



DATA DRIVEN
DECISIONS

Sommersemester 2026

Datenmanagement & -analyse

Prof. Dr. Christoph M. Flath
Lehrstuhl für Wirtschaftsinformatik und Business Analytics

Julius-Maximilians-Universität Würzburg

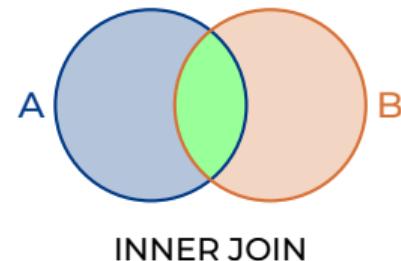


- ▶ 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- 4 Views: Virtuelle Tabellen
- 5 Transaktionen: ACID
- 6 Concurrency-Probleme
- 7 Zusammenfassung

Rückblick: JOINS

Was wir gelernt haben:

- INNER JOIN: Schnittmenge
- LEFT/RIGHT JOIN: Erhalt einer Seite
- Self-Joins: Tabelle mit sich selbst
- Join-Performance: Indizes nutzen



Heute: Komplexere Abfragen strukturieren

Wie können wir Abfragen **modular** und **wiederverwendbar** gestalten?

Motivation: Komplexe Abfragen

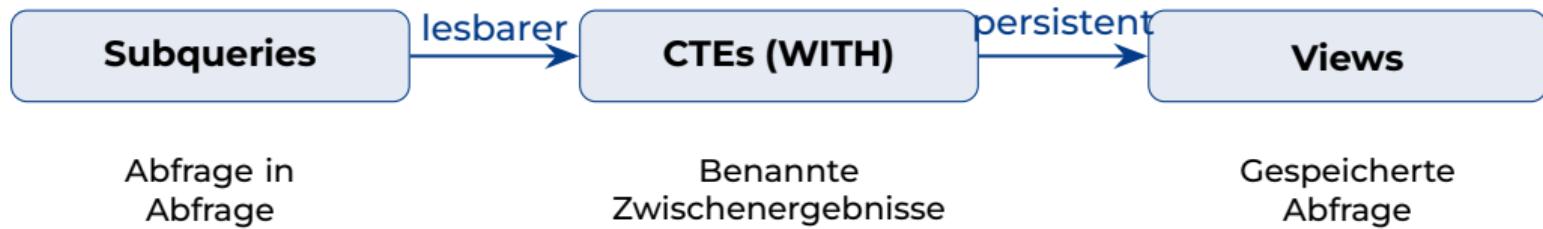
Problem: Abfragen werden schnell unübersichtlich

```
SELECT Mannschaft, Punkte,  
       (SELECT AVG(Punkte) FROM bundesliga) AS Durchschnitt,  
       Punkte - (SELECT AVG(Punkte) FROM bundesliga) AS Differenz  
FROM bundesliga  
WHERE Punkte > (SELECT AVG(Punkte) FROM bundesliga)  
ORDER BY Punkte DESC;
```

Probleme dieser Abfrage

- Durchschnitt wird 3x berechnet (redundant)
- Schwer zu lesen und zu warten
- Fehleranfällig bei Änderungen

Lösungsansätze



Und danach: Transaktionen – wie Änderungen sicher durchgeführt werden

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- 4 Views: Virtuelle Tabellen
- 5 Transaktionen: ACID
- 6 Concurrency-Probleme
- 7 Zusammenfassung

Subquery-Typen

Scalar
(1 Wert)

Column
(1 Spalte)

Table
(Tabelle)

Scalar Subquery:

```
SELECT Mannschaft,  
       Punkte - (  
           SELECT AVG(Punkte)  
             FROM bundesliga  
        ) AS Diff  
      FROM bundesliga;
```

Column Subquery:

```
SELECT Mannschaft  
  FROM bundesliga  
 WHERE Mannschaft IN (  
     SELECT Mannschaft  
       FROM champions  
    );
```

