



DATA DRIVEN  
DECISIONS

Sommersemester 2026

# Datenmanagement & -analyse

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

*Julius-Maximilians-Universität Würzburg*



# Agenda

- ▶ 1 Rückblick & Motivation
- 2 INNER JOIN
- 3 LEFT & RIGHT JOIN
- 4 Self-Joins
- 5 Exkurs: Graphen in SQL
- 6 Join-Performance
- 7 Zusammenfassung



## Letzte Session:

- Funktionale Abhängigkeiten
- Normalformen (1NF, 2NF, 3NF)
- Zerlegung in redundanzfreie Tabellen

## Diese Session:

- Wie fügen wir die Daten wieder zusammen?
- **JOINS** – das Gegenstück zur Normalisierung!

# Motivation: Warum JOINS?

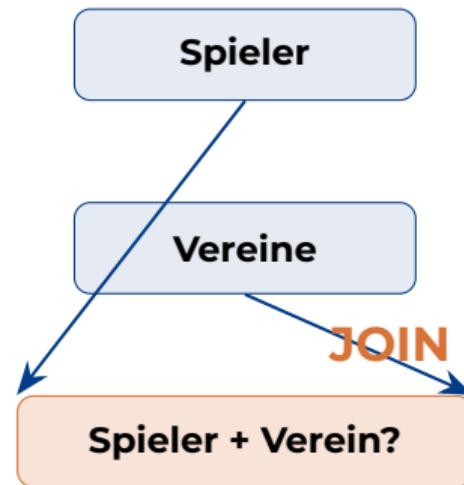
## Nach der Normalisierung:

- Daten sind aufgeteilt
- Keine Redundanz
- Aber: Zusammenhänge verteilt!

## Problem:

“Welcher Spieler spielt für welchen Verein?”

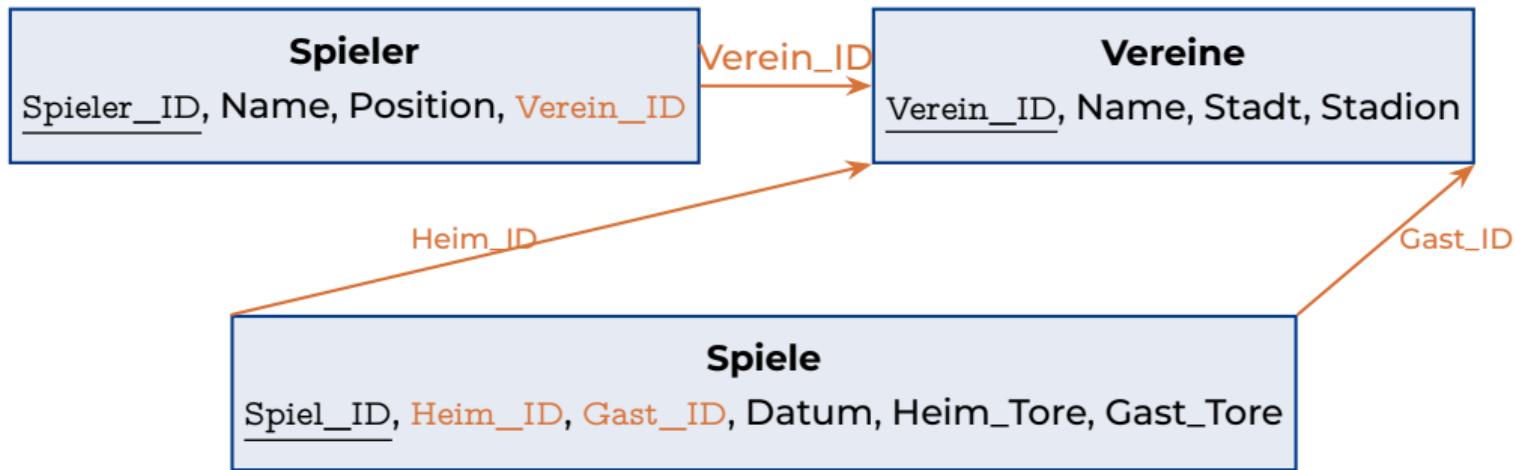
→ Information in **zwei Tabellen!**



## Kernidee

**JOIN** = Verknüpfung von Zeilen aus verschiedenen Tabellen anhand einer gemeinsamen Bedingung.

## Normalisierte Bundesliga-Datenbank:



## Fragen, die JOINS beantworten:

- Welche Spieler spielen für Bayern München?
- In welchen Stadien fanden Spiele statt?
- Welche Vereine haben noch keine Spieler?

