



合肥大學
HEFEI UNIVERSITY



Datenbanken

10. Fabrik-Datenbank: Tabelle *customer*

Thomas Weise (汤卫思)
tweise@hfu.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

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-Code 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

1. Einleitung
2. Erstellen der Tabelle
3. Einfügen von Daten
4. Daten Selektieren
5. Zusammenfassung





Einleitung



Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.

Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.
- Wir wollen aber nicht nur Produktinformationen speichern, sondern auch Kundeninformationen.

Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.
- Wir wollen aber nicht nur Produktinformationen speichern, sondern auch Kundeninformationen.
- In unserem Beispiel werden wir also auch mit Kundendaten spielen.

Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.
- Wir wollen aber nicht nur Produktinformationen speichern, sondern auch Kundeninformationen.
- In unserem Beispiel werden wir also auch mit Kundendaten spielen.
- Diese sind hauptsächlich Textdaten.

Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.
- Wir wollen aber nicht nur Produktinformationen speichern, sondern auch Kundeninformationen.
- In unserem Beispiel werden wir also auch mit Kundendaten spielen.
- Diese sind hauptsächlich Textdaten.
- Das gibt uns die Gelegenheit, mit einigen fortgeschrittenen Text-Features eines DBMS zu spielen.

Einleitung



- Unsere Datenbank hat nun genau eine Tabelle, nämlich `product`.
- Wir wollen aber nicht nur Produktinformationen speichern, sondern auch Kundeninformationen.
- In unserem Beispiel werden wir also auch mit Kundendaten spielen.
- Diese sind hauptsächlich Textdaten.
- Das gibt uns die Gelegenheit, mit einigen fortgeschrittenen Text-Features eines DBMS zu spielen.
- Gleichzeitig werden wir Möglichkeiten kennenlernen, um die Wertebereiche für Spalten weiter einzuschränken.



Erstellen der Tabelle



Grobstruktur

- Wir nennen unsere Tabelle für Kunden `customer`.



Grobstruktur

- Wir nennen unsere Tabelle für Kunden `customer`.
- Wir wollen folgende Informationen zu jedem Kunden darin speichern



Grobstruktur

- Wir nennen unsere Tabelle für Kunden `customer`.
- Wir wollen folgende Informationen zu jedem Kunden darin speichern
 - den Namen



Grobstruktur



- Wir nennen unsere Tabelle für Kunden `customer`.
- Wir wollen folgende Informationen zu jedem Kunden darin speichern
 - den Namen,
 - die Telefonnummer



Grobstruktur



- Wir nennen unsere Tabelle für Kunden `customer`.
- Wir wollen folgende Informationen zu jedem Kunden darin speichern
 - den Namen,
 - die Telefonnummer, und
 - die Adresse.



Grobstruktur



- Wir nennen unsere Tabelle für Kunden `customer`.
- Wir wollen folgende Informationen zu jedem Kunden darin speichern
 - den Namen,
 - die Telefonnummer, und
 - die Adresse.
- Insgesamt könnten die Daten also wie folgt aussehen:

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

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.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Erstellen der Tabelle mit CREATE TABLE



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

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf create_table_customer.sql
2 CREATE TABLE
3  tablename
4  -----
5  product
6  customer
7  (2 rows)
8
9 # psql 16.11 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.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     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.
- Wir erkennen einige Komponenten des Kommandos wieder.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     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.
- Wir erkennen einige Komponenten des Kommandos wieder.
- Andere, insbesondere der Teil mit `CONSTRAINT`, sind neu.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     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.
- Wir erkennen einige Komponenten des Kommandos wieder.
- Andere, insbesondere der Teil mit `CONSTRAINT`, sind neu.
- Arbeiten wir uns durch das Kommando durch!

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     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 (EN: *primary key*) 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 (EN: *primary key*) für unsere Tabelle dar.
- Genaugenommen ist die Spalte wieder ein sogenannter Ersatzschlüssel (EN: *surrogate key*).

Die Spalte name

- Kunden haben Namen.



Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen. Die Spalte wird daher mit `NOT NULL` annotiert.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen. Die Spalte wird daher mit `NOT NULL` annotiert.
- Müssen Namen einmalig sein?

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen. Die Spalte wird daher mit `NOT NULL` annotiert.
- Müssen Namen einmalig sein? Nein.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen. Die Spalte wird daher mit `NOT NULL` annotiert.
- Müssen Namen einmalig sein? Nein. Es ist völlig OK wenn zwei Kunden den selben Namen haben.

Die Spalte name



- Kunden haben Namen.
- Als nächstes definieren wir also eine Spalte `name`.
- Namen sind Text und Namen sind verschieden lang. Nehmen wir also wieder `VARCHAR(100)` als Datentyp – 100 Zeichen sollten reichen für vernünftige Namen.
- Natürlich braucht jeder Kunde einen Namen. Die Spalte wird daher mit `NOT NULL` annotiert.
- Müssen Namen einmalig sein? Nein. Es ist völlig OK wenn zwei Kunden den selben Namen haben. Wir müssen die Spalte also *nicht* mit `UNIQUE` annotieren.

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).



Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können.

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen **NOT NULL** nicht...

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (`NOT NULL`).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen `NOT NULL` nicht...
- Was bedeutet eigentlich *korrekt*?

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (`NOT NULL`).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen `NOT NULL` nicht...
- Was bedeutet eigentlich *korrekt*?
- Unser DBMS kann nicht feststellen, ob der Kunde Herr Bebbo tatsächlich Bebbo heißt.

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen **NOT NULL** nicht...
- Was bedeutet eigentlich *korrekt*?
- Unser DBMS kann nicht feststellen, ob der Kunde Herr Bebbo tatsächlich Bebbo heißt.
- Was wir aber verhindern können, ist z. B. das aus Versehen „Bebbo“ statt „Bebbo“ eingegeben wird.

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (**NOT NULL**).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen **NOT NULL** nicht...
- Was bedeutet eigentlich *korrekt*?
- Unser DBMS kann nicht feststellen, ob der Kunde Herr Bebbo tatsächlich Bebbo heißt.
- Was wir aber verhindern können, ist z. B. das aus Versehen „Bebbo“ statt „Bebbo“ eingegeben wird.
- Und wir können leere Zeichenfolgen verbieten.

Die Spalte name: CONSTRAINT



- Namen sind also Zeichenketten mit maximal 100 Zeichen und müssen angegeben werden (`NOT NULL`).
- Ist das ausreichend, um sicherzustellen, dass die gespeicherten Kundennamen korrekt sind?
- Wenn wir zu unserer Tabelle `product` zurückgehen, stellen wir fest, dass wir problemlos `' bla blab '`, `' '`, oder sogar die leere Zeichenkette `' '` als Produktname eingeben können. Nur weglassen können wir den Namen wegen `NOT NULL` nicht...
- Was bedeutet eigentlich *korrekt*?
- Unser DBMS kann nicht feststellen, ob der Kunde Herr Bebbo tatsächlich Bebbo heißt.
- Was wir aber verhindern können, ist z. B. das aus Versehen „Bebbo“ statt „Bebbo“ eingegeben wird.
- Und wir können leere Zeichenfolgen verbieten.
- Mit Hilfe von Einschränkungen, die als `CONSTRAINT` formuliert werden¹⁰.

CONSTRAINT

- Beim Erstellen einer Tabelle können wir zusätzliche Einschränkungen als **CONSTRAINT** angeben.



