合肥大學
HEFEI UNIVERSITY

Datenbanken

40. Logisches Schema: Beziehungen höheren Grades

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO) | 应用优化研究所
School of Artificial Intelligence and Big Data | 人工智能与大数据学院
Hefei University | 合肥大学
Hefei, Anhui, China | 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).

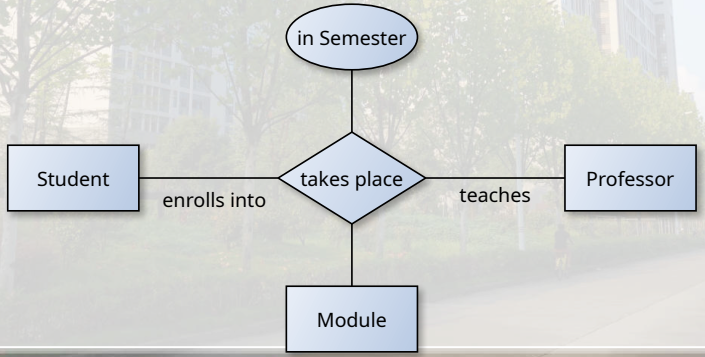
Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/databases (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielen finden Sie unter https://github.com/thomasWeise/databasesCode.

Outline

Einleitung

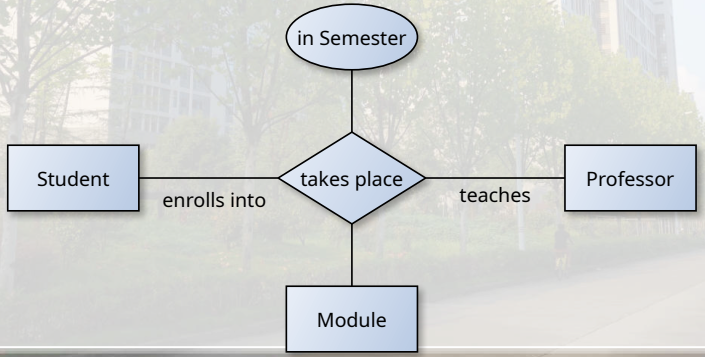
Einleitung

- Hier reproduzieren wir eines unserer frühen ERDs mit einer ternäre Beziehung.



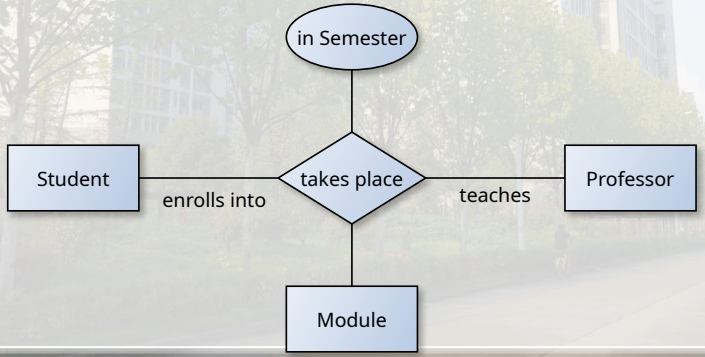
Einleitung

- Hier reproduzieren wir eines unserer frühen ERDs mit einer ternäre Beziehung.
- Die Entitätstypen *Professor*, *Student*, und *Module* stehen miteinander in Beziehung und diese Beziehung hat sogar Attribute.



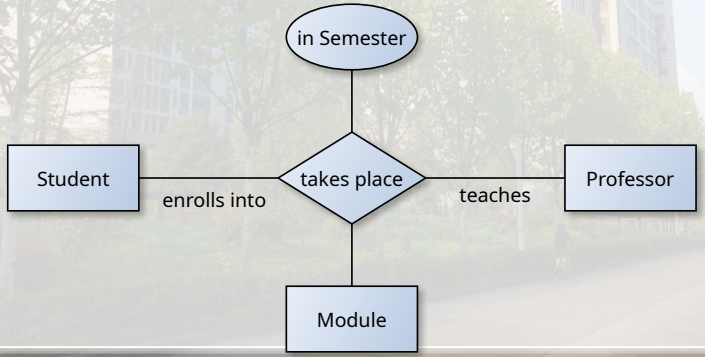
Einleitung

- Hier reproduzieren wir eines unserer frühen ERDs mit einer ternäre Beziehung.
- Die Entitätstypen *Professor*, *Student*, und *Module* stehen miteinander in Beziehung und diese Beziehung hat sogar Attribute.
- Der Professor unterrichtet in Modul in einem bestimmten Semester.



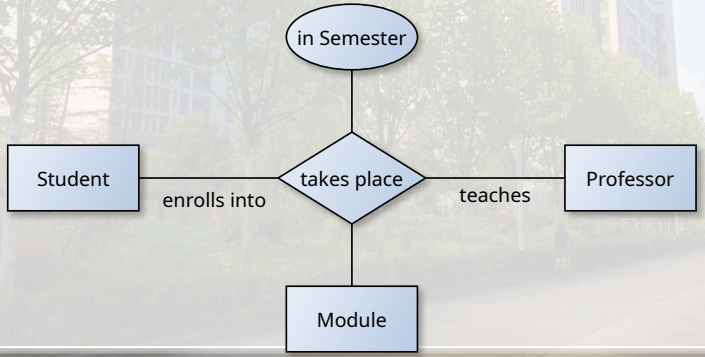
Einleitung

- Hier reproduzieren wir eines unserer frühen ERDs mit einer ternäre Beziehung.
- Die Entitätstypen *Professor*, *Student*, und *Module* stehen miteinander in Beziehung und diese Beziehung hat sogar Attribute.
- Der Professor unterrichtet in Modul in einem bestimmten Semester.
- Die Studentin schreibt sich in das unterrichtete Modul in dem Semester ein.



