

# Session 14 – Transaktionen & ACID

**Session-Typ:** Vorlesung **Dauer:** 90 Minuten **Lernziele:** ACID verstehen, Transaktionssteuerung anwenden, Isolation Levels vergleichen

Willkommen zu Session 14! Heute geht es um eines der fundamentalsten Konzepte relationaler Datenbanksysteme: Transaktionen und ACID. Wir haben bisher viel über SQL-Abfragen, Modellierung und komplexe Queries gelernt – aber was passiert, wenn mehrere Nutzer gleichzeitig auf dieselben Daten zugreifen? Wie garantieren wir Konsistenz bei Systemausfällen? Diese Fragen beantworten Transaktionen.

## Motivation: Warum Transaktionen?

### Szenario: Geldüberweisung zwischen Konten

Stellen Sie sich vor, Sie überweisen 100 Euro von Konto A nach Konto B. Das klingt simpel, aber technisch sind das zwei separate Operationen: Erst wird Konto A belastet, dann Konto B gutgeschrieben. Was passiert, wenn zwischen diesen beiden Schritten der Server abstürzt? Oder wenn eine andere Transaktion genau in diesem Moment auf Konto A zugreift?

#### Ohne Transaktionen: Probleme

```
-- Schritt 1: 100 Euro von Konto A abziehen
UPDATE accounts SET balance = balance - 100 WHERE id = 'A';

-- ✗ Server-Crash hier!

-- Schritt 2: 100 Euro auf Konto B gutschreiben (wird nie ausgeführt)
UPDATE accounts SET balance = balance + 100 WHERE id = 'B';
```

**Ergebnis:** 100 Euro sind verschwunden! Konto A ist belastet, aber Konto B wurde nie gutgeschrieben.

Genau solche Inkonsistenzen verhindern Transaktionen. Eine Transaktion ist eine logische Arbeitseinheit, die garantiert, dass entweder alle Operationen erfolgreich durchgeführt werden – oder gar keine. Das ist das „Alles-oder-Nichts“-Prinzip.

#### Mit Transaktionen: Atomar & Sicher

```
BEGIN TRANSACTION;

-- Schritt 1: 100 Euro von Konto A abziehen
UPDATE accounts SET balance = balance - 100 WHERE id = 'A';

-- Schritt 2: 100 Euro auf Konto B gutschreiben
UPDATE accounts SET balance = balance + 100 WHERE id = 'B';
```





`COMMIT`; -- Erst jetzt werden beide Änderungen dauerhaft gespeichert

**Garantie:** Wenn `COMMIT` erfolgreich ist, sind beide Änderungen persistent. Bei einem Fehler vor `COMMIT` wird automatisch `ROLLBACK` ausgeführt – keine Änderung bleibt bestehen.

Eine Transaktion ist also ein Paket mit Garantiesiegel: Entweder kommt alles an – oder gar nichts. Damit sind wir schon beim ersten Buchstaben von ACID.

## ACID-Eigenschaften

ACID ist ein Akronym für vier fundamentale Eigenschaften, die jede Datenbanktransaktion erfüllen sollte: Atomicity, Consistency, Isolation und Durability. Diese Eigenschaften wurden in den 1980ern von Jim Gray definiert und sind bis heute der Goldstandard für transaktionale Systeme.

			
Atomicity	Consistency	Isolation	Durability

← ACID Database →

### A – Atomicity (Atomarität)

Atomarität bedeutet: Eine Transaktion ist eine unteilbare Einheit. Entweder werden alle Operationen ausgeführt – oder keine. Es gibt keine Zwischenzustände, die nach außen sichtbar sind.

**Metapher:** Wie ein Atom (griech. „átomos“ = unteilbar) ist eine Transaktion eine Einheit, die nicht weiter zerlegbar ist.

