

Let a transakce T_i consists of the following elementary operations:

- **READ$_i$(A)** – read object A in context of transaction T_i
- **WRITE$_i$(A)** - write (modify) object A in context of transaction T_i
- **ROLLBACK$_i$ –** revert all objects modified by T_i to the status as it was at the beginning of T_i
- **COMMIT$_i$ –** confirmation of the successful end of T_i

4 cases possible:

READ$_I$(A) - READ$_J$(A)	No conflict	Order not significant
READ$_I$(A) - WRITE$_J$(A)	**Conflict**	Order significant
WRITE$_I$(A) - READ$_J$(A)	**Conflict**	Order significant
WRITE$_I$(A) - WRITE$_J$(A)	**Conflict**	Order significant

Only (mutually) conflicting operations are interesting.

Two **histories H_1 a H_2** (on the same set of transactions) are equivalent, iff **all conflicting operations** of (non-interrupted transactions are carried out **in the same order**.

For any two equivalent histories and an ordering $<_{H1}$ induced by history H_1 and $<_{H2}$ induced by history H_2 the following holds: if p_i and q_j are conflicting operations such that $p_i <_{H1} q_j$, the following has to hold $p_i <_{H2} q_j$, too. The order of non-conflicting operations is not interesting.

Not every history is serializable:

	Non-serializable history	
Step	T_1	T_2
1	**BOT**	
2	READ(A)	
3	WRITE(A)	
4		**BOT**
5		READ(A)
6		WRITE(A)
7		READ(B)
8		WRITE(B)
9		**COMMIT**
10	READ(B)	
11	WRITE(B)	
12	**COMMIT**	

Reason:

Transakce T_1 **is before** T_2 when processing object **A**, but T_2 **is before** T_1 when processing objectu **B**.

This is why this history is not equivalent neither to serial execution T_1T_2 or to serial execution T_2T_1.

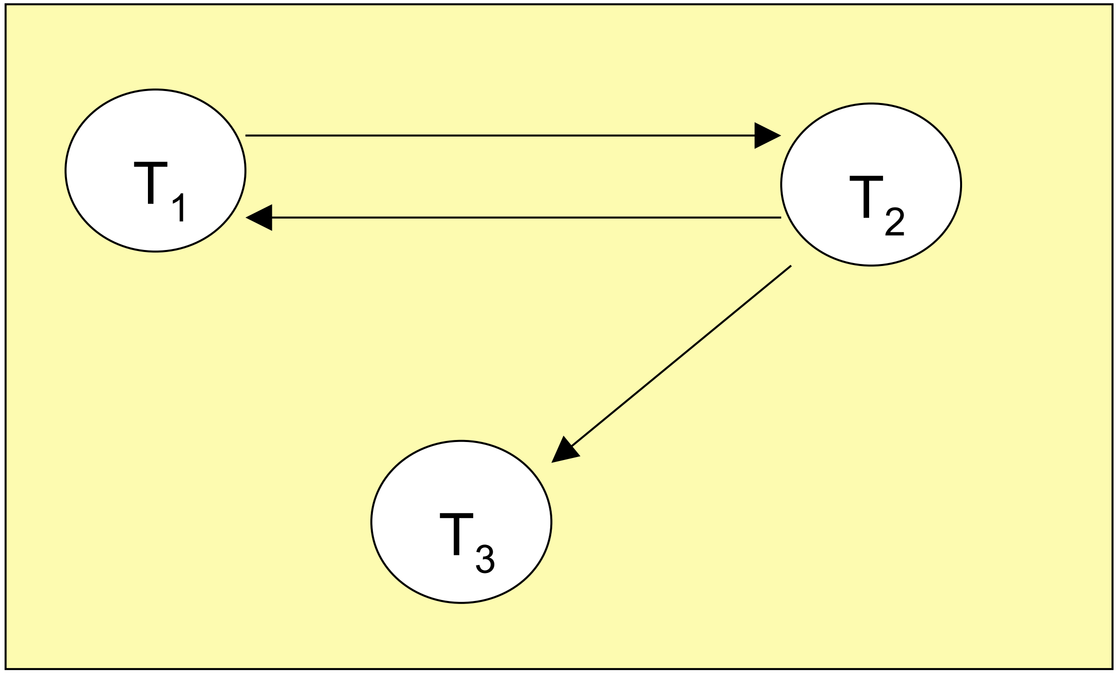
Hence, this history **is not serializable**.

