合肥大學
HEFEI UNIVERSITY

Datenbanken

39. Logisches Schema: Beziehungsattribute

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: 2025-11-28

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

- Beziehungen in konzeptuellen Modellen können Attribute haben, wie wir in Einheit 29 gelernt haben.

Einleitung

- Beziehungen in konzeptuellen Modellen können Attribute haben, wie wir in Einheit 29 gelernt haben.
- Weil es Beziehungen im relationalen Datenmodell nicht als einzelne Objekte gibt, müssen wir diese Attribute also irgendwo anders hintun.

Einleitung

- Beziehungen in konzeptuellen Modellen können Attribute haben, wie wir in Einheit 29 gelernt haben.
- Weil es Beziehungen im relationalen Datenmodell nicht als einzelne Objekte gibt, müssen wir diese Attribute also irgendwo anders hintun.
- Im relationalen Modell gibt es nur Relationen, die als Tabellen implementiert werden.

Einleitung

- Beziehungen in konzeptuellen Modellen können Attribute haben, wie wir in Einheit 29 gelernt haben.
- Weil es Beziehungen im relationalen Datenmodell nicht als einzelne Objekte gibt, müssen wir diese Attribute also irgendwo anders hintun.
- Im relationalen Modell gibt es nur Relationen, die als Tabellen implementiert werden.
- Also müssen die Attribute von Beziehungen Tabellenspalten werden.

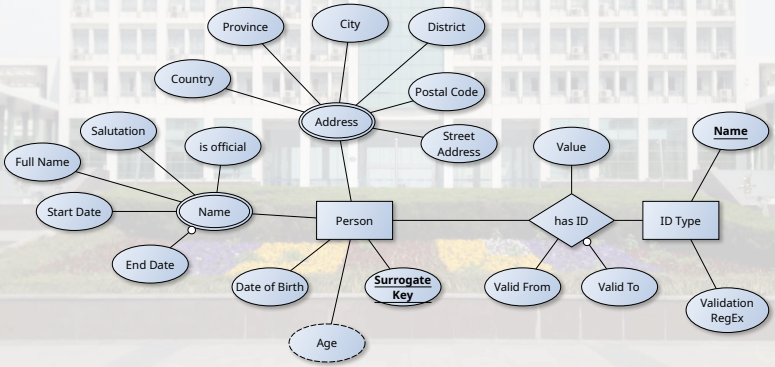
Beispiel

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.

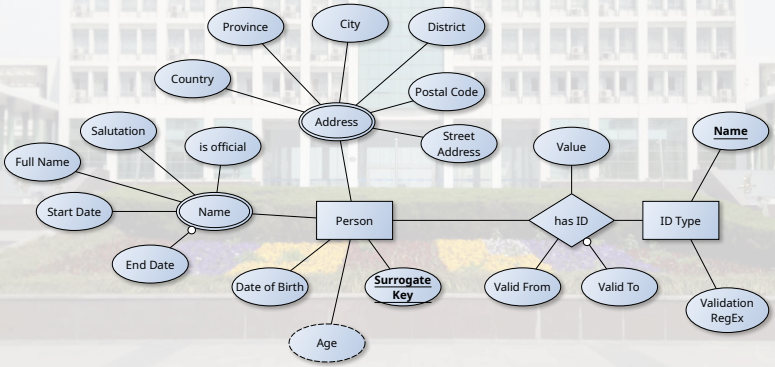
Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.



Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.



Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\rightarrow\!\!\circ\!\!-\!\!\prec$ *ID Type*- oder eine *Person* $\rightarrow\!\!|\!\!-\!\!\prec$ *ID Type*-Beziehung wäre.

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*- oder eine *Person* $\gg\!\!|\!\!-\!\!\circ\!\!\ll$ *ID Type*-Beziehung wäre.
- Wir können beliebig viele (Arten von) IDs für jede Person speichern.

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*- oder eine *Person* $\gg\!\!|\!\!-\!\!\circ\!\!\ll$ *ID Type*-Beziehung wäre.
- Wir können beliebig viele (Arten von) IDs für jede Person speichern.
- Jede Art von ID kann von beliebig vielen Personen verwendet werden.

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*- oder eine *Person* $\gg\!\!|\!\!-\!\!\circ\!\!\ll$ *ID Type*-Beziehung wäre.
- Wir können beliebig viele (Arten von) IDs für jede Person speichern.
- Jede Art von ID kann von beliebig vielen Personen verwendet werden.
- In den letzten Slides hatten wir die schwere Variante einer Beziehung modelliert...
  ...nahmen wir diesmal die leichte.

Beispiel: Konzeptuelle Ebene

- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*- oder eine *Person* $\gg\!\!|\!\!-\!\!\circ\!\!\ll$ *ID Type*-Beziehung wäre.
- Wir können beliebig viele (Arten von) IDs für jede Person speichern.
- Jede Art von ID kann von beliebig vielen Personen verwendet werden.
- In den letzten Slides hatten wir die schwere Variante einer Beziehung modelliert...
  ...nahmen wir diesmal die leichte.
- Wir wählen *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*.

Beispiel: Konzeptuelle Ebene

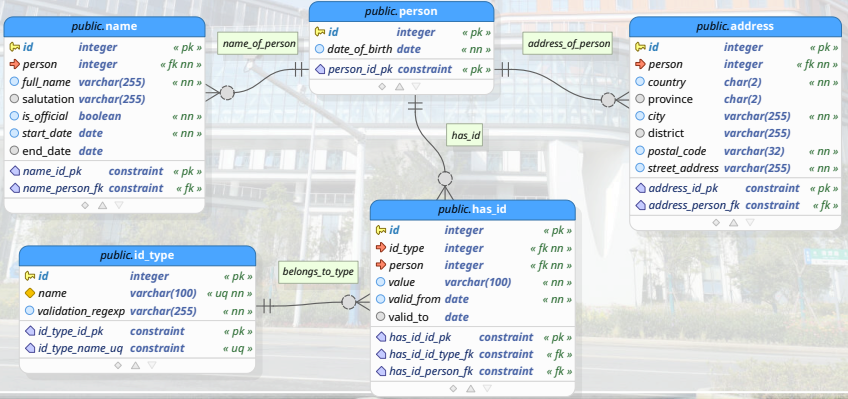
- In welche Tabelle die Beziehungsattribute kommen hängt von der Situation ab.
- Erinnern wir uns zurück an ein frühes Modell unserer *Person*-Entität.
- Hier gibt es eine Beziehung *has ID*, die die *Person*-Entitäten mit dem Entitätstyp *ID Type* verbindet.
- Wir hatten damals keine Kardinalitäten modelliert, weil wir damals noch nicht so weit waren.
- Es ist aber ziemlich klar, dass das entweder eine *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*- oder eine *Person* $\gg\!\!|\!\!-\!\!\circ\!\!\ll$ *ID Type*-Beziehung wäre.
- Wir können beliebig viele (Arten von) IDs für jede Person speichern.
- Jede Art von ID kann von beliebig vielen Personen verwendet werden.
- In den letzten Slides hatten wir die schwere Variante einer Beziehung modelliert...
  ...nahmen wir diesmal die leichte.
- Wir wählen *Person* $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ *ID Type*.
- Wir folgen also dem Schema O $\gg\!\!\circ\!\!-\!\!\circ\!\!\ll$ P.

Beispiel: Logische Ebene

- Im logischen Schema brauchen wir für diese Beziehungsform eine zusätzliche Tabelle.

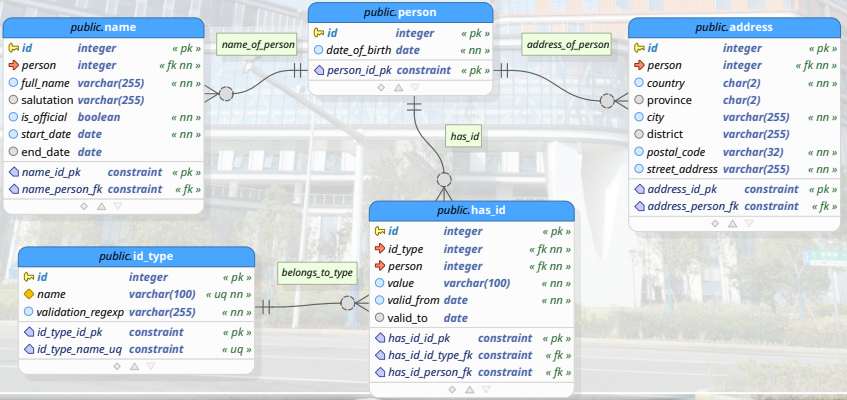
Beispiel: Logische Ebene

- Im logischen Schema brauchen wir für diese Beziehungsform eine zusätzliche Tabelle.
- Wir folgen der selben Methode wie damals bei O >o—o< P, benutzen aber natürlich andere Namen.



