



合肥大學
HEFEI UNIVERSITY



Datenbanken

37. Logisches Schema: Beziehungen

Thomas Weise (汤卫思)
twaise@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

Databases



Dies ist ein Kurs über Datenbanken an der Universität Hefei (合肥大学).

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



Outline



1. Einleitung

2. $A \vdash \text{---} \text{---} \vdash B$

3. $C \vdash \text{---} \text{---} \parallel D$

4. $E \vdash \text{---} \text{---} \times F$

5. $G \vdash \text{---} \text{---} \leq H$

6. Zusammenfassung





Einleitung



Einleitung



- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.

Einleitung



- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.
- Damals haben wir unseren Weg durch alle diese Typen gearbeitet und Beispiele von existierenden Quellen gefunden.

Einleitung



- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.
- Damals haben wir unseren Weg durch alle diese Typen gearbeitet und Beispiele von existierenden Quellen gefunden.
- Wir können binäre Beziehungen als Anforderungen verstehen, die auf zwei Entitätstypen definiert wurden.

- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.
- Damals haben wir unseren Weg durch alle diese Typen gearbeitet und Beispiele von existierenden Quellen gefunden.
- Wir können binäre Beziehungen als Anforderungen verstehen, die auf zwei Entitätstypen definiert wurden.
- In einer $C \text{ } \vdash \bigcirc \text{ } \dashv \vdash D$ -Beziehung, z. B., ist es erforderlich, dass jede Entität von Typ C immer mit genau einer Entität vom Typ D verbunden sein *muss*.

- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.
- Damals haben wir unseren Weg durch alle diese Typen gearbeitet und Beispiele von existierenden Quellen gefunden.
- Wir können binäre Beziehungen als Anforderungen verstehen, die auf zwei Entitätstypen definiert wurden.
- In einer $C \text{ } \vdash \bigcirc \text{ } \dashv \vdash D$ -Beziehung, z. B., ist es erforderlich, dass jede Entität von Typ C immer mit genau einer Entität vom Typ D verbunden sein *muss*.
- Beziehungen sind bi-direktional, also wenn eine Entität vom Typ C mit einer Entität vom Typ D verbunden ist, dann gilt das auch andersherum.

- In Einheit 31 haben wir die verschiedenen Arten von binären Beziehungen zwischen zwei Entitätstypen diskutiert, die in einem ERD auftreten können.
- Damals haben wir unseren Weg durch alle diese Typen gearbeitet und Beispiele von existierenden Quellen gefunden.
- Wir können binäre Beziehungen als Anforderungen verstehen, die auf zwei Entitätstypen definiert wurden.
- In einer $C \text{ } \oplus \text{---} \text{ } \parallel D$ -Beziehung, z. B., ist es erforderlich, dass jede Entität von Typ C immer mit genau einer Entität vom Typ D verbunden sein *muss*.
- Beziehungen sind bi-direktional, also wenn eine Entität vom Typ C mit einer Entität vom Typ D verbunden ist, dann gilt das auch andersherum.
- Das Beziehungsmuster $C \text{ } \oplus \text{---} \text{ } \parallel D$ erlaubt, dass eine Entität vom Type D entweder mit einer oder keiner Entität vom Typ C verbunden ist.



Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.



Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.

Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.
- SQL bietet uns die folgenden Werkzeuge, um die Beziehungen korrekt darzustellen.

Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.
- SQL bietet uns die folgenden Werkzeuge, um die Beziehungen korrekt darzustellen
 - die Primärschlüssel-Einschränkung **PRIMARY KEY**.



Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.
- SQL bietet uns die folgenden Werkzeuge, um die Beziehungen korrekt darzustellen
 - die Primärschlüssel-Einschränkung **PRIMARY KEY**,
 - die Fremdschlüssel-Einschränkung **REFERENCES**.

Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.
- SQL bietet uns die folgenden Werkzeuge, um die Beziehungen korrekt darzustellen
 - die Primärschlüssel-Einschränkung **PRIMARY KEY**,
 - die Fremdschlüssel-Einschränkung **REFERENCES**,
 - die **NOT NULL**-Einschränkung, die verhindert, dass ein Attribut undefiniert (**NULL**) bleibt.

Definition: Referentielle Integrität

Eine Datenbank wo die Beziehungseinschränkungen korrekt implementiert sind und durchgesetzt werden hat die Eigenschaft *referentielle Integrität* (EN: *referential integrity*).

- Wenn wir ein konzeptuelles Modell in das relationale Datenmodell übersetzen, dann müssen wir auch die Beziehungsmuster mit übersetzen.
- Das bedeutet, dass wir im Grunde die Krähenfußnotation nach SQL übersetzen müssen.
- SQL bietet uns die folgenden Werkzeuge, um die Beziehungen korrekt darzustellen
 - die Primärschlüssel-Einschränkung **PRIMARY KEY**,
 - die Fremdschlüssel-Einschränkung **REFERENCES**,
 - die **NOT NULL**-Einschränkung, die verhindert, dass ein Attribut undefiniert (**NULL**) bleibt und
 - die **UNIQUE**-Einschränkung, die verhindert, dass ein Wert mehrmals in einer Spalte vorkommt.

Was jetzt kommt

- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.



Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.
- Betrachten Sie das gleichzeitig als eine Übung in SQL.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.
- Betrachten Sie das gleichzeitig als eine Übung in SQL.
- Es geht nicht darum, sich genau zu merken, wie welcher Beziehungstyp dargestellt wird.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.
- Betrachten Sie das gleichzeitig als eine Übung in SQL.
- Es geht nicht darum, sich genau zu merken, wie welcher Beziehungstyp dargestellt wird.
- Es geht mehr darum, die Werkzeuge, die SQL uns bietet, besser zu verstehen.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.
- Betrachten Sie das gleichzeitig als eine Übung in SQL.
- Es geht nicht darum, sich genau zu merken, wie welcher Beziehungstyp dargestellt wird.
- Es geht mehr darum, die Werkzeuge, die SQL uns bietet, besser zu verstehen.
- Wir gucken uns genau an, wie `NOT NULL`, `REFERENCES`, `UNIQUE` und `PRIMARY KEY`²² zusammen mit `INNER JOIN`³⁹ verwendet werden, um die referentielle Integrität von Tabellen sicherzustellen.

Was jetzt kommt



- Wir werden jetzt schauen, wie *jeder* der zehn binären Beziehungstypen in SQL implementiert werden kann.
- Wir machen das direkt in SQL und benutzen den PgModeler nicht, denn wir fangen ja schon mit einer visuellen Notation als Ausgangspunkt an und wollen diese nach SQL übersetzen.
- Der PgModeler bietet hier keinen zusätzlichen Nutzen. Er ist für größere Modelle gedacht, wohingegen wir uns hier durch viele kleine Modelle durchkämpfen.
- Betrachten Sie das gleichzeitig als eine Übung in SQL.
- Es geht nicht darum, sich genau zu merken, wie welcher Beziehungstyp dargestellt wird.
- Es geht mehr darum, die Werkzeuge, die SQL uns bietet, besser zu verstehen.
- Wir gucken uns genau an, wie `NOT NULL`, `REFERENCES`, `UNIQUE` und `PRIMARY KEY`²² zusammen mit `INNER JOIN`³⁹ verwendet werden, um die referentielle Integrität von Tabellen sicherzustellen.
- Für manche komplizierten Beziehungen macht es auch Spaß, auszuknobeln, wie man sie implementieren könnte.

In Aktion

- Natürlich schreiben wir nicht nur SQL irgendwie hin ... wir probieren es aus!



In Aktion



- Natürlich schreiben wir nicht nur SQL irgendwie hin ... wir probieren es aus!
- Dazu erstellen wir eine neue Datenbank.

```
1  /* Initialize the database for our examples */  
2  
3  -- Create the database.  
4  CREATE DATABASE relationships;
```

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



A $\vdash \circ - \circ \vdash$ B



A $\oplus$ $\ominus$ $\oplus$ B

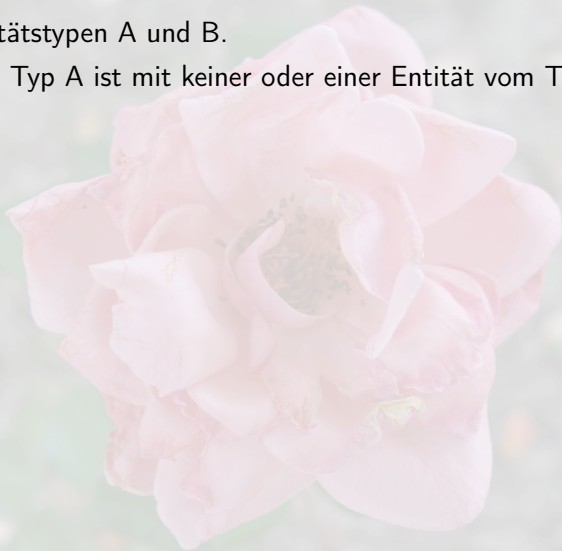
- Es gibt zwei Entitätstypen A und B.



A $\oplus$ --- $\oplus$ B



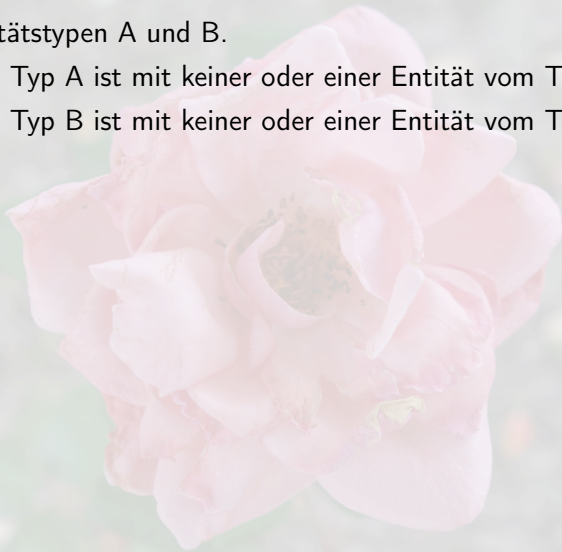
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.



A $\oplus$ --- $\oplus$ B



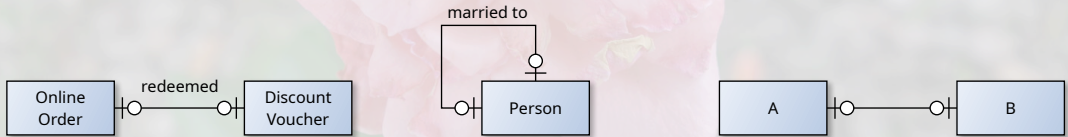
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.



A  B



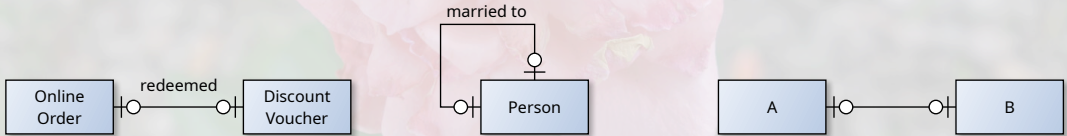
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:



A   B



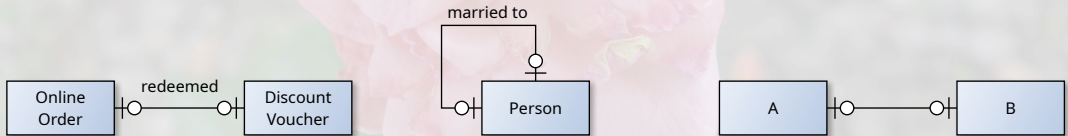
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können.



