L13: Advanced SQL – SET Operations & Views

Session 13 – Lecture Dauer: 90 Minuten **Lernziele:** LZ 2 (Relationale DB & SQL praktisch anwenden)
Block: 3 – SQL Vertiefung

Willkommen zur dreizehnten Session! In den letzten Sessions haben Sie gelernt, wie man Daten mit Joins kombiniert, Aggregationen durchführt und mit Subqueries arbeitet. Heute erweitern wir Ihr SQL-Toolkit um zwei mächtige Konzepte: SET Operations und Views.

SET Operations erlauben es uns, Ergebnismengen mathematisch zu kombinieren – wie Vereinigung, Schnittmenge und Differenz aus der Mengenlehre. Views hingegen geben uns die Möglichkeit, komplexe Queries als wiederverwendbare virtuelle Tabellen zu speichern.

Wir arbeiten heute mit unserem bekannten E-Commerce-Schema aus Session 10 und erweitern es um eine neue Tabelle: Mitarbeiter. Denn stellen Sie sich vor: Einige Ihrer Mitarbeiter bestellen auch privat im Shop – und genau hier kommen SET Operations ins Spiel!

Datenbank-Setup: Online-Shop erweitert

Bevor wir loslegen, initialisieren wir unsere Datenbank. Wir nutzen das bekannte E-Commerce-Schema und fügen eine neue Tabelle hinzu: Mitarbeiter.

```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 REFERENCES locations(location_id)
);
```

```
);

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

-- Products: Produktkatalog
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  product_name TEXT NOT NULL,
  price DECIMAL(10,2)
);

-- Product_Categories: N:M Beziehung zwischen Produkten und Kategori
CREATE TABLE product_categories (
  product_id INTEGER REFERENCES products(product_id),
  category_id INTEGER REFERENCES categories(category_id),
  PRIMARY KEY (product_id, category_id)
);

-- Order_Items: Bestellpositionen
CREATE TABLE order_items (
  order_item_id INTEGER PRIMARY KEY,
  order_id INTEGER REFERENCES orders(order_id),
  product_id INTEGER REFERENCES products(product_id),
  quantity INTEGER,
  line_total DECIMAL(10,2)
);

-- NEU: Employees - Mitarbeitertabelle
CREATE TABLE employees (
  employee_id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE,
  department TEXT,
  hire_date DATE
);

-- 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');

```

```sql
-- 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', 'Paper products and writing supplies'),
  (4, 'Office Equipment', 'Printers, scanners, and office machines')

-- Sample Data: Customers (erweitert)
INSERT INTO customers(customer_id, first_name, last_name, email, str
  street_number, location_id) VALUES
  (1, 'Alice', 'Smith', 'alice.smith@example.com', 'Main Street', '4
    ),
  (2, 'Bob', 'Johnson', 'bob.johnson@example.com', 'Oak Avenue', '15
  (3, 'Carol', 'Williams', 'carol.williams@example.com', 'Elm Road',
    1),
  (4, 'David', 'Lee', 'david.lee@example.com', 'Maple Lane', '23', 3
  (5, 'Emma', 'Brown', 'emma.brown@example.com', 'Pine Street', '7',

-- Sample Data: Employees (einige überlappen mit Customers!)
INSERT INTO employees(employee_id, first_name, last_name, email,
  department, hire_date) VALUES
  (1, 'Alice', 'Smith', 'alice.smith@shop-corp.com', 'Sales', '2020-
    ),
  (2, 'David', 'Lee', 'david.lee@shop-corp.com', 'IT', '2019-07-01')
  (3, 'Frank', 'Wilson', 'frank.wilson@shop-corp.com', 'HR', '2021-1
    ),
  (4, 'Grace', 'Taylor', 'grace.taylor@shop-corp.com', 'Marketing',
    -02-14'),
  (5, 'Hannah', 'Martinez', 'hannah.martinez@shop-corp.com', 'Financ
    '2020-09-10');

-- Sample Data: Orders
INSERT INTO orders(order_id, customer_id, order_date, total_amount,
  ) VALUES
  (101, 1, '2024-01-15', 299.99, 'delivered'),
  (102, 1, '2024-02-20', 139.97, 'delivered'),
  (103, 2, '2024-01-22', 999.99, 'delivered'),
  (104, 3, '2024-03-01', 749.94, 'processing'),
  (105, 4, '2024-02-10', 199.99, 'delivered');

-- Sample Data: Products
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', 39.99),
  (9, 'Paper (500 sheets)', 12.99);
```

```sql

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

-- 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, 104, 5, 1, 199.99),
  (7, 105, 5, 1, 199.99);
```

```
-- 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 REFERENCES locations(location_id)
)
```

ok

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

ok

```
-- Products: Produktkatalog
CREATE TABLE products (
  product_id INTEGER PRIMARY KEY,
  product_name TEXT NOT NULL,
  price DECIMAL(10,2)
)
```

ok

```
-- Product_Categories: N:M Beziehung zwischen Produkten und Kategorien
CREATE TABLE product_categories (
  product_id INTEGER REFERENCES products(product_id),
  category_id INTEGER REFERENCES categories(category_id),
  PRIMARY KEY (product_id, category_id)
)
```

ok

```
-- Order_Items: Bestellpositionen
CREATE TABLE order_items (
  order_item_id INTEGER PRIMARY KEY,
  order_id INTEGER REFERENCES orders(order_id),
  product_id INTEGER REFERENCES products(product_id),
  quantity INTEGER,
  line_total DECIMAL(10,2)
)
```

ok

```
-- NEU: Employees – Mitarbeitertabelle
CREATE TABLE employees (
  employee_id INTEGER PRIMARY KEY,
  first_name TEXT NOT NULL,
  last_name TEXT NOT NULL,
  email TEXT UNIQUE,
  department TEXT,
  hire_date DATE
)
```

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')
```

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', 'Paper products and writing supplies'),
  (4, 'Office Equipment', 'Printers, scanners, and office machines')
```

ok

```
-- Sample Data: Customers (erweitert)
INSERT INTO customers(customer_id, first_name, last_name, email, street,
street_number, location_id) VALUES
  (1, 'Alice', 'Smith', 'alice.smith@example.com', 'Main Street', '42', 1),
  (2, 'Bob', 'Johnson', 'bob.johnson@example.com', 'Oak Avenue', '15', 2),
  (3, 'Carol', 'Williams', 'carol.williams@example.com', 'Elm Road', '8', 1),
  (4, 'David', 'Lee', 'david.lee@example.com', 'Maple Lane', '23', 3),
  (5, 'Emma', 'Brown', 'emma.brown@example.com', 'Pine Street', '7', 4)
```

ok

```
-- Sample Data: Employees (einige überlappen mit Customers!)
INSERT INTO employees(employee_id, first_name, last_name, email, department,
hire_date) VALUES
  (1, 'Alice', 'Smith', 'alice.smith@shop-corp.com', 'Sales', '2020-03-15'),
  (2, 'David', 'Lee', 'david.lee@shop-corp.com', 'IT', '2019-07-01'),
  (3, 'Frank', 'Wilson', 'frank.wilson@shop-corp.com', 'HR', '2021-11-20'),
  (4, 'Grace', 'Taylor', 'grace.taylor@shop-corp.com', 'Marketing', '2022-02-14'),
  (5, 'Hannah', 'Martinez', 'hannah.martinez@shop-corp.com', 'Finance', '2020-09-10')
```