# Beispieldaten

## Spieler

ID	Name	Pos	V_ID
1	Müller	ST	1
2	Kimmich	MF	1
3	Wirtz	MF	2
4	Tah	DF	2
5	Neuer	TW	1
6	Haaland	ST	NULL

## Vereine

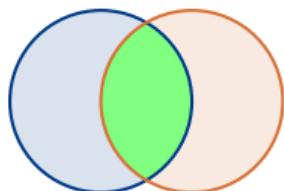
ID	Name	Stadt
1	Bayern	München
2	Leverkusen	Leverkusen
3	Dortmund	Dortmund
4	Frankfurt	Frankfurt

### Beachte:

- Haaland hat NULL als Verein\_ID (vereinslos)
- Dortmund und Frankfurt haben keine Spieler in unserer Tabelle

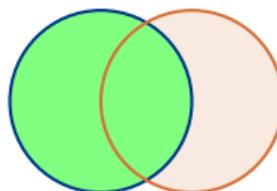
# Die JOIN-Familie: Übersicht

## INNER JOIN



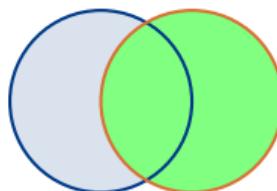
Nur Treffer

## LEFT JOIN



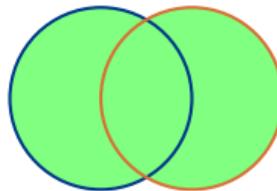
Alle links + Treffer

## RIGHT JOIN



Treffer + alle rechts

## FULL OUTER JOIN



Alle + alle

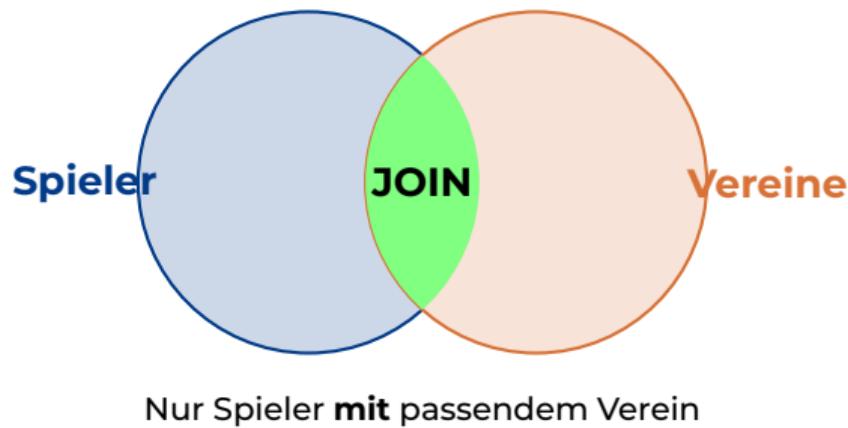
# Agenda

- 1 Rückblick & Motivation
- ▶ **2 INNER JOIN**
- 3 LEFT & RIGHT JOIN
- 4 Self-Joins
- 5 Exkurs: Graphen in SQL
- 6 Join-Performance
- 7 Zusammenfassung

# INNER JOIN: Die Schnittmenge

## Definition

Der **INNER JOIN** gibt nur die Zeilen zurück, bei denen die Join-Bedingung in **beiden** Tabellen erfüllt ist.



**Merke:** Zeilen ohne Treffer werden **nicht** ausgegeben!

## Explizite Syntax (empfohlen):

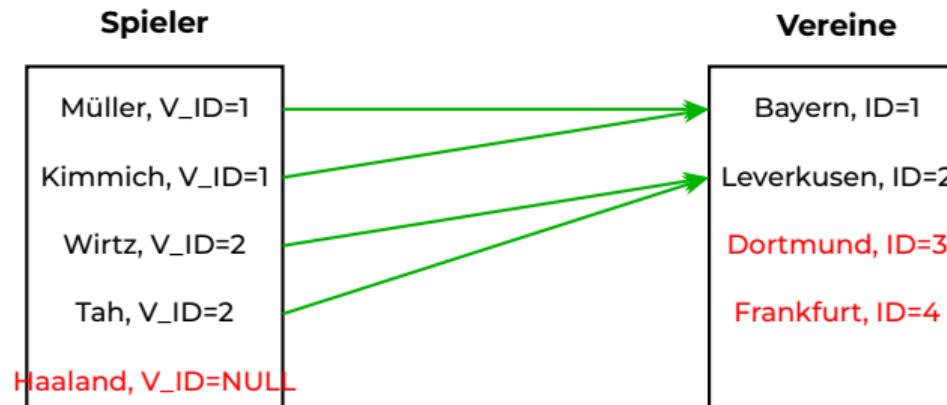
```
SELECT s.Name, v.Name AS Verein  
FROM Spieler s  
INNER JOIN Vereine v ON s.Verein_ID = v.Verein_ID;
```

