L10: Daten über Tabellengrenzen hinweg kombinieren – Von Subqueries zu CTEs zu Joins

Session 10 – Lecture Dauer: 90 Minuten **Lernziele:** LZ 2 (Relationale DB & SQL praktisch anwenden)
Block: 2 – SQL Einführung & Grundlagen

Willkommen zur zehnten Session! Bisher haben Sie mit einzelnen Tabellen gearbeitet – SELECT, WHERE, GROUP BY, alles auf einer Tabelle. Das war wichtig zum Lernen, aber die wahre Macht relationaler Datenbanken liegt woanders: in Beziehungen.

Kunden haben Bestellungen. Bestellungen haben Positionen. Produkte gehören zu Kategorien. All diese Informationen leben in verschiedenen Tabellen – aber wie kombinieren wir sie? Es gibt drei Hauptansätze: Subqueries, CTEs und Joins.

In dieser Session lernen Sie alle drei Techniken kennen – und verstehen, wann Sie welche einsetzen. Wir beginnen mit dem intuitiven Weg (Subqueries), zeigen dessen Grenzen, verbessern ihn mit CTEs, und landen schließlich bei der elegantesten Lösung: Joins.

Los geht's mit der Frage: Warum müssen wir Daten überhaupt kombinieren?

Warum Daten kombinieren?

Stellen Sie sich vor, Sie speichern alles in einer Tabelle: Kundendaten, Bestellungen, Produktdetails – alles zusammen. Was passiert?

Problem 1: Redundanz Sie speichern die Kundenadresse bei jeder Bestellung neu. Zieht der Kunde um, müssen Sie Dutzende Zeilen aktualisieren.

Problem 2: Inkonsistenz Bei manchen Bestellungen steht „Berlin„, bei anderen „Berln“ – Tippfehler.

Problem 3: Update-Anomalien Sie ändern den Preis eines Produkts – aber welche Bestellungen bekommen den neuen Preis? Die alten sollten den alten Preis behalten!

Die Lösung? Normalisierung. Wir teilen Daten in mehrere Tabellen auf. Jede Tabelle hat eine klar definierte Verantwortung. Beziehungen werden über Foreign Keys hergestellt.

Normalisierung führt zu mehreren Tabellen.

Joins rekonstruieren die Informationen.

Unser E-Commerce-Schema

Für alle Beispiele heute nutzen wir ein realistisches E-Commerce-Schema mit sieben normalisierten Tabellen inklusive einer N:M-Beziehung über eine Junction Table.

```sql
-- Locations: Normalisierte Orte mit PLZ
CREATE TABLE locations (
  location_id INTEGER PRIMARY KEY,
  city TEXT NOT NULL,
  postal_code TEXT NOT NULL,
  country TEXT DEFAULT 'Germany'
);

-- Categories: Normalisierte Produktkategorien
CREATE TABLE categories (
  category_id INTEGER PRIMARY KEY,
  category_name TEXT NOT NULL UNIQUE,
  description TEXT
);

-- Customers: Erweitert mit strukturierten Adressdaten
CREATE TABLE customers (
  customer_id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE,
  street TEXT,
  street_number TEXT,
  location_id INTEGER,
  FOREIGN KEY (location_id) REFERENCES locations(location_id)
);

-- Orders: Unverändert
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  order_date DATE,
  total_amount DECIMAL(10,2),
  status TEXT,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Products: Ohne direkte Category-Referenz (N:M über Junction Table
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  product_name TEXT NOT NULL,
  price DECIMAL(10,2)
);
```

```sql

-- Product Categories: Junction Table für N:M-Beziehung
CREATE TABLE product_categories (
  product_id INTEGER,
  category_id INTEGER,
  PRIMARY KEY (product_id, category_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id),
  FOREIGN KEY (category_id) REFERENCES categories(category_id)
);

-- Order Items: Unverändert
CREATE TABLE order_items (
  order_item_id INTEGER PRIMARY KEY,
  order_id INTEGER,
  product_id INTEGER,
  quantity INTEGER,
  line_total DECIMAL(10,2),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);

-- Sample Data: Locations
INSERT INTO locations(location_id, city, postal_code, country) VALUE
  (1, 'Berlin', '10115', 'Germany'),
  (2, 'Hamburg', '20095', 'Germany'),
  (3, 'Munich', '80331', 'Germany'),
  (4, 'Cologne', '50667', 'Germany'),
  (5, 'Frankfurt', '60311', 'Germany');

-- Sample Data: Categories
INSERT INTO categories(category_id, category_name, description) VALU
  (1, 'Electronics', 'Electronic devices and accessories'),
  (2, 'Furniture', 'Office and home furniture'),
  (3, 'Stationery', 'Office supplies and paper products'),
  (4, 'Office Equipment', 'Professional office tools and devices');

-- Sample Data: Customers (mit strukturierten Adressen)
INSERT INTO customers(customer_id, first_name, last_name, email, str
  street_number, location_id) VALUES
  (1, 'Alice', 'Anderson', 'alice@email.com', 'Unter den Linden', '4
    ),
  (2, 'Bob',   'Brown',    'bob@email.com',   'Reeperbahn',       '1
    ),
  (3, 'Carol', 'Clark',    'carol@email.com', 'Marienplatz',      '
    ),
  (4, 'David', 'Davis',    'david@email.com', 'Hohe Straße',     '12
    ),
  (5, 'Emma',  'Evans',    'emma@email.com',  'Zeil',             '9
    ),
  (99, 'Zoe',   'Zimmer',  'zoe@emailcom',    'Unter den Linden',  '
    ); -- Orphaned customer! No orders.
```

```sql

-- Sample Data: Orders (Note: Customer 5 has NO orders! Order 106 ha
  invalid customer_id!)
INSERT INTO orders(order_id, customer_id, order_date, total_amount,
  ) VALUES
  (101, 1, '2024-01-15', 299.99, 'completed'),
  (102, 1, '2024-02-20', 149.50, 'completed'),
  (103, 2, '2024-01-22', 499.99, 'completed'),
  (104, 3, '2024-03-10',  89.99, 'pending'  ),
  (105, 4, '2024-03-15', 199.99, 'completed'),
  (106, 99, '2023-03-20', 79.99, 'completed'); -- Orphaned order! Cu
    99 doesn't exist!

-- Sample Data: Products (ohne direkte Category-Referenz)
INSERT INTO products(product_id, product_name, price) VALUES
  (1, 'Laptop',     999.99),
  (2, 'Mouse',       29.99),
  (3, 'Keyboard',    79.99),
  (4, 'Monitor',    299.99),
  (5, 'Desk Chair', 199.99),
  (6, 'Notebook',     9.99),
  (7, 'USB Cable',   14.99),
  (8, 'Desk Lamp',   49.99),
  (9, 'Paper',        4.99);

-- Sample Data: Product Categories (N:M-Beziehungen)
INSERT INTO product_categories(product_id, category_id) VALUES
  (1, 1),  -- Laptop → Electronics
  (1, 4),  -- Laptop → Office Equipment
  (2, 1),  -- Mouse → Electronics
  (2, 4),  -- Mouse → Office Equipment
  (3, 1),  -- Keyboard → Electronics
  (3, 4),  -- Keyboard → Office Equipment
  (4, 1),  -- Monitor → Electronics
  (4, 4),  -- Monitor → Office Equipment
  (5, 2),  -- Desk Chair → Furniture
  (5, 4),  -- Desk Chair → Office Equipment
  (6, 3),  -- Notebook → Stationery (nur eine Kategorie!)
  (7, 1),  -- USB Cable → Electronics (nicht verkauft!)
  (8, 2),  -- Desk Lamp → Furniture (nicht verkauft!)
  (8, 4),  -- Desk Lamp → Office Equipment
  (9, 3);  -- Paper → Stationery (nicht verkauft!)

-- Sample Data: Order Items
INSERT INTO order_items(order_item_id, order_id, product_id, quantit
  line_total) VALUES
  (1, 101, 4, 1, 299.99),
  (2, 102, 2, 2,  59.98),
  (3, 102, 3, 1,  79.99),
  (4, 103, 1, 1, 999.99),
  (5, 104, 6, 5,  49.95),
```

```
  (6, 105, 5, 1, 199.99),
  (7, 106, 7, 5,  74.95); -- Orphaned order 106: USB Cable

-- Create orphan data for testing purposes
UPDATE orders
SET customer_id = NULL
WHERE customer_id = 99;

DELETE FROM customers
WHERE customer_id = 99;
```

```
-- Locations: Normalisierte Orte mit PLZ
CREATE TABLE locations (
  location_id INTEGER PRIMARY KEY,
  city TEXT NOT NULL,
  postal_code TEXT NOT NULL,
  country TEXT DEFAULT 'Germany'
)
```

ok

```
-- Categories: Normalisierte Produktkategorien
CREATE TABLE categories (
  category_id INTEGER PRIMARY KEY,
  category_name TEXT NOT NULL UNIQUE,
  description TEXT
)
```

ok

```
-- Customers: Erweitert mit strukturierten Adressdaten
CREATE TABLE customers (
  customer_id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE,
  street TEXT,
  street_number TEXT,
  location_id INTEGER,
  FOREIGN KEY (location_id) REFERENCES locations(location_id)
)
```

ok

```
-- Orders: Unverändert
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  order_date DATE,
  total_amount DECIMAL(10,2),
  status TEXT,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
)
```

ok

```
-- Products: Ohne direkte Category-Referenz (N:M über Junction Table)
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  product_name TEXT NOT NULL,
  price DECIMAL(10,2)
)
```

ok

```
-- Product Categories: Junction Table für N:M-Beziehung
CREATE TABLE product_categories (
  product_id INTEGER,
  category_id INTEGER,
  PRIMARY KEY (product_id, category_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id),
  FOREIGN KEY (category_id) REFERENCES categories(category_id)
)
```

ok

```
-- Order Items: Unverändert
CREATE TABLE order_items (
  order_item_id INTEGER PRIMARY KEY,
  order_id INTEGER,
  product_id INTEGER,
  quantity INTEGER,
  line_total DECIMAL(10,2),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
)
```

ok

```
-- Sample Data: Locations
INSERT INTO locations(location_id, city, postal_code, country) VALUES
  (1, 'Berlin', '10115', 'Germany'),
  (2, 'Hamburg', '20095', 'Germany'),
  (3, 'Munich', '80331', 'Germany'),
  (4, 'Cologne', '50667', 'Germany'),
  (5, 'Frankfurt', '60311', 'Germany')
```

ok

```
-- Sample Data: Categories
INSERT INTO categories(category_id, category_name, description) VALUES
  (1, 'Electronics', 'Electronic devices and accessories'),
  (2, 'Furniture', 'Office and home furniture'),
  (3, 'Stationery', 'Office supplies and paper products'),
  (4, 'Office Equipment', 'Professional office tools and devices')
```

ok

```
-- Sample Data: Customers (mit strukturierten Adressen)
INSERT INTO customers(customer_id, first_name, last_name, email,
street, street_number, location_id) VALUES
  (1, 'Alice', 'Anderson', 'alice@email.com', 'Unter den Linden', '42',
1),
  (2, 'Bob',   'Brown',    'bob@email.com',   'Reeperbahn',       '15',
```

```
2),
  (3, 'Carol', 'Clark',    'carol@email.com', 'Marienplatz',       '8',
3),
  (4, 'David', 'Davis',    'david@email.com', 'Hohe Straße',     '123',
4),
  (5, 'Emma',  'Evans',    'emma@email.com',  'Zeil',             '99',
5),
  (99, 'Zoe',   'Zimmer',  'zoe@emailcom',    'Unter den Linden',  '1',
1)
```

ok

```
-- Orphaned customer! No orders.

-- Sample Data: Orders (Note: Customer 5 has NO orders! Order 106 has
invalid customer_id!)
INSERT INTO orders(order_id, customer_id, order_date, total_amount,
status) VALUES
  (101, 1, '2024-01-15', 299.99, 'completed'),
  (102, 1, '2024-02-20', 149.50, 'completed'),
  (103, 2, '2024-01-22', 499.99, 'completed'),
  (104, 3, '2024-03-10',  89.99, 'pending'  ),
  (105, 4, '2024-03-15', 199.99, 'completed'),
  (106, 99, '2023-03-20', 79.99, 'completed')
```

ok

```
-- Orphaned order! Customer 99 doesn't exist!

-- Sample Data: Products (ohne direkte Category-Referenz)
INSERT INTO products(product_id, product_name, price) VALUES
  (1, 'Laptop',     999.99),
  (2, 'Mouse',       29.99),
  (3, 'Keyboard',    79.99),
  (4, 'Monitor',    299.99),
  (5, 'Desk Chair', 199.99),
  (6, 'Notebook',     9.99),
  (7, 'USB Cable',   14.99),
  (8, 'Desk Lamp',   49.99),
  (9, 'Paper',        4.99)
```