ok

```
-- Sample Data: Orders
INSERT INTO orders(order_id, customer_id, order_date, total_amount, status) VALUES
  (101, 1, '2024-01-15', 299.99, 'delivered'),
  (102, 1, '2024-02-20', 139.97, 'delivered'),
  (103, 2, '2024-01-22', 999.99, 'delivered'),
  (104, 3, '2024-03-01', 749.94, 'processing'),
  (105, 4, '2024-02-10', 199.99, 'delivered')
```

ok

```
-- Sample Data: Products
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', 39.99),
  (9, 'Paper (500 sheets)', 12.99)
```

ok

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

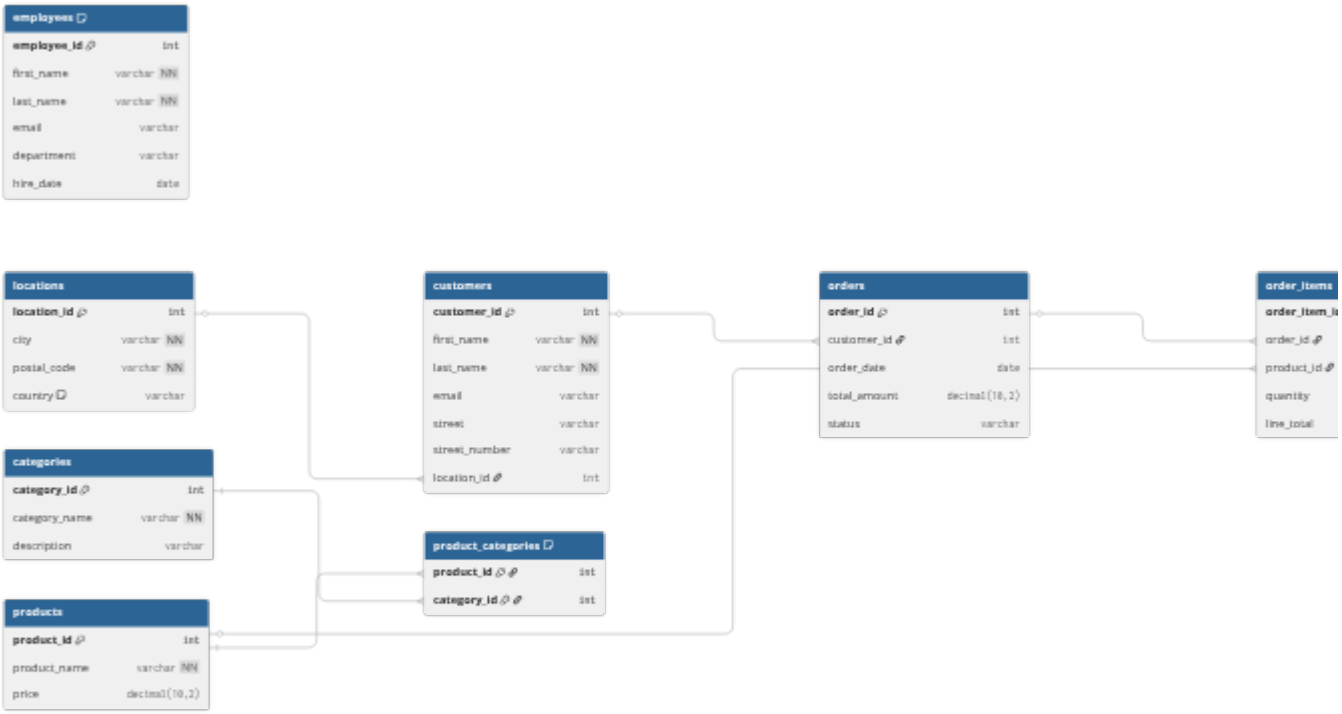
ok

```
-- Paper → Stationery

-- 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, 104, 5, 1, 199.99),
  (7, 105, 5, 1, 199.99)
```

ok

Schema-Übersicht:



dbdiagram.io

dbdiagram.io

Beachten Sie die neue Employees-Tabelle! Alice Smith und David Lee sind sowohl Kunden als auch Mitarbeiter – das werden wir gleich nutzen, um SET Operations zu demonstrieren.

Motivation & Kontext

Stellen Sie sich folgende Business-Szenarien vor:

Szenario 1: Newsletter-Kampagne

Ihr Marketing-Team möchte einen Newsletter versenden – an ALLE Personen in Ihrer Datenbank: Kunden UND Mitarbeiter. Wie kombinieren Sie diese beiden Listen effizient?

Szenario 2: Mitarbeiter-Rabatt-Programm

Sie möchten herausfinden, welche Mitarbeiter AUCH als Kunden bei Ihnen einkaufen, um ihnen spezielle Mitarbeiter-Rabatte anzubieten. Wie identifizieren Sie Überschneidungen?

Szenario 3: Komplexe Analysen wiederverwenden

Ihre Analyse-Queries für „VIP-Kunden“ und „Aktive Kunden“ werden immer länger und müssen in mehreren Reports verwendet werden. Wie vermeiden Sie Code-Duplikation?

Die Antworten liegen in zwei mächtigen SQL-Features: SET Operations für Mengen-Kombinationen und Views für Query-Wiederverwendung. Beginnen wir mit SET Operations!

Teil 1: SET Operations – Mengenlehre für SQL

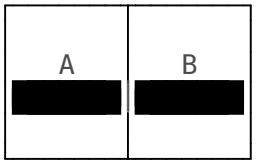
SET Operations basieren direkt auf mathematischen Mengenoperationen. Wenn Sie Venn-Diagramme aus der Schule kennen, werden Sie diese intuitiv verstehen.

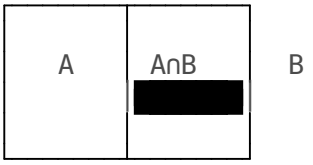
Überblick: Die drei SET Operations

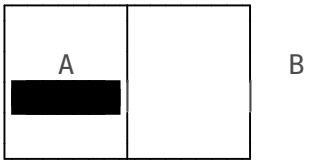
SQL bietet drei Haupt-SET-Operations: UNION, INTERSECT und EXCEPT. Jede löst ein spezifisches Problem.

```
Venn-Diagramme für SET Operations:

UNION (Vereinigung)          INTERSECT (Schnittmenge)     EXCEPT (Differenz)




Alle Elemente aus A oder B   Nur gemeinsame Elemente      Nur Elemente aus A
                               von A und B                  (nicht in B)
```

Operation	Bedeutung	Use Case
UNION	Vereinigung beider Mengen	Newsletter an Kunden UND Mitarbeiter
INTERSECT	Nur gemeinsame Elemente	Mitarbeiter, die auch Kunden sind
EXCEPT	Nur Elemente aus A, nicht in B	Kunden, die KEINE Mitarbeiter sind

Schauen wir uns jede Operation im Detail an, beginnend mit UNION.

UNION – Vereinigung von Mengen

UNION kombiniert die Ergebnisse zweier SELECT-Statements zu einer einzigen Ergebnismenge. Duplikate werden automatisch entfernt – es sei denn, Sie verwenden UNION ALL.

Syntax:

```sql
SELECT spalten FROM tabelle1
UNION [ALL]
SELECT spalten FROM tabelle2;
```

Wichtige Regeln:

- Beide SELECT-Statements müssen die **gleiche Anzahl** von Spalten haben
- Spaltentypen müssen **kompatibel** sein (oder konvertierbar)
- **UNION** entfernt Duplikate → langsamer
- **UNION ALL** behält Duplikate → schneller

Lassen Sie uns das praktisch demonstrieren. Wir kombinieren Kunden und Mitarbeiter für eine Newsletter-Liste.