**Beispiel:**

Aktion	Ohne Atomarität	Mit Atomarität
UPDATE accounts A	✓ Erfolg	✓ Erfolg
✗ Server-Crash	💥 Inkonsistenter Zustand	✓ Automatisches ROLLBACK
UPDATE accounts B	✗ Wird nie ausgeführt	✗ Beide Updates rückgängig

### C – Consistency (Konsistenz)

Konsistenz bedeutet: Eine Transaktion überführt die Datenbank von einem gültigen Zustand in einen anderen gültigen Zustand. Alle Constraints, Trigger und Integritätsbedingungen werden eingehalten – vor und nach der Transaktion.

Beispiel:

```
-- Constraint: Balance darf nie negativ werden
ALTER TABLE accounts ADD CONSTRAINT balance_positive CHECK (balance >= 0);

BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 1000 WHERE id = 'A';
-- ✗ Fehler: Constraint verletzt → automatisches ROLLBACK
COMMIT; -- wird nie erreicht
```

**Garantie:** Die Datenbank bleibt in einem konsistenten Zustand – Constraints werden *immer* durchgesetzt.

## I – Isolation

Isolation bedeutet: Parallel laufende Transaktionen beeinflussen sich nicht gegenseitig. Jede Transaktion hat die Illusion, als wäre sie allein auf der Datenbank. Wie stark diese Isolation ist, können wir über Isolation Levels steuern – dazu gleich mehr.

Beispiel: Ticketbuchung

Zeitpunkt	Nutzer A	Nutzer B
T1	<div>SELECT * FROM tickets</div> <div>WHERE seat = '12A'</div>	
T2		<div>SELECT * FROM tickets</div> <div>WHERE seat = '12A'</div>
T3	<div>UPDATE tickets SET</div> <div>reserved = true</div> <div>WHERE seat = '12A'</div>	
T4		<div>UPDATE tickets SET</div> <div>reserved = true</div> <div>WHERE seat = '12A'</div>

**Ohne Isolation:** Beide sehen Sitz 12A als frei → Doppelbuchung!

**Mit Isolation:** Nutzer B muss warten, bis Nutzer A seine Transaktion abgeschlossen hat.

## D – Durability (Dauerhaftigkeit)

Dauerhaftigkeit bedeutet: Sobald eine Transaktion mit COMMIT bestätigt wurde, sind die Änderungen dauerhaft gespeichert – selbst wenn direkt danach ein Stromausfall oder Server-Crash passiert.

#### Technische Umsetzung:

- **Write-Ahead Log (WAL):** Änderungen werden zuerst in ein Log geschrieben (auf Festplatte), bevor die Datenbank-Seiten aktualisiert werden.
- **Crash Recovery:** Nach einem Neustart liest die Datenbank das WAL und stellt den Zustand wieder her.

**Garantie:** Nach `COMMIT` geht keine Änderung verloren – auch bei Hardware-Ausfällen.

Diese vier Eigenschaften zusammen machen Transaktionen zum Rückgrat relationaler Datenbanksysteme. Aber wie steuern wir Transaktionen konkret in SQL?

## Transaktionssteuerung in SQL

### Basic Commands

In SQL steuern wir Transaktionen mit vier grundlegenden Befehlen: BEGIN zum Starten, COMMIT zum Bestätigen, ROLLBACK zum Rückgängigmachen und SAVEPOINT für partielle Rollbacks.

`BEGIN` / `START TRANSACTION`

Startet eine neue Transaktion. Ab jetzt werden alle Änderungen zunächst nur temporär gespeichert.

```
BEGIN TRANSACTION;  
-- oder: START TRANSACTION;
```



`COMMIT`

Bestätigt alle Änderungen seit BEGIN. Ab jetzt sind sie dauerhaft und für andere sichtbar.

```
COMMIT;
```



`ROLLBACK`

Macht alle Änderungen seit BEGIN rückgängig. Die Datenbank kehrt zum Zustand vor BEGIN zurück.

```
ROLLBACK;
```