A B



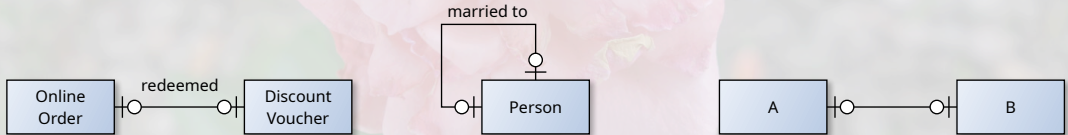
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus, die die Bestellkosten um 20% reduzieren.



A B



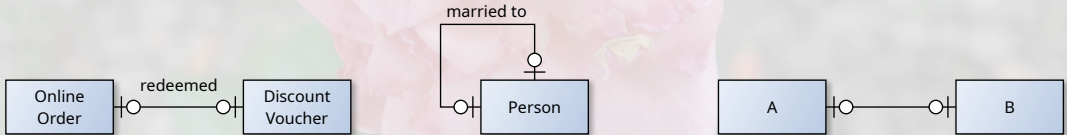
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus, die die Bestellkosten um 20% reduzieren. Jeder Gutschein kann für genau eine Bestellung eingelöst werden (oder auch gar nicht).



A B



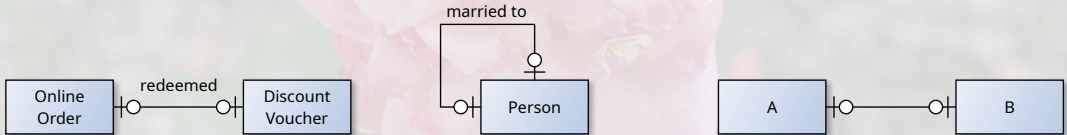
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus, die die Bestellkosten um 20% reduzieren. Jeder Gutschein kann für genau eine Bestellung eingelöst werden (oder auch gar nicht). Für jede Bestellung kann maximal ein Gutschein eingelöst werden (oder auch keiner).



A B



- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus, die die Bestellkosten um 20% reduzieren. Jeder Gutschein kann für genau eine Bestellung eingelöst werden (oder auch gar nicht). Für jede Bestellung kann maximal ein Gutschein eingelöst werden (oder auch keiner).
 - Jede Person kann mit keiner oder genau einer (anderen) Person verheiratet sein.



Zwei Realisierungen

- Es gibt zwei Möglichkeiten, das zu realisieren.



Zwei Realisierungen

- Es gibt zwei Möglichkeiten, das zu realisieren:
 1. Wir können eine Lösung mit drei Tabellen bauen.



Zwei Realisierungen

- Es gibt zwei Möglichkeiten, das zu realisieren:
 1. Wir können eine Lösung mit drei Tabellen bauen.
 - Jeder Entitätstyp bekommt eine eigene Tabelle.



Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:
 1. Wir können eine Lösung mit drei Tabellen bauen.
 - Jeder Entitätstyp bekommt eine eigene Tabelle.
 - Die Beziehungen werden in einer dritten Tabelle gemanaged.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier **aid** sein.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle `a` steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier `aid` sein. Wir geben ihr eine weitere Spalte `x`, in der wir einen Text aus drei Zeichen speichern.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier **aid** sein. Wir geben ihr eine weitere Spalte **x**, in der wir einen Text aus drei Zeichen speichern.
2. Tabelle **b** steht für Entitätstyp B.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier **aid** sein. Wir geben ihr eine weitere Spalte **x**, in der wir einen Text aus drei Zeichen speichern.
2. Tabelle **b** steht für Entitätstyp B. Ihr Ersatz-Primärschlüssel soll hier **bid** sein.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.
 - Jeder Entitätstyp bekommt eine eigene Tabelle.
 - Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier **aid** sein. Wir geben ihr eine weitere Spalte **x**, in der wir einen Text aus drei Zeichen speichern.
2. Tabelle **b** steht für Entitätstyp B. Ihr Ersatz-Primärschlüssel soll hier **bid** sein. Wir geben ihr eine weitere Spalte **y**, in der wir einen Text aus zwei Zeichen speichern.

Zwei Realisierungen



- Es gibt zwei Möglichkeiten, das zu realisieren:

1. Wir können eine Lösung mit drei Tabellen bauen.

- Jeder Entitätstyp bekommt eine eigene Tabelle.
- Die Beziehungen werden in einer dritten Tabelle gemanaged.

Diese Lösung ist sehr sinnvoll, wenn es nicht so viele verbundene Paare von A und B Entitäten gibt.

2. Wie können eine Lösung mit zwei Tabellen bauen, wo die Beziehung in einer zusätzlichen Spalte in einer der Tabellen gemanaged wird. Diese Lösung ist besser, wenn viele A und B-Paare verbunden sind.

- So oder so, wir brauchen mindestens die folgenden beiden Tabellen:

1. Tabelle **a** steht für Entitätstyp A. Ihr Ersatz-Primärschlüssel soll hier **aid** sein. Wir geben ihr eine weitere Spalte **x**, in der wir einen Text aus drei Zeichen speichern.
2. Tabelle **b** steht für Entitätstyp B. Ihr Ersatz-Primärschlüssel soll hier **bid** sein. Wir geben ihr eine weitere Spalte **y**, in der wir einen Text aus zwei Zeichen speichern.