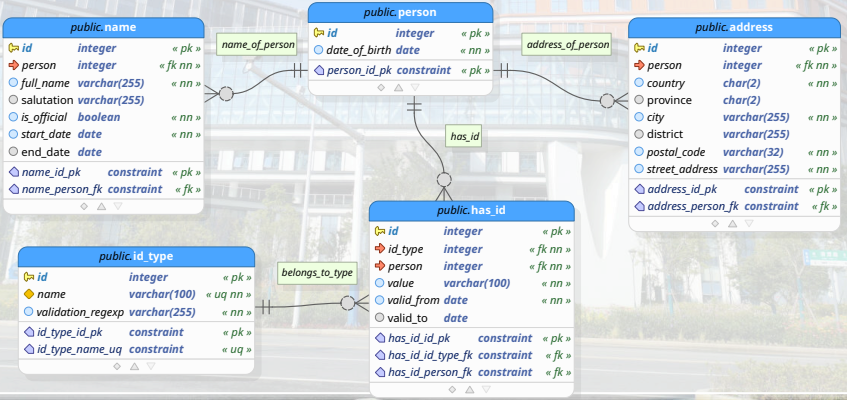
Beispiel: Logische Ebene

- Im logischen Schema brauchen wir für diese Beziehungsform eine zusätzliche Tabelle.
- Wir folgen der selben Methode wie damals bei O ⋊—⋉ P, benutzen aber natürlich andere Namen.
- Wir benutzen wieder den PgModeler.



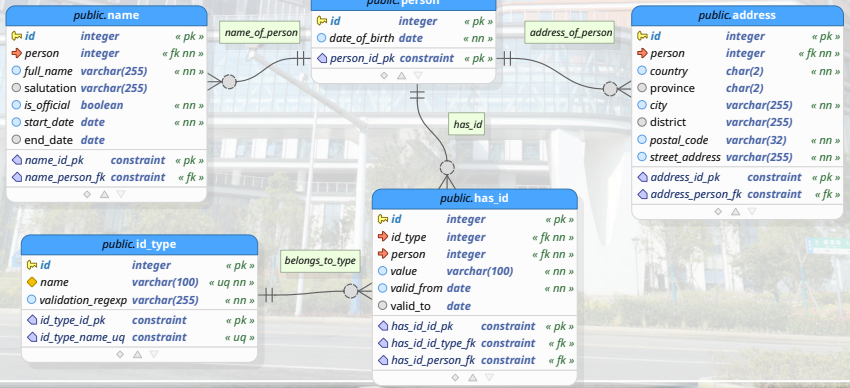
Beispiel: Logische Ebene

- Wir folgen der selben Methode wie damals bei O⋊—○⋉P, benutzen aber natürlich andere Namen.
- Wir benutzen wieder den PgModeler.
- Wir nennen die zusätzliche Tabelle `has_id`.



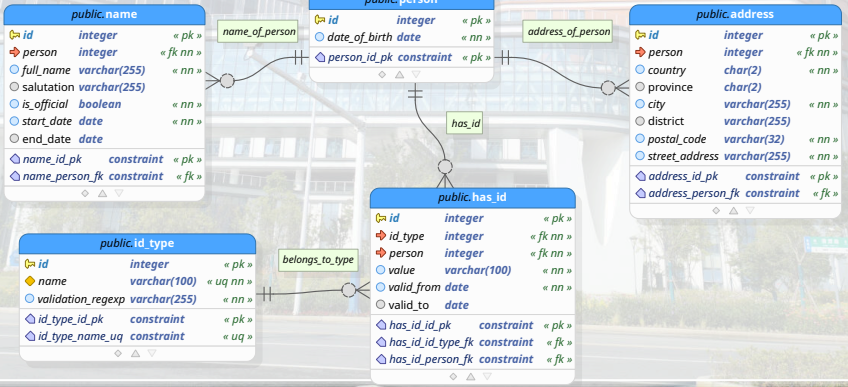
Beispiel: Logische Ebene

- Wir benutzen wieder den PgModeler.
- Wir nennen die zusätzliche Tabelle `has_id`.
- Die Beziehungsattribute werden in diese Tabelle gehen.



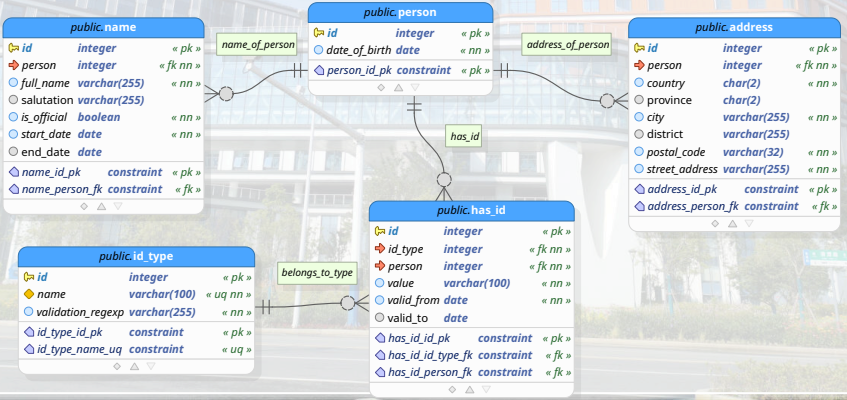
Beispiel: Logische Ebene

- Wir nennen die zusätzliche Tabelle `has_id`.
- Die Beziehungsattribute werden in diese Tabelle gehen.
- Und wieder bekommen wir eine neue Perspektive auf Beziehungen.



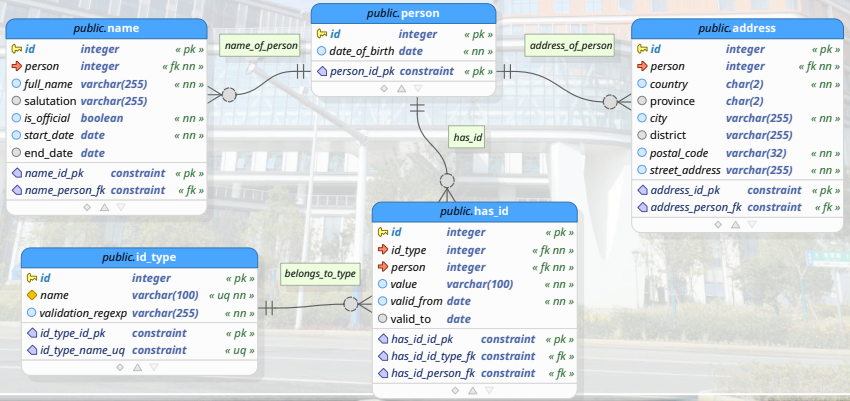
Beispiel: Logische Ebene

- Die Beziehungsattribute werden in diese Tabelle gehen.
- Und wieder bekommen wir eine neue Perspektive auf Beziehungen:
- Seinerzeit hatten wir O >o—o< P mit der zusätzlichen Tabelle `relate_o_and_p` implementiert.



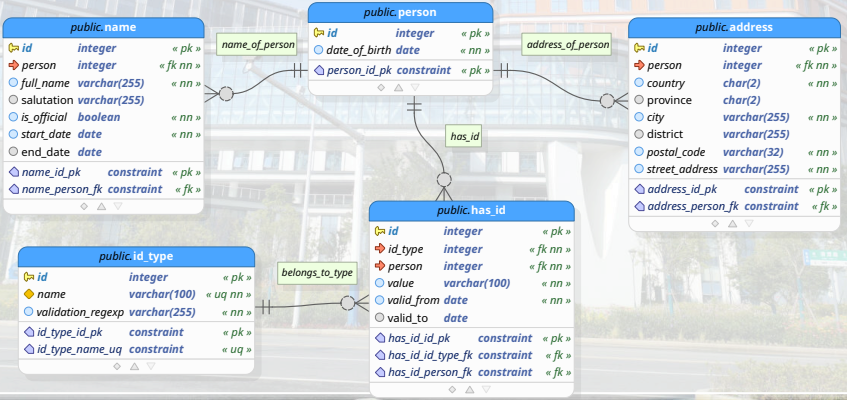
Beispiel: Logische Ebene

- Und wieder bekommen wir eine neue Perspektive auf Beziehungen:
- Seinerzeit hatten wir O >o—o< P mit der zusätzlichen Tabelle `relate_o_and_p` implementiert.
- Eine andere Perspektive wäre, das wir eigentlich zwei Beziehungen implementiert haben.



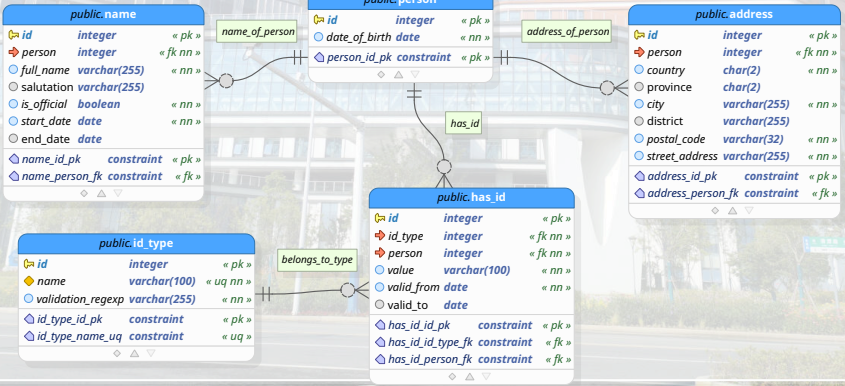
Beispiel: Logische Ebene

- Seinerzeit hatten wir O >o—o< P mit der zusätzlichen Tabelle `relate_o_and_p` implementiert.
- Eine andere Perspektive wäre, das wir eigentlich zwei Beziehungen implementiert haben:
- `o` ||—o< `relate_o_and_p` und `p` ||—o< `relate_o_and_p`.