Einleitung

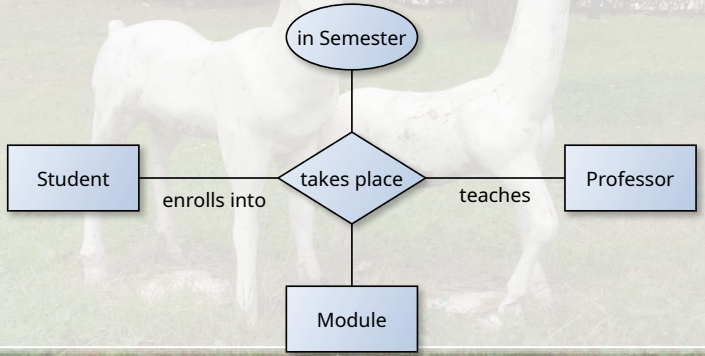
- Die Entitätstypen *Professor*, *Student*, und *Module* stehen miteinander in Beziehung und diese Beziehung hat sogar Attribute.
- Der Professor unterrichtet in Modul in einem bestimmten Semester.
- Die Studentin schreibt sich in das unterrichtete Modul in dem Semester ein.
- Das wollen wir nun in ein logisches Modell transferrieren.



Beispiel

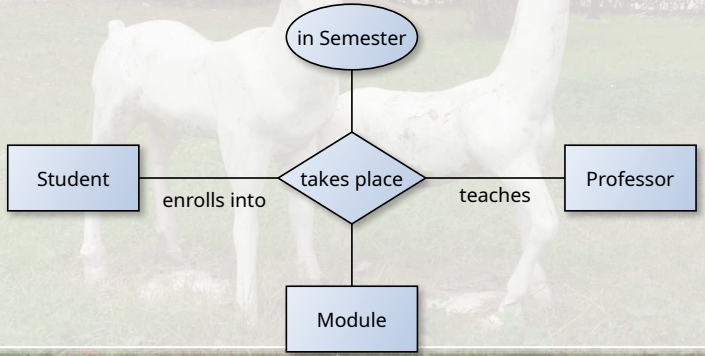
Logisches Modell: Grundlagen

- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.



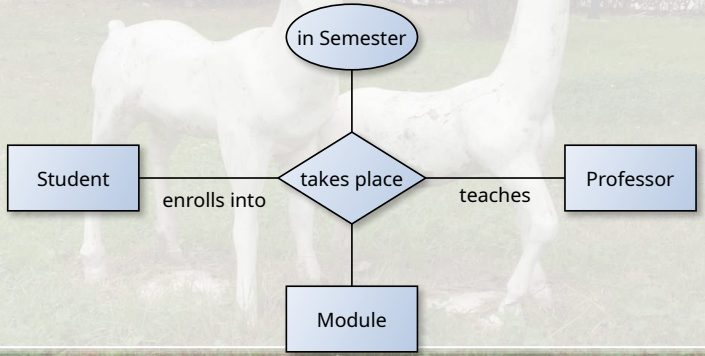
Logisches Modell: Grundlagen

- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.
- Erst machen wir mal die einfachen Dinge.



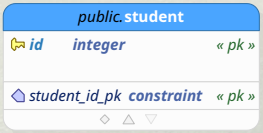
Logisches Modell: Grundlagen

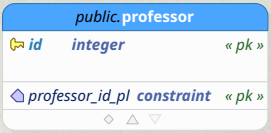
- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.
- Erst machen wir mal die einfachen Dinge.
- Aus den Entitätstypen werden wieder Tabellen mit Ersatz-Primärschlüsseln.



Logisches Modell: Grundlagen

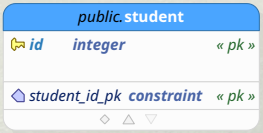
- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.
- Erst machen wir mal die einfachen Dinge.
- Aus den Entitätstypen werden wieder Tabellen mit Ersatz-Primärschlüsseln.

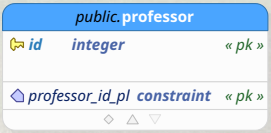




Logisches Modell: Grundlagen

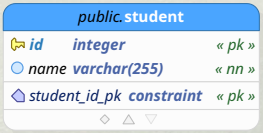
- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.
- Erst machen wir mal die einfachen Dinge.
- Aus den Entitätstypen werden wieder Tabellen mit Ersatz-Primärschlüsseln.
- Wir bekommen die Tabellen `module`, `professor` und `student`.

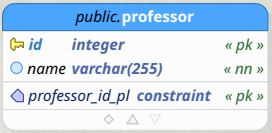




Logisches Modell: Grundlagen

- Bauen wir nun ein logisches Modell, dass zu diesem Szenario passt.
- Erst machen wir mal die einfachen Dinge.
- Aus den Entitätstypen werden wieder Tabellen mit Ersatz-Primärschlüsseln.
- Wir bekommen die Tabellen `module`, `professor` und `student`.
- Damit das Beispiel etwas anschaulicher wird, fügen wir noch das Attribut `name` zu den Tabellen `student` und `professor` hinzu und `title` zu `module`.





Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern. Eine Modul-Instanz ist sozusagen, wenn ein Modul von einem Professor in einem Semester unterrichtet wird.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern. Eine Modul-Instanz ist sozusagen, wenn ein Modul von einem Professor in einem Semester unterrichtet wird. Die Tabelle hat daher Fremdschlüssel auf die Tabellen `professor` und `module` und die Spalte `semester`.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern. Eine Modul-Instanz ist sozusagen, wenn ein Modul von einem Professor in einem Semester unterrichtet wird. Die Tabelle hat daher Fremdschlüssel auf die Tabellen `professor` und `module` und die Spalte `semester`. Wir erstellen dann eine zweite Tabelle `enrolls` mit zwei Fremdschlüsseln, eine auf die Tabelle `student` und eine auf die Tabelle `course`.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern. Eine Modul-Instanz ist sozusagen, wenn ein Modul von einem Professor in einem Semester unterrichtet wird. Die Tabelle hat daher Fremdschlüssel auf die Tabellen `professor` und `module` und die Spalte `semester`. Wir erstellen dann eine zweite Tabelle `enrolls` mit zwei Fremdschlüsseln, eine auf die Tabelle `student` und eine auf die Tabelle `course`. Sie speichert die „schreibt-sich-ein“ (EN: *enrolls*)-Beziehung der Studenten zu den Modul-Instanzen.

