合肥大學
HEFEI UNIVERSITY

Datenbanken

10. Fabrik-Datenbank: Joins und Views

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-07-10

Databases

Dies ist ein Kurs über Datenbanken an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/databases (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielen finden Sie unter https://github.com/thomasWeise/databasesCode.

Outline

Einleitung

Positive Seite

- Bisher läuft es ganz gut.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.
- Das ist schonmal sehr schön.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.
- Das ist schonmal sehr schön.
- Wir haben auch gelernt, wie wir die Integrität unserer Daten schützen können.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.
- Das ist schonmal sehr schön.
- Wir haben auch gelernt, wie wir die Integrität unserer Daten schützen können.
- Auf der einen Seite kennen wir Datentypen wie `DECIMAL` und `DATE`, die sicherstellen, das unsere Daten bestimmte Kriterien einhalten.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.
- Das ist schonmal sehr schön.
- Wir haben auch gelernt, wie wir die Integrität unserer Daten schützen können.
- Auf der einen Seite kennen wir Datentypen wie `DECIMAL` und `DATE`, die sicherstellen, das unsere Daten bestimmte Kriterien einhalten.
- Auf der anderen Seite kennen wir Einschränkungen wie `NOT NULL`, `UNIQUE`, `CONSTRAINT`, und `REFERENCES`.

Positive Seite

- Bisher läuft es ganz gut.
- Wir haben gelernt, wie wir die Daten, die in einem Managementsystem für eine Fabrik anfallen, in unterschiedliche Kategorien einteilen können.
- Wir haben gelernt, wie wir mit dem DBMS PostgreSQL interagieren können und wie wir für jede dieser Datenkategorien eine Tabelle in einer Datenbank erstellen können.
- Das ist schonmal sehr schön.
- Wir haben auch gelernt, wie wir die Integrität unserer Daten schützen können.
- Auf der einen Seite kennen wir Datentypen wie `DECIMAL` und `DATE`, die sicherstellen, das unsere Daten bestimmte Kriterien einhalten.
- Auf der anderen Seite kennen wir Einschränkungen wie `NOT NULL`, `UNIQUE`, `CONSTRAINT`, und `REFERENCES`.
- Damit sind Datenbanken Spreadsheets, wie wir sie mit Microsoft Excel oder LibreOffice Calc erstellen können, weit überlegen.

Negative Seite

- Auf der anderen Seite ist das Arbeiten mit Datenbanken kompliziert.

Negative Seite

- Auf der anderen Seite ist das Arbeiten mit Datenbanken kompliziert.
- Die Interaktion mit dem PostgreSQL-Server über psql ist nicht einfach und andere DBMS sind auch nicht einfacher zu bedienen.

Negative Seite

- Auf der anderen Seite ist das Arbeiten mit Datenbanken kompliziert.
- Die Interaktion mit dem PostgreSQL-Server über psql ist nicht einfach und andere DBMS sind auch nicht einfacher zu bedienen.
- Ärgerlich ist, dass Daten, die mehrere Tabellen verbinden, schwer zu verstehen sind.

Negative Seite

- Ärgerlich ist, dass Daten, die mehrere Tabellen verbinden, schwer zu verstehen sind.
- Wir erinnern uns an die Tabelle `demand`:

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf insert_into_table_demand.sql
 id | customer | product | amount | ordered
----+----------+---------+--------+---------
(0 rows)

INSERT 0 8
 id | customer | product | amount |  ordered
----+----------+---------+--------+------------
  1 |        1 |       7 |     12 | 2024-11-21
  2 |        2 |       3 |      2 | 2024-12-09
  3 |        3 |       2 |      7 | 2024-12-16
  4 |        2 |       5 |      7 | 2024-12-30
  5 |        1 |       5 |      3 | 2025-01-05
  6 |        2 |       6 |      4 | 2025-01-12
  7 |        3 |      11 |     10 | 2025-01-16
  8 |        2 |       3 |      6 | 2025-02-05
(8 rows)