CONSTRAINT



- Beim Erstellen einer Tabelle können wir zusätzliche Einschränkungen als **CONSTRAINT** angeben.

```
1  -- Erstellen einer neuen Tabelle mit zusätzlichen Einschränkungen.
2  --
3  -- table_name: der Name für die zu erstellende Tabelle
4  -- column_X: der Name der X-ten Spalte
5  -- type_of_column_X: der Datentyp für die X-te Spalte
6  -- constraints_on_column_X: Einschränkungen für die X-te Spalte, kann weg-
7  -- gelassen werden, oder eine Kombination von
8  -- UNIQUE, NOT NULL, PRIMARY KEY, etc.
9  -- constraint_name_Y: der Name der Y-ten Einschränkung
10 -- expression_Y: der Prüfausdruck der Y-ten Einschränkung
11 --
12 -- Jedes mal, wenn eine Zeile in die Tabelle angefügt oder geändert
13 -- wird, werden die Prüfausdrücke aller Einschränkungen ausgewertet.
14 -- Wenn auch nur einer von ihnen nicht Wahr=True ergibt, wird die
15 -- Operation abgebrochen, nicht ausgeführt, und ein Fehler berichtet.
16 CREATE TABLE table_name (
17     column1 type_of_column1 constraints_on_column1,
18     column2 type_of_column2 constraints_on_column2,
19     ...
20     CONSTRAINT constraint_name1 CHECK expression1,
21     CONSTRAINT constraint_name2 CHECK expression2,
22     ...
23 );
```

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte name so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.
- Man kann natürlich immer noch „sgjw9345 s熊猫fki345Q“ als Name eingeben.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.
- Man kann natürlich immer noch „sgjw9345 s熊猫fki345Q“ als Name eingeben ... aber Leerzeichen oder Zahlen am Anfang und Ende sind unmöglich und leere Zeichenketten auch.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.
- Man kann natürlich immer noch „sgjw9345 s熊猫fki345Q“ als Name eingeben ... aber Leerzeichen oder Zahlen am Anfang und Ende sind unmöglich und leere Zeichenketten auch.
- Bisher haben wir keine Möglichkeit, so eine Einschränkung zu formulieren.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.
- Man kann natürlich immer noch „sgjw9345 s熊猫fki345Q“ als Name eingeben ... aber Leerzeichen oder Zahlen am Anfang und Ende sind unmöglich und leere Zeichenketten auch.
- Bisher haben wir keine Möglichkeit, so eine Einschränkung zu formulieren.
- Mit `LIKE` (wie in unsere SQL-Anfrage nach Schuhen. . .) geht es leider nicht.

Die Spalte name: CONSTRAINT



- Wir wollen die Werte in der Spalte `name` so einschränken, dass sie mit einem „Wort-Zeichen“, z. B. „A“, „b“, oder „张“, beginnen und enden müssen. Dazwischen erlauben wir beliebige Zeichen.
- Man kann natürlich immer noch „sgjw9345 s熊猫fki345Q“ als Name eingeben ... aber Leerzeichen oder Zahlen am Anfang und Ende sind unmöglich und leere Zeichenketten auch.
- Bisher haben wir keine Möglichkeit, so eine Einschränkung zu formulieren.
- Mit `LIKE` (wie in unsere SQL-Anfrage nach Schuhen. . .) geht es leider nicht.
- Aber mit einem **Regulären Ausdruck**.

Reguläre Ausdrücke



- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.



Reguläre Ausdrücke



- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.
- Wenn ein regulärer Ausdruck zu einer Zeichenkette passt, dann spricht man von einem *match*.





- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.
- Wenn ein regulärer Ausdruck zu einer Zeichenkette passt, dann spricht man von einem *match*.
- Regexes werden von sehr vielen Werkzeugen und Programmiersprachen (wie z. B. Python⁵¹) unterstützt – und auch von SQL und, daher, auch von PostgreSQL³⁵.

- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.
- Wenn ein regulärer Ausdruck zu einer Zeichenkette passt, dann spricht man von einem *match*.
- Regexes werden von sehr vielen Werkzeugen und Programmiersprachen (wie z. B. Python⁵¹) unterstützt – und auch von SQL und, daher, auch von PostgreSQL³⁵.
- Regexes sind ein komplett anderes Fachgebiet.



- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.
- Wenn ein regulärer Ausdruck zu einer Zeichenkette passt, dann spricht man von einem *match*.
- Regexes werden von sehr vielen Werkzeugen und Programmiersprachen (wie z. B. Python⁵¹) unterstützt – und auch von SQL und, daher, auch von PostgreSQL³⁵.
- Regexes sind ein komplett anderes Fachgebiet.
- Sie können darüber in “POSIX Regular Expressions”³⁵ nachlesen.



- Reguläre Ausdrücke – regular expressions (regexes) – sind Vorlagen für Zeichenketten, die mit anderen Zeichenketten verglichen werden können.
- Wenn ein regulärer Ausdruck zu einer Zeichenkette passt, dann spricht man von einem *match*.
- Regexes werden von sehr vielen Werkzeugen und Programmiersprachen (wie z. B. Python⁵¹) unterstützt – und auch von SQL und, daher, auch von PostgreSQL³⁵.
- Regexes sind ein komplett anderes Fachgebiet.
- Sie können darüber in “POSIX Regular Expressions”³⁵ nachlesen.
- Hier wollen wir sie nur kurz vorstellen und direkt einsetzen.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\w` steht für ein „Wort-Zeichen“.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\\w.*\\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\\w` steht für ein „Wort-Zeichen“.
- `^\\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\w` steht für ein „Wort-Zeichen“.
- `^\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- Der Punkt `.` steht für ein beliebiges Zeichen.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\\w.*\\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\\w` steht für ein „Wort-Zeichen“.
- `^\\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- Der Punkt `.` steht für ein beliebiges Zeichen.
- Der Stern `*` bedeutet, dass der Ausdruck davor beliebig oft (0, 1, 2, ...) vorkommen darf.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\\w.*\\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\\w` steht für ein „Wort-Zeichen“.
- `^\\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- Der Punkt `.` steht für ein beliebiges Zeichen.
- Der Stern `*` bedeutet, dass der Ausdruck davor beliebig oft (0, 1, 2, ...) vorkommen darf.
- `.*` nach dem `^\\w` bedeutet also, dass nach dem anfänglichen Wort-Zeichen beliebig viele andere Zeichen beliebiger Art im Namen vorkommen dürfen.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\\w.*\\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\\w` steht für ein „Wort-Zeichen“.
- `^\\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- Der Punkt `.` steht für ein beliebiges Zeichen.
- Der Stern `*` bedeutet, dass der Ausdruck davor beliebig oft (0, 1, 2, ...) vorkommen darf.
- `.*` nach dem `^\\w` bedeutet also, dass nach dem anfänglichen Wort-Zeichen beliebig viele andere Zeichen beliebiger Art im Namen vorkommen dürfen.
- Das Dollarzeichen `$` passt zum Ende einer Zeichenkette.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\\w.*\\w$'` passen müssen.
- Das Zeichen `^` stellt den Beginn des Textes dar.
- `\\w` steht für ein „Wort-Zeichen“.
- `^\\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- Der Punkt `.` steht für ein beliebiges Zeichen.
- Der Stern `*` bedeutet, dass der Ausdruck davor beliebig oft (0, 1, 2, ...) vorkommen darf.
- `.*` nach dem `^\\w` bedeutet also, dass nach dem anfänglichen Wort-Zeichen beliebig viele andere Zeichen beliebiger Art im Namen vorkommen dürfen.
- Das Dollarzeichen `$` passt zum Ende einer Zeichenkette.
- Wenn wir also schreiben `\\w$`, dann bedeutet das, dass am Ende der Zeichenkette wieder ein Wort-Zeichen kommen muss.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.
- `^\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- `.*` nach dem `^\w` bedeutet also, dass nach dem anfänglichen Wort-Zeichen beliebig viele andere Zeichen beliebiger Art im Namen vorkommen dürfen.
- Wenn wir also schreiben `\w$`, dann bedeutet das, dass am Ende der Zeichenkette wieder ein Wort-Zeichen kommen muss.
- `'^\w.*\w$'` passt also zu „Thomas Weise“ oder „李白“, **nicht** aber zu „Bibbo“, „熊貓3“, oder „李“.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir spezifizieren, dass die Werte in der Spalte `name` immer zum regex `'^\w.*\w$'` passen müssen.
- `^\w` bedeutet also, dass ein Wort-Zeichen am Anfang der Zeichenkette stehen muss.
- `.*` nach dem `^\w` bedeutet also, dass nach dem anfänglichen Wort-Zeichen beliebig viele andere Zeichen beliebiger Art im Namen vorkommen dürfen.
- Wenn wir also schreiben `\w$`, dann bedeutet das, dass am Ende der Zeichenkette wieder ein Wort-Zeichen kommen muss.
- `'^\w.*\w$'` passt also zu „Thomas Weise“ oder „李白“, **nicht** aber zu „Bibbo“, „熊貓3“, oder „李“.
- Wir schreiben `CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$')`, wobei `name ~ '^\w.*\w$'` bedeutet, dass der Wert für `name` dem gegebenen regex entsprechen muss.