Beispiel 1: Newsletter-Liste erstellen

```sql
-- UNION: Alle Personen (Kunden + Mitarbeiter) ohne Duplikate
-- Problem: E-Mail-Adressen sind unterschiedlich (@example.com vs @sh
  -corp.com)
-- Lösung: Nur Name vergleichen, nicht E-Mail!
SELECT
  first_name,
  last_name
FROM customers

UNION

SELECT
  first_name,
  last_name
FROM employees

ORDER BY last_name, first_name;
```

```
-- UNION: Alle Personen (Kunden + Mitarbeiter) ohne Duplikate
-- Problem: E-Mail-Adressen sind unterschiedlich (@example.com vs @shop-corp.com)
-- Lösung: Nur Name vergleichen, nicht E-Mail!
SELECT
  first_name,
  last_name
FROM customers

UNION

SELECT
  first_name,
  last_name
FROM employees

ORDER BY last_name, first_name
```

#	first_name	last_name
1	Emma	Brown
2	Bob	Johnson
3	David	Lee
4	Hannah	Martinez
5	Alice	Smith
6	Grace	Taylor
7	Carol	Williams
8	Frank	Wilson

8 rows

Was sehen Sie?

Jetzt sehen Sie 8 Personen statt 10! Alice Smith und David Lee erscheinen nur einmal. Warum? UNION vergleicht ALLE Spalten – hier nur first*name und last*name. Da beide Personen in beiden Tabellen vorkommen (gleicher Name), werden die Duplikate entfernt!

Wichtig: Wenn wir email mit einbeziehen würden, wären Alice und David KEINE Duplikate, weil ihre E-Mail-Adressen unterschiedlich sind (alice.smith@example.com vs alice.smith@shop-corp.com)!

UNION ALL – Alle Einträge behalten:

```sql
-- UNION ALL: Behält ALLE Einträge (keine Deduplizierung)
SELECT
  first_name,
  last_name,
  'Customer' AS type
```

```sql
FROM customers

UNION ALL

SELECT
  first_name,
  last_name,
  'Employee' AS type
FROM employees

ORDER BY last_name, first_name;
```

```sql
-- UNION ALL: Behält ALLE Einträge (keine Deduplizierung)
SELECT
  first_name,
  last_name,
  'Customer' AS type
FROM customers

UNION ALL

SELECT
  first_name,
  last_name,
  'Employee' AS type
FROM employees

ORDER BY last_name, first_name
```

#	first_name	last_name	type
1	Emma	Brown	Customer
2	Bob	Johnson	Customer
3	David	Lee	Employee
4	David	Lee	Customer
5	Hannah	Martinez	Employee
6	Alice	Smith	Employee
7	Alice	Smith	Customer
8	Grace	Taylor	Employee
9	Carol	Williams	Customer
10	Frank	Wilson	Employee

10 rows

Jetzt sehen Sie 10 Zeilen! Alice und David erscheinen zweimal – einmal als Customer, einmal als Employee. UNION ALL ist schneller, weil keine Deduplizierung nötig ist.

Warum E-Mail-Adressen problematisch sind:

```sql
-- UNION mit E-Mail: Keine Deduplizierung wegen unterschiedlicher E-M
SELECT
  first_name,
  last_name,
  email
FROM customers

UNION

SELECT
  first_name,
  last_name,
  email
FROM employees

ORDER BY last_name, first_name;
```

```
-- UNION mit E-Mail: Keine Deduplizierung wegen unterschiedlicher E-Mails!
SELECT
  first_name,
  last_name,
  email
FROM customers

UNION

SELECT
  first_name,
  last_name,
  email
FROM employees

ORDER BY last_name, first_name
```

#	first_name	last_name	email
1	Emma	Brown	emma.brown@example.com
2	Bob	Johnson	bob.johnson@example.com
3	David	Lee	david.lee@example.com
4	David	Lee	david.lee@shop-corp.com
5	Hannah	Martinez	hannah.martinez@shop-corp.com
6	Alice	Smith	alice.smith@shop-corp.com
7	Alice	Smith	alice.smith@example.com
8	Grace	Taylor	grace.taylor@shop-corp.com
9	Carol	Williams	carol.williams@example.com
10	Frank	Wilson	frank.wilson@shop-corp.com

10 rows

Überraschung: Wieder 10 Zeilen! Warum? UNION vergleicht ALLE Spalten. Obwohl Alice Smith in beiden Tabellen vorkommt, sind die E-Mails unterschiedlich (alice.smith@example.com vs alice.smith@shop-corp.com) – also keine Deduplizierung! Wenn Sie nur eindeutige Personen wollen, vergleichen Sie nur die Spalten, die wirklich identisch sein müssen (z.B. nur Name).

Performance-Tipp:

> Nutzen Sie **UNION ALL**, wenn Sie wissen, dass keine Duplikate existieren oder Duplikate gewünscht sind. Das spart die kostspielige Deduplizierung!

INTERSECT – Schnittmenge finden

INTERSECT gibt nur die Zeilen zurück, die in BEIDEN Ergebnismengen vorkommen. Perfekt, um Gemeinsamkeiten zu identifizieren.

Syntax:

```
SELECT spalten FROM tabelle1
INTERSECT
SELECT spalten FROM tabelle2;
```

Unser Business-Szenario: Finden Sie alle Mitarbeiter, die auch als Kunden im Shop einkaufen, um ihnen Mitarbeiter-Rabatte anzubieten.

Beispiel 2: Mitarbeiter-Kunden identifizieren

```
1  -- Mitarbeiter, die auch Kunden sind (basierend auf Namen)
2  SELECT
3    first_name,
4    last_name
5  FROM customers
6
7  INTERSECT
8
9  SELECT
10   first_name,
11   last_name
12 FROM employees
13
14 ORDER BY last_name;
```

```
-- Mitarbeiter, die auch Kunden sind (basierend auf Namen)
SELECT
  first_name,
  last_name
FROM customers

INTERSECT

SELECT
  first_name,
  last_name
FROM employees

ORDER BY last_name
```

#	first_name	last_name
1	David	Lee
2	Alice	Smith

2 rows

Ergebnis:

Nur Alice Smith und David Lee erscheinen! Das sind exakt die Personen, die in beiden Tabellen vorkommen (gleicher Name). Perfekt für unser Mitarbeiter-Rabatt-Programm. Beachten Sie: Wir vergleichen nur Namen, nicht E-Mails, da diese unterschiedlich sein können!

Alternative mit JOIN:

Man könnte das auch mit einem JOIN lösen, aber INTERSECT ist oft lesbarer für diesen spezifischen Use-Case.

```sql
-- Gleichwertig mit INNER JOIN
SELECT DISTINCT
  c.first_name,
  c.last_name,
  c.email
FROM customers c
INNER JOIN employees e
  ON c.first_name = e.first_name
  AND c.last_name = e.last_name
ORDER BY c.last_name;
```

```
-- Gleichwertig mit INNER JOIN
SELECT DISTINCT
  c.first_name,
  c.last_name,
  c.email
FROM customers c
INNER JOIN employees e
  ON c.first_name = e.first_name
  AND c.last_name = e.last_name
ORDER BY c.last_name
```

#	first_name	last_name	email
1	David	Lee	david.lee@example.com
2	Alice	Smith	alice.smith@example.com

2 rows

Beide Queries liefern das gleiche Ergebnis, aber INTERSECT macht die Intention klarer: „Zeige mir die Überschneidung!“

EXCEPT – Differenz finden

EXCEPT gibt die Zeilen aus der ersten Ergebnismenge zurück, die NICHT in der zweiten vorkommen. Ideal, um „fehlende“ oder „exklusive“ Datensätze zu identifizieren.

Syntax:

```
SELECT spalten FROM tabelle1
EXCEPT
SELECT spalten FROM tabelle2;
```

Business-Szenario: Finden Sie alle Kunden, die KEINE Mitarbeiter sind, für eine reine Kunden-Marketing-Kampagne.