# psql 16.9 succeeded with exit code 0.
```

Negative Seite

- Ärgerlich ist, dass Daten, die mehrere Tabellen verbinden, schwer zu verstehen sind.
- Wir erinnern uns an die Tabelle `demand`.
- Nun wollen wir also lernen, wie wir die Daten aus verschiedenen Tabellen zusammenführen und klar verständlich präsentieren können.

LEFT JOIN

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.
- Mit anderen Worten, wir wollen eine SQL-Anfrage, die zwei Spalten produziert.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.
- Mit anderen Worten, wir wollen eine SQL-Anfrage, die zwei Spalten produziert.
- In der ersten Spalte – nennen wir sie `name` – wollen wir die Namen der Kunden.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.
- Mit anderen Worten, wir wollen eine SQL-Anfrage, die zwei Spalten produziert.
- In der ersten Spalte – nennen wir sie `name` – wollen wir die Namen der Kunden.
- In der zweiten Spalte – nennen wir sie `demand_id` – wollen wir die `id`-Werte der Bestellungen des Kunden.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.
- Mit anderen Worten, wir wollen eine SQL-Anfrage, die zwei Spalten produziert.
- In der ersten Spalte – nennen wir sie `name` – wollen wir die Namen der Kunden.
- In der zweiten Spalte – nennen wir sie `demand_id` – wollen wir die `id`-Werte der Bestellungen des Kunden.
- Wenn Kunden mehrere Bestellungen aufgegeben haben, dann werden sie mehrmals im Ergebnis auftauchen, jeweils mit einer Bestellung.

Bestellungen der Kunden

- Wir wollen nun eine Liste unserer Kunden.
- Für jeden Kunden wollen wir die `id`-Werte aller ihrer Bestellungen.
- Mit anderen Worten, wir wollen eine SQL-Anfrage, die zwei Spalten produziert.
- In der ersten Spalte – nennen wir sie `name` – wollen wir die Namen der Kunden.
- In der zweiten Spalte – nennen wir sie `demand_id` – wollen wir die `id`-Werte der Bestellungen des Kunden.
- Wenn Kunden mehrere Bestellungen aufgegeben haben, dann werden sie mehrmals im Ergebnis auftauchen, jeweils mit einer Bestellung.
- Wenn ein Kunde keine Bestellungen aufgegeben hat, dann soll er trotzdem auftauchen, aber nur einmal, mit einem `NULL`-Wert als `demand_id`.

Erstellen der Anfrage

- Im ersten Schritt brauchen wir also alle Kundennamen.

Erstellen der Anfrage

- Im ersten Schritt brauchen wir also alle Kundennamen.
- `SELECT name FROM customer;` macht das.

Erstellen der Anfrage

- Im ersten Schritt brauchen wir also alle Kundennamen.
- `SELECT name FROM customer;` macht das.
- Nun müssen wir die `id`-Werte der Tabelle `customer` mit der Spalte `customer` der Tabelle `demand` referenzieren.

Erstellen der Anfrage

- Im ersten Schritt brauchen wir also alle Kundennamen.
- `SELECT name FROM customer;` macht das.
- Nun müssen wir die `id`-Werte der Tabelle `customer` mit der Spalte `customer` der Tabelle `demand` referenzieren.
- Das geht mit einem so-genannten `LEFT JOIN`[3].

LEFT JOIN: Syntax

```
-- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
-- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
-- Table_2 kommen sollten, mit NULL belegt.
SELECT Table_1.x, Table_2.y FROM Table_1
    LEFT JOIN Table_2 ON (Table_1.a = Table_2.b);

--     Table_1          Table_2          Query Result
--     a | x            b | y            x | y
--    ---+--------     ---+-----     -------+--------
--     1 | Hello        1 | A          Hello | A
--     2 | World        2 | B          World | B
--     3 | from         2 | C          World | E
--     4 | HFUU         4 | D          from  | <NULL>
--                      2 | E          HFUU  | D
```

LEFT JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a`.

```
1  -- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
11 -- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
12 -- Table_2 kommen sollten, mit NULL belegt.
13 SELECT Table_1.x, Table_2.y FROM Table_1
```

LEFT JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.

```
-- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
-- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
-- Table_2 kommen sollten, mit NULL belegt.
SELECT Table_1.x, Table_2.y FROM Table_1
```

LEFT JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.

```
1  -- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
11 -- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
12 -- Table_2 kommen sollten, mit NULL belegt.
13 SELECT Table_1.x, Table_2.y FROM Table_1
```

LEFT JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.

```
1  -- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
11 -- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
12 -- Table_2 kommen sollten, mit NULL belegt.
13 SELECT Table_1.x, Table_2.y FROM Table_1
```

LEFT JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.
- Gibt es keine solche Zeile, so trotzdem eine Ergebniszeile generiert, jedoch alle Werte, die vielleicht aus `Table_2` übernommen würden, werden als `NULL` angegeben.

```
1  -- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
11 -- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
12 -- Table_2 kommen sollten, mit NULL belegt.
13 SELECT Table_1.x, Table_2.y FROM Table_1
```

`LEFT JOIN`: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.
- Gibt es keine solche Zeile, so trotzdem eine Ergebniszeile generiert, jedoch alle Werte, die vielleicht aus `Table_2` übernommen würden, werden als `NULL` angegeben.
- Beachten Sie: Um Klarheit zu schaffen, können wir den Tabellenname als Prefix, gefolgt von einem Punkt, dem Spaltennamen voranstellen.

`LEFT JOIN`: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 LEFT JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.
- Gibt es keine solche Zeile, so trotzdem eine Ergebniszeile generiert, jedoch alle Werte, die vielleicht aus `Table_2` übernommen würden, werden als `NULL` angegeben.
- Beachten Sie: Um Klarheit zu schaffen, können wir den Tabellenname als Prefix, gefolgt von einem Punkt, dem Spaltennamen voranstellen.
- `Table_1.a` bedeutet also "Spalte `a` aus Tabelle `Table_1`".

LEFT JOIN: Syntax

```
-- Die Syntax eines LEFT JOIN. The Syntax of a LEFT JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2 werden
-- Ergebniszeilen generiert, bei denen alle Werte, die eigentlich von
-- Table_2 kommen sollten, mit NULL belegt.
SELECT Table_1.x, Table_2.y FROM Table_1
    LEFT JOIN Table_2 ON (Table_1.a = Table_2.b);

--     Table_1          Table_2          Query Result
--     a | x            b | y            x | y
--    ---+--------     ---+-----     -------+--------
--     1 | Hello        1 | A          Hello | A
--     2 | World        2 | B          World | B
--     3 | from         2 | C          World | E
--     4 | HFUU         4 | D          from  | <NULL>
--                      2 | E          HFUU  | D
```

LEFT JOIN von customer mit demand

- Übertragen wir also diese Syntax auf unsere Situation.

