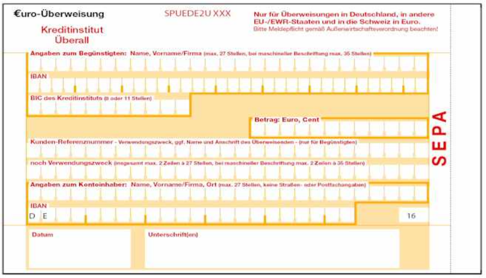
TRANSACTIONS



€uro-Überweisung

SPUEDE2U XXX

Nur für Überweisungen in Deutschland, in andere EU-/EWR-Staaten und in die Schweiz in Euro.
Bitte Meldepflicht gemäß Außenwirtschaftsverordnung beachten!

Kreditinstitut
Überall

Angaben zum Begünstigten: Name, Vorname/Firma (max. 27 Stellen, bei maschineller Beschriftung max. 35 Stellen)

IBAN

BIC des Kreditinstituts (8 oder 11 Stellen)

Betrag: Euro, Cent

Kunden-Referenznummer - Verwendungszweck, ggf. Name und Anschrift des Überweisenden - (nur für Begünstigten)

noch Verwendungszweck (insgesamt max. 2 Zeilen à 27 Stellen, bei maschineller Beschriftung max. 2 Zeilen à 35 Stellen)

Angaben zum Kontoinhaber: Name, Vorname/Firma, Ort (max. 27 Stellen, keine Straßen- oder Postfachangaben)

IBAN

D E

16

Datum

Unterschrift(en)

SEPA

Example: Transfer Euro 50 from A to B

1. Read balance of A from DB into Variable *a*: **read**(*A*,*a*);
2. Subtract 50.- Euro from the balance: $a := a - 50$;
3. Write new balance back into DB: **write**(*A*,*a*);
4. Read balance of B from DB into Variable *b*: **read**(*B*,*b*);
5. Add 50,- Euro to balance: $b := b + 50$;
6. Write new balance back into DB: **write**(*B*, *b*);