- In den folgenden Beispielen, wir verwenden immer so eine Struktur.

Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁰.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁰.
- Wir brauchen eine dritte Tabelle, relate_a_and_b, die die Entitäten der Typen A und B in Verbindung setzt.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE           REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁰.
- Wir brauchen eine dritte Tabelle, relate_a_and_b, die die Entitäten der Typen A und B in Verbindung setzt.
- Sie hat zwei Spalten: fkaid speichert den Primärschlüssel der A-Entitäten als Fremdschlüssel und fkbid speichert die Primärschlüssel der B-Entitäten als Fremdschlüssel.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁰.
- Wir brauchen eine dritte Tabelle, relate_a_and_b, die die Entitäten der Typen A und B in Verbindung setzt.
- Sie hat zwei Spalten: fkaid speichert den Primärschlüssel der A-Entitäten als Fremdschlüssel und fkbid speichert die Primärschlüssel der B-Entitäten als Fremdschlüssel.
- Wir speichern nur existierende Paare, daher müssen beide Spalten **NOT NULL** sein.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE           REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁰.
- Wir brauchen eine dritte Tabelle, relate_a_and_b, die die Entitäten der Typen A und B in Verbindung setzt.
- Sie hat zwei Spalten: fkaid speichert den Primärschlüssel der A-Entitäten als Fremdschlüssel und fkbid speichert die Primärschlüssel der B-Entitäten als Fremdschlüssel.
- Wir speichern nur existierende Paare, daher müssen beide Spalten **NOT NULL** sein.
- Beide Spalten haben **REFERENCES**-Einschränkungen für ihre Fremdschlüssel.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE           REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Sie hat zwei Spalten: `fkaid` speichert den Primärschlüssel der A-Entitäten als Fremdschlüssel und `fkbid` speichert die Primärschlüssel der B-Entitäten als Fremdschlüssel.
- Wir speichern nur existierende Paare, daher müssen beide Spalten `NOT NULL` sein.
- Beide Spalten haben `REFERENCES`-Einschränkungen für ihre Fremdschlüssel.
- Weil jede Entität jeder Tabelle nur mit maximal einer Entität der anderen Tabelle verbunden sein kann, darf jeder Wert in `fkaid` höchstens einmal vorkommen.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE           REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Wir speichern nur existierende Paare, daher müssen beide Spalten **NOT NULL** sein.
- Beide Spalten haben **REFERENCES**-Einschränkungen für ihre Fremdschlüssel.
- Weil jede Entität jeder Tabelle nur mit maximal einer Entität der anderen Tabelle verbunden sein kann, darf jeder Wert in **fkaid** höchstens einmal vorkommen (und das selbe gilt auch für **fkbid**).

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE           REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Wir speichern nur existierende Paare, daher müssen beide Spalten **NOT NULL** sein.
- Beide Spalten haben **REFERENCES**-Einschränkungen für ihre Fremdschlüssel.
- Weil jede Entität jeder Tabelle nur mit maximal einer Entität der anderen Tabelle verbunden sein kann, darf jeder Wert in **fkaid** höchstens einmal vorkommen (und das selbe gilt auch für **fkbid**).
- Beide Spalten sind daher **UNIQUE**.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22     ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Beide Spalten haben **REFERENCES**-Einschränkungen für ihre Fremdschlüssel.
- Weil jede Entität jeder Tabelle nur mit maximal einer Entität der anderen Tabelle verbunden sein kann, darf jeder Wert in **fkaid** höchstens einmal vorkommen (und das selbe gilt auch für **fkbid**).
- Beide Spalten sind daher **UNIQUE**.
- Beide können als **PRIMARY KEY** verwendet werden.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Weil jede Entität jeder Tabelle nur mit maximal einer Entität der anderen Tabelle verbunden sein kann, darf jeder Wert in `fkaid` höchstens einmal vorkommen (und das selbe gilt auch für `fkbid`).
- Beide Spalten sind daher `UNIQUE`.
- Beide können als `PRIMARY KEY` verwendet werden.
- Wir würden wahrscheinlich den Schlüssel wählen, der zu der „Richtung“ gehört, aus der wir am häufigsten zugreifen.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Beide Spalten sind daher **UNIQUE**.
- Beide können als **PRIMARY KEY** verwendet werden.
- Wir würden wahrscheinlich den Schlüssel wählen, der zu der „Richtung“ gehört, aus der wir am häufigsten zugreifen.
- Wenn wir häufig die passende B-Entitäten zu einer gegebenen A-Entität suchen würden, dann würden wir **fkaid** als **PRIMARY KEY** benutzen.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table B: Each row in B is related to zero or one row in A.
10 CREATE TABLE b (
11     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between A and B.
16 CREATE TABLE relate_a_and_b (
17     fkaid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES a (aid),
18     fkbid INT NOT NULL UNIQUE          REFERENCES b (bid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Wir können zuerst Daten in die Tabellen `a` und `b` füllen.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Wir können zuerst Daten in die Tabellen `a` und `b` füllen.
- Die Primärschlüsselwerte werden dabei automatisch generiert.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Wir können zuerst Daten in die Tabellen `a` und `b` füllen.
- Die Primärschlüsselwerte werden dabei automatisch generiert.
- Wir geben also nur Werte für die Attribute `x` und `y` an.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Wir können zuerst Daten in die Tabellen `a` und `b` füllen.
- Die Primärschlüsselwerte werden dabei automatisch generiert.
- Wir geben also nur Werte für die Attribute `x` und `y` an.
- Danach können wir die Beziehungen zwischen den Zeilen dieser beiden Tabellen herstellen, in dem wir Einträge in die Tabelle `relate_a_and_b` mit den Primärschlüsseln der entsprechenden Zeilen eingeben.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINS.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  -----+-----+-----+-----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Die Primärschlüsselwerte werden dabei automatisch generiert.
- Wir geben also nur Werte für die Attribute `x` und `y` an.
- Danach können wir die Beziehungen zwischen den Zeilen dieser beiden Tabellen herstellen, in dem wir Einträge in die Tabelle `relate_a_and_b` mit den Primärschlüsseln der entsprechenden Zeilen eingeben.
- `INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (2, 3);`, z. B., setzt die Zeile `aid = 2` in Tabelle `a` mit der Zeile mit `bid = 3` in Tabelle `b` in Beziehung.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINS.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir geben also nur Werte für die Attribute `x` und `y` an.
- Danach können wir die Beziehungen zwischen den Zeilen dieser beiden Tabellen herstellen, in dem wir Einträge in die Tabelle `relate_a_and_b` mit den Primärschlüsseln der entsprechenden Zeilen eingeben.
- `INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (2, 3);`,
z. B., setzt die Zeile `aid = 2` in Tabelle `a` mit der Zeile mit `bid = 3` in Tabelle `b` in Beziehung.
- Wenn wir Informationen über eine A Entität via `SELECT` holen, brauchen aber auch Informationen über die eventuell existierende zugehörige