ok

```
-- Sample Data: Product Categories (N:M-Beziehungen)
INSERT INTO product_categories(product_id, category_id) VALUES
  (1, 1),  -- Laptop → Electronics
  (1, 4),  -- Laptop → Office Equipment
  (2, 1),  -- Mouse → Electronics
  (2, 4),  -- Mouse → Office Equipment
  (3, 1),  -- Keyboard → Electronics
  (3, 4),  -- Keyboard → Office Equipment
```

```
  (4, 1),  -- Monitor → Electronics
  (4, 4),  -- Monitor → Office Equipment
  (5, 2),  -- Desk Chair → Furniture
  (5, 4),  -- Desk Chair → Office Equipment
  (6, 3),  -- Notebook → Stationery (nur eine Kategorie!)
  (7, 1),  -- USB Cable → Electronics (nicht verkauft!)
  (8, 2),  -- Desk Lamp → Furniture (nicht verkauft!)
  (8, 4),  -- Desk Lamp → Office Equipment
  (9, 3)
```

ok

```
-- Paper → Stationery (nicht verkauft!)

-- Sample Data: Order Items
INSERT INTO order_items(order_item_id, order_id, product_id, quantity,
line_total) VALUES
  (1, 101, 4, 1, 299.99),
  (2, 102, 2, 2,  59.98),
  (3, 102, 3, 1,  79.99),
  (4, 103, 1, 1, 999.99),
  (5, 104, 6, 5,  49.95),
  (6, 105, 5, 1, 199.99),
  (7, 106, 7, 5,  74.95)
```

ok

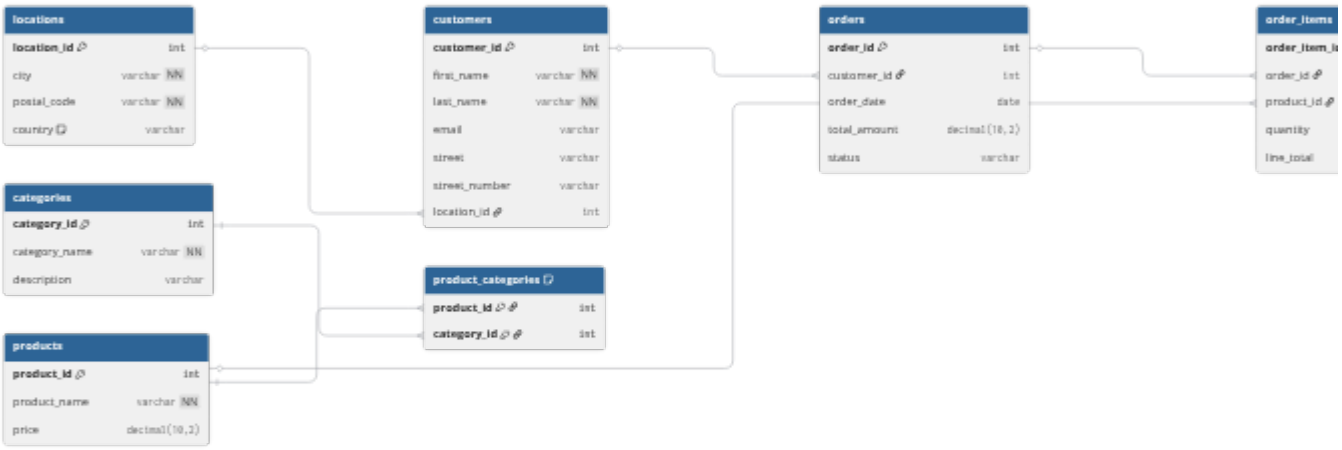
```
-- Orphaned order 106: USB Cable

-- Create orphan data for testing purposes
UPDATE orders
SET customer_id = NULL
WHERE customer_id = 99
```

ok

```
DELETE FROM customers
WHERE customer_id = 99
```

ok

Beziehungen:



dbdiagram.io

dbdiagram.io

- Ein Ort kann viele Kunden haben (1:N)
- **Ein Produkt kann viele Kategorien haben (N:M über product_categories)**
- **Eine Kategorie kann viele Produkte haben (N:M über product_categories)**
- Ein Kunde kann viele Bestellungen haben (1:N)
- Eine Bestellung hat viele Positionen (1:N)
- Ein Produkt kann in vielen Positionen vorkommen (N:M über order_items)

Diese Struktur ist typisch für relationale Datenbanken. Aber wie kombinieren wir diese Informationen? Schauen wir uns vier Ansätze an – beginnend mit dem ältesten, aber wichtigsten zum Verstehen.

Technik 0: Verknüpfen von Tabellen über `FROM`

Bevor wir zu modernen Techniken kommen, müssen wir die Basis verstehen: Wie kombiniert SQL überhaupt Tabellen? Die Antwort liegt im FROM. Sie können mehrere Tabellen einfach durch Kommata trennen – und erhalten das kartesische Produkt.

Das kartesische Produkt: Alle Kombinationen

Wenn Sie zwei Tabellen im FROM auflisten, verbindet SQL jede Zeile der ersten Tabelle mit jeder Zeile der zweiten Tabelle. Das nennt man kartesisches Produkt oder Cross Product.

Konzept:

Customers (3 Zeilen)

ID	Name
1	Alice
2	Bob
5	Emma

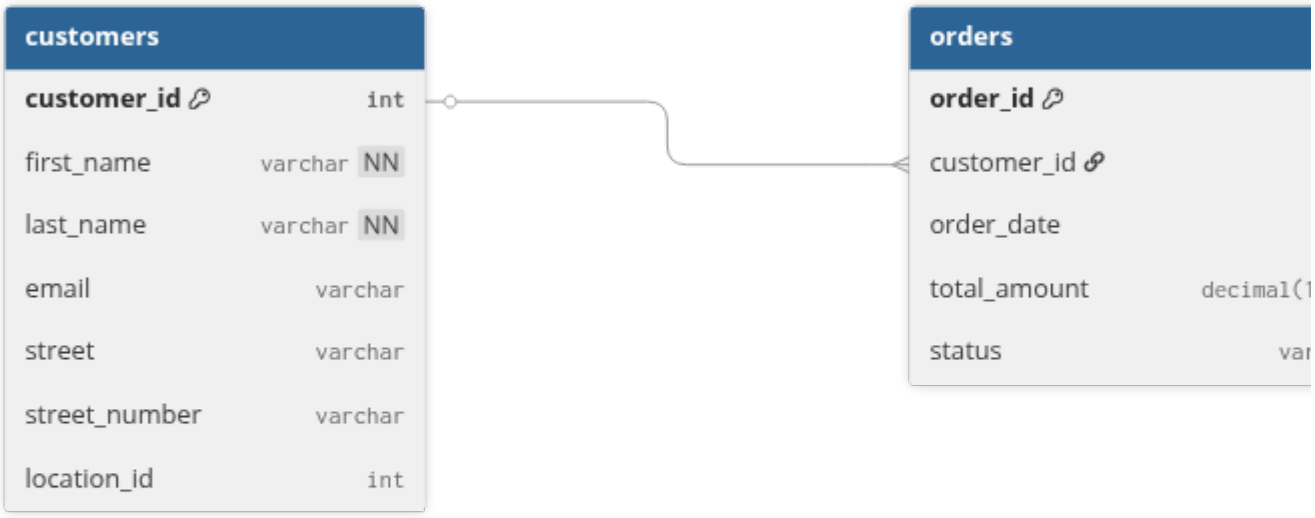
Orders (5 Zeilen)

OID	CID
101	1
102	1
103	2
104	3
105	4

FROM customers, orders → 3 × 5 = 15 Kombinationen!

Jeder Kunde wird mit jeder Bestellung kombiniert – auch wenn die Bestellung gar nicht zu diesem Kunden gehört! Das ist meist nicht das, was wir wollen.

Live-Beispiel: Kartesisches Produkt im Online-Shop



dbdiagram.io

Experiment: Listen Sie Customers und Orders im FROM auf, ohne Bedingung.

```sql
-- VORSICHT: Kartesisches Produkt!
SELECT
  c.customer_id,
  c.first_name,
  o.order_id,
  o.customer_id AS order_customer_id
FROM customers c, orders o
LIMIT 10;  -- Nur erste 10 Zeilen zeigen
```

```
-- VORSICHT: Kartesisches Produkt!
SELECT
  c.customer_id,
  c.first_name,
  o.order_id,
  o.customer_id AS order_customer_id
FROM customers c, orders o
LIMIT 10
```

#	customer_id	first_name	order_id	order_customer_id
1	1	Alice	101	1
2	1	Alice	102	1
3	1	Alice	103	2
4	1	Alice	104	3
5	1	Alice	105	4
6	1	Alice	106	*null*
7	2	Bob	101	1
8	2	Bob	102	1
9	2	Bob	103	2
10	2	Bob	104	3

10 rows

```
-- Nur erste 10 Zeilen zeigen
```

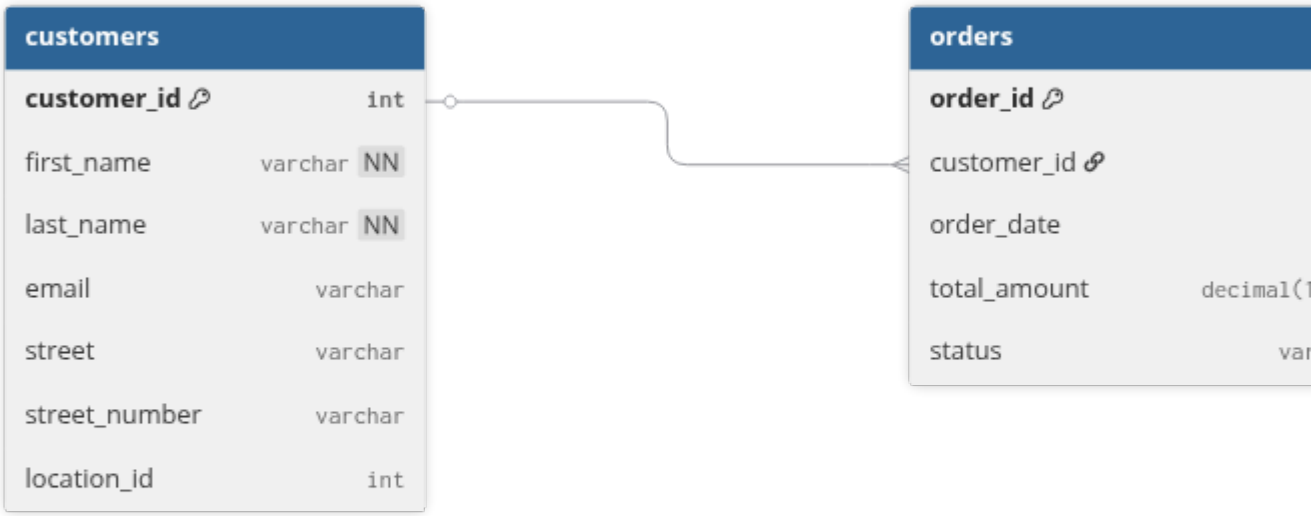
ok

Führen Sie die Query aus. Was sehen Sie? Alice (customer_id = 1) erscheint mit allen Bestellungen – auch mit Bestellungen von Bob und Carol! Das ist das kartesische Produkt: 5 Kunden × 5 Bestellungen = 25 Zeilen.

Problem: Die meisten dieser Kombinationen sind unsinnig! Alice sollte nur mit ihren eigenen Bestellungen verknüpft werden.