```sql
/* Get the demands per customer. */

-- List all demand IDs for each customer.
-- LEFT JOIN: 'Bobbo' also appears once, with a NULL demand.
SELECT customer.name AS name, demand.id AS demand_id FROM customer
    LEFT JOIN demand ON (customer.id = demand.customer)
    ORDER BY name;
```

LEFT JOIN von customer mit demand

- Übertragen wir also diese Syntax auf unsere Situation.

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_customer_demand_1.sql
 name  | demand_id
-------+-----------
 Bebba |         7
 Bebba |         3
 Bebbo |         8
 Bebbo |         4
 Bebbo |         2
 Bebbo |         6
 Bibbo |         1
 Bibbo |         5
 Bobbo |
(9 rows)

# psql 16.9 succeeded with exit code 0.
```

Zählen der Bestellungen

- Jetzt wollen wir zählen, wie viele Bestellungen jeder Kunde bisher gemacht hat.

Zählen der Bestellungen

- Jetzt wollen wir zählen, wie viele Bestellungen jeder Kunde bisher gemacht hat.
- Wir kennen inzwischen alles, was wir dazu brauchen:

Zählen der Bestellungen

- Jetzt wollen wir zählen, wie viele Bestellungen jeder Kunde bisher gemacht hat.
- Wir kennen inzwischen alles, was wir dazu brauchen:
- Mit `LEFT JOIN` von der Tabelle `customer` auf die Tabelle `demand` bekommen wir die Bestellungen pro Kunde.

Zählen der Bestellungen

- Jetzt wollen wir zählen, wie viele Bestellungen jeder Kunde bisher gemacht hat.
- Wir kennen inzwischen alles, was wir dazu brauchen:
- Mit `LEFT JOIN` von der Tabelle `customer` auf die Tabelle `demand` bekommen wir die Bestellungen pro Kunde.
- Alles, was dann zu tun ist, ist die Daten nache Kundenname zu gruppieren (`GROUP BY name`).

Zählen der Bestellungen

- Jetzt wollen wir zählen, wie viele Bestellungen jeder Kunde bisher gemacht hat.
- Wir kennen inzwischen alles, was wir dazu brauchen:
- Mit `LEFT JOIN` von der Tabelle `customer` auf die Tabelle `demand` bekommen wir die Bestellungen pro Kunde.
- Alles, was dann zu tun ist, ist die Daten nache Kundenname zu gruppieren (`GROUP BY name`).
- Dann können wir die Zeilen pro `name`-Wert zählen, mit `COUNT(*)`.

LEFT JOIN: Bestellungen pro Kunde zählen

```
/* Get the number of demands per customer. */

-- Now we count the demands.
SELECT customer.name AS name, COUNT(*) AS demands FROM customer
    LEFT JOIN demand ON (customer.id = demand.customer)
    GROUP BY name ORDER BY name;
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_customer_demand_2.sql
 name  | demands
-------+---------
 Bebba |       2
 Bebbo |       4
 Bibbo |       2
 Bobbo |       1
(4 rows)

# psql 16.9 succeeded with exit code 0.
```

INNER JOIN

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.
- Erstellen wir eine Anfrage, die für jede Bestellung den Kundenname, Produktname, Preise, bestellte Anzahl, und Bestelldatum heraussucht.

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.
- Erstellen wir eine Anfrage, die für jede Bestellung den Kundenname, Produktname, Preise, bestellte Anzahl, und Bestelldatum heraussucht.
- Das Ergebnis dieser Anfrage wäre dann eine klare und lesbare Übersicht unserer Geschäftstransaktionen.

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.
- Erstellen wir eine Anfrage, die für jede Bestellung den Kundenname, Produktname, Preise, bestellte Anzahl, und Bestelldatum heraussucht.
- Das Ergebnis dieser Anfrage wäre dann eine klare und lesbare Übersicht unserer Geschäftstransaktionen.
- Basierend auf unserem vorigen Beispiel können wir erwarten, dass wir nun zwei “Joins” brachen.

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.
- Erstellen wir eine Anfrage, die für jede Bestellung den Kundenname, Produktname, Preise, bestellte Anzahl, und Bestelldatum heraussucht.
- Das Ergebnis dieser Anfrage wäre dann eine klare und lesbare Übersicht unserer Geschäftstransaktionen.
- Basierend auf unserem vorigen Beispiel können wir erwarten, dass wir nun zwei "Joins" brachen.
- Wir müssen ja Daten aus allen drei Tabellen – `demand`, `customer`, und `product` – kombinieren.

Verkaufsdaten Zusammenführen

- Wir können nun Daten von verschiedenen Tabellen zusammenführen.
- Erstellen wir eine Anfrage, die für jede Bestellung den Kundenname, Produktname, Preise, bestellte Anzahl, und Bestelldatum heraussucht.
- Das Ergebnis dieser Anfrage wäre dann eine klare und lesbare Übersicht unserer Geschäftstransaktionen.
- Basierend auf unserem vorigen Beispiel können wir erwarten, dass wir nun zwei "Joins" brachen.
- Wir müssen ja Daten aus allen drei Tabellen – `demand`, `customer`, und `product` – kombinieren.
- Die Anfrage wid daher etwas komplizierter werden.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle* `demand` *brauchen wir die entsprechende Zeile in Tabelle* `customer` *und die entsprechende Zeile in Tabelle* `product`*."*