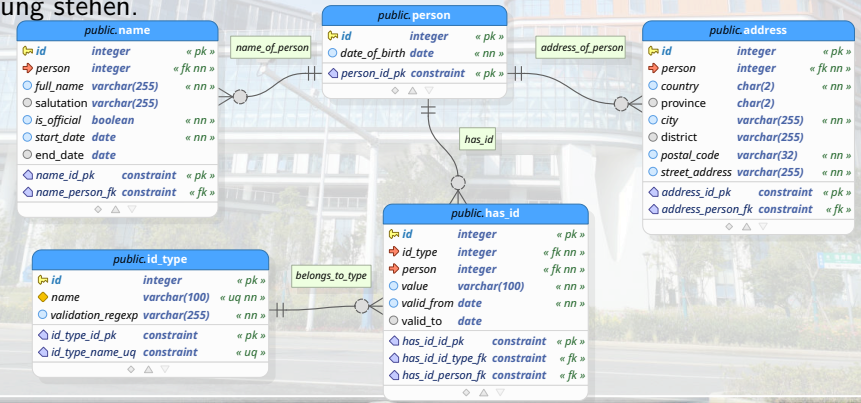
Beispiel: Logische Ebene

- Eine andere Perspektive wäre, das wir eigentlich zwei Beziehungen implementiert haben:
- `o` ||—o< `relate_o_and_p` und `p` ||—o< `relate_o_and_p`.
- Jede Zeile in Tabelle `relate_o_and_p` muss mit einer Zeile in Tabelle `o` in Beziehung stehen *und* mit einer Zeile in Tabele `p`.



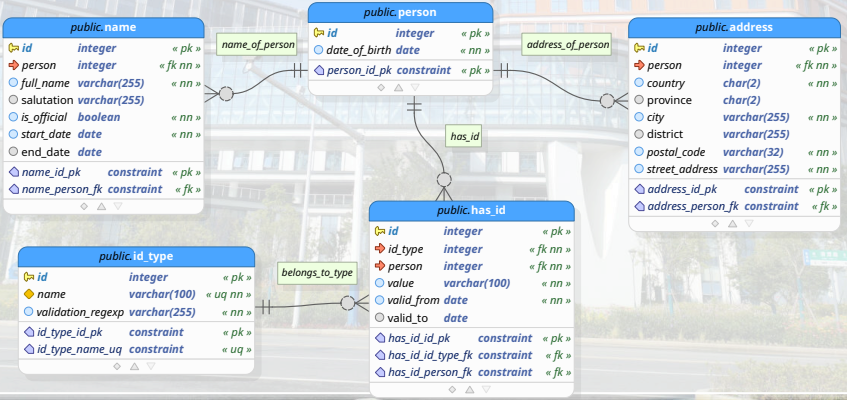
Beispiel: Logische Ebene

- `o` ||—o< `relate_o_and_p` und `p` ||—o< `relate_o_and_p`.
- Jede Zeile in Tabelle `relate_o_and_p` muss mit einer Zeile in Tabelle `o` in Beziehung stehen *und* mit einer Zeile in Tabele `p`.
- Jede Zeile in Tabelle `o` kann mit beliebig vielen Zeilen in Tabelle `relate_o_and_p` in Beziehung stehen.



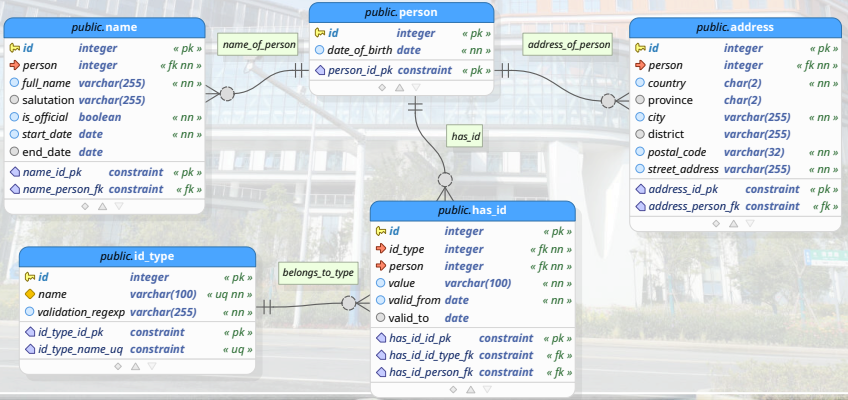
Beispiel: Logische Ebene

- Jede Zeile in Tabelle `o` kann mit beliebig vielen Zeilen in Tabelle `relate_o_and_p` in Beziehung stehen.
- Jede Zeile in Tabelle `p` kann mit beliebig vielen Zeilen in Tabelle `relate_o_and_p` in Beziehung stehen.



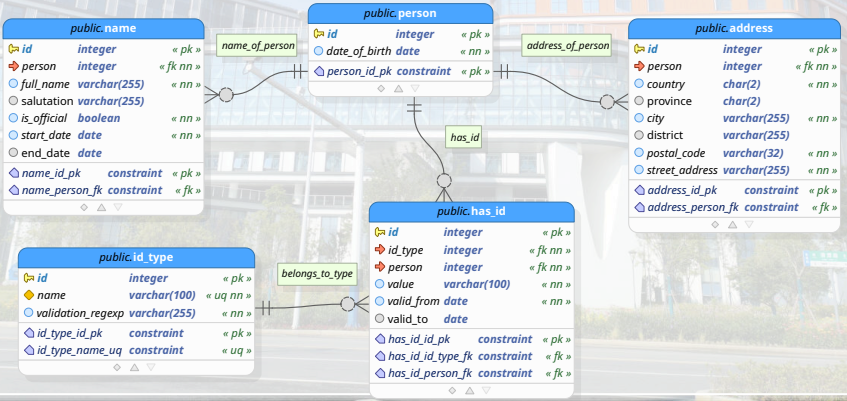
Beispiel: Logische Ebene

- Jede Zeile in Tabelle `p` kann mit beliebig vielen Zeilen in Tabelle `relate_o_and_p` in Beziehung stehen.
- Fürwahr, das erzeugt eine O⋊—⋉P-Beziehung.



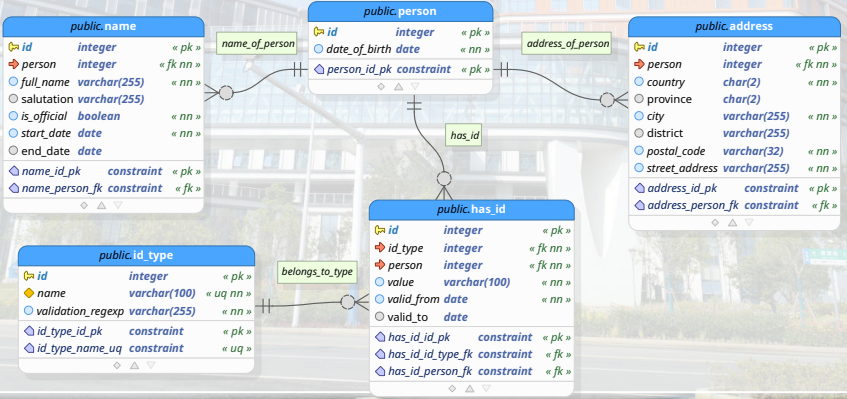
Beispiel: Logische Ebene

- Jede Zeile in Tabelle `p` kann mit beliebig vielen Zeilen in Tabelle `relate_o_and_p` in Beziehung stehen.
- Fürwahr, das erzeugt eine O⋊—⋉P-Beziehung.
- Und das ist auch in dem Diagramm für das logische Modell hier reflektiert.



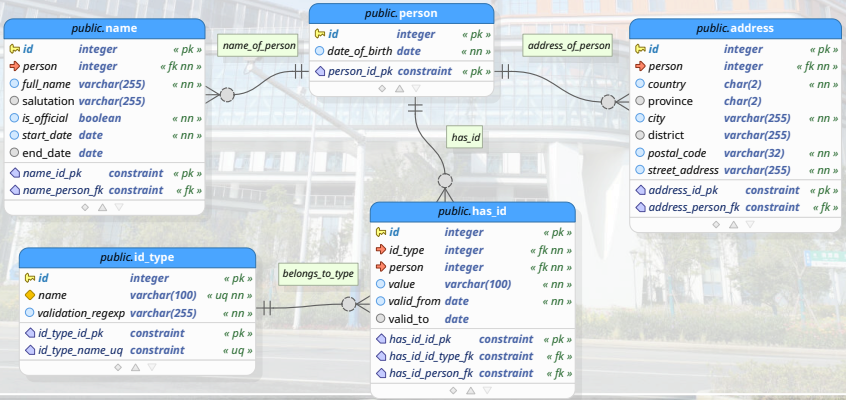
Beispiel: Logische Ebene

- Fürwahr, das erzeugt eine O⩾—o⩽P-Beziehung.
- Und das ist auch in dem Diagramm für das logische Modell hier reflektiert.
- Auch begegnen uns hier wieder mehrwertige Attribute: *Name* und *Address*.



