合肥大學
HEFEI UNIVERSITY

Datenbanken

38. Logisches Schema: Schwache Entitäten

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: 2025-12-25

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

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.
- Attribute können einwertig, mehrwertig, einfach, oder zusammengesetzt sein.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.
- Attribute können einwertig, mehrwertig, einfach, oder zusammengesetzt sein.
- Dann benutzen wir das relationale Datenmodell als logisches Modell für unsere Anwendung.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.
- Attribute können einwertig, mehrwertig, einfach, oder zusammengesetzt sein.
- Dann benutzen wir das relationale Datenmodell als logisches Modell für unsere Anwendung.
- Im Kern bietet das genau nur eine Modellierungskomponente an: Relationen.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.
- Attribute können einwertig, mehrwertig, einfach, oder zusammengesetzt sein.
- Dann benutzen wir das relationale Datenmodell als logisches Modell für unsere Anwendung.
- Im Kern bietet das genau nur eine Modellierungskomponente an: Relationen.
- Relationen haben Attribute, die immer einwertig und einfach sind.

Einleitung

- Wenn wir eine Anwendung bauen, dann fangen wir mit dem konzeptuellen Modell an.
- Dafür können wir ERDs verwenden.
- ERDs geben uns sehr viel Freiheit darin, wie wir die reale Welt modellieren.
- Wir können starke und schwache Entitäten verwenden oder Beziehungen – und alle drei dieser Elemente können Attribute haben.
- Attribute können einwertig, mehrwertig, einfach, oder zusammengesetzt sein.
- Dann benutzen wir das relationale Datenmodell als logisches Modell für unsere Anwendung.
- Im Kern bietet das genau nur eine Modellierungskomponente an: Relationen.
- Relationen haben Attribute, die immer einwertig und einfach sind.
- Tabellen sind praktische Implementierungen von Relationen in einem DBMS und Spalten repräsentieren Attribute.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.
- Beziehungen auf der konzeptuellen Ebene existieren nicht als Obkjete im relationalen Model.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.
- Beziehungen auf der konzeptuellen Ebene existieren nicht als Obkjete im relationalen Model.
- Stattdessen werden sie zu Tabellen und Einschränkungen.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.
- Beziehungen auf der konzeptuellen Ebene existieren nicht als Obkjete im relationalen Model.
- Stattdessen werden sie zu Tabellen und Einschränkungen.
- Mehrwertige Attribute von der konzeptuellen Ebene werden ebenso zu Tabellen in der logischen Ebene.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.
- Beziehungen auf der konzeptuellen Ebene existieren nicht als Obkjete im relationalen Model.
- Stattdessen werden sie zu Tabellen und Einschränkungen.
- Mehrwertige Attribute von der konzeptuellen Ebene werden ebenso zu Tabellen in der logischen Ebene.
- Zusammengesetzt Attribute werden auseinandergebrochen und ihre Komponenten werden zu Spalten.

Konzeptuelle und Logische Ebene

- Das relationale Datenmodell fühlt sich sehr natürlich an.
- Viele Komponenten des konzeptuellen Modells können direkt in das logische Modell übertragen werden.
- Z. B. Entitäten werden Tabellen.
- Beziehungen auf der konzeptuellen Ebene existieren nicht als Obkjete im relationalen Model.
- Stattdessen werden sie zu Tabellen und Einschränkungen.
- Mehrwertige Attribute von der konzeptuellen Ebene werden ebenso zu Tabellen in der logischen Ebene.
- Zusammengesetzt Attribute werden auseinandergebrochen und ihre Komponenten werden zu Spalten.
- Das haben wir alles schon besprochen.

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig.

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig:
  1. Wir habe starke Entitäten diskutiert, aber nicht schwache Entitäten (Einheit 30).

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig:
  1. Wir habe starke Entitäten diskutiert, aber nicht schwache Entitäten (Einheit 30).
  2. Wir haben Beziehungen diskutiert, aber nicht Beziehungsattribute.

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig:
  1. Wir habe starke Entitäten diskutiert, aber nicht schwache Entitäten (Einheit 30).
  2. Wir haben Beziehungen diskutiert, aber nicht Beziehungsattribute.
  3. Wir haben auch abgeleitete Attribute nicht diskutiert.

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig:
  1. Wir habe starke Entitäten diskutiert, aber nicht schwache Entitäten (Einheit 30).
  2. Wir haben Beziehungen diskutiert, aber nicht Beziehungsattribute.
  3. Wir haben auch abgeleitete Attribute nicht diskutiert.
  4. Und während wir alle binären Beziehungen komplette durchgespielt haben, haben wir Beziehungen höheren Grades (zwischen drei und mehr Entitäten) nicht besprochen.

Was noch fehlt

- Einige Objekte der konzeptuellen Ebene sind aber noch übrig:
  1. Wir habe starke Entitäten diskutiert, aber nicht schwache Entitäten (Einheit 30).
  2. Wir haben Beziehungen diskutiert, aber nicht Beziehungsattribute.
  3. Wir haben auch abgeleitete Attribute nicht diskutiert.
  4. Und während wir alle binären Beziehungen komplette durchgespielt haben, haben wir Beziehungen höheren Grades (zwischen drei und mehr Entitäten) nicht besprochen.
- Da müssen wir uns also noch drum kümmern.

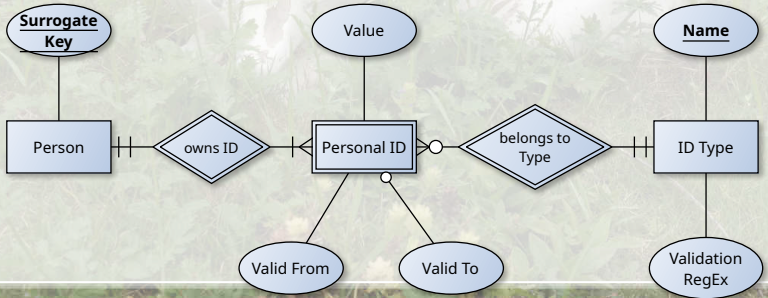
Schwache Entitäten

Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).

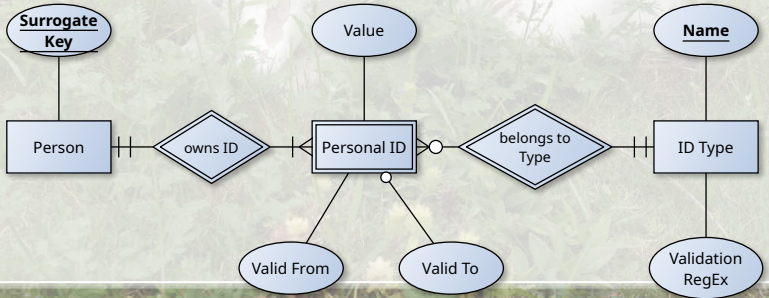
Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.



Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.



Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.
- Wenn wir solche identifizierenden Beziehungen nach SQL übersetzen, müssen wir diese mit zwingenden Enden gegenüberliegend vom schwachen Entitätstyp definieren.

Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.
- Wenn wir solche identifizierenden Beziehungen nach SQL übersetzen, müssen wir diese mit zwingenden Enden gegenüberliegend vom schwachen Entitätstyp definieren.
- Die Beziehung muss also so sein, dass jede schwache Entität mit den starken Entitäten verbunden sein muss, sonst darf sie nicht existieren.

Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.
- Wenn wir solche identifizierenden Beziehungen nach SQL übersetzen, müssen wir diese mit zwingenden Enden gegenüberliegend vom schwachen Entitätstyp definieren.
- Die Beziehung muss also so sein, dass jede schwache Entität mit den starken Entitäten verbunden sein muss, sonst darf sie nicht existieren.
- In unserem Fall müssten wir also zwei Beziehungen modellieren.