`SAVEPOINT`

Setzt einen Zwischenpunkt innerhalb einer Transaktion. Erlaubt partielle Rollbacks.

```
BEGIN;  
UPDATE accounts SET balance = balance - 100 WHERE id = 'A';  
  
SAVEPOINT transfer_step1;
```



```
UPDATE accounts SET balance = balance + 100 WHERE id = 'B';  
-- Fehler! Konto B existiert nicht
```

```
ROLLBACK TO transfer_step1; -- Nur Schritt 2 rückgängig, Schritt 1 bleibt  
COMMIT;
```

## Live-Demo: Geldüberweisung

Schauen wir uns das Ganze in Aktion an. Ich starte mit einer einfachen Konten-Tabelle und zeige, was mit und ohne Transaktion passiert.

```
1 CREATE TABLE accounts (  
2     id TEXT PRIMARY KEY,  
3     owner TEXT,  
4     balance INTEGER CHECK (balance >= 0)  
5 );  
6  
7 INSERT INTO accounts VALUES  
8     ('A', 'Alice', 500),  
9     ('B', 'Bob', 300);  
10  
11 SELECT * FROM accounts;
```

```
CREATE TABLE accounts (  
  id TEXT PRIMARY KEY,  
  owner TEXT,  
  balance INTEGER CHECK (balance >= 0)  
)
```

ok

```
INSERT INTO accounts VALUES  
  ('A', 'Alice', 500),  
  ('B', 'Bob', 300)
```

ok

```
SELECT * FROM accounts
```

#	id	owner	balance
1	A	Alice	500
2	B	Bob	300

2 rows

Jetzt führen wir eine Überweisung ohne Transaktion durch – und simulieren einen Fehler nach dem ersten UPDATE.

```
1  -- Ohne Transaktion: Gefährlich!
2  UPDATE accounts SET balance = balance - 100 WHERE id = 'A';
3  -- ✗ Fehler: System-Crash simuliert
4  -- UPDATE accounts SET balance = balance + 100 WHERE id = 'B';
5
6  SELECT * FROM accounts;
7  -- Ergebnis: A hat 400 Euro, B hat 300 Euro → 100 Euro verschwunden!
```

```
-- Ohne Transaktion: Gefährlich!
UPDATE accounts SET balance = balance - 100 WHERE id = 'A'
```

ok

```
-- ✗ Fehler: System-Crash simuliert
-- UPDATE accounts SET balance = balance + 100 WHERE id = 'B';
```

```
SELECT * FROM accounts
```

#	id	owner	balance
1	B	Bob	300
2	A	Alice	400

2 rows

```
-- Ergebnis: A hat 400 Euro, B hat 300 Euro → 100 Euro verschwunden!
```

ok

Und jetzt dasselbe mit Transaktion. Wenn ein Fehler auftritt, wird automatisch ein ROLLBACK durchgeführt.

```
1  -- Reset
2  UPDATE accounts SET balance = 500 WHERE id = 'A';
3
4  BEGIN TRANSACTION;
5
6  UPDATE accounts SET balance = balance - 100 WHERE id = 'A';
7
8  -- Fehler simulieren (ungültige Constraint-Verletzung)
9  UPDATE accounts SET balance = balance + 100 WHERE id = 'Z'; -- Konto
   existiert nicht
10
11 ROLLBACK; -- Manuell rückgängig gemacht
12
```

```
13 SELECT * FROM accounts;  
14 -- Ergebnis: A hat 500 Euro, B hat 300 Euro → Alles wie vorher!
```

```
-- Reset  
UPDATE accounts SET balance = 500 WHERE id = 'A'
```

ok

```
BEGIN TRANSACTION
```

ok

```
UPDATE accounts SET balance = balance - 100 WHERE id = 'A'
```

ok

```
-- Fehler simulieren (ungültige Constraint-Verletzung)  
UPDATE accounts SET balance = balance + 100 WHERE id = 'Z'
```

ok

```
-- Konto existiert nicht
```

```
ROLLBACK
```

ok

```
-- Manuell rückgängig gemacht
```

```
SELECT * FROM accounts
```

#	id	owner	balance
1	B	Bob	300
2	A	Alice	500

2 rows

```
-- Ergebnis: A hat 500 Euro, B hat 300 Euro → Alles wie vorher!
```

ok

Perfekt! Mit Transaktionen haben wir Atomarität garantiert. Aber was passiert, wenn mehrere Transaktionen parallel laufen? Hier kommen Isolation Levels ins Spiel.