## Ergebnis:

Name	Verein
Müller	Bayern
Kimmich	Bayern
Wirtz	Leverkusen
Tah	Leverkusen
Neuer	Bayern

**Beachte:** Haaland fehlt (NULL) und Dortmund/Frankfurt fehlen (keine Spieler)!

# INNER JOIN: Wie funktioniert es?



**Ergebnis: 4 Zeilen (nur Treffer)**

## Explizite Syntax (SQL-92):

```
SELECT s.Name, v.Name
FROM Spieler s
INNER JOIN Vereine v
ON s.Verein_ID = v.Verein_ID;
```

- ✓ Klar erkennbar als JOIN
- ✓ Trennung von Join und Filter
- ✓ Empfohlener Standard

## Implizite Syntax (alte Variante):

```
SELECT s.Name, v.Name
FROM Spieler s, Vereine v
WHERE s.Verein_ID = v.Verein_ID;
```

- ✗ JOIN versteckt in WHERE
- ✗ Leicht vergessbar
- ✗ Wird kartesisches Produkt bei Fehler

## Empfehlung

Immer die **explizite** JOIN-Syntax verwenden! Klarer, sicherer, wartbarer.

**Problem:** Alle Spiele mit Heim- und Gastverein-Namen?

```
SELECT
    sp.Datum,
    h.Name AS Heim,
    sp.Heim_Tore,
    sp.Gast_Tore,
    g.Name AS Gast
FROM Spiele sp
INNER JOIN Vereine h ON sp.Heim_ID = h.Verein_ID
INNER JOIN Vereine g ON sp.Gast_ID = g.Verein_ID;
```

**Beachte:**

- Zwei JOINS auf die **gleiche** Tabelle (Vereine)
- Unterschiedliche Aliase (h für Heim, g für Gast)
- JOINS werden von links nach rechts ausgeführt

# Quiz: INNER JOIN verstehen

## Gegeben:

- Spieler: 6 Zeilen (5 mit Verein, 1 mit NULL)
- Vereine: 4 Zeilen

## Wie viele Zeilen liefert dieser Query?

```
SELECT s.Name, v.Name  
FROM Spieler s  
INNER JOIN Vereine v ON s.Verein_ID = v.Verein_ID;
```

- A) 4 Zeilen (so viele wie Vereine)
- B) 5 Zeilen (Spieler mit Verein)
- C) 6 Zeilen (alle Spieler)
- D) 24 Zeilen (kartesisches Produkt)

# Quiz: INNER JOIN verstehen

## Gegeben:

- Spieler: 6 Zeilen (5 mit Verein, 1 mit NULL)
- Vereine: 4 Zeilen

## Wie viele Zeilen liefert dieser Query?

```
SELECT s.Name, v.Name
FROM Spieler s
INNER JOIN Vereine v ON s.Verein_ID = v.Verein_ID;
```

- A) 4 Zeilen (so viele wie Vereine)
- B) 5 Zeilen (Spieler mit Verein)
- C) 6 Zeilen (alle Spieler)
- D) 24 Zeilen (kartesisches Produkt)