Schwache Entitäten zum Logischen Modell

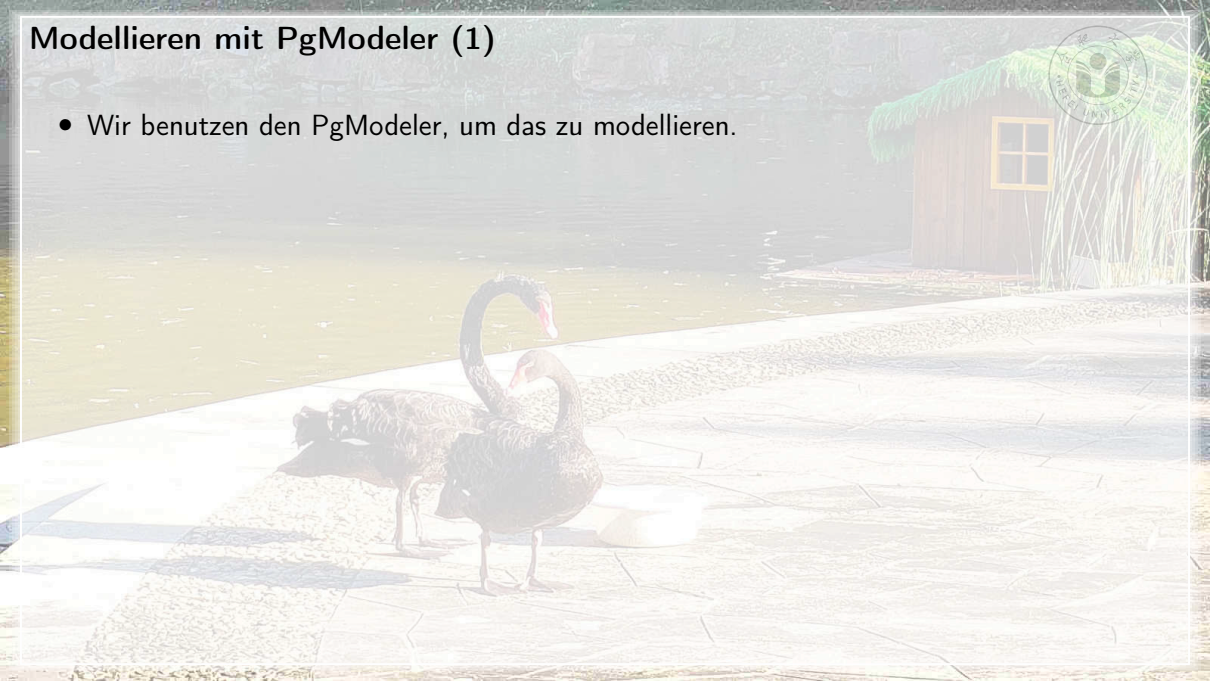
- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.
- Wenn wir solche identifizierenden Beziehungen nach SQL übersetzen, müssen wir diese mit zwingenden Enden gegenüberliegend vom schwachen Entitätstyp definieren.
- Die Beziehung muss also so sein, dass jede schwache Entität mit den starken Entitäten verbunden sein muss, sonst darf sie nicht existieren.
- In unserem Fall müssten wir also zwei Beziehungen modellieren:
  1. Person $\shortparallel\!\!-\!\!\dashv\!\!\prec$ Personal ID, die also auf dem M $\shortparallel\!\!-\!\!\dashv\!\!\prec$ N-Schema beruht.

Schwache Entitäten zum Logischen Modell

- Schwache Entitätstypen sind immer mit mindestens einer identifizierenden Beziehung mit einem starken Entitätstyp verbunden[56] (oder mit einem anderen schwachen Entitätstyp der identifiziert werden kann).
- Wir erinnern uns, dass wir damals die Beziehung von *Person*, *Personal ID* – einem schwachen Entitätstype – und *ID Type* so modelliert hatten.
- Jede *Personal ID* ist in einer identifizierenden Beziehung mit einer Entität vom Type *Person* und in einer identifizierenden Beziehung mit einer Entität vom Typ *ID Type*.
- Wenn wir solche identifizierenden Beziehungen nach SQL übersetzen, müssen wir diese mit zwingenden Enden gegenüberliegend vom schwachen Entitätstyp definieren.
- Die Beziehung muss also so sein, dass jede schwache Entität mit den starken Entitäten verbunden sein muss, sonst darf sie nicht existieren.
- In unserem Fall müssten wir also zwei Beziehungen modellieren:
  1. Person ‖—|< Personal ID, die also auf dem M ‖—|< N-Schema beruht und
  2. Personal ID >o—‖ ID Type, also, ID Type ‖—o< Personal ID, also nach Schema K ‖—o< L.

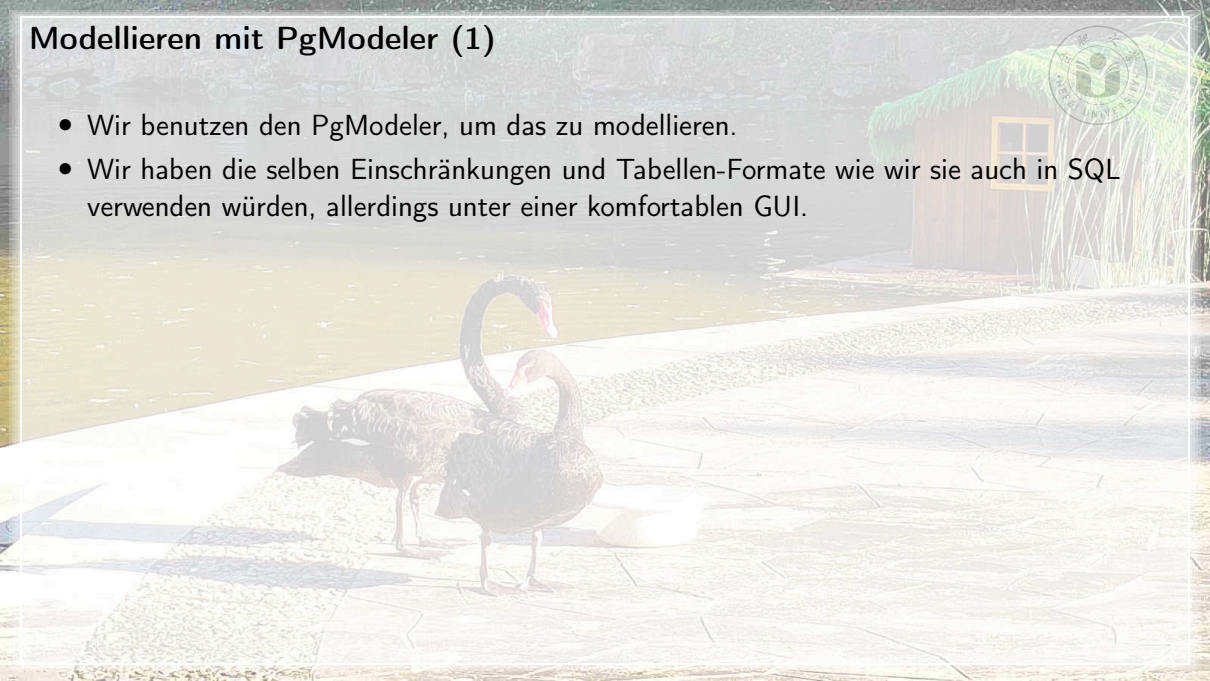
Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.



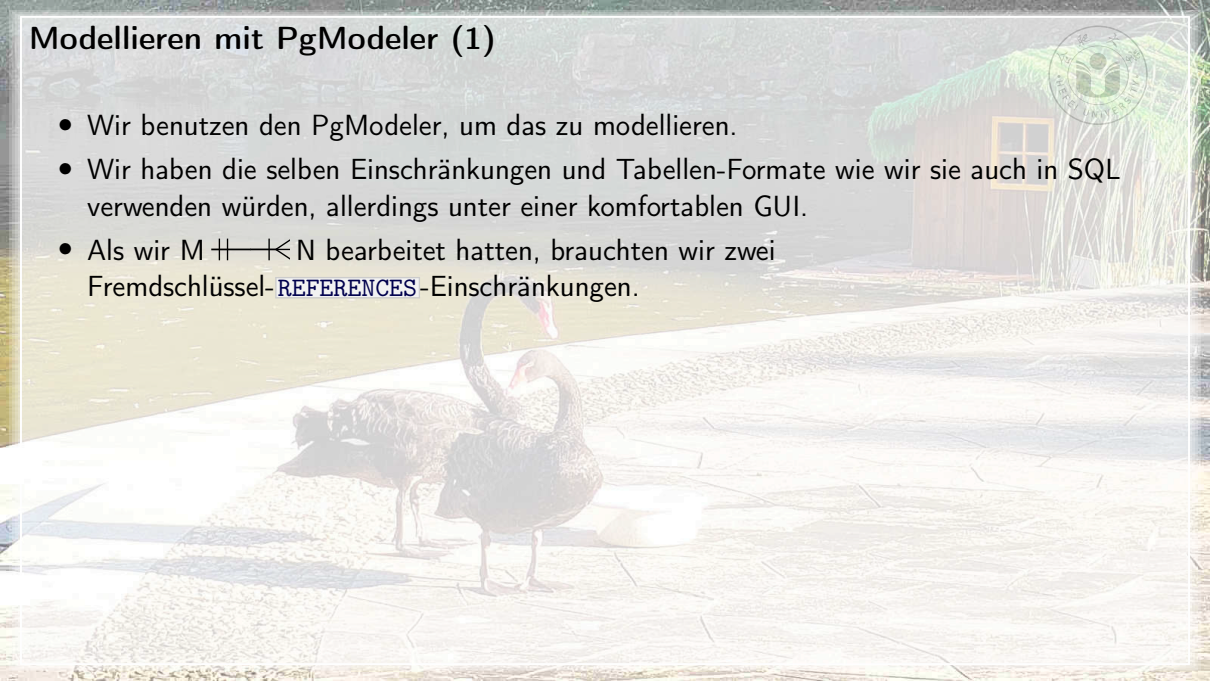
Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.



Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.
- Als wir M $\shortmid\!\!-\!\!\shortmid\!\!<$ N bearbeitet hatten, brauchten wir zwei Fremdschlüssel-`REFERENCES`-Einschränkungen.



Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.
- Als wir M ⫫⊣⋉ N bearbeitet hatten, brauchten wir zwei Fremdschlüssel-`REFERENCES`-Einschränkungen.
- Auf der einen Seite hatten wir Tabelle `M` von Tabelle `N` mit einem einzelenen Fremdschlüssel referenziert, der `NOT NULL` war.

Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.
- Als wir M ⫫⊣⋉ N bearbeitet hatten, brauchten wir zwei Fremdschlüssel-`REFERENCES`-Einschränkungen.
- Auf der einen Seite hatten wir Tabelle `M` von Tabelle `N` mit einem einzelenen Fremdschlüssel referenziert, der `NOT NULL` war.
- So haben wir sichergestellt, dass es immer mindestens eine Zeile in Tabelle `M` für jede Zeile in Tabelle `N` gibt.

Modellieren mit PgModeler (1)

- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.
- Als wir M ⫫⊣⋉ N bearbeitet hatten, brauchten wir zwei Fremdschlüssel-`REFERENCES`-Einschränkungen.
- Auf der einen Seite hatten wir Tabelle `M` von Tabelle `N` mit einem einzelenen Fremdschlüssel referenziert, der `NOT NULL` war.
- So haben wir sichergestellt, dass es immer mindestens eine Zeile in Tabelle `M` für jede Zeile in Tabelle `N` gibt.
- Auf der anderen Seite hatten wir Tabelle `N` von Tabelle `M` mit einem zusammengesetzten Fremdschlüssel referenziert.

Modellieren mit PgModeler (1)

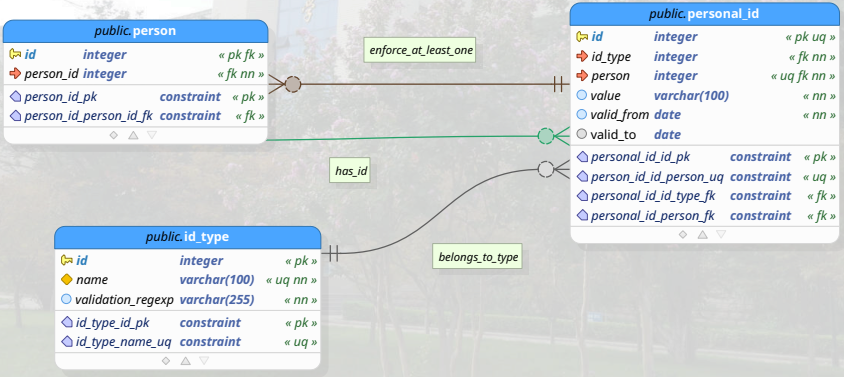
- Wir benutzen den PgModeler, um das zu modellieren.
- Wir haben die selben Einschränkungen und Tabellen-Formate wie wir sie auch in SQL verwenden würden, allerdings unter einer komfortablen GUI.
- Als wir M ⫫⊣⋉ N bearbeitet hatten, brauchten wir zwei Fremdschlüssel-`REFERENCES`-Einschränkungen.
- Auf der einen Seite hatten wir Tabelle `M` von Tabelle `N` mit einem einzelenen Fremdschlüssel referenziert, der `NOT NULL` war.
- So haben wir sichergestellt, dass es immer mindestens eine Zeile in Tabelle `M` für jede Zeile in Tabelle `N` gibt.
- Auf der anderen Seite hatten wir Tabelle `N` von Tabelle `M` mit einem zusammengesetzten Fremdschlüssel referenziert.
- Das hat erzwungen, dass jede Zeile in Tabelle `M` mit einer Zeile in Tabelle `N` verbunden war, ohne zu verhindern, dass es mehr Zeile in Tabelle `N` gibt, die mit ihr verbunden sind.

Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.

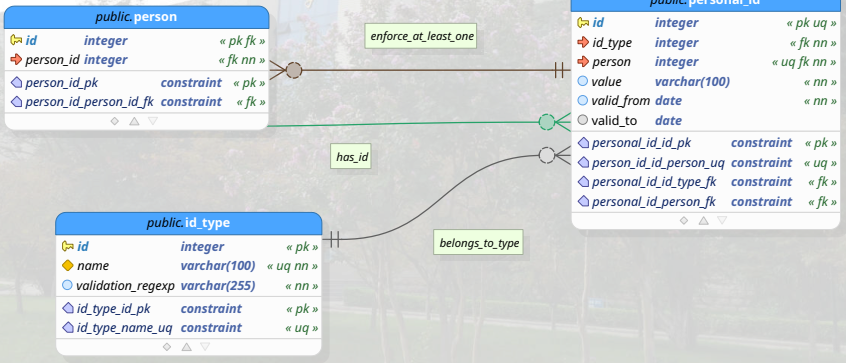
Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.



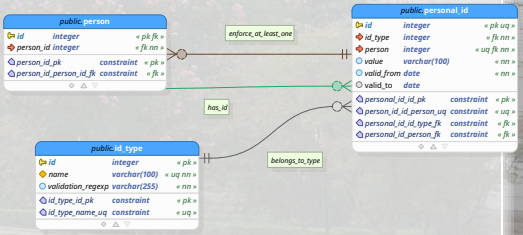
Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.



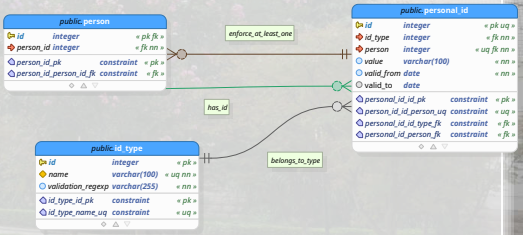
Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⊰ N Beziehungen in SQL dargestellt haben.



Modellieren mit PgModeler (2)

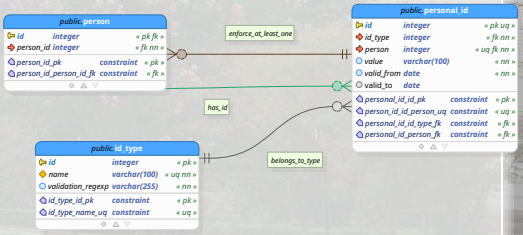
- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⫤ N Beziehungen in SQL dargestellt haben: als eine Kombination aus einer M ⫞○⫤ N und einer M ⫫○⫤ N Beziehung.



Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⫤ N Beziehungen in SQL dargestellt haben: als eine Kombination aus einer M ⫤○⫫ N und einer M ⫫⟜⫤ N Beziehung

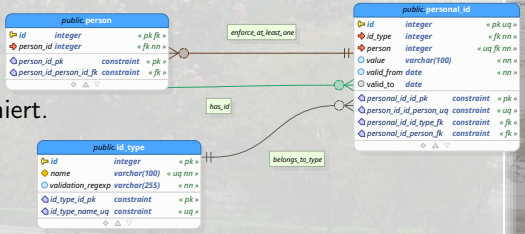
1. die braune `person` ⟜○⫫ `personal_id`-Beziehung *enforce_at_least_one* benutzt den zusammengesetzten Fremdschlüssel und erzwingt, dass für jede Zeile in Tabelle `person` eine verbundene Zeile in Tabelle `personal_id` existiert.



Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⫤ N Beziehungen in SQL dargestellt haben: als eine Kombination aus einer M ⟜⫫ N und einer M ⫫⟝ N Beziehung

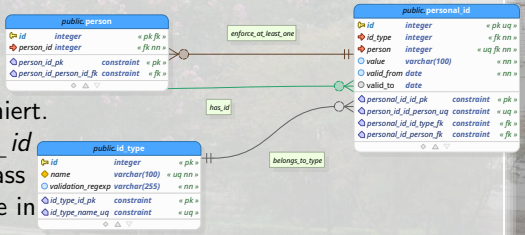
1. die braune `person` ⟜⫫ `personal_id`-Beziehung *enforce_at_least_one* benutzt den zusammengesetzten Fremdschlüssel und erzwingt, dass für jede Zeile in Tabelle `person` eine verbundene Zeile in Tabelle `personal_id` existiert. Sie ist auf Tabelle `person` definiert.



Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⫤ N Beziehungen in SQL dargestellt haben: als eine Kombination aus einer M ⫞○⫫ N und einer M ⫫⫤ N Beziehung

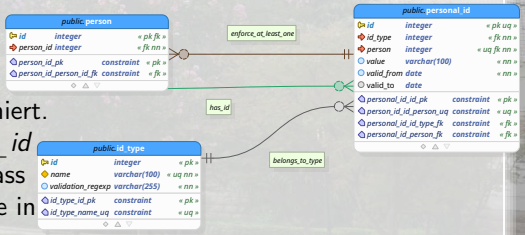
1. die braune `person` ⫸○⫫ `personal_id`-Beziehung *enforce_at_least_one* benutzt den zusammengesetzten Fremdschlüssel und erzwingt, dass für jede Zeile in Tabelle `person` eine verbundene Zeile in Tabelle `personal_id` existiert. Sie ist auf Tabelle `person` definiert.
2. Die grüne `person` ⫫⫤ `personal_id`-Beziehung *has_id* benutzt einen einfachen Fremdschlüssel und erzwingt, dass jede Zeile in Tabelle `personal_id` mit genau einer Zeile in Tabelle `person` verbunden ist.