TRANSACTIONS

Definition: Transaction

Sequence of DML/DRL statements

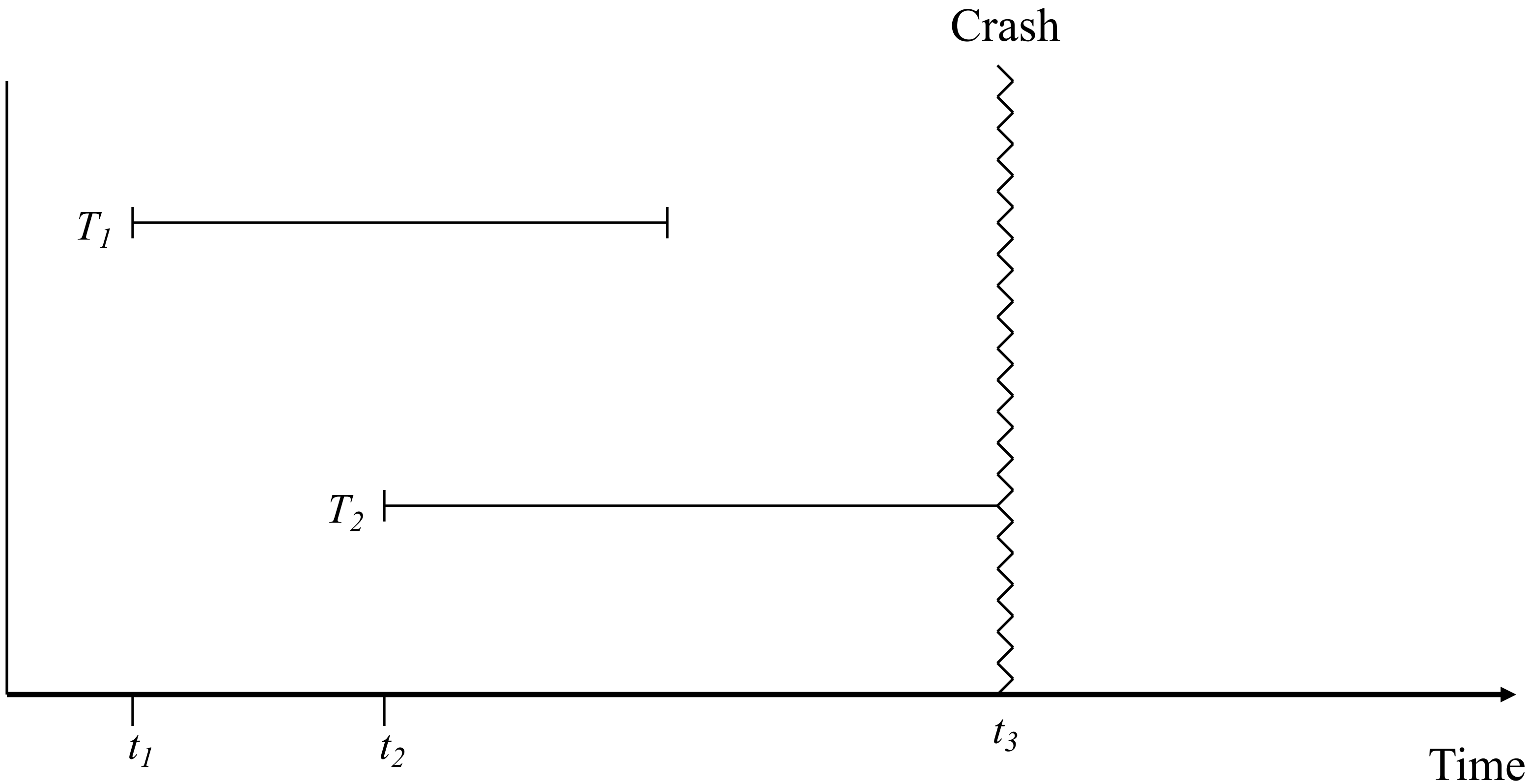
Transforms the data base from one consistent state to another consistent state -> ACID

ACID-Principle

Transactions obey the following four properties

- **Atomicity**: "All or Nothing"-Property (error isolation)
  → Undo changes if there is a problem
- **Consistency**: Maintaining DB consistency (defined integrity constraints)
  → Check integrity constraints at the end of a TA
- **Isolation**: Execution as if it is the only transaction in the system (no impact on other parallel transactions)
  → Synchronize operations of concurrent TAs
- **Durability**: Holding all committed updates even if the system fails or restarts (persistency)
  → Redo changes if there is a problem

Database Failures



Types of Failures: R1-R4 Recovery

1. Abort of a single TA (application, system)
   - *R1* Recovery: Undo a single TA
2. System crash: lose main memory, keep disk
   - *R2* Recovery: Redo committed TAs
   - *R3* Recovery: Undo active TAs
3. System crash with loss of disks
   - *R4* Recovery: Read backup of DB from tape

ACID-Principle cont.

The database system guarantees the ACID properties

What’s the task of the application programmer?

- ➢ Define borders of transactions
  - as large as necessary
  - as small as possible