Logisches Modell: Beziehungen

- Nun wollen wir die ternäre Beziehung bauen.
- Dafür gibt es im Grunde zwei Möglichkeiten:
  1. Wir können eine einzelne neue Tabelle `takes_place` bauen. Diese bekommt drei Fremdschlüssel auf die Tabellen `professor`, `student` und `module`, sowie eine Spalte `semester`. Sie speichert dann alle Daten der Beziehung.
  2. Wir können eine Tabelle `course` mit einem Ersatz-Primärschlüssel erstellen und benutzen sie, um die Instanzen von Modulen zu speichern. Eine Modul-Instanz ist sozusagen, wenn ein Modul von einem Professor in einem Semester unterrichtet wird. Die Tabelle hat daher Fremdschlüssel auf die Tabellen `professor` und `module` und die Spalte `semester`. Wir erstellen dann eine zweite Tabelle `enrolls` mit zwei Fremdschlüsseln, eine auf die Tabelle `student` und eine auf die Tabelle `course`. Sie speichert die „schreibt-sich-ein“ (EN: *enrolls*)-Beziehung der Studenten zu den Modul-Instanzen.
- Beide Lösungen sind machbar.

Lösung

- Beide Lösungen sind machbar.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile:
  1. Erstens vermeidet er Redundanz.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile:
  1. Erstens vermeidet er Redundanz. Nach der ersten Idee würden wir den Fakt dass „Frau Prof. Bibba“ das Modul „Mathematics 101“ unterrichtet mehrfach speichern.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile:
  1. Erstens vermeidet er Redundanz. Nach der ersten Idee würden wir den Fakt dass „Frau Prof. Bibba“ das Modul „Mathematics 101“ unterrichtet mehrfach speichern. Wir speichern diesen Fakt nun nur noch einmal, nämlich in der Tabelle `course`.

Lösung

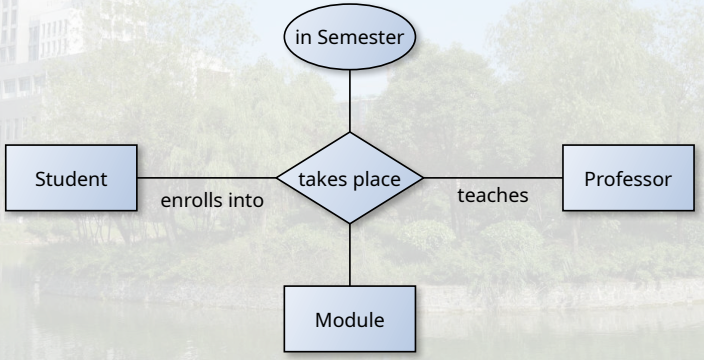
- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile:
  1. Erstens vermeidet er Redundanz. Nach der ersten Idee würden wir den Fakt dass „Frau Prof. Bibba“ das Modul „Mathematics 101“ unterrichtet mehrfach speichern. Wir speichern diesen Fakt nun nur noch einmal, nämlich in der Tabelle `course`.
  2. Mit der zweiten Methode können wir auch die Situation repräsentieren, das ein Professor das selbe Modul zweimal in einem Semester unterichtet.

Lösung

- Beide Lösungen sind machbar. Wir nehmen aber die zweite.
- Die Grundidee ist, dass wir die Realisierung eines Moduls von einem Professor in einem Semester als Entität betrachten können.
- Diese Entität hat das Attribut `semester`, könnte aber auch noch mehr Attribute haben.
- Die Studenten dann mit dieser `course` Tabelle über eine weitere, separate Tabelle in Beziehung zu setzen ergibt Sinn.
- Dieser Ansatz hat zwei große Vorteile:
  1. Erstens vermeidet er Redundanz. Nach der ersten Idee würden wir den Fakt dass „Frau Prof. Bibba“ das Modul „Mathematics 101“ unterrichtet mehrfach speichern. Wir speichern diesen Fakt nun nur noch einmal, nämlich in der Tabelle `course`.
  2. Mit der zweiten Methode können wir auch die Situation repräsentieren, das ein Professor das selbe Modul zweimal in einem Semester unterichtet. Kann ja sein, dass das auf Grund von Zeit- oder Raumkonflikten notwendig ist.

Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.

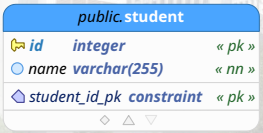


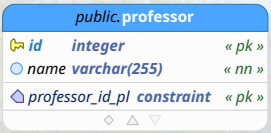
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.

Lösung

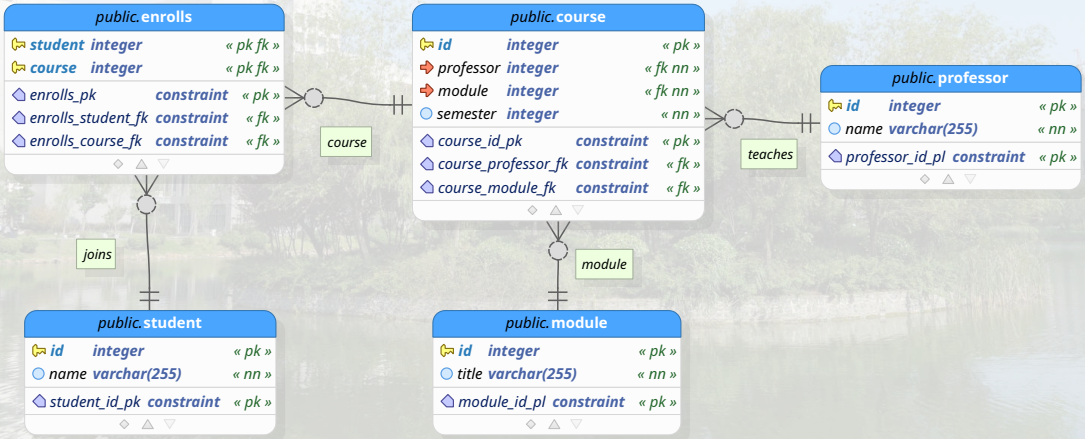
- Wir zeichnen also ein neues ERD mit dem PgModeler.





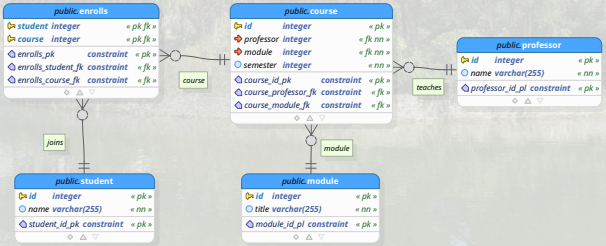
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.



