合肥大學
HEFEI UNIVERSITY

# Datenbanken

## 10. Fabrik-Datenbank: Tabelle *demand*

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

# Einleitung

- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.

# Einleitung

- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.
- Als nächstes wollen wir auch Bestellungen speichern können.

# Einleitung

- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.
- Als nächstes wollen wir auch Bestellungen speichern können.
- Wir wählen dafür ein extrem simplifiziertes Konzept.

# Einleitung

- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.
- Als nächstes wollen wir auch Bestellungen speichern können.
- Wir wählen dafür ein extrem simplifiziertes Konzept Ein Kunde kann mit einer Bestellung eine Menge von genau einem Produkt ordern.

# Einleitung

- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.
- Als nächstes wollen wir auch Bestellungen speichern können.
- Wir wählen dafür ein extrem simplifiziertes Konzept Ein Kunde kann mit einer Bestellung eine Menge von genau einem Produkt ordern.
- In echten Online-Shops können Bestellungen aus mehreren Produkten bestehen, Zahlungsoptionen haben, und vielleicht auch Lieferadressen...

# Einleitung


- Jetzt können wir sowohl Kunden als auch Produkte in unserer Datenbank speichern.
- Als nächstes wollen wir auch Bestellungen speichern können.
- Wir wählen dafür ein extrem simplifiziertes Konzept Ein Kunde kann mit einer Bestellung eine Menge von genau einem Produkt ordern.
- In echten Online-Shops können Bestellungen aus mehreren Produkten bestehen, Zahlungsoptionen haben, und vielleicht auch Lieferadressen... ...wir wollen unser Beispiel aber schön einfach halten.

# Erstellen der Tabelle

## Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`.

## Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...

**Gute Praxis**

Never use Structured Query Language (SQL) keywords or reserved words as names, e.g., for columns or tables[1].

## Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...
- Die Tabelle muss für jede Bestellung folgende Daten speichern

# Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...
- Die Tabelle muss für jede Bestellung folgende Daten speichern:
  - den Kunde, der die Bestellung aufgibt

# Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...
- Die Tabelle muss für jede Bestellung folgende Daten speichern:
  - den Kunde, der die Bestellung aufgibt,
  - das bestellte Produkt

# Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...
- Die Tabelle muss für jede Bestellung folgende Daten speichern:
  - den Kunde, der die Bestellung aufgibt,
  - das bestellte Produkt,
  - die Anzahl der bestellten Einheiten des Produktes

# Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`. Ich würde sie lieber `order` nennen, aber das ist ja ein SQL-Schlüsselwort...
- Die Tabelle muss für jede Bestellung folgende Daten speichern:
  - den Kunde, der die Bestellung aufgibt,
  - das bestellte Produkt,
  - die Anzahl der bestellten Einheiten des Produktes, und
  - und das Datum, an dem die Bestellung aufgegeben wurde.

## Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`.
- Insgesamt könnten die Daten also wie folgt aussehen:

## Grobstruktur

- Wir nennen unsere Tabelle für Bestellungen `demand`.
- In den Spalten für Kunde und Produkt werden die Primärschlüssel aus den entsprechenden Tabellen stehen...

| id | customer | product | amount | ordered |
|---|---|---|---|---|
| 1 | 1 (Bibbo) | 7 (Shoe, Size 42) | 12 | 2024-11-21 |
| 2 | 2 (Bebbo) | 3 (Shoe, Size 38) | 2 | 2024-12-09 |
| 3 | 3 (Bebba) | 2 (Shoe, Size 37) | 7 | 2024-12-16 |
| 4 | 2 (Bebbo) | 5 (Shoe, Size 40) | 7 | 2024-12-30 |
| 5 | 1 (Bibbo) | 5 (Shoe, Size 40) | 3 | 2025-01-05 |
| 6 | 2 (Bebbo) | 6 (Shoe, Size 41) | 4 | 2025-01-12 |
| 7 | 3 (Bebba) | 11 (Large Purse) | 10 | 2025-01-16 |
| 8 | 2 (Bebbo) | 3 (Shoe, Size 38) | 6 | 2025-02-05 |
| ... | ... | ... | ... | ... |

## Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf create_table_demand.sql
CREATE TABLE
 tablename
-----------
 product
 customer
 demand
(3 rows)

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

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.
- Wir haben dieses Kommando via psql an den PostgreSQL-Server geschickt und das Erstellen der Tabelle hat funktioniert.

```sql
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.
- Wir haben dieses Kommando via psql an den PostgreSQL-Server geschickt und das Erstellen der Tabelle hat funktioniert.
- Wieder gibt es einige interessante neue Komponenten.

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.
- Wir haben dieses Kommando via psql an den PostgreSQL-Server geschickt und das Erstellen der Tabelle hat funktioniert.
- Wieder gibt es einige interessante neue Komponenten: Was bedeutet `REFERENCES`?

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Erstellen der Tabelle mit `CREATE TABLE`

- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.
- Wir haben dieses Kommando via psql an den PostgreSQL-Server geschickt und das Erstellen der Tabelle hat funktioniert.
- Wieder gibt es einige interessante neue Komponenten: Was bedeutet `REFERENCES`? Was ist ein `DATE`?

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Erstellen der Tabelle mit `CREATE TABLE`

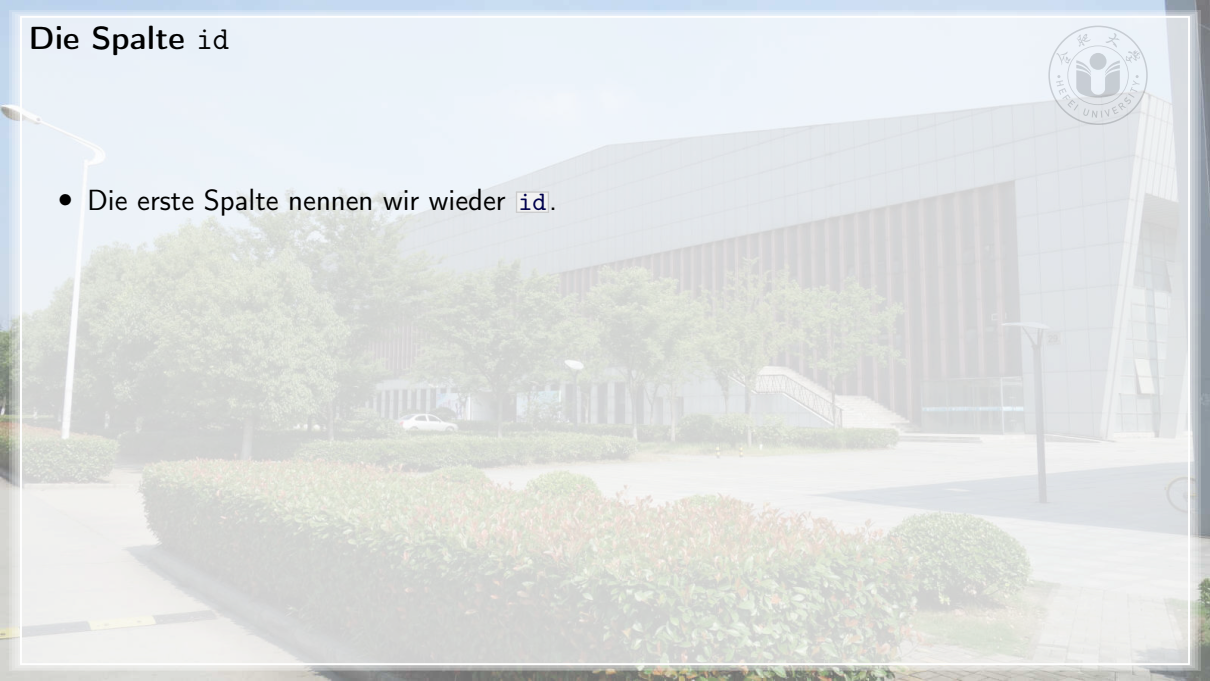
- Wir wissen ja schon, dass man Tabellen mit `CREATE TABLE` erstellt.
- Wir haben dieses Kommando via psql an den PostgreSQL-Server geschickt und das Erstellen der Tabelle hat funktioniert.
- Wieder gibt es einige interessante neue Komponenten: Was bedeutet `REFERENCES`? Was ist ein `DATE`? Schauen wir uns das Schritt für Schritt an.

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

# Die Spalte `id`



- Die erste Spalte nennen wir wieder `id`.

# Die Spalte `id`


- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.

# Die Spalte `id`


- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- In anderen Worten

# Die Spalte `id`


- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- In anderen Worten:
  - ihre Werte sind ganzzahlig (`INT`)

# Die Spalte `id`


- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- In anderen Worten:
  - ihre Werte sind ganzzahlig (`INT`),
  - werden automatisch von einer automatisch inkrementierten Sequenz generiert (`GENERATED BY DEFAULT AS IDENTITY`)

# Die Spalte `id`

- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- In anderen Worten:
  - ihre Werte sind ganzzahlig (`INT`),
  - werden automatisch von einer automatisch inkrementierten Sequenz generiert (`GENERATED BY DEFAULT AS IDENTITY`), und
  - stellen den Primärschlüssel für unsere Tabelle dar.

# Die Spalte `id`

- Die erste Spalte nennen wir wieder `id`.
- Sie hat den Typ `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- In anderen Worten:
  - ihre Werte sind ganzzahlig (`INT`),
  - werden automatisch von einer automatisch inkrementierten Sequenz generiert (`GENERATED BY DEFAULT AS IDENTITY`), und
  - stellen den Primärschlüssel für unsere Tabelle dar.
- Genaugenommen ist die Spalte wieder ein sogenannter *Surrogate Key*, zu Deutsch ein Ersatzschlüssel.

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.
- Die Daten der Kunden sind in der Tabelle `customer` gespeichert.

## Die Spalte `customer`


- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.