Programming with Transactions

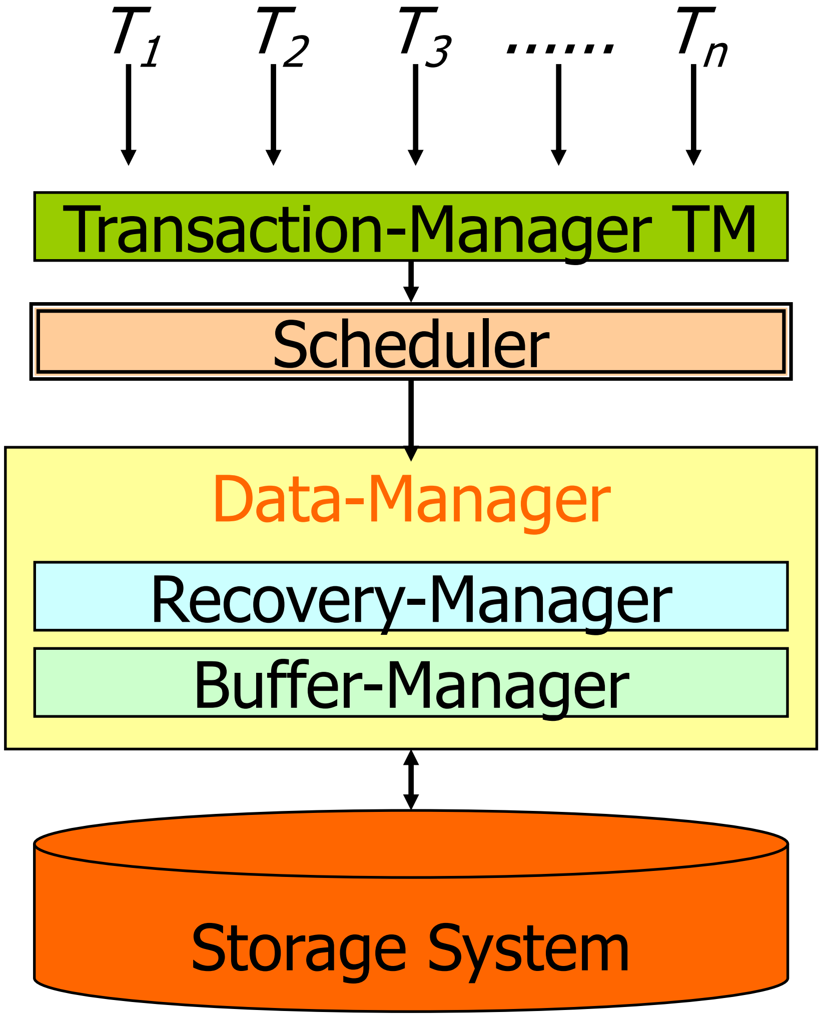
- **begin of transaction (BOT)**: Starts a new TA
- **commit**: End a TA (success).
  - Application wants to make all changes durable.
- **abort**: End a TA (failure).
  - Application wants to undo all changes.
- NB: Many APIs (e.g., JDBC) have an auto-commit option:
  - Every SQL statement run in its own TA.

SQL Example

```sql
begin;

insert into Lectures
        values (5275, `Kernphysik`, 3, 2141);
insert into Professors
        values (2141, `Meitner`, `FP`, 205);
commit;
```

Database-Scheduler



T_1 T_2 T_3 …… T_n

Transaction-Manager TM

Scheduler

Data-Manager

Recovery-Manager

Buffer-Manager

Storage System

Concurrency Anomalies

In multi-user operation following concurrency anomalies can occur:

1. Lost Update
2. Dirty Read
3. Non-Repeatable Read
4. Phantom Reads

Syntax on the following slides:

read/write(databaseItem, localVariable)

Anomaly 1: Lost Update

Lost Update:

time

Step	T1	T2
1	read(A, a1)	
2	a1 = a1 – 300	
3		read(A, a2)
4		a2 = a2 *1,03
5		write(A, a2)
6	write(A, a1)	
7	read(B, b1)	
8	b1 = b1 + 300	
9	write(B, b1)	

T1: transfer 300 € from account **A** to **B**.

T2: credit account **A** 3% interest.

Problem:

update of **T2** (line 5) overwritten by **T1** (line 6) and thereby lost.

Anomaly 2: Dirty Read

Dirty Read:

Step	T1	T2
1	read(A, a1)	
2	a1 = a1 – 300	
3	write(A, a1)	
4		read(A, a2)
5		a2 = a2 * 1,03
6		write(A, a2)
7	read(B, b1)	
8	...	
9	abort	

T1: transfer 300 € from account **A** to **B**.

T2: credit account **A** 3% interest.

Problem:

T1 is aborted, but **T2** has credited account **A** the interest in steps 5/6 - computed based on the ‚wrong' value of **A**.

Anomaly 3: Non-Repeatable Read

Non-Repeatabe Read:

Step	T1	T2
1	select distinct deptnr from emp where salary < 1000	
2		update emp set salary = salary + 10 where deptnr = 2
3	select distinct deptnr from emp where salary < 1000	

T1: list all department numbers with cheap employees (twice).