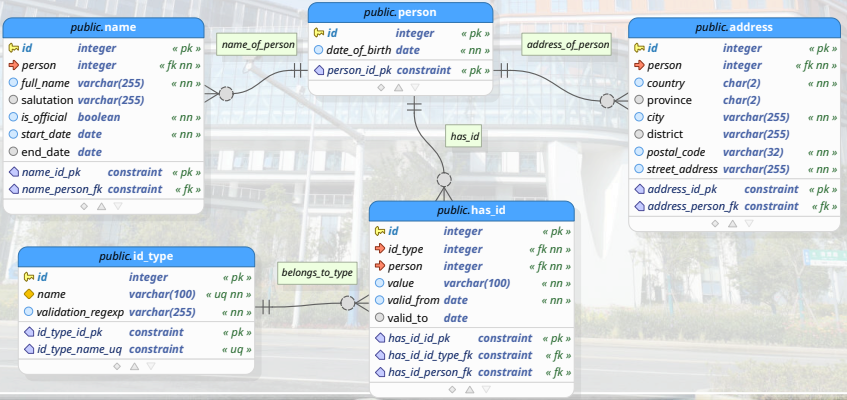
Beispiel: Logische Ebene

- Und das ist auch in dem Diagramm für das logische Modell hier reflektiert.
- Auch begegnen uns hier wieder mehrwertige Attribute: *Name* und *Address*.
- Wie wir schon wissen, gehen die in zusätzliche Tabellen, die wir hier `name` und `address` nennen.



Beispiel: Logische Ebene

- Auch begegnen uns hier wieder mehrwertige Attribute: *Name* und *Address*.
- Wie wir schon wissen, gehen die in zusätzliche Tabellen, die wir hier `name` und `address` nennen.
- Natürlich erzeugen wir wieder passende Fremdschlüssel-`REFERENCES`-Einschränkungen.



Generiertes SQL

Datenbank erstellen

- Erstellen wir nun die Datenbank und Tabellen mit den Skripten, die der PgModeler für uns generiert hat.

Datenbank erstellen

- Erstellen wir nun die Datenbank und Tabellen mit den Skripten, die der PgModeler für uns generiert hat.
- Fangen wir mit der Datenbank an.

```
1 -- object: person_database | type: DATABASE --
2 -- DROP DATABASE IF EXISTS person_database;
3 CREATE DATABASE person_database;
4 -- ddl-end --
```

```
1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -ebf 01
    ↪ _person_database_database_2001.sql
2 CREATE DATABASE
3 # psql 16.11 succeeded with exit code 0.
```

Tabelle person

- Hier sehen wir das auto-generierte Skript, um die Tabelle `person` zu erstellen.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    date_of_birth date NOT NULL,
    CONSTRAINT person_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.person OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_person_table_5071.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle person

- Hier sehen wir das auto-generierte Skript, um die Tabelle `person` zu erstellen.
- Wir verwenden einen Ersatz-Primärschlüssel.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    date_of_birth date NOT NULL,
    CONSTRAINT person_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.person OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_person_table_5071.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle person

- Hier sehen wir das auto-generierte Skript, um die Tabelle `person` zu erstellen.
- Wir verwenden einen Ersatz-Primärschlüssel.
- Dazu gibt es ein Attribut `date_of_birth` für das Geburtsdatum.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    date_of_birth date NOT NULL,
    CONSTRAINT person_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.person OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_person_table_5071.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Hier sehen wir das auto-generierte Skript, um die Tabelle `name` zu erstellen.

```sql
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Hier sehen wir das auto-generierte Skript, um die Tabelle `name` zu erstellen.
- Sie ist für das zusammengesetzte, mehrwertige Attribut *Name*.

```sql
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Hier sehen wir das auto-generierte Skript, um die Tabelle `name` zu erstellen.
- Sie ist für das zusammengesetzte, mehrwertige Attribut *Name*.
- Jede Zeile speichert einen Fremdschlüssel `person` der mit der Tabelle `person` über eine `REFERENCES`-Einschränkung verbunden ist.

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Hier sehen wir das auto-generierte Skript, um die Tabelle `name` zu erstellen.
- Sie ist für das zusammengesetzte, mehrwertige Attribut *Name*.
- Jede Zeile speichert einen Fremdschlüssel `person` der mit der Tabelle `person` über eine `REFERENCES`-Einschränkung verbunden ist.
- Diese wird ein paar Slides weiter über `ALTER TABLE` erstellt.

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Hier sehen wir das auto-generierte Skript, um die Tabelle `name` zu erstellen.
- Sie ist für das zusammengesetzte, mehrwertige Attribut *Name*.
- Jede Zeile speichert einen Fremdschlüssel `person` der mit der Tabelle `person` über eine `REFERENCES`-Einschränkung verbunden ist.
- Diese wird ein paar Slides weiter über `ALTER TABLE` erstellt.
- Davon abgesehen fügen wir einen kleinen Gimmick aus Spaß ein.

```
1  -- object: public.name | type: TABLE --
2  -- DROP TABLE IF EXISTS public.name CASCADE;
3  CREATE TABLE public.name (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      person integer NOT NULL,
6      full_name varchar(255) NOT NULL,
7      salutation varchar(255),
8      is_official boolean NOT NULL DEFAULT True,
9      start_date date NOT NULL DEFAULT CURRENT_DATE,
10     end_date date,
11     CONSTRAINT name_id_pk PRIMARY KEY (id)
12 );
13 -- ddl-end --
14 ALTER TABLE public.name OWNER TO postgres;
15 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/person_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Sie ist für das zusammengesetzte, mehrwertige Attribut *Name*.
- Jede Zeile speichert einen Fremdschlüssel `person` der mit der Tabelle `person` über eine `REFERENCES`-Einschränkung verbunden ist.
- Diese wird ein paar Slides weiter über `ALTER TABLE` erstellt.
- Davon abgesehen fügen wir einen kleinen Gimmick aus Spaß ein:
- Die Spalte `start_date` wird generiert als `NOT NULL DEFAULT CURRENT_DATE`.

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Jede Zeile speichert einen Fremdschlüssel `person` der mit der Tabelle `person` über eine `REFERENCES`-Einschränkung verbunden ist.
- Diese wird ein paar Slides weiter über `ALTER TABLE` erstellt.
- Davon abgesehen fügen wir einen kleinen Gimmick aus Spaß ein:
- Die Spalte `start_date` wird generiert als `NOT NULL DEFAULT CURRENT_DATE`.
- Das bedeutet, das für jede Zeile in Tabelle `name` ein vernünftiges Start-Datum gespeichert werden muss (wegen dem `NOT NULL`).

```
1  -- object: public.name | type: TABLE --
2  -- DROP TABLE IF EXISTS public.name CASCADE;
3  CREATE TABLE public.name (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      person integer NOT NULL,
6      full_name varchar(255) NOT NULL,
7      salutation varchar(255),
8      is_official boolean NOT NULL DEFAULT True,
9      start_date date NOT NULL DEFAULT CURRENT_DATE,
10     end_date date,
11     CONSTRAINT name_id_pk PRIMARY KEY (id)
12 );
13 -- ddl-end --
14 ALTER TABLE public.name OWNER TO postgres;
15 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/person_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Diese wird ein paar Slides weiter über `ALTER TABLE` erstellt.
- Davon abgesehen fügen wir einen kleinen Gimmick aus Spaß ein:
- Die Spalte `start_date` wird generiert als `NOT NULL DEFAULT CURRENT_DATE`.
- Das bedeutet, das für jede Zeile in Tabelle `name` ein vernünftiges Start-Datum gespeichert werden muss (wegen dem `NOT NULL`).
- Wenn wir es beim Erstellen der Zeile nicht angeben, dann wird stattdessen ein `DEFAULT`-Wert gespeichert.

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Davon abgesehen fügen wir einen kleinen Gimmick aus Spaß ein:
- Die Spalte `start_date` wird generiert als `NOT NULL DEFAULT CURRENT_DATE`.
- Das bedeutet, das für jede Zeile in Tabelle `name` ein vernünftiges Start-Datum gespeichert werden muss (wegen dem `NOT NULL`).
- Wenn wir es beim Erstellen der Zeile nicht angeben, dann wird stattdessen ein `DEFAULT`-Wert gespeichert.
- Dieses `DEFAULT` ist `CURRENT_DATE`, welches, wie der Name schon erklärt, das aktuelle Datum in genau dem Moment ist, wo wir die Zeile erstellen[21].

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle name

- Das bedeutet, das für jede Zeile in Tabelle `name` ein vernünftiges Start-Datum gespeichert werden muss (wegen dem `NOT NULL`).
- Wenn wir es beim Erstellen der Zeile nicht angeben, dann wird stattdessen ein `DEFAULT`-Wert gespeichert.
- Dieses `DEFAULT` ist `CURRENT_DATE`, welches, wie der Name schon erklärt, das aktuelle Datum in genau dem Moment ist, wo wir die Zeile erstellen[21].
- Es git noch ähnliche Funktionen wie `CURRENT_TIME` und, besonders wichtig, `CURRENT_TIMESTAMP`[21], der oft für Timestamping verwendet wird.

```
-- object: public.name | type: TABLE --
-- DROP TABLE IF EXISTS public.name CASCADE;
CREATE TABLE public.name (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    full_name varchar(255) NOT NULL,
    salutation varchar(255),
    is_official boolean NOT NULL DEFAULT True,
    start_date date NOT NULL DEFAULT CURRENT_DATE,
    end_date date,
    CONSTRAINT name_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.name OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_name_table_5075.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle address

- Hier sehen wir das auto-generierte Skript, um die Tabelle `address` zu erstellen.