**Antwort: B)** – Nur Spieler mit passendem Verein (5 Treffer)

# Hands-on

## INNER JOIN anwenden

marimo: 09-joins.py

Aufgaben 9.1 – 9.2

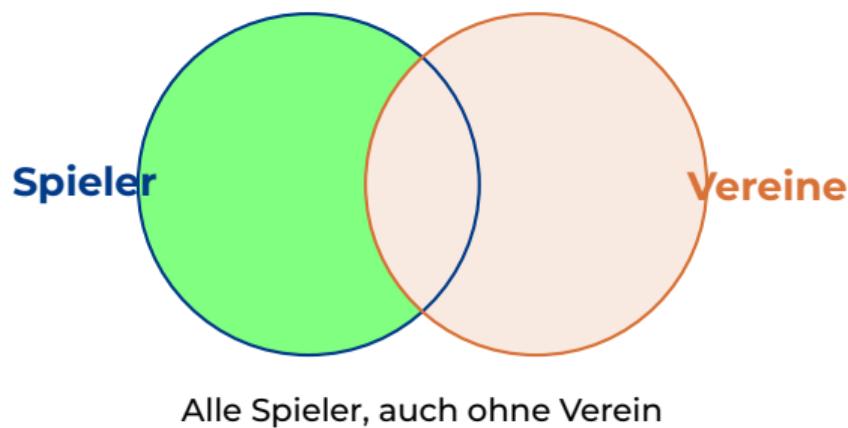
# Agenda

- 1 Rückblick & Motivation
- 2 INNER JOIN
- **3 LEFT & RIGHT JOIN**
- 4 Self-Joins
- 5 Exkurs: Graphen in SQL
- 6 Join-Performance
- 7 Zusammenfassung

# LEFT JOIN: Alle von links erhalten

## Definition

Der **LEFT JOIN** gibt **alle** Zeilen der linken Tabelle zurück, auch wenn kein Treffer in der rechten Tabelle existiert. Fehlende Werte werden mit **NUL**L aufgefüllt.



# LEFT JOIN: Syntax und Beispiel

```
SELECT s.Name, v.Name AS Verein
FROM Spieler s
LEFT JOIN Vereine v ON s.Verein_ID = v.Verein_ID;
```

## Ergebnis:

Name	Verein
Müller	Bayern
Kimmich	Bayern
Wirtz	Leverkusen
Tah	Leverkusen
Neuer	Bayern
Haaland	NULL

**Neu:** Haaland erscheint jetzt – mit NULL für Verein!

**Häufiges Muster:** "Welche Spieler haben keinen Verein?"

```
SELECT s.Name
FROM Spieler s
LEFT JOIN Vereine v ON s.Verein_ID = v.Verein_ID
WHERE v.Verein_ID IS NULL;
```

**Ergebnis:**

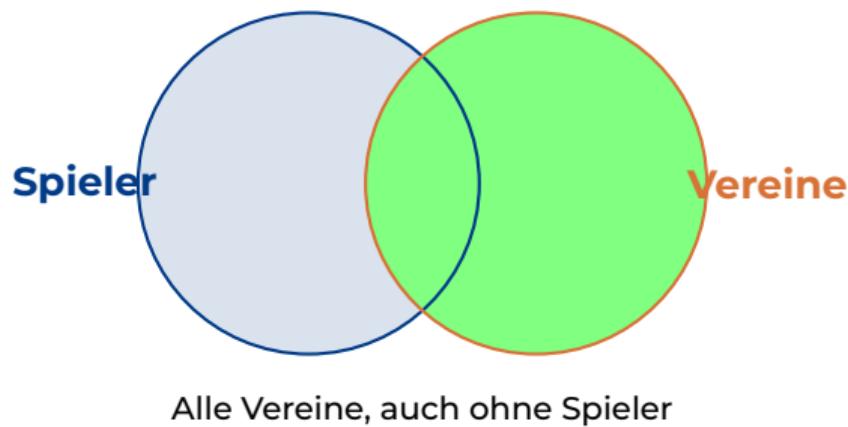
Name
Haaland

**Wichtiges Muster**

LEFT JOIN ... WHERE rechts.ID IS NULL = "Finde Zeilen ohne Treffer"

## Definition

Der **RIGHT JOIN** gibt **alle** Zeilen der rechten Tabelle zurück, auch wenn kein Treffer in der linken Tabelle existiert.



**Hinweis:** RIGHT JOIN ist selten – meist kann man die Tabellen tauschen und LEFT JOIN nutzen.

# RIGHT JOIN: Beispiel

“Welche Vereine haben (k)eine Spieler?”

```
SELECT v.Name AS Verein, COUNT(s.Spieler_ID) AS Anzahl
FROM Spieler s
RIGHT JOIN Vereine v ON s.Verein_ID = v.Verein_ID
GROUP BY v.Name;
```