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle `demand` brauchen wir die entsprechende Zeile in Tabelle `customer` und die entsprechende Zeile in Tabelle `product`."*
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `customer` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `product` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle `demand` brauchen wir die entsprechende Zeile in Tabelle `customer` und die entsprechende Zeile in Tabelle `product`."*
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `customer` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `product` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Es gibt keine Zeile in `demand` ohne entsprechende Zeile in `customer` oder `product`.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle `demand` brauchen wir die entsprechende Zeile in Tabelle `customer` und die entsprechende Zeile in Tabelle `product`."*
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `customer` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `product` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Es gibt keine Zeile in `demand` ohne entsprechende Zeile in `customer` oder `product`.
- Wir könnten wieder `LEFT JOIN` verwenden, denn es könnte niemals `NULL` erzeugen.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle `demand` brauchen wir die entsprechende Zeile in Tabelle `customer` und die entsprechende Zeile in Tabelle `product`."*
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `customer` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `product` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Es gibt keine Zeile in `demand` ohne entsprechende Zeile in `customer` oder `product`.
- Wir könnten wieder `LEFT JOIN` verwenden, denn es könnte niemals `NULL` erzeugen.
- Lassen Sie uns aber stattdessen `INNER JOIN`[3] nutzen.

Grundidee

- Wir beginnen, die Anfrage zu konstruieren.
- Was wir wollen ist das: *"Für jede Zeile in Tabelle `demand` brauchen wir die entsprechende Zeile in Tabelle `customer` und die entsprechende Zeile in Tabelle `product`."*
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `customer` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Wir wissen, dass es immer eine entsprechende Zeile in der Tabelle `product` geben muss, denn die `REFERENCES`-Einschränkung erzwingt das.
- Es gibt keine Zeile in `demand` ohne entsprechende Zeile in `customer` oder `product`.
- Wir könnten wieder `LEFT JOIN` verwenden, denn es könnte niemals `NULL` erzeugen.
- Lassen Sie uns aber stattdessen `INNER JOIN`[3] nutzen.
- Anders als `LEFT JOIN` wird `INNER JOIN` nur Zeilen ausgeben, wenn alle Tabellen zueinanderpassende Zeilen haben.

INNER JOIN: Syntax

```sql
-- Die Syntax von einem INNER JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
-- Ergebniszeile generiert.
SELECT Table_1.x, Table_2.y FROM Table_1
    INNER JOIN Table_2 ON (Table_1.a = Table_2.b);

--     Table_1          Table_2          Query Result
--     a | x            b | y            x | y
--    ---+--------     ---+-----     -------+---------
--     1 | Hello        1 | A          Hello | A
--     2 | World        2 | B          World | B
--     3 | from         2 | C          World | C
--     4 | HFUU         4 | D          World | E
--                      2 | E          HFUU  | D
```

INNER JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a`.

```sql
-- Die Syntax von einem INNER JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
-- Ergebniszeile generiert.
SELECT Table_1.x, Table_2.y FROM Table_1
```

INNER JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.

```
-- Die Syntax von einem INNER JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
-- Ergebniszeile generiert.
SELECT Table_1.x, Table_2.y FROM Table_1
```

INNER JOIN: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 INNER JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.

```
1  -- Die Syntax von einem INNER JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
11 -- Ergebniszeile generiert.
12 SELECT Table_1.x, Table_2.y FROM Table_1
```

`INNER JOIN`: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 INNER JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.

```
1  -- Die Syntax von einem INNER JOIN.
2  -- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.
3
4  -- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
5  -- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
6  -- referenziert wird.
7  -- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
8  -- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
9  -- haben.
10 -- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
11 -- Ergebniszeile generiert.
12 SELECT Table_1.x, Table_2.y FROM Table_1
```

`INNER JOIN`: Syntax

- Angenommen, Tabelle `Table_1` hat den Primärschlüssel `a` und Tabelle `Table_2` hat eine Spalte `b`, die als Fremdschlüssel auf `Table_1.a` verweist.
- `SELECT ... FROM Table_1 INNER JOIN Table_2 ON (Table_1.a = Table_2.b)` geht dann Zeile für Zeile durch `Table_1` und sucht für jeden Wert von `Table_1.a` alle passenden Zeilen aus `Table_2` heraus, also alle Zeilen, für die `Table_1.a = Table_2.b` gilt.
- Jede dieser Zeilen wird dann eine Zeile im Ergebnis.
- Gibt es keine solche Zeile, wird anders als bei `LEFT JOIN` *keine* Ergebniszeile generiert.

INNER JOIN: Syntax

```sql
-- Die Syntax von einem INNER JOIN.
-- Wir vereinigen Informationen aus den Tabellen Table_1 und Table_2.

-- Oft, aber nicht immer, hat Table_1 die Spalte 'a' als Primär-
-- schlüssel, der von Spalte 'b' in Table_2 als Fremdschlüssel
-- referenziert wird.
-- In diesem Beispiel geben wir die Werte von Spalte 'x' der Table_1
-- und 'y' von Table_2 zurück, nachdem wir beide Tabellen vereinigt
-- haben.
-- Für Zeilen in Table_1 ohne passende Zeilen in Table_2, wird keine
-- Ergebniszeile generiert.
SELECT Table_1.x, Table_2.y FROM Table_1
    INNER JOIN Table_2 ON (Table_1.a = Table_2.b);

--     Table_1          Table_2          Query Result
--     a | x            b | y            x | y
--    ---+--------     ---+-----     -------+---------
--     1 | Hello        1 | A          Hello | A
--     2 | World        2 | B          World | B
--     3 | from         2 | C          World | C
--     4 | HFUU         4 | D          World | E
--                      2 | E          HFUU  | D
```