```
-- object: public.address | type: TABLE --
-- DROP TABLE IF EXISTS public.address CASCADE;
CREATE TABLE public.address (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    country char(2) NOT NULL DEFAULT 'CN',
    province char(2) DEFAULT 'AH',
    city varchar(255) NOT NULL,
    district varchar(255),
    postal_code varchar(32) NOT NULL,
    street_address varchar(255) NOT NULL,
    CONSTRAINT address_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.address OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_address_table_5084.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle address

- Hier sehen wir das auto-generierte Skript, um die Tabelle `address` zu erstellen.
- Da gibt es nicht viel Besonderes zu sehen.

```
-- object: public.address | type: TABLE --
-- DROP TABLE IF EXISTS public.address CASCADE;
CREATE TABLE public.address (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    country char(2) NOT NULL DEFAULT 'CN',
    province char(2) DEFAULT 'AH',
    city varchar(255) NOT NULL,
    district varchar(255),
    postal_code varchar(32) NOT NULL,
    street_address varchar(255) NOT NULL,
    CONSTRAINT address_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.address OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_address_table_5084.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle address

- Hier sehen wir das auto-generierte Skript, um die Tabelle `address` zu erstellen.
- Da gibt es nicht viel Besonderes zu sehen.
- Wir verwenden `DEAULT`-Werte auch für `country` und `province`.

```
-- object: public.address | type: TABLE --
-- DROP TABLE IF EXISTS public.address CASCADE;
CREATE TABLE public.address (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    country char(2) NOT NULL DEFAULT 'CN',
    province char(2) DEFAULT 'AH',
    city varchar(255) NOT NULL,
    district varchar(255),
    postal_code varchar(32) NOT NULL,
    street_address varchar(255) NOT NULL,
    CONSTRAINT address_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.address OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_address_table_5084.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle address

- Hier sehen wir das auto-generierte Skript, um die Tabelle `address` zu erstellen.
- Da gibt es nicht viel Besonderes zu sehen.
- Wir verwenden `DEAULT`-Werte auch für `country` und `province`.
- Wenn man diese beim Eingeben der Daten nicht angibt, werden sie auf `'CN'` und `'AH'`, gesetzt.

```
-- object: public.address | type: TABLE --
-- DROP TABLE IF EXISTS public.address CASCADE;
CREATE TABLE public.address (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    country char(2) NOT NULL DEFAULT 'CN',
    province char(2) DEFAULT 'AH',
    city varchar(255) NOT NULL,
    district varchar(255),
    postal_code varchar(32) NOT NULL,
    street_address varchar(255) NOT NULL,
    CONSTRAINT address_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.address OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_address_table_5084.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle address

- Hier sehen wir das auto-generierte Skript, um die Tabelle `address` zu erstellen.
- Da gibt es nicht viel Besonderes zu sehen.
- Wir verwenden `DEAULT`-Werte auch für `country` und `province`.
- Wenn man diese beim Eingeben der Daten nicht angibt, werden sie auf `'CN'` und `'AH'`, gesetzt.
- Der Fremdschlüssel `person` zeigt auf Tabelle `person` über eine `REFERENCES`-Einschränkung, die wir in paar Slides später zeigen.

```
-- object: public.address | type: TABLE --
-- DROP TABLE IF EXISTS public.address CASCADE;
CREATE TABLE public.address (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    person integer NOT NULL,
    country char(2) NOT NULL DEFAULT 'CN',
    province char(2) DEFAULT 'AH',
    city varchar(255) NOT NULL,
    district varchar(255),
    postal_code varchar(32) NOT NULL,
    street_address varchar(255) NOT NULL,
    CONSTRAINT address_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.address OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_address_table_5084.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle id_type

- Hier sehen wir das auto-generierte Skript, um die Tabelle `id_type` zu erstellen.

```
-- object: public.id_type | type: TABLE --
-- DROP TABLE IF EXISTS public.id_type CASCADE;
CREATE TABLE public.id_type (
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ,
    name varchar(100) NOT NULL,
    validation_regexp varchar(255) NOT NULL DEFAULT '.+',
    CONSTRAINT id_type_id_pk PRIMARY KEY (id),
    CONSTRAINT id_type_name_uq UNIQUE (name)
);
-- ddl-end --
ALTER TABLE public.id_type OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_id_type_table_5094.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle id_type

- Hier sehen wir das auto-generierte Skript, um die Tabelle `id_type` zu erstellen.
- Das machen wir im Grunde genauso wie in der letzten Einheit.

```
-- object: public.id_type | type: TABLE --
-- DROP TABLE IF EXISTS public.id_type CASCADE;
CREATE TABLE public.id_type (
    id integer NOT NULL GENERATED ALWAYS AS IDENTITY ,
    name varchar(100) NOT NULL,
    validation_regexp varchar(255) NOT NULL DEFAULT '.+',
    CONSTRAINT id_type_id_pk PRIMARY KEY (id),
    CONSTRAINT id_type_name_uq UNIQUE (name)
);
-- ddl-end --
ALTER TABLE public.id_type OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_id_type_table_5094.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle has_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `has_id` zu erstellen.

```
-- object: public.has_id | type: TABLE --
-- DROP TABLE IF EXISTS public.has_id CASCADE;
CREATE TABLE public.has_id (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    id_type integer NOT NULL,
    person integer NOT NULL,
    value varchar(100) NOT NULL,
    valid_from date NOT NULL,
    valid_to date,
    CONSTRAINT has_id_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.has_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 07_public_has_id_table_5100.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle has_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `has_id` zu erstellen.
- Sie folgt im Grunde dem selben Design als Tabelle `personal_id` in der vorigen Einheit.

```
1  -- object: public.has_id | type: TABLE --
2  -- DROP TABLE IF EXISTS public.has_id CASCADE;
3  CREATE TABLE public.has_id (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      id_type integer NOT NULL,
6      person integer NOT NULL,
7      value varchar(100) NOT NULL,
8      valid_from date NOT NULL,
9      valid_to date,
10     CONSTRAINT has_id_id_pk PRIMARY KEY (id)
11 );
12 -- ddl-end --
13 ALTER TABLE public.has_id OWNER TO postgres;
14 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/person_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 07_public_has_id_table_5100.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle has_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `has_id` zu erstellen.
- Sie folgt im Grunde dem selben Design als Tabelle `personal_id` in der vorigen Einheit.
- Sie beinhaltet die Beziehungsattribute, die wir im konzeptuellen Modell definiert haben.

```
-- object: public.has_id | type: TABLE --
-- DROP TABLE IF EXISTS public.has_id CASCADE;
CREATE TABLE public.has_id (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    id_type integer NOT NULL,
    person integer NOT NULL,
    value varchar(100) NOT NULL,
    valid_from date NOT NULL,
    valid_to date,
    CONSTRAINT has_id_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.has_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 07_public_has_id_table_5100.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkung (1)

- Hier sehen wir das auto-generierte Skript, um dir Fremdschlüssel-Einschränkung zu erstellen, die sicherstellt, dass jede Zeile in Tabelle `name` mit einer Zeile in Tabelle `person` verbunden ist.

```
1 -- object: name_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.name DROP CONSTRAINT IF EXISTS name_person_fk
    ↪ CASCADE;
3 ALTER TABLE public.name ADD CONSTRAINT name_person_fk FOREIGN KEY (
    ↪ person)
4 REFERENCES public.person (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --
```

```
1 $ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 08_public_name_name_person_fk_constraint_5108
    ↪ .sql
2 ALTER TABLE
3 # psql 16.11 succeeded with exit code 0.
```

Einschränkung (2)

- Hier sehen wir das auto-generierte Skript, um dir Fremdschlüssel-Einschränkung zu erstellen, die sicherstellt, dass jede Zeile in Tabelle `address` mit einer Zeile in Tabelle `person` verbunden ist.

```
1 -- object: address_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.address DROP CONSTRAINT IF EXISTS
    ↪ address_person_fk CASCADE;
3 ALTER TABLE public.address ADD CONSTRAINT address_person_fk FOREIGN KEY
    ↪ (person)
4 REFERENCES public.person (id) MATCH SIMPLE
5 ON DELETE NO ACTION ON UPDATE NO ACTION;
6 -- ddl-end --
```

```
1 $ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 09
    ↪ _public_address_address_person_fk_constraint_5109.sql
2 ALTER TABLE
3 # psql 16.11 succeeded with exit code 0.
```

Einschränkung (3)

- Hier sehen wir das auto-generierte Skript, um dir Fremdschlüssel-Einschränkung zu erstellen, die sicherstellt, dass jede Zeile in Tabelle `has_id` mit einer Zeile in Tabelle `id_type` verbunden ist.