# Isolation Levels

Zeit	Transaction A	Transaction B
T1	<div><pre>SELECT balance FROM accounts WHERE id = 'A'</pre><div>Ergebnis → 100</div></div>	<div><pre>SELECT balance FROM accounts WHERE id = 'A'</pre></div>

## Probleme bei parallelen Transaktionen

Isolation ist die komplizierteste der vier ACID-Eigenschaften. Warum? Weil perfekte Isolation extrem teuer ist – sie würde bedeuten, dass immer nur eine Transaktion gleichzeitig laufen darf. Deshalb gibt es verschiedene Isolation Levels, die einen Trade-off zwischen Konsistenz und Performance erlauben.

### Welche Probleme können auftreten?

Wenn Transaktionen parallel laufen, gibt es vier klassische Anomalien:

#### 1. Dirty Read (Schmutziges Lesen)

Transaktion A liest Daten, die von Transaktion B geändert, aber noch nicht committed wurden.

Zeit	Transaktion A	Transaktion B
T1		<div><pre>UPDATE accounts SET balance = 1000 WHERE id = 'A'</pre></div>
T2	<div><pre>SELECT balance FROM accounts WHERE id = 'A'</pre><div>→ Ergebnis: 1000</div></div>	
T3		<div><pre>ROLLBACK;</pre></div>
T4	<div>– A hat 1000 gelesen, aber das war nie committed!</div>	

**Problem:** A hat einen Wert gelesen, der nie existiert hat.

#### 2. Non-Repeatable Read (Nicht-wiederholbares Lesen)



Transaktion A liest denselben Datensatz zweimal und bekommt unterschiedliche Werte.

Zeit	Transaktion A	Transaktion B
T1	<pre>SELECT balance FROM accounts WHERE id = 'A'</pre> <p>→ Ergebnis: 500</p>	
T2		<pre>UPDATE accounts SET balance = 1000 WHERE id = 'A'; COMMIT;</pre>
T3	<pre>SELECT balance FROM accounts WHERE id = 'A'</pre> <p>→ Ergebnis: 1000</p>	

**Problem:** A liest zweimal – und bekommt unterschiedliche Ergebnisse innerhalb derselben Transaktion.

### 3. Phantom Read (Phantom-Lesen)

Transaktion A führt dieselbe Abfrage zweimal aus und findet beim zweiten Mal zusätzliche Zeilen.

Zeit	Transaktion A	Transaktion B
T1	<pre>SELECT * FROM tickets WHERE reserved = false</pre> <p>→ Ergebnis: 5 Tickets</p>	
T2		<pre>INSERT INTO tickets VALUES ('12F', false); COMMIT;</pre>
T3	<pre>SELECT * FROM tickets WHERE reserved = false</pre> <p>→ Ergebnis: 6 Tickets</p>	

**Problem:** Plötzlich sind neue Zeilen aufgetaucht – wie ein Phantom.

### 4. Lost Update (Verlorenes Update)

Zwei Transaktionen lesen denselben Wert, ändern ihn parallel – und eine Änderung geht verloren.