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  -----+-----+-----+-----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Danach können wir die Beziehungen zwischen den Zeilen dieser beiden Tabellen herstellen, in dem wir Einträge in die Tabelle `relate_a_and_b` mit den Primärschlüsseln der entsprechenden Zeilen eingeben.
- `INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (2, 3);`, z. B., setzt die Zeile `aid = 2` in Tabelle `a` mit der Zeile mit `bid = 3` in Tabelle `b` in Beziehung.
- Wenn wir Informationen über eine A Entität via `SELECT` holen, brauchen aber auch Informationen über die eventuell existierende zugehörige B Entität.
- Wir benutzen ein `INNER JOIN` aus

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  -----+-----+-----+-----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- `INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (2, 3);`,
z. B., setzt die Zeile `aid = 2` in
Tabelle `a` mit der Zeile mit `bid = 3` in
Tabelle `b` in Beziehung.
- Wenn wir Informationen über eine
A Entität via `SELECT` holen, brauchen
aber auch Informationen über die
eventuell existierende zugehörige
B Entität.
- Wir benutzen ein `INNER JOIN` aus
Richtung Tabelle `a` auf die dritte
Tabelle `relate_a_and_b` basierend auf
dem Primärschlüssel `aid` von
Tabelle `a`.
- Dann brauchen wir noch ein
`INNER JOIN` mit der Tabelle `b` auf

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  ----+-----+-----+----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wenn wir Informationen über eine A Entität via `SELECT` holen, brauchen aber auch Informationen über die eventuell existierende zugehörige B Entität.
- Wir benutzen ein `INNER JOIN` aus Richtung Tabelle `a` auf die dritte Tabelle `relate_a_and_b` basierend auf dem Primärschlüssel `aid` von Tabelle `a`.
- Dann brauchen wir noch ein `INNER JOIN` mit der Tabelle `b` auf ihren Primärschlüssel `bid`.
- Wir bekommen die zusammengehörigen Daten in zwei Schritten.

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 1), (2, 3), (3, 4);
11
12 -- Combine the rows from A and B. This needs two INNER JOINs.
13 SELECT aid, x, bid as bid, y FROM relate_a_and_b
14     INNER JOIN a ON a.aid = relate_a_and_b.fkaid
15     INNER JOIN b ON b.bid = relate_a_and_b.fkbid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  aid | x | bid | y
6  -----+-----+-----+-----
7  1 | 123 | 1 | AB
8  2 | 456 | 3 | EF
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Der Inhalt der Drei Tabellen



- Dies sind die Daten in den drei Tabellen.

Table a	
aid	x
1	„123“
2	„456“
3	„789“
4	„101“

Table b	
bid	y
1	„AB“
2	„CD“
3	„EF“
4	„GH“

Table relate_a_and_b	
fkaid	fkbid
1	1
2	3
3	4

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **a** zu haben, die mit mehr als einer Zeile in Tabelle **b** in Beziehung steht.

```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because an A entry is already assigned to an B entry.
5  -- The A entity with ID 1 is already related to B entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_insert_error_1.sql
2  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "relate_a_and_b_pkey"
3  DETAIL:  Key (fkaid)=(1) already exists.
4  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because an A entry is already assigned to an B entry.
7  -- The A entity with ID 1 is already related to B entity with ID 1.
8  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);
9  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle `a` zu haben, die mit mehr als einer Zeile in Tabelle `b` in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte `fkaid` in `relate_a_and_b` verbietet das.

```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because an A entry is already assigned to an B entry.
5  -- The A entity with ID 1 is already related to B entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_insert_error_1.sql
2  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "relate_a_and_b_pkey"
3  DETAIL:  Key (fkaid)=(1) already exists.
4  psql:conceptualToRelational/AB_1_insert_error_1.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because an A entry is already assigned to an B entry.
7  -- The A entity with ID 1 is already related to B entity with ID 1.
8  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (1, 2);
9  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle `a` zu haben, die mit mehr als einer Zeile in Tabelle `b` in Beziehung steht.
- Die `UNIQUE`-Einschränkung auf Spalte `fkaid` in `relate_a_and_b` verbietet das.
- Es ist auch unmöglich, eine Zeile in Tabelle `b` zu haben, die mit mehr als einer Zeile in Tabelle `a` in Beziehung steht.

```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_insert_error_2.sql
2  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "relate_a_and_b_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 1.
8  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);
9  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle `a` zu haben, die mit mehr als einer Zeile in Tabelle `b` in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte `fkaid` in `relate_a_and_b` verbietet das.
- Es ist auch unmöglich, eine Zeile in Tabelle `b` zu haben, die mit mehr als einer Zeile in Tabelle `a` in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte `fkbid` in Tabelle `relate_a_and_b` verbietet das.