Beispiel 3: Reine Kunden identifizieren

```
1  -- Kunden, die KEINE Mitarbeiter sind
2  SELECT
3    first_name,
4    last_name
5  FROM customers
6
7  EXCEPT
8
9  SELECT
10   first_name,
11   last_name
12 FROM employees
```

```
13
14  ORDER BY last_name;
```

```
-- Kunden, die KEINE Mitarbeiter sind
SELECT
  first_name,
  last_name
FROM customers

EXCEPT

SELECT
  first_name,
  last_name
FROM employees

ORDER BY last_name
```

#	first_name	last_name
1	Emma	Brown
2	Bob	Johnson
3	Carol	Williams

3 rows

Ergebnis:

Nur Bob, Carol und Emma erscheinen – die drei Kunden, die NICHT in der Mitarbeiter-Tabelle sind (basierend auf Namen). Alice und David werden herausgefiltert, weil sie auch als Mitarbeiter existieren **Alternative mit LEFT JOIN:**

EXCEPT kann auch mit einem Anti-Join (LEFT JOIN + NULL Check) gelöst werden.

```
1  -- Gleichwertig mit LEFT JOIN + NULL
2  SELECT
3    c.first_name,
4    c.last_name,
5    c.email
6  FROM customers c
7  LEFT JOIN employees e
8    ON c.first_name = e.first_name
9    AND c.last_name = e.last_name
10 WHERE e.employee_id IS NULL
11 ORDER BY c.last_name;
```

```
-- Gleichwertig mit LEFT JOIN + NULL
SELECT
  c.first_name,
  c.last_name,
  c.email
FROM customers c
LEFT JOIN employees e
  ON c.first_name = e.first_name
  AND c.last_name = e.last_name
WHERE e.employee_id IS NULL
ORDER BY c.last_name
```

#	first_name	last_name	email
1	Emma	Brown	emma.brown@example.com
2	Bob	Johnson	bob.johnson@example.com
3	Carol	Williams	carol.williams@example.com

3 rows

Wieder: Beide Ansätze funktionieren, aber EXCEPT ist semantisch klarer für „Zeige mir A ohne B“.

Wichtig: Reihenfolge zählt!

EXCEPT ist nicht kommutativ! A EXCEPT B ist NICHT das gleiche wie B EXCEPT A.

```sql
-- Umgekehrt: Mitarbeiter, die KEINE Kunden sind
SELECT first_name, last_name, email FROM employees
EXCEPT
SELECT first_name, last_name, email FROM customers
ORDER BY last_name;
```

```
-- Umgekehrt: Mitarbeiter, die KEINE Kunden sind
SELECT first_name, last_name, email FROM employees
EXCEPT
SELECT first_name, last_name, email FROM customers
ORDER BY last_name
```

#	first_name	last_name	email
1	David	Lee	david.lee@shop-corp.com
2	Hannah	Martinez	hannah.martinez@shop-corp.com
3	Alice	Smith	alice.smith@shop-corp.com
4	Grace	Taylor	grace.taylor@shop-corp.com
5	Frank	Wilson	frank.wilson@shop-corp.com

5 rows

Jetzt sehen Sie Frank, Grace und Hannah – die drei Mitarbeiter, die nicht in der Kunden-Tabelle sind!

Komplexes Beispiel: Produkte ohne Verkäufe

Ein sehr praktisches Beispiel: Finden Sie alle Produkte, die noch NIE verkauft wurden. Das sind Ihre „Ladenhüter“, die Sie vielleicht aus dem Sortiment nehmen oder bewerben sollten.

```sql
-- Produkte, die noch nie verkauft wurden
SELECT
  product_id,
  product_name,
  price
FROM products

EXCEPT

SELECT
  p.product_id,
  p.product_name,
  p.price
FROM products p
INNER JOIN order_items oi ON p.product_id = oi.product_id

ORDER BY product_name;
```

```
-- Produkte, die noch nie verkauft wurden
SELECT
  product_id,
  product_name,
  price
FROM products

EXCEPT

SELECT
  p.product_id,
  p.product_name,
  p.price
FROM products p
INNER JOIN order_items oi ON p.product_id = oi.product_id

ORDER BY product_name
```

#	product_id	product_name	price
1	8	Desk Lamp	39.99
2	9	Paper (500 sheets)	12.99
3	7	USB Cable	14.99

3 rows

Ergebnis:

USB Cable, Desk Lamp und Paper wurden nie verkauft! Das ist wertvolle Business-Intelligence. Schauen wir uns die Alternative mit LEFT JOIN an.

```sql
-- Alternative: LEFT JOIN + NULL Check
SELECT
  p.product_id,
  p.product_name,
  p.price
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
WHERE oi.order_item_id IS NULL
ORDER BY p.product_name;
```

```
-- Alternative: LEFT JOIN + NULL Check
SELECT
  p.product_id,
  p.product_name,
  p.price
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
WHERE oi.order_item_id IS NULL
ORDER BY p.product_name
```

#	product_id	product_name	price
1	8	Desk Lamp	39.99
2	9	Paper (500 sheets)	12.99
3	7	USB Cable	14.99

3 rows

Identisches Ergebnis! Welche Variante ist besser? Das hängt von der Datenbank und den Indizes ab. Bei modernen Datenbanken sind beide meist gleich schnell.

Performance & Best Practices

SET Operations haben ihre eigenen Performance-Charakteristika. Hier sind die wichtigsten Punkte.

Performance-Matrix:

Operation	Sortierung nötig?	Deduplizierung?	Performance-Tipp
UNION	Ja	Ja	Nutze UNION ALL wenn möglich
UNION ALL	Nein	Nein	Schnellste Option
INTERSECT	Ja	Ja	Hash-Algorithmus effizient
EXCEPT	Ja	Ja	Anti-Join Alternative prüfen

Best Practices:

- ✅ **UNION ALL statt UNION**, wenn Duplikate OK sind
- ✅ **Spalten-Typen kompatibel halten** – implizite Konvertierungen vermeiden
- ✅ **Indexe auf Join-Spalten** setzen (bei der Alternative mit JOINs)
- ✅ **EXPLAIN ANALYZE nutzen**, um Performance zu vergleichen
- ⚠️ **Große Mengen vorsichtig** – SET Ops können Sorts auslösen
- ⚠️ **WHERE-Filter VOR SET Ops** anwenden, um Datenmenge zu reduzieren

Performance-Optimierung:

```sql
-- ❌ Ineffizient: Große Mengen erst kombinieren, dann filtern
SELECT *
FROM (
  SELECT first_name, last_name, email FROM customers
  UNION
  SELECT first_name, last_name, email FROM employees
)
WHERE last_name LIKE 'S%';  -- Filter NACH UNION

-- ✅ Besser: Erst filtern, dann kombinieren
SELECT first_name, last_name, email
FROM customers
WHERE last_name LIKE 'S%'
UNION
SELECT first_name, last_name, email
FROM employees
WHERE last_name LIKE 'S%';
```

```
-- ❌ Ineffizient: Große Mengen erst kombinieren, dann filtern
SELECT *
FROM (
  SELECT first_name, last_name, email FROM customers
  UNION
  SELECT first_name, last_name, email FROM employees
)
WHERE last_name LIKE 'S%'
```

#	first_name	last_name	email
1	Alice	Smith	alice.smith@example.com
2	Alice	Smith	alice.smith@shop-corp.com

2 rows

```
-- Filter NACH UNION

-- ✅ Besser: Erst filtern, dann kombinieren
SELECT first_name, last_name, email
FROM customers
WHERE last_name LIKE 'S%'
UNION
SELECT first_name, last_name, email
FROM employees
WHERE last_name LIKE 'S%'
```

#	first_name	last_name	email
1	Alice	Smith	alice.smith@example.com
2	Alice	Smith	alice.smith@shop-corp.com

2 rows

Durch frühes Filtern reduzieren wir die Datenmengen vor der UNION – das spart Ressourcen!