Die Lösung: WHERE-Bedingung

Um nur sinnvolle Kombinationen zu bekommen, filtern wir im WHERE: Verbinde nur Zeilen, wo customer_id übereinstimmt.



dbdiagram.io

dbdiagram.io

```
-- TODO: Filtern Sie das kartesische Produkt:
-- Zeigen Sie nur Kunden mit ihren eigenen Bestellungen.
-- Tipp: c.customer_id = o.customer_id
SELECT
  customer_id,
  first_name,
  order_id,
  order_date,
  total_amount
FROM ???
```

```
-- TODO: Filtern Sie das kartesische Produkt:
-- Zeigen Sie nur Kunden mit ihren eigenen Bestellungen.
-- Tipp: c.customer_id = o.customer_id
SELECT
  customer_id,
  first_name,
  order_id,
  order_date,
  total_amount
FROM ???
```

```
syntax error at or near "???"
```

So funktioniert das:

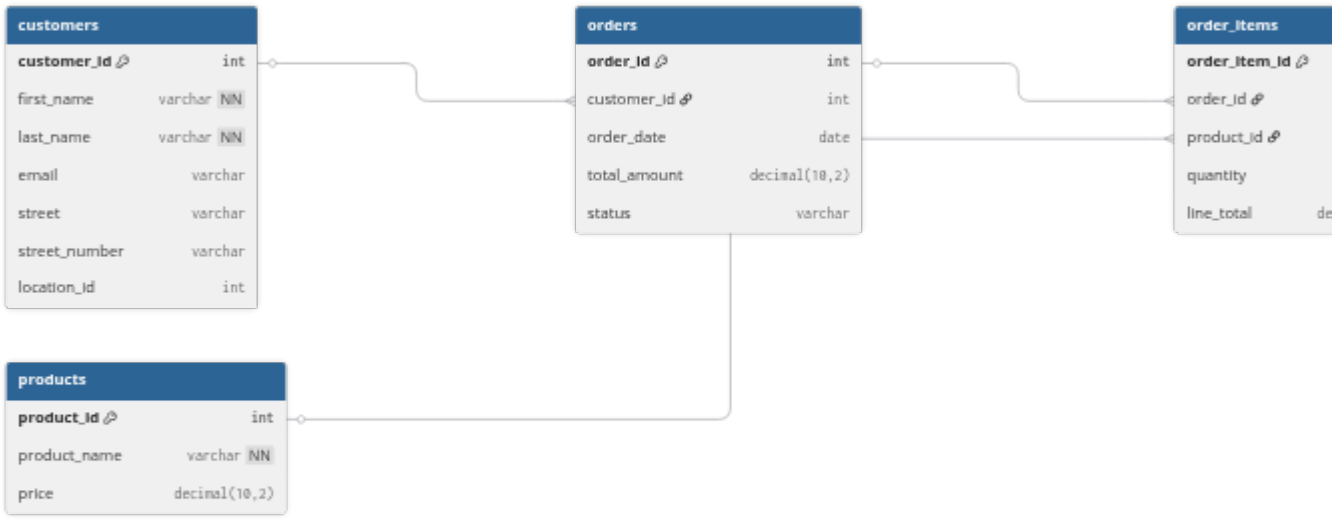
1. SQL erzeugt zunächst das kartesische Produkt (5 × 5 = 25 Zeilen)
2. Dann filtert WHERE: Nur Zeilen, wo customer_id übereinstimmt
3. Ergebnis: Nur 5 Zeilen (Kunde mit seiner Bestellung)

Das ist die klassische Methode, Tabellen zu verbinden – und genau so wurde SQL in den 1980ern geschrieben! Aber diese Syntax hat Nachteile.

Mehrere Tabellen kombinieren

Sie können beliebig viele Tabellen auflisten – aber die WHERE-Bedingungen werden schnell komplex.

Aufgabe: Zeigen Sie Kunde, Bestellung UND Produkt zusammen.



dbdiagram.io

dbdiagram.io

```sql
-- TODO: Verbinden Sie customers, orders, order_items, products
-- Hinweis: Sie brauchen 3 WHERE-Bedingungen!
SELECT
  first_name || ' ' || last_name AS customer,
  order_id,
  product_name,
  quantity
FROM ???
WHERE ???
```

```
-- TODO: Verbinden Sie customers, orders, order_items, products
-- Hinweis: Sie brauchen 3 WHERE-Bedingungen!
SELECT
  first_name || ' ' || last_name AS customer,
  order_id,
  product_name,
  quantity
FROM ???
WHERE ???
```

```
syntax error at or near "???"
```

Das funktioniert, aber:

- Die WHERE-Bedingungen mischen Join-Logik mit Filter-Logik
- Bei 4 Tabellen sind das schon 3 Bedingungen – bei 10 Tabellen?
- Vergessen Sie eine Bedingung → versehentliches kartesisches Produkt!
- Unleserlich: Was ist Join, was ist Filter?

Das Problem mit der impliziten Syntax

Diese Methode wird **implizite Join-Syntax** genannt. Sie hat mehrere Nachteile:

Problem	Beschreibung	Beispiel
Unleserlich	Join-Bedingungen vermischt mit Filter-Bedingungen	`WHERE a.id = b.id AND b.status = 'active'`
Fehleranfällig	Vergessene Bedingung → kartesisches Produkt	`FROM t1, t2, t3 WHERE t1.id = t2.id` (t3 fehlt!)
Kein LEFT/RIGHT	Outer Joins nicht möglich (ohne proprietary Syntax)	Oracle: (`+`) Notation
Veraltet	SQL-92 Standard hat explizite JOIN-Syntax eingeführt	Vor 30+ Jahren!

Deshalb gilt heute: **Nutzen Sie immer die explizite JOIN-Syntax!** Die ist moderner, klarer und mächtiger.

Warum Sie das trotzdem kennen müssen

Warum habe ich Ihnen dann die implizite Syntax gezeigt? Drei Gründe:

1. Legacy-Code verstehen

Viele alte Datenbanken und Anwendungen nutzen diese Syntax noch. Wenn Sie bestehenden Code warten, werden Sie ihr begegnen.

2. Kartesisches Produkt verstehen

Die explizite JOIN-Syntax versteckt, was wirklich passiert. Mit FROM + WHERE sehen Sie: SQL erzeugt zunächst alle Kombinationen, dann filtert es. Das hilft beim Performance-Verständnis.

3. CROSS JOIN erkennen

Wenn Sie versehentlich mehrere Tabellen auflisten ohne JOIN-Bedingung, passiert ein CROSS JOIN (kartesisches Produkt). Sie müssen das erkennen können!

```sql
-- ❌ Versehentlicher CROSS JOIN (häufiger Fehler!):
SELECT * FROM customers, orders;
-- 5 × 5 = 25 Zeilen, meist ungewollt!

-- ✅ Expliziter CROSS JOIN (wenn gewollt):
SELECT * FROM customers CROSS JOIN orders;
```

```
-- ❌ Versehentlicher CROSS JOIN (häufiger Fehler!):
SELECT * FROM customers, orders
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	2	Alice	Anderson	alice@email.com	Unter den Linden
4	3	Alice	Anderson	alice@email.com	Unter den Linden
5	4	Alice	Anderson	alice@email.com	Unter den Linden
6	*null*	Alice	Anderson	alice@email.com	Unter den Linden
7	1	Bob	Brown	bob@email.com	Reeperbahn
8	1	Bob	Brown	bob@email.com	Reeperbahn
9	2	Bob	Brown	bob@email.com	Reeperbahn
10	3	Bob	Brown	bob@email.com	Reeperbahn
11	4	Bob	Brown	bob@email.com	Reeperbahn
12	*null*	Bob	Brown	bob@email.com	Reeperbahn
13	1	Carol	Clark	carol@email.com	Marienplatz
14	1	Carol	Clark	carol@email.com	Marienplatz
15	2	Carol	Clark	carol@email.com	Marienplatz
16	3	Carol	Clark	carol@email.com	Marienplatz
17	4	Carol	Clark	carol@email.com	Marienplatz
18	*null*	Carol	Clark	carol@email.com	Marienplatz
19	1	David	Davis	david@email.com	Hohe Straße
20	1	David	Davis	david@email.com	Hohe Straße
21	2	David	Davis	david@email.com	Hohe Straße
22	3	David	Davis	david@email.com	Hohe Straße
23	4	David	Davis	david@email.com	Hohe Straße

24	*null*	David	Davis	david@email.com	Hohe Straße
25	1	Emma	Evans	emma@email.com	Zeil
26	1	Emma	Evans	emma@email.com	Zeil
27	2	Emma	Evans	emma@email.com	Zeil
28	3	Emma	Evans	emma@email.com	Zeil
29	4	Emma	Evans	emma@email.com	Zeil
30	*null*	Emma	Evans	emma@email.com	Zeil

30 rows

```
-- 5 × 5 = 25 Zeilen, meist ungewollt!

-- ✅ Expliziter CROSS JOIN (wenn gewollt):
SELECT * FROM customers CROSS JOIN orders
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	2	Alice	Anderson	alice@email.com	Unter den Linden
4	3	Alice	Anderson	alice@email.com	Unter den Linden
5	4	Alice	Anderson	alice@email.com	Unter den Linden
6	*null*	Alice	Anderson	alice@email.com	Unter den Linden
7	1	Bob	Brown	bob@email.com	Reeperbahn
8	1	Bob	Brown	bob@email.com	Reeperbahn
9	2	Bob	Brown	bob@email.com	Reeperbahn
10	3	Bob	Brown	bob@email.com	Reeperbahn
11	4	Bob	Brown	bob@email.com	Reeperbahn
12	*null*	Bob	Brown	bob@email.com	Reeperbahn
13	1	Carol	Clark	carol@email.com	Marienplatz
14	1	Carol	Clark	carol@email.com	Marienplatz
15	2	Carol	Clark	carol@email.com	Marienplatz
16	3	Carol	Clark	carol@email.com	Marienplatz
17	4	Carol	Clark	carol@email.com	Marienplatz
18	*null*	Carol	Clark	carol@email.com	Marienplatz
19	1	David	Davis	david@email.com	Hohe Straße
20	1	David	Davis	david@email.com	Hohe Straße
21	2	David	Davis	david@email.com	Hohe Straße
22	3	David	Davis	david@email.com	Hohe Straße
23	4	David	Davis	david@email.com	Hohe Straße

24	*null*	David	Davis	david@email.com	Hohe Straße
25	1	Emma	Evans	emma@email.com	Zeil
26	1	Emma	Evans	emma@email.com	Zeil
27	2	Emma	Evans	emma@email.com	Zeil
28	3	Emma	Evans	emma@email.com	Zeil
29	4	Emma	Evans	emma@email.com	Zeil
30	*null*	Emma	Evans	emma@email.com	Zeil

30 rows

Vergleich: Implizit vs. Explizit

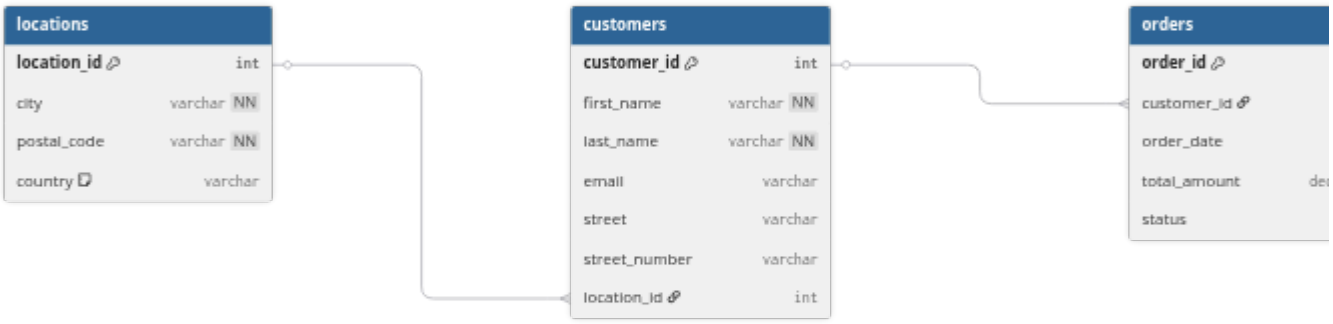
Schauen wir uns beide Syntaxen direkt nebeneinander an – für die gleiche Aufgabe.

Implizite Syntax (veraltet)	Explizite Syntax (modern)
`FROM customers c, orders o`	`FROM customers c INNER JOIN orders o`
`WHERE c.customer_id = o.customer_id`	`ON c.customer_id = o.customer_id`
Filter UND Join vermischt	Join getrennt von Filter
Kein LEFT/RIGHT JOIN möglich	Alle Join-Typen verfügbar
Kartesisches Produkt bei Fehler	Fehler bei fehlender ON-Bedingung

Faustregel: Implizite Syntax = INNER JOIN ohne `ON`. Mehr geht nicht.

Übung: Implizit

Aufgabe: Zeigen Sie Vorname, Stadt und Bestelldatum für alle abgeschlossenen Bestellungen.



dbdiagram.io

dbdiagram.io

```sql
-- Gegeben (implizit):
SELECT
  first_name,
  city,
  order_date
FROM ???
WHERE ???
```

```
-- Gegeben (implizit):
SELECT
  first_name,
  city,
  order_date
FROM ???
WHERE ???
```

```
syntax error at or near "???"
```

Zusammenfassung: FROM mit mehreren Tabellen

Was passiert intern:

1. SQL erzeugt das kartesische Produkt aller Tabellen im FROM
2. WHERE filtert dann die gewünschten Kombinationen
3. Das ist ineffizient – aber so funktioniert die logische Verarbeitung

Warum explizite JOINs besser sind:

- ✅ Klar getrennt: Join-Bedingungen (`ON`) vs. Filter (`WHERE`)
- ✅ Alle Join-Typen verfügbar (`LEFT`, `RIGHT`, `FULL OUTER`)
- ✅ Weniger fehleranfällig (kein versehentliches kartesisches Produkt)
- ✅ Bessere Performance-Optimierung durch Query Planner

> **Best Practice:** Nutzen Sie immer `JOIN ... ON` statt `FROM ..., ... WHERE`!

Jetzt, wo Sie verstehen, was im Hintergrund passiert, schauen wir uns die ersten echten Abfrage-Techniken an: Subqueries!