Zeit	Transaktion A	Transaktion B
T1	<pre>SELECT balance FROM accounts WHERE id = 'A'</pre> → Ergebnis: 500	<pre>SELECT balance FROM accounts WHERE id = 'A'</pre> → Ergebnis: 500
T2	balance = 500 - 100 = 400	balance = 500 + 200 = 700
T3	<pre>UPDATE accounts SET balance = 400 WHERE id = 'A'; COMMIT;</pre>	
T4		<pre>UPDATE accounts SET balance = 700 WHERE id = 'A'; COMMIT;</pre>

**Problem:** A's Update (400) wurde von B's Update (700) überschrieben. Die -100 sind verloren!

## Die vier Isolation Levels

Um diese Probleme zu adressieren, definiert der SQL-Standard vier Isolation Levels – von schwach (schnell, aber unsicher) bis stark (sicher, aber langsam).

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update
READ UNCOMMITTED	⚠️ Möglich	⚠️ Möglich	⚠️ Möglich	⚠️ Möglich
READ COMMITTED	✅ Verhindert	✅ Verhindert	⚠️ Möglich	⚠️ Möglich
REPEATABLE READ	✅ Verhindert	✅ Verhindert	⚠️ Möglich	✅ Verhindert
SERIALIZABLE	✅ Verhindert	✅ Verhindert	✅ Verhindert	✅ Verhindert

Standard in PostgreSQL: **READ COMMITTED**

Standard in MySQL: **REPEATABLE READ**

Wie setzen wir das in SQL? Mit dem SET TRANSACTION Befehl.

```
-- Isolation Level für die nächste Transaktion setzen
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;
-- Alle Operationen hier laufen mit SERIALIZABLE Isolation
COMMIT;
```

In der Praxis reicht **READ COMMITTED** für die meisten Anwendungen. Nur bei kritischen Operationen wie Ticketbuchungen oder Finanztransaktionen brauchen wir stärkere Isolation.

## Praktische Beispiele

### Szenario 1: Ticketbuchung

Schauen wir uns ein klassisches Concurrency-Problem an: Zwei Nutzer wollen gleichzeitig denselben Sitzplatz buchen.

```
1 CREATE TABLE tickets (
2     seat TEXT PRIMARY KEY,
3     reserved BOOLEAN DEFAULT false
4 );
```

```
5  ',
6  INSERT INTO tickets VALUES ('12A', false), ('12B', false), ('12C', fa
7
8  -- Transaktion 1 (Nutzer Alice)
9  BEGIN TRANSACTION;
10 SELECT * FROM tickets WHERE seat = '12A' AND reserved = false;
11 -- Ergebnis: Sitz ist frei
12
13 -- 🌟 Gleichzeitig startet Transaktion 2 (Nutzer Bob)
14 -- BEGIN TRANSACTION;
15 -- SELECT * FROM tickets WHERE seat = '12A' AND reserved = false;
16 -- Ergebnis: Sitz ist frei (falls READ COMMITTED)
17
18 UPDATE tickets SET reserved = true WHERE seat = '12A';
19 COMMIT;
20
21 -- Transaktion 2 würde jetzt ebenfalls versuchen:
22 -- UPDATE tickets SET reserved = true WHERE seat = '12A';
23 -- ❌ Mit SERIALIZABLE: Fehler oder Warten
24 -- ⚠️ Mit READ COMMITTED: Überschreibt still (Doppelbuchung!)
```

```
CREATE TABLE tickets (  
  seat TEXT PRIMARY KEY,  
  reserved BOOLEAN DEFAULT false  
)
```

ok

```
INSERT INTO tickets VALUES ('12A', false), ('12B', false), ('12C', false)
```

ok

```
-- Transaktion 1 (Nutzer Alice)  
BEGIN TRANSACTION
```

ok

```
SELECT * FROM tickets WHERE seat = '12A' AND reserved = false
```

#	seat	reserved
1	12A	false

1 rows

```
-- Ergebnis: Sitz ist frei
```

```
-- ✨ Gleichzeitig startet Transaktion 2 (Nutzer Bob)  
-- BEGIN TRANSACTION;  
-- SELECT * FROM tickets WHERE seat = '12A' AND reserved = false;  
-- Ergebnis: Sitz ist frei (falls READ COMMITTED)
```

```
UPDATE tickets SET reserved = true WHERE seat = '12A'
```