- Die Daten der Kunden sind in der Tabelle `customer` gespeichert.
- Sie werden über den Primärschlüssel identifiziert, den wir `id` genannt haben.

# Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.
- Die Daten der Kunden sind in der Tabelle `customer` gespeichert.
- Sie werden über den Primärschlüssel identifiziert, den wir `id` genannt haben.
- Die Werte für `id` sind einzigartig in der Tabelle `customer`: Jeder Record = jede Zeile der Tabelle hat einen anderen `id`-Wert.

# Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.
- Die Daten der Kunden sind in der Tabelle `customer` gespeichert.
- Sie werden über den Primärschlüssel identifiziert, den wir `id` genannt haben.
- Die Werte für `id` sind einzigartig in der Tabelle `customer`: Jeder Record = jede Zeile der Tabelle hat einen anderen `id`-Wert.
- Wir könnten also für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.

# Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- In der zweiten Spalte, die wir `customer` nennen, wollen wir also die Kunden-Einträge referenzieren.
- Die Daten der Kunden sind in der Tabelle `customer` gespeichert.
- Sie werden über den Primärschlüssel identifiziert, den wir `id` genannt haben.
- Die Werte für `id` sind einzigartig in der Tabelle `customer`: Jeder Record = jede Zeile der Tabelle hat einen anderen `id`-Wert.
- Wir könnten also für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.

## Die Spalte `customer`


- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).

## Die Spalte `customer`


- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).
- Wie stellen wir sicher, dass die Zahlen, die wir in der Spalte `customer` unserer neuen Tabelle `demand` speichern, wirklich passende Kunden-`id`s sind?

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).
- Wie stellen wir sicher, dass die Zahlen, die wir in der Spalte `customer` unserer neuen Tabelle `demand` speichern, wirklich passende Kunden-`id`s sind?
- Irgendwie müssen wir dem DBMS sagen: *"Diese neue Spalte hier referenziert die Spalte `id` der Tabelle `customer`. Es dürfen nur Werte auftreten, die auch dort vorkommen!"*

## Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle speichern.
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).
- Wie stellen wir sicher, dass die Zahlen, die wir in der Spalte `customer` unserer neuen Tabelle `demand` speichern, wirklich passende Kunden-`id`s sind?
- Irgendwie müssen wir dem DBMS sagen: *"Diese neue Spalte hier referenziert die Spalte `id` der Tabelle `customer`. Es dürfen nur Werte auftreten, die auch dort vorkommen!"*
- Das geht, in dem wir `REFERENCES customer(id)` schreiben, wobei sich `customer` auf die Tabelle `customer` und `id` auf die Spalte `id` der Tabelle `customer` bezieht.

## Die Spalte `customer`


- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).
- Wie stellen wir sicher, dass die Zahlen, die wir in der Spalte `customer` unserer neuen Tabelle `demand` speichern, wirklich passende Kunden-`id`s sind?
- Irgendwie müssen wir dem DBMS sagen: *"Diese neue Spalte hier referenziert die Spalte `id` der Tabelle `customer`. Es dürfen nur Werte auftreten, die auch dort vorkommen!"*
- Das geht, in dem wir `REFERENCES customer(id)` schreiben, wobei sich `customer` auf die Tabelle `customer` und `id` auf die Spalte `id` der Tabelle `customer` bezieht.