Technik 1: Subqueries (Verschachtelte SELECT)

Der erste Ansatz, um Daten aus verschiedenen Tabellen zu kombinieren, sind Subqueries – verschachtelte SELECT-Statements. Das fühlt sich natürlich an: „Ich brauche Daten aus Tabelle B, um Tabelle A zu filtern."

Aber was ist eine Subquery genau? Und wo können wir sie überall einsetzen?

Was ist eine Subquery?

Eine Subquery ist ein SELECT-Statement, das innerhalb eines anderen SQL-Statements ausgeführt wird. Statt erst eine Query auszuführen, das Ergebnis zu notieren und dann in einer zweiten Query zu verwenden, verschachteln wir beide.

Konzept:

```sql
-- Ohne Subquery (zwei Schritte):
-- Schritt 1: Welche customer_ids haben Bestellungen?
SELECT DISTINCT customer_id FROM orders;
-- Ergebnis: 1, 2, 3, 4

-- Schritt 2: Zeige diese Kunden
SELECT * FROM customers WHERE customer_id IN (1, 2, 3, 4);
```

```
-- Ohne Subquery (zwei Schritte):
-- Schritt 1: Welche customer_ids haben Bestellungen?
SELECT DISTINCT customer_id FROM orders
```

#	customer_id
1	1
2	2
3	3
4	4
5	*null*

5 rows

```
-- Ergebnis: 1, 2, 3, 4

-- Schritt 2: Zeige diese Kunden
SELECT * FROM customers WHERE customer_id IN (1, 2, 3, 4)
```

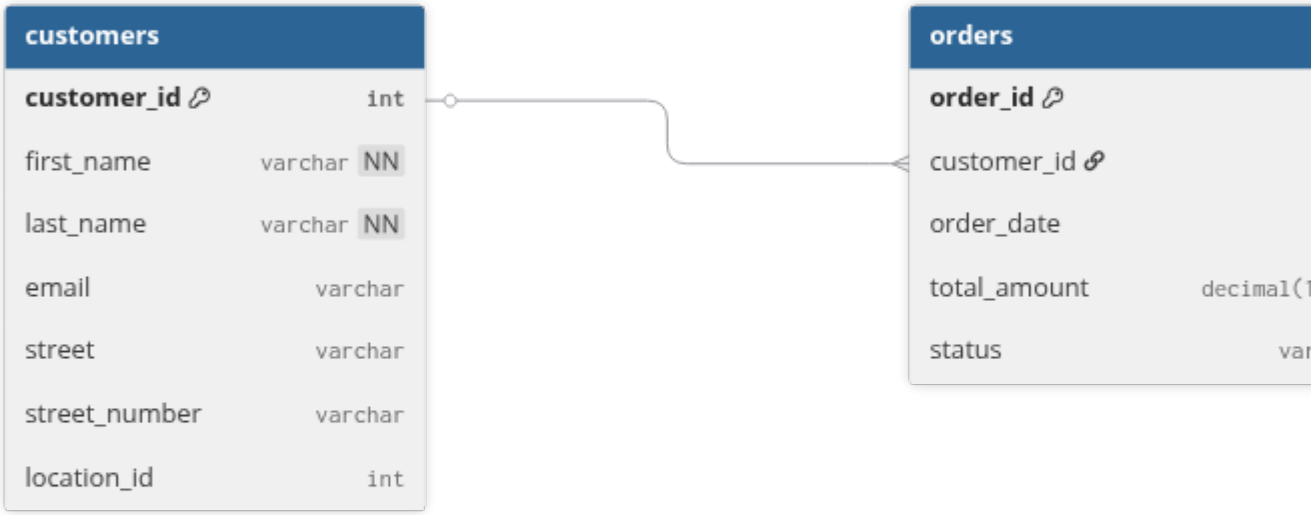
#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	2	Bob	Brown	bob@email.com	Reeperbahn
3	3	Carol	Clark	carol@email.com	Marienplatz
4	4	David	Davis	david@email.com	Hohe Straße

4 rows

Mit Subquery (ein Schritt):

```sql
SELECT * FROM "TABLE"
```

```
WHERE id IN (
  SELECT ... FROM "OTHER TABLE"
);
```



dbdiagram.io

dbdiagram.io

```
1  -- TODO: Schreiben sie die Bestellung um und ermitteln sie alle Kunden
     eine Bestellung haben.
2  SELECT *
3  FROM customers c, orders o
4  WHERE c.customer_id = o.customer_id;
```

```
-- TODO: Schreiben sie die Bestellung um und ermitteln sie alle Kunden,
die eine Bestellung haben.
SELECT *
FROM customers c, orders o
WHERE c.customer_id = o.customer_id
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	2	Bob	Brown	bob@email.com	Reeperbahn
4	3	Carol	Clark	carol@email.com	Marienplatz
5	4	David	Davis	david@email.com	Hohe Straße

5 rows

Die innere Query (Subquery) wird zuerst ausgeführt. Ihr Ergebnis wird dann von der äußeren Query verwendet.

Subquery-Typen: Übersicht

Je nachdem, was eine Subquery zurückgibt, unterscheiden wir verschiedene Typen:

Subquery-Typ	Rückgabewert	Beispiel-Operator	Use Case
Scalar Subquery	Ein einzelner Wert	`=`, `>`, `<`	Durchschnitt, Maximum vergleichen
Row Subquery	Eine Zeile (mehrere Spalten)	`= (col1, col2)`	Selten, Multi-Column-Vergleich
Table Subquery	Mehrere Zeilen, eine Spalte	`IN`, `ANY`, `ALL`	Filtern mit Liste
Derived Table	Mehrere Zeilen/Spalten	Im `FROM`	Komplexe Aggregationen
Correlated	Referenziert äußere Query	Mit Spalte aus äußerer Q.	Pro-Zeile-Berechnung

Schauen wir uns jetzt die wichtigsten dieser Typen im Detail an, beginnend mit dem häufigsten: WHERE Subqueries.

WHERE Subqueries: Filtern mit Ergebnissen aus anderen Tabellen

Die häufigste Form: Eine Subquery im WHERE liefert Werte zum Filtern.

Aufgabe: Zeigen Sie alle Kunden, die mindestens eine Bestellung haben.

```
-- Ohne Subquery (zwei Schritte):
-- Schritt 1: Welche customer_ids haben Bestellungen?
SELECT DISTINCT customer_id FROM orders;
-- Ergebnis: 1, 2, 3, 4

-- Schritt 2: Zeige diese Kunden
SELECT * FROM customers WHERE customer_id IN (1, 2, 3, 4);
```

```
-- Ohne Subquery (zwei Schritte):
-- Schritt 1: Welche customer_ids haben Bestellungen?
SELECT DISTINCT customer_id FROM orders
```

#	customer_id
1	1
2	2
3	3
4	4
5	*null*

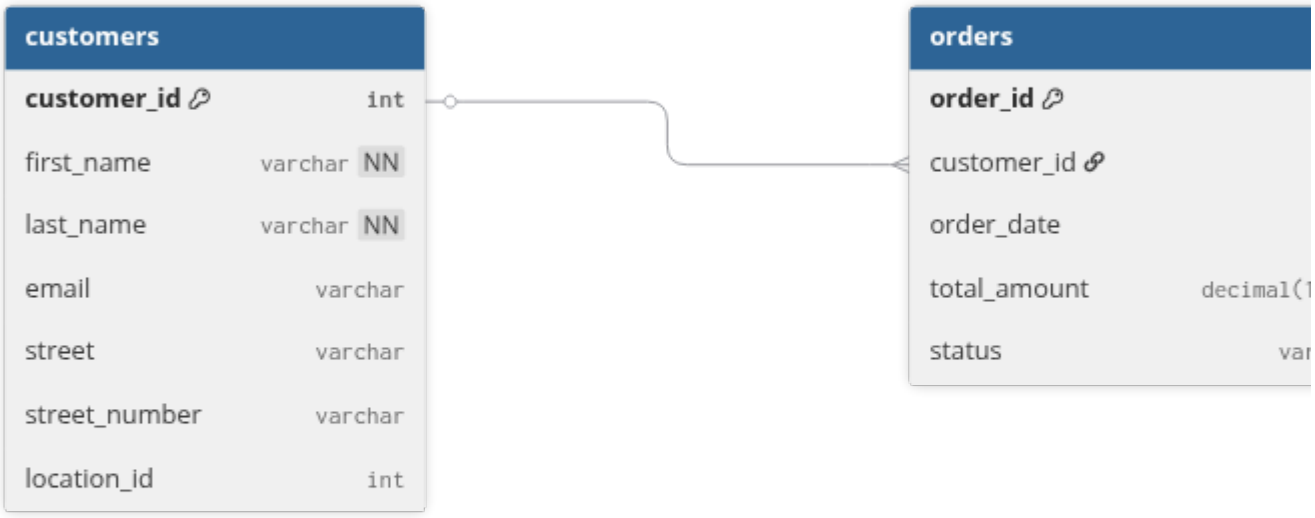
5 rows

```
-- Ergebnis: 1, 2, 3, 4

-- Schritt 2: Zeige diese Kunden
SELECT * FROM customers WHERE customer_id IN (1, 2, 3, 4)
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	2	Bob	Brown	bob@email.com	Reeperbahn
3	3	Carol	Clark	carol@email.com	Marienplatz
4	4	David	Davis	david@email.com	Hohe Straße

4 rows

```sql
SELECT
  column_1,
  column_2,
  ...
FROM table_1
WHERE column_x IN (
  -- Subquery
  SELECT column
  FROM table_n
  WHERE condition
);
```



dbdiagram.io

dbdiagram.io

```sql
-- TODO: Verändern sie die folgende Query, sodass sie eine Subquery im
  nutzt,
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email
FROM customers c, orders o
WHERE c.customer_id = o.customer_id;
```

```
-- TODO: Verändern sie die folgende Query, sodass sie eine Subquery im
WHERE nutzt,
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email
FROM customers c, orders o
WHERE c.customer_id = o.customer_id
```

#	customer_id	first_name	last_name	email
1	1	Alice	Anderson	alice@email.com
2	1	Alice	Anderson	alice@email.com
3	2	Bob	Brown	bob@email.com
4	3	Carol	Clark	carol@email.com
5	4	David	Davis	david@email.com

5 rows

Wie funktioniert das?

1. Die **innere Query** (Subquery) wird zuerst ausgeführt: `SELECT customer_id FROM orders`
2. Ergebnis: Liste von customer_ids, die Bestellungen haben: `(1, 1, 2, 3, 4)`
3. Die **äußere Query** filtert damit: `WHERE customer_id IN (1, 1, 2, 3, 4)`

Das ist einfach zu lesen und zu verstehen. Aber Emma (customer_id = 5) fehlt – die hat keine Bestellung. Subqueries im WHERE sind gut für „zeige mir nur die mit...“

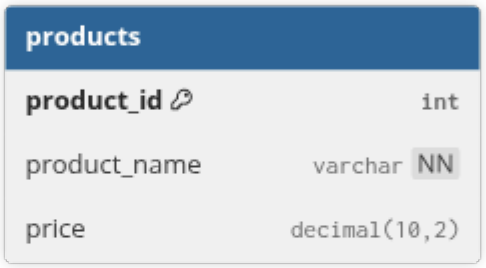
Scalar Subqueries: Einzelwerte berechnen

Eine Subquery kann auch einen einzelnen Wert zurückgeben – zum Vergleichen oder Berechnen.

```
SELECT
    column_a,
    column_b,
    ...,
    (
        -- Subquery: liefert einen Wert (z. B. Durchschnitt)
        SELECT AGG(target_column)
        FROM source_table
    ) AS computed_value
FROM main_table
```

```sql
WHERE filter_column > (
        -- Subquery: derselbe Wert für die WHERE-Bedingung
        SELECT AGG(target_column)
        FROM source_table
    );
```

Aufgabe: Zeigen Sie alle Produkte, die teurer sind als der Durchschnittspreis.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  product_id,
  product_name,
  price,
  ??? AS avg_price
FROM products
WHERE ???;
```

```
SELECT
  product_id,
  product_name,
  price,
  ??? AS avg_price
FROM products
WHERE ???
```

