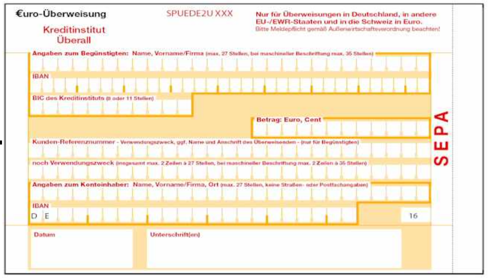
TRANSACTIONS



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/DDL statements

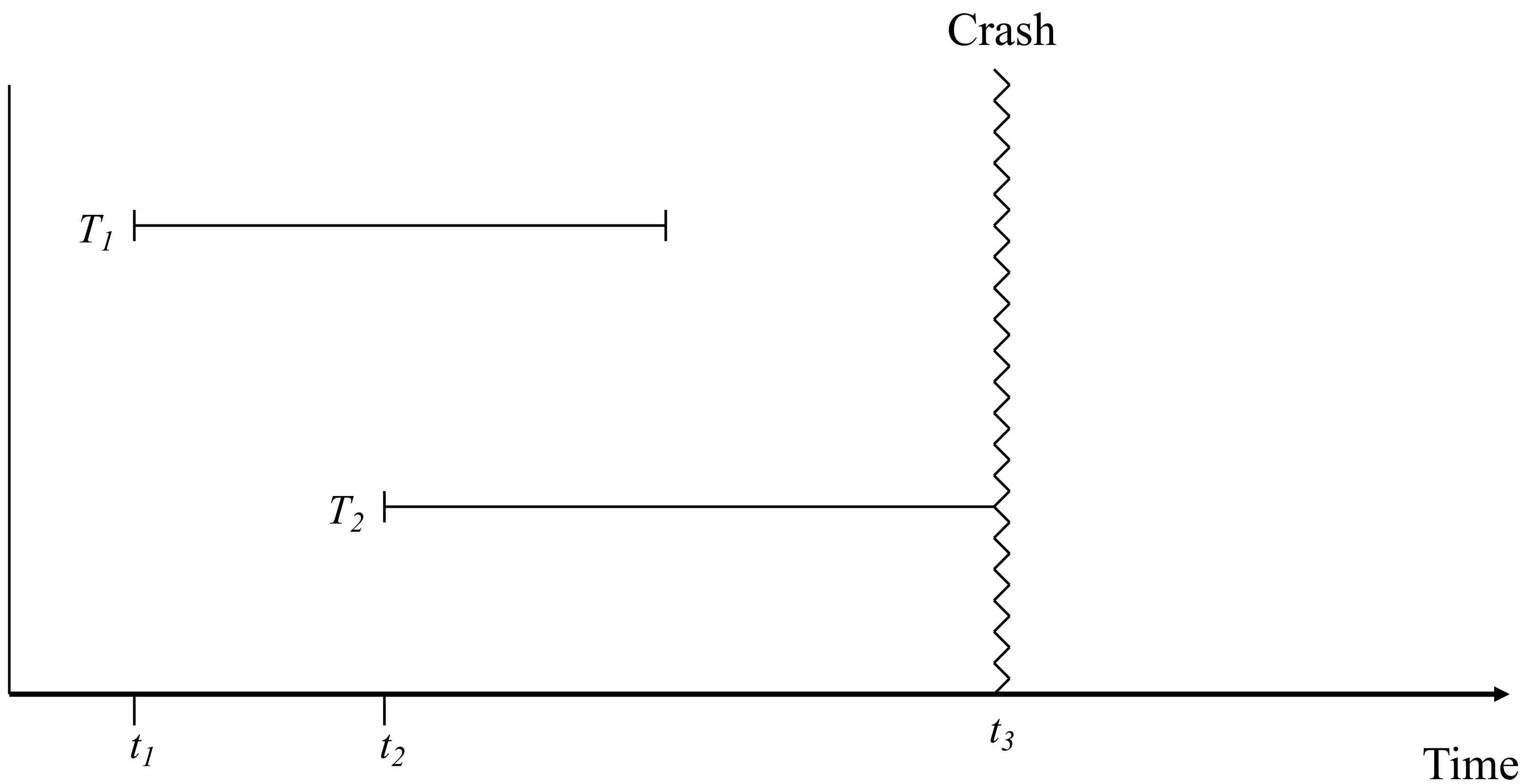
Transforms the data base from one consistent state to another consistent state