- Durch `REFERENCES customer(id)` wird die neue Spalte zum sogenannten Fremdschlüssel (Foreign Key).

## Die Spalte `customer`


- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Kunden-Record in der Tabelle `customer` finden und z.B. den Namen und die Adresse auslesen.
- Wir würden also eine neue Spalte namens `customer` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `customer` ist ja auch ein `INT`).
- Wie stellen wir sicher, dass die Zahlen, die wir in der Spalte `customer` unserer neuen Tabelle `demand` speichern, wirklich passende Kunden-`id`s sind?
- Irgendwie müssen wir dem DBMS sagen: *"Diese neue Spalte hier referenziert die Spalte `id` der Tabelle `customer`. Es dürfen nur Werte auftreten, die auch dort vorkommen!"*
- Das geht, in dem wir `REFERENCES customer(id)` schreiben, wobei sich `customer` auf die Tabelle `customer` und `id` auf die Spalte `id` der Tabelle `customer` bezieht.
- Durch `REFERENCES customer(id)` wird die neue Spalte zum sogenannten Fremdschlüssel (Foreign Key).
- Zusätzlich muss die Spalte als `NOT NULL` markiert werden.

# Die Spalte `customer`

- Zu jeder Bestellung gehört ein Kunde.
- Wir könnten für jede Bestellung den `id`-Wert des Kunden-Eintrags aus der `customer`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Produkt-Record in der Tabelle `product` finden und z.B. den Preis auslesen.

# Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Produkt-Record in der Tabelle `product` finden und z.B. den Preis auslesen.
- Wir würden also eine neue Spalte namens `product` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `product` ist ja auch ein `INT`).

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Produkt-Record in der Tabelle `product` finden und z.B. den Preis auslesen.
- Wir würden also eine neue Spalte namens `product` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `product` ist ja auch ein `INT`).
- Dazu müssen wir `REFERENCES product(id)` schreiben, wobei sich `product` auf die Tabelle `product` und `id` auf die Spalte `id` der Tabelle `product` bezieht.

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Produkt-Record in der Tabelle `product` finden und z.B. den Preis auslesen.
- Wir würden also eine neue Spalte namens `product` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `product` ist ja auch ein `INT`).
- Dazu müssen wir `REFERENCES product(id)` schreiben, wobei sich `product` auf die Tabelle `product` und `id` auf die Spalte `id` der Tabelle `product` bezieht.
- Dadurch wird sichergestellt, dass es nur Bestellungen geben kann, deren `product`-Wert zu einer Zeile in der Tabelle `product` passt.

## Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].
- Mit diesem Wert können wir dann den Produkt-Record in der Tabelle `product` finden und z.B. den Preis auslesen.
- Wir würden also eine neue Spalte namens `product` in der Tabelle `demand` erstellen, die vom Datentype `INT` ist (denn die Spalte `id` in er Tabelle `product` ist ja auch ein `INT`).
- Dazu müssen wir `REFERENCES product(id)` schreiben, wobei sich `product` auf die Tabelle `product` und `id` auf die Spalte `id` der Tabelle `product` bezieht.
- Dadurch wird sichergestellt, dass es nur Bestellungen geben kann, deren `product`-Wert zu einer Zeile in der Tabelle `product` passt.
- Zusätzlich muss die Spalte als `NOT NULL` markiert werden.

# Die Spalte `product`

- Zu jeder Bestellung gehört ein Produkt.