Die Spalte name: CONSTRAINT und Reguläre Ausdrücke



- Wir schreiben `CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$')`, wobei `~ '^\w.*\w$'` bedeutet, dass der Wert für `name` nur aus Buchstaben, Zahlen, Unterstrichen und Bindestrichen bestehen darf.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Die Spalte phone



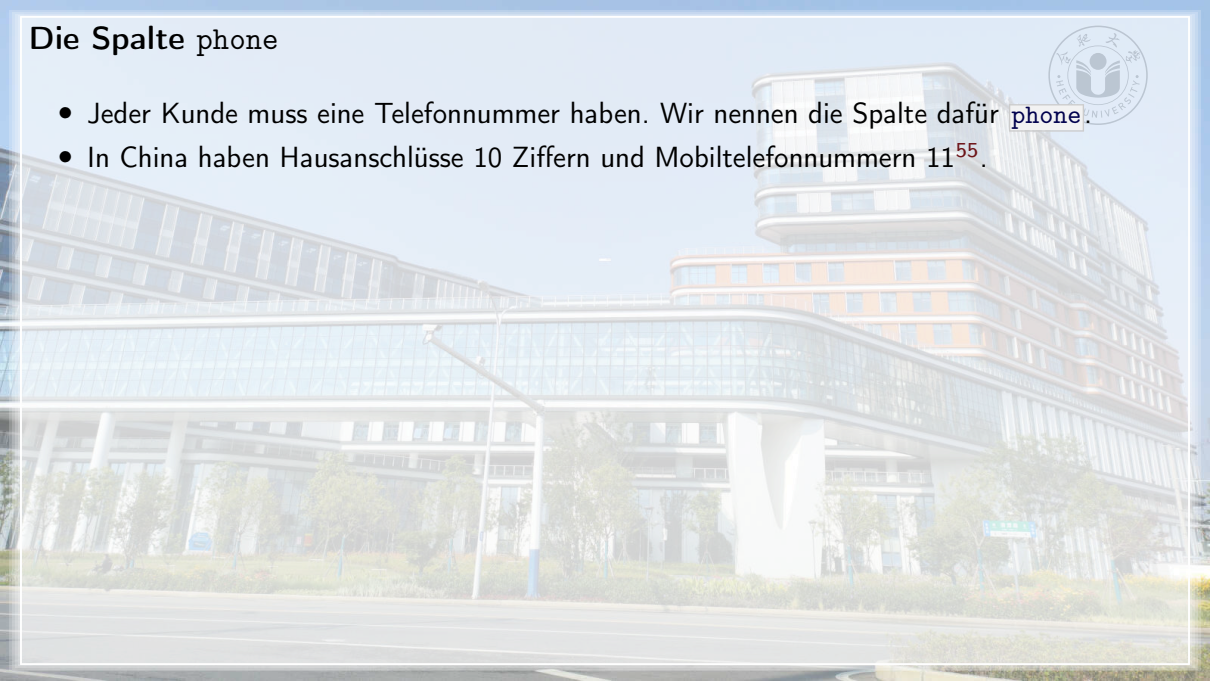
- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür **phone**.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,          -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,   -- the phone number
9      address     VARCHAR(255) NOT NULL,          -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Die Spalte phone



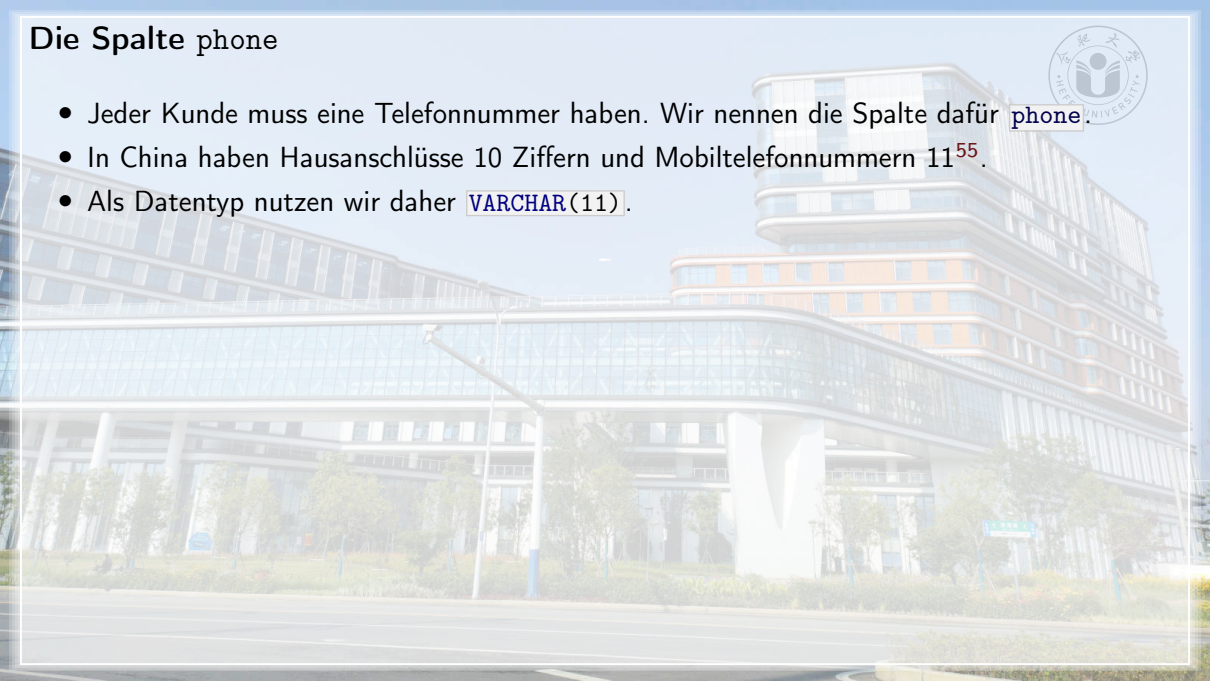
- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür phone.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.



Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür `phone`.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher `VARCHAR(11)`.



Die Spalte phone



- Jeder Kunde **muss** eine Telefonnummer haben. Wir nennen die Spalte dafür **phone**.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher **VARCHAR(11)**.
- Da jeder Kunde eine eine Telefonnummer haben **muss**, ist die Spalte auch **NOT NULL**.

Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür **phone**.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher **VARCHAR(11)**.
- Da jeder Kunde eine eine Telefonnummer haben muss, ist die Spalte auch **NOT NULL**.
- Jede Telefonnummer kann auch nur von einem Kunden verwendet werden, also spezifizieren die **UNIQUE**-Einschränkung.

Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür `phone`.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher `VARCHAR(11)`.
- Da jeder Kunde eine eine Telefonnummer haben muss, ist die Spalte auch `NOT NULL`.
- Jede Telefonnummer kann auch nur von einem Kunden verwendet werden, also spezifizieren die `UNIQUE`-Einschränkung.
- `VARCHAR(11)` lässt beliebige Zeichenketten mit maximal 11 Zeichen zu.

Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür `phone`.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher `VARCHAR(11)`.
- Da jeder Kunde eine eine Telefonnummer haben muss, ist die Spalte auch `NOT NULL`.
- Jede Telefonnummer kann auch nur von einem Kunden verwendet werden, also spezifizieren die `UNIQUE`-Einschränkung.
- `VARCHAR(11)` lässt beliebige Zeichenketten mit maximal 11 Zeichen zu.
- Wir wollen aber nur Telefonnummern zulassen.

Die Spalte phone



- Jeder Kunde muss eine Telefonnummer haben. Wir nennen die Spalte dafür `phone`.
- In China haben Hausanschlüsse 10 Ziffern und Mobiltelefonnummern 11⁵⁵.
- Als Datentyp nutzen wir daher `VARCHAR(11)`.
- Da jeder Kunde eine eine Telefonnummer haben muss, ist die Spalte auch `NOT NULL`.
- Jede Telefonnummer kann auch nur von einem Kunden verwendet werden, also spezifizieren die `UNIQUE`-Einschränkung.
- `VARCHAR(11)` lässt beliebige Zeichenketten mit maximal 11 Zeichen zu.
- Wir wollen aber nur Telefonnummern zulassen.
- Regexes to the rescue!

Die Spalte phone: CONSTRAINT

- Wir wollen nur Telefonnummern zulassen.



Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.
- `\\d` steht für ein einzelnes Zahlenzeichen, also 0...9.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.
- `\\d` steht für ein einzelnes Zahlenzeichen, also 0...9.
- `{10,11}` bedeutet, dass der Ausdruck davor mindestens 10 Mal und höchstens 11 Mal hintereinander vorkommen darf.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.
- `\\d` steht für ein einzelnes Zahlenzeichen, also 0...9.
- `{10,11}` bedeutet, dass der Ausdruck davor mindestens 10 Mal und höchstens 11 Mal hintereinander vorkommen darf.
- `^\\d{10,11}$` bedeutet also, dass direkt am Anfang der Zeichenkette entweder 10 oder 11 Ziffern kommen und danach endet die Zeichenkette.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.
- `\\d` steht für ein einzelnes Zahlenzeichen, also 0...9.
- `{10,11}` bedeutet, dass der Ausdruck davor mindestens 10 Mal und höchstens 11 Mal hintereinander vorkommen darf.
- `^\\d{10,11}$` bedeutet also, dass direkt am Anfang der Zeichenkette entweder 10 oder 11 Ziffern kommen und danach endet die Zeichenkette.
- Natürlich können wir nicht verhindern, dass ungültige Telefonnummern eingegeben werden.

Die Spalte phone: CONSTRAINT



- Wir wollen nur Telefonnummern zulassen.
- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\\d{10,11}$')`.
- Die Werte in der Spalte `phone` müssen immer zum regulären Ausdruck `^\\d{10,11}$` passen.
- `\\d` steht für ein einzelnes Zahlenzeichen, also 0...9.
- `{10,11}` bedeutet, dass der Ausdruck davor mindestens 10 Mal und höchstens 11 Mal hintereinander vorkommen darf.
- `^\\d{10,11}$` bedeutet also, dass direkt am Anfang der Zeichenkette entweder 10 oder 11 Ziffern kommen und danach endet die Zeichenkette.
- Natürlich können wir nicht verhindern, dass ungültige Telefonnummern eingegeben werden.
- Aber wir erzwingen zumindest, dass die eingegebenen Zeichenketten zum generellen Schema chinesischer Telefonnummern passen.

Die Spalte phone: CONSTRAINT



- Wir schreiben daher `CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')`

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Die Spalte address

- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.



Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.
- Wir nehmen wieder `VARCHAR(255)`.

Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.
- Wir nehmen wieder `VARCHAR(255)`.
- Wir markieren die Spalte als `NOT NULL`.

Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.
- Wir nehmen wieder `VARCHAR(255)`.
- Wir markieren die Spalte als `NOT NULL`.
- Wir markieren sie *nicht* als `UNIQUE`.

Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.
- Wir nehmen wieder `VARCHAR(255)`.
- Wir markieren die Spalte als `NOT NULL`.
- Wir markieren sie *nicht* als `UNIQUE`.
- Sie können sich selbst ein `CONSTRAINT` als Übung ausdenken.

Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.

```
1  /* We create the new table 'customer' in our factory database. */
2
3  -- The table 'customer' stores all the customers that we have.
4  -- Each row of this table identifies one such a customer.
5  CREATE TABLE customer (
6      id          INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
7      name        VARCHAR(100) NOT NULL,           -- must exist
8      phone       VARCHAR(11)  NOT NULL UNIQUE,    -- the phone number
9      address     VARCHAR(255) NOT NULL,           -- the address
10     CONSTRAINT customer_name_ok CHECK (name ~ '^\w.*\w$'),
11     CONSTRAINT customer_phone_ok CHECK (phone ~ '^\d{10,11}$')
12 );
13
14 -- List all tables of the user 'boss' in database 'customer'
15 -- Now we see the table 'customer'.
16 SELECT tablename FROM pg_catalog.pg_tables
17     WHERE tableowner='boss';
```

Die Spalte address



- Fügen wir nun die Spalte `address` für die Kundenadresse hinzu.
- Damit sind wir mit dem Erstellen der Tabelle `customer` fertig.



Einfügen von Daten



Daten Einfügen



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

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbba und Herr Bobbo.

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbba und Herr Bobbo:
 - Herr Bibbo lebt im Südcampus unserer Universität Hefei (合肥大学)

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbä und Herr Bobbo:
 - Herr Bibbo lebt im Südcampus unserer Universität Hefei (合肥大学),
 - Herr Bebbo hat seine Geschäftsadresse im Rathaus der Stadt Chemnitz in Deutschland

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbba und Herr Bobbo:
 - Herr Bibbo lebt im Südcampus unserer Universität Hefei (合肥大学),
 - Herr Bebbo hat seine Geschäftsadresse im Rathaus der Stadt Chemnitz in Deutschland,
 - Frau Bebbba wohnt direkt am Times Square in New York, USA

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbba und Herr Bobbo:
 - Herr Bibbo lebt im Südcampus unserer Universität Hefei (合肥大学),
 - Herr Bebbo hat seine Geschäftsadresse im Rathaus der Stadt Chemnitz in Deutschland,
 - Frau Bebbba wohnt direkt am Times Square in New York, USA, und
 - Herr Bobbo arbeitet am Eiffelturm in Paris, Frankreich.

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Daten Einfügen



- Nun haben wir die Tabelle `customer` erstellt, aber sie ist leer.
- Füllen wir sie mit Daten!
- Wir haben vier Kunden: Herr Bibbo, Herr Bebbo, Frau Bebbba und Herr Bobbo:
 - Herr Bibbo lebt im Südcampus unserer Universität Hefei (合肥大学),
 - Herr Bebbo hat seine Geschäftsadresse im Rathaus der Stadt Chemnitz in Deutschland,
 - Frau Bebbba wohnt direkt am Times Square in New York, USA, und
 - Herr Bobbo arbeitet am Eiffelturm in Paris, Frankreich.
- Ihre Telefonnummern sind 11 Wiederholungen einer einzigen Ziffer.

id	name	phone	address
1	Bibbo	999999999999	Hefei, Jinxiu Dadao 99, China
2	Bebbo	555555555555	Rathaus, Chemnitz, Germany
3	Bebba	333333333333	Times Square, NY, USA
4	Bobbo	444444444444	Eiffel Tower, Paris, France
...	...	...	...