```
syntax error at or near "AS"
```

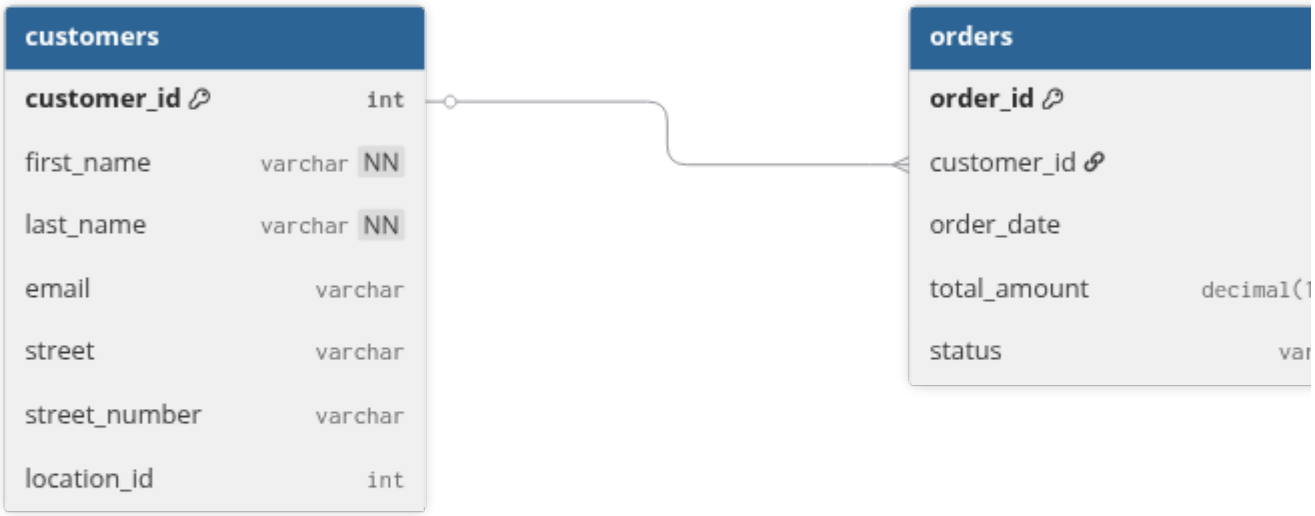
Problem: Wir berechnen den Durchschnitt zweimal! Subquery in SELECT UND in WHERE. Das ist ineffizient und schwer wartbar.
Scalar Subqueries sind nützlich, aber wenn Sie denselben Wert mehrfach brauchen, wird es unübersichtlich. Später sehen wir, wie CTEs dieses Problem lösen.

Subqueries in SELECT: Spalten aus anderen Tabellen

Sie können Subqueries auch nutzen, um zusätzliche Spalten zu berechnen.

```
SELECT
    column_1,
    column_2,
    ...,
    (
        -- Subquery: berechnet einen Wert pro Zeile der äußeren Tabelle
        SELECT AGG(*)
        FROM inner_table
        WHERE inner_table.foreign_key = outer_table.primary_key
    ) AS computed_value
FROM outer_table;
```

Aufgabe: Zeigen Sie für jeden Kunden die Anzahl seiner Bestellungen.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  (???)
FROM customers c;
```

```
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  (???)
FROM customers c
```

```
syntax error at or near ")"
```

Das ist eine correlated Subquery – sie referenziert die äußere Query (`c.customer_id`). Für jeden Kunden wird die Subquery neu ausgeführt.

Funktioniert, aber: Bei 10.000 Kunden wird die Subquery 10.000 Mal ausgeführt! Performance-Problem.

Subquery-Grenzen: Wann wird es problematisch?

Subqueries sind intuitiv, aber sie haben Grenzen:

Problem	Beschreibung	Beispiel
Unleserlich	Verschachtelte Queries sind schwer zu verstehen	3+ Ebenen Verschachtelung
Nicht wiederverwendbar	Berechnete Werte können nicht mehrfach genutzt werden	Durchschnitt 2× berechnen
Performance	Correlated Subqueries werden oft wiederholt ausgeführt	10.000 Kunden = 10.000 Subqueries
Keine parallelen Spalten	Schwierig, Spalten aus mehreren Tabellen parallel zu zeigen	Kunde + Bestellung + Produkt

Es muss einen besseren Weg geben! Und den gibt es: CTEs (Common Table Expressions).

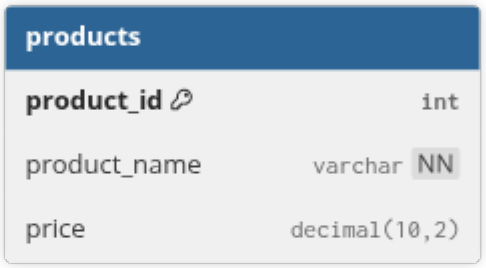
Technik 2: CTEs (WITH) – Benannte Zwischenergebnisse

CTEs sind „benannte Subqueries“. Sie machen Queries lesbarer und wiederverwendbar. Statt alles in einer verschachtelten Monster-Query zu schreiben, teilen Sie es in logische Schritte auf.

CTE-Syntax: WITH … AS

Die Syntax ist einfach: `WITH name AS (SELECT ...)`.

Aufgabe: Durchschnittspreis berechnen und wiederverwenden.



dbdiagram.io

dbdiagram.io

```sql
WITH avg_price_cte AS (
  SELECT AVG(price) AS avg_price FROM products
)

SELECT
  p.product_id,
  p.product_name,
  p.price,
  (SELECT avg_price FROM avg_price_cte) AS avg_price,
  p.price - (SELECT avg_price FROM avg_price_cte) AS difference
FROM products p
WHERE p.price > (SELECT avg_price FROM avg_price_cte);
```

```
WITH avg_price_cte AS (
  SELECT AVG(price) AS avg_price FROM products
)

SELECT
  p.product_id,
  p.product_name,
  p.price,
  (SELECT avg_price FROM avg_price_cte) AS avg_price,
  p.price - (SELECT avg_price FROM avg_price_cte) AS difference
FROM products p
WHERE p.price > (SELECT avg_price FROM avg_price_cte)
```

#	product_id	product_name	price	avg_price	difference
1	1	Laptop	999.99	187.7677777777777778	812.222222
2	4	Monitor	299.99	187.7677777777777778	112.222222
3	5	Desk Chair	199.99	187.7677777777777778	12.2222222

3 rows

Vorteil: Der Durchschnitt wird nur einmal in der CTE berechnet. Die Query ist lesbar: „Was ist avg*price*cte? Schaue am Anfang!“

Multiple CTEs: Logische Schritte

Sie können mehrere CTEs definieren – jede kann auf vorherige zugreifen.

Aufgabe: Finden Sie alle Produkte, die teurer sind als der durchschnittliche Preis in ihrer Kategorie.

```
WITH product_with_categories AS (
  SELECT
    p.product_id,
    p.product_name,
    p.price,
    pc.category_id
  FROM products p, product_categories pc
  WHERE p.product_id = pc.product_id
),
category_avg_prices AS (
  SELECT
    category_id,
    AVG(price) AS avg_price
  FROM product_with_categories
  GROUP BY category_id
)
```

```
)
SELECT
  pwc.product_name,
  pwc.price,
  pwc.category_id,
  (SELECT avg_price FROM category_avg_prices cap WHERE cap.category_i
    pwc.category_id) AS category_avg
FROM product_with_categories pwc
WHERE pwc.price > (SELECT avg_price FROM category_avg_prices cap WHER
  .category_id = pwc.category_id);
```

```
WITH product_with_categories AS (
  SELECT
    p.product_id,
    p.product_name,
    p.price,
    pc.category_id
  FROM products p, product_categories pc
  WHERE p.product_id = pc.product_id
),
category_avg_prices AS (
  SELECT
    category_id,
    AVG(price) AS avg_price
  FROM product_with_categories
  GROUP BY category_id
)
SELECT
  pwc.product_name,
  pwc.price,
  pwc.category_id,
  (SELECT avg_price FROM category_avg_prices cap WHERE cap.category_id
= pwc.category_id) AS category_avg
FROM product_with_categories pwc
WHERE pwc.price > (SELECT avg_price FROM category_avg_prices cap WHERE
cap.category_id = pwc.category_id)
```

#	product_name	price	category_id	category_avg
1	Laptop	999.99	1	284.9900000000000000
2	Laptop	999.99	4	276.6566666666666667
3	Monitor	299.99	1	284.9900000000000000
4	Monitor	299.99	4	276.6566666666666667
5	Desk Chair	199.99	2	124.9900000000000000
6	Notebook	9.99	3	7.4900000000000000

6 rows

Das ist jetzt viel lesbarer!

1. `product_with_categories`: Produkte mit ihren Kategorien verknüpfen
2. `category_avg_prices`: Durchschnittspreis pro Kategorie berechnen
3. Hauptquery: Zeigt Produkte, die teurer als der Kategorie-Durchschnitt sind

Jeder Schritt ist klar benannt. Die Logik ist in kleine, verständliche Blöcke aufgeteilt.

CTEs vs. Subqueries: Wann was?

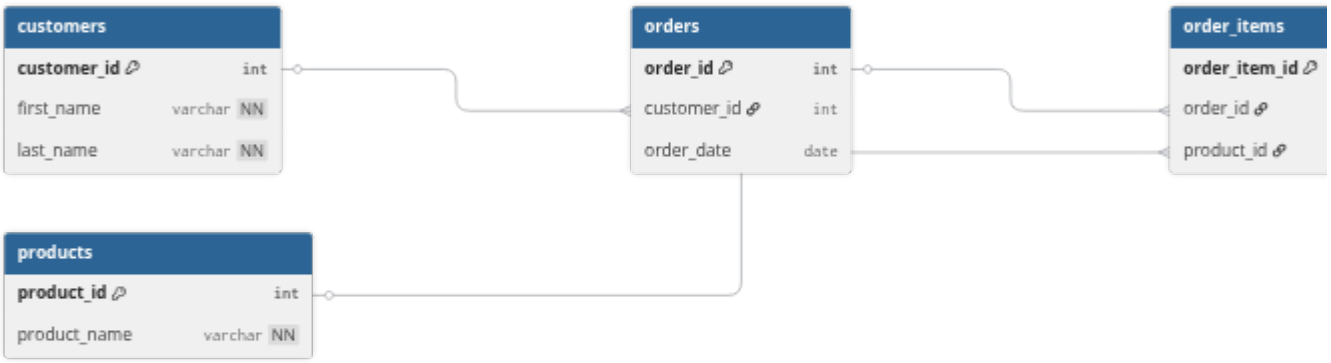
Kriterium	Subqueries	CTEs
Lesbarkeit	Schlecht bei Verschachtelung	Gut (logische Schritte)
Wiederverwendung	Nein	Ja (mehrfach referenzierbar)
Performance	Identisch	Identisch (meist)
Komplexität	Einfache Fälle ok	Komplexe Queries besser

> **Faustregel:** Bei mehr als einer Verschachtelungsebene → nutzen Sie CTEs!

CTEs: Die Grenze

CTEs sind großartig für komplexe Berechnungen und schrittweise Aggregationen, aber sie haben eine Einschränkung: Das **parallele Zusammenführen von Spalten aus mehreren Tabellen** wird schnell unübersichtlich.

Problem: Zeigen Sie für jede Bestellung den Kundennamen UND die bestellten Produkte.



dbdiagram.io

dbdiagram.io

```sql
-- Mit CTE und Subqueries: Umständlich!
WITH order_data AS (
  SELECT
    o.order_id,
    o.order_date,
    o.customer_id
  FROM orders o
)
SELECT
  od.order_id,
  od.order_date,
  (SELECT c.first_name || ' ' || c.last_name
   FROM customers c
   WHERE c.customer_id = od.customer_id) AS customer_name,
  (SELECT p.product_name
   FROM order_items oi, products p
   WHERE oi.order_id = od.order_id
```

```
    WHERE oi.order_id = od.order_id
      AND oi.product_id = p.product_id
    LIMIT 1) AS first_product
FROM order_data od;
```

```
-- Mit CTE und Subqueries: Umständlich!
WITH order_data AS (
  SELECT
    o.order_id,
    o.order_date,
    o.customer_id
  FROM orders o
)
SELECT
  od.order_id,
  od.order_date,
  (SELECT c.first_name || ' ' || c.last_name
   FROM customers c
   WHERE c.customer_id = od.customer_id) AS customer_name,
  (SELECT p.product_name
   FROM order_items oi, products p
   WHERE oi.order_id = od.order_id
     AND oi.product_id = p.product_id
   LIMIT 1) AS first_product
FROM order_data od
```

#	order_id	order_date	customer_name	first_product
1	101	2024-01-15	Alice Anderson	Monitor
2	102	2024-02-20	Alice Anderson	Mouse
3	103	2024-01-22	Bob Brown	Laptop
4	104	2024-03-10	Carol Clark	Notebook
5	105	2024-03-15	David Davis	Desk Chair
6	106	2023-03-20	*null*	USB Cable

6 rows

Problem mit diesem Ansatz:

- Mehrere verschachtelte Subqueries – schwer zu lesen
- Zeigt nur das *erste* Produkt pro Bestellung (LIMIT 1)
- Performance: Subqueries werden für jede Zeile neu ausgeführt
- Wenn eine Bestellung mehrere Produkte hat, fehlen diese

CTEs helfen bei Komplexität und schrittweisen Berechnungen, aber für das **elegante Zusammenführen von Daten aus mehreren Tabellen** brauchen wir ein besseres Werkzeug: Joins!