- Wir könnten für jede Bestellung den `id`-Wert des Produkt-Eintrags aus der `product`-Tabelle als Fremdschlüssel (Foreign Key) speichern[3].

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

## Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.

## Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.

## Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.

# Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.
- Stellt das aber sicher, dass die Menge “richtig” ist?

# Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.
- Stellt das aber sicher, dass die Menge “richtig” ist?
- Nein. Wir könnten 0 als Menge eingeben. Oder -5.

# Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.
- Stellt das aber sicher, dass die Menge "richtig" ist?
- Nein. Wir könnten 0 als Menge eingeben. Oder -5.
- Wir fügen also ein `CONSTRAINT` hinzu, das verlangt `(amount > 0)AND (amount < 1000000)`.

# Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.
- Stellt das aber sicher, dass die Menge “richtig” ist?
- Nein. Wir könnten 0 als Menge eingeben. Oder -5.
- Wir fügen also ein `CONSTRAINT` hinzu, das verlangt `(amount > 0)AND (amount < 1000000)`.
- Randnotiz: Solche Einschränkungen wären auch für die Tabelle `product` sinnvoll gewesen, allerdings war diese Lehreinheit schon kompliziert genug...

# Die Spalte `amount`

- Nun wollen wir noch speichern, wie viele Einheiten der Kunde von dem Produkt bestellt hat.
- Wir erstellen also eine Spalte namens `amount` vom Datentyp `INT`.
- Die Menge muss immer angegeben werden, also definieren wir sie als `NOT NULL`.

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?
- Zum Glück gibt es in SQL Datentypen für Datums und Zeitangaben.

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?
- Zum Glück gibt es in SQL Datentypen für Datums und Zeitangaben.
- Hier verwenden wir den Datentyp `DATE`.

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?
- Zum Glück gibt es in SQL Datentypen für Datums und Zeitangaben.
- Hier verwenden wir den Datentyp `DATE`, der ein Datum nach unserem (dem Gregorianischen) Kalender darstellt[4,5].

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?
- Zum Glück gibt es in SQL Datentypen für Datums und Zeitangaben.
- Hier verwenden wir den Datentyp `DATE`, der ein Datum nach unserem (dem Gregorianischen) Kalender darstellt[4,5].
- Also Schreibweise wird das ISO-Standardformat "`YYYY-MM-DD`"[2] verwendet, wobei `YYYY` das Jahr in vier Ziffern, `MM` den Monat in zwei Zifern, und `DD` den Tag in zwei Ziffern darstellen.

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered` (da `WHEN` und `DATE` reservierte SQL-Schlüsselworte sind[7]).
- Aber welchen Datentyp sollten wir benutzen?
- Zum Glück gibt es in SQL Datentypen für Datums und Zeitangaben.
- Hier verwenden wir den Datentyp `DATE`, der ein Datum nach unserem (dem Gregorianischen) Kalender darstellt[4,5].
- Also Schreibweise wird das ISO-Standardformat "`YYYY-MM-DD`"[2] verwendet, wobei `YYYY` das Jahr in vier Ziffern, `MM` den Monat in zwei Zifern, und `DD` den Tag in zwei Ziffern darstellen.
- Als Sicherheitscheck definieren wir ein `CONSTRAINT`, dass sicherstellt das nur Bestellungen nach dem 01.10.2024 und vor dem 01.01.3000 eingegeben werden können.

# Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered`.
- Hier verwenden wir den Datentyp `DATE`.