Modellieren mit PgModeler (2)

- Bis auf ein Feature, was wir nach dieser Slide gleich diskutieren, wissen Sie schon alles, was sie Brauchen, um diese Situation mit dem PgModeler zu modellieren.
- Wir bekommen dieses Modell.
- Beachten Sie, dass die Einschränkungen separat angezeigt werden.
- Das gibt uns eine andere Perspektive darauf, wie wir die M ⫫⫤ N Beziehungen in SQL dargestellt haben: als eine Kombination aus einer M ⟜⫫ N und einer M ⫫⟜ N Beziehung

1. die braune `person` ⟜⫫ `personal_id`-Beziehung *enforce_at_least_one* benutzt den zusammengesetzten Fremdschlüssel und erzwingt, dass für jede Zeile in Tabelle `person` eine verbundene Zeile in Tabelle `personal_id` existiert. Sie ist auf Tabelle `person` definiert.
2. Die grüne `person` ⫫⟜ `personal_id`-Beziehung *has_id* benutzt einen einfachen Fremdschlüssel und erzwingt, dass jede Zeile in Tabelle `personal_id` mit genau einer Zeile in Tabelle `person` verbunden ist. Sie wird in Tabelle `personal_id` definiert.



Sequenz in PgModeler Erstellen

- Wir erinnern uns an die vorige Einheit.

Sequenz in PgModeler Erstellen

- Wir erinnern uns an die vorige Einheit.
- Für M ⫫⊰ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.

Sequenz in PgModeler Erstellen

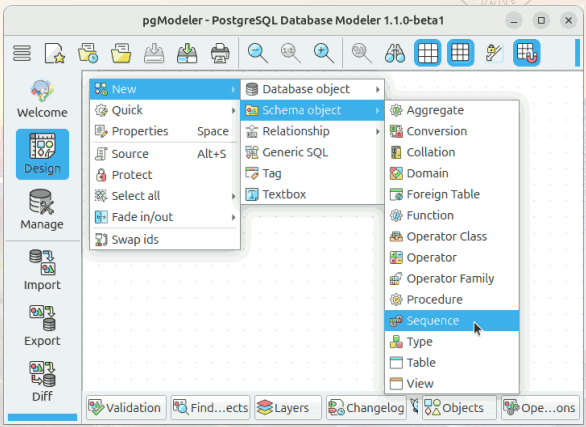
- Wir erinnern uns an die vorige Einheit.
- Für M ⫫⊣⋉ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.
- Das haben wir noch nicht in PgModeler gemacht.

Sequenz in PgModeler Erstellen

- Wir erinnern uns an die vorige Einheit.
- Für M ⫫⊰ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.
- Das haben wir noch nicht in PgModeler gemacht.
- Aber das geht auch da und das lernen wir jetzt.

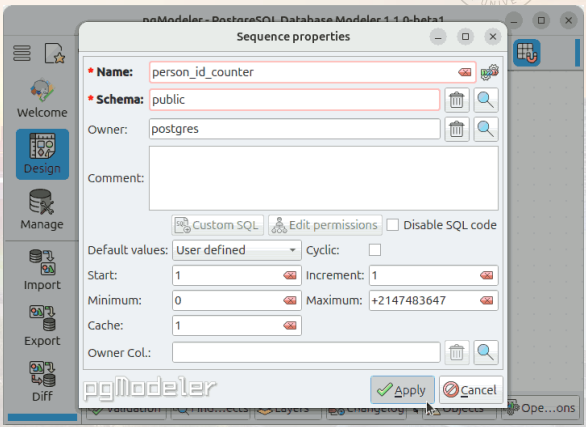
Sequenz in PgModeler Erstellen

- Wir erinnern uns an die vorige Einheit.
- Für M ⫫⊣⫤ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.
- Das haben wir noch nicht in PgModeler gemacht.
- Aber das geht auch da und das lernen wir jetzt.
- In der Designansicht im PgModeler rechts-klicken wir und wählen New › Schema Object › Sequence.



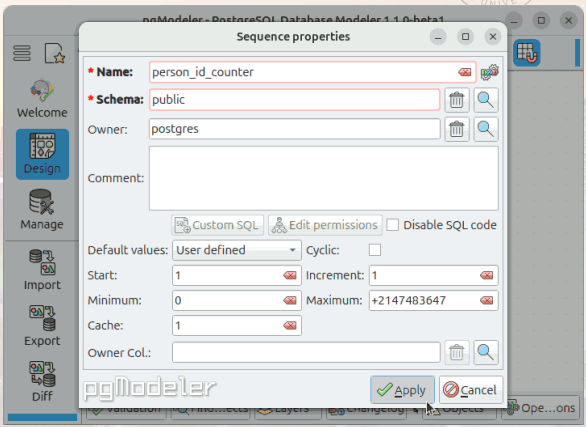
Sequenz in PgModeler Erstellen

- Wir erinnern uns an die vorige Einheit.
- Für M ⫫⊰ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.
- Das haben wir noch nicht in PgModeler gemacht.
- Aber das geht auch da und das lernen wir jetzt.
- In der Designansicht im PgModeler rechts-klicken wir und wählen New › Schema Object › Sequence.
- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.



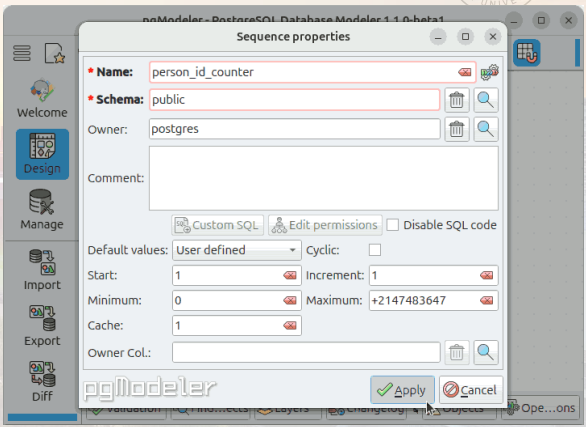
Sequenz in PgModeler Erstellen

- Für M ⫫⊰ N-Beziehungen mussten wir explizit eine Sequenz in PostgreSQL für die Primärschlüssel der Tabelle `m` erstellen.
- Das haben wir noch nicht in PgModeler gemacht.
- Aber das geht auch da und das lernen wir jetzt.
- In der Designansicht im PgModeler rechts-klicken wir und wählen New › Schema Object › Sequence.
- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.
- Wir wählen hier `person_id_counter`.



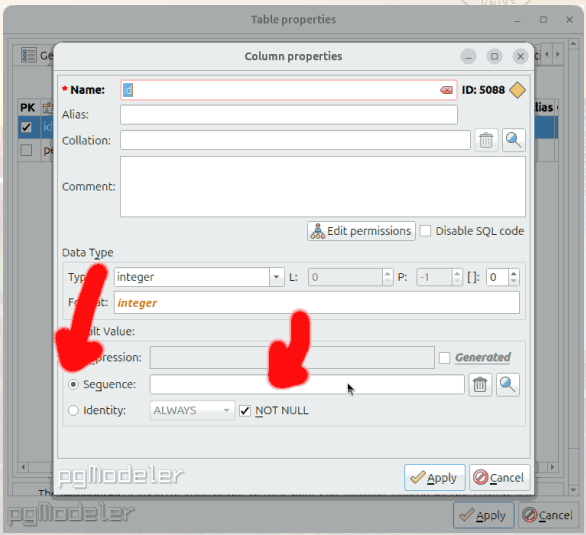
Sequenz in PgModeler Erstellen

- Das haben wir noch nicht in PgModeler gemacht.
- Aber das geht auch da und das lernen wir jetzt.
- In der Designansicht im PgModeler rechts-klicken wir und wählen New › Schema Object › Sequence.
- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.
- Wir wählen hier `person_id_counter`.
- Dann klicken wir auf Apply.



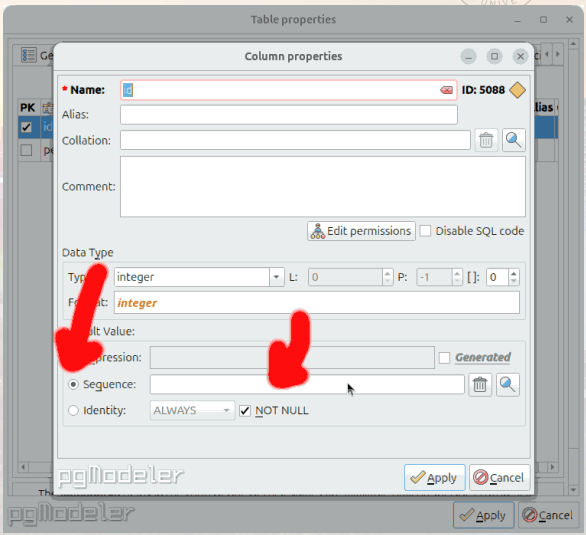
Sequenz in PgModeler Erstellen

- Aber das geht auch da und das lernen wir jetzt.
- In der Designansicht im PgModeler rechts-klicken wir und wählen New > Schema Object > Sequence.
- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.
- Wir wählen hier `person_id_counter`.
- Dann klicken wir auf Apply.
- Wir benutzen diese Sequenz später, wenn wir Spalten in eine Tabelle einfügen.