Table Subquery:

```
SELECT t.Mannschaft  
  FROM (  
     SELECT Mannschaft,  
            Punkte  
      FROM bundesliga  
     WHERE Punkte > 50  
   ) AS t;
```

Gibt genau einen Wert zurück – kann überall stehen, wo ein Wert erwartet wird

```
-- Durchschnittspunkte aller Teams
SELECT Mannschaft, Punkte,
       (SELECT AVG(Punkte) FROM bundesliga) AS Liga_Schnitt
FROM bundesliga
ORDER BY Punkte DESC;
```

Verwendbar in:

- SELECT (wie oben)
- WHERE-Bedingungen
- HAVING-Bedingungen
- Berechnungen

Achtung

Scalar Subquery muss **genau 1 Zeile** und **1 Spalte** liefern!
Sonst: Fehler.

Column Subquery: IN, ANY, ALL

Gibt eine Liste von Werten zurück

```
-- Teams, die im Pokal-Halbfinale waren
SELECT Mannschaft, Punkte
FROM bundesliga
WHERE Mannschaft IN (
    SELECT team FROM pokal_halbfinale
);
```

Operatoren:

- IN: Wert in Liste?
- NOT IN: Wert nicht in Liste?
- ANY/SOME: mind. einer erfüllt
- ALL: alle erfüllen

```
-- Mehr Punkte als IRgendein
-- Team aus Berlin
WHERE Punkte > ANY (
    SELECT Punkte
    FROM bundesliga
    WHERE Mannschaft LIKE '%Berlin%'
);

-- Mehr Punkte als ALLE
-- Teams aus Berlin
WHERE Punkte > ALL (...);
```

Korrelierte Subquery

Referenziert die äußere Abfrage – wird für jede Zeile neu ausgeführt

```
-- Teams mit überdurchschnittlich vielen Siegen für ihre Punktzahl  
SELECT m1.Mannschaft, m1.Punkte, m1.Siege  
FROM bundesliga m1  
WHERE m1.Siege > (  
    SELECT AVG(m2.Siege)  
    FROM bundesliga m2  
    WHERE m2.Punkte BETWEEN m1.Punkte - 5 AND m1.Punkte + 5  
);
```



Performance

Korrelierte Subqueries können langsam sein – für jede Zeile wird die innere Abfrage ausgeführt!

EXISTS: Existenzprüfung

Prüft, ob Subquery Ergebnisse liefert

```
-- Teams, die mindestens ein Heimspiel mit 5+ Toren hatten
SELECT DISTINCT m.Mannschaft
FROM bundesliga m
WHERE EXISTS (
    SELECT 1 FROM spiele s
    WHERE s.Heim = m.Mannschaft
    AND s.Heimtore >= 5
);
```

EXISTS vs. IN:

- EXISTS: Stoppt beim ersten Treffer
- IN: Prüft alle Werte
- EXISTS oft schneller bei großen Subqueries

Tipp

SELECT 1 in EXISTS – der Wert ist egal, nur die Existenz zählt.

Hands-on

Subqueries üben

marimo: 10-subqueries-views-transaktionen.py

Aufgaben 10.1 – 10.3

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- ▶ **3 Common Table Expressions (CTEs)**
- 4 Views: Virtuelle Tabellen
- 5 Transaktionen: ACID
- 6 Concurrency-Probleme
- 7 Zusammenfassung

Benannte Zwischenergebnisse – macht komplexe Abfragen lesbar

```
WITH durchschnitt AS (
    SELECT AVG(Punkte) AS avg_punkte
    FROM bundesliga
)
SELECT Mannschaft, Punkte,
       Punkte - (SELECT avg_punkte FROM durchschnitt) AS Differenz
FROM bundesliga, durchschnitt
WHERE Punkte > durchschnitt.avg_punkte
ORDER BY Punkte DESC;
```