Teil 2: Views – Abstraktion & Wiederverwendung

Nachdem wir SET Operations gemeistert haben, kommen wir zu Views. Views lösen ein anderes Problem: Code-Wiederverwendung und Abstraktion komplexer Queries.

Was sind Views?

Eine View ist eine gespeicherte SELECT-Query, die wie eine Tabelle abgefragt werden kann – aber keine eigenen Daten speichert. Man nennt sie auch „virtuelle Tabelle“.

Konzept:

```
CREATE VIEW view_name AS
```

```
SELECT spalten
FROM tabellen
WHERE bedingungen;

-- Dann wie eine Tabelle nutzen:
SELECT * FROM view_name;
```

Wichtige Eigenschaften:

- ❌ **Keine Datenspeicherung** – Views speichern nur die Query-Definition
- ✅ **Immer aktuell** – Daten werden bei jeder Abfrage neu gelesen
- ✅ **Code-Wiederverwendung** – Komplexe Queries einmal definieren
- ✅ **Zugriffskontrolle** – Beschränkung auf bestimmte Spalten/Zeilen
- ⚠️ **Performance** – Views sind so schnell (oder langsam) wie die zugrundeliegende Query

Einfache Views erstellen

Beginnen wir mit einem einfachen Beispiel: Eine View für alle VIP-Kunden (die mehr als 500€ ausgegeben haben).

```sql
-- View für VIP-Kunden erstellen
CREATE VIEW vip_customers AS
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email,
  SUM(o.total_amount) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
HAVING SUM(o.total_amount) > 500;

-- View abfragen
SELECT * FROM vip_customers ORDER BY total_spent DESC;
```

```
-- View für VIP-Kunden erstellen
CREATE VIEW vip_customers AS
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email,
  SUM(o.total_amount) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
HAVING SUM(o.total_amount) > 500
```

ok

```
-- View abfragen
SELECT * FROM vip_customers ORDER BY total_spent DESC
```

#	customer_id	first_name	last_name	email	total_spent
1	2	Bob	Johnson	bob.johnson@example.com	999.99
2	3	Carol	Williams	carol.williams@example.com	749.94

2 rows

Was sehen Sie?

Alice (439.96€) und Bob (999.99€) sind unsere VIP-Kunden! Die komplexe Query mit JOIN, GROUP BY und HAVING ist jetzt in einer View gekapselt. Sie können sie beliebig oft wiederverwenden, ohne den Code zu duplizieren.

Views weiter filtern:

Das Schöne: Sie können Views wie normale Tabellen behandeln – filtern, sortieren, joinen!

```sql
-- View weiter filtern
SELECT
  first_name,
  last_name,
  total_spent
FROM vip_customers
WHERE total_spent > 400
ORDER BY last_name;
```

```
-- View weiter filtern
SELECT
  first_name,
  last_name,
  total_spent
FROM vip_customers
WHERE total_spent > 400
ORDER BY last_name
```

#	first_name	last_name	total_spent
1	Bob	Johnson	999.99
2	Carol	Williams	749.94

2 rows

Die Datenbank kombiniert intern Ihre Filter mit der View-Definition – das nennt man „View Merging“. Moderne Datenbanken sind hier sehr effizient!

Views für häufige Analysen

Views sind ideal, um wiederkehrende Analyse-Queries zu speichern. Schauen wir uns weitere praktische Beispiele an.

View für Produktkategorien:

```sql
-- View: Produkte mit ihren Kategorien (N:M Beziehung aufgelöst)
CREATE VIEW products_with_categories AS
SELECT
  p.product_id,
  p.product_name,
  p.price,
  c.category_name,
  c.description AS category_description
FROM products p
INNER JOIN product_categories pc ON p.product_id = pc.product_id
INNER JOIN categories c ON pc.category_id = c.category_id;

-- View nutzen
SELECT * FROM products_with_categories
ORDER BY category_name, product_name;
```

```
-- View: Produkte mit ihren Kategorien (N:M Beziehung aufgelöst)
CREATE VIEW products_with_categories AS
SELECT
  p.product_id,
  p.product_name,
  p.price,
  c.category_name,
  c.description AS category_description
FROM products p
INNER JOIN product_categories pc ON p.product_id = pc.product_id
INNER JOIN categories c ON pc.category_id = c.category_id
```

ok

```
-- View nutzen
SELECT * FROM products_with_categories
ORDER BY category_name, product_name
```

#	product_id	product_name	price	category_name	category_description
1	3	Keyboard	79.99	Electronics	Electronic devices and accessories
2	1	Laptop	999.99	Electronics	Electronic devices and accessories
3	4	Monitor	299.99	Electronics	Electronic devices and accessories
4	2	Mouse	29.99	Electronics	Electronic devices and accessories
5	7	USB Cable	14.99	Electronics	Electronic devices and accessories
6	5	Desk Chair	199.99	Furniture	Office and home furniture
7	8	Desk Lamp	39.99	Furniture	Office and home furniture
8	5	Desk Chair	199.99	Office Equipment	Printers, scanners, and office machines
9	8	Desk Lamp	39.99	Office Equipment	Printers, scanners, and office machines
10	4	Monitor	299.99	Office Equipment	Printers, scanners, and office machines
11	6	Notebook	9.99	Stationery	Paper products and writing supplies
12	9	Paper (500 sheets)	12.99	Stationery	Paper products and writing supplies

12 rows

View für aktive Kunden:

```sql
-- View: Kunden mit Bestellungen in 2024
CREATE VIEW active_customers_2024 AS
SELECT DISTINCT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email,
  l.city
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
INNER JOIN locations l ON c.location_id = l.location_id
WHERE EXTRACT(YEAR FROM o.order_date) = 2024;

-- View nutzen
SELECT * FROM active_customers_2024 ORDER BY city, last_name;
```

```
-- View: Kunden mit Bestellungen in 2024
CREATE VIEW active_customers_2024 AS
SELECT DISTINCT
  c.customer_id,
  c.first_name,
  c.last_name,
  c.email,
  l.city
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
INNER JOIN locations l ON c.location_id = l.location_id
WHERE EXTRACT(YEAR FROM o.order_date) = 2024
```

ok

```
-- View nutzen
SELECT * FROM active_customers_2024 ORDER BY city, last_name
```

#	customer_id	first_name	last_name	email	city
1	1	Alice	Smith	alice.smith@example.com	Berlin
2	3	Carol	Williams	carol.williams@example.com	Berlin
3	2	Bob	Johnson	bob.johnson@example.com	Hamburg
4	4	David	Lee	david.lee@example.com	Munich

4 rows

Diese View kapselt die Logik für „aktive Kunden“ – alle Reports können sie wiederverwenden!

Views & SET Operations kombinieren

Jetzt wird es mächtig! Wir können SET Operations in Views einbetten und Views mit SET Operations kombinieren.