ok

```
COMMIT
```

ok

```
-- Transaktion 2 würde jetzt ebenfalls versuchen:  
-- UPDATE tickets SET reserved = true WHERE seat = '12A';  
-- ❌ Mit SERIALIZABLE: Fehler oder Warten  
-- ⚠️ Mit READ COMMITTED: Überschreibt still (Doppelbuchung!)
```

ok

Lösung: Verwende SERIALIZABLE oder SELECT FOR UPDATE, um den Sitz zu locken.

```

1  -- Bessere Variante: SELECT FOR UPDATE
2  BEGIN TRANSACTION;
3  SELECT * FROM tickets WHERE seat = '12A' AND reserved = false FOR UPDATE
4  -- Sperrt die Zeile → andere Transaktionen müssen warten
5
6  UPDATE tickets SET reserved = true WHERE seat = '12A';
7  COMMIT;

```

```

-- Bessere Variante: SELECT FOR UPDATE
BEGIN TRANSACTION

```

ok

```
SELECT * FROM tickets WHERE seat = '12A' AND reserved = false FOR UPDATE
```

#	seat	reserved
---	------	----------

0 rows

```
-- Sperrt die Zeile → andere Transaktionen müssen warten
```

```
UPDATE tickets SET reserved = true WHERE seat = '12A'
```

ok

```
COMMIT
```

ok

## Szenario 2: Inventarverwaltung

Ein weiteres Beispiel: Ein Online-Shop aktualisiert den Lagerbestand, während parallel eine Bestellung aufgegeben wird.

```

1  CREATE TABLE inventory (
2      product_id TEXT PRIMARY KEY,
3      stock INTEGER CHECK (stock >= 0)
4  );
5
6  INSERT INTO inventory VALUES ('laptop_123', 5);
7
8  -- Transaktion 1: Kunde kauft 2 Laptops
9  BEGIN TRANSACTION;
10 UPDATE inventory SET stock = stock - 2 WHERE product_id = 'laptop_123';
11 COMMIT;
12
13 -- Transaktion 2: Lieferung kommt (3 neue Laptops)
14 BEGIN TRANSACTION;

```

```
14 BEGIN TRANSACTION;
15 UPDATE inventory SET stock = stock + 3 WHERE product_id = 'laptop_123';
16 COMMIT;
17
18 SELECT * FROM inventory;
19 -- Ergebnis: stock = 6 (5 - 2 + 3)
```

```
CREATE TABLE inventory (  
  product_id TEXT PRIMARY KEY,  
  stock INTEGER CHECK (stock >= 0)  
)
```

ok

```
INSERT INTO inventory VALUES ('laptop_123', 5)
```

ok

```
-- Transaktion 1: Kunde kauft 2 Laptops  
BEGIN TRANSACTION
```

ok

```
UPDATE inventory SET stock = stock - 2 WHERE product_id = 'laptop_123'
```

ok

```
COMMIT
```

ok

```
-- Transaktion 2: Lieferung kommt (3 neue Laptops)  
BEGIN TRANSACTION
```

ok

```
UPDATE inventory SET stock = stock + 3 WHERE product_id = 'laptop_123'
```

ok

```
COMMIT
```

ok

```
SELECT * FROM inventory
```

#	product_id	stock
1	laptop_123	6

1 rows

```
-- Ergebnis: stock = 6 (5 - 2 + 3)
```



ok

Hier ist READ COMMITTED ausreichend, da beide Transaktionen unabhängig sind – keine Konflikte.

## Deadlocks

Ein Deadlock entsteht, wenn zwei Transaktionen gegenseitig aufeinander warten. Klassisches Beispiel: Transaktion A sperrt Zeile 1 und will Zeile 2, während Transaktion B Zeile 2 sperrt und Zeile 1 will.