Füllen der Tabelle product



```
1  /* Store some data into the table 'customer'. */
2
3  -- Print all the contents from table 'customer': Nothing.
4  SELECT * FROM customer;
5
6  -- Insert 4 customers into our table.
7  INSERT INTO customer (name, phone, address)
8  VALUES ('Bibbo', '99999999999', 'Hefei, Jinxiu Dadao 99, China'),
9          ('Bebbo', '55555555555', 'Rathaus, Chemnitz, Germany'),
10         ('Bebba', '33333333333', 'Times Square, NY, USA'),
11         ('Bobbo', '44444444444', 'Eiffel Tower, Paris, France');
12
13  -- Now there are 4 rows.
14  SELECT * FROM customer;
```

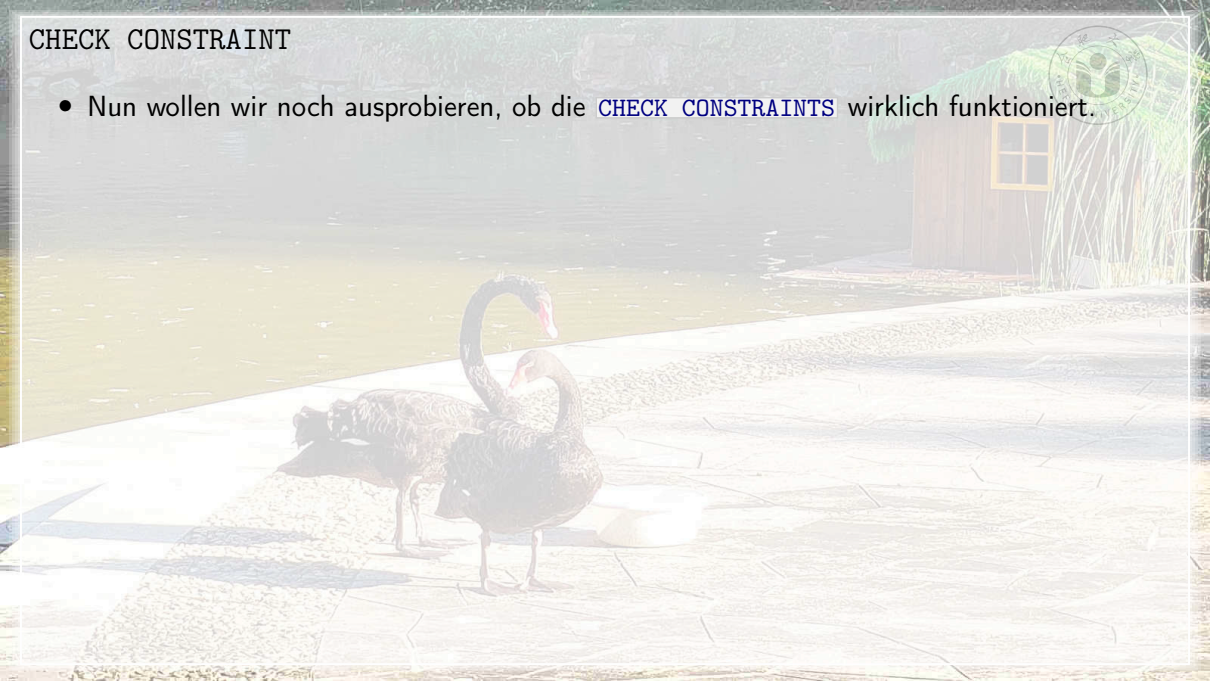
Füllen der Tabelle product



```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf insert_into_table_customer.sql
2 id | name | phone | address
3 -----
4 (0 rows)
5
6 INSERT 0 4
7 id | name | phone | address
8 -----
9 1 | Bibbo | 999999999999 | Hefei, Jinxiu Dadao 99, China
10 2 | Bebbo | 555555555555 | Rathaus, Chemnitz, Germany
11 3 | Bebba | 333333333333 | Times Square, NY, USA
12 4 | Bobbo | 444444444444 | Eiffel Tower, Paris, France
13 (4 rows)
14
15 # psql 16.11 succeeded with exit code 0.
```

CHECK CONSTRAINT

- Nun wollen wir noch ausprobieren, ob die **CHECK CONSTRAINTS** wirklich funktioniert.



CHECK CONSTRAINT

- Nun wollen wir noch ausprobieren, ob die **CHECK CONSTRAINTS** wirklich funktioniert.
- Was denken Sie, kann die SQL-Anweisung hier funktionieren? Warum/nicht?

```
1  /* Store some faulty data into the table 'customer'. */
2
3  -- Try to insert a faulty customer record. Can you spot the error?
4  INSERT INTO customer (name, phone, address)
5  VALUES ('Eugen', '88888B88888', 'Hochschule Osnabrück, Germany');
```