T2: grant salary increases to all employees from department number 2.

Problem:

The update of **T2** might affect the result of the query in **T1**.

Anomaly 4: Phantom Read

Phantom Read:

Step	T1	T2
1	select sum(balance) from accounts	
2		insert into accounts values (C, 1000)
3	select sum(balance) from accounts	

T1: read the sum of all account balances (twice).

T2: insert a new account with a balance of 1000 €.

Problem:

T1 computes two different sums.

Synchronization (1)

Criteria for correctness (goal):

- logical single user mode, i.e. avoiding all multi user anomalies

Criterion for correctness: “Serializability”

Parallel execution of a set of transactions is serializable, if there exists a serial execution of the same set of transactions:
- given the same data base state,
- yielding the same results as the original execution

Synchronization (2)

But: Serializability restricts parallel execution of transactions

➔ Accepting anomalies enables less hindrance of transactions
use very **carefully**!!

How to guarantee serializability?

… via locking

… via snapshotting

…

Read-Write Locking

RX locking:

- Read (R)-lock
- Write- or exclusive (X)-lock

Compatibility matrix:

	none	R	X
R	+	+	-
X	+	-	-

"+" means: lock is granted
"-" means: lock conflict

Read-Write Locking in DBMS

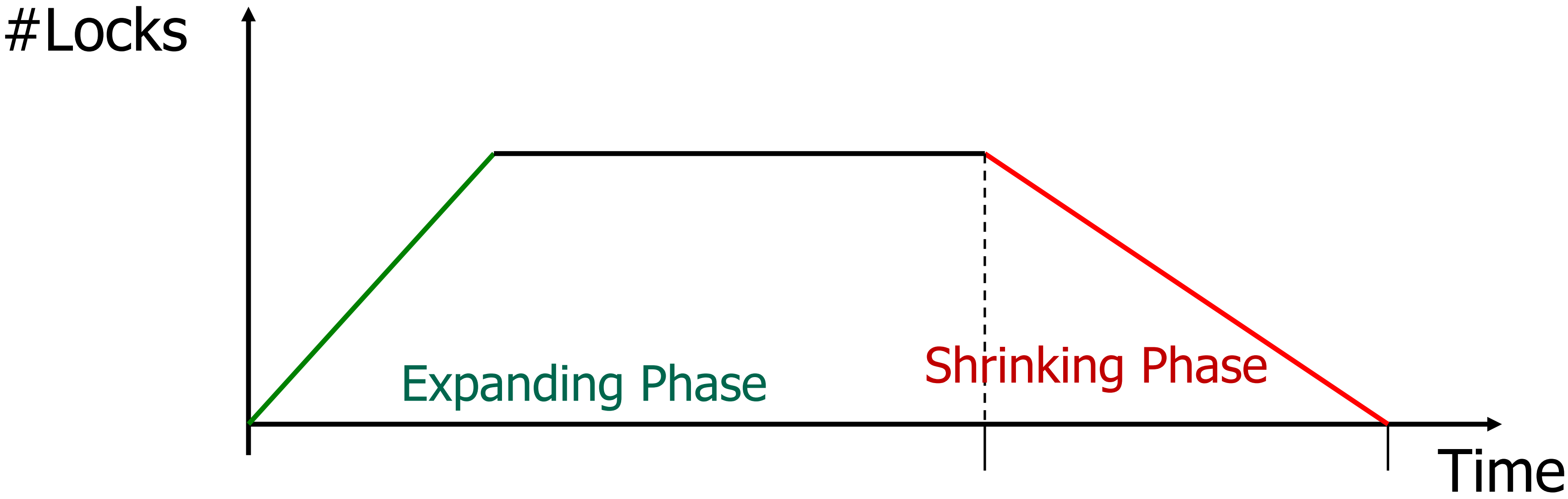
When a transaction starts …

1. Lock the entire database
2. Lock each table
3. Lock each tuple

But, how long should the lock be kept ?