Serializability theory (II)

Example: Let H be a history of three transactions T_1, T_2, T_3:

$w_2(B) < r_1(B)$; $w_1(A) < r_2(A)$; $w_2(C) < r_3(C)$; $w_2(A) < r_3(A)$

Graph of dependency: $T_2 \rightarrow T_1$ $T_1 \rightarrow T_2$ $T_2 \rightarrow T_3$ $T_2 \rightarrow T_3$



History H is serializable iff its graph of dependency is acyclic.

Locking:

2 types of locks:

- SLOCK: Shared lock.
- XLOCK: eXclusive lock.

Well formed transaction:

- Before any READ of a DB object, this DB objects has to be locked by SLOCK,
- Before any WRITE to a DB object, this DB object has to be locked by XLOCK
- UNLOCK of a DB object can be done only if the object is locked with SLOCK/XLOCK
- any SLOCK/XLOCK is followed by corresponding UNLOCK in the course of the transaction.

Locks compatibility

		Existing lock		
		not locked	SLOCK	XLOCK
Requested lock	SLOCK	OK	OK	Conflict
	XLOCK	OK	Conflict	Conflict

Legal history:

Any history following the lock compatibility rules is called **legal history.**

Actions and transactions

Actions on objects: READ, WRITE, XLOCK, SLOCK, UNLOCK

Global actions: BEGIN, COMMIT, ROLLBACK

T'	BEGIN		T''	BEGIN	
	SLOCK	A		SLOCK	A
	XLOCK	B		READ	A
	READ	A		XLOCK	B
	WRITE	B		WRITE	B
	COMMIT			ROLLBACK	

Let us rid off the COMMIT and ROLLBACK operations by a conversion to a (from consistence perspective) equivalent transaction model – see next page

Simple transaction:

1) Consists of READ, WRITE, XLOCK, SLOCK a UNLOCK.
2) COMMIT replaced with a sequence commands UNLOCK A, for each DB object A, that was locked by SLOCK A or XLOCK A in the course of T
3) ROLLBACK replaced with a sequence of actions:
   - WRITE A for each DB object A, tha was subject of WRITE A in the course of T
   - UNLOCK A for each DB object A that was locked by SLOCK A or XLOCK A in the course of T.

T'	SLOCK	A	T''	SLOCK	A
	XLOCK	B		READ	A
	READ	A		XLOCK	B
	WRITE	B		WRITE	B
	UNLOCK	A		WRITE (undo)	B
	UNLOCK	B		UNLOCK	A
				UNLOCK	B

Two-phase transaction

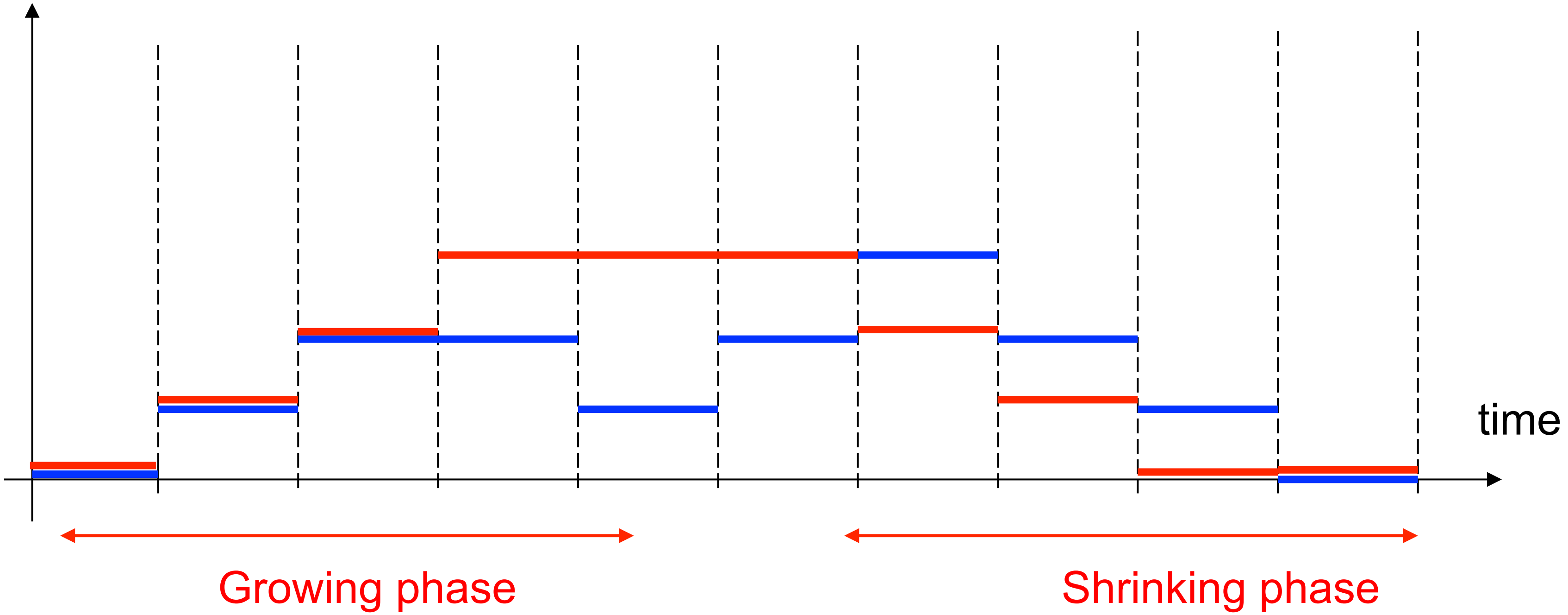
All LOCK actions carried out before all UNLOCK actions.