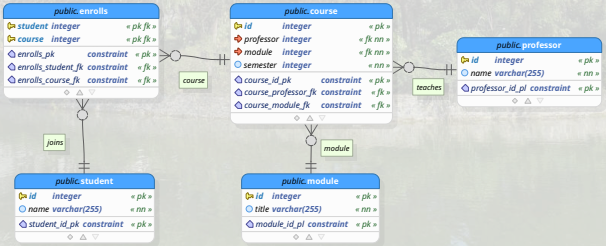
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen.



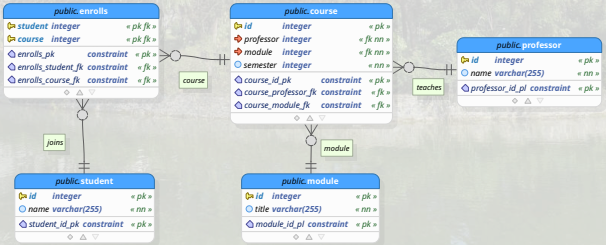
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.



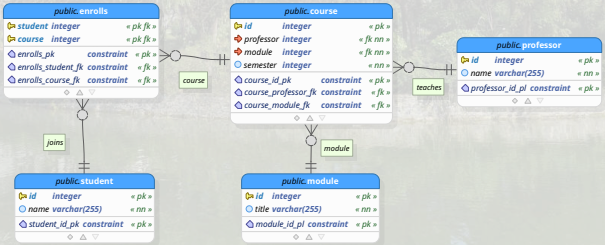
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.
  2. Jeder Kurs gehört zu exakt einem Modul.



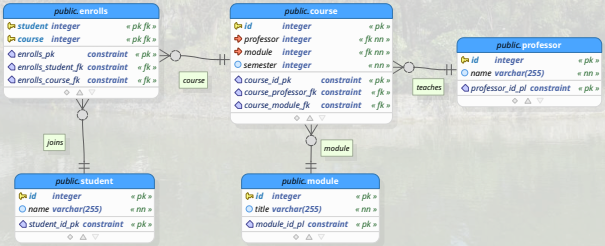
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.
  2. Jeder Kurs gehört zu exakt einem Modul.
  3. Jeder Professor kann beliebig viele Module unterrichten.



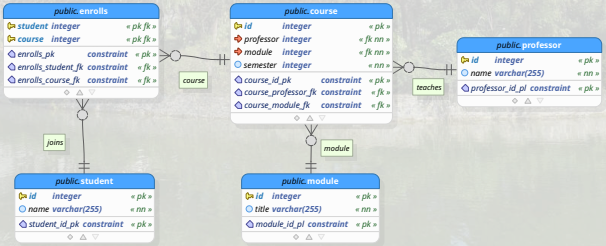
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.
  2. Jeder Kurs gehört zu exakt einem Modul.
  3. Jeder Professor kann beliebig viele Module unterrichten.
  4. Jeder Kurs gehört zu exakt einem Professor.



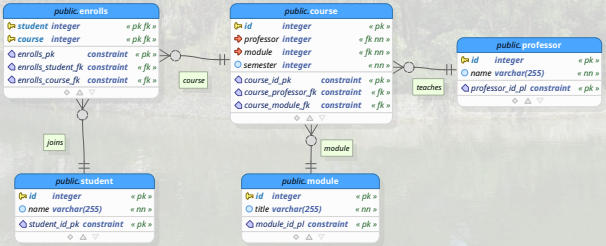
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.
  2. Jeder Kurs gehört zu exakt einem Modul.
  3. Jeder Professor kann beliebig viele Module unterrichten.
  4. Jeder Kurs gehört zu exakt einem Professor.
  5. Jede Studentin kann sich in beliebig viele Kurse einschreiben.



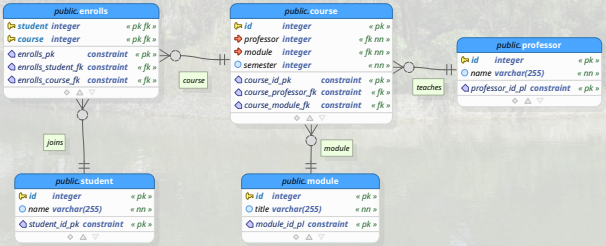
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  1. Jedes Modul kann beliebig oft als Kurs instantiiert werden.
  2. Jeder Kurs gehört zu exakt einem Modul.
  3. Jeder Professor kann beliebig viele Module unterrichten.
  4. Jeder Kurs gehört zu exakt einem Professor.
  5. Jede Studentin kann sich in beliebig viele Kurse einschreiben.
  6. Jede „Einschreibung“ ist exakt einer Studentin und exakt einem Kurs zugeordnet.



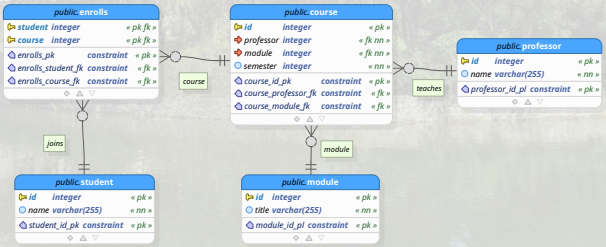
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  2. Jeder Kurs gehört zu exakt einem Modul.
  3. Jeder Professor kann beliebig viele Module unterrichten.
  4. Jeder Kurs gehört zu exakt einem Professor.
  5. Jede Studentin kann sich in beliebig viele Kurse einschreiben.
  6. Jede „Einschreibung“ ist exakt einer Studentin und exakt einem Kurs zugeordnet.
  7. Beliebig viele Studenten können sich in einen Kurs einschreiben.



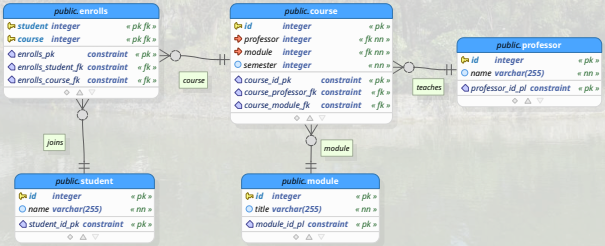
Lösung

- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  3. Jeder Professor kann beliebig viele Module unterrichten.
  4. Jeder Kurs gehört zu exakt einem Professor.
  5. Jede Studentin kann sich in beliebig viele Kurse einschreiben.
  6. Jede „Einschreibung“ ist exakt einer Studentin und exakt einem Kurs zugeordnet.
  7. Beliebig viele Studenten können sich in einen Kurs einschreiben.
  8. Jede Studentin kann sich nicht mehr als einmal in einen Kurs einschreiben.