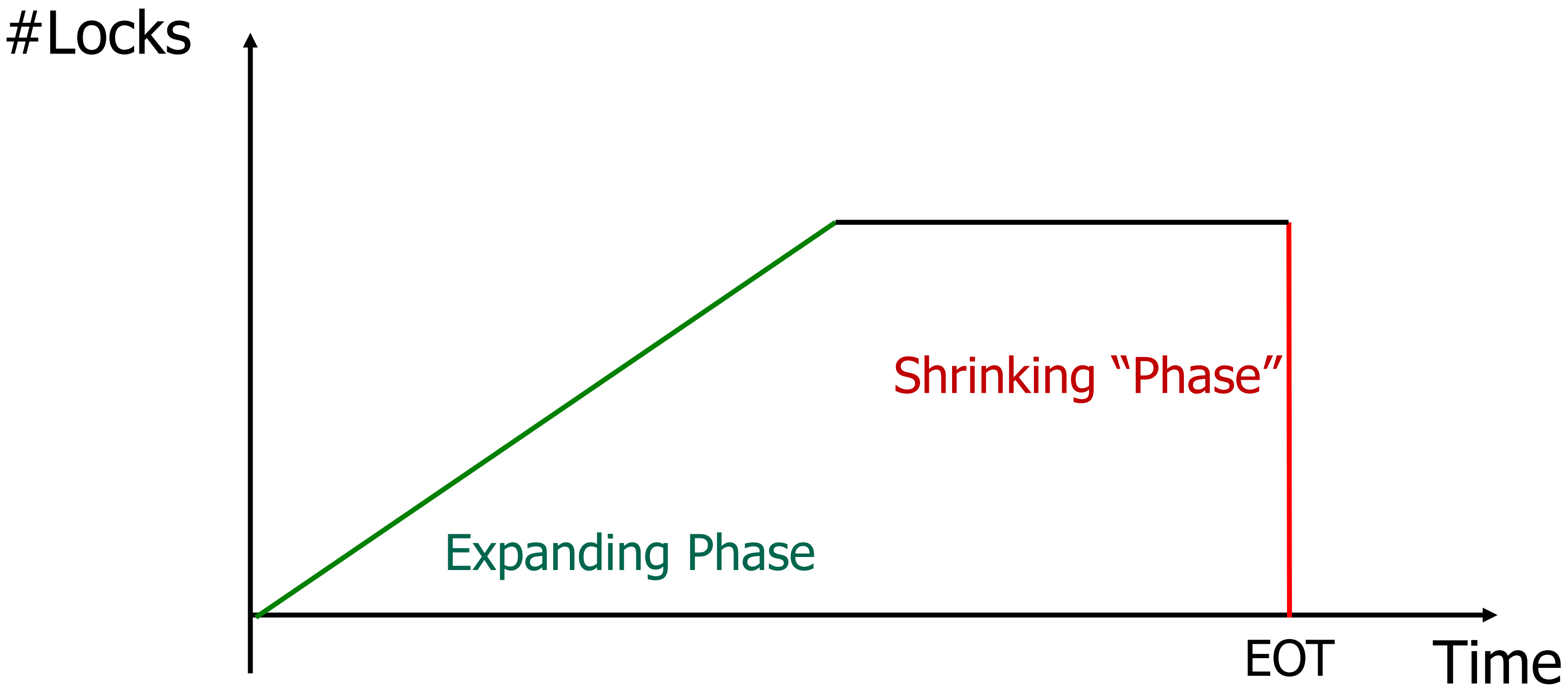
Two-Phase Locking Protocol

- Lock conflict -> requesting transaction has to wait until incompatible lock(s) is (are) removed
- Each transaction has two phases:
  - Expanding: Locks may be requested (but not released)
  - Shrinking: Locks are released (but not requested)
- Blocking and deadlocks possible



Strict 2-Phase Locking Protocol

- Keep all (write) locks until end of transaction and release atomically with commit