```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 1.
6  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_1_insert_error_2.sql
2  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "relate_a_and_b_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_1_insert_error_2.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 1.
8  INSERT INTO relate_a_and_b (fkaid, fkbid) VALUES (4, 3);
9  # psql 16.11 failed with exit code 3.
```

Probleme mit der Drei-Tabellen-Methode



- Es gibt zwei Probleme mit dieser Drei-Tabellen-Methode.



Probleme mit der Drei-Tabellen-Methode



- Es gibt zwei Probleme mit dieser Drei-Tabellen-Methode:
 1. Wir brauchen zwei **INNER JOIN**-Statements, um die Daten der Entitäten A und B zu verbinden.

Probleme mit der Drei-Tabellen-Methode



- Es gibt zwei Probleme mit dieser Drei-Tabellen-Methode:
 1. Wir brauchen zwei `INNER JOIN`-Statements, um die Daten der Entitäten A und B zu verbinden.
 2. Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen A und B Entitäten gibt.

Probleme mit der Drei-Tabellen-Methode



- Es gibt zwei Probleme mit dieser Drei-Tabellen-Methode:
 1. Wir brauchen zwei `INNER JOIN`-Statements, um die Daten der Entitäten A und B zu verbinden.
 2. Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen A und B Entitäten gibt.
 - Im schlimmsten Fall sind alle Entitäten von Typ A mit Entitäten von Typ B verbunden (oder umgekehrt).

Probleme mit der Drei-Tabellen-Methode



- Es gibt zwei Probleme mit dieser Drei-Tabellen-Methode:
 1. Wir brauchen zwei `INNER JOIN`-Statements, um die Daten der Entitäten A und B zu verbinden.
 2. Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen A und B Entitäten gibt.
 - Im schlimmsten Fall sind alle Entitäten von Typ A mit Entitäten von Typ B verbunden (oder umgekehrt).
 - Dann ist die dritte Tabelle so groß wie die kleiner der Tabellen `a` und `b`.

Zwei-Tabellen Lösung

- Wenn die meisten Entitäten von Typ A mit solchen von Typ B verbunden sind ... dann könnten wir genauso gut einfach die Fremdschlüssel direkt in Tabelle [a](#) speichern...



Zwei-Tabellen Lösung



- Wenn die meisten Entitäten von Typ A mit solchen von Typ B verbunden sind ... dann könnten wir genauso gut einfach die Fremdschlüssel direkt in Tabelle [a](#) speichern...
- Das probieren wir jetzt aus.

Zwei-Tabellen Lösung

- Wenn die meisten Entitäten von Typ A mit solchen von Typ B verbunden sind ... dann könnten wir genauso gut einfach die Fremdschlüssel direkt in Tabelle `a` speichern...
- Das probieren wir jetzt aus.
- Dafür löschen wir die Tabellen erstmal wieder.

```
1  /* Drop the tables for the A-|o-----o|-B relationship. */
2
3  DROP TABLE IF EXISTS relate_a_and_b;
4  DROP TABLE IF EXISTS a;
5  DROP TABLE IF EXISTS b;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_cleanup.sql
2  DROP TABLE
3  DROP TABLE
4  DROP TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Wenn die meisten Entitäten von Typ A mit solchen von Typ B verbunden sind ... dann könnten wir genauso gut einfach die Fremdschlüssel direkt in Tabelle `a` speichern. . .
- Das probieren wir jetzt aus.
- Dafür löschen wir die Tabellen erstmal wieder.
- Und jetzt erstellen wir sie neu.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid  INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x      CHAR(3)      -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)      -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Wenn die meisten Entitäten von Typ A mit solchen von Typ B verbunden sind ... dann könnten wir genauso gut einfach die Fremdschlüssel direkt in Tabelle `a` speichern. ...
- Das probieren wir jetzt aus.
- Dafür löschen wir die Tabellen erstmal wieder.
- Und jetzt erstellen wir sie neu.
- Die neue Spalte `fkbid` wird zu Tabelle `a` hinzugefügt.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid  INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x      CHAR(3)      -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)      -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Das probieren wir jetzt aus.
- Dafür löschen wir die Tabellen erstmal wieder.
- Und jetzt erstellen wir sie neu.
- Die neue Spalte `fkbid` wird zu Tabelle `a` hinzugefügt.
- Sie darf `NULL` sein, weil ja nicht alle Entitäten vom Typ A mit Entitäten vom Typ B verbunden seien müssen (und umgekehrt).

```
1  /* Create the tables for an A-o----o-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later <-- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Dafür löschen wir die Tabellen erstmal wieder.
- Und jetzt erstellen wir sie neu.
- Die neue Spalte `fkbid` wird zu Tabelle `a` hinzugefügt.
- Sie darf `NULL` sein, weil ja nicht alle Entitäten vom Typ A mit Entitäten vom Typ B verbunden seien müssen (und umgekehrt).
- Sie muss `UNIQUE` sein, denn keine Entität vom Typ B kann mit mehr als einer Entität vom Typ A verbunden sein.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid  INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x      CHAR(3)      -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)      -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Und jetzt erstellen wir sie neu.
- Die neue Spalte `fkbid` wird zu Tabelle `a` hinzugefügt.
- Sie darf `NULL` sein, weil ja nicht alle Entitäten vom Typ A mit Entitäten vom Typ B verbunden seien müssen (und umgekehrt).
- Sie muss `UNIQUE` sein, denn keine Entität vom Typ B kann mit mehr als einer Entität vom Typ A verbunden sein.
- Diese Einschränkung gilt aber nur für die `fkbid`-Einträge, die nicht `NULL` sind.