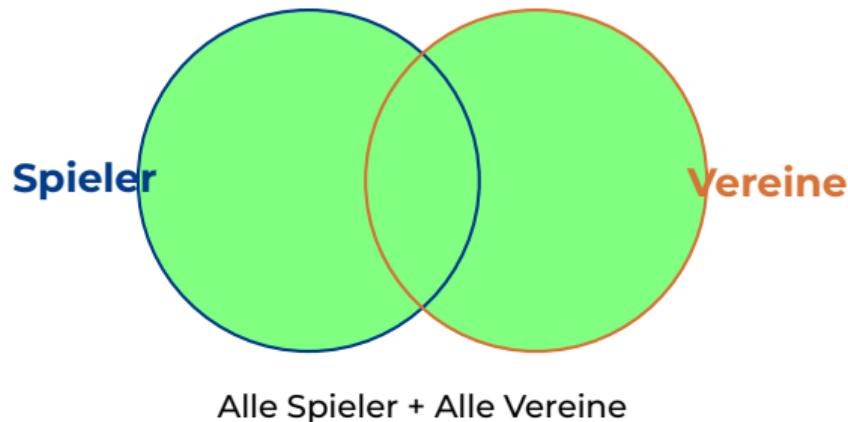
Ergebnis:

Verein	Anzahl
Bayern	3
Leverkusen	2
Dortmund	0
Frankfurt	0

Neu: Dortmund und Frankfurt erscheinen mit Anzahl 0!

## Definition

Der **FULL OUTER JOIN** gibt **alle** Zeilen beider Tabellen zurück. Fehlende Werte werden mit **NUL**L aufgefüllt.



## SQLite-Hinweis

SQLite unterstützt **FULL OUTER JOIN nicht** direkt. Workaround: LEFT JOIN + RIGHT JOIN (oder LEFT JOIN + RIGHT JOIN simulieren)

# JOIN-Typen: Zusammenfassung

JOIN-Typ	Beschreibung	Ohne Match links	Ohne Match rechts
INNER JOIN	Nur Treffer	✗ ausgeschlossen	✗ ausgeschlossen
LEFT JOIN	Alle links, Treffer rechts	✓ mit NULL	✗ ausgeschlossen
RIGHT JOIN	Treffer links, alle rechts	✗ ausgeschlossen	✓ mit NULL
FULL OUTER	Alle beider Seiten	✓ mit NULL	✓ mit NULL

## Praxis-Empfehlung:

- **INNER JOIN:** Standard, wenn nur vollständige Daten benötigt
- **LEFT JOIN:** Wenn die linke Tabelle komplett bleiben soll
- **RIGHT JOIN:** Selten – lieber Tabellen tauschen und LEFT nutzen
- **FULL OUTER:** Selten – bei Datenabgleich/-vergleich

# Quiz: Welcher JOIN?

**Anforderung:** "Liste alle Vereine mit der Anzahl ihrer Spieler. Vereine ohne Spieler sollen mit 0 erscheinen."

Welcher JOIN ist korrekt?

- A) Vereine INNER JOIN Spieler
- B) Spieler LEFT JOIN Vereine
- C) Vereine LEFT JOIN Spieler
- D) Spieler INNER JOIN Vereine

# Quiz: Welcher JOIN?

**Anforderung:** "Liste alle Vereine mit der Anzahl ihrer Spieler. Vereine ohne Spieler sollen mit 0 erscheinen."

Welcher JOIN ist korrekt?

- A) Vereine INNER JOIN Spieler
- B) Spieler LEFT JOIN Vereine
- C) Vereine LEFT JOIN Spieler
- D) Spieler INNER JOIN Vereine

**Antwort: C)** – Alle Vereine (links) erhalten, auch ohne Spieler-Treffer.

```
SELECT v.Name, COUNT(s.Spieler_ID) AS Anzahl  
FROM Vereine v  
LEFT JOIN Spieler s ON v.Verein_ID = s.Verein_ID  
GROUP BY v.Name;
```

# Hands-on

## LEFT JOIN und fehlende Daten finden

marimo: 09-joins.py

Aufgaben 9.3 – 9.4

**Pause  
15 Minuten  
Kaffee holen!**

# Agenda

- 1 Rückblick & Motivation
- 2 INNER JOIN
- 3 LEFT & RIGHT JOIN
- ▶ **4 Self-Joins**
- 5 Exkurs: Graphen in SQL
- 6 Join-Performance
- 7 Zusammenfassung

## Definition

Ein **Self-Join** verknüpft eine Tabelle mit **sich selbst**. Dabei werden Aliase verwendet, um die beiden “Kopien” zu unterscheiden.