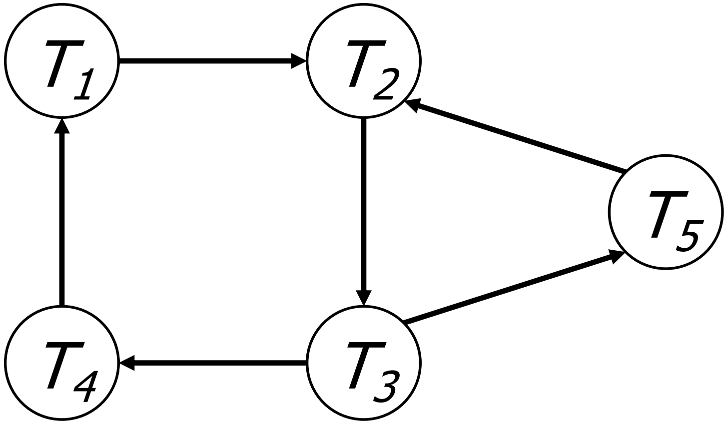


Deadlock Detection

Wait-for Graph

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$

$T_2 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$



- Abort T_3 will resolve both cycles
- Alternative: Deadlock detection with timeouts. Pros/cons?

Deadlock Handling

Incompatibility of a lock request:
→ Transaction has to wait

Deadlock detection:

Search for deadlocks in periodical time intervals (adjustable), usually done by cycle detection, resolved by abort of transaction(s)

Timeout:

Maximum time for waiting for a lock (adjustable), abort of transaction when reached

Optimizations (Further Reading)

- Hierarchical locking
- Reduced consistency level
- Multi version approach
- More lock modes

- Alternatives to locking: Optimistic concurrency control

Consistency Levels SQL

- Four consistency levels (isolation levels) determined by the anomalies which may occur
- Lost Updates are always avoided
- Default: Serializable

	Dirty Read	Non-Repeatable Read	Phantoms
Read Uncommitted	+	+	+
Read Committed	-	+	+
Repeatable Read	-	-	+
Serializable	-	-	-

Consistency levels PostgreSQL (1)

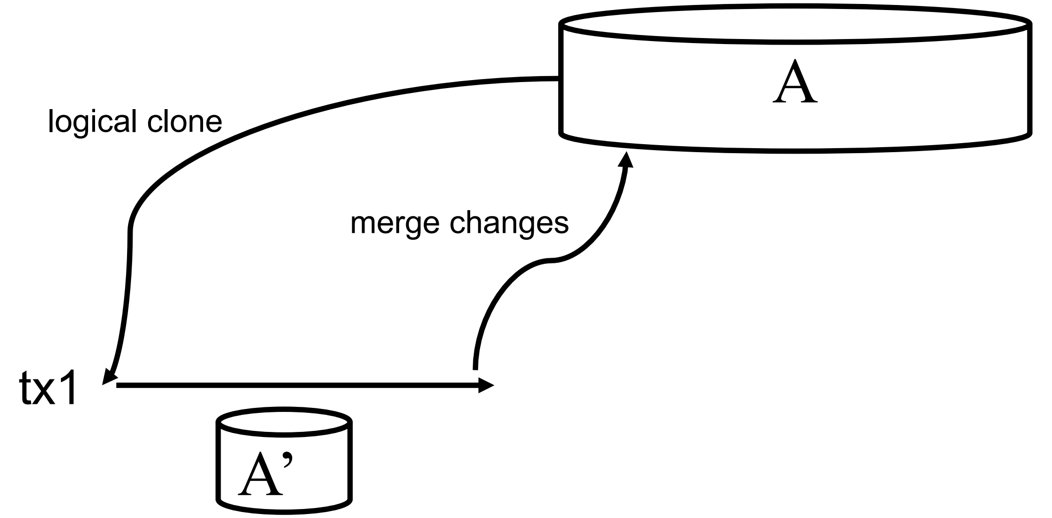
=

	Dirty Read	Non-Repeatable Read	Phantoms
Read Uncommitted	~~+~~ -	+	+
Read Committed	-	+	+
Repeatable Read	-	-	~~+~~ -
Serializable	-	-	-

No anomalies ≠ serializable !! (phantoms still possible)