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

2. 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

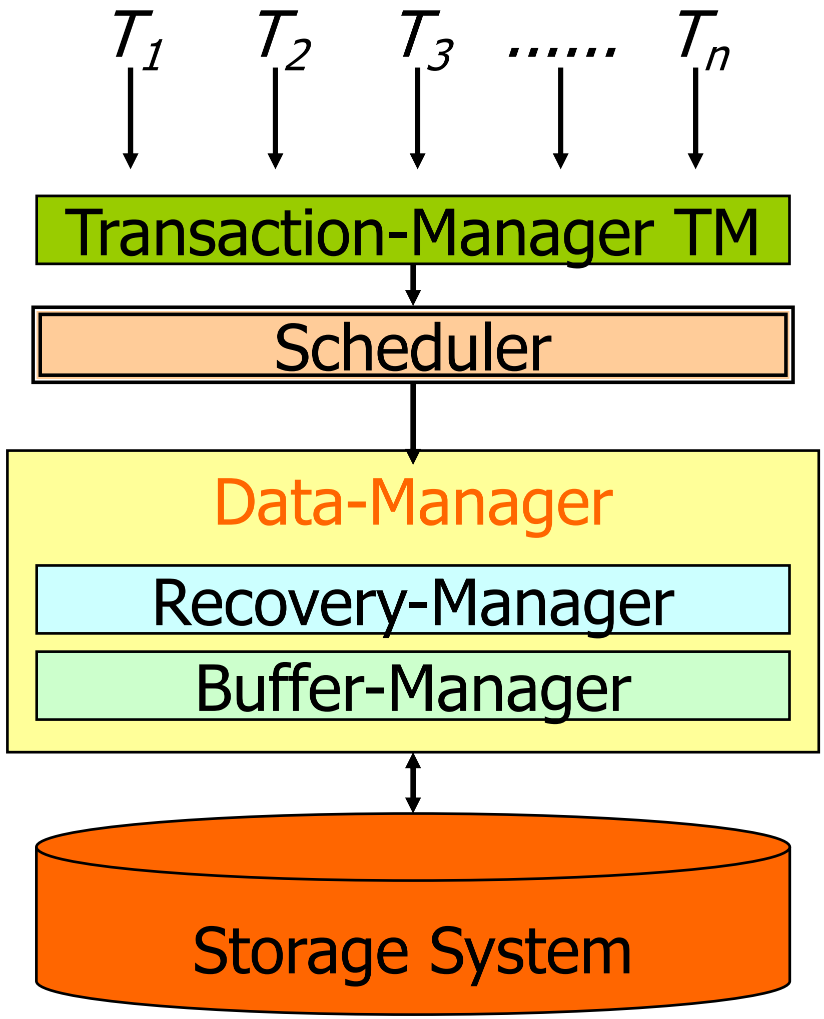
- **begin of transaction (BOT)**: Starts a new TA
- **commit**: End a TA (success).
  - Application wants to make all changes durable.
- **abort**: End a TA (failure).
  - Application wants to undo all changes.
- N.B. Many APIs (e.g., JDBC) have an auto-commit option:
  - Every SQL statement run in its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

SQL Example

```
begin;

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

Database-Scheduler



Concurrency Anomalies

In multi-user operation following concurrency anomalies can occur:

- Lost Update
- Dirty Read
- Non-Repeatable Read
- Phantom Reads

Anomalies (2)

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 transfers 300 € from account A to B.

T2 credits account A 3% interest.

Interesting steps: 5 and 6

update of TA 2 without (again) reading A overwritten and thereby lost.

Anomalies (3)

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 transfers 300 € from account A to B.

T2 credits account A 3% interest.

Interesting steps: 4 and 9

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

Anomalies (4)

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 lists (twice) all department numbers where there exists an employee with a salary less than 1000.

T2 grants salary increases to all employees from department number 2.

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

Anomalies (5)

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 reads twice the sum of all account balances.

T2 **inserts** a new account with a balance of 1000 €.

T1 computes two different sums.

Synchronization (1)

Criterion for correctness (goal):

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

Formal criterion for correctness :
Serializability:

Parallel execution of a set of transactions is serializable, if there exists ***one*** serial execution of the same set of transactions, yielding the
- same data base state and
- 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

…

Locking (1)

Example: RX-locking (simple)

Two lock modes:

Read (R)-lock

Write- or exclusive (X)-lock

Compatibility matrix:

	none	R	X
R	+	+	-
X	+	-	-

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

Locking (2)

- With lock conflict requesting transaction has to wait until incompatible lock(s) is (are) removed
- Blocking and deadlocks possible
- Locks are potentially held until end of transaction

Possible optimizations:

- Hierarchical locking
- Reduced consistency level
- Multi version approach

Locking (3)

Incompatibility of a lock request:
→ Transaction has to wait

Deadlock:

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

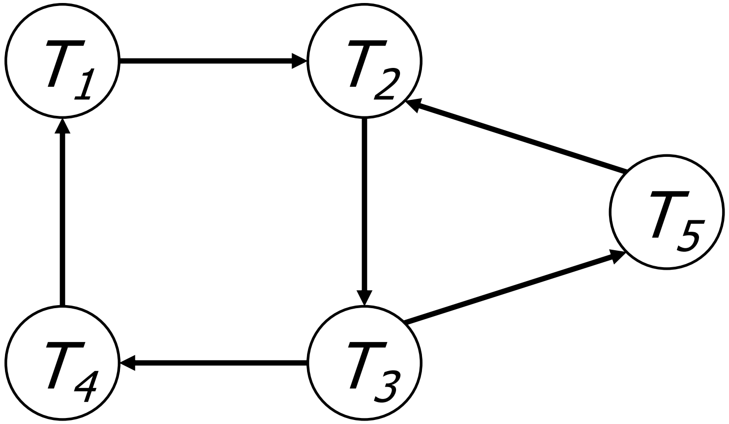
Timeout: Maximum time for waiting for a lock (adjustable), abort of transaction when reached

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?

Consistency levels SQL

Four Consistency levels (isolation levels)

determined by the anomalies which may occur

Lost Update always avoided: write locks until end of transaction

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

Multi-Version Concurrency Control in PostgreSQL (1)

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

Multi-version concurrency control in PostgreSQL (2)

- 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

Multi-version concurrency control in PostgreSQL (3)

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