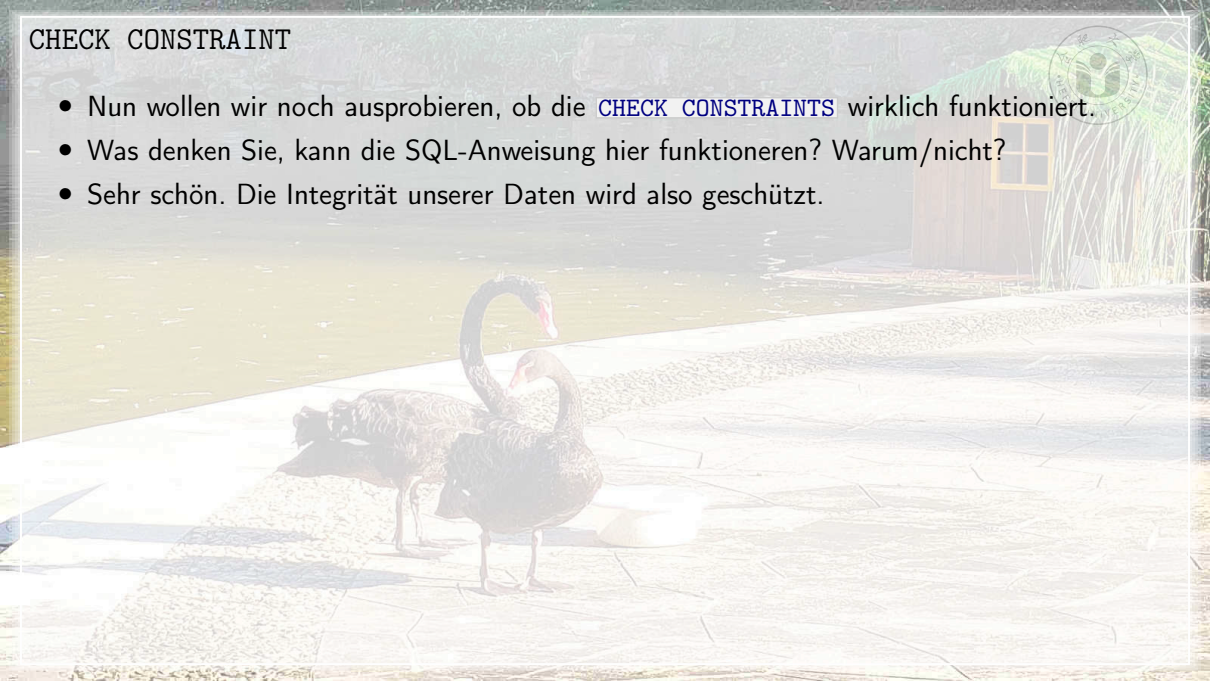
CHECK CONSTRAINT

- Nun wollen wir noch ausprobieren, ob die **CHECK CONSTRAINTS** wirklich funktioniert.
- Was denken Sie, kann die SQL-Anweisung hier funktionieren? Warum/nicht?

```
1 $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↳ =1 -ebf insert_into_table_customer_error.sql
2 psql:factory/insert_into_table_customer_error.sql:5: ERROR:  new row for
   ↳ relation "customer" violates check constraint "customer_phone_ok"
3 DETAIL:  Failing row contains (5, Eugen, 88888B88888, Hochschule Osnabrü
   ↳ ck, Germany).
4 psql:factory/insert_into_table_customer_error.sql:5: STATEMENT:  /*
   ↳ Store some faulty data into the table 'customer'. */
5 -- Try to insert a faulty customer record. Can you spot the error?
6 INSERT INTO customer (name, phone, address)
7 VALUES ('Eugen', '88888B88888', 'Hochschule Osnabrück, Germany');
8 # psql 16.11 failed with exit code 3.
```

CHECK CONSTRAINT

- Nun wollen wir noch ausprobieren, ob die **CHECK CONSTRAINTS** wirklich funktioniert.
- Was denken Sie, kann die SQL-Anweisung hier funktionieren? Warum/nicht?
- Sehr schön. Die Integrität unserer Daten wird also geschützt.





Daten Selektieren



Daten Selektieren: LIKE

- Nun haben wir Daten in die Tabelle `customer` eingefügt.



Daten Selektieren: LIKE



- Nun haben wir Daten in die Tabelle `customer` eingefügt.
- Holen wir sie wieder heraus!

Daten Selektieren: LIKE



- Nun haben wir Daten in die Tabelle `customer` eingefügt.
- Holen wir sie wieder heraus!
- Zuersteinmal wählen wir alle Kunden aus, die in Frankreich wohnen.

Daten Selektieren: LIKE



- Zuersteinmal wählen wir alle Kunden aus, die in Frankreich wohnen.
- `...WHERE address LIKE '%France%'` limitiert die Ausgabe von `SELECT` auf Zeilen, in denen das Wort „France“ – genauso geschrieben – irgendwo im Wert von `address` vorkommt³⁴.

```
1  /* Extract information from the table customer. */
2
3  -- Try to find customers who may be living in France.
4  SELECT name, address FROM customer WHERE address LIKE '%France%';
```

```
1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_from_table_customer_1.sql
2  name      | address
3  -----+-----
4  Bobbo     | Eiffel Tower, Paris, France
5  (1 row)
6
7  # psql 16.11 succeeded with exit code 0.
```

Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.



Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Zuerst erstellen wir ein synthetisches Attribute `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).



Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Zuerst erstellen wir ein synthetisches Attribut `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).
- Das können wir mit `address LIKE '%China%' AS domestic` machen³⁴.

Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Zuerst erstellen wir ein synthetisches Attribute `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).
- Das können wir mit `address LIKE '%China%' AS domestic` machen³⁴.
- Wir wollen aber diesmal lieber `ILIKE` nehmen: `LIKE` beachtet Groß- und Kleinschreibung, `ILIKE` nicht.

Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Zuerst erstellen wir ein synthetisches Attribut `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).
- Das können wir mit `address LIKE '%China%' AS domestic` machen³⁴.
- Wir wollen aber diesmal lieber `ILIKE` nehmen: `LIKE` beachtet Groß- und Kleinschreibung, `ILIKE` nicht.
- Für `LIKE` sind `'china'` und `'China'` verschieden.

Daten Selektieren: ILIKE



- Als nächstes wollen wir wissen, wie viele inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Zuerst erstellen wir ein synthetisches Attribute `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).
- Das können wir mit `address LIKE '%China%' AS domestic` machen³⁴.
- Wir wollen aber diesmal lieber `ILIKE` nehmen: `LIKE` beachtet Groß- und Kleinschreibung, `ILIKE` nicht.
- Für `LIKE` sind `'china'` und `'China'` verschieden.
- Für `ILIKE` sind `'china'`, `'China'` und `'CHINA'` das selbe.

Daten Selektieren: ILIKE



- Zuerst erstellen wir ein synthetisches Attribute `domestic` (DE: *inländisch*), dass wahr (`TRUE`) ist, wenn ein Kunde in China wohnt und sonst falsch (`FALSE`).
- `t` steht für `TRUE` und `f` für `FALSE`.

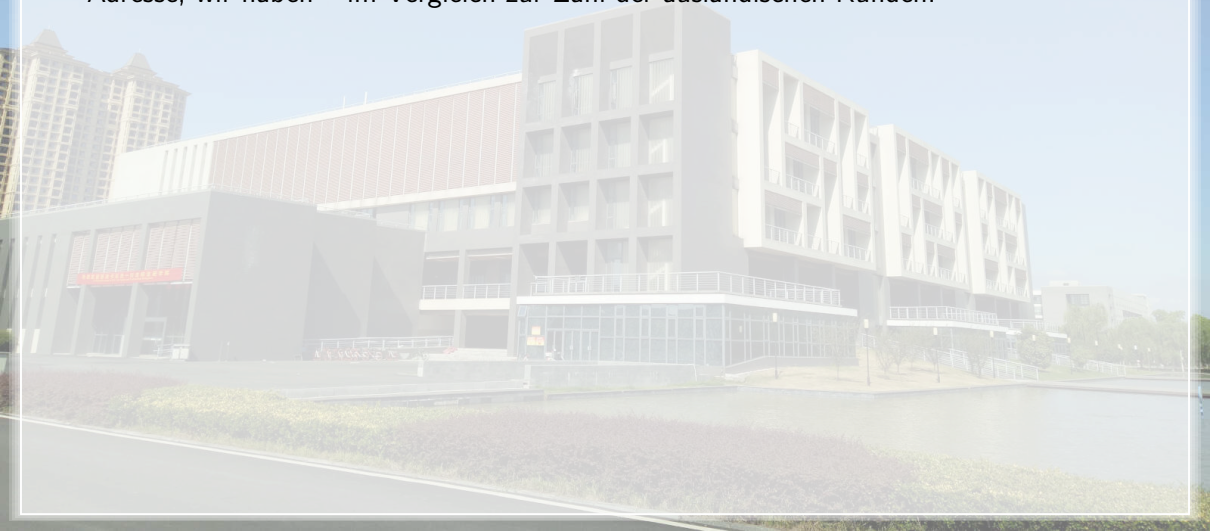
```
1  /* Extract information from the table customer. */
2
3  -- Which customers are domestic, i.e., live in China?
4  SELECT name, address ILIKE '%china%' as domestic FROM customer;
```

```
1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_from_table_customer_2.sql
2  name  | domestic
3  -----+-----
4  Bibbo | t
5  Bebbo | f
6  Bebbä | f
7  Bobbo | f
8  (4 rows)
9
10 # psql 16.11 succeeded with exit code 0.
```

Daten Selektieren: GROUP BY und COUNT



- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.