Vorteile von CTEs

- Lesbarkeit: Komplexe Logik in benannte Blöcke aufteilen
- Wiederverwendung: CTE kann mehrfach referenziert werden
- Debugging: Einzelne CTEs separat testen

CTE Syntax

```
WITH cte_name AS (
    SELECT ...
),
weitere_cte AS (
    SELECT ...
)
SELECT ... FROM cte_name ...
```

- CTEs werden mit WITH eingeleitet
- Mehrere CTEs durch Komma getrennt
- Jede CTE hat einen Namen und eine Abfrage
- Hauptabfrage folgt nach der letzten CTE

Mehrere CTEs verketten

```
WITH
-- Schritt 1: Nur Top-Teams
top_teams AS (
    SELECT Mannschaft, Punkte, Siege
    FROM bundesliga
    WHERE Punkte > 50
),
-- Schritt 2: Statistiken berechnen
team_stats AS (
    SELECT Mannschaft,
        Punkte,
        CAST(Siege AS FLOAT) / NULLIF(Spiele, 0) AS Siegquote
    FROM top_teams
)
-- Hauptabfrage
SELECT * FROM team_stats
ORDER BY Siegquote DESC;
```

Tipp: Jede CTE kann vorherige CTEs referenzieren!

Rekursive CTEs

Für hierarchische Daten (Org-Charts, Stücklisten, Graphen)

```
WITH RECURSIVE mitarbeiter_hierarchie AS (
    -- Basisfall: CEO (kein Vorgesetzter)
    SELECT id, name, chef_id, 1 AS ebene
    FROM mitarbeiter
    WHERE chef_id IS NULL

    UNION ALL

    -- Rekursion: Mitarbeiter des aktuellen Levels
    SELECT m.id, m.name, m.chef_id, h.ebene + 1
    FROM mitarbeiter m
    JOIN mitarbeiter_hierarchie h ON m.chef_id = h.id
)
SELECT * FROM mitarbeiter_hierarchie;
```

Vorsicht

Endlosschleifen vermeiden! Immer Abbruchbedingung sicherstellen.

Hands-on

CTEs strukturieren Abfragen

marimo: 10-subqueries-views-transaktionen.py

Aufgaben 10.4 – 10.5

Pause
15 Minuten

Agenda

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- ▶ **4 Views: Virtuelle Tabellen**
- 5 Transaktionen: ACID
- 6 Concurrency-Probleme
- 7 Zusammenfassung

Gespeicherte Abfrage, die sich wie eine Tabelle verhält

```
CREATE VIEW top_teams AS
SELECT Mannschaft, Punkte, Tordifferenz
FROM bundesliga
WHERE Punkte > 50;
```

```
-- Verwendung wie eine normale Tabelle
SELECT * FROM top_teams ORDER BY Punkte DESC;
```

View ist:

- Keine Datenkopie
- Wird bei jedem Aufruf ausgeführt
- Immer aktuell

Vorteile:

- Abstraktion komplexer Abfragen
- Zugriffskontrolle (nur View freigeben)
- Wiederverwendbarkeit

```
-- View erstellen
CREATE VIEW team_statistik AS
SELECT Mannschaft,
       Punkte,
       ToreGeschossen,
       ToreKassiert,
       ROUND(CAST(Siege AS FLOAT) / Spiele * 100, 1) AS Siegquote
FROM bundesliga;

-- View aktualisieren (DROP + CREATE oder:)
CREATE OR REPLACE VIEW team_statistik AS
SELECT ... -- neue Definition

-- View löschen
DROP VIEW team_statistik;

-- View anzeigen (DuckDB)
SHOW TABLES; -- Views werden auch gelistet
```

Views: Wann einsetzen?