Lösung

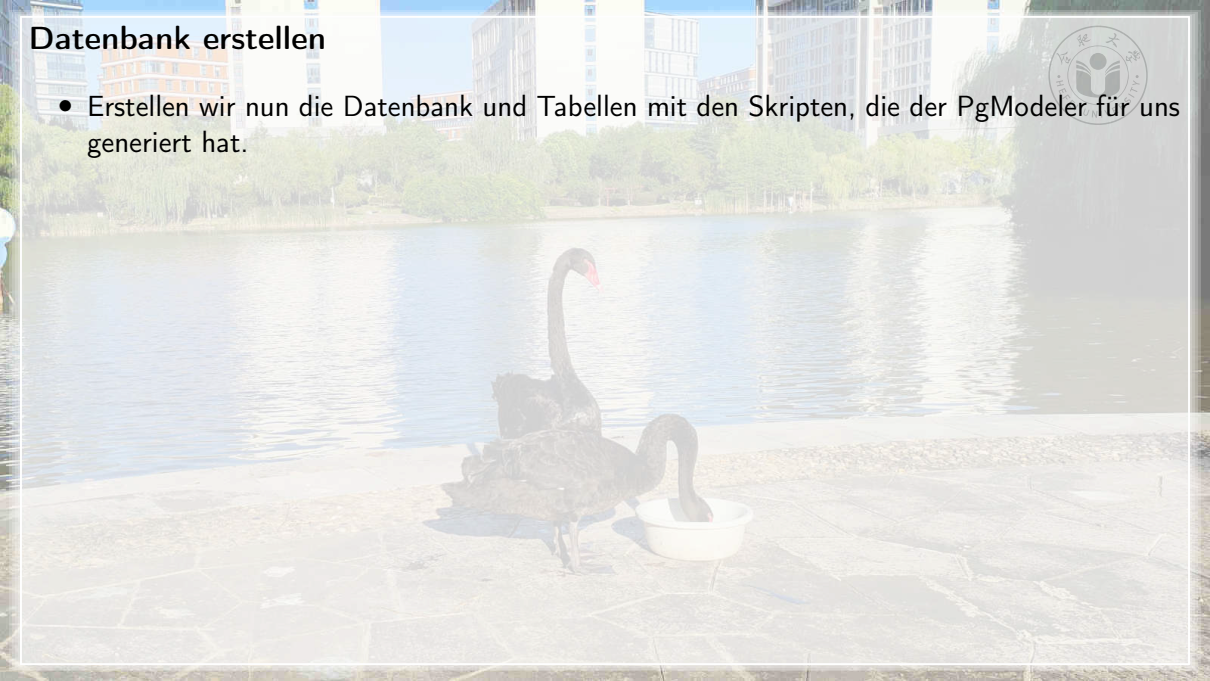
- Wir zeichnen also ein neues ERD mit dem PgModeler.
- Wir treffen folgende Annahmen:
  4. Jeder Kurs gehört zu exakt einem Professor.
  5. Jede Studentin kann sich in beliebig viele Kurse einschreiben.
  6. Jede „Einschreibung“ ist exakt einer Studentin und exakt einem Kurs zugeordnet.
  7. Beliebig viele Studenten können sich in einen Kurs einschreiben.
  8. Jede Studentin kann sich nicht mehr als einmal in einen Kurs einschreiben. Wir machen die beiden Fremdschlüssel in `enrolls` zu einem kombinierten Primärschlüssel und erzwingen so ihre `UNIQUE`ness.



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

```
1 $ psql "postgres://postgres:XXX@localhost" -v ON_ERROR_STOP=1 -ebf 01
    ↪ _teaching_database_database_2001.sql
2 CREATE DATABASE
3 # psql 16.11 succeeded with exit code 0.
```

Tabelle student

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

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    name varchar(255) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_student_table_5071.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle student

- Hier sehen wir das auto-generierte Skript, um die Tabelle `student` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Student* zu speichern.

```sql
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    name varchar(255) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_student_table_5071.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle student

- Hier sehen wir das auto-generierte Skript, um die Tabelle `student` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Student* zu speichern.
- Sie hat den Ersatz-Primärschlüssel `id` und speichert die Namen der Studenten als variabel-langen String.

```
1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      name varchar(255) NOT NULL,
6      CONSTRAINT student_id_pk PRIMARY KEY (id)
7  );
8  -- ddl-end --
9  ALTER TABLE public.student OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 03_public_student_table_5071.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle professor

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

```
-- object: public.professor | type: TABLE --
-- DROP TABLE IF EXISTS public.professor CASCADE;
CREATE TABLE public.professor (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    name varchar(255) NOT NULL,
    CONSTRAINT professor_id_pl PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.professor OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_professor_table_5087.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle professor

- Hier sehen wir das auto-generierte Skript, um die Tabelle `professor` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Professor* zu speichern.

```
-- object: public.professor | type: TABLE --
-- DROP TABLE IF EXISTS public.professor CASCADE;
CREATE TABLE public.professor (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    name varchar(255) NOT NULL,
    CONSTRAINT professor_id_pl PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.professor OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_professor_table_5087.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle professor

- Hier sehen wir das auto-generierte Skript, um die Tabelle `professor` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Professor* zu speichern.
- Sie hat genau die gleiche Struktur, wie die Tabelle `student`.

```
-- object: public.professor | type: TABLE --
-- DROP TABLE IF EXISTS public.professor CASCADE;
CREATE TABLE public.professor (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    name varchar(255) NOT NULL,
    CONSTRAINT professor_id_pl PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.professor OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 04_public_professor_table_5087.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle student

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

```
-- object: public.module | type: TABLE --
-- DROP TABLE IF EXISTS public.module CASCADE;
CREATE TABLE public.module (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    title varchar(255) NOT NULL,
    CONSTRAINT module_id_pl PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.module OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_module_table_5095.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle student

- Hier sehen wir das auto-generierte Skript, um die Tabelle `module` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Module* zu speichern.

```
-- object: public.module | type: TABLE --
-- DROP TABLE IF EXISTS public.module CASCADE;
CREATE TABLE public.module (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
    title varchar(255) NOT NULL,
    CONSTRAINT module_id_pl PRIMARY KEY (id)
);
-- ddl-end --
ALTER TABLE public.module OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_module_table_5095.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle student

- Hier sehen wir das auto-generierte Skript, um die Tabelle `module` zu erstellen.
- Sie ist dafür da, die Entitäten vom Typ *Module* zu speichern.
- Sie hat ebenfalls einen Ersatz-Primärschlüssel `id` und speichert die Titel der Module als variabel-langen String.