Daten Selektieren: GROUP BY und COUNT



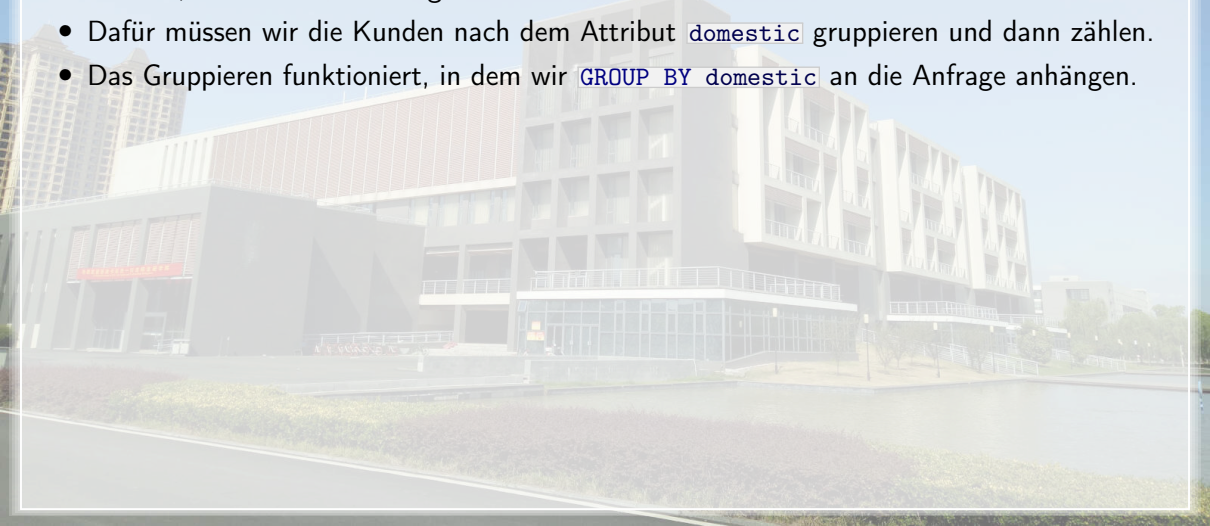
- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Dafür müssen wir die Kunden nach dem Attribut `domestic` gruppieren und dann zählen.



Daten Selektieren: GROUP BY und COUNT



- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Dafür müssen wir die Kunden nach dem Attribut `domestic` gruppieren und dann zählen.
- Das Gruppieren funktioniert, in dem wir `GROUP BY domestic` an die Anfrage anhängen.



Daten Selektieren: GROUP BY und COUNT



- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Dafür müssen wir die Kunden nach dem Attribut `domestic` gruppieren und dann zählen.
- Das Gruppieren funktioniert, in dem wir `GROUP BY domestic` an die Anfrage anhängen.
- Statt die Kundennamen zu drucken, zeigen wir dann `COUNT(*)` an.

Daten Selektieren: GROUP BY und COUNT



- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Dafür müssen wir die Kunden nach dem Attribut `domestic` gruppieren und dann zählen.
- Das Gruppieren funktioniert, in dem wir `GROUP BY domestic` an die Anfrage anhängen.
- Statt die Kundennamen zu drucken, zeigen wir dann `COUNT(*)` an.
- Wie `AVG` ist `COUNT` eine Aggregatfunktion, die eine Statistik über alle Elemente der Gruppe berechnet ... und diese Statistik ist hier einfach „Anzahl“.

Daten Selektieren: GROUP BY und COUNT



- Wir wollen aber wissen, **wie viele** inländische Kunden, also Kunden mit chinesischer Adresse, wir haben – im Vergleich zur Zahl der ausländischen Kunden.
- Wir haben also einen inländischen und drei ausländische Kunden.

```
1  /* Extract information from the table customer. */
2
3  -- Count how many domestic and foreign customers we have.
4  SELECT COUNT(*), address ILIKE '%china%' as domestic FROM customer
5         GROUP BY domestic;
```

```
1  $ psql "postgres://boss:superboss123@localhost/factory" -v ON_ERROR_STOP
   ↪ =1 -ebf select_from_table_customer_3.sql
2
3  count | domestic
4  -----+-----
5         3 | f
6         1 | t
7  (2 rows)
8
9  # psql 16.11 succeeded with exit code 0.
```

Adressen sind kompliziert

- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.



Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China. . .

Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China, würden 中国 aber nicht erkennen.
- Zum anderen würden wir eine *Donut Factory Vive la France* in Shanghai, China sowohl als Französisch als auch als inländisch erkennen.

Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China, würden 中国 aber nicht erkennen.
- Zum anderen würden wir eine *Donut Factory Vive la France* in Shanghai, China sowohl als Französisch als auch als inländisch erkennen.
- Um das letztere Problem zu lösen, könnten wir die Adresse in mehrere Spalten aufteilen, mit einer Spalte für das Land.

Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China, würden 中国 aber nicht erkennen.
- Zum anderen würden wir eine *Donut Factory Vive la France* in Shanghai, China sowohl als Französisch als auch als inländisch erkennen.
- Um das letztere Problem zu lösen, könnten wir die Adresse in mehrere Spalten aufteilen, mit einer Spalte für das Land.
- Selbst dann hätten wir noch Probleme mit verschiedenen Schreibweisen für Länder, z. B. China, china, 中国, People's Republic of China, PRC, P.R.C., 中华人民共和国, République populaire de Chine, Chine, usw.

Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China, würden 中国 aber nicht erkennen.
- Zum anderen würden wir eine *Donut Factory Vive la France* in Shanghai, China sowohl als Französisch als auch als inländisch erkennen.
- Um das letztere Problem zu lösen, könnten wir die Adresse in mehrere Spalten aufteilen, mit einer Spalte für das Land.
- Selbst dann hätten wir noch Probleme mit verschiedenen Schreibweisen für Länder, z. B. China, china, 中国, People's Republic of China, PRC, P.R.C., 中华人民共和国, République populaire de Chine, Chine, usw.
- Als endgültige Lösung würden wir wohl eine weitere Tabelle nur für Länder brauchen, die von unserer Kundentabelle referenziert wird.

Adressen sind kompliziert



- Wenn Sie gut aufgepasst haben, dann merken Sie, dass unsere Art der Adressbehandlung problematisch ist.
- Zum einen erkennen wir zwar 'China' und 'china' beides als China, würden 中国 aber nicht erkennen.
- Zum anderen würden wir eine *Donut Factory Vive la France* in Shanghai, China sowohl als Französisch als auch als inländisch erkennen.
- Um das letztere Problem zu lösen, könnten wir die Adresse in mehrere Spalten aufteilen, mit einer Spalte für das Land.
- Selbst dann hätten wir noch Probleme mit verschiedenen Schreibweisen für Länder, z. B. China, china, 中国, People's Republic of China, PRC, P.R.C., 中华人民共和国, République populaire de Chine, Chine, usw.
- Als endgültige Lösung würden wir wohl eine weitere Tabelle nur für Länder brauchen, die von unserer Kundentabelle referenziert wird.
- Das wollen wir jetzt nicht tun ... aber in der nächsten Einheit lernen Sie, wie das prinzipiell geht.



Zusammenfassung



Zusammenfassung



- Jetzt haben wir unsere zweite Tabelle erstellt, die Tabelle `customer`.

Zusammenfassung



- Jetzt haben wir unsere zweite Tabelle erstellt, die Tabelle `customer`.
- Wir haben einige neue Funktionen zum Arbeiten mit Texten gelernt, z. B. `regexes` und `ILIKE`.

Zusammenfassung



- Jetzt haben wir unsere zweite Tabelle erstellt, die Tabelle `customer`.
- Wir haben einige neue Funktionen zum Arbeiten mit Texten gelernt, z. B. `regexes` und `ILIKE`.
- Wir haben gelernt, wie wir mit `CONSTRAINT` zusätzliche, komplexere Einschränkungen beim Erstellen einer Tabelle definieren kann.