```
/* We create the new table 'demand' in our factory database. */

-- The table 'demand' stores all the customer orders.
CREATE TABLE demand (
    id       INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    customer INT  NOT NULL REFERENCES customer(id),
    product  INT  NOT NULL REFERENCES product(id),
    amount   INT  NOT NULL,
    ordered  DATE NOT NULL,
    CONSTRAINT ordered_amount_ok CHECK (
        (amount > 0) AND (amount < 1000000)),
    CONSTRAINT ordered_date_ok CHECK (
        (ordered > '2024-10-01') AND (ordered < '3000-01-01'))
);

-- List all tables of the user 'boss' in database 'factory'
-- Now we see the table 'demand'.
SELECT tablename FROM pg_catalog.pg_tables
    WHERE tableowner='boss';
```

## Die Spalte `ordered`

- Wir müssen auch das Bestelldatum speichern.
- Wir erstellen also eine Spalte namens `ordered`.
- Hier verwenden wir den Datentyp `DATE`.
- Damit ist unsere Tabelle `demand` fertig.

# Einfügen von Daten

## Daten Einfügen

- Nun haben wir die Tabelle `demand` erstellt, aber sie ist leer.

## Daten Einfügen

- Nun haben wir die Tabelle `demand` erstellt, aber sie ist leer.
- Wir wollen nun folgende acht Bestellungen speichern:

# Daten Einfügen

- Nun haben wir die Tabelle `demand` erstellt, aber sie ist leer.
- Wir wollen nun folgende acht Bestellungen speichern:

| id | customer | product | amount | ordered |
|---|---|---|---|---|
| 1 | 1 (Bibbo) | 7 (Shoe, Size 42) | 12 | 2024-11-21 |
| 2 | 2 (Bebbo) | 3 (Shoe, Size 38) | 2 | 2024-12-09 |
| 3 | 3 (Bebba) | 2 (Shoe, Size 37) | 7 | 2024-12-16 |
| 4 | 2 (Bebbo) | 5 (Shoe, Size 40) | 7 | 2024-12-30 |
| 5 | 1 (Bibbo) | 5 (Shoe, Size 40) | 3 | 2025-01-05 |
| 6 | 2 (Bebbo) | 6 (Shoe, Size 41) | 4 | 2025-01-12 |
| 7 | 3 (Bebba) | 11 (Large Purse) | 10 | 2025-01-16 |
| 8 | 2 (Bebbo) | 3 (Shoe, Size 38) | 6 | 2025-02-05 |
| … | … | … | … | … |

# Füllen der Tabelle demand

```sql
/* Store some data into the table 'demand'. */

-- Print all the contents from table 'demand': Nothing.
SELECT * from demand;

-- Insert 8 orders into our table.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1,  7, 12, '2024-11-21'), (2, 3, 2, '2024-12-09'),
       (3,  2,  7, '2024-12-16'), (2, 5, 7, '2024-12-30'),
       (1,  5,  3, '2025-01-05'), (2, 6, 4, '2025-01-12'),
       (3, 11, 10, '2025-01-16'), (2, 3, 6, '2025-02-05');

-- Now there are 8 rows.
SELECT * from demand;
```

# Füllen der 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.
```

## Schutz vor Fehlern

- Schützt das DBMS uns auch wirklich vor Fehlern?