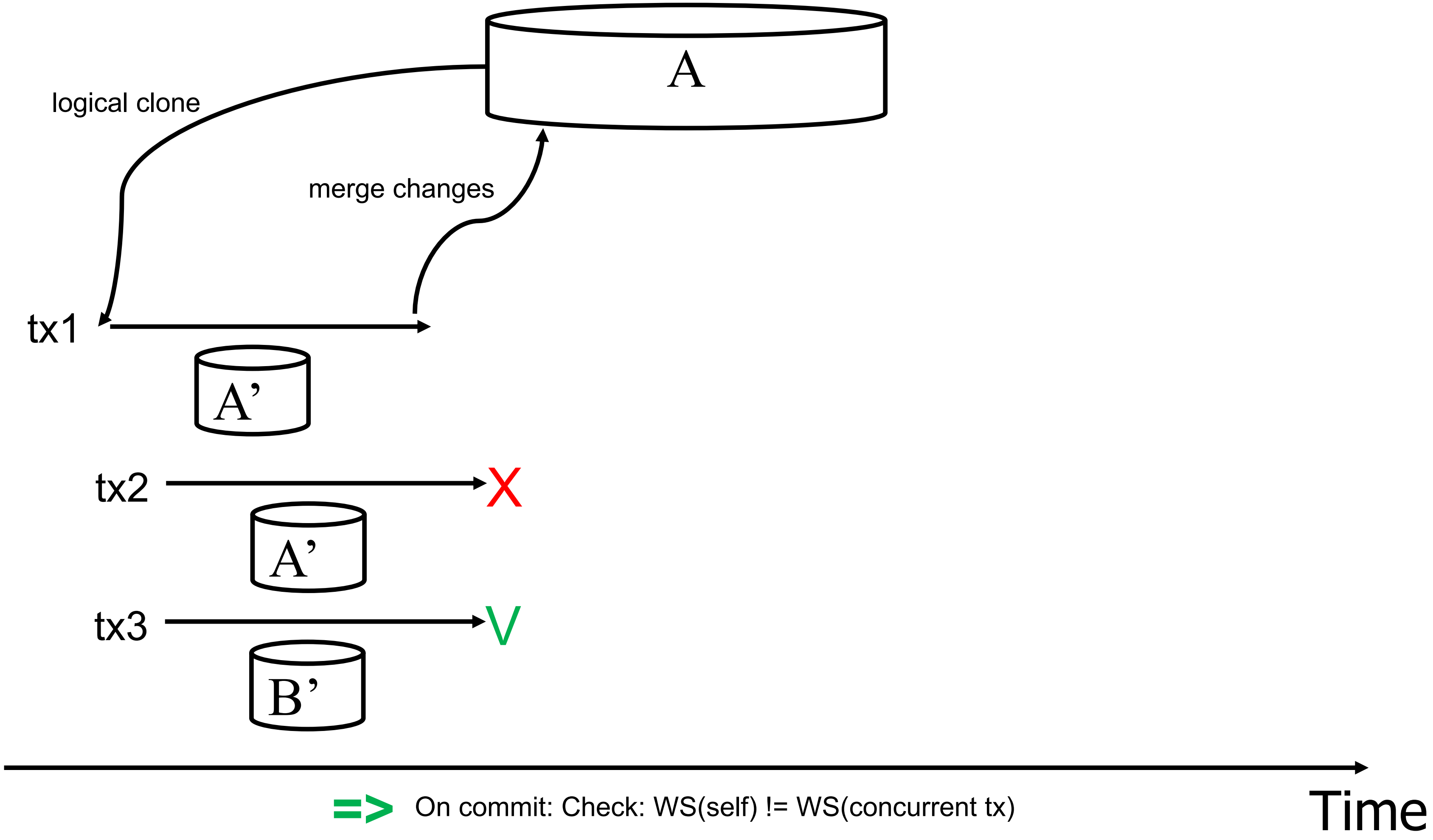
Critique: definition of anomalies stem from a synchronization method using locking