View mit UNION – Alle Kontakte:

```sql
-- View: Alle Personen (Kunden + Mitarbeiter)
CREATE VIEW all_contacts AS
  SELECT
    customer_id AS person_id,
    first_name,
    last_name,
    email,
    'Customer' AS type
  FROM customers

  UNION ALL

  SELECT
    employee_id AS person_id,
    first_name,
    last_name,
    email,
    'Employee' AS type
  FROM employees;

-- View abfragen
SELECT * FROM all_contacts ORDER BY last_name, first_name;
```

```
-- View: Alle Personen (Kunden + Mitarbeiter)
CREATE VIEW all_contacts AS
  SELECT
    customer_id AS person_id,
    first_name,
    last_name,
    email,
    'Customer' AS type
  FROM customers

  UNION ALL

  SELECT
    employee_id AS person_id,
    first_name,
    last_name,
    email,
    'Employee' AS type
  FROM employees
```

ok

```
-- View abfragen
SELECT * FROM all_contacts ORDER BY last_name, first_name
```

#	person_id	first_name	last_name	email	type
1	5	Emma	Brown	emma.brown@example.com	Customer
2	2	Bob	Johnson	bob.johnson@example.com	Customer
3	2	David	Lee	david.lee@shop-corp.com	Employee
4	4	David	Lee	david.lee@example.com	Customer
5	5	Hannah	Martinez	hannah.martinez@shop-corp.com	Employee
6	1	Alice	Smith	alice.smith@shop-corp.com	Employee
7	1	Alice	Smith	alice.smith@example.com	Customer
8	4	Grace	Taylor	grace.taylor@shop-corp.com	Employee
9	3	Carol	Williams	carol.williams@example.com	Customer
10	3	Frank	Wilson	frank.wilson@shop-corp.com	Employee

10 rows

Vorteil:

Die UNION-Logik ist jetzt wiederverwendbar! Jeder Report, der alle Kontakte braucht, kann einfach FROM all_contacts selektieren.

Views kombinieren:

```sql
-- Zwei separate Views für verschiedene Analysen
CREATE VIEW customers_berlin AS
SELECT c.*, l.city
FROM customers c
INNER JOIN locations l ON c.location_id = l.location_id
WHERE l.city = 'Berlin';

CREATE VIEW employees_sales AS
SELECT *
FROM employees
WHERE department = 'Sales';

-- Views mit UNION kombinieren
SELECT first_name, last_name, 'Customer' AS type FROM customers_berli
UNION ALL
SELECT first_name, last_name, 'Employee' AS type FROM employees_sales
ORDER BY last_name;
```

```
-- Zwei separate Views für verschiedene Analysen
CREATE VIEW customers_berlin AS
SELECT c.*, l.city
FROM customers c
INNER JOIN locations l ON c.location_id = l.location_id
WHERE l.city = 'Berlin'
```

ok

```
CREATE VIEW employees_sales AS
SELECT *
FROM employees
WHERE department = 'Sales'
```

ok

```
-- Views mit UNION kombinieren
SELECT first_name, last_name, 'Customer' AS type FROM customers_berlin
UNION ALL
SELECT first_name, last_name, 'Employee' AS type FROM employees_sales
ORDER BY last_name
```

#	first_name	last_name	type
1	Alice	Smith	Customer
2	Alice	Smith	Employee
3	Carol	Williams	Customer

3 rows

Alice erscheint zweimal: Als Berliner Kundin und als Sales-Mitarbeiterin. Jede View kann unabhängig gewartet werden!

Views für Zugriffskontrolle

Ein wichtiger Use-Case für Views ist die Zugriffskontrolle. Sie können sensible Daten verbergen oder nur bestimmte Zeilen/Spalten freigeben.

Szenario: Public vs. Internal Data

```sql
-- View für öffentliche Produktdaten (ohne Preise)
CREATE VIEW public_products AS
SELECT
  product_id,
  product_name
FROM products;

-- View für interne Analysten (mit Preisen)
```

```sql
CREATE VIEW internal_products AS
SELECT
  product_id,
  product_name,
  price
FROM products;

-- Externe API würde nur public_products sehen:
SELECT * FROM public_products;
```

```
-- View für öffentliche Produktdaten (ohne Preise)
CREATE VIEW public_products AS
SELECT
  product_id,
  product_name
FROM products
```

ok

```
-- View für interne Analysten (mit Preisen)
CREATE VIEW internal_products AS
SELECT
  product_id,
  product_name,
  price
FROM products
```

ok

```
-- Externe API würde nur public_products sehen:
SELECT * FROM public_products
```

#	product_id	product_name
1	1	Laptop
2	2	Mouse
3	3	Keyboard
4	4	Monitor
5	5	Desk Chair
6	6	Notebook
7	7	USB Cable
8	8	Desk Lamp
9	9	Paper (500 sheets)

9 rows

Zugriffskontrolle in Praxis:

In echten Systemen würden Sie Datenbank-Permissions setzen: Externe User bekommen nur Zugriff auf public*products, interne auf internal*products. Die Basistabelle products bleibt geschützt!

View für gefilterte Daten:

```sql
-- View: Nur abgeschlossene Bestellungen
CREATE VIEW delivered_orders AS
```

```sql
SELECT
  order_id,
  customer_id,
  order_date,
  total_amount
FROM orders
WHERE status = 'delivered';

-- Reports nutzen nur gelieferte Bestellungen
SELECT * FROM delivered_orders;
```

```
-- View: Nur abgeschlossene Bestellungen
CREATE VIEW delivered_orders AS
SELECT
  order_id,
  customer_id,
  order_date,
  total_amount
FROM orders
WHERE status = 'delivered'
```

ok

```
-- Reports nutzen nur gelieferte Bestellungen
SELECT * FROM delivered_orders
```

#	order_id	customer_id	order_date	total_amount
1	101	1	2024-01-15	299.99
2	102	1	2024-02-20	139.97
3	103	2	2024-01-22	999.99
4	105	4	2024-02-10	199.99

4 rows

Der Report-User sieht nur fertige Bestellungen – egal ob versehentlich oder absichtlich, unfertige Bestellungen sind nicht zugänglich.

Views verwalten

Wie erstellt, ändert und löscht man Views? Hier die wichtigsten Operationen.

View erstellen oder ersetzen:

```sql
-- Neue View erstellen
CREATE VIEW my_view AS SELECT ...;

-- View ersetzen (falls existiert)
```

```
-- In PGlite: DROP + CREATE
DROP VIEW IF EXISTS my_view;
CREATE VIEW my_view AS SELECT ...;
```

View löschen:

```
-- View löschen
DROP VIEW IF EXISTS my_view;

-- Mehrere Views löschen
DROP VIEW IF EXISTS view1, view2, view3;
```

Best Practices für View-Management:

- ✅ **Benennungskonvention** nutzen (z.B. `vw_` Präfix oder `_view` Suffix)
- ✅ **Views dokumentieren** (Kommentare im SQL, README)
- ✅ **Views versionieren** (wie Code in Git)
- ✅ **Views testen** mit realistischen Datenmengen
- ⚠️ **View-Hierarchien begrenzen** (max. 2-3 Ebenen)
- ⚠️ **Verwaiste Views vermeiden** (regelmäßig aufräumen)

Materialized Views – Performance durch Caching

Standard-Views sind virtuelle Tabellen ohne eigene Datenspeicherung. Bei jeder Abfrage wird die zugrundeliegende Query neu ausgeführt. Materialized Views hingegen speichern das Ergebnis physisch – wie ein Snapshot.