## Schutz vor Fehlern

- Schützt das DBMS uns auch wirklich vor Fehlern?
- Was passiert, wenn wir eine falsche Produkt-ID verwenden?

```sql
/* Store some incorrect data into the table 'demand'. */

-- Trying to insert an order with an invalid product ID.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1, 77, 12, '2024-11-21');
```

# Schutz vor Fehlern


- Schützt das DBMS uns auch wirklich vor Fehlern?
- Wir können keine falsche Produkt-ID vwerden.

```
/* Store some incorrect data into the table 'demand'. */

-- Trying to insert an order with an invalid product ID.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1, 77, 12, '2024-11-21');
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf insert_into_table_demand_error_1.sql
psql:factory/insert_into_table_demand_error_1.sql:5: ERROR:  insert or
    ↪ update on table "demand" violates foreign key constraint "
    ↪ demand_product_fkey"
DETAIL:  Key (product)=(77) is not present in table "product".
psql:factory/insert_into_table_demand_error_1.sql:5: STATEMENT:  /*
    ↪ Store some incorrect data into the table 'demand'. */
-- Trying to insert an order with an invalid product ID.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1, 77, 12, '2024-11-21');
# psql 16.9 failed with exit code 3.
```

## Schutz vor Fehlern


- Schützt das DBMS uns auch wirklich vor Fehlern?
- Wir können keine falsche Produkt-ID vwerden.
- Was passiert, wenn wir ein falsches Datum eingeben?

```sql
/* Store some incorrect data into the table 'demand'. */

-- Trying to insert an order with an invalid date.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1,  7, 12, '1024-11-21');
```

# Schutz vor Fehlern

- Schützt das DBMS uns auch wirklich vor Fehlern?
- Wir können keine falsche Produkt-ID vwerden.
- Wir können kein ungültiges Datum eingeben.

```
/* Store some incorrect data into the table 'demand'. */

-- Trying to insert an order with an invalid date.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1,  7, 12, '1024-11-21');
```

```
$ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
    ↪ =1 -ebf insert_into_table_demand_error_2.sql
psql:factory/insert_into_table_demand_error_2.sql:5: ERROR:  new row for
    ↪  relation "demand" violates check constraint "ordered_date_ok"
DETAIL:  Failing row contains (10, 1, 7, 12, 1024-11-21).
psql:factory/insert_into_table_demand_error_2.sql:5: STATEMENT:  /*
    ↪ Store some incorrect data into the table 'demand'. */
-- Trying to insert an order with an invalid date.
INSERT INTO demand (customer, product, amount, ordered)
VALUES (1,  7, 12, '1024-11-21');
# psql 16.9 failed with exit code 3.
```

## Understanding the Data

- Wenn wir uns die Daten in unserer skizzierten Tabelle anschauen, dann können wir sie ganz gut verstehen, weil wir die Bedeutungen der IDs mit rangeschrieben haben.

| id | customer | product | amount | ordered |
|---|---|---|---|---|
| 1 | 1 (Bibbo) | 7 (Shoe, Size 42) | 12 | 2024-11-21 |
| 2 | 2 (Bebbo) | 3 (Shoe, Size 38) | 2 | 2024-12-09 |
| 3 | 3 (Bebba) | 2 (Shoe, Size 37) | 7 | 2024-12-16 |
| 4 | 2 (Bebbo) | 5 (Shoe, Size 40) | 7 | 2024-12-30 |
| 5 | 1 (Bibbo) | 5 (Shoe, Size 40) | 3 | 2025-01-05 |
| 6 | 2 (Bebbo) | 6 (Shoe, Size 41) | 4 | 2025-01-12 |
| 7 | 3 (Bebba) | 11 (Large Purse) | 10 | 2025-01-16 |
| 8 | 2 (Bebbo) | 3 (Shoe, Size 38) | 6 | 2025-02-05 |
| ... | ... | ... | ... | ... |