INNER JOIN: Mehrere INNER JOIN zusammen

- Um Daten aus mehreren Tabellen zu kombinieren, brauchen wir mehrere Joins.

```
1  -- Die Syntax von *zwei* kombinierten INNER JOINs.
2  -- Wir vereinigen Informationen der Tabellen Table_1, Table_2, Table_3.
3
4  -- Nur wenn Zeilen in allen drei Tabellen passen, werden Ergebniszeilen
5  -- generiert.
6  SELECT Table_1.x, Table_2.y, Table_3.z FROM Table_1
7      INNER JOIN Table_2 ON (Table_1.a = Table_2.b)
8      INNER JOIN Table_3 ON (Table_2.y = Table_3.c);
9
10 --    Table_1          Table_2          Table_3          Query Result
11 --    a | x            b | y            c | z              x | y | z
12 --   ---+--------     ---+----------   ---+----        --------+--------
13 --    1 | Hello        1 | A            A | 10         Hello   | A | 10
14 --    2 | World        2 | B            B | 20         World   | B | 20
15 --    3 | from         2 | C            B | 30         World   | B | 30
16 --    4 | HFUU         4 | D            C | 40         World   | C | 40
17 --                     2 | E            D | 50         HFUU    | D | 50
18 --                                     F | 60
```

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.
- Nun fügen wir einfach das nächste `INNER JOIN` an, in dem wir direkt danach `INNER JOIN product ON (product.id = demand.product)` schreiben.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.
- Nun fügen wir einfach das nächste `INNER JOIN` an, in dem wir direkt danach `INNER JOIN product ON (product.id = demand.product)` schreiben.
- Damit bekommen wir die passende Zeile aus der Tabelle `product`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.
- Nun fügen wir einfach das nächste `INNER JOIN` an, in dem wir direkt danach `INNER JOIN product ON (product.id = demand.product)` schreiben.
- Damit bekommen wir die passende Zeile aus der Tabelle `product`.
- Da wir diesmal sowohl Kunden- als auch Produktnamen haben, selektieren wir `customer.name AS customer_name` und `product.name AS product_name`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.
- Nun fügen wir einfach das nächste `INNER JOIN` an, in dem wir direkt danach `INNER JOIN product ON (product.id = demand.product)` schreiben.
- Damit bekommen wir die passende Zeile aus der Tabelle `product`.
- Da wir diesmal sowohl Kunden- als auch Produktnamen haben, selektieren wir `customer.name AS customer_name` und `product.name AS product_name`.
- Wir selektieren ebenfalls `product.price AS price`, `demand.amount AS amount`, und `demand.ordered as ordered`.

Anfrage Zusammenbauen

- Wir fangen an mit `SELECT <something> FROM demand`.
- Dann schreiben wir `INNER JOIN customer ON (customer.id = demand.customer)`.
- Für jede Zeile in `demand` gibt uns das die passende Zeile in `customer`.
- Nun fügen wir einfach das nächste `INNER JOIN` an, in dem wir direkt danach `INNER JOIN product ON (product.id = demand.product)` schreiben.
- Damit bekommen wir die passende Zeile aus der Tabelle `product`.
- Da wir diesmal sowohl Kunden- als auch Produktnamen haben, selektieren wir `customer.name AS customer_name` und `product.name AS product_name`.
- Wir selektieren ebenfalls `product.price AS price`, `demand.amount AS amount`, und `demand.ordered as ordered`.
- Wir sortieren die Ausgabe mit `ORDER BY` lexikographisch nach `customer_name`, `ordered`, und `product_name`, `price`, und `amount`.

Die Anfrage

```
/* Query the sales information. */

-- We combine the three tables customer, product, and demand.
-- Basically, for each row in 'demand', we find the corresponding rows
-- in the 'customer' and 'product' tables (via the INNER JOIN).
-- This gives us one long row with all the information for one 'demand'.
-- We now choose only some elements of this long row and rename them.
-- We extract the name of the customer and refer to it "customer_name".
-- We extract the name of the product and refer to it as "product_name".
-- We also print the "amount" from each customer demand and the price.
-- We also print when the purchase as made.
SELECT customer.name  AS customer_name, product.name  AS product_name,
       product.price  AS price,         demand.amount AS amount,
       demand.ordered AS ordered
FROM demand INNER JOIN customer ON (customer.id = demand.customer)
            INNER JOIN product  ON (product.id  = demand.product)
ORDER BY customer_name, ordered, product_name, price, amount;
```

Die Anfrage

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_sale_data.sql
 customer_name | product_name   | price  | amount |  ordered
---------------+----------------+--------+--------+------------
 Bebba         | Shoe, Size 37  | 152.99 |      7 | 2024-12-16
 Bebba         | Large Purse    | 150.00 |     10 | 2025-01-16
 Bebbo         | Shoe, Size 38  | 154.99 |      2 | 2024-12-09
 Bebbo         | Shoe, Size 40  | 158.99 |      7 | 2024-12-30
 Bebbo         | Shoe, Size 41  | 160.99 |      4 | 2025-01-12
 Bebbo         | Shoe, Size 38  | 154.99 |      6 | 2025-02-05
 Bibbo         | Shoe, Size 42  | 162.99 |     12 | 2024-11-21
 Bibbo         | Shoe, Size 40  | 158.99 |      3 | 2025-01-05
(8 rows)