```
1  /* Create the tables for an A-o----o-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later <-- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Die neue Spalte `fkbid` wird zu Tabelle `a` hinzugefügt.
- Sie darf `NULL` sein, weil ja nicht alle Entitäten vom Typ A mit Entitäten vom Typ B verbunden seien müssen (und umgekehrt).
- Sie muss `UNIQUE` sein, denn keine Entität vom Typ B kann mit mehr als einer Entität vom Typ A verbunden sein.
- Diese Einschränkung gilt aber nur für die `fkbid`-Einträge, die nicht `NULL` sind.
- Sie braucht auch eine `REFERENCES`-Einschränkung.

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid  INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x      CHAR(3)      -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)      -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Sie darf **NULL** sein, weil ja nicht alle Entitäten vom Typ A mit Entitäten vom Typ B verbunden sein müssen (und umgekehrt).
- Sie muss **UNIQUE** sein, denn keine Entität vom Typ B kann mit mehr als einer Entität vom Typ A verbunden sein.
- Diese Einschränkung gilt aber nur für die **fkbid**-Einträge, die nicht **NULL** sind.
- Sie braucht auch eine **REFERENCES**-Einschränkung.
- Das fügen wir später mit **ALTER TABLE**¹ hinzu (PgModeler macht das ja auch so).

```
1  /* Create the tables for an A-|o-----o|-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later <-- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Sie muss **UNIQUE** sein, denn keine Entität vom Typ B kann mit mehr als einer Entität vom Typ A verbunden sein.
- Diese Einschränkung gilt aber nur für die **fkbid**-Einträge, die nicht **NULL** sind.
- Sie braucht auch eine **REFERENCES**-Einschränkung.
- Das fügen wir später mit **ALTER TABLE**¹ hinzu (PgModeler macht das ja auch so).
- Sonst müssten wie die Tabelle **b** vor Tabelle **a** erstellen.

```
1  /* Create the tables for an A-o----o-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later <-- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Diese Einschränkung gilt aber nur für die `fkbid`-Einträge, die nicht `NULL` sind.
- Sie braucht auch eine `REFERENCES`-Einschränkung.
- Das fügen wir später mit `ALTER TABLE`¹ hinzu (PgModeler macht das ja auch so).
- Sonst müssten wie die Tabelle `b` vor Tabelle `a` erstellen.
- Wenn wir viele Tabellen hätten, müssten wir die immer in einer bestimmten Reihenfolge sortieren, was alles komplizierter und daher fehleranfälliger macht.

```
1  /* Create the tables for an A-to-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later -- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Sie braucht auch eine **REFERENCES**-Einschränkung.
- Das fügen wir später mit **ALTER TABLE**¹ hinzu (PgModeler macht das ja auch so).
- Sonst müssten wie die Tabelle **b** vor Tabelle **a** erstellen.
- Wenn wir viele Tabellen hätten, müssten wir die immer in einer bestimmten Reihenfolge sortieren, was alles komplizierter und daher fehleranfälliger macht.
- Deshalb werden wir immer zuerst die Tabellen erstellen und danach die Einschränkungen, die nicht sofort erstellt werden können.

```
1  /* Create the tables for an A-to-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later -- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Das fügen wir später mit `ALTER TABLE`¹ hinzu (PgModeler macht das ja auch so).
- Sonst müssten wie die Tabelle `b` vor Tabelle `a` erstellen.
- Wenn wir viele Tabellen hätten, müssten wir die immer in einer bestimmten Reihenfolge sortieren, was alles komplizierter und daher fehleranfälliger macht.
- Deshalb werden wir immer zuerst die Tabellen erstellen und danach die Einschränkungen, die nicht sofort erstellt werden können.
- So oder so, wir können mit zwei Tabellen auskommen.

```
1  /* Create the tables for an A-to-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid INT UNIQUE, -- foreign key to B, see later -- can be NULL.
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Wenn wir viele Tabellen hätten, müssten wir die immer in einer bestimmten Reihenfolge sortieren, was alles komplizierter und daher fehleranfälliger macht.
- Deshalb werden wir immer zuerst die Tabellen erstellen und danach die Einschränkungen, die nicht sofort erstellt werden können.
- So oder so, wir können mit zwei Tabellen auskommen.
- Wir hätten es auch andersherum machen können, in dem wir Tabelle `b` einen Fremdschlüssel `fkaid` auf Tabelle `a` geben.