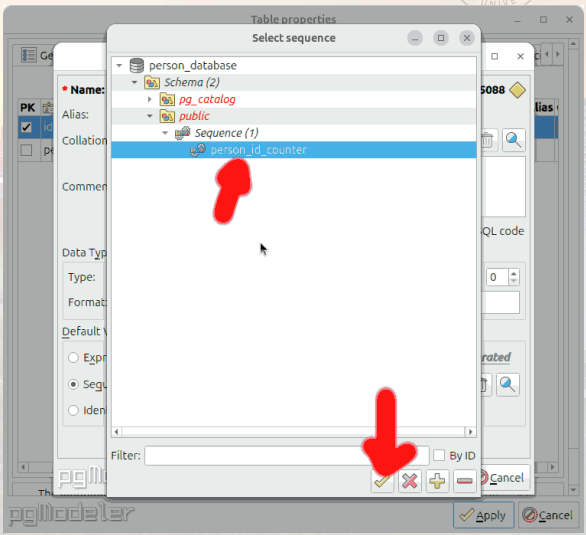
Sequenz in PgModeler Erstellen

- In der Designansicht im PgModeler rechts-klicken wir und wählen New > Schema Object > Sequence.
- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.
- Wir wählen hier `person_id_counter`.
- Dann klicken wir auf Apply.
- Wir benutzen diese Sequenz später, wenn wir Spalten in eine Tabelle einfügen.
- Dann wählen wir *Sequence* als Default Value für die Spalte und klicken in die Textbox rechts von *Sequence*.



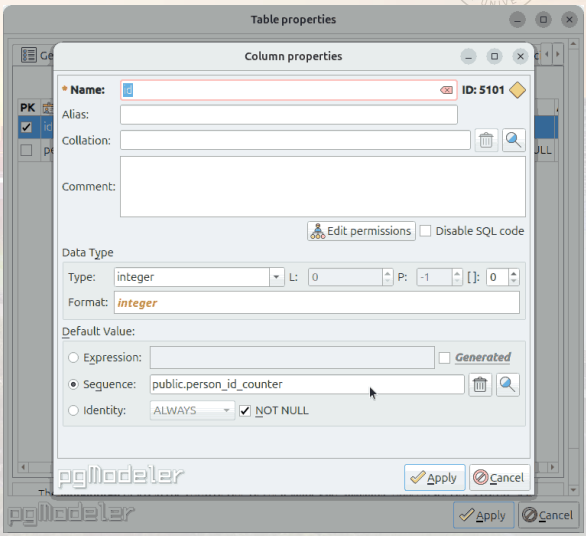
Sequenz in PgModeler Erstellen

- In dem Dialog, der sich öffnet, geben wir einen vernünftigen Namen für unsere Sequenz ein.
- Wir wählen hier `person_id_counter`.
- Dann klicken wir auf Apply.
- Wir benutzen diese Sequenz später, wenn wir Spalten in eine Tabelle einfügen.
- Dann wählen wir *Sequence* als Default Value für die Spalte und klicken in die Textbox rechts von *Sequence*.
- In dem Dialog, der sich öffnet, klicken wir durch den Objektbaum, finden unsere neue `Sequence` und klicken dann den Häkchen-Button unten.



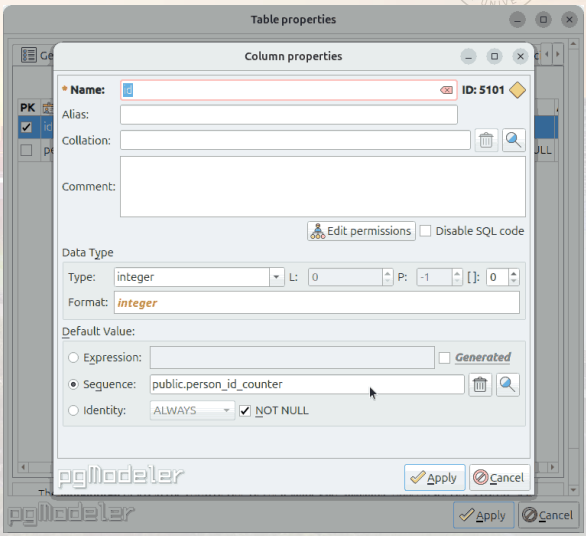
Sequenz in PgModeler Erstellen

- Wir wählen hier `person_id_counter`.
- Dann klicken wir auf Apply.
- Wir benutzen diese Sequenz später, wenn wir Spalten in eine Tabelle einfügen.
- Dann wählen wir *Sequence* als Default Value für die Spalte und klicken in die Textbox rechts von *Sequence*.
- In dem Dialog, der sich öffnet, klicken wir durch den Objektbaum, finden unsere neue `Sequence` und klicken dann den Häkchen-Button unten.
- Die neue Sequenz ist nun ausgewählt und die Spalte wird ihre Default-Werte von dort beziehen.



Sequenz in PgModeler Erstellen

- Wir benutzen diese Sequenz später, wenn wir Spalten in eine Tabelle einfügen.
- Dann wählen wir *Sequence* als `Default Value` für die Spalte und klicken in die Textbox rechts von *Sequence*.
- In dem Dialog, der sich öffnet, klicken wir durch den Objektbaum, finden unsere neue `Sequence` und klicken dann den Häkchen-Button unten.
- Die neue Sequenz ist nun ausgewählt und die Spalte wird ihre Default-Werte von dort beziehen.
- Damit können wir nun das Modell fertig malen.



Ausprobieren

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.
```

Sequenz person_id_counter

- Hier sehen wir das auto-generierte Skript, um die Sequenz zu erstellen.

```
-- object: public.person_id_counter | type: SEQUENCE --
-- DROP SEQUENCE IF EXISTS public.person_id_counter CASCADE;
CREATE SEQUENCE public.person_id_counter
    INCREMENT BY 1
    MINVALUE 0
    MAXVALUE 2147483647
    START WITH 1
    CACHE 1
    NO CYCLE
    OWNED BY NONE;

-- ddl-end --
ALTER SEQUENCE public.person_id_counter OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_person_id_counter_sequence_5071.sql
CREATE SEQUENCE
ALTER SEQUENCE
# psql 16.11 succeeded with exit code 0.
```

Sequenz person_id_counter

- Hier sehen wir das auto-generierte Skript, um die Sequenz zu erstellen.
- Wir werden diese Sequenz verwenden, um die Primärschlüsselwerte `id` für die Tabelle `person` zu generieren.

```
-- object: public.person_id_counter | type: SEQUENCE --
-- DROP SEQUENCE IF EXISTS public.person_id_counter CASCADE;
CREATE SEQUENCE public.person_id_counter
    INCREMENT BY 1
    MINVALUE 0
    MAXVALUE 2147483647
    START WITH 1
    CACHE 1
    NO CYCLE
    OWNED BY NONE;

-- ddl-end --
ALTER SEQUENCE public.person_id_counter OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 03_public_person_id_counter_sequence_5071.sql
CREATE SEQUENCE
ALTER SEQUENCE
# 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 04_public_id_type_table_5072.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.
- Die Tabelle hat den Primärschlüssel `id` und ein Attribut `name`, was `UNIQUE` sein muss.

```sql
-- 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 04_public_id_type_table_5072.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.
- Die Tabelle hat den Primärschlüssel `id` und ein Attribut `name`, was `UNIQUE` sein muss.
- Es hat auch eine Spalte `validation_regex`, in der wir einen regulären Ausdruck speichern, den eine Anwendung verwenden kann, um die entsprechenden ID-Werte zu prüfen.

```
-- 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 04_public_id_type_table_5072.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.
- Die Tabelle hat den Primärschlüssel `id` und ein Attribut `name`, was `UNIQUE` sein muss.
- Es hat auch eine Spalte `validation_regex`, in der wir einen regulären Ausdruck speichern, den eine Anwendung verwenden kann, um die entsprechenden ID-Werte zu prüfen.
- Diese Spalte muss `NOT NULL` sein und hat den Default-Wert `'.+'`, also einen regex für „mindestens ein Zeichen“.

```
-- 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 04_public_id_type_table_5072.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle id_type

- Die Tabelle hat den Primärschlüssel `id` und ein Attribut `name`, was `UNIQUE` sein muss.
- Es hat auch eine Spalte `validation_regex`, in der wir einen regulären Ausdruck speichern, den eine Anwendung verwenden kann, um die entsprechenden ID-Werte zu prüfen.
- Diese Spalte muss `NOT NULL` sein und hat den Default-Wert `'.+'`, also einen regex für „mindestens ein Zeichen“.
- Wenn der DBA keinen besseren regex angibt, dann werden Anwendungen nur verlangen, dass eine ID mindestens ein Zeichen hat.

```
-- 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 04_public_id_type_table_5072.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

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

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `personal_id` zu erstellen.
- In dieser Tabelle speichern wir die *Personal ID*-Entitäten.

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `personal_id` zu erstellen.
- In dieser Tabelle speichern wir die *Personal ID*-Entitäten.
- Wie alle unsere Tabellen hat sie den Ersatz-Primärschlüssel `id`.