Materialized View erstellen:

```
CREATE VIEW vw_vip_customers AS
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  SUM(o.total_amount) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(o.total_amount) > 500;

-- Materialized View für VIP-Kunden (Ausgaben > 500€)
CREATE MATERIALIZED VIEW mv_vip_customers AS
SELECT * FROM vw_vip_customers;

-- Abfragen wie eine normale Tabelle
SELECT * FROM mv_vip_customers ORDER BY total_spent DESC;
```

```
CREATE VIEW vw_vip_customers AS
SELECT
  c.customer_id,
  c.first_name,
  c.last_name,
  SUM(o.total_amount) AS total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING SUM(o.total_amount) > 500
```

ok

```
-- Materialized View für VIP-Kunden (Ausgaben > 500€)
CREATE MATERIALIZED VIEW mv_vip_customers AS
SELECT * FROM vw_vip_customers
```

ok

```
-- Abfragen wie eine normale Tabelle
SELECT * FROM mv_vip_customers ORDER BY total_spent DESC
```

#	customer_id	first_name	last_name	total_spent
1	2	Bob	Johnson	999.99
2	3	Carol	Williams	749.94

2 rows

Alice und Bob sind unsere VIP-Kunden mit 439.96€ und 999.99€! Die Daten sind jetzt physisch gespeichert – nicht nur als Query-Definition.

Das Problem: Veraltete Daten nach Updates

Fügen wir eine neue Bestellung für Alice hinzu – was passiert mit der Materialized View?

```
1  -- Neue Bestellung für Alice (wird zum VIP)
2  INSERT INTO orders(order_id, customer_id, order_date, total_amount, st
3  VALUES (106, 1, '2024-03-20', 199.99, 'delivered');
4
5  -- Prüfe direkt in orders: Alice hat jetzt mehr ausgegeben
6  SELECT * FROM vw_vip_customers;
7
8  -- ABER: Materialized View zeigt noch alte Werte!
9  SELECT * FROM mv_vip_customers;
```

```
-- Neue Bestellung für Alice (wird zum VIP)
INSERT INTO orders(order_id, customer_id, order_date, total_amount, status)
VALUES (106, 1, '2024-03-20', 199.99, 'delivered')
```

ok

```
-- Prüfe direkt in orders: Alice hat jetzt mehr ausgegeben
SELECT * FROM vw_vip_customers
```

#	customer_id	first_name	last_name	total_spent
1	1	Alice	Smith	639.95
2	2	Bob	Johnson	999.99
3	3	Carol	Williams	749.94

3 rows

```
-- ABER: Materialized View zeigt noch alte Werte!
SELECT * FROM mv_vip_customers
```

#	customer_id	first_name	last_name	total_spent
1	2	Bob	Johnson	999.99
2	3	Carol	Williams	749.94

2 rows

Sehen Sie das Problem? Die echte Abfrage zeigt 639.95€ für Alice, aber die Materialized View zeigt noch 439.96€! Materialized Views werden NICHT automatisch aktualisiert.

Lösung: REFRESH MATERIALIZED VIEW

```
-- Materialized View manuell aktualisieren
REFRESH MATERIALIZED VIEW mv_vip_customers;

-- Jetzt sind die Daten aktuell!
SELECT * FROM mv_vip_customers ORDER BY total_spent DESC;
```

```
-- Materialized View manuell aktualisieren
REFRESH MATERIALIZED VIEW mv_vip_customers
```

ok

```
-- Jetzt sind die Daten aktuell!
SELECT * FROM mv_vip_customers ORDER BY total_spent DESC
```

#	customer_id	first_name	last_name	total_spent
1	2	Bob	Johnson	999.99
2	3	Carol	Williams	749.94
3	1	Alice	Smith	639.95

3 rows

Nach dem REFRESH zeigt die Materialized View die aktuellen Werte! Alice hat jetzt 639.95€ insgesamt.

Performance-Überlegungen

Views sind praktisch, aber haben Performance-Implikationen. Hier die wichtigsten Punkte.

Performance-Fakten:

- Views speichern **keine Daten** → kein zusätzlicher Speicher
- Views nutzen **Indexe der Basistabellen** → keine eigenen Indexe
- Views werden bei **jeder Abfrage neu ausgeführt** → keine Caching
- Moderne DBs nutzen **View Merging** → Filter werden optimiert

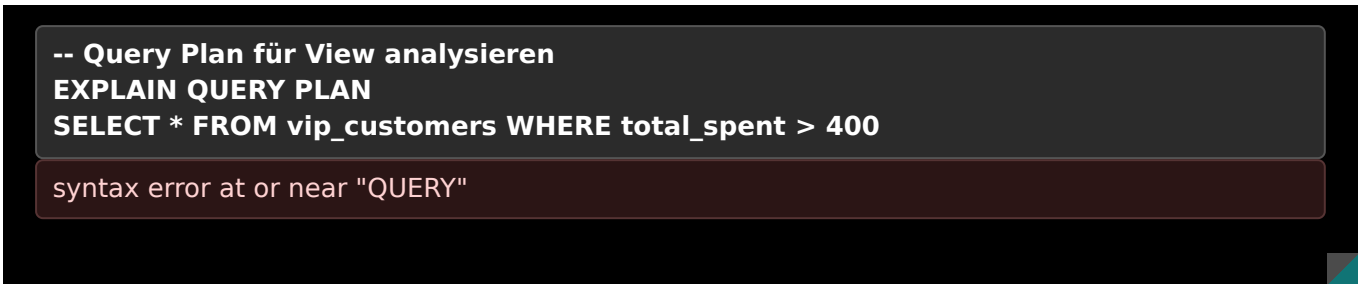
Wann Views langsam werden:

Problem	Symptom	Lösung
Komplexe Aggregationen	Langsame Abfragen	Materialized View erwägen
Viele Joins (5+)	Lange Laufzeit	Query-Optimierung, Indexe
Views auf Views (tief)	Verschachtelte Executions	Flachere Hierarchie
Große Datenmengen	Timeouts	WHERE-Filter, Partitionierung

Performance prüfen:

```
1  -- Query Plan für View analysieren
```

```
EXPLAIN QUERY PLAN
SELECT * FROM vip_customers WHERE total_spent > 400;
```



Der Query Plan zeigt Ihnen, wie die Datenbank die View-Query + Ihre Filter kombiniert. Achten Sie auf Index-Nutzung und Scan-Methoden!

Teil 3: Praktische Patterns & Best Practices

Jetzt, wo Sie beide Konzepte kennen, schauen wir uns praktische Patterns an, die SET Operations und Views kombinieren.

Pattern 1: Layered Views

Strukturieren Sie Views in logischen Schichten – von Basis-Views bis zu Business-Logic-Views.

```
-- Layer 1: Basis-Views (direkte Tabellenzugriffe)
CREATE VIEW base_customers AS
SELECT customer_id, first_name, last_name, email, location_id
FROM customers;

CREATE VIEW base_employees AS
SELECT employee_id, first_name, last_name, email, department
FROM employees;

-- Layer 2: Business-Logic Views
CREATE VIEW all_persons AS
  SELECT
    customer_id AS person_id,
    first_name,
    last_name,
    email,
    'Customer' AS type,
    NULL AS department
  FROM base_customers
  UNION ALL
  SELECT
    employee_id AS person_id,
    first_name,
    last_name,
    email,
    'Employee' AS type,
    department
  FROM base_employees;
```

```sql

-- Layer 3: Report-Views (User-facing)
CREATE VIEW berlin_persons AS
SELECT p.*
FROM all_persons p
WHERE type = 'Customer'
  AND person_id IN (
    SELECT customer_id FROM customers c
    INNER JOIN locations l ON c.location_id = l.location_id
    WHERE l.city = 'Berlin'
  );

-- Reports nutzen Layer 3
SELECT * FROM berlin_persons;
```

```
-- Layer 1: Basis-Views (direkte Tabellenzugriffe)
CREATE VIEW base_customers AS
SELECT customer_id, first_name, last_name, email, location_id
FROM customers
```

ok

```
CREATE VIEW base_employees AS
SELECT employee_id, first_name, last_name, email, department
FROM employees
```

ok

```
-- Layer 2: Business-Logic Views
CREATE VIEW all_persons AS
  SELECT
    customer_id AS person_id,
    first_name,
    last_name,
    email,
    'Customer' AS type,
    NULL AS department
  FROM base_customers
  UNION ALL
  SELECT
    employee_id AS person_id,
    first_name,
    last_name,
    email,
    'Employee' AS type,
    department
  FROM base_employees
```

ok

```
-- Layer 3: Report-Views (User-facing)
CREATE VIEW berlin_persons AS
SELECT p.*
FROM all_persons p
WHERE type = 'Customer'
  AND person_id IN (
    SELECT customer_id FROM customers c
    INNER JOIN locations l ON c.location_id = l.location_id
    WHERE l.city = 'Berlin'
  )
```

ok

```
-- Reports nutzen Layer 3
SELECT * FROM berlin_persons
```

#	person_id	first_name	last_name	email	type	departme
1	1	Alice	Smith	alice.smith@example.com	Customer	*null*
2	3	Carol	Williams	carol.williams@example.com	Customer	*null*