Growing phase - all LOCK actions carried out in the course of the growing phase.

Shrinking phase – all UNLOCK actions carried out in the course of the schrinking phase

Two-phase transaction: growning and shrinking phases do not overlap.

The number of applied locks:



A transaction is serializable (with exception of the phantom problem), iff:

- it is well formed
- it is legal
- it is two-phase
- and holds all XLOCKS until COMMIT/ROLLBACK

Isolation degrees

(our simplified model – we still do not consider phantoms)

	Transaction	**Name**	**Locking protocol**
0°	0° T does not overwrite dirty data of another transaction, if this (the other) transaction is at least 1°	anarchie	well formed for WRITE
1°	1° T does not have lost updates	browse	Two-phase for XLOCK and well formed for WRITE
2°	2° T does not have lost updates and/or dirty reads		Two-phase for XLOCK a well formed for WRITE and READ
3°	3° T does not have lost updates, dirty reads and/or unrepeatable reads	isolated transaction serializable repeatable read	Two-phase for XLOCK i SLOCK and well formed for WRITE and READ

Cursor

```
char title[51], year[11], result[102, star_name[51];
```

```
EXEC SQL DECLARE CURSOR movie_cursor FOR
SELECT title, CAST (year_released AS CHARACTER(10))
FROM movie_titles;
```

```
while (/* cyklus pres jednotlive filemy */ )
{
        EXEC SQL FETCH NEXT FROM movie_cursor INTO :title, :year ;
 ...

}
```

Cursor stability

SQL DBMSs usually implement an enriched protocol 2° called **cursor stability.**

Shared lock applied to records addressed by a (some) cursor
⇒ **cursor stability.**

One of particular implementations described in
http://jazz.external.hp.com/training/sqltables/c5s38.html

Cursor stability

FETCH operation:

1. Pointers in source tables will move so that they point to the next candidate cursor record.
2. Records of source tables referenced by pointers will be locked by SLOCK.
3. Check whether this candidate reallly belongs to the cursor.
4. If not, release (unlocke) SLOCKS andgo to point 1.
5. If yes, the records remain locked by SLOCK until the cursor is closed. If a record will be modified, the corresponding source tables records locks will be changed from SLOCK to XLOCK.

The imoortant aspect is that the FETCH operation does not unlock the previous record.

```
SET TRANSACTION ISOLATION LEVEL    [ READ UNCOMMITTED]
                                   [ READ COMMITTED ]
                                   [ REPEATABLE READ ]
                                   [ SERIALIZABLE ]
```

READ UNCOMMITTED - 1° browse - for read-only transactions
READ COMMITTED - cursor stability (improved 2°)
REPEATABLE READ - 3° without phantom protection
SERIALIZABLE - 3° with phantom protection

Method of timestamps:

- On start of any transaction, the transaction receives a timestamp (at the rate of 1 tick/ms – 32 bit timestamp is enough for 49 days)
- If a transaction accesses an DB object for READ, the object‘s **tr** timestamp will be assigned the highest timestamp of all transactions that reading the object
- If a transaction accesses an DB object for WRITE, the object‘s **tw** timestamp will be assigned the timestamp of the particular transaction.

Constraints:

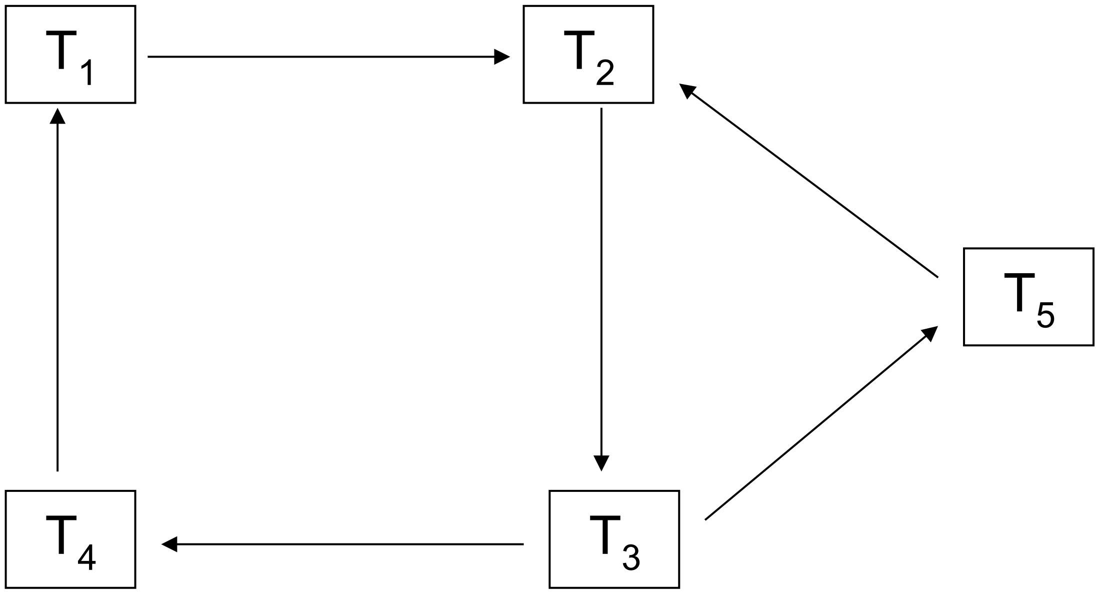
- Transaction with a timestamp **t** is not allowed to read objects with **tw** > **t**. ROLLBACK follows.
- Transaction with a timestamp **t** is not allowed to overwrite objects with **tr** > **t**. ROLLBACK follows.
- 2 transactions may read the same DB object at any moment.
- If a transaction with timestamp **t** is going to overwrite an object with **tw > t**, the transaction has to wait until **tw** is removed from the object.

Deadlock

Step	**T_1**	**T_2**	
1	**BOT**		
2	**LockX(A)**		
3		**BOT**	
4		**LockS(B)**	
5		Read(B)	
6	Read(A)		
7	Write(A)		
8	**LockX(B)**		T_1 has to wait for T_2
9		**LockS(A)**	T_2 has to wait for T_1
10	…	…	

Deadlock

Mutual waiting graph:



Removal of cycles – strategy:

- Rollback as young transaction as possible (to influence as few transactions as possible)
- Rollback a transaction with highest number of locks applied.
- Do not rollback a transaction tha was already rollbacked.
- Rollback a transaction, that participates in multiple cycles.

Phantom protection:

The only reliable protection – **predicate locks**.

SELECT * FROM T Where P1()

Predicate P1() is put to the list of active predicate locks.

If I wish to execute **INSERT INTO T ....** in parallel, I have to:

1. Check whether the record to be inserted does not meet any of the active predicate locks.
2. If yes, conflict, the INSERT can not be executed, its transaction needs to be rolled back.

Predicate locks computationally expensive => DB vendors usually do not implement them.

What else if not predicate locks?

- Timestamps
- MVCC – Multiversion Concurrency Control

MVCC – multiversion Concurrency Control

- The method is using timestamps
- Snapshot isolation
  - A „snapshot“ of (the relevant part of the) database is created.

  - Modifications done by this transaction are visible in this transactions‘s snapshot but not in the snapshots of the parallel transactions.

  - At the end of the transaction, a trial to execute a **commit** is done.

  - If the comitted data is in conflict (detected by means of timestamps) with updates of transactions that did the commit after our transaction created the snapshot, our transaction has to ROLLBACK.

ISO:	**Postgre SQL**
READ UNCOMMITTED	**READ COMMITTED**
READ COMMITTED	**READ COMMITTED**
REPEATABLE READ	**SERIALIZABLE**
SERIALIZABLE	**SERIALIZABLE**

READ COMMITED in PostreSQL:

Snapshot created at the beginning of SELECT

Notice that two successive **SELECT**s can see different data, even though they are within a single transaction, when other transactions commit changes during execution of the first **SELECT**.

SERIALIZABLE in PostreSQL:

Snapshot created at the beginning of the transaction.

This is different from Read Committed in that the **SELECT** sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction.

PostgreSQL – programmer‘s manual section 12.2.2.1:

Class	Value
1	10
1	20
2	100
2	200

Let us execute in parallel:

1. Insert result of *SELECT 2, SUM(value) FROM mytab WHERE class = 1;* into **mytab**
2. Insert result of *SELECT 1, SUM(value) FROM mytab WHERE class = 2;* into **mytab**

What will be the result?