# psql 16.9 succeeded with exit code 0.
```

Sichten / Views

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren. Oder pro Produkt.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren. Oder pro Produkt.
- Jedes Mal würden wir eine ähnliche Abfrage schreiben.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren. Oder pro Produkt.
- Jedes Mal würden wir eine ähnliche Abfrage schreiben. Das ist nicht gut.

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren. Oder pro Produkt.
- Jedes Mal würden wir eine ähnliche Abfrage schreiben. Das ist nicht gut.
- Zum Glück gibt es eine Lösung: Wir können Anfragen als sogenannte *Views* (Sichten) speichern![2].

Sichten / Views

- Nun haben wir sehr schön die Verkaufsdaten unserer Fabrik aufbereitet.
- Wahrscheinlich brauchen wir diese Anfrage sehr häufig, wahrscheinlich jedes Mal leicht verändert.
- Vielleicht wollen wir nur Bestellungen aus dem aktuellen Jahr. Dann würden wir eine `WHERE`-Klausel anfügen.
- Vielleicht wollen wir die Geldmenge pro Kunde aufaddieren. Oder pro Produkt.
- Jedes Mal würden wir eine ähnliche Abfrage schreiben. Das ist nicht gut.
- Zum Glück gibt es eine Lösung: Wir können Anfragen als sogenannte *Views* (Sichten) speichern![2].
- Ein View funktioniert im Grunde wie eine Tabelle.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem auf.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.
- Mobiltelefonnummern sind aber eindeutig in unserer Datenbank.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.
- Mobiltelefonnummern sind aber eindeutig in unserer Datenbank.
- Als `customer_name`, werden wir also `customer.name || ', ' || customer.phone` benutzen.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.
- Mobiltelefonnummern sind aber eindeutig in unserer Datenbank.
- Als `customer_name`, werden wir also `customer.name || ', ' || customer.phone` benutzen.
- Der doppelten Strich `||` steht für das aneinanderreihen von Zeichenketten.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.
- Mobiltelefonnummern sind aber eindeutig in unserer Datenbank.
- Als `customer_name`, werden wir also `customer.name || ', ' || customer.phone` benutzen.
- Der doppelten Strich `||` steht für das aneinanderreihen von Zeichenketten. `'Hello ' || 'Word!'` ergibt `'Hello World!'`.

Problem mit unserer Anfrage

- Wir brauchen unsere Anfrage also nur als Sicht zu speichern.
- Als wir das tuen wollen, fällt uns ein Problem aufKundennamen sind nicht notwendigerweise eindeutig (nicht `UNIQUE`).
- Wenn wir unsere Anfrage später nutzen wollen, um Verkäufe pro Kunde aufzuaddieren, dann würden alle Verkäufe nach Kundennamen gruppiert und dann summiert. Wenn zwei Kunden den selben Namen haben, werden ihre Bestellungen vermengt.
- Mobiltelefonnummern sind aber eindeutig in unserer Datenbank.
- Als `customer_name`, werden wir also `customer.name || ', ' || customer.phone` benutzen.
- Der doppelten Strich `||` steht für das aneinanderreihen von Zeichenketten. `'Hello ' || 'Word!'` ergibt `'Hello World!'`.
- Problem gelöst: Da Mobiltelefonnummern eindeutig sind, sind Name + Mobiltelefonnummer auch eindeutig.

Anfrage als Sicht speichern und benutzen

- Wir können unsere Anfrage nun als Sicht speichern.

Anfrage als Sicht speichern und benutzen

- Wir können unsere Anfrage nun als Sicht speichern.
- Das geht sehr einfach: Wir schreiben `CREATE VIEW sale AS ...`, wobei `...` durch unsere Anfrage zu ersetzen ist.

Anfrage als Sicht speichern und benutzen

- Wir können unsere Anfrage nun als Sicht speichern.
- Das geht sehr einfach: Wir schreiben `CREATE VIEW sale AS ...`, wobei `...` durch unsere Anfrage zu ersetzen ist.
- Damit wird unsere Anfrage unter dem Namen `sale` gespeichert.

Anfrage als Sicht speichern und benutzen

- Wir können unsere Anfrage nun als Sicht speichern.
- Das geht sehr einfach: Wir schreiben `CREATE VIEW sale AS ...`, wobei `...` durch unsere Anfrage zu ersetzen ist.
- Damit wird unsere Anfrage unter dem Namen `sale` gespeichert.
- Sie kann dann wie eine Tabelle verwendet werden.

Anfrage als Sicht speichern und benutzen

- Wir können unsere Anfrage nun als Sicht speichern.
- Das geht sehr einfach: Wir schreiben `CREATE VIEW sale AS ...`, wobei `...` durch unsere Anfrage zu ersetzen ist.
- Damit wird unsere Anfrage unter dem Namen `sale` gespeichert.
- Sie kann dann wie eine Tabelle verwendet werden.
- Wir können schreiben `SELECT * from sale;`.

Die Anfrage

```sql
/* Create a view showing the sales information. */

-- We combine the three tables customer, product, and demand.
-- Basically, for each row in 'demand', we find the corresponding rows
-- in the 'customer' and 'product' tables (via the INNER JOIN).
-- This gives us one long row with all the information for one 'demand'.
-- We now choose only some elements of this long row and rename them.
-- We extract the name of the customer combined with their phone number
-- and refer to it "customer_name".
-- We extract the name of the product and refer to it as "product_name".
-- We also print the "amount" from each customer demand and the price.
-- We also print when the purchase as made.
CREATE VIEW sale AS
    SELECT customer.name || ', ' || customer.phone AS customer_name,
           product.name  AS product_name, product.price  AS price,
           demand.amount AS amount,       demand.ordered AS ordered
    FROM demand INNER JOIN customer ON (customer.id = demand.customer)
                INNER JOIN product  ON (product.id  = demand.product)
    ORDER BY customer_name, ordered, product_name, price, amount;