Zusammenfassung



- Jetzt haben wir unsere zweite Tabelle erstellt, die Tabelle `customer`.
- Wir haben einige neue Funktionen zum Arbeiten mit Texten gelernt, z. B. `regexes` und `ILIKE`.
- Wir haben gelernt, wie wir mit `CONSTRAINT` zusätzliche, komplexere Einschränkungen beim Erstellen einer Tabelle definieren kann.
- Wir haben gelernt, dass wir Daten in `SELECT`-Anfragen mit `GROUP BY` gruppieren können und Aggregatfunktionen wie `COUNT` über die Gruppen berechnen können.



- Jetzt haben wir unsere zweite Tabelle erstellt, die Tabelle `customer`.
- Wir haben einige neue Funktionen zum Arbeiten mit Texten gelernt, z. B. `regexes` und `ILIKE`.
- Wir haben gelernt, wie wir mit `CONSTRAINT` zusätzliche, komplexere Einschränkungen beim Erstellen einer Tabelle definieren kann.
- Wir haben gelernt, dass wir Daten in `SELECT`-Anfragen mit `GROUP BY` gruppieren können und Aggregatfunktionen wie `COUNT` über die Gruppen berechnen können.
- Und wir haben ein wenig ein Gefühl dafür entwickelt, dass Daten wie „Adressen“ auf den ersten Blick einfach aussehen, auf den zweiten Blick aber doch komplexer sein können als erwartet.



谢谢你们！
Thank you!
Vielen Dank!



References I



- [1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 156, 157).
- [2] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt> (besucht am 2025-02-05) (siehe S. 158).
- [3] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 156).
- [4] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 156).
- [5] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 156).
- [6] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 157).
- [7] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: <https://cacm.acm.org/research/50-years-of-queries> (besucht am 2025-01-09) (siehe S. 157).
- [8] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 157, 158).
- [9] Edgar Frank „Ted“ Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (besucht am 2025-01-05) (siehe S. 157).
- [10] "Constraints". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.5. URL: <https://www.postgresql.org/docs/17/ddl-constraints.html> (besucht am 2025-02-28) (siehe S. 42–50).

References II



- [11] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 157).
- [12] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (besucht am 2025-02-27) (siehe S. 157).
- [13] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 157).
- [14] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide Web: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 158).
- [15] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 156).
- [16] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 157).
- [17] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 157).
- [18] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 156).
- [19] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 158).
- [20] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 157).

References III



- [21] *IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX(TM))--Part 2: Shell and Utilities*. IEEE Std 1003.2-1992. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), 23. Juni 1993. URL: <https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/POSIX/all.pdf> (besucht am 2025-03-27). Board Approved: 1992-09-17, ANSI Approved: 1993-04-05. See unapproved draft IEEE P1003.2 Draft 11.2 of 9 1991 at the url (siehe S. 157).
- [22] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. 157).
- [23] Andrew M. Kuchling. *Python 3 Documentation. Regular Expression HOWTO*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/howto/regex.html> (besucht am 2024-11-01) (siehe S. 157).
- [24] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 157).
- [25] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 157).
- [26] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: <https://www.britannica.com/technology/client-server-architecture> (besucht am 2025-01-20) (siehe S. 156).
- [27] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 157).
- [28] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 157).
- [29] Zsolt Nagy. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. New York, NY, USA: Apress Media, LLC, Aug. 2018. ISBN: 978-1-4842-3876-9 (siehe S. 157).

References IV



- [30] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: **978-0-596-00965-6** (siehe S. **156**).
- [31] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., Juni 2019. ISBN: **978-1-4920-8255-2** (siehe S. **157**).
- [32] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: **978-1-4919-6336-4** (siehe S. **156**).
- [33] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: **978-0-471-31615-2** (siehe S. **156**).
- [34] "Pattern Matching". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.7. URL: <https://www.postgresql.org/docs/17/functions-matching.html> (besucht am 2025-02-27) (siehe S. **119–128**).
- [35] "POSIX Regular Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.7.3. URL: <https://www.postgresql.org/docs/17/functions-matching.html#FUNCTIONS-POSIX-REGEXP> (besucht am 2025-02-27) (siehe S. **60–65, 157**).
- [36] *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).
- [37] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. **156**).
- [38] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: **978-1-78883-546-6** (siehe S. **156**).
- [39] "`re` – Regular Expression Operations". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/re.html#module-re> (besucht am 2024-11-01) (siehe S. **157**).
- [40] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: **978-1-4920-4345-4** (siehe S. **156**).

References V



- [41] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: **978-0-596-15448-6** (siehe S. **156**).
- [42] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: **0001-0782**. doi:**10.1145/361020.361025** (siehe S. **157**).
- [43] "SQL Commands". 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-commands.html> (besucht am 2025-02-25) (siehe S. **157**).
- [44] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: **978-0-672-32451-2** (siehe S. **154**, **157**).
- [45] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: **978-3-8272-2020-2**. Translation of⁴⁴ (siehe S. **157**).
- [46] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: **978-1-4842-3841-7** (siehe S. **157**).
- [47] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: **978-1-80323-347-5** (siehe S. **156**).
- [48] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: **0001-0782**. doi:**10.1145/299157.299165** (siehe S. **156**).
- [49] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: **978-0-13-792931-3** (siehe S. **156**).

References VI



- [50] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2025. URL: <https://thomasweise.github.io/databases> (besucht am 2025-01-05) (siehe S. 156, 157).
- [51] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: <https://thomasweise.github.io/programmingWithPython> (besucht am 2025-01-05) (siehe S. 60–65, 157).
- [52] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (besucht am 2025-01-05) (siehe S. 157).
- [53] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: <https://www.techtarget.com/searchdatamanagement/definition/database-management-system> (besucht am 2025-01-11) (siehe S. 156).
- [54] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 156).
- [55] “手机号码：电话管理部门为手机设定的号码 (Mobile Phone Number: A Number Set by the Telephone Management Department for Mobile Phones)”. In: 百度百科 (*Baidu Baike*). China, Beijing (中国北京市): Baidu (百度公司), 6. Jan. 2025. URL: <https://baike.baidu.com/item/%E6%89%8B%E6%9C%BA%E5%8F%B7%E7%A0%81> (besucht am 2025-04-17) (siehe S. 78–86).

Glossary (in English) I



Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs^{5,30,54}. Learn more at <https://www.gnu.org/software/bash>.

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as psql.

client-server architecture is a system design where a central server receives requests from one or multiple clients^{3,26,33,38,40}. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*⁵⁰.

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB⁵³.

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows^{1,18,41,48,49}. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.

Microsoft Windows is a commercial proprietary operating system⁴. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.

PostgreSQL An open source object-relational DBMS^{15,32,37,47}. See <https://postgresql.org> for more information.

psql is the client program used to access the PostgreSQL DBMS server.

Glossary (in English) II



Python The Python programming language^{20,25,27,51}, i.e., what you will learn about in our book⁵¹. Learn more at <https://python.org>.

regex A *Regular Expression*, often called „regex“ for short, is a sequence of characters that defines a search pattern for text strings^{21,23,29,31}. In Python, the `re` module offers functionality work with regular expressions^{23,39}. In PostgreSQL, regex-based pattern matching is supported as well³⁵.

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other^{9,16,17,42,46,50,52}.

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers⁶ in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“²⁴.

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases^{7,11-13,22,28,43-46}. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference⁴³.

terminal A terminal is a text-based window where you can enter commands and execute them^{1,8}. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf `Win`+`R`, dann Schreiben von `cmd`, dann Druck auf `↵`. Under Ubuntu Linux, `Ctrl`+`Alt`+`T` opens a terminal, which then runs a Bash shell inside.

Glossary (in English) III



Ubuntu is a variant of the open source operating system Linux^{8,19}. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at <https://ubuntu.com>. If you are in China, you can download it from <https://mirrors.ustc.edu.cn/ubuntu-releases>.

WWW World Wide Web^{2,14}