✓ Gute Anwendungsfälle:

- Komplexe JOINs abstrahieren
- Berechnete Spalten bereitstellen
- Zugriff auf Teilmengen erlauben
- Konsistente Schnittstelle für Anwendungen

✗ Weniger geeignet:

- Sehr häufige, teure Abfragen
→ Materialized View
- Einfache Abfragen
→ Direkt abfragen
- Einmalige Analysen
→ CTEs nutzen

Materialized Views (Konzept)

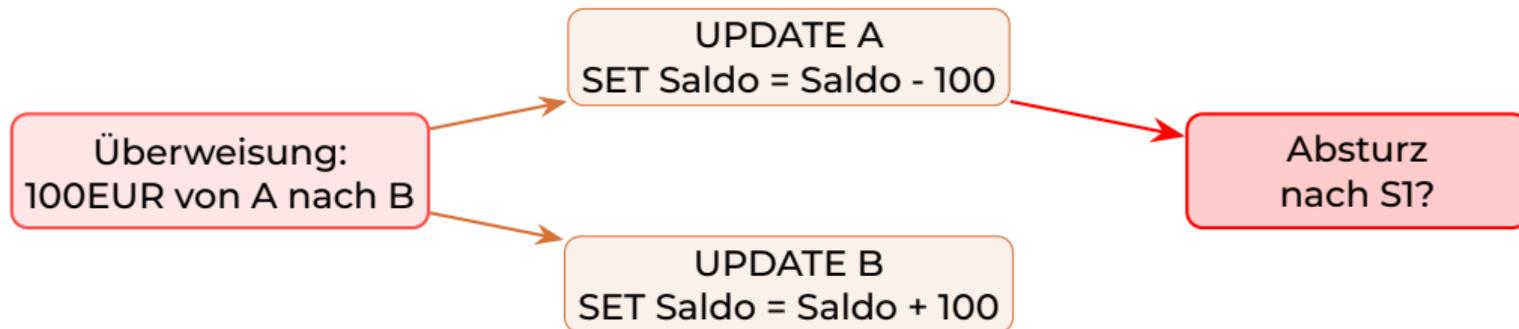
Materialized Views speichern das Ergebnis physisch – schneller, aber muss aktualisiert werden. Nicht in allen Datenbanken verfügbar.

Agenda

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- 4 Views: Virtuelle Tabellen
- ▶ **5 Transaktionen: ACID**
- 6 Concurrency-Probleme
- 7 Zusammenfassung

Transaktionen: Warum?

Problem: Mehrere zusammengehörige Änderungen



Das Problem

A hat 100EUR weniger, B hat nicht mehr bekommen!

Lösung: Transaktionen – “Alles oder nichts”

ACID-Eigenschaften

Atomicity
Alles oder nichts

Consistency
Konsistenz

Isolation
Isolation

Durability
Dauerhaftigkeit

Atomicity: Transaktion wird ganz oder gar nicht ausgeführt

Consistency: Datenbank bleibt in konsistentem Zustand

Isolation: Parallelle Transaktionen beeinflussen sich nicht

Durability: Bestätigte Änderungen überleben Systemausfälle

Transaktion: Syntax

```
BEGIN TRANSACTION;

-- Alle Änderungen hier sind Teil der Transaktion
UPDATE konten SET saldo = saldo - 100 WHERE konto_id = 'A';
UPDATE konten SET saldo = saldo + 100 WHERE konto_id = 'B';

-- Alles erfolgreich? Speichern!
COMMIT;

-- Oder: Fehler? Alles rückgängig!
ROLLBACK;
```

COMMIT:

- Alle Änderungen werden permanent
- Transaktion ist abgeschlossen

ROLLBACK:

- Alle Änderungen werden verworfen
- Zustand vor BEGIN wiederhergestellt

Beispiel: Sichere Überweisung

```
BEGIN TRANSACTION;

-- Prüfen: Hat A genug Geld?
SELECT saldo FROM konten WHERE konto_id = 'A';
-- Annahme: 500EUR -> OK

-- Abbuchen
UPDATE konten SET saldo = saldo - 100 WHERE konto_id = 'A';

-- Gutschreiben
UPDATE konten SET saldo = saldo + 100 WHERE konto_id = 'B';

-- Prüfen: Saldo von A nicht negativ?
SELECT saldo FROM konten WHERE konto_id = 'A';
-- Falls < 0: ROLLBACK; sonst:

COMMIT;
```