Zeit für Technik 3: Joins – die Lösung für genau dieses Problem!

Technik 3: Joins – Die elegante Lösung

Joins sind das Werkzeug, um Spalten aus mehreren Tabellen **parallel** in einer Zeile zusammenzuführen. Statt verschachtelt zu denken (Subqueries) oder in Schritten (CTEs), denken Sie horizontal: „Füge Tabellen nebeneinander zusammen."

Ein Join ist wie ein Reißverschluss: Sie haben zwei Listen und verbinden passende Einträge. Kunden und ihre Bestellungen. Produkte und ihre Kategorien. Das Ergebnis? Eine Zeile mit Informationen aus beiden Tabellen.

Aber es gibt verschiedene Arten von Joins – je nachdem, was Sie mit nicht-passenden Einträgen machen wollen. Schauen wir uns die wichtigsten an.

Die JOIN-Syntax

Moderne Joins nutzen das Schlüsselwort `JOIN` mit einer `ON`-Bedingung. Das trennt die Join-Logik sauber vom WHERE-Filter.

```
SELECT
  spalten_aus_tabelle_a,
  spalten_aus_tabelle_b
FROM tabelle_a
JOIN_TYP tabelle_b ON tabelle_a.key = tabelle_b.key
WHERE weitere_filter;
```

Bestandteile:

- `JOIN_TYP`: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, CROSS JOIN
- `ON`: Die Bedingung, wie Zeilen zusammenpassen (meist Foreign Key = Primary Key)
- `WHERE`: Zusätzliche Filter (optional, nach dem Join)

Wichtig: ON definiert die Beziehung, WHERE filtert das Ergebnis. Das nicht zu verwechseln macht Queries klar und wartbar!

Überblick: Die 5 Join-Typen

Es gibt fünf Haupt-Join-Typen. Jeder beantwortet eine andere Frage.

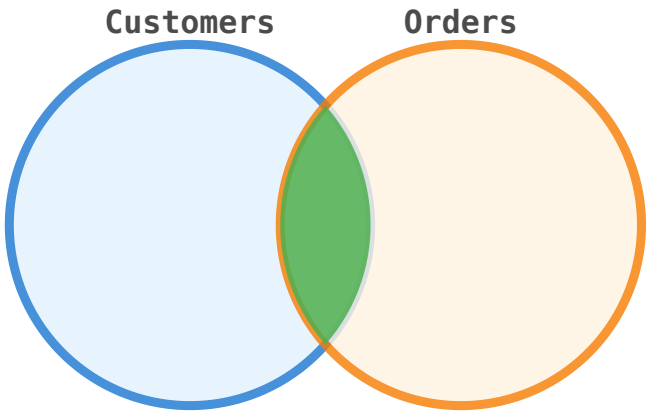
Join-Typ	Frage	Wann nutzen?
INNER JOIN	Zeige nur Einträge, die in beiden Tabellen existieren	Standard-Fall, nur Matches wichtig
LEFT JOIN	Zeige alle aus Tabelle A, auch ohne Match in B	„Wer hat KEINE Bestellung?“
RIGHT JOIN	Zeige alle aus Tabelle B, auch ohne Match in A	Selten (meist LEFT stattdessen)
FULL OUTER JOIN	Zeige alles aus beiden Tabellen	Vergleiche, Sync-Checks
CROSS JOIN	Zeige alle Kombinationen (kartesisches Produkt)	Test-Kombinationen, Kalender

In der Praxis machen INNER JOIN und LEFT JOIN etwa 95% aller Joins aus. Die anderen sind Spezialfälle. Beginnen wir mit dem häufigsten: INNER JOIN.

INNER JOIN: Nur die Matches

INNER JOIN ist der Standard-Join. Er gibt nur Zeilen zurück, bei denen es in **beiden** Tabellen einen passenden Eintrag gibt.

Visualisierung: Venn-Diagramm



Denken Sie an die Überschneidung zweier Kreise: Nur der grüne Bereich (wo sich beide überlappen) kommt ins Ergebnis. Alles andere wird ignoriert.

Konzept: Wie funktioniert INNER JOIN?

Stellen Sie sich zwei Listen vor:

Customers

ID	Name
1	Alice
2	Bob
3	Carol
5	Emma

Orders

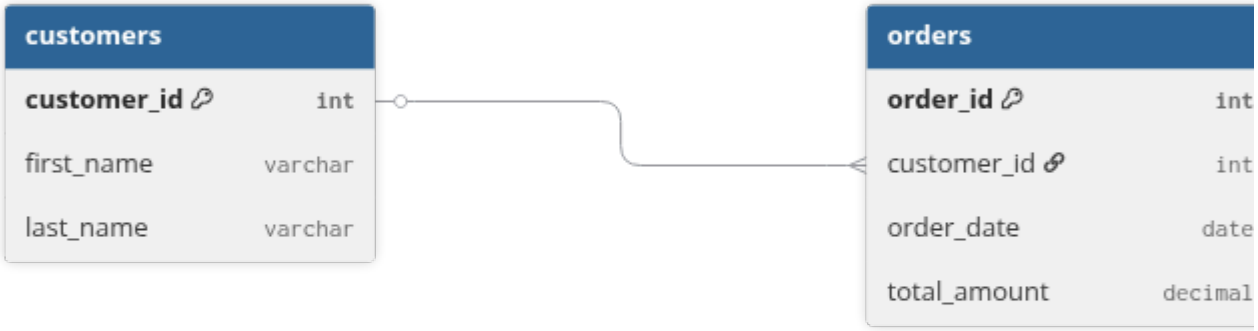
OID	customer_id	
101	1	← Passt zu Alice
102	1	← Passt zu Alice
103	2	← Passt zu Bob
105	4	← Passt zu David

```
INNER JOIN ON customer_id:
Alice — Order 101 ✓
Alice — Order 102 ✓
Bob   — Order 103 ✓
David — Order 105 ✓

Emma? Hat keine Bestellung → kommt NICHT ins Ergebnis!
```

SQL geht beide Tabellen durch und verbindet nur Zeilen, wo die customer_id übereinstimmt. Emma hat keine Bestellung, also keine Übereinstimmung, also kein Ergebnis.

Beispiel 1: Kunden mit ihren Bestellungen

Zeigen Sie jeden Kunden zusammen mit seinen Bestellungen.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  c.first_name,
  c.last_name,
  o.order_id,
  o.order_date
FROM customers c, orders o
WHERE c.customer_id = o.customer_id
ORDER BY c.last_name, o.order_date;
```

```
SELECT
  c.first_name,
  c.last_name,
  o.order_id,
  o.order_date
FROM customers c, orders o
WHERE c.customer_id = o.customer_id
ORDER BY c.last_name, o.order_date
```

#	first_name	last_name	order_id	order_date
1	Alice	Anderson	101	2024-01-15
2	Alice	Anderson	102	2024-02-20
3	Bob	Brown	103	2024-01-22
4	Carol	Clark	104	2024-03-10
5	David	Davis	105	2024-03-15

5 rows

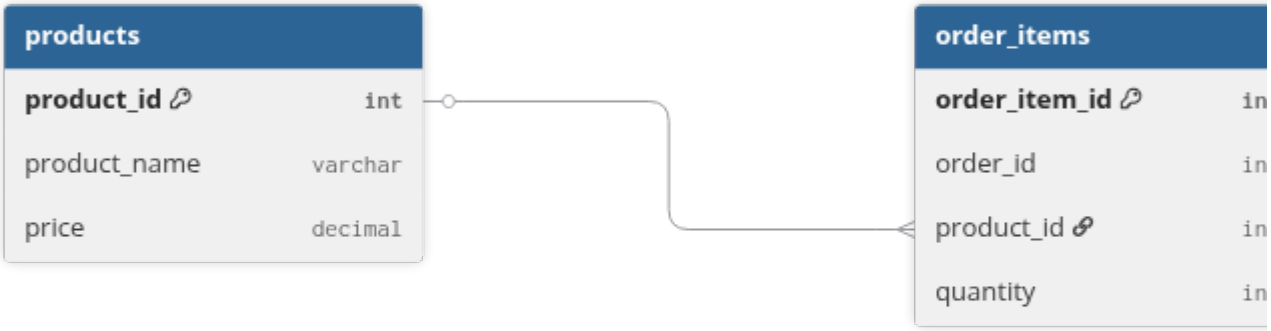
Was passiert hier?

1. SQL nimmt jeden Kunden aus `customers`
2. Sucht alle passenden Bestellungen in `orders` (wo `customer_id` übereinstimmt)
3. Erstellt eine Zeile pro Match: Kunde + Bestellung
4. Emma hat keine Bestellung → erscheint nicht

Führen Sie die Query aus. Sie sehen: Alice erscheint zweimal (hat zwei Bestellungen), Emma fehlt komplett.

Beispiel 2: Bestellungen mit Produktnamen

Zeigen Sie für jede Bestellposition das Produkt mit Namen.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  oi.order_id,
  oi.quantity
FROM order_items oi
```

```
SELECT
  oi.order_id,
  oi.quantity
FROM order_items oi
```

#	order_id	quantity
1	101	1
2	102	2
3	102	1
4	103	1
5	104	5
6	105	1
7	106	5

7 rows

Was sehen Sie?

- Order 101: Monitor
- Order 102: Mouse (2×) + Keyboard
- Order 103: Laptop
- …
- Jede Zeile kombiniert Bestellposition mit Produktdetails

Das ist die Essenz von Joins: Informationen aus verschiedenen Tabellen landen in einer Zeile. Praktisch!

Wann INNER JOIN nutzen?

INNER JOIN ist Ihre Standard-Wahl, wenn Sie nur an **existierenden Beziehungen** interessiert sind.

Typische Anwendungsfälle:

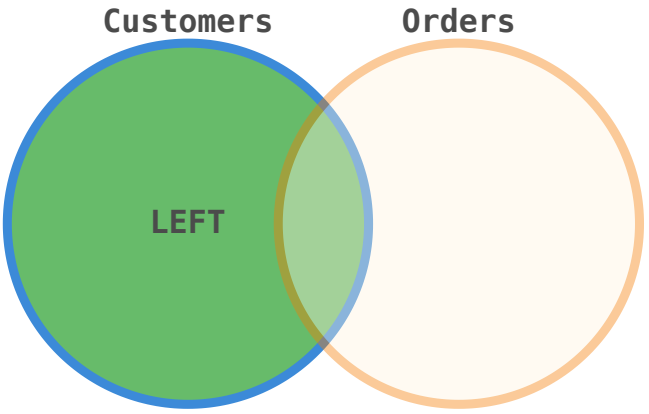
- Bestellungen mit Kundendaten anzeigen (nur abgeschlossene Bestellungen)
- Produkte mit Kategorien (nur kategorisierte Produkte)
- Rechnungen mit Zahlungen (nur bezahlte Rechnungen)
- Log-Einträge mit User-Details (nur bekannte User)

Faustregel: INNER JOIN = „Zeige mir nur, wo beides existiert“

LEFT JOIN: Alle von links + Matches

LEFT JOIN (auch LEFT OUTER JOIN genannt) gibt **alle Zeilen der linken Tabelle** zurück – auch wenn es rechts keinen Match gibt. Fehlende Matches werden mit NULL aufgefüllt.

Visualisierung: Venn-Diagramm



Alle Werte aus der linken Tabelle (Customer) + deren Matches (LEFT JOIN)

Der komplette linke Kreis ist grün – das bedeutet: ALLE Einträge aus der linken Tabelle kommen ins Ergebnis, egal ob es rechts einen Match gibt.

Konzept: Wie funktioniert LEFT JOIN?

LEFT JOIN behält alle Zeilen der linken Tabelle und fügt passende Daten von rechts hinzu – oder NULL, wenn nichts passt.

Customers (links)

ID	Name
1	Alice
2	Bob
3	Carol
4	David
5	Emma

Orders (rechts)

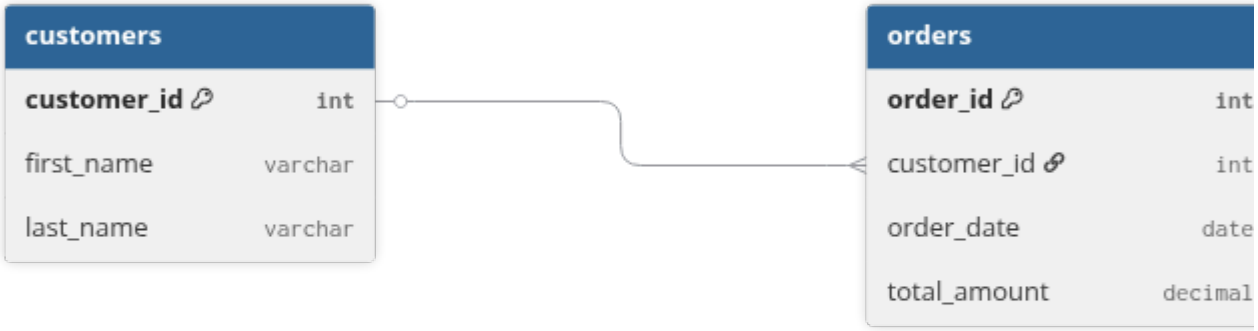
OID	customer_id	
101	1	← Match
102	1	← Match
103	2	← Match
105	4	← Match
(keine Bestellung)		

```
LEFT JOIN ON customer_id:
Alice — Order 101 ✓
Alice — Order 102 ✓
Bob   — Order 103 ✓
David — Order 105 ✓
Emma  — NULL      ← Emma bleibt im Ergebnis, aber Order-Felder sind NULL!
```

Das ist der Schlüssel: Die linke Tabelle bestimmt, welche Zeilen im Ergebnis erscheinen. Die rechte Tabelle ergänzt nur.