# Understanding the Data

- Wenn wir uns die Daten in unserer skizzierten Tabelle anschauen, dann können wir sie ganz gut verstehen, weil wir die Bedeutungen der IDs mit rangeschrieben haben.
- Ein `SELECT` gibt uns aber nur die IDs für Produkte und Kunden ... und ist daher schwer zu verstehen.

```
$ 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.
```

## Understanding the Data

- Wenn wir uns die Daten in unserer skizzierten Tabelle anschauen, dann können wir sie ganz gut verstehen, weil wir die Bedeutungen der IDs mit rangeschrieben haben.
- Ein `SELECT` gibt uns aber nur die IDs für Produkte und Kunden ... und ist daher schwer zu verstehen.
- In der nächsten Einheit lernen wir, wie wir die Daten aus den verschiedenen Tabellen zusammenführen und auswerten können.

# Zusammenfassung

# Zusammenfassung

- Nun haben wir die dritte und letzte Tabelle unserer Fabrikdatenbank erstellt.

# Zusammenfassung

- Nun haben wir die dritte und letzte Tabelle unserer Fabrikdatenbank erstellt.
- Anders als die beiden Tabellen davor existiert sie nicht “alleine für sich.”

# Zusammenfassung

- Nun haben wir die dritte und letzte Tabelle unserer Fabrikdatenbank erstellt.
- Anders als die beiden Tabellen davor existiert sie nicht “alleine für sich.”
- Stattdessen referenziert sie die Tabellen `customer` und `product` über Fremdschlüssel, also in dem sie Spalten hat, die Werte von deren Primärschlüssel speichern.

# Zusammenfassung


- Nun haben wir die dritte und letzte Tabelle unserer Fabrikdatenbank erstellt.
- Anders als die beiden Tabellen davor existiert sie nicht “alleine für sich.”
- Stattdessen referenziert sie die Tabellen `customer` und `product` über Fremdschlüssel, also in dem sie Spalten hat, die Werte von deren Primärschlüssel speichern.
- Somit können wir speichern, welcher Kunde welches Produkt bestellt hat.

# Zusammenfassung


- Nun haben wir die dritte und letzte Tabelle unserer Fabrikdatenbank erstellt.
- Anders als die beiden Tabellen davor existiert sie nicht "alleine für sich."
- Stattdessen referenziert sie die Tabellen `customer` und `product` über Fremdschlüssel, also in dem sie Spalten hat, die Werte von deren Primärschlüssel speichern.
- Somit können wir speichern, welcher Kunde welches Produkt bestellt hat.
- Als nächstes werden wir lernen, wie wir die Daten in unserer Datenbank auswerten können.

谢谢您们！

Thank you!

Vielen Dank!

# References I


[1] Ben Brumm. "SQL Best Practices and Style Guide". In: *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., 10. Dez. 2024. URL: https://www.databasestar.com/sql-best-practices (besucht am 2025-02-26) (siehe S. 13).

[2] *Date and Time – Representations for Information Interchange – Part 1: Basic Rules*. International Standard ISO 8601-1:2019(E), Edition 1. Geneva, Switzerland: International Organization for Standardization (ISO), Feb. 2019 (siehe S. 67–74).

[3] "Foreign Keys". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.5.5. URL: https://www.postgresql.org/docs/17/ddl-constraints.html#DDL-CONSTRAINTS-FK (besucht am 2025-02-28) (siehe S. 48–58).

[4] "History of Units". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. B.6. URL: https://www.postgresql.org/docs/17/datetime-units-history.html (besucht am 2025-03-01) (siehe S. 72–74).

[5] Pope Gregory XIII. *Inter Gravissimas*. Proclamation / Papal Bull. Vatican: Catholic Church, 24. Feb. 1582 (siehe S. 72–74).

[6] *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).

[7] "SQL Key Words". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Appendix C. URL: https://www.postgresql.org/docs/17/sql-keywords-appendix.html (besucht am 2025-07-04) (siehe S. 67–74).