## Typische Anwendungsfälle:

- Hierarchien (Mitarbeiter → Vorgesetzter)
- Vergleiche innerhalb einer Tabelle
- Beziehungen zwischen Elementen der gleichen Menge
- Graphen und Netzwerke



# Self-Join: Beispiel Hierarchie

**Szenario:** Mitarbeiter mit ihren Vorgesetzten

MA_ID	Name	Chef_ID
1	Müller (CEO)	NULL
2	Schmidt	1
3	Weber	1
4	Fischer	2
5	Bauer	2

```
SELECT m.Name AS Mitarbeiter , c.Name AS Chef
FROM Mitarbeiter m
LEFT JOIN Mitarbeiter c ON m.Chef_ID = c.MA_ID;
```

**Ergebnis:** Schmidt → Müller, Weber → Müller, Fischer → Schmidt, ...

**Problem:** Spiele mit Heim- und Gastnamen (beide aus Vereine)

```
SELECT
    sp.Datum,
    h.Name AS Heim,
    sp.Heim_Tore || ':' || sp.Gast_Tore AS Ergebnis,
    g.Name AS Gast
FROM Spiele sp
JOIN Vereine h ON sp.Heim_ID = h.Verein_ID
JOIN Vereine g ON sp.Gast_ID = g.Verein_ID;
```

**Ergebnis:**

Datum	Heim	Ergebnis	Gast
2026-03-15	Bayern	2:1	Dortmund
2026-03-16	Leverkusen	3:0	Frankfurt

**Beachte:** Zwei JOINs auf Vereine mit unterschiedlichen Aliasen (h, g)!

**Problem:** Welche Spieler spielen im gleichen Verein?

```
SELECT
    s1.Name AS Spieler1 ,
    s2.Name AS Spieler2 ,
    v.Name AS Verein
FROM Spieler s1
JOIN Spieler s2 ON s1.Verein_ID = s2.Verein_ID
    AND s1.Spieler_ID < s2.Spieler_ID
JOIN Vereine v ON s1.Verein_ID = v.Verein_ID ;
```

**Ergebnis:**

Spieler1	Spieler2	Verein
Müller	Kimmich	Bayern
Müller	Neuer	Bayern
Kimmich	Neuer	Bayern
Wirtz	Tah	Leverkusen

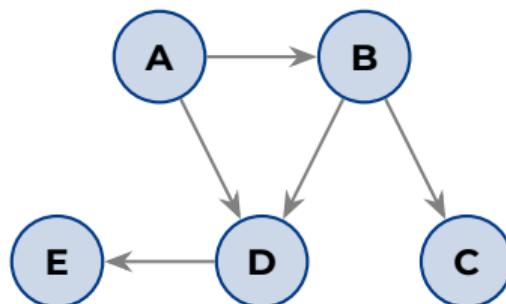
**Wichtig:** s1.ID < s2.ID verhindert Duplikate und Selbst-Paare!

# Agenda

- 1 Rückblick & Motivation
- 2 INNER JOIN
- 3 LEFT & RIGHT JOIN
- 4 Self-Joins
- ▶ **5 Exkurs: Graphen in SQL**
- 6 Join-Performance
- 7 Zusammenfassung

**Motivation:** Self-Joins ermöglichen einfache Graph-Operationen!

**Graph als Edge-List:**



**Als Tabelle:**

von	nach
A	B
A	D
B	C
B	D
D	E

**Idee:** Jede Kante ist eine Zeile mit (Quelle, Ziel).

## Beispiel: Freundschaftsbeziehungen

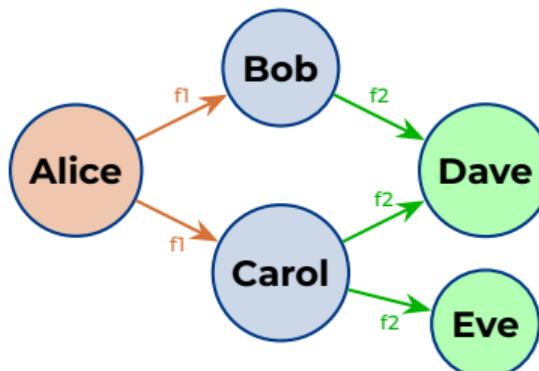
```
CREATE TABLE friendships (
    person_a TEXT,
    person_b TEXT
);
```

person_a	person_b
Alice	Bob
Alice	Carol
Bob	Dave
Carol	Dave
Carol	Eve

**Frage:** Wer sind die “Freunde von Freunden” von Alice?

## 2-Hop-Pfad mit Self-Join:

```
SELECT DISTINCT f2.person_b AS friend_of_friend
FROM friendships f1
JOIN friendships f2 ON f1.person_b = f2.person_a
WHERE f1.person_a = 'Alice'
AND f2.person_b <> 'Alice';
```