Beispiel 1: Alle Kunden (auch ohne Bestellungen)

Zeigen Sie ALLE Kunden – egal ob sie bestellt haben oder nicht.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  first_name,
  last_name
FROM customers c
-- JOIN
ORDER BY last_name;
```

```
SELECT
  first_name,
  last_name
FROM customers c
-- JOIN
ORDER BY last_name
```

#	first_name	last_name
1	Alice	Anderson
2	Bob	Brown
3	Carol	Clark
4	David	Davis
5	Emma	Evans

5 rows

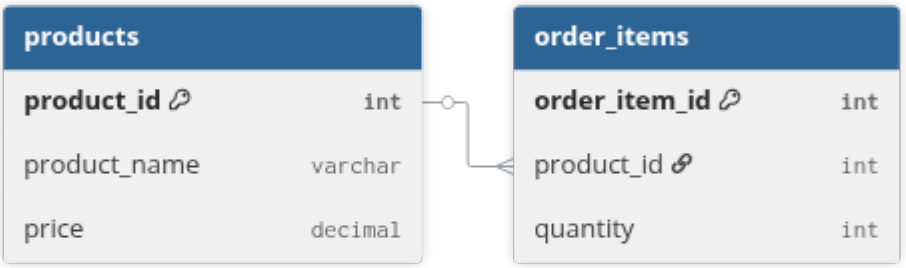
Was sehen Sie?

- Alice, Bob, Carol, David: Jeweils mit ihren Bestellungen
- Emma: Erscheint auch! Aber `order_id`, `order_date`, `total_amount` sind NULL

Das ist der Unterschied zu INNER JOIN: Emma wird nicht ignoriert. Links bestimmt das Ergebnis!

Beispiel 2: Produkte mit Verkaufszahlen (auch unverkaufte)

Zeigen Sie alle Produkte – auch die, die noch nie verkauft wurden.



dbdiagram.io

dbdiagram.io

```sql
SELECT
  p.product_name,
  p.price,
FROM products p
```

```
SELECT
  p.product_name,
  p.price,
FROM products p
syntax error at or near "FROM"
```

Was passiert hier?

- Produkte mit Verkäufen: `times_sold` > 0
- Unverkaufte Produkte: `times_sold` = 0 (COUNT zählt NULL als 0)

LEFT JOIN ermöglicht es, fehlende Beziehungen zu finden. Das ist extrem wertvoll für Analysen!

Wann LEFT JOIN nutzen?

LEFT JOIN ist perfekt, wenn Sie **fehlende Beziehungen** identifizieren wollen.

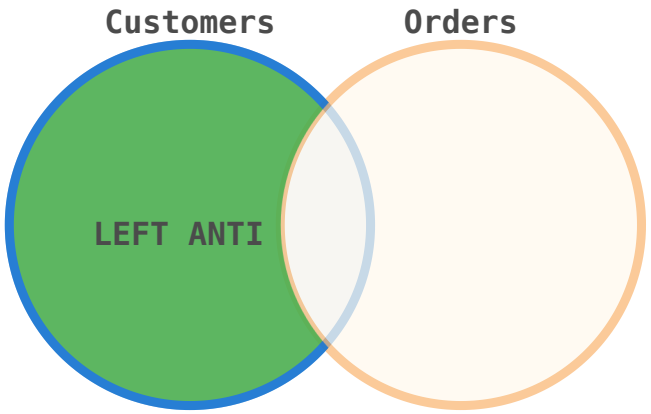
Typische Anwendungsfälle:

- Kunden ohne Bestellungen (Inaktive finden)
- Produkte ohne Verkäufe (Ladenhüter)
- Artikel ohne Übersetzungen (Content-Lücken)
- Rechnungen ohne Zahlung (Offene Posten)

> **Faustregel:** LEFT JOIN = „Zeige alle von links, ergänze rechts wenn möglich“

Anti-Join: Fehlende finden mit IS NULL

Eine mächtige Technik: LEFT JOIN + WHERE IS NULL = „Zeige nur die OHNE Match“



Nur Customers, die KEINE passenden Orders haben (LEFT ANTI JOIN)

Frage: Welche Kunden haben noch NIE bestellt?

dbdiagram.io

dbdiagram.io

```sql
SELECT
  c.customer_id,
  c.first_name,
  c.last_name
FROM customers c
```

```
SELECT
  c.customer_id,
  c.first_name,
  c.last_name
FROM customers c
```

#	customer_id	first_name	last_name
1	1	Alice	Anderson
2	2	Bob	Brown
3	3	Carol	Clark
4	4	David	Davis
5	5	Emma	Evans

5 rows

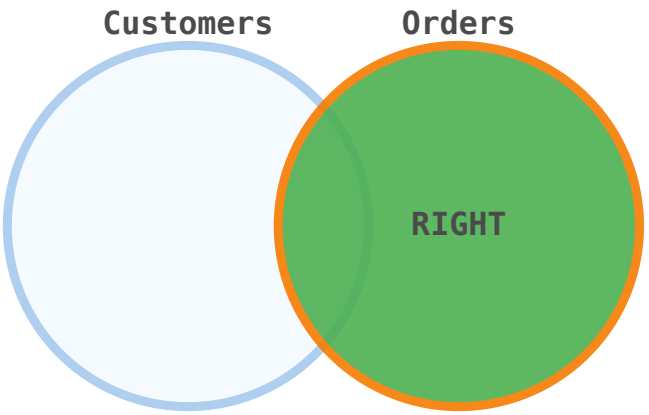
Trick: Nach dem LEFT JOIN filtern Sie auf NULL in der rechten Tabelle. Das sind exakt die Zeilen ohne Match!

Diese Technik heißt „Anti-Join“ und ist in der Praxis extrem häufig. Sie finden damit Lücken in Ihren Daten.

RIGHT JOIN: Alle von rechts + Matches

RIGHT JOIN ist das Spiegelbild von LEFT JOIN: Alle Zeilen der **rechten** Tabelle bleiben erhalten, links wird ergänzt.

Visualisierung: Venn-Diagramm



Alle Werte aus Orders + deren Matches (RIGHT JOIN)

Der komplette rechte Kreis ist grün. Alle Bestellungen kommen ins Ergebnis – auch wenn der Kunde unbekannt ist (was eigentlich nicht passieren sollte, aber theoretisch möglich ist).

In der Praxis: Selten genutzt

RIGHT JOIN wird in der Praxis kaum verwendet. Warum? Weil Sie fast immer eine LEFT JOIN Alternative schreiben können, die leichter zu verstehen ist.

```
1  -- RIGHT JOIN:
2  SELECT * FROM customers c
3  RIGHT JOIN orders o ON c.customer_id = o.customer_id;
4
5  -- Definieren Sie das gleichbedeutende LEFT JOIN:
```

```
-- RIGHT JOIN:
SELECT * FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	2	Bob	Brown	bob@email.com	Reeperbahn
4	3	Carol	Clark	carol@email.com	Marienplatz
5	4	David	Davis	david@email.com	Hohe Straße
6	*null*	*null*	*null*	*null*	*null*

6 rows

```
-- Definieren Sie das gleichbedeutende LEFT JOIN:
```

ok

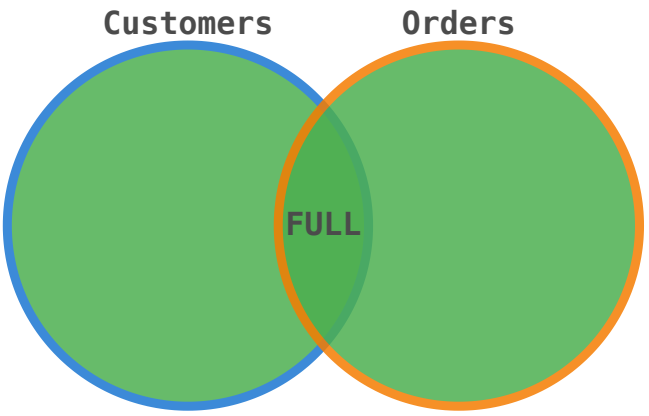
Beide Queries liefern identische Ergebnisse! Die zweite ist aber intuitiver: Links ist die Haupttabelle.

Mein Rat: Vermeiden Sie RIGHT JOIN. Schreiben Sie stattdessen LEFT JOIN mit vertauschter Reihenfolge. Das ist Standard in den meisten Teams.

FULL OUTER JOIN: Alles aus beiden Tabellen

FULL OUTER JOIN (oder nur FULL JOIN) kombiniert LEFT und RIGHT JOIN: Alle Zeilen aus **beiden** Tabellen kommen ins Ergebnis. Matches werden verbunden, fehlende Matches mit NULL aufgefüllt.

Visualisierung: Venn-Diagramm



Alle Werte aus Customers und Orders (FULL JOIN)

Beide Kreise sind komplett grün. Jede Zeile aus jeder Tabelle erscheint mindestens einmal – entweder mit Match oder mit NULLs.

Konzept: Die vollständige Vereinigung

FULL OUTER JOIN ist wie: „Zeige mir alles – Matches, Nur-Links, Nur-Rechts."

Customers

1	Alice
2	Bob
5	Emma

Orders

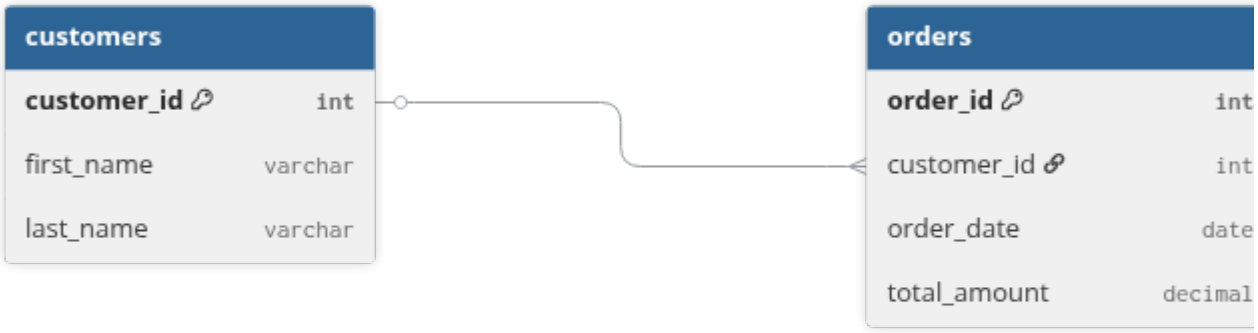
101	1	← Match mit Alice
102	1	← Match mit Alice
106	NULL	← Kunde wurde gelöscht!

```
FULL OUTER JOIN:
Alice – Order 101 ✓
Alice – Order 102 ✓
Bob   – NULL      ✓ (Bob hat keine Bestellung)
Emma  – NULL      ✓ (Emma hat keine Bestellung)
NULL  – Order 106 ✓ (Bestellung hat ungültigen Kunden)
```

Sie sehen: Sowohl Emma (Kunde ohne Bestellung) als auch Order 106 (Bestellung ohne Kunden) erscheinen im Ergebnis. Nichts geht verloren!

Beispiel 1: Vollständiger Datenabgleich

Zeigen Sie ALLE Kunden und ALLE Bestellungen – auch wenn Kunden keine Bestellung haben ODER Bestellungen keinen gültigen Kunden haben.



dbdiagram.io

```sql
SELECT
  c.customer_id,
  c.first_name || ' ' || c.last_name AS customer_name,
--  o.order_id,
--  o.order_date,
--  o.total_amount
FROM customers c
```

```
SELECT
  c.customer_id,
  c.first_name || ' ' || c.last_name AS customer_name,
--  o.order_id,
--  o.order_date,
--  o.total_amount
FROM customers c
```

```
syntax error at or near "FROM"
```

Was sehen Sie?