2 rows

Diese Architektur ist wartbar: Änderungen in Layer 1 propagieren automatisch nach oben, jede Schicht hat eine klare Verantwortung!

Pattern 2: Views für Data Quality

Nutzen Sie Views, um Datenqualitäts-Regeln zentral zu definieren.

```sql
-- View: Nur valide Bestellungen
CREATE VIEW valid_orders AS
SELECT
  o.*,
  c.first_name,
  c.last_name
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
WHERE o.total_amount > 0
  AND o.order_date IS NOT NULL
  AND o.status IN ('processing', 'delivered', 'shipped');

-- Alle Reports nutzen nur valide Daten
SELECT * FROM valid_orders;
```

```
-- View: Nur valide Bestellungen
CREATE VIEW valid_orders AS
SELECT
  o.*,
  c.first_name,
  c.last_name
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id
WHERE o.total_amount > 0
  AND o.order_date IS NOT NULL
  AND o.status IN ('processing', 'delivered', 'shipped')
```

ok

```
-- Alle Reports nutzen nur valide Daten
SELECT * FROM valid_orders
```

#	order_id	customer_id	order_date	total_amount	status	first_name	last_na
1	101	1	2024-01-15	299.99	delivered	Alice	Smith
2	102	1	2024-02-20	139.97	delivered	Alice	Smith
3	103	2	2024-01-22	999.99	delivered	Bob	Johnson
4	104	3	2024-03-01	749.94	processing	Carol	Williams
5	105	4	2024-02-10	199.99	delivered	David	Lee
6	106	1	2024-03-20	199.99	delivered	Alice	Smith

6 rows

Data-Quality-Regeln sind jetzt zentral! Ändern Sie die Definition in der View, und alle abhängigen Queries nutzen automatisch die neuen Regeln.

Pattern 3: Views als API-Schicht

Views können als stabile API für externe Systeme dienen – auch wenn sich das interne Schema ändert.

```sql
-- API-View: Stabile Schnittstelle für externe Systeme
CREATE VIEW api_products AS
SELECT
  product_id AS id,
  product_name AS name,
  price,
  CASE
    WHEN price < 50 THEN 'Budget'
    WHEN price < 200 THEN 'Mid-Range'
    ELSE 'Premium'
  END AS price_category
FROM products;
```

```
13
14  -- Externe API liest nur aus dieser View
15  SELECT * FROM api_products;
```

```
-- API-View: Stabile Schnittstelle für externe Systeme
CREATE VIEW api_products AS
SELECT
  product_id AS id,
  product_name AS name,
  price,
  CASE
    WHEN price < 50 THEN 'Budget'
    WHEN price < 200 THEN 'Mid-Range'
    ELSE 'Premium'
  END AS price_category
FROM products
```

ok

```
-- Externe API liest nur aus dieser View
SELECT * FROM api_products
```

#	id	name	price	price_category
1	1	Laptop	999.99	Premium
2	2	Mouse	29.99	Budget
3	3	Keyboard	79.99	Mid-Range
4	4	Monitor	299.99	Premium
5	5	Desk Chair	199.99	Mid-Range
6	6	Notebook	9.99	Budget
7	7	USB Cable	14.99	Budget
8	8	Desk Lamp	39.99	Budget
9	9	Paper (500 sheets)	12.99	Budget

9 rows

Wenn Sie später die products-Tabelle umbenennen oder umstrukturieren, müssen Sie nur die View anpassen – externe Systeme merken nichts!

Vergleichstabelle: Wann was?

Fassen wir zusammen: Wann nutzen Sie SET Operations, wann Views?

Anforderung	Empfehlung	Begründung
Zwei Listen kombinieren	**UNION / UNION ALL**	Mengenvereinigung
Überschneidungen finden	**INTERSECT**	Schnittmenge
Exklusive Elemente finden	**EXCEPT** oder **LEFT JOIN + NULL**	Differenz
Query wiederverwenden	**VIEW**	Code-Reuse
Komplexe Query kapseln	**VIEW**	Abstraktion
Zugriffskontrolle	**VIEW**	Spalten/Zeilen beschränken
SET Ops wiederverwenden	**VIEW mit SET Ops**	Beste aus beiden Welten
Performance bei teuren Queries	**Materialized View** (andere DBs)	Caching

Best Practices Cheat Sheet

Abschließend die wichtigsten Best Practices auf einen Blick.

DO's:

- ✅ Nutze **UNION ALL** statt UNION, wenn Duplikate OK sind
- ✅ Nutze **Views** für wiederverwendbare Queries
- ✅ Nutze **klare Benennungskonventionen** (`vw_`, `_view`)
- ✅ **Dokumentiere** Views (Zweck, Author, Datum)
- ✅ Nutze **Views für Zugriffskontrolle**
- ✅ Kombiniere **SET Ops und Views** für Flexibilität
- ✅ **Teste Performance** mit realistischen Datenmengen
- ✅ Nutze **WHERE-Filter VOR SET Ops** (Performance)

DON'Ts:

- ❌ Keine **tiefen View-Hierarchien** (max. 2-3 Ebenen)
- ❌ Kein **SELECT *** in View-Definitionen (Wartbarkeit)
- ❌ Keine **vergessenen Views** (View Sprawl)
- ❌ Kein **UNION**, wenn UNION ALL reicht
- ❌ Keine **INTERSECT/EXCEPT**, wenn JOIN lesbarer ist
- ❌ Keine **ungetesteten Views** in Produktion

Wrap-up & Zusammenfassung

Fassen wir zusammen, was Sie heute gelernt haben.

SET Operations:

- **UNION** kombiniert Listen (mit oder ohne Duplikate)
- **INTERSECT** findet Gemeinsamkeiten
- **EXCEPT** findet Unterschiede
- Performance: UNION ALL > UNION, Filter vorher anwenden

Views:

- Virtuelle Tabellen ohne Datenspeicherung
- Code-Wiederverwendung und Abstraktion
- Zugriffskontrolle und API-Schicht
- Performance = Basistabellen-Performance

Kombinationen:

- Views können SET Operations enthalten
- Views können mit SET Operations kombiniert werden
- Layered Architecture möglich

> **Pro-Tipp für Ihre Projekte:** Beginnen Sie früh mit Views für häufige Analysen. Das spart später enorm Zeit und reduziert Bugs durch Code-Duplikation!

Referenzen & Weiterführendes:

- SQL Standards: ISO/IEC 9075 (SET Operations seit SQL-92)
- PostgreSQL Views Dokumentation
- SQL Performance Explained (Markus Winand)
- Database Design Patterns (Fowler et al.)