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `personal_id` zu erstellen.
- In dieser Tabelle speichern wir die *Personal ID*-Entitäten.
- Wie alle unsere Tabellen hat sie den Ersatz-Primärschlüssel `id`.
- Die *Personal ID*-Entitäten sind schwache Entitäten.

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Hier sehen wir das auto-generierte Skript, um die Tabelle `personal_id` zu erstellen.
- In dieser Tabelle speichern wir die *Personal ID*-Entitäten.
- Wie alle unsere Tabellen hat sie den Ersatz-Primärschlüssel `id`.
- Die *Personal ID*-Entitäten sind schwache Entitäten.
- Sie können nicht ohne identifizierende Beziehungen zu einer *Person*- und zu einer *ID Type*-Entität existieren.

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- In dieser Tabelle speichern wir die *Personal ID*-Entitäten.
- Wie alle unsere Tabellen hat sie den Ersatz-Primärschlüssel `id`.
- Die *Personal ID*-Entitäten sind schwache Entitäten.
- Sie können nicht ohne identifizierende Beziehungen zu einer *Person*- und zu einer *ID Type*-Entität existieren.
- Deshalb speichert diese Tabelle zwei Fremdschlüssel: `id_type` und `person`.

```sql
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Wie alle unsere Tabellen hat sie den Ersatz-Primärschlüssel `id`.
- Die *Personal ID*-Entitäten sind schwache Entitäten.
- Sie können nicht ohne identifizierende Beziehungen zu einer *Person*- und zu einer *ID Type*-Entität existieren.
- Deshalb speichert diese Tabelle zwei Fremdschlüssel: `id_type` und `person`.
- Die entsprechenden Einschränkungen werden ein paar Slides weiter mit `ALTER TABLE` hinzugefügt.

```
-- object: public.personal_id | type: TABLE --
-- DROP TABLE IF EXISTS public.personal_id CASCADE;
CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
    CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
);
-- ddl-end --
ALTER TABLE public.personal_id OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Die *Personal ID*-Entitäten sind schwache Entitäten.
- Sie können nicht ohne identifizierende Beziehungen zu einer *Person*- und zu einer *ID Type*-Entität existieren.
- Deshalb speichert diese Tabelle zwei Fremdschlüssel: `id_type` und `person`.
- Die entsprechenden Einschränkungen werden ein paar Slides weiter mit `ALTER TABLE` hinzugefügt.
- Die Tabelle hat eine `UNIQUE`-Einschränkung auf die Tupel aus Ersatzschlüssel `id` und Fremdschlüssel `person`.

```
1  -- object: public.personal_id | type: TABLE --
2  -- DROP TABLE IF EXISTS public.personal_id CASCADE;
3  CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
11     CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
12 );
13 -- ddl-end --
14 ALTER TABLE public.personal_id OWNER TO postgres;
15 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/person_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
2  CREATE TABLE
3  ALTER TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Tabelle personal_id

- Sie können nicht ohne identifizierende Beziehungen zu einer *Person*- und zu einer *ID Type*-Entität existieren.
- Deshalb speichert diese Tabelle zwei Fremdschlüssel: `id_type` und `person`.
- Die entsprechenden Einschränkungen werden ein paar Slides weiter mit `ALTER TABLE` hinzugefügt.
- Die Tabelle hat eine `UNIQUE`-Einschränkung auf die Tupel aus Ersatzschlüssel `id` und Fremdschlüssel `person`.
- Wir brauchen das später, weil die Tabelle `person` diese Tupel als Fremdschlüssel verwenden wird.

```
1  -- object: public.personal_id | type: TABLE --
2  -- DROP TABLE IF EXISTS public.personal_id CASCADE;
3  CREATE TABLE public.personal_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 personal_id_id_pk PRIMARY KEY (id),
11     CONSTRAINT person_id_id_person_uq UNIQUE (id,person)
12 );
13 -- ddl-end --
14 ALTER TABLE public.personal_id OWNER TO postgres;
15 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/person_database" -v
       ↪ ON_ERROR_STOP=1 -ebf 05_public_personal_id_table_5078.sql
2  CREATE TABLE
3  ALTER TABLE
4  # 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 DEFAULT nextval('public.person_id_counter'::
        ↪ regclass),
    person_id integer 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 06_public_person_table_5087.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.
- In diesem Teil unseres Beispiels modellieren wir keine weiteren Informationen über *Person*-Entitäten.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL DEFAULT nextval('public.person_id_counter'::
        ↪ regclass),
    person_id integer 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 06_public_person_table_5087.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.
- In diesem Teil unseres Beispiels modellieren wir keine weiteren Informationen über *Person*-Entitäten.
- Deshalb hat diese Tabelle nur den Primärschlüssel `id`.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL DEFAULT nextval('public.person_id_counter'::
        ↪ regclass),
    person_id integer 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 06_public_person_table_5087.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.
- In diesem Teil unseres Beispiels modellieren wir keine weiteren Informationen über *Person*-Entitäten.
- Deshalb hat diese Tabelle nur den Primärschlüssel `id`.
- Wir definieren aber die Anforderung, dass unser System eine Form von ID für jeden Eintrag vom Typ *Person* speichern muss.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL DEFAULT nextval('public.person_id_counter'::
        ↪ regclass),
    person_id integer 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 06_public_person_table_5087.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.
- In diesem Teil unseres Beispiels modellieren wir keine weiteren Informationen über *Person*-Entitäten.
- Deshalb hat diese Tabelle nur den Primärschlüssel `id`.
- Wir definieren aber die Anforderung, dass unser System eine Form von ID für jeden Eintrag vom Typ *Person* speichern muss.
- Daher gibt es auch den Fremdschlüssel `person_id` auf Tabelle `personal_id` mit den schwachen Entitäten vom Typ *Personal ID*.

```
-- object: public.person | type: TABLE --
-- DROP TABLE IF EXISTS public.person CASCADE;
CREATE TABLE public.person (
    id integer NOT NULL DEFAULT nextval('public.person_id_counter'::
        ↪ regclass),
    person_id integer 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 06_public_person_table_5087.sql
CREATE TABLE
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkungen (1)

- Das auto-generierte Skript um die Einschränkung für die Beziehungen zwischen den Zeilen in den Tabellen `personal_id` und `id_type` zu erstellen.

```
-- object: personal_id_id_type_fk | type: CONSTRAINT --
-- ALTER TABLE public.personal_id DROP CONSTRAINT IF EXISTS
   ↪ personal_id_id_type_fk CASCADE;
ALTER TABLE public.personal_id ADD CONSTRAINT personal_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 07
   ↪ _public_personal_id_personal_id_id_type_fk_constraint_5093.sql
ALTER TABLE
# psql 16.11 succeeded with exit code 0.
```

Einschränkungen (2)

- Das erste auto-generierte Skript um die Einschränkung für die Beziehungen zwischen den Zeilen in den Tabellen `personal_id` und `person` zu erstellen: Jede Zeile in Tabelle `personal_id` ist verbunden mit genau einer Zeile in Tabelle `person`.

```
1 -- object: personal_id_person_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.personal_id DROP CONSTRAINT IF EXISTS
    ↪ personal_id_person_fk CASCADE;
3 ALTER TABLE public.personal_id ADD CONSTRAINT personal_id_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_personal_id_personal_id_person_fk_constraint_5094.sql
2 ALTER TABLE
3 # psql 16.11 succeeded with exit code 0.
```

Einschränkungen (3)

- Das zweite auto-generierte Skript um die Einschränkung für die Beziehungen zwischen den Zeilen in den Tabellen `personal_id` und `person` zu erstellen: Für jede Zeile in Tabelle `person` muss es mindestens eine zugehörige Zeile in Tabelle `personal_id` geben.