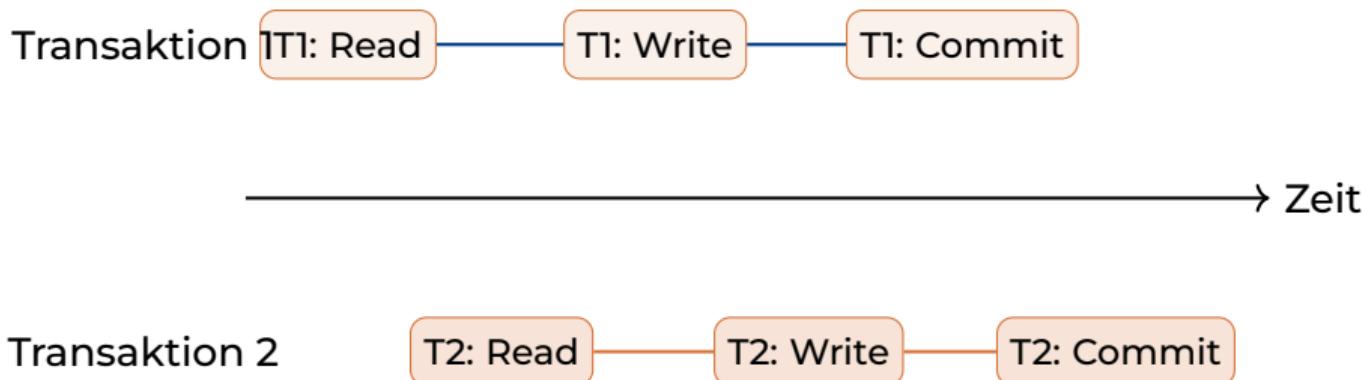
Bei Fehler oder Absturz: Automatisches ROLLBACK (Atomicity)

Agenda

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- 4 Views: Virtuelle Tabellen
- 5 Transaktionen: ACID
- ▶ **6 Concurrency-Probleme**
- 7 Zusammenfassung

Concurrency-Probleme

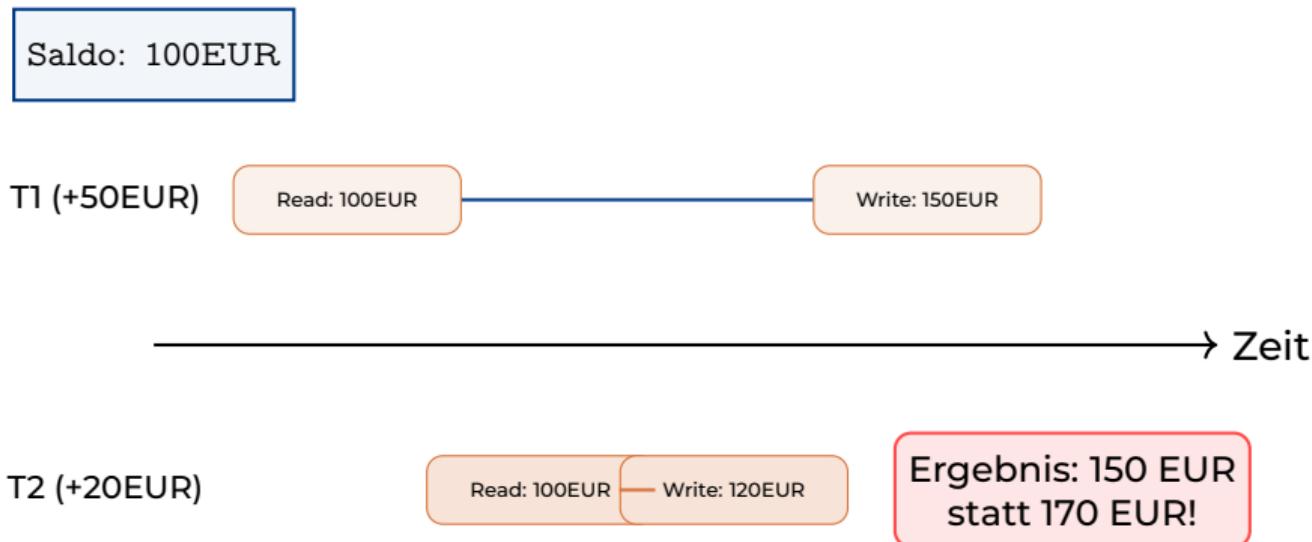
Was passiert bei gleichzeitigen Zugriffen?