```
1  /* Create the tables for an A-o----o-B relationship. */
2
3  -- Table A: Each row in A is related to zero or one row in B.
4  CREATE TABLE a (
5      aid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkbid  INT UNIQUE,    -- foreign key to B, see later <-- can be NULL.
7      x      CHAR(3)      -- example of other attributes
8  );
9
10 -- Table B: Each row in B is related to zero or one row in A.
11 CREATE TABLE b (
12     bid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)      -- example of other attributes
14 );
15
16 -- To table A, we add the foreign key reference constraint to table B.
17 ALTER TABLE a ADD CONSTRAINT a_fkbid_fk FOREIGN KEY (fkbid)
18 REFERENCES b (bid);
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf AB_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nun Daten genauso wie vorher in die Tabellen einfügen.

```
1  /* Inserting data into the tables for the A-o----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  UPDATE 1
5  UPDATE 1
6  UPDATE 1
7  aid | x | bid | y
8  ----+-----+-----+----
9      1 | 123 | 1 | AB
10     2 | 456 | 3 | EF
11     3 | 789 | 4 | GH
12 (3 rows)
13
14 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nun Daten genauso wie vorher in die Tabellen einfügen.
- Anders als vorher kommen die Beziehungen nicht in eine zusätzliche Tabelle.

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  UPDATE 1
5  UPDATE 1
6  UPDATE 1
7  aid | x | bid | y
8  ----+-----+-----+----
9      1 | 123 | 1 | AB
10     2 | 456 | 3 | EF
11     3 | 789 | 4 | GH
12 (3 rows)
13
14 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nun Daten genauso wie vorher in die Tabellen einfügen.
- Anders als vorher kommen die Beziehungen nicht in eine zusätzliche Tabelle.
- Stattdessen benutzen wir **UPDATE**-Statements⁷¹.

```
1  /* Inserting data into the tables for the A-o-----o-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  UPDATE 1
5  UPDATE 1
6  UPDATE 1
7  aid | x | bid | y
8  ----+-----+-----+----
9      1 | 123 | 1 | AB
10     2 | 456 | 3 | EF
11     3 | 789 | 4 | GH
12 (3 rows)
13
14 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nun Daten genauso wie vorher in die Tabellen einfügen.
- Anders als vorher kommen die Beziehungen nicht in eine zusätzliche Tabelle.
- Stattdessen benutzen wir **UPDATE**-Statements⁷¹.
- **UPDATE a SET fkbid = 3 WHERE aid = 2;**, z. B., setzt die Zeile mit **aid = 2** in Tabelle **a** mit der Zeile mit **bid = 3** in Tabelle **b** in Beziehung.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  UPDATE 1
5  UPDATE 1
6  UPDATE 1
7  aid | x | bid | y
8  ----+-----+-----+----
9      1 | 123 | 1 | AB
10     2 | 456 | 3 | EF
11     3 | 789 | 4 | GH
12 (3 rows)
13
14 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nun Daten genauso wie vorher in die Tabellen einfügen.
- Anders als vorher kommen die Beziehungen nicht in eine zusätzliche Tabelle.
- Stattdessen benutzen wir `UPDATE`-Statements⁷¹.
- `UPDATE a SET fkbid = 3 WHERE aid = 2;`, z. B., setzt die Zeile mit `aid = 2` in Tabelle `a` mit der Zeile mit `bid = 3` in Tabelle `b` in Beziehung.
- Wir können die Daten nun über ein einzelnes `INNER JOIN` zusammenführen.

```
1  /* Inserting data into the tables for the A-|o-----o|-B relationship. */
2
3  -- Insert some rows into the table for entity type A.
4  INSERT INTO a (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type B.
7  INSERT INTO b (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the A and B rows.
10 UPDATE a SET fkbid = 1 WHERE aid = 1;
11 UPDATE a SET fkbid = 3 WHERE aid = 2;
12 UPDATE a SET fkbid = 4 WHERE aid = 3;
13
14 -- Combine the rows from A and B. Only one INNER JOIN is needed.
15 SELECT aid, x, bid, y FROM a INNER JOIN b ON a.fkbid = b.bid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf AB_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  UPDATE 1
5  UPDATE 1
6  UPDATE 1
7  aid | x | bid | y
8  ----+-----+-----+----
9      1 | 123 | 1 | AB
10     2 | 456 | 3 | EF
11     3 | 789 | 4 | GH
12 (3 rows)
13
14 # psql 16.11 succeeded with exit code 0.
```

Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

Table a

aid	fkbid	x
4	NULL	„101“
1	1	„123“
2	3	„456“
3	4	„789“

Table b

bid	y
1	„AB“
2	„CD“
3	„EF“
4	„GH“

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle [a](#) zu haben, die mit mehr als einer Zeile in Tabelle [b](#) in Beziehung steht.



Testen der Einschränkungen



- Es ist unmöglich, eine Zeile in Tabelle `a` zu haben, die mit mehr als einer Zeile in Tabelle `b` in Beziehung steht.
- Das Feld `fkbid` kann ja nur maximal einen Wert haben (`NULL` oder ein gültiger Fremdschlüssel).

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **a** zu haben, die mit mehr als einer Zeile in Tabelle **b** in Beziehung steht.
- Das Feld **fkbid** kann ja nur maximal einen Wert haben (**NULL** oder ein gültiger Fremdschlüssel).
- Es ist auch unmöglich, eine Zeile in Tabelle **b** zu haben, die mit mehr als einer Zeile in Tabelle **a** in Beziehung steht.



```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 2.
6  UPDATE a SET fkbid = 3 where aid = 4;

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_2_insert_error_2.sql
2  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "a_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 2.
8  UPDATE a SET fkbid = 3 where aid = 4;
9  # psql 16.11 failed with exit code 3.
```

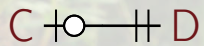
Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **a** zu haben, die mit mehr als einer Zeile in Tabelle **b** in Beziehung steht.
- Das Feld **fkbid** kann ja nur maximal einen Wert haben (**NULL** oder ein gültiger Fremdschlüssel).
- Es ist auch unmöglich, eine Zeile in Tabelle **b** zu haben, die mit mehr als einer Zeile in Tabelle **a** in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte **fkbid** in Tabelle **a** verbietet das.



```
1  /* Insert a wrong row into tables for A-|o-----o|-B relationship. */
2
3  -- Create an error in the relationships between the A and B rows.
4  -- This fails because a B entry is already assigned to an A entry.
5