```
1 -- object: person_id_person_id_fk | type: CONSTRAINT --
2 -- ALTER TABLE public.person DROP CONSTRAINT IF EXISTS
    ↪ person_id_person_id_fk CASCADE;
3 ALTER TABLE public.person ADD CONSTRAINT person_id_person_id_fk FOREIGN
    ↪ KEY (person_id,id)
4 REFERENCES public.personal_id (id,person) 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_person_person_id_person_id_fk_constraint_5092.sql
2 ALTER TABLE
3 # psql 16.11 succeeded with exit code 0.
```

Tabellen Befüllen

- Nun fügen wir Daten ein.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Nun fügen wir Daten ein.
- Wir fangen damit an, dass wir zwei Zeilen in die Tabelle `id_type` einfügen: eine für chinesesische Ausweisnummern(中国公民身份号码) und eine für chinesische Mobiltelefonnummern.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Nun fügen wir Daten ein.
- Wir fangen damit an, dass wir zwei Zeilen in die Tabelle `id_type` einfügen: eine für chinesesische Ausweisnummern(中国公民身份号码) und eine für chinesische Mobiltelefonnummern.
- Wir speichern die selben regexes die wir schon in Einheit 36 verwendet hatten.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Nun fügen wir Daten ein.
- Wir fangen damit an, dass wir zwei Zeilen in die Tabelle `id_type` einfügen: eine für chinesesische Ausweisnummern(中国公民身份号码) und eine für chinesische Mobiltelefonnummern.
- Wir speichern die selben regexes die wir schon in Einheit 36 verwendet hatten.
- Natürlich arbeiten wir nur mit einem Teil des Systems.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Nun fügen wir Daten ein.
- Wir fangen damit an, dass wir zwei Zeilen in die Tabelle `id_type` einfügen: eine für chinesesische Ausweisnummern(中国公民身份号码) und eine für chinesische Mobiltelefonnummern.
- Wir speichern die selben regexes die wir schon in Einheit 36 verwendet hatten.
- Natürlich arbeiten wir nur mit einem Teil des Systems.
- Es gibt also keine Anwendung, die diese regexes wirklich verwendet. . .

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Wir fangen damit an, dass wir zwei Zeilen in die Tabelle `id_type` einfügen: eine für chinesesische Ausweisnummern(中国公民身份号码) und eine für chinesische Mobiltelefonnummern.
- Wir speichern die selben regexes die wir schon in Einheit 36 verwendet hatten.
- Natürlich arbeiten wir nur mit einem Teil des Systems.
- Es gibt also keine Anwendung, die diese regexes wirklich verwendet...
- Aber wir können es uns zumindest vorstellen...

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
    INSERT INTO person (id, person_id) SELECT person, id FROM
       ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
    INSERT INTO person (id, person_id) SELECT person, id FROM
       ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Wir speichern die selben regexes die wir schon in Einheit 36 verwendet hatten.
- Natürlich arbeiten wir nur mit einem Teil des Systems.
- Es gibt also keine Anwendung, die diese regexes wirklich verwendet...
- Aber wir können es uns zumindest vorstellen...
- Dann schreiben wir den selben Kode, den wir damals für das Einfügen von Daten nach dem M ⫫ N-Schema verwendet hatten. um die Tabellen `person` und `personal_id` zu befüllen.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Natürlich arbeiten wir nur mit einem Teil des Systems.
- Es gibt also keine Anwendung, die diese regexes wirklich verwendet. . .
- Aber wir können es uns zumindest vorstellen. . .
- Dann schreiben wir den selben Kode, den wir damals für das Einfügen von Daten nach dem M ⫫ N-Schema verwendet hatten. um die Tabellen `person` und `personal_id` zu befüllen.
- Zuerst erstellen wir einen `person`-Datensatz mit passender chinesischer Ausweisnummer.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Es gibt also keine Anwendung, die diese regexes wirklich verwendet. . .
- Aber wir können es uns zumindest vorstellen. . .
- Dann schreiben wir den selben Kode, den wir damals für das Einfügen von Daten nach dem M ⫫ N-Schema verwendet hatten. um die Tabellen `person` und `personal_id` zu befüllen.
- Zuerst erstellen wir einen `person`-Datensatz mit passender chinesischer Ausweisnummer.
- Dann hängen wir noch eine Mobiltelefonnummer an diesen Datensatz an.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
    INSERT INTO person (id, person_id) SELECT person, id FROM
       ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
    INSERT INTO person (id, person_id) SELECT person, id FROM
       ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Aber wir können es uns zumindest vorstellen...
- Dann schreiben wir den selben Kode, den wir damals für das Einfügen von Daten nach dem M ⫫ N-Schema verwendet hatten. um die Tabellen `person` und `personal_id` zu befüllen.
- Zuerst erstellen wir einen `person`-Datensatz mit passender chinesischer Ausweisnummer.
- Dann hängen wir noch eine Mobiltelefonnummer an diesen Datensatz an.
- Dann erstellen wir einen zweiten Datensatz, wieder mit chinesischer Ausweisnummer.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Zuerst erstellen wir einen `person`-Datensatz mit passender chinesischer Ausweisnummer.
- Dann hängen wir noch eine Mobiltelefonnummer an diesen Datensatz an.
- Dann erstellen wir einen zweiten Datensatz, wieder mit chinesischer Ausweisnummer.
- Zu guter Letzt kombinieren wir die Informationen wieder zusammen mit einem `INNER JOIN`.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
    ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Zuerst erstellen wir einen `person`-Datensatz mit passender chinesischer Ausweisnummer.
- Dann hängen wir noch eine Mobiltelefonnummer an diesen Datensatz an.
- Dann erstellen wir einen zweiten Datensatz, wieder mit chinesischer Ausweisnummer.
- Zu guter Letzt kombinieren wir die Informationen wieder zusammen mit einem `INNER JOIN`.
- Man kann jetzt fragen, ob es sich wirklich lohnt, sicherzustellen, dass jede *Person* immer mindestens eine passende *Personal ID*-Entität haben muss.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
    ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Dann hängen wir noch eine Mobiltelefonnummer an diesen Datensatz an.
- Dann erstellen wir einen zweiten Datensatz, wieder mit chinesischer Ausweisnummer.
- Zu guter Letzt kombinieren wir die Informationen wieder zusammen mit einem `INNER JOIN`.
- Man kann jetzt fragen, ob es sich wirklich lohnt, sicherzustellen, dass jede *Person* immer mindestens eine passende *Personal ID*-Entität haben muss.
- Das ist eine vernünftige Frage.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Dann erstellen wir einen zweiten Datensatz, wieder mit chinesischer Ausweisnummer.
- Zu guter Letzt kombinieren wir die Informationen wieder zusammen mit einem `INNER JOIN`.
- Man kann jetzt fragen, ob es sich wirklich lohnt, sicherzustellen, dass jede *Person* immer mindestens eine passende *Personal ID*-Entität haben muss.
- Das ist eine vernünftige Frage.
- Die Einschränkungen zu erstellen, damit unser logisches das konzeptuelle Modell voll repräsentiert erfordert ja viel Arbeit.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
    ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Zu guter Letzt kombinieren wir die Informationen wieder zusammen mit einem `INNER JOIN`.
- Man kann jetzt fragen, ob es sich wirklich lohnt, sicherzustellen, dass jede *Person* immer mindestens eine passende *Personal ID*-Entität haben muss.
- Das ist eine vernünftige Frage.
- Die Einschränkungen zu erstellen, damit unser logisches das konzeptuelle Modell voll repräsentiert erfordert ja viel Arbeit.
- Es zwingt uns z. B., die `SEQUENCE`- und `RETURNING`-Features von PostgreSQL zu verwenden, die nicht alle DBMSe unterstützen.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Das ist eine vernünftige Frage.
- Die Einschränkungen zu erstellen, damit unser logisches das konzeptuelle Modell voll repräsentiert erfordert ja viel Arbeit.
- Es zwingt uns z. B., die `SEQUENCE`- und `RETURNING`-Features von PostgreSQL zu verwenden, die nicht alle DBMSe unterstützen.
- Es könnte viel einfacher sein, stattdessen einfach eine Person ⫫—o≪ Personal ID-Beziehungsstruktur zu implementieren (anstatt des Person ⫫—⊣≪ Personal ID-Schemas).

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Die Einschränkungen zu erstellen, damit unser logisches das konzeptuelle Modell voll repräsentiert erfordert ja viel Arbeit.
- Es zwingt uns z. B., die `SEQUENCE`- und `RETURNING`-Features von PostgreSQL zu verwenden, die nicht alle DBMSe unterstützen.
- Es könnte viel einfacher sein, stattdessen einfach eine Person ||—o< Personal ID-Beziehungsstruktur zu implementieren (anstatt des Person ||—|< Personal ID-Schemas).
- Dadurch könnten wir viel einfaches SQL verwenden.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Es zwingt uns z. B., die `SEQUENCE`- und `RETURNING`-Features von PostgreSQL zu verwenden, die nicht alle DBMSe unterstützen.
- Es könnte viel einfacher sein, stattdessen einfach eine Person ||—o< Personal ID-Beziehungsstruktur zu implementieren (anstatt des Person ||—|< Personal ID-Schemas).
- Dadurch könnten wir viel einfaches SQL verwenden.
- Wir würden uns darauf verlassen, dass die Person, die die Daten eingibt, schon alles richtig macht.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Es könnte viel einfacher sein, stattdessen einfach eine Person ⫫—○≪ Personal ID-Beziehungsstruktur zu implementieren (anstatt des Person ⫫—|≪ Personal ID-Schemas).
- Dadurch könnten wir viel einfaches SQL verwenden.
- Wir würden uns darauf verlassen, dass die Person, die die Daten eingibt, schon alles richtig macht.
- Wir würden aber auch von unserem konzeptuellen Modell abweichen.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Dadurch könnten wir viel einfaches SQL verwenden.
- Wir würden uns darauf verlassen, dass die Person, die die Daten eingibt, schon alles richtig macht.
- Wir würden aber auch von unserem konzeptuellen Modell abweichen.
- Und wir würden von der Idee der „defense in depth“ abweichen, also der Idee, so oft und auf so vielen Ebenen wie möglich die Daten zu prüfen.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Dadurch könnten wir viel einfaches SQL verwenden.
- Wir würden uns darauf verlassen, dass die Person, die die Daten eingibt, schon alles richtig macht.
- Wir würden aber auch von unserem konzeptuellen Modell abweichen.
- Und wir würden von der Idee der „defense in depth“ abweichen, also der Idee, so oft und auf so vielen Ebenen wie möglich die Daten zu prüfen.
- Wir bekämen aber einfacheren Kode.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
            ↪ pers_id
            RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
    ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
    ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Wir würden uns darauf verlassen, dass die Person, die die Daten eingibt, schon alles richtig macht.
- Wir würden aber auch von unserem konzeptuellen Modell abweichen.
- Und wir würden von der Idee der „defense in depth“ abweichen, also der Idee, so oft und auf so vielen Ebenen wie möglich die Daten zu prüfen.
- Wir bekämen aber einfacheren Kode.
- Was ist besser?

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Wir würden aber auch von unserem konzeptuellen Modell abweichen.
- Und wir würden von der Idee der „defense in depth“ abweichen, also der Idee, so oft und auf so vielen Ebenen wie möglich die Daten zu prüfen.
- Wir bekämen aber einfacheren Kode.
- Was ist besser?
- Man kann diese Frage nicht einfach beantworten.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Und wir würden von der Idee der „defense in depth“ abweichen, also der Idee, so oft und auf so vielen Ebenen wie möglich die Daten zu prüfen.
- Wir bekämen aber einfacheren Kode.
- Was ist besser?
- Man kann diese Frage nicht einfach beantworten.