Klassische Probleme:

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

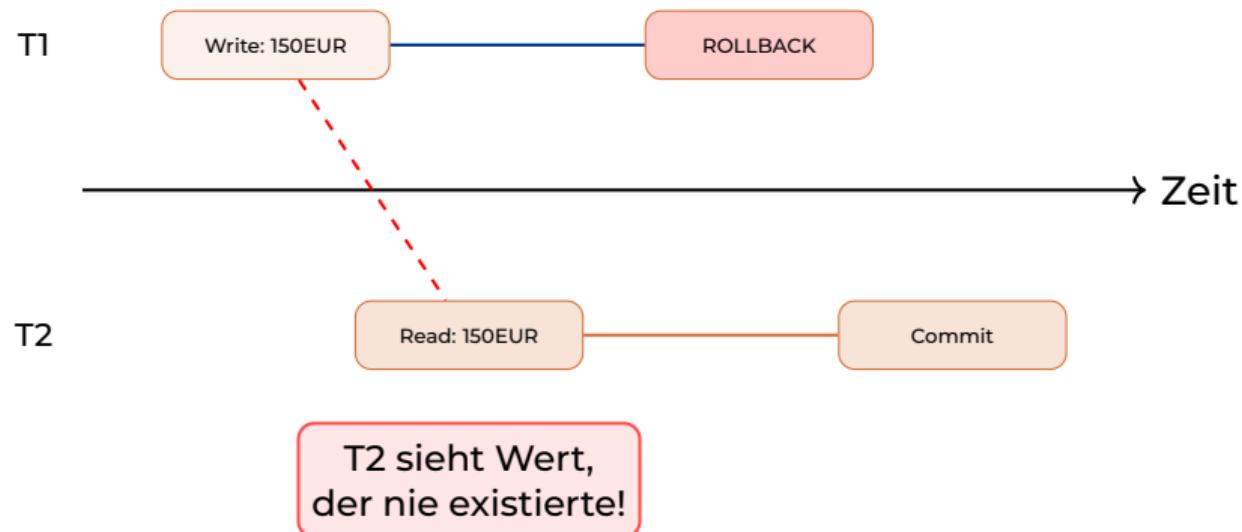
Zwei Transaktionen überschreiben sich gegenseitig



Problem: T2's Änderung (+20EUR) geht verloren, weil T1 den alten Wert überschreibt.

Dirty Read

Lesen von nicht-committeten Daten



Problem: T2 arbeitet mit Daten, die T1 später zurückrollt.