```
-- object: has_id_id_type_fk | type: CONSTRAINT --
-- ALTER TABLE public.has_id DROP CONSTRAINT IF EXISTS has_id_id_type_fk
   ↪ CASCADE;
ALTER TABLE public.has_id ADD CONSTRAINT has_id_id_type_fk FOREIGN KEY (
   ↪ id_type)
REFERENCES public.id_type (id) MATCH SIMPLE
ON DELETE NO ACTION ON UPDATE NO ACTION;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf 10
   ↪ _public_has_id_has_id_id_type_fk_constraint_5110.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkung (4)

- Hier sehen wir das auto-generierte Skript, um dir Fremdschlüssel-Einschränkung zu erstellen, die sicherstellt, dass jede Zeile in Tabelle `has_id` mit einer Zeile in Tabelle `person` verbunden ist.

```
-- object: has_id_person_fk | type: CONSTRAINT --
-- ALTER TABLE public.has_id DROP CONSTRAINT IF EXISTS has_id_person_fk
    ↪ CASCADE;
ALTER TABLE public.has_id ADD CONSTRAINT has_id_person_fk FOREIGN KEY (
    ↪ person)
REFERENCES public.person (id) MATCH SIMPLE
ON DELETE NO ACTION ON UPDATE NO ACTION;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 11
    ↪ _public_has_id_has_id_person_fk_constraint_5111.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Fügen wir nun Daten in die Tabellen ein.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Fügen wir nun Daten in die Tabellen ein.
- Wir fangen mit drei Datensätzen für Tabelle `person` an.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Fügen wir nun Daten in die Tabellen ein.
- Wir fangen mit drei Datensätzen für Tabelle `person` an.
- Dabei müssen wir nur jeweils das Geburtsdatum angeben.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',          '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Fügen wir nun Daten in die Tabellen ein.
- Wir fangen mit drei Datensätzen für Tabelle `person` an.
- Dabei müssen wir nur jeweils das Geburtsdatum angeben.
- Danach speichern wir die Namen der drei Leute in Tabelle `name`.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',          '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Fügen wir nun Daten in die Tabellen ein.
- Wir fangen mit drei Datensätzen für Tabelle `person` an.
- Dabei müssen wir nur jeweils das Geburtsdatum angeben.
- Danach speichern wir die Namen der drei Leute in Tabelle `name`.
- Interessanterweise speichern wir vier Namenseinträge.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Wir fangen mit drei Datensätzen für Tabelle `person` an.
- Dabei müssen wir nur jeweils das Geburtsdatum angeben.
- Danach speichern wir die Namen der drei Leute in Tabelle `name`.
- Interessanterweise speichern wir vier Namenseinträge.
- Der Grund ist, dass am 2. February 1989 die 26-jährige *Ms. Bibbi* geheiratet hat und sich entschlossen hat, den Namen ihres Ehemannes zu übernehmen.

```sql
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Dabei müssen wir nur jeweils das Geburtsdatum angeben.
- Danach speichern wir die Namen der drei Leute in Tabelle `name`.
- Interessanterweise speichern wir vier Namenseinträge.
- Der Grund ist, dass am 2. February 1989 die 26-jährige *Ms. Bibbi* geheiratet hat und sich entschlossen hat, den Namen ihres Ehemannes zu übernehmen.
- Ihr Name hat sich also zu *Mrs. Bebba* geändert.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Danach speichern wir die Namen der drei Leute in Tabelle `name`.
- Interessanterweise speichern wir vier Namenseinträge.
- Der Grund ist, dass am 2. February 1989 die 26-jährige *Ms. Bibbi* geheiratet hat und sich entschlossen hat, den Namen ihres Ehemannes zu übernehmen.
- Ihr Name hat sich also zu *Mrs. Bebba* geändert.
- Das sehen wir an den letzten beiden Zeilen, die in Tabelle `name` eingefügt wurden.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Der Grund ist, dass am 2. February 1989 die 26-jährige *Ms. Bibbi* geheiratet hat und sich entschlossen hat, den Namen ihres Ehemannes zu übernehmen.
- Ihr Name hat sich also zu *Mrs. Bebba* geändert.
- Das sehen wir an den letzten beiden Zeilen, die in Tabelle `name` eingefügt wurden.
- Die selben beiden ID-Typen wie in den vergangenen Beispielen – chinesische Ausweisnummer und Mobiltelefonnummer – werden in Tabelle `id_type` erstellt.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Ihr Name hat sich also zu *Mrs. Bebba* geändert.
- Das sehen wir an den letzten beiden Zeilen, die in Tabelle `name` eingefügt wurden.
- Die selben beiden ID-Typen wie in den vergangenen Beispielen – chinesische Ausweisnummer und Mobiltelefonnummer – werden in Tabelle `id_type` erstellt.
- Für jeden ID-Typ und jede Person speichern wir dann einen Wert in Tabelle `has_id`.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Ihr Name hat sich also zu *Mrs. Bebba* geändert.
- Das sehen wir an den letzten beiden Zeilen, die in Tabelle `name` eingefügt wurden.
- Die selben beiden ID-Typen wie in den vergangenen Beispielen – chinesische Ausweisnummer und Mobiltelefonnummer – werden in Tabelle `id_type` erstellt.
- Für jeden ID-Typ und jede Person speichern wir dann einen Wert in Tabelle `has_id`.
- Wir benutzen dabei die Beziehungsattribute, die dort gespeichert werden.

```
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Das sehen wir an den letzten beiden Zeilen, die in Tabelle `name` eingefügt wurden.
- Die selben beiden ID-Typen wie in den vergangenen Beispielen – chinesische Ausweisnummer und Mobiltelefonnummer – werden in Tabelle `id_type` erstellt.
- Für jeden ID-Typ und jede Person speichern wir dann einen Wert in Tabelle `has_id`.
- Wir benutzen dabei die Beziehungsattribute, die dort gespeichert werden.
- Wir geben dann noch für jede Person einen `address`-Datensatz ein.