- Emma (customer_id = 5): Erscheint mit NULL bei Bestellungen → Kunde ohne Bestellung
- Order 106: Erscheint mit NULL bei Kundendaten → Bestellung ohne gültigen Kunden (customer_id = 99 existiert nicht!)
- Alle anderen: Normale Matches

FULL OUTER JOIN ist perfekt für Datenqualitäts-Checks: „Zeige mir ALLES, damit ich Inkonsistenzen erkenne." Hier sehen wir beide Probleme: Emma hat nicht bestellt UND Order 106 hat einen ungültigen Kunden.

Wann FULL OUTER JOIN nutzen?

FULL OUTER JOIN ist selten, aber für spezielle Aufgaben perfekt.

Typische Anwendungsfälle:

- Datenbank-Sync prüfen (Quelle vs. Ziel)
- Inkonsistenzen finden (Orphaned Records auf beiden Seiten)
- Audit-Reports (vollständige Übersicht)

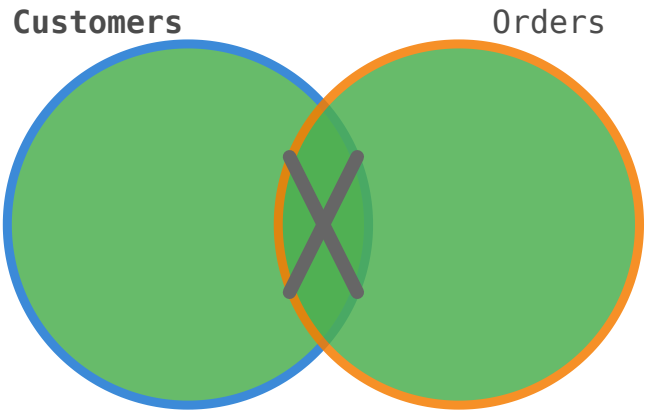
> **Faustregel:** FULL OUTER JOIN = „Zeige alles aus beiden Welten"

In der Praxis wird FULL OUTER JOIN selten genutzt – oft kann man das Problem mit zwei LEFT JOINs + UNION lösen. Aber wenn Sie ihn brauchen, ist er unschlagbar praktisch!

CROSS JOIN: Alle Kombinationen (Kartesisches Produkt)

CROSS JOIN ist der ungewöhnlichste Join: Er verbindet **jede Zeile** der ersten Tabelle mit **jeder Zeile** der zweiten Tabelle. Keine Bedingung, keine Filter – alle Kombinationen.

Visualisierung: Venn-Diagramm



Kartesisches Produkt: jede Zeile × jede Zeile

Die Kreise überlappen nicht – weil CROSS JOIN keine Beziehung braucht. Er erzeugt einfach alle Kombinationen. Das nennt man kartesisches Produkt.

Konzept: Alle Kombinationen

CROSS JOIN ist wie eine Tabelle mit allen möglichen Paarungen erstellen.

Sizes

Size
S
M
L

Colors

Color
Red
Blue
Green

CROSS JOIN → 3 × 3 = 9 Kombinationen:

S – Red
S – Blue
S – Green
M – Red
M – Blue
M – Green
L – Red
L – Blue
L – Green

Jede Größe wird mit jeder Farbe kombiniert. Kein Filter, keine Bedingung – einfach alle Möglichkeiten.

Syntax: Zwei Varianten

CROSS JOIN kann explizit oder implizit geschrieben werden.

```sql
-- Explizit (empfohlen):
SELECT * FROM customers CROSS JOIN products;

-- Implizit (veraltet):
SELECT * FROM customers, products;
```

```
-- Explizit (empfohlen):
SELECT * FROM customers CROSS JOIN products
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	1	Alice	Anderson	alice@email.com	Unter den Linden
4	1	Alice	Anderson	alice@email.com	Unter den Linden
5	1	Alice	Anderson	alice@email.com	Unter den Linden
6	1	Alice	Anderson	alice@email.com	Unter den Linden
7	1	Alice	Anderson	alice@email.com	Unter den Linden
8	1	Alice	Anderson	alice@email.com	Unter den Linden
9	1	Alice	Anderson	alice@email.com	Unter den Linden
10	2	Bob	Brown	bob@email.com	Reeperbahn
11	2	Bob	Brown	bob@email.com	Reeperbahn
12	2	Bob	Brown	bob@email.com	Reeperbahn
13	2	Bob	Brown	bob@email.com	Reeperbahn
14	2	Bob	Brown	bob@email.com	Reeperbahn
15	2	Bob	Brown	bob@email.com	Reeperbahn
16	2	Bob	Brown	bob@email.com	Reeperbahn
17	2	Bob	Brown	bob@email.com	Reeperbahn
18	2	Bob	Brown	bob@email.com	Reeperbahn
19	3	Carol	Clark	carol@email.com	Marienplatz
20	3	Carol	Clark	carol@email.com	Marienplatz
21	3	Carol	Clark	carol@email.com	Marienplatz

22	3	Carol	Clark	carol@email.com	Marienplatz
23	3	Carol	Clark	carol@email.com	Marienplatz
24	3	Carol	Clark	carol@email.com	Marienplatz
25	3	Carol	Clark	carol@email.com	Marienplatz
26	3	Carol	Clark	carol@email.com	Marienplatz
27	3	Carol	Clark	carol@email.com	Marienplatz
28	4	David	Davis	david@email.com	Hohe Straße
29	4	David	Davis	david@email.com	Hohe Straße
30	4	David	Davis	david@email.com	Hohe Straße
31	4	David	Davis	david@email.com	Hohe Straße
32	4	David	Davis	david@email.com	Hohe Straße
33	4	David	Davis	david@email.com	Hohe Straße
34	4	David	Davis	david@email.com	Hohe Straße
35	4	David	Davis	david@email.com	Hohe Straße
36	4	David	Davis	david@email.com	Hohe Straße
37	5	Emma	Evans	emma@email.com	Zeil
38	5	Emma	Evans	emma@email.com	Zeil
39	5	Emma	Evans	emma@email.com	Zeil
40	5	Emma	Evans	emma@email.com	Zeil
41	5	Emma	Evans	emma@email.com	Zeil
42	5	Emma	Evans	emma@email.com	Zeil
43	5	Emma	Evans	emma@email.com	Zeil
44	5	Emma	Evans	emma@email.com	Zeil
45	5	Emma	Evans	emma@email.com	Zeil

45 rows

```
-- Implizit (veraltet):
SELECT * FROM customers, products
```

#	customer_id	first_name	last_name	email	street
1	1	Alice	Anderson	alice@email.com	Unter den Linden
2	1	Alice	Anderson	alice@email.com	Unter den Linden
3	1	Alice	Anderson	alice@email.com	Unter den Linden
4	1	Alice	Anderson	alice@email.com	Unter den Linden
5	1	Alice	Anderson	alice@email.com	Unter den Linden
6	1	Alice	Anderson	alice@email.com	Unter den Linden
7	1	Alice	Anderson	alice@email.com	Unter den Linden
8	1	Alice	Anderson	alice@email.com	Unter den Linden
9	1	Alice	Anderson	alice@email.com	Unter den Linden
10	2	Bob	Brown	bob@email.com	Reeperbahn
11	2	Bob	Brown	bob@email.com	Reeperbahn
12	2	Bob	Brown	bob@email.com	Reeperbahn
13	2	Bob	Brown	bob@email.com	Reeperbahn
14	2	Bob	Brown	bob@email.com	Reeperbahn
15	2	Bob	Brown	bob@email.com	Reeperbahn
16	2	Bob	Brown	bob@email.com	Reeperbahn
17	2	Bob	Brown	bob@email.com	Reeperbahn
18	2	Bob	Brown	bob@email.com	Reeperbahn
19	3	Carol	Clark	carol@email.com	Marienplatz
20	3	Carol	Clark	carol@email.com	Marienplatz
21	3	Carol	Clark	carol@email.com	Marienplatz

22	3	Carol	Clark	carol@email.com	Marienplatz
23	3	Carol	Clark	carol@email.com	Marienplatz
24	3	Carol	Clark	carol@email.com	Marienplatz
25	3	Carol	Clark	carol@email.com	Marienplatz
26	3	Carol	Clark	carol@email.com	Marienplatz
27	3	Carol	Clark	carol@email.com	Marienplatz
28	4	David	Davis	david@email.com	Hohe Straße
29	4	David	Davis	david@email.com	Hohe Straße
30	4	David	Davis	david@email.com	Hohe Straße
31	4	David	Davis	david@email.com	Hohe Straße
32	4	David	Davis	david@email.com	Hohe Straße
33	4	David	Davis	david@email.com	Hohe Straße
34	4	David	Davis	david@email.com	Hohe Straße
35	4	David	Davis	david@email.com	Hohe Straße
36	4	David	Davis	david@email.com	Hohe Straße
37	5	Emma	Evans	emma@email.com	Zeil
38	5	Emma	Evans	emma@email.com	Zeil
39	5	Emma	Evans	emma@email.com	Zeil
40	5	Emma	Evans	emma@email.com	Zeil
41	5	Emma	Evans	emma@email.com	Zeil
42	5	Emma	Evans	emma@email.com	Zeil
43	5	Emma	Evans	emma@email.com	Zeil
44	5	Emma	Evans	emma@email.com	Zeil
45	5	Emma	Evans	emma@email.com	Zeil

45 rows

Achtung: Die implizite Syntax (`FROM a, b`) ist gefährlich! Wenn Sie vergessen, eine WHERE-Bedingung hinzuzufügen, passiert ein versehentlicher CROSS JOIN.

Nutzen Sie immer die explizite Syntax – dann ist klar: „Ich WILL alle Kombinationen!“

Beispiel 1: Produktkombinationen generieren

Erstellen Sie alle möglichen Kombinationen von zwei Produkten (z.B. für Paket-Angebote).

dbdiagram.io

dbdiagram.io

```sql
SELECT
  p1.product_name AS product_1,
  p2.product_name AS product_2,
  p1.price + p2.price AS bundle_price
FROM products p1
```

```
FROM products p1
CROSS JOIN products p2
WHERE p1.product_id < p2.product_id  -- Vermeidet Duplikate (A-B vs B-
ORDER BY bundle_price
LIMIT 5;
```

```
SELECT
  p1.product_name AS product_1,
  p2.product_name AS product_2,
  p1.price + p2.price AS bundle_price
FROM products p1
CROSS JOIN products p2
WHERE p1.product_id < p2.product_id  -- Vermeidet Duplikate (A-B vs B-
A)
ORDER BY bundle_price
LIMIT 5
```

#	product_1	product_2	bundle_price
1	Notebook	Paper	14.98
2	USB Cable	Paper	19.98
3	Notebook	USB Cable	24.98
4	Mouse	Paper	34.98
5	Mouse	Notebook	39.98

5 rows

Was passiert?

- Jedes Produkt wird mit jedem anderen kombiniert
- `WHERE p1.product_id < p2.product_id`: Verhindert, dass „Laptop + Mouse“ und „Mouse + Laptop“ beide erscheinen
- Ergebnis: Alle möglichen 2er-Pakete mit Gesamtpreis

Das ist praktisch für Preis-Kombinationen, Test-Daten oder Kalender-Aufgaben!

Beispiel 2: Datumsreihen generieren

CROSS JOIN ist perfekt, um alle Kombinationen aus zwei Listen zu erzeugen – z.B. jeden Kunden mit jedem Datum (für Reports).

```
-- Simuliere eine Datumsreihe mit VALUES
WITH dates AS (
```

```
  SELECT * FROM (VALUES
    ('2024-01-01'),
    ('2024-01-02'),
    ('2024-01-03')
  ) AS d(date)
)
SELECT
  c.customer_id,
  c.first_name,
  dates.date
FROM customers c
CROSS JOIN dates
ORDER BY dates.date, c.customer_id
LIMIT 10;
```

```
-- Simuliere eine Datumsreihe mit VALUES
WITH dates AS (
  SELECT * FROM (VALUES
    ('2024-01-01'),
    ('2024-01-02'),
    ('2024-01-03')
  ) AS d(date)
)
SELECT
  c.customer_id,
  c.first_name,
  dates.date
FROM customers c
CROSS JOIN dates
ORDER BY dates.date, c.customer_id
LIMIT 10
```

#	customer_id	first_name	date
1	1	Alice	2024-01-01
2	2	Bob	2024-01-01
3	3	Carol	2024-01-01
4	4	David	2024-01-01
5	5	Emma	2024-01-01
6	1	Alice	2024-01-02
7	2	Bob	2024-01-02
8	3	Carol	2024-01-02
9	4	David	2024-01-02
10	5	Emma	2024-01-02

10 rows

Ergebnis: Jeder Kunde erscheint für jedes Datum. Perfekt für Kalender-Grids oder A/B-Test-Setups!

Gefahr: Zeilen-Explosion!

Join-Zusammenfassung: Welchen wann?

Quick Reference: Join-Cheat-Sheet

Mehrere Tabellen verbinden (Multi-Table Joins)

Die Kette: JOIN nach JOIN

Beispiel: Vollständige Bestellung (4 Tabellen)

Best Practices für Multi-Table Joins

Abschluss: Joins meistern