### Was ist ein Deadlock?

Zeit	Transaktion A	Transaktion B
T1	<div>UPDATE accounts</div> <div>SET balance = balance - 100</div> <div>WHERE id = 'A'</div> <div>→ Sperrt Zeile A</div>	<div>UPDATE accounts</div> <div>SET balance = balance - 50</div> <div>WHERE id = 'B'</div> <div>→ Sperrt Zeile B</div>
T2	<div>UPDATE accounts</div> <div>SET balance = balance + 100</div> <div>WHERE id = 'B'</div> <div>⌚ Wartet auf Lock von B</div>	<div>UPDATE accounts</div> <div>SET balance = balance + 50</div> <div>WHERE id = 'A'</div> <div>⌚ Wartet auf Lock von A</div>
T3	💀 Deadlock!	💀 Deadlock!

Beide warten ewig aufeinander.

Die Datenbank erkennt Deadlocks automatisch (über einen Deadlock Detector) und bricht eine der Transaktionen ab.

### Deadlock-Erkennung

```
ERROR: deadlock detected
DETAIL: Process 1234 waits for ShareLock on transaction 5678;
        blocked by process 5678.
HINT: See server log for query details.
```



Eine Transaktion wird automatisch mit ROLLBACK abgebrochen, die andere kann fortfahren.

Wie vermeiden wir Deadlocks? Konsistente Lock-Reihenfolge!

### Deadlock-Vermeidung

Falsch (kann Deadlock verursachen):

```
-- Transaktion A
UPDATE accounts SET ... WHERE id = 'A';
UPDATE accounts SET ... WHERE id = 'B';

-- Transaktion B
UPDATE accounts SET ... WHERE id = 'B';
UPDATE accounts SET ... WHERE id = 'A';
```

Richtig (immer alphabetische Reihenfolge):

```
-- Transaktion A
UPDATE accounts SET ... WHERE id = 'A';
UPDATE accounts SET ... WHERE id = 'B';

-- Transaktion B
UPDATE accounts SET ... WHERE id = 'A'; -- Wartet auf A
UPDATE accounts SET ... WHERE id = 'B';
```

Regel: Immer Ressourcen in derselben Reihenfolge sperren.

## Best Practices

Zum Abschluss noch ein paar praktische Tipps für den Umgang mit Transaktionen.

### 1. Transaktionen kurz halten

Warum? Lange Transaktionen sperren Ressourcen → andere müssen warten → Performance leidet.

Falsch:

```
BEGIN;
SELECT * FROM orders WHERE status = 'pending'; -- 10.000 Zeilen
-- Jetzt 5 Minuten warten, während Nutzer Eingaben macht...
UPDATE orders SET status = 'processed' WHERE id = 123;
COMMIT;
```

Richtig:

```
-- Lesen außerhalb der Transaktion
SELECT * FROM orders WHERE status = 'pending';

-- Transaktion nur für Updates
BEGIN;
UPDATE orders SET status = 'processed' WHERE id = 123;
COMMIT;
```

## 2. Explizites COMMIT/ROLLBACK

Warum? Autocommit ist praktisch für Ad-hoc-Queries, aber gefährlich in Produktionscode.

```
-- Explizit ist besser als implizit
BEGIN TRANSACTION;
-- Operationen
COMMIT;
```

## 3. Passenden Isolation Level wählen

Faustregel:

Anwendungsfall	Empfohlener Level
Analytics (Read-only)	READ COMMITTED
Standard CRUD	READ COMMITTED
Ticketbuchung, Sitzplätze	SERIALIZABLE
Finanztransaktionen	SERIALIZABLE
High-throughput Logging	READ UNCOMMITTED (sehr selten!)