```sql
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Die selben beiden ID-Typen wie in den vergangenen Beispielen – chinesische Ausweisnummer und Mobiltelefonnummer – werden in Tabelle `id_type` erstellt.
- Für jeden ID-Typ und jede Person speichern wir dann einen Wert in Tabelle `has_id`.
- Wir benutzen dabei die Beziehungsattribute, die dort gespeichert werden.
- Wir geben dann noch für jede Person einen `address`-Datensatz ein.
- Damit haben wir alle Daten eingegeben.

```sql
/** Insert some data into the tables of our person database. */

-- Create the persons Bebbo, Bibbo, and Bebba.
INSERT INTO person (date_of_birth) VALUES
    ('2005-08-07'), ('1995-01-02'), ('1963-11-13');

-- Create the names of Bebbo, Bibbo, and Bebba.
INSERT INTO name (person, full_name, salutation, is_official,
                  start_date, end_date) VALUES
    (1, 'Bebbo', 'Bebbo Machine', TRUE,  '2005-08-07', NULL),
    (2, 'Bibbo', 'The Bib-Man',   TRUE,  '1995-01-02', NULL),
    (3, 'Bibbi', 'Ms. Bibbi',     FALSE, '1963-11-13', '1989-02-12'),
    (3, 'Bebba', 'Mrs. Bebba',    TRUE,  '1989-02-12', NULL);

-- Create two ID types: Chinese national ID and mobile phone numbers.
INSERT INTO id_type (name, validation_regexp) VALUES
    ('national ID',         '^\d{6}((19)|(20))\d{9}[0-9X]$'),
    ('mobile phone number', '^\d{11}$');

-- Insert the personal IDs and mobile phone numbers of the people.
INSERT INTO has_id (id_type, person, value, valid_from) VALUES
    (1, 1, '123456200508071234', '2021-09-21'),
    (1, 2, '123456199501021234', '2024-12-01'),
    (1, 3, '123456196311131234', '1983-10-03'),
    (2, 1, '22222222222',        '2020-09-12'),
    (2, 2, '11111111111',        '2012-07-30'),
    (2, 3, '44444444444',        '2012-07-30');

-- Provide an address for each person.
INSERT INTO address (person, country, province, city,
                     district, postal_code, street_address) VALUES
    (1, 'DE', 'SN', 'Chemnitz', 'Zentrum', '09111', 'Rathaus'),
    (2, 'CN', 'AH', 'Hefei', 'Jinkaiqu', '230601', 'Hefei University'),
    (3, 'US', 'NY', 'New York', 'Manhattan', '10036', 'Times Square');
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert.sql
INSERT 0 3
INSERT 0 4
INSERT 0 2
INSERT 0 6
INSERT 0 3
# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Und jetzt lesen wir die Daten wieder aus.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                city || ', ' || street_address ||
                ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Und jetzt lesen wir die Daten wieder aus.
- Wir verwenden dafür das vielleicht größte `INNER JOIN`, das wir bisher verwendet haben.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                city || ', ' || street_address ||
                ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Und jetzt lesen wir die Daten wieder aus.
- Wir verwenden dafür das vielleicht größte `INNER JOIN`, das wir bisher verwendet haben.
- Wir wollen die Namen der Personen mit ihren Addressen und Mobiltelefonnummern kombinieren.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Und jetzt lesen wir die Daten wieder aus.
- Wir verwenden dafür das vielleicht größte `INNER JOIN`, das wir bisher verwendet haben.
- Wir wollen die Namen der Personen mit ihren Addressen und Mobiltelefonnummern kombinieren.
- Das geht, in dem wir den Zeichenketten-Verbinden-Operator `||` verwenden.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Und jetzt lesen wir die Daten wieder aus.
- Wir verwenden dafür das vielleicht größte `INNER JOIN`, das wir bisher verwendet haben.
- Wir wollen die Namen der Personen mit ihren Addressen und Mobiltelefonnummern kombinieren.
- Das geht, in dem wir den Zeichenketten-Verbinden-Operator `||` verwenden.
- Um die benötigten Daten zu bekommen, gehen wir zeilenweise durch die Tabelle `name`.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Wir wollen die Namen der Personen mit ihren Addressen und Mobiltelefonnummern kombinieren.
- Das geht, in dem wir den Zeichenketten-Verbinden-Operator `||` verwenden.
- Um die benötigten Daten zu bekommen, gehen wir zeilenweise durch die Tabelle `name`.
- Später, in der `WHERE`-Klausel, stellen wir sicher, das wir nur die aktuell gültigen Namen verwenden, in dem wir mit `name.is_official = TRUE` selektieren.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Das geht, in dem wir den Zeichenketten-Verbinden-Operator `||` verwenden.
- Um die benötigten Daten zu bekommen, gehen wir zeilenweise durch die Tabelle `name`.
- Später, in der `WHERE`-Klausel, stellen wir sicher, das wir nur die aktuell gültigen Namen verwenden, in dem wir mit `name.is_official = TRUE` selektieren.
- Für jede solche Zeile in Tabelle `name` holen wir uns die passende Zeile in Tabelle `person` über den Fremdschlüssel.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Um die benötigten Daten zu bekommen, gehen wir zeilenweise durch die Tabelle `name`.
- Später, in der `WHERE`-Klausel, stellen wir sicher, das wir nur die aktuell gültigen Namen verwenden, in dem wir mit `name.is_official = TRUE` selektieren.
- Für jede solche Zeile in Tabelle `name` holen wir uns die passende Zeile in Tabelle `person` über den Fremdschlüssel.
- Mit dem Primärschlüssel dieses Datensatzes können wir uns dann die passenden Zeilen aus den Tabellen `address` und `has_id` holen.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-------------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Für jede solche Zeile in Tabelle `name` holen wir uns die passende Zeile in Tabelle `person` über den Fremdschlüssel.
- Mit dem Primärschlüssel dieses Datensatzes können wir uns dann die passenden Zeilen aus den Tabellen `address` und `has_id` holen.
- Wir behalten nur die IDs vom Typ `2` über `has_id.id_type = 2` in der `WHERE`-Klausel, also die Mobiltelefonnummern.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-------------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Für jede solche Zeile in Tabelle `name` holen wir uns die passende Zeile in Tabelle `person` über den Fremdschlüssel.
- Mit dem Primärschlüssel dieses Datensatzes können wir uns dann die passenden Zeilen aus den Tabellen `address` und `has_id` holen.
- Wir behalten nur die IDs vom Typ `2` über `has_id.id_type = 2` in der `WHERE`-Klausel, also die Mobiltelefonnummern.
- Wenn eine Person keine Mobiltelefonnummer oder Adresse hätte, dann würde sie nicht in der Liste auftauchen.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Mit dem Primärschlüssel dieses Datensatzes können wir uns dann die passenden Zeilen aus den Tabellen `address` und `has_id` holen.
- Wir behalten nur die IDs vom Typ `2` über `has_id.id_type = 2` in der `WHERE`-Klausel, also die Mobiltelefonnummern.
- Wenn eine Person keine Mobiltelefonnummer oder Adresse hätte, dann würde sie nicht in der Liste auftauchen.
- Was würde passieren, wenn eine Person mehrere Mobiltelefonnummern und/oder Adressen hat?

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Mit dem Primärschlüssel dieses Datensatzes können wir uns dann die passenden Zeilen aus den Tabellen `address` und `has_id` holen.
- Wir behalten nur die IDs vom Typ `2` über `has_id.id_type = 2` in der `WHERE`-Klausel, also die Mobiltelefonnummern.
- Wenn eine Person keine Mobiltelefonnummer oder Adresse hätte, dann würde sie nicht in der Liste auftauchen.
- Was würde passieren, wenn eine Person mehrere Mobiltelefonnummern und/oder Adressen hat?
- Probieren Sie es doch selbst aus.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Wir behalten nur die IDs vom Typ 2 über `has_id.id_type = 2` in der `WHERE`-Klausel, also die Mobiltelefonnummern.
- Wenn eine Person keine Mobiltelefonnummer oder Adresse hätte, dann würde sie nicht in der Liste auftauchen.
- Was würde passieren, wenn eine Person mehrere Mobiltelefonnummern und/oder Adressen hat?
- Probieren Sie es doch selbst aus.
- Egal. Wir sortieren die Zeilen im Ergebnis nach dem Geburtsdatum.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                city || ', ' || street_address ||
                ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Wenn eine Person keine Mobiltelefonnummer oder Adresse hätte, dann würde sie nicht in der Liste auftauchen.
- Was würde passieren, wenn eine Person mehrere Mobiltelefonnummern und/oder Adressen hat?
- Probieren Sie es doch selbst aus.
- Egal. Wir sortieren die Zeilen im Ergebnis nach dem Geburtsdatum.
- Die älteste Person kommt also zuerst.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                city || ', ' || street_address ||
                ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-------------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Daten Auslesen

- Was würde passieren, wenn eine Person mehrere Mobiltelefonnummern und/oder Adressen hat?
- Probieren Sie es doch selbst aus.
- Egal. Wir sortieren die Zeilen im Ergebnis nach dem Geburtsdatum.
- Die älteste Person kommt also zuerst.
- Wir bekommen das erwartete Ergebnis.

```
/** Combine people, names, addresses, and phone numbers. */

SELECT full_name || ': ' || country || '-' || province || ', ' ||
                  city || ', ' || street_address ||
                  ' (call ' || value || ')' AS contact FROM name
    INNER JOIN person  ON person.id = name.person
    INNER JOIN address ON person.id = address.person
    INNER JOIN has_id  ON person.id = has_id.person
    WHERE name.is_official = TRUE AND  -- use official name only
          has_id.id_type   = 2       -- use ID type for mobile phone
    ORDER BY person.date_of_birth;  -- sort by age, oldest people first
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf select.sql
                          contact
-----------------------------------------------------------
 Bebba: US-NY, New York, Times Square (call 44444444444)
 Bibbo: CN-AH, Hefei, Hefei University (call 11111111111)
 Bebbo: DE-SN, Chemnitz, Rathaus (call 22222222222)
(3 rows)

# psql 16.11 succeeded with exit code 0.
```

Aufräumen

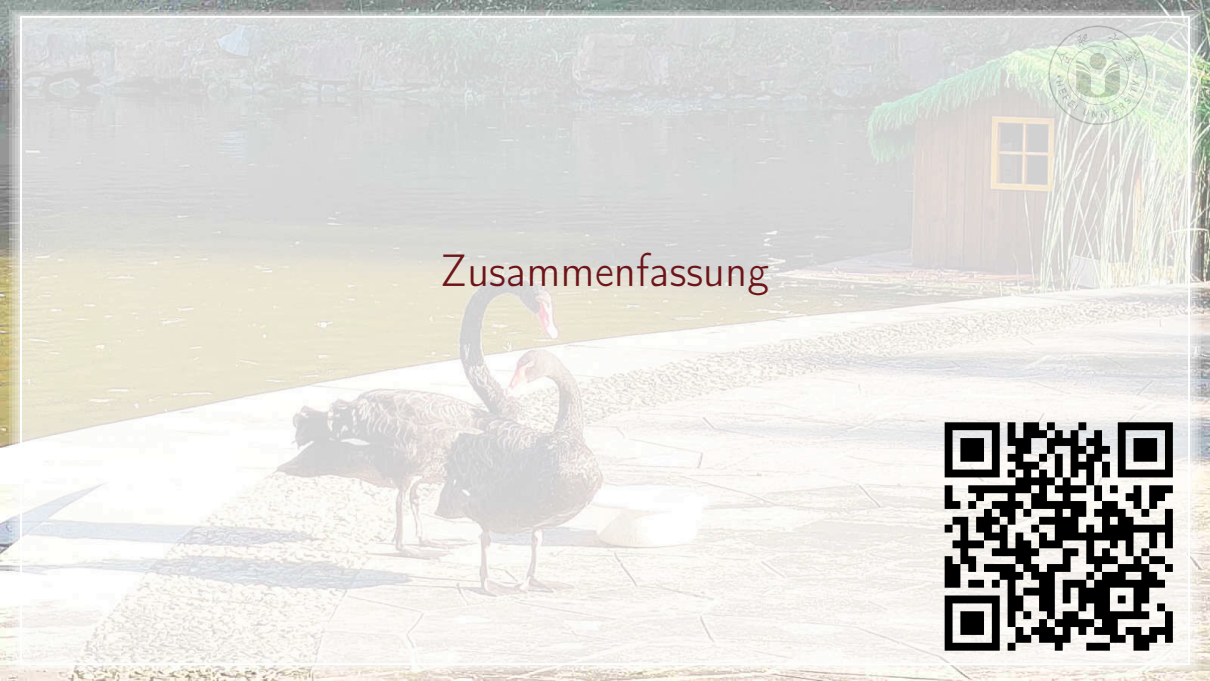
- Räumen wir nun auf.

```
/* Cleanup after the example: Delete the person database. */

DROP DATABASE IF EXISTS person_database;
```

```
$ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -ebf
    ↪ cleanup.sql
DROP DATABASE
# psql 16.11 succeeded with exit code 0.
```

Zusammenfassung



Zusammenfassung

- Was haben wir also gelernt?

Zusammenfassung

- Was haben wir also gelernt?
- Auf der konzeptuellen Ebene gibt es Beziehungsattribute.

Zusammenfassung

- Was haben wir also gelernt?
- Auf der konzeptuellen Ebene gibt es Beziehungsattribute.
- In logischen Schemas basierend auf dem relationalen Datenmodell gibt es Beziehungen nicht als eigenständige Objekte.

Zusammenfassung

- Was haben wir also gelernt?
- Auf der konzeptuellen Ebene gibt es Beziehungsattribute.
- In logischen Schemas basierend auf dem relationalen Datenmodell gibt es Beziehungen nicht als eigenständige Objekte.
- Deshalb haben sie auch keine Attribute.

Zusammenfassung

- Was haben wir also gelernt?
- Auf der konzeptuellen Ebene gibt es Beziehungsattribute.
- In logischen Schemas basierend auf dem relationalen Datenmodell gibt es Beziehungen nicht als eigenständige Objekte.
- Deshalb haben sie auch keine Attribute.
- Deshalb werden die Beziehungsattribute von der konzeptuellen Ebene zu Spalten in Tabellen im logischen Schema.

Zusammenfassung

- Was haben wir also gelernt?
- Auf der konzeptuellen Ebene gibt es Beziehungsattribute.
- In logischen Schemas basierend auf dem relationalen Datenmodell gibt es Beziehungen nicht als eigenständige Objekte.
- Deshalb haben sie auch keine Attribute.
- Deshalb werden die Beziehungsattribute von der konzeptuellen Ebene zu Spalten in Tabellen im logischen Schema.
- Und diese Spalten tun wir dann in die Tabellen, die die jeweiligen Beziehungen repräsentieren.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Raphael „rkhaotix“ Araújo e Silva. *pgModeler – PostgreSQL Database Modeler*. Palmas, Tocantins, Brazil, 2006–2025. URL: https://pgmodeler.io (besucht am 2025-04-12) (siehe S. 112).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[3] (siehe S. 104, 112).

[3] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[2] (siehe S. 104, 112).

[4] Richard Barker. *Case*Method: Entity Relationship Modelling (Oracle)*. 1. Aufl. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., Jan. 1990. ISBN: 978-0-201-41696-1 (siehe S. 111).

[5] 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. 112, 113).

[6] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 112).

[7] 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. 113).

[8] 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. 111).

[9] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 112).

[10] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 112).

[11] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 111).

References II

[12] Ben Brumm. "A Guide to the Entity Relationship Diagram (ERD)". In: *Database Star*. Armadale, VIC, Australia: Elevated Online Services PTY Ltd., 30. Juli 2019–23. Dez. 2023. URL: https://www.databasestar.com/entity-relationship-diagram (besucht am 2025-03-29) (siehe S. 111).

[13] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 111, 113).

[14] 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. 113).

[15] Peter Pin-Shan Chen. "Entity-Relationship Modeling: Historical Events, Future Trends, and Lessons Learned". In: *Software Pioneers: Contributions to Software Engineering*. Hrsg. von Manfred Broy und Ernst Denert. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, Feb. 2002, S. 296–310. doi:10.1007/978-3-642-59412-0_17. URL: http://bit.csc.lsu.edu/%7Echen/pdf/Chen_Pioneers.pdf (besucht am 2025-03-06) (siehe S. 111).

[16] Peter Pin-Shan Chen. "The Entity-Relationship Model – Toward a Unified View of Data". *ACM Transactions on Database Systems (TODS)* 1(1):9–36, März 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0362-5915. doi:10.1145/320434.320440 (siehe S. 105, 111).

[17] Peter Pin-Shan Chen. "The Entity-Relationship Model: Toward a Unified View of Data". In: *1st International Conference on Very Large Data Bases (VLDB'1975)*. 22.–24. Sep. 1975, Framingham, MA, USA. Hrsg. von Douglas S. Kerr. New York, NY, USA: Association for Computing Machinery (ACM), 1975, S. 173. ISBN: 978-1-4503-3920-9. doi:10.1145/1282480.1282492. See[16] for a more comprehensive introduction. (Siehe S. 111).

[18] 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. 113).

[19] 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. 112).

References III

[20] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 113).

[21] "Date/Time Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.9. URL: https://www.postgresql.org/docs/17/functions-datetime.html (besucht am 2025-05-01) (siehe S. 43–52, 113).

[22] "Date/Time Types". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 8.5. URL: https://www.postgresql.org/docs/17/datatype-datetime.html (besucht am 2025-02-28) (siehe S. 113).

[23] 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. 113).

[24] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 113).

[25] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 113).

[26] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 112).

[27] 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. 112).

[28] 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. 112).

[29] 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. 112).

[30] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 112).

References IV

[31] 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. 111, 113).

[32] ."When should a database table use timestamps?" In: *Software Engineering*. Hrsg. von GWed. New York, NY, USA: Stack Exchange Inc., 29. Jan. 2014–16. Sep. 2016. URL: https://softwareengineering.stackexchange.com/questions/225903 (besucht am 2025-02-27) (siehe S. 113).

[33] 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. 112).

[34] *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 (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 113).

[35] Shannon Kempe und Paul Williams. *A Short History of the ER Diagram and Information Modeling*. Studio City, CA, USA: Dataversity Digital LLC, 25. Sep. 2012. URL: https://www.dataversity.net/a-short-history-of-the-er-diagram-and-information-modeling (besucht am 2025-03-06) (siehe S. 111).

[36] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 113).

[37] 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. 112).

[38] 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. 111).

[39] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 112).

References V

[40] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 112).

[41] 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. 113).

[42] 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. 111).

[43] 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. 112).

[44] 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. 111).

[45] *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).

[46] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 112).

[47] 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. 111).

[48] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 112).

[49] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 112).

References VI

[50] 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. 111).

[51] Yuriy Shamshin. "Conceptual Database Model. Entity Relationship Diagram (ERD)". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 04. URL: https://dbs.academy.lv/lection/dbs_LS04EN_erd.pdf (besucht am 2025-03-29) (siehe S. 111).

[52] 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. 112).

[53] 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. 112).

[54] "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. 113).

[55] 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. 109, 113).

[56] 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 of[55] (siehe S. 113).

[57] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 112, 113).

[58] 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. 112).

[59] Kristian Torp, Christian S. Jensen und Richard T. Snodgrass. "Effective Timestamping in Databases". 8(3-4):267–288, Feb. 2000. doi:10.1007/S007780050008. URL: https://people.cs.aau.dk/~csj/Thesis/pdf/chapter40.pdf (besucht am 2025-05-01) (siehe S. 113).

References VII

[60] 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. 112).

[61] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 112).

[62] 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. 111, 112).

[63] 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. 112).

[64] Matthew West. *Developing High Quality Data Models*. Version: 2.0, Issue: 2.1. London, England, UK: Shell International Limited und European Process Industries STEP Technical Liaison Executive (EPISTLE); Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, 8. Dez. 1995–Dez. 2010. ISBN: 978-0-12-375107-2. URL: https://www.researchgate.net/publication/286610894 (besucht am 2025-03-24). Edited by Julian Fowler (siehe S. 111).

[65] *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. 112).

[66] Ulf Michael „Monty" Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 112).

[67] 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. 111).

[68] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 111).

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[11,42,68]. 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[8,38,44,47,50]. 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*[62].

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 DB[67].

DOB Date of Birth

ERD Entity relationship diagrams show the relationships between objects, e.g., between the tables in a DB and how they reference each other[4,12,15–17,35,51,64]

IT information technology

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[13,31].

Glossary (in English) II

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[5,30,52,60,61]. 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.

MariaDB An open source relational database management system that has forked off from MySQL[2,3,6,26,40,48]. See https://mariadb.org for more information.

Microsoft Windows is a commercial proprietary operating system[10]. 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.

MySQL An open source relational database management system[9,26,49,58,66]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

PgModeler the PostgreSQL DB modeler is a tool that allows for graphical modeling of logical schemas for DBs using an entity relationship diagram (ERD)-like notation[1]. Learn more at https://pgmodeler.io.

PostgreSQL An open source object-relational DBMS[27,43,46,58]. See https://postgresql.org for more information.

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

Python The Python programming language[33,37,39,63], i.e., what you will learn about in our book[63]. Learn more at https://python.org.

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[19,28,29,53,57,62,65].

Glossary (in English) III

server
: In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring 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 servers[13] 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“[36].

SQL
: The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[14,20,23,24,34,41,54–57]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[54].

terminal
: A terminal is a text-based window where you can enter commands and execute them[5,18]. 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 ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

timestamping
: is a technique used in DBs to keep track of the creation and/or modification time of data[32,59]. For each table in a relational database to which it is applied, an additional column with datatype `TIMESTAMP`[22] is added for the creation timestamp. This column is usually annotated as `NOT NULL DEFAULT CURRENT_TIMESTAMP`[21], meaning that the DBMS will automatically store the current date and time when a row is created. If data can be changed, then another column for the last update time can be added to the table as well.

Ubuntu
: is a variant of the open source operating system Linux[18,31]. 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[7,25]