**Ergebnis:** Dave, Eve (Freunde von Freunden)

## Grenzen von SQL bei Graphen:

### Geht mit SQL:

- Direkte Nachbarn (1 Hop)
- 2-Hop-Pfade (Self-Join)
- N-Hop-Pfade (N JOINs)
- Zyklen fester Länge

### Geht NICHT (oder schwer):

- Kürzester Pfad beliebiger Länge
- PageRank
- Zusammenhangskomponenten
- Transitive Hülle (alle Pfade)

## Fazit

Für komplexe Graph-Analysen: Spezialisierte Tools nutzen!

- **Python:** NetworkX
- **Datenbanken:** Neo4j, Amazon Neptune
- **Big Data:** Apache Spark GraphX

SQL kann mehr mit “Recursive CTE”:



Damit möglich:

- Transitive Hülle (alle erreichbaren Knoten)
- Hierarchie-Traversierung beliebiger Tiefe
- Pfade in Graphen

**Hinweis:** Wird in Session 10 (Subqueries & CTEs) vertieft.

# Agenda

- 1 Rückblick & Motivation
- 2 INNER JOIN
- 3 LEFT & RIGHT JOIN
- 4 Self-Joins
- 5 Exkurs: Graphen in SQL
- ▶ **6 Join-Performance**
- 7 Zusammenfassung

## JOINS können teuer sein:



## Kartesisches Produkt:

- Ohne Bedingung:  $n \times m$  Kombinationen
- $1.000 \times 1.000 = 1.000.000$  potentielle Paare
- Datenbankoptimizer reduziert das – aber nur mit Index!

# Join-Algorithmen: Überblick

Wie die Datenbank JOINs ausführt:

Algorithmus	Funktionsweise	Wann gut?
<b>Nested Loop</b>	Für jede Zeile links: durchsuche rechts	Kleine Tabellen, Index rechts
<b>Hash Join</b>	Hash-Tabelle bauen, dann matchen	Große Tabellen, kein Index
<b>Merge Join</b>	Beide sortiert, parallel durchlaufen	Bereits sortierte Daten

Der Optimizer wählt automatisch!

Aber: Wir können helfen durch:

- Indizes auf Join-Spalten
- Sinnvolle Join-Reihenfolge
- Vermeidung von Funktionen auf Join-Spalten

# Indizes für JOIN-Performance

## Regel

Join-Spalten sollten **indiziert** sein – besonders die Fremdschlüssel!

### Ohne Index:

- Datenbank muss **alle** Zeilen durchsuchen
- Bei großen Tabellen: sehr langsam

### Mit Index:

- Direkter Zugriff auf passende Zeilen
- Logarithmische statt linearer Suche

1.000.000 Zeilen scannen



20 Index-Lookups

Mit EXPLAIN sehen, was die Datenbank plant:

```
EXPLAIN QUERY PLAN
SELECT s.Name, v.Name
FROM Spieler s
JOIN Vereine v ON s.Verein_ID = v.Verein_ID;
```

**Beispiel-Ausgabe (SQLite):**

```
QUERY PLAN
|--SCAN Spieler
'--SEARCH Vereine USING INDEX ...
```

**Interpretation:**

- SCAN: Alle Zeilen durchlaufen (langsam bei großen Tabellen)
- SEARCH ... USING INDEX: Index-Lookup (schnell)

## 1. Indizes erstellen:

- Alle Fremdschlüssel indizieren
- Häufig gefilterte Spalten indizieren

## 2. Früh filtern:

- WHERE-Bedingungen so früh wie möglich
- Weniger Zeilen im JOIN = schneller

## 3. Nur benötigte Spalten:

- SELECT \* vermeiden
- Explizit nur benötigte Spalten auswählen

## 4. Join-Reihenfolge beachten:

- Kleine Tabelle zuerst (bei manchen Datenbanken)
- Optimizer macht das meist automatisch

## Definition

Ein **CROSS JOIN** erzeugt alle möglichen Kombinationen – **ohne** Bedingung.