-- We can use the view as if it was a table!
SELECT * from sale;
```

Die Anfrage

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf create_view_sale.sql
CREATE VIEW
    customer_name     | product_name   | price  | amount |  ordered
----------------------+----------------+--------+--------+------------
 Bebba, 33333333333 | Shoe, Size 37 | 152.99 |      7 | 2024-12-16
 Bebba, 33333333333 | Large Purse    | 150.00 |     10 | 2025-01-16
 Bebbo, 55555555555 | Shoe, Size 38 | 154.99 |      2 | 2024-12-09
 Bebbo, 55555555555 | Shoe, Size 40 | 158.99 |      7 | 2024-12-30
 Bebbo, 55555555555 | Shoe, Size 41 | 160.99 |      4 | 2025-01-12
 Bebbo, 55555555555 | Shoe, Size 38 | 154.99 |      6 | 2025-02-05
 Bibbo, 99999999999 | Shoe, Size 42 | 162.99 |     12 | 2024-11-21
 Bibbo, 99999999999 | Shoe, Size 40 | 158.99 |      3 | 2025-01-05
(8 rows)

# psql 16.9 succeeded with exit code 0.
```

Die Sicht Benutzen

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?
- Wir können den Gesamtpreis jeder Bestellung einfach ausrechnen.

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?
- Wir können den Gesamtpreis jeder Bestellung einfach ausrechnen. Wir kennen ja die Anzahl der bestellten Einheiten (`amount`) und den Preis pro Einheit (`price`).
- Pro Zeile müssen wir also einfach `amount * price` berechnen.

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?
- Wir können den Gesamtpreis jeder Bestellung einfach ausrechnen. Wir kennen ja die Anzahl der bestellten Einheiten (`amount`) und den Preis pro Einheit (`price`).
- Pro Zeile müssen wir also einfach `amount * price` berechnen.
- Da wir die Summe dieser Werte pro Kunde bzw. pro Produkt wollen, müssen wir die Zeilen jeweils nach `customer_name` oder `product_name` gruppieren.

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?
- Wir können den Gesamtpreis jeder Bestellung einfach ausrechnen. Wir kennen ja die Anzahl der bestellten Einheiten (`amount`) und den Preis pro Einheit (`price`).
- Pro Zeile müssen wir also einfach `amount * price` berechnen.
- Da wir die Summe dieser Werte pro Kunde bzw. pro Produkt wollen, müssen wir die Zeilen jeweils nach `customer_name` oder `product_name` gruppieren.
- Pro Gruppe berechnen wir dann `SUM(amount * price)`[1].

Benutzen unserer neuen Sicht

- Benutzen wir nun unsere neue Sicht, um folgende Fragen zu beantworten:
  - Welcher Kunde hat den meisten Umsatz gemacht?
  - Welches Produkt hat den meisten Umsatz gemacht?
- Wir können den Gesamtpreis jeder Bestellung einfach ausrechnen. Wir kennen ja die Anzahl der bestellten Einheiten (`amount`) und den Preis pro Einheit (`price`).
- Pro Zeile müssen wir also einfach `amount * price` berechnen.
- Da wir die Summe dieser Werte pro Kunde bzw. pro Produkt wollen, müssen wir die Zeilen jeweils nach `customer_name` oder `product_name` gruppieren.
- Pro Gruppe berechnen wir dann `SUM(amount * price)`[1].
- Schließlich sortieren wir die Werte absteigend, weil wir ja die Bestseller am Anfang haben wollen.

Verkaufserlös nach Kunde

```
/* Get the total sales per customer. */
SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
    GROUP BY customer_name ORDER BY customer_sale DESC;
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_from_view_sale_1a.sql
   customer_name    | customer_sale
--------------------+---------------
 Bebbo, 55555555555 |       2996.81
 Bebba, 33333333333 |       2570.93
 Bibbo, 99999999999 |       2432.85
(3 rows)

# psql 16.9 succeeded with exit code 0.
```

Verkaufserlös nach Produkt

```sql
/* Get the total sales per product. */
SELECT product_name, SUM(amount) AS total_amount,
    SUM(amount * price) AS product_sale FROM sale
    GROUP BY product_name ORDER BY product_sale DESC;
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_from_view_sale_1b.sql
 product_name  | total_amount | product_sale
---------------+--------------+--------------
 Shoe, Size 42 |           12 |      1955.88
 Shoe, Size 40 |           10 |      1589.90
 Large Purse   |           10 |      1500.00
 Shoe, Size 38 |            8 |      1239.92
 Shoe, Size 37 |            7 |      1070.93
 Shoe, Size 41 |            4 |       643.96
(6 rows)

# psql 16.9 succeeded with exit code 0.
```

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.
- Man kann es sich so vorstellen: Wenn wir ein `SELECT` über sie ausführen, dann wird intern zuerst die Anfrage unserer Sicht ausgeführt und unser `SELECT` dann darüber angewendet.

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.
- Man kann es sich so vorstellen: Wenn wir ein `SELECT` über sie ausführen, dann wird intern zuerst die Anfrage unserer Sicht ausgeführt und unser `SELECT` dann darüber angewendet.
- Nehmen wir nun an, das unsere Datenbank schon mehrere Jahre existiert.

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.
- Man kann es sich so vorstellen: Wenn wir ein `SELECT` über sie ausführen, dann wird intern zuerst die Anfrage unserer Sicht ausgeführt und unser `SELECT` dann darüber angewendet.
- Nehmen wir nun an, das unsere Datenbank schon mehrere Jahre existiert.
- Wir wollen dann vielleicht nicht alle Verkäufe, die jemals stattfanden, aufsummieren.