```
1  -- object: public.module | type: TABLE --
2  -- DROP TABLE IF EXISTS public.module CASCADE;
3  CREATE TABLE public.module (
4      id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY ,
5      title varchar(255) NOT NULL,
6      CONSTRAINT module_id_pl PRIMARY KEY (id)
7  );
8  -- ddl-end --
9  ALTER TABLE public.module OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
     ↪ ON_ERROR_STOP=1 -ebf 05_public_module_table_5095.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle course

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

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle course

- Hier sehen wir das auto-generierte Skript, um die Tabelle `course` zu erstellen.
- Diese Tabelle ist etwas interessanter.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle course

- Hier sehen wir das auto-generierte Skript, um die Tabelle `course` zu erstellen.
- Diese Tabelle ist etwas interessanter.
- Sie hat natürlich wieder einen Ersatz-Primärschlüssel `id`.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle course

- Hier sehen wir das auto-generierte Skript, um die Tabelle `course` zu erstellen.
- Diese Tabelle ist etwas interessanter.
- Sie hat natürlich wieder einen Ersatz-Primärschlüssel `id`.
- Davon abgesehen speichert sie zwei Fremdschlüssel – `professor` und `module` – die auf die Tabellen mit den selben Namen zeigen (was ein paar Slides weiter mit `REFERENCES`-Einschränkungen erzwungen wird).

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle course

- Diese Tabelle ist etwas interessanter.
- Sie hat natürlich wieder einen Ersatz-Primärschlüssel `id`.
- Davon abgesehen speichert sie zwei Fremdschlüssel – `professor` und `module` – die auf die Tabellen mit den selben Namen zeigen (was ein paar Slides weiter mit `REFERENCES`-Einschränkungen erzwungen wird).
- Dazu speichern wir noch das Semester als einfache Ganzzahl.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle course

- Diese Tabelle ist etwas interessanter.
- Sie hat natürlich wieder einen Ersatz-Primärschlüssel `id`.
- Davon abgesehen speichert sie zwei Fremdschlüssel – `professor` und `module` – die auf die Tabellen mit den selben Namen zeigen (was ein paar Slides weiter mit `REFERENCES`-Einschränkungen erzwungen wird).
- Dazu speichern wir noch das Semester als einfache Ganzzahl.
- Wir werden die „Jahreszahl * 10“ verwenden, gefolgt von einer 1 oder 2, je nachdem, ob wir im Frühlings- oder Herbstsemester sind.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 06_public_course_table_5105.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

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

```
-- object: public.enrolls | type: TABLE --
-- DROP TABLE IF EXISTS public.enrolls CASCADE;
CREATE TABLE public.enrolls (
    student integer NOT NULL,
    course integer NOT NULL,
    CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
);
-- ddl-end --
ALTER TABLE public.enrolls OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

- Hier sehen wir das auto-generierte Skript, um die Tabelle `enrolls` zu erstellen.
- Diese Tabelle ist anders: Sie hat keinen Ersatz-Primärschlüssel!

```
-- object: public.enrolls | type: TABLE --
-- DROP TABLE IF EXISTS public.enrolls CASCADE;
CREATE TABLE public.enrolls (
    student integer NOT NULL,
    course integer NOT NULL,
    CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
);
-- ddl-end --
ALTER TABLE public.enrolls OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

- Hier sehen wir das auto-generierte Skript, um die Tabelle `enrolls` zu erstellen.
- Diese Tabelle ist anders: Sie hat keinen Ersatz-Primärschlüssel!
- Sie referenziert die Tabellen `student` und `course` über entsprechende Fremdschlüssel (die wir nachher mit Einschränkungen durchsetzen).

```
1  -- object: public.enrolls | type: TABLE --
2  -- DROP TABLE IF EXISTS public.enrolls CASCADE;
3  CREATE TABLE public.enrolls (
4      student integer NOT NULL,
5      course integer NOT NULL,
6      CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
7  );
8  -- ddl-end --
9  ALTER TABLE public.enrolls OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
     ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

- Hier sehen wir das auto-generierte Skript, um die Tabelle `enrolls` zu erstellen.
- Diese Tabelle ist anders: Sie hat keinen Ersatz-Primärschlüssel!
- Sie referenziert die Tabellen `student` und `course` über entsprechende Fremdschlüssel (die wir nachher mit Einschränkungen durchsetzen).
- Diese beiden Fremdschlüssel bilden gleichzeitig den Primärschlüssel der Tabelle.

```
1  -- object: public.enrolls | type: TABLE --
2  -- DROP TABLE IF EXISTS public.enrolls CASCADE;
3  CREATE TABLE public.enrolls (
4      student integer NOT NULL,
5      course integer NOT NULL,
6      CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
7  );
8  -- ddl-end --
9  ALTER TABLE public.enrolls OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
     ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

- Hier sehen wir das auto-generierte Skript, um die Tabelle `enrolls` zu erstellen.
- Diese Tabelle ist anders: Sie hat keinen Ersatz-Primärschlüssel!
- Sie referenziert die Tabellen `student` und `course` über entsprechende Fremdschlüssel (die wir nachher mit Einschränkungen durchsetzen).
- Diese beiden Fremdschlüssel bilden gleichzeitig den Primärschlüssel der Tabelle.
- Dadurch sind die Paare ihrer Wert automatisch `UNIQUE`.

```
1  -- object: public.enrolls | type: TABLE --
2  -- DROP TABLE IF EXISTS public.enrolls CASCADE;
3  CREATE TABLE public.enrolls (
4      student integer NOT NULL,
5      course integer NOT NULL,
6      CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
7  );
8  -- ddl-end --
9  ALTER TABLE public.enrolls OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
     ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle enrolls

- Diese Tabelle ist anders: Sie hat keinen Ersatz-Primärschlüssel!
- Sie referenziert die Tabellen `student` und `course` über entsprechende Fremdschlüssel (die wir nachher mit Einschränkungen durchsetzen).
- Diese beiden Fremdschlüssel bilden gleichzeitig den Primärschlüssel der Tabelle.
- Dadurch sind die Paare ihrer Wert automatisch `UNIQUE`.
- Dadurch kann sich kein Student mehr als einmal in einen Kurs einschreiben.