**Selten gewollt, aber nützlich für:**

- Kalender/Zeitreihen generieren
- Alle Kombinationen erzeugen
- Test-Datensätze erstellen

Links (3)	Rechts (4)	CROSS (12)
A	1	A-1, A-2, A-3, A-4
B	2	B-1, B-2, B-3, B-4
C	3	C-1, C-2, C-3, C-4
	4	

## Kurzschrifweis (mit Vorsicht zu genießen):

### USING – wenn Spaltenname identisch:

```
SELECT * FROM Spieler
JOIN Vereine USING (Verein_ID);
```

- ✓ Kurz und lesbar bei gleichnamigen Spalten

### NATURAL JOIN – automatisch nach gleichen Namen:

```
SELECT * FROM Spieler
NATURAL JOIN Vereine;
```

- ✗ Gefährlich! Joined auf **alle** gleichnamigen Spalten – auch unbeabsichtigt!

## Empfehlung

USING ist akzeptabel. NATURAL JOIN vermeiden – zu implizit, fehleranfällig.

# Hands-on

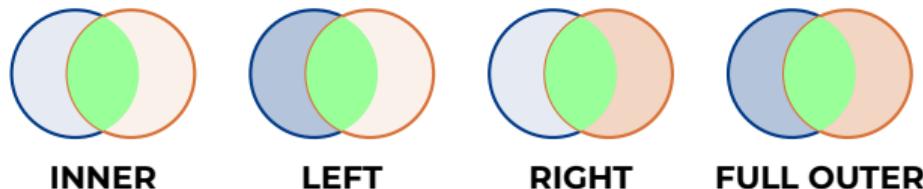
## Self-Joins und Performance

marimo: 09-joins.py

Aufgaben 9.5 – 9.7

# Agenda

- 1 Rückblick & Motivation
- 2 INNER JOIN
- 3 LEFT & RIGHT JOIN
- 4 Self-Joins
- 5 Exkurs: Graphen in SQL
- 6 Join-Performance
- ▶ **7 Zusammenfassung**



- **INNER JOIN:** Nur Treffer beider Seiten
- **LEFT JOIN:** Alle links + Treffer (NULL wenn kein Match)
- **RIGHT JOIN:** Treffer + Alle rechts (selten benötigt)
- **FULL OUTER:** Alle beider Seiten (nicht in SQLite)

## 1. Fehlende Daten finden:

- LEFT JOIN ... WHERE rechts.ID IS NULL
- "Spieler ohne Verein", "Vereine ohne Spieler"

## 2. Self-Join für interne Beziehungen:

- Gleiche Tabelle mit verschiedenen Aliasen
- Hierarchien, Vergleiche, Graphen

## 3. Mehrere JOINS verketten:

- FROM A JOIN B JOIN C
- Spiele mit Heim- und Gast-Vereinsnamen

## 4. Performance beachten:

- Indizes auf Fremdschlüsseln
- EXPLAIN zur Analyse

Muster	SQL-Syntax
INNER JOIN	FROM A INNER JOIN B ON A.id = B.a_id
LEFT JOIN	FROM A LEFT JOIN B ON A.id = B.a_id
RIGHT JOIN	FROM A RIGHT JOIN B ON A.id = B.a_id
Self-Join	FROM T t1 JOIN T t2 ON t1.x = t2.y
Mehrfach-Join	FROM A JOIN B ON ... JOIN C ON ...
Fehlende finden	LEFT JOIN ... WHERE B.id IS NULL
USING	FROM A JOIN B USING (gemeinsame_spalte)

## Merksatz:

*“Die linke Tabelle bestimmt die Grundmenge bei LEFT JOIN – alle ihre Zeilen bleiben erhalten.”*

# Häufige Fehler vermeiden

## **X Fehler:**

- JOIN ohne ON-Bedingung  
→ Kartesisches Produkt!
- Falsche Spalte im JOIN  
→ Unerwartete Ergebnisse
- LEFT statt INNER verwechselt  
→ NULL-Werte unerwartet
- Fehlende Aliase bei Self-Join  
→ Syntax-Fehler

## **✓ Best Practices:**

- Immer explizite ON-Klausel
- Aliase bei mehrdeutigen Spaltennamen
- Tabellenname.Spalte bei JOINS
- EXPLAIN bei langsamem Queries
- Indizes auf FK-Spalten

## Tipp

Bei unerwarteten Ergebnissen: Erst ohne WHERE prüfen, dann Filter hinzufügen.

## Vorlesung 10: Subqueries, Views & Transaktionen

- Subqueries: Abfragen in Abfragen
- Common Table Expressions (WITH)
- Views: Virtuelle Tabellen
- ACID und Transaktionen



**Mit JOINs und Subqueries sind fast alle SQL-Abfragen möglich!**

# Fragen?

christoph.flath@uni-wuerzburg.de