Snapshot Isolation



=> On commit: Check: WS(self) != WS(concurrent tx)

Snapshot Isolation



=> On commit: Check: WS(self) != WS(concurrent tx)

Snapshot Isolation

Each transaction sees the database in that state it was in when the transaction started

== reads the last committed values that existed at the time it started

→ All reads made in a transaction will see a consistent snapshot of the database

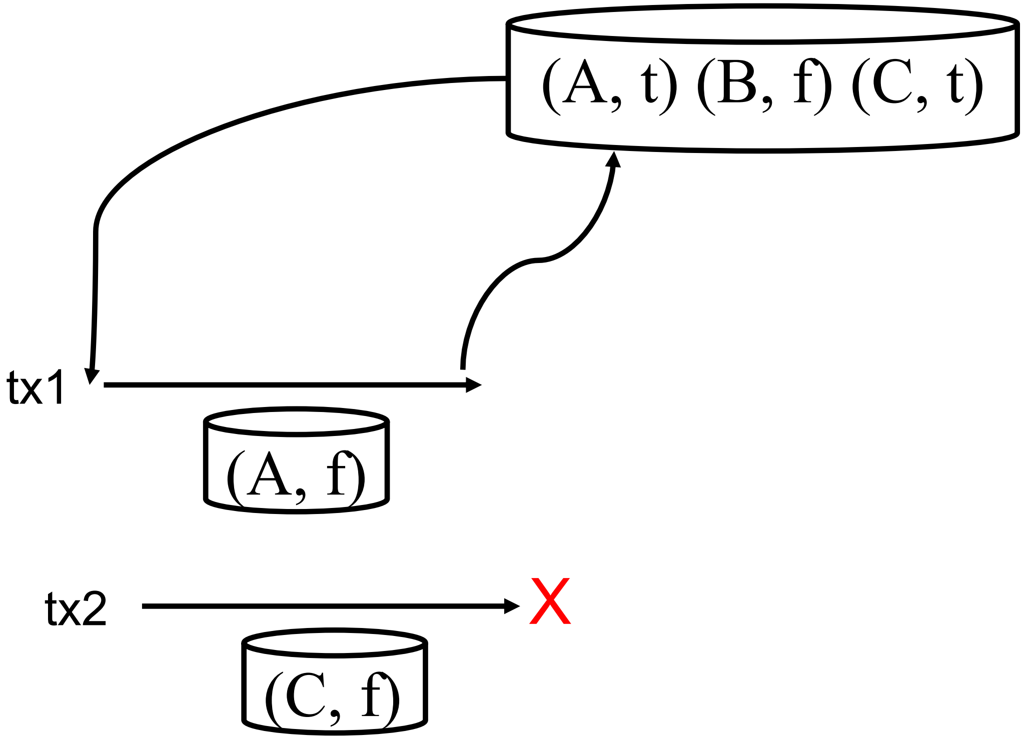
→ Transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot
→ Only write-write conflicts checked before commit

Snapshot Isolation

- Such a write-write conflict will cause the transaction to abort
- Snapshot isolation is implemented by multi-version concurrency control (MVCC)
- Advantage: no reader waits for a writer
  no writer waits for a reader
- Disadvantage: needs more space for new versions (no update in place)
  needs cleaning

→ Good if mainly read transactions

Serializable Snapshot Isolation



Invariant: Someone is there

name	is_there
A	true
B	false
C	true

```
x = select count(*)
    from Doctors
    where is_there;
if(x >= 2) {
   update Doctors
   set is_there = f;
   where name=‘%1’
}
```

Time

=> On commit: Check: WS(self) U RS(self) != WS(concurrent tx)

Serializable Snapshot Isolation

Example: write skew anomaly

T1, T2 start concurrently on the same snapshot

T1 sets V1 to V1 – 200, checks that V1+V2 >= 0

T2 sets V2 to V2 – 200, checks that V1+V2 >= 0

both finally concurrently commit

none has seen the update performed by the other

→ no serializable schedule

but no non-repeatable read anomaly!

snapshot isolation may lead to non serializable schedules
→ serializable snapshot isolation