## 4. Fehlerbehandlung mit ROLLBACK

```
1 // Create a table with sample data
2 await db.exec(`
3   CREATE TABLE accounts (
4     id TEXT,
5     name TEXT,
6     balance INTEGER CHECK (balance >= 0)
7   );
8
9   INSERT INTO accounts VALUES
10    ('A', 'Alice', 1500),
11    ('B', 'Bob', 2300);
12 `);
13
14 try {
15   await db.exec("BEGIN TRANSACTION;");
16   await db.exec("UPDATE accounts SET balance = balance - 100 WHERE 'A'");
17   // Simuliere einen Fehler
18   //throw new Error("Simulierter Fehler während der Transaktion");
19   await db.exec("UPDATE accounts SET balance = balance - 10000 WHERE 'A'");
20 }
```

```
        'B');"");
20      await db.exec("COMMIT;");
21    } catch (error) {
22      await db.exec("ROLLBACK;");
23      console.error(JSON.stringify(error, null, 2) || error.message);
24    }
25
26    let result = await db.query("SELECT * FROM accounts;");
27
28    console.debug(JSON.stringify(result, null, 2))
```

```
{
  "length": 226,
  "name": "error",
  "severity": "ERROR",
  "code": "23514",
  "detail": "Failing row contains (B, Bob, -7700).",
  "schema": "public",
  "table": "accounts",
  "constraint": "accounts_balance_check",
  "file": "execMain.c",
  "line": "2039",
  "routine": "ExecConstraints",
  "query": "UPDATE accounts SET balance = balance - 10000 WHERE id =
'B';"
}
{
  "rows": [
    {
      "id": "A",
      "name": "Alice",
      "balance": 1500
    },
    {
      "id": "B",
      "name": "Bob",
      "balance": 2300
    }
  ],
  "fields": [
    {
      "name": "id",
      "dataTypeID": 25
    },
    {
      "name": "name",
      "dataTypeID": 25
    },
    {
      "name": "balance",
      "dataTypeID": 23
    }
  ]
}
```

```
],  
  "affectedRows": 0  
}
```

## 5. Vermeide SELECT ohne WHERE in Transaktionen

Warum? Sperrt potenziell die ganze Tabelle.

```
-- ✗ Gefährlich  
BEGIN;  
SELECT * FROM orders FOR UPDATE; -- Sperrt alle Zeilen!  
-- ...  
COMMIT;  
  
-- ✓ Besser  
BEGIN;  
SELECT * FROM orders WHERE id = 123 FOR UPDATE;  
-- ...  
COMMIT;
```

## Zusammenfassung

Was haben wir heute gelernt? Transaktionen sind das Fundament für konsistente Datenbanken – sie garantieren ACID-Eigenschaften auch bei parallelen Zugriffen und Systemausfällen.

### Kernpunkte

1. **Transaktionen** = Logische Arbeitseinheit („Alles oder Nichts“)
2. **ACID** = Atomicity, Consistency, Isolation, Durability
3. **SQL-Commands**: `BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`
4. **Isolation Levels**: Trade-off zwischen Konsistenz und Performance
  - `READ COMMITTED`: Standard, verhindert Dirty Reads
  - `SERIALIZABLE`: Maximale Isolation, aber teuer
5. **Probleme**: Dirty Reads, Non-Repeatable Reads, Phantom Reads, Lost Updates
6. **Deadlocks**: Automatisch erkannt, vermeidbar durch konsistente Lock-Reihenfolge
7. **Best Practices**: Kurze Transaktionen, explizite Steuerung, passender Isolation Level

## Referenzen & Quellen

### Dokumentation

- [PostgreSQL: Transaction Isolation](#)
- [MySQL: InnoDB Locking](#)
- [SQLite: Transactions](#)

## **Bücher**

- Martin Kleppmann: „Designing Data-Intensive Applications“ (Kapitel 7: Transactions)
- Abraham Silberschatz et al.: „Database System Concepts“ (Kapitel 14: Transactions)

## **Papers**

- Jim Gray: „The Transaction Concept: Virtues and Limitations“ (1981)
- ISO/IEC 9075: SQL Standard (Transaction Management)

## **Tools**

- [PostgreSQL EXPLAIN Visualizer](#)
- [SQL Fiddle](#) – Test Isolation Levels online

---

**Nächste Session:** Performance Optimization – Indexe, Query Plans & Concurrency Control