```
1  -- object: public.enrolls | type: TABLE --
2  -- DROP TABLE IF EXISTS public.enrolls CASCADE;
3  CREATE TABLE public.enrolls (
4      student integer NOT NULL,
5      course integer NOT NULL,
6      CONSTRAINT enrolls_pk PRIMARY KEY (student,course)
7  );
8  -- ddl-end --
9  ALTER TABLE public.enrolls OWNER TO postgres;
10 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
     ↪ ON_ERROR_STOP=1 -ebf 07_public_enrolls_table_5131.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Einschränkungen (1)

- Hier sehen wir das auto-generierte Skript, das die Einschränkung erstellt, die erzwingt, dass jede Zeile in Tabelle `course` mit genau einer Zeile in Tabelle `professor` verbunden ist.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
   ↪ ON_ERROR_STOP=1 -ebf 08
   ↪ _public_course_course_professor_fk_constraint_5116.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkungen (2)

- Hier sehen wir das auto-generierte Skript, das die Einschränkung erstellt, die erzwingt, dass jede Zeile in Tabelle `course` mit genau einer Zeile in Tabelle `module` verbunden ist.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 09
    ↪ _public_course_course_module_fk_constraint_5117.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkungen (3)

- Hier sehen wir das auto-generierte Skript, das die Einschränkung erstellt, die erzwingt, dass jede Zeile in Tabelle `enrolls` mit genau einer Zeile in Tabelle `student` verbunden ist.

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

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 10
    ↪ _public_enrolls_enrolls_student_fk_constraint_5145.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkungen (4)

- Hier sehen wir das auto-generierte Skript, das die Einschränkung erstellt, die erzwingt, dass jede Zeile in Tabelle `enrolls` mit genau einer Zeile in Tabelle `course` verbunden ist.

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

```
1 $ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 11
    ↪ _public_enrolls_enrolls_course_fk_constraint_5151.sql
2 ALTER TABLE
3 # psql 16.11 succeeded with exit code 0.
```

Daten Einfügen

- Nun schreiben wir ein eigenes SQL-Skript, um einige Daten in die Tabellen einzupflegen.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Nun schreiben wir ein eigenes SQL-Skript, um einige Daten in die Tabellen einzupflegen.
- Wir definieren die drei Studenten Bibbo, Bebbo und Bebba.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Nun schreiben wir ein eigenes SQL-Skript, um einige Daten in die Tabellen einzupflegen.
- Wir definieren die drei Studenten Bibbo, Bebbo und Bebba.
- Als Professoren nehmen wir Weise und Bobbo.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Nun schreiben wir ein eigenes SQL-Skript, um einige Daten in die Tabellen einzupflegen.
- Wir definieren die drei Studenten Bibbo, Bebbo und Bebba.
- Als Professoren nehmen wir Weise und Bobbo.
- Wir erstellen drei Module: *Python*, *Databases* und *Java*.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Nun schreiben wir ein eigenes SQL-Skript, um einige Daten in die Tabellen einzupflegen.
- Wir definieren die drei Studenten Bibbo, Bebbo und Bebba.
- Als Professoren nehmen wir Weise und Bobbo.
- Wir erstellen drei Module: *Python*, *Databases* und *Java*.
- Über die Tabelle `courses` spezifizieren wir, dass Prof. Weise teaches *Python* und *Databases* im Semester 20252 unterrichtet, also im Herbstsemester 2025.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Wir definieren die drei Studenten Bibbo, Bebbo und Bebba.
- Als Professoren nehmen wir Weise und Bobbo.
- Wir erstellen drei Module: *Python*, *Databases* und *Java*.
- Über die Tabelle `courses` spezifizieren wir, dass Prof. Weise teaches *Python* und *Databases* im Semester 20252 unterrichtet, also im Herbstsemester 2025.
- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Als Professoren nehmen wir Weise und Bobbo.
- Wir erstellen drei Module: *Python*, *Databases* und *Java*.
- Über die Tabelle `courses` spezifizieren wir, dass Prof. Weise teaches *Python* und *Databases* im Semester 20252 unterrichtet, also im Herbstsemester 2025.
- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.
- Herr Bibbo schreibt sich in den *Java*-Kurs von Prof. Bobbo ein und in beide Kurse von by Prof. Weise.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Wir erstellen drei Module: *Python*, *Databases* und *Java*.
- Über die Tabelle `courses` spezifizieren wir, dass Prof. Weise teaches *Python* und *Databases* im Semester 20252 unterrichtet, also im Herbstsemester 2025.
- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.
- Herr Bibbo schreibt sich in den *Java*-Kurs von Prof. Bobbo ein und in beide Kurse von by Prof. Weise.
- Herr Bebbo nimmt stattdessen *Java* und *Python*.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Über die Tabelle `courses` spezifizieren wir, dass Prof. Weise teaches *Python* und *Databases* im Semester 20252 unterrichtet, also im Herbstsemester 2025.
- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.
- Herr Bibbo schreibt sich in den *Java*-Kurs von Prof. Bobbo ein und in beide Kurse von by Prof. Weise.
- Herr Bebbo nimmt stattdessen *Java* und *Python*.
- Frau Bebba schreibt sich in *Java* und *Databases* ein.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.
- Herr Bibbo schreibt sich in den *Java*-Kurs von Prof. Bobbo ein und in beide Kurse von by Prof. Weise.
- Herr Bebbo nimmt stattdessen *Java* und *Python*.
- Frau Bebba schreibt sich in *Java* und *Databases* ein.
- Mit 4 `INNER JOIN`s können wir alle Daten wieder zusammensetzen.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Daten Einfügen