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.
- Man kann es sich so vorstellen: Wenn wir ein `SELECT` über sie ausführen, dann wird intern zuerst die Anfrage unserer Sicht ausgeführt und unser `SELECT` dann darüber angewendet.
- Nehmen wir nun an, das unsere Datenbank schon mehrere Jahre existiert.
- Wir wollen dann vielleicht nicht alle Verkäufe, die jemals stattfanden, aufsummieren.
- Stattdessen wollen wir vielleicht nur die Verkäufe im Jahr 2025 beachten.

Weitere Schritte

- Wie konnten die Sicht tatsächlich wie eine Tabelle verwenden.
- Man kann es sich so vorstellen: Wenn wir ein `SELECT` über sie ausführen, dann wird intern zuerst die Anfrage unserer Sicht ausgeführt und unser `SELECT` dann darüber angewendet.
- Nehmen wir nun an, das unsere Datenbank schon mehrere Jahre existiert.
- Wir wollen dann vielleicht nicht alle Verkäufe, die jemals stattfanden, aufsummieren.
- Stattdessen wollen wir vielleicht nur die Verkäufe im Jahr 2025 beachten.
- Alles, was wir dafür ändern müssen, ist eine `WHERE`-Klausel an unsere Anfragen anhängen, mit der Bedingung `(ordered >= '2025-01-01')` and `(ordered < '2026-01-01')`.

Verkaufserlös nach Kunde in 2025


```sql
/* Get the total sales per customer in 2025. */
SELECT customer_name, SUM(amount * price) AS customer_sale FROM sale
    WHERE (ordered >= '2025-01-01') AND (ordered < '2026-01-01')
    GROUP BY customer_name ORDER BY customer_sale DESC;
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf select_from_view_sale_2a.sql
    customer_name    | customer_sale
---------------------+---------------
 Bebbo, 55555555555 |       1573.90
 Bebba, 33333333333 |       1500.00
 Bibbo, 99999999999 |        476.97
(3 rows)

# psql 16.9 succeeded with exit code 0.
```

Verkaufserlös nach Produkt in 2025

```sql
/* Get the total sales per product in 2025. */
SELECT product_name, SUM(amount) AS total_amount,
    SUM(amount * price) AS product_sale FROM sale
    WHERE (ordered >= '2025-01-01') AND (ordered < '2026-01-01')
    GROUP BY product_name ORDER BY product_sale DESC;
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf select_from_view_sale_2b.sql
 product_name  | total_amount | product_sale
---------------+--------------+--------------
 Large Purse   |           10 |      1500.00
 Shoe, Size 38 |            6 |       929.94
 Shoe, Size 41 |            4 |       643.96
 Shoe, Size 40 |            3 |       476.97
(4 rows)

# psql 16.9 succeeded with exit code 0.
```

Zusammenfassung

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.
- Das ist eines der wichtigsten Features, das Datenbanken uns bieten.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.
- Das ist eines der wichtigsten Features, das Datenbanken uns bieten.
- Wir haben auch gelernt, wie wir Sichten (Views) erstellen können, also gespeicherte Anfragen.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.
- Das ist eines der wichtigsten Features, das Datenbanken uns bieten.
- Wir haben auch gelernt, wie wir Sichten (Views) erstellen können, also gespeicherte Anfragen.
- Damit ist es uns nicht nur möglich, bestimmte, oft benötigte Anfragen zu speichern um unnötiges, mehrmaliges eingeben zu verhindern.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.
- Das ist eines der wichtigsten Features, das Datenbanken uns bieten.
- Wir haben auch gelernt, wie wir Sichten (Views) erstellen können, also gespeicherte Anfragen.
- Damit ist es uns nicht nur möglich, bestimmte, oft benötigte Anfragen zu speichern um unnötiges, mehrmaliges eingeben zu verhindern.
- Sichten sind auch ein zentrales Werkzeug, um verschiedenen Werkzeugen und Benutzern ... nun ... verschiedene Sichten auf die Daten zu bieten.

Zusammenfassung

- Wir sind nun ein ganzes Stück stärker geworden.
- Mit `JOIN`s sind wir nun in der Lage, Daten aus verschiedenen Tabellen zusammenzufügen.
- Das ist eines der wichtigsten Features, das Datenbanken uns bieten.
- Wir haben auch gelernt, wie wir Sichten (Views) erstellen können, also gespeicherte Anfragen.
- Damit ist es uns nicht nur möglich, bestimmte, oft benötigte Anfragen zu speichern um unnötiges, mehrmaliges eingeben zu verhindern.
- Sichten sind auch ein zentrales Werkzeug, um verschiedenen Werkzeugen und Benutzern ... nun ... verschiedene Sichten auf die Daten zu bieten.
- Wir können sie nutzen, um bestimmten Nutzern oder Applikationen nur bestimmte Teile unserer gespeicherten Daten zugänglich zu machen.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] "Aggregate Functions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.21. URL: https://www.postgresql.org/docs/17/functions-aggregate.html (besucht am 2025-02-27) (siehe S. 108–115).

[2] "`CREATE VIEW`". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-createview.html (besucht am 2025-03-03) (siehe S. 80–89).

[3] "Joined Tables". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 7.2.1.1. URL: https://www.postgresql.org/docs/17/queries-table-expressions.html#QUERIES-JOIN (besucht am 2025-03-01) (siehe S. 26–29, 54–60).

[4] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (besucht am 2025-02-25).