- Es hängt von der Situation ab.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
             id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Wir bekämen aber einfacheren Kode.
- Was ist besser?
- Man kann diese Frage nicht einfach beantworten.
- Es hängt von der Situation ab.
- Wir wählen hier die schwierigere Variante.

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
        (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Was ist besser?
- Man kann diese Frage nicht einfach beantworten.
- Es hängt von der Situation ab.
- Wir wählen hier die schwierigere Variante.
- Erstens wollen wir sehen, ob wir das hinbekommen (tun wir).

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(3 rows)

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

Tabellen Befüllen

- Man kann diese Frage nicht einfach beantworten.
- Es hängt von der Situation ab.
- Wir wählen hier die schwierigere Variante.
- Erstens wollen wir sehen, ob wir das hinbekommen (tun wir).
- Zweitens lernt man mehr . . . wir müssen schwierigeres SQL verstehen. . .

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

-- 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 a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456199501021234', '2024-12-01' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Insert a new personal ID for an existing person record.
INSERT INTO personal_id (id_type, person, value, valid_from) VALUES
       (2, 1, '1234567890', '2023-02-07');

-- Insert a new person record and a new ID record at the same time.
WITH pers_id AS (SELECT NEXTVAL('person_id_counter') AS person),
     new_pers_id AS (INSERT INTO personal_id (
            id_type, person, value, valid_from)
        SELECT 1, person, '123456200508071234', '2021-09-21' FROM
           ↪ pers_id
           RETURNING id, person)
     INSERT INTO person (id, person_id) SELECT person, id FROM
        ↪ new_pers_id;

-- Print the records that were inserted.
SELECT person, personal_id.id as pk, value, valid_from, name AS id_type
   ↪ FROM personal_id
    INNER JOIN id_type ON personal_id.id_type = id_type.id;
```

```
$ psql "postgres://postgres:XXX@localhost/person_database" -v
   ↪ ON_ERROR_STOP=1 -ebf insert_and_select.sql
INSERT 0 2
INSERT 0 1
INSERT 0 1
INSERT 0 1
 person | pk |       value        | valid_from |       id_type
--------+----+--------------------+------------+---------------------
      1 |  1 | 123456199501021234 | 2024-12-01 | national ID
      1 |  2 | 1234567890         | 2023-02-07 | mobile phone number
      2 |  3 | 123456200508071234 | 2021-09-21 | national ID
(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?
- Das schwache Entitäten im Grunde genauso wie starke Entitäten implementiert werden können.

Zusammenfassung

- Was haben wir also gelernt?
- Das schwache Entitäten im Grunde genauso wie starke Entitäten implementiert werden können.
- Der Hauptunterschied ist, dass die zugehörigen Enden ihrer identifizierenden Beziehungen zwingend seien müssen.

Zusammenfassung

- Was haben wir also gelernt?
- Das schwache Entitäten im Grunde genauso wie starke Entitäten implementiert werden können.
- Der Hauptunterschied ist, dass die zugehörigen Enden ihrer identifizierenden Beziehungen zwingend seien müssen.
- Was sie ja auch im konzeptuellen Modell sowieso schon seien müssten.

Zusammenfassung

- Was haben wir also gelernt?
- Das schwache Entitäten im Grunde genauso wie starke Entitäten implementiert werden können.
- Der Hauptunterschied ist, dass die zugehörigen Enden ihrer identifizierenden Beziehungen zwingend seien müssen.
- Was sie ja auch im konzeptuellen Modell sowieso schon seien müssten.
- Also haben wir gelernt, das es auf logischer Ebene im Grunde fast Wurscht ist, ob eine konzeptuelle Entität stark oder schwach ist.

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. 137).

[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. 128, 137).

[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. 128, 137).

[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. 136).

[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. 137, 138).

[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. 137).

[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. 138).

[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. 136).

[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. 137).

[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. 137).

[11] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 136).

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. 136).

[13] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 137, 138).

[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. 138).

[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. 136).

[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. 129, 136).

[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. 136).

[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. 138).

[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. 137).

References III

[20] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 138).

[21] 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. 138).

[22] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 138).

[23] 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. 138).

[24] 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. 137).

[25] 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. 137).

[26] 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. 137).

[27] 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. 137).

[28] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 137).

[29] 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. 137, 138).

[30] 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. 137).

References IV

[31] *IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX(TM))--Part 2: Shell and Utilities*. IEEE Std 1003.2-1992. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), 23. Juni 1993. URL: https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/POSIX/all.pdf (besucht am 2025-03-27). Board Approved: 1992-09-17, ANSI Approved: 1993-04-05. See unapproved draft IEEE P1003.2 Draft 11.2 of 9 1991 at the url (siehe S. 137).

[32] *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. 138).

[33] 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. 136).

[34] Andrew M. Kuchling. *Python 3 Documentation. Regular Expression HOWTO*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/howto/regex.html (besucht am 2024-11-01) (siehe S. 137).

[35] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 138).

[36] 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. 137).

[37] 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. 136).

[38] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 137).

[39] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 137).

References V

[40] 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. 138).

[41] Zsolt Nagy. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. New York, NY, USA: Apress Media, LLC, Aug. 2018. ISBN: 978-1-4842-3876-9 (siehe S. 137).

[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. 136).

[43] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., Juni 2019. ISBN: 978-1-4920-8255-2 (siehe S. 137).

[44] 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. 137).

[45] 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. 136).

[46] "POSIX Regular Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.7.3. URL: https://www.postgresql.org/docs/17/functions-matching.html#FUNCTIONS-POSIX-REGEXP (besucht am 2025-02-27) (siehe S. 137).

[47] *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).

[48] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 137).

[49] 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. 136).

[50] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 137).

References VI

[51] "`re` – Regular Expression Operations". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/re.html#module-re (besucht am 2024-11-01) (siehe S. 137).

[52] 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. 137).

[53] 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. 136).

[54] 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. 136).

[55] Yuriy Shamshin. *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. URL: https://dbs.academy.lv (besucht am 2025-01-11).

[56] Yuriy Shamshin. "Mapping ER Diagrams to Relation Data Model". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 06. URL: https://dbs.academy.lv/lection/dbs_LS06EN_er2rm.pdf (besucht am 2025-04-20) (siehe S. 29–36).

[57] 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. 137).

[58] 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. 137).

[59] "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. 138).

References VII

[60] 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. 134, 138).

[61] 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[60] (siehe S. 138).

[62] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 137, 138).

[63] 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. 137).

[64] 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. 137).

[65] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 137).

[66] 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. 136, 137).

[67] 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. 137).

[68] 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. 136).

References VIII

[69] *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. 137).

[70] 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. 137).

[71] 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. 136).

[72] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 136).

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[11,42,72]. 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[8,37,45,49,53]. 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*[66].

DBA A *database administrator* is the person or group responsible for the effective use of database technology in an organization or enterprise.

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[71].

ERD Entity relationship diagrams show the relationships between objects, e.g., between the tables in a DB and how they reference each other[4,12,15–17,33,54,68]

GUI graphical user interface

IT information technology

Glossary (in English) II

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[13,29].

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[5,28,57,64,65]. 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[2,3,6,24,39,50]. 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[9,24,52,63,70]. 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[25,44,48,63]. See https://postgresql.org for more information.

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

Python The Python programming language[30,36,38,67], i.e., what you will learn about in our book[67]. Learn more at https://python.org.

regex A *Regular Expression*, often called „regex“ for short, is a sequence of characters that defines a search pattern for text strings[31,34,41,43]. In Python, the `re` module offers functionality work with regular expressions[34,51]. In PostgreSQL, regex-based pattern matching is supported as well[46].

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[19,26,27,58,62,66,69].

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“[35].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[14,20–22,32,40,59–62]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[59].

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.

Ubuntu is a variant of the open source operating system Linux[18,29]. 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[7,23]