Trade-off: Mehr Isolation = weniger Parallelität

Isolation Level	Dirty Read	Non-Rep. Read	Phantom	Performance
READ UNCOMMITTED	✗	✗	✗	
READ COMMITTED	✓	✗	✗	
REPEATABLE READ	✓	✓	✗	
SERIALIZABLE	✓	✓	✓	

- **READ COMMITTED** ist oft der Standard
- **SERIALIZABLE** verhält sich wie sequentielle Ausführung
- DuckDB: Verwendet optimistisches Locking

Hands-on

Views & Transaktionen

marimo: 10-subqueries-views-transaktionen.py

Aufgaben 10.6 – 10.8

Agenda

- 1 Rückblick & Motivation
- 2 Subqueries: Grundlagen
- 3 Common Table Expressions (CTEs)
- 4 Views: Virtuelle Tabellen
- 5 Transaktionen: ACID
- 6 Concurrency-Probleme
- ▶ **7 Zusammenfassung**

Zusammenfassung

Subqueries:

- Scalar: Ein Wert
- Column: Liste (IN, ANY, ALL)
- Table: Zwischentabelle
- Korreliert: Referenz auf äußere Abfrage
- EXISTS: Existenzprüfung

CTEs (WITH):

- Benannte Zwischenergebnisse
- Mehrfach verwendbar
- Rekursiv möglich

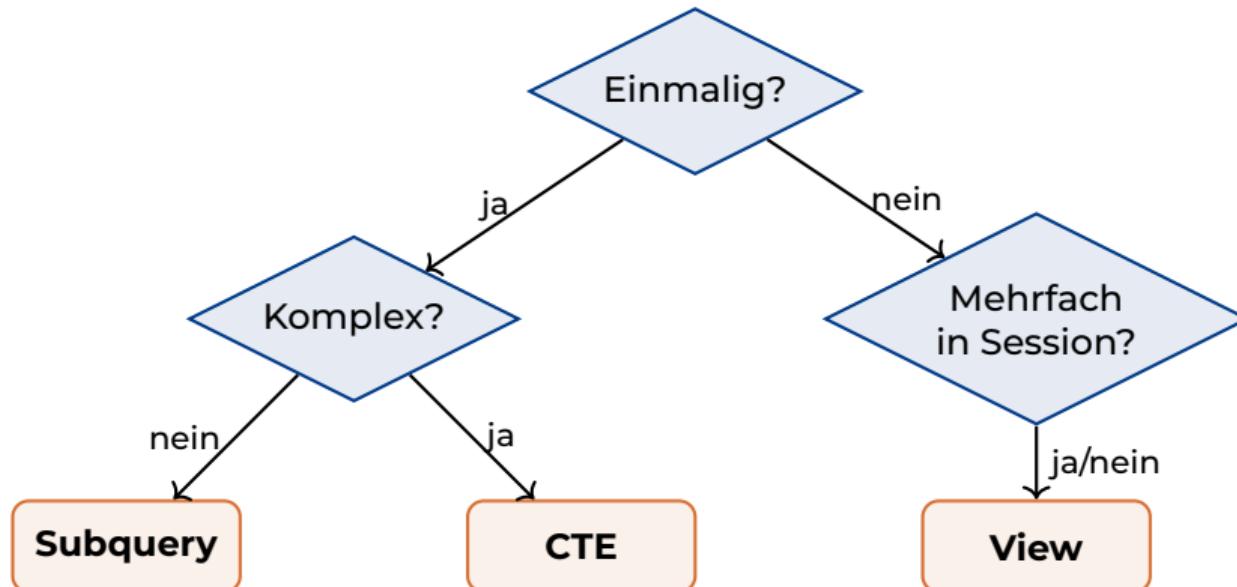
Views:

- Gespeicherte Abfragen
- Verhält sich wie Tabelle
- Abstraktion & Zugriffskontrolle

Transaktionen:

- ACID-Eigenschaften
- BEGIN, COMMIT, ROLLBACK
- Concurrency-Probleme
- Isolation Levels

Entscheidungshilfe: Was wann nutzen?



Explorative Datenanalyse (EDA)

- Systematische Datenexploration
- Univariate & bivariate Analyse
- Visualisierung mit SQL + Python
- CASE WHEN für Kategorisierung
- Binning und Histogramme

Vorbereitung

- Notebook 10 abschließen
- Wiederholen: Aggregatfunktionen, GROUP BY

Fragen?

christoph.flath@uni-wuerzburg.de