- Prof. Bobbo unterrichtet *Java* im Frühlings- und Herbstsemester 2026.
- Herr Bibbo schreibt sich in den *Java*-Kurs von Prof. Bobbo ein und in beide Kurse von by Prof. Weise.
- Herr Bebbo nimmt stattdessen *Java* und *Python*.
- Frau Bebba schreibt sich in *Java* und *Databases* ein.
- Mit 4 `INNER JOIN`s können wir alle Daten wieder zusammensetzen.
- Wir sortieren die Ergebnisse nach Studentenname, Professorname, Modultitel und Semester.

```sql
/** Insert data into the teaching database and then merge it. */

-- Insert several student records.
INSERT INTO student (name) VALUES ('Bibbo'), ('Bebbo'), ('Bebba');

-- Insert several professor records.
INSERT INTO professor (name) VALUES ('Weise'), ('Bobbo');

-- Insert several module records.
INSERT INTO module (title) VALUES ('Python'), ('Databases'), ('Java');

-- Create the courses, i.e., the instances of the modules in semesters.
INSERT INTO course (professor, module, semester) VALUES
    (1, 1, 20252), (1, 2, 20252), (2, 3, 20261), (2, 3, 20262);

-- Enroll students into the courses.
INSERT INTO enrolls (student, course) VALUES
    (1, 1), (1, 2), (1, 3), (2, 1), (2, 4), (3, 2), (3, 3);

-- Print the enrollment information.
SELECT student.name AS student, professor.name AS teacher,
       module.title AS module, semester FROM enrolls
    INNER JOIN student   ON enrolls.student   = student.id
    INNER JOIN course    ON enrolls.course    = course.id
    INNER JOIN professor ON course.professor = professor.id
    INNER JOIN module    ON course.module    = module.id
    ORDER BY student.name, professor.name, module.title, semester;
```

```
$ psql "postgres://postgres:XXX@localhost/teaching_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 3
INSERT 0 2
INSERT 0 3
INSERT 0 4
INSERT 0 7
 student | teacher |  module   | semester
---------+---------+-----------+----------
 Bebba   | Bobbo   | Java      |    20261
 Bebba   | Weise   | Databases |    20252
 Bebbo   | Bobbo   | Java      |    20262
 Bebbo   | Weise   | Python    |    20252
 Bibbo   | Bobbo   | Java      |    20261
 Bibbo   | Weise   | Databases |    20252
 Bibbo   | Weise   | Python    |    20252
(7 rows)

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

Aufräumen

- Räumen wir nun auf.

```sql
/* Cleanup after the example: Delete the teaching database. */

DROP DATABASE IF EXISTS teaching_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: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.
- Wir können uns hier aber nicht alle möglichen ternären Beziehungen anschauen.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.
- Wir können uns hier aber nicht alle möglichen ternären Beziehungen anschauen.
- Die Frage, wie wir ternäre Beziehungen oder Beziehungen noch höherer Ordnung implementieren ist also nicht einfach oder erschöpfend zu beantworten.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.
- Wir können uns hier aber nicht alle möglichen ternären Beziehungen anschauen.
- Die Frage, wie wir ternäre Beziehungen oder Beziehungen noch höherer Ordnung implementieren ist also nicht einfach oder erschöpfend zu beantworten.
- Man braucht einfach Erfahrung, um dafür ein Gefühl entwickeln zu können.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.
- Wir können uns hier aber nicht alle möglichen ternären Beziehungen anschauen.
- Die Frage, wie wir ternäre Beziehungen oder Beziehungen noch höherer Ordnung implementieren ist also nicht einfach oder erschöpfend zu beantworten.
- Man braucht einfach Erfahrung, um dafür ein Gefühl entwickeln zu können.
- So oder so wird man die Beziehungen oft auf mehrere binäre Beziehungen herunterbrechen.

Zusammenfassung: Beziehungen Höherer Ordnung

- Wir haben gesehen, dass wir eine konzeptuelle ternäre Beziehung auf verschiedene Arten im logischen Schema implementieren können.
- Diesmal waren es zwei verschiedene Arten, aber es hätten genauso gut auch mehr sein können.
- Wir können uns hier aber nicht alle möglichen ternären Beziehungen anschauen.
- Die Frage, wie wir ternäre Beziehungen oder Beziehungen noch höherer Ordnung implementieren ist also nicht einfach oder erschöpfend zu beantworten.
- Man braucht einfach Erfahrung, um dafür ein Gefühl entwickeln zu können.
- So oder so wird man die Beziehungen oft auf mehrere binäre Beziehungen herunterbrechen.
- Und wie man die implementiert wissen wir ja.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.
- Wir können Entitäten zu Tabellen machen.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.
- Wir können Entitäten zu Tabellen machen.
- Wir können Beziehungen implementieren.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.
- Wir können Entitäten zu Tabellen machen.
- Wir können Beziehungen implementieren.
- Wir kommen mit allen möglichen Randfällen wie Beziehungsattributen, schwachen Entitäten, und Beziehungen höherer Ordnung klar.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.
- Wir können Entitäten zu Tabellen machen.
- Wir können Beziehungen implementieren.
- Wir kommen mit allen möglichen Randfällen wie Beziehungsattributen, schwachen Entitäten, und Beziehungen höherer Ordnung klar.

Zusammenfassung

- An dieser Stelle haben wir eigentlich alles abgearbeitet, was man wissen muss, um konzeptuelle Schemas in logische Modelle zu transformieren.
- Wir können Entitäten zu Tabellen machen.
- Wir können Beziehungen implementieren.
- Wir kommen mit allen möglichen Randfällen wie Beziehungsattributen, schwachen Entitäten, und Beziehungen höherer Ordnung klar.
- Was uns noch fehlt sind ein paar grundlegende Richtlinien, wie man *gute* logische Modelle erstellt.

谢谢您们！

Thank you!

Vielen Dank!

Glossary (in English) I

C is a programming language, which is very successful in system programming situations[13,26].

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*[34].

DBMS A *database management system* is the software layer located between the user or application and the database (DB). The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[38].

ERD Entity relationship diagrams show the relationships between objects, e.g., between the tables in a DB and how they reference each other[2,4,6–8,19,27,36]

Java is another very successful programming language, with roots in the C family of languages[3,21].

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 database management system (DBMS)[14,24,25,33]. See https://postgresql.org for more information.

Python The Python programming language[17,20,22,35], i.e., what you will learn about in our book[35]. 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[9,15,16,28,32,34,37].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[5,10–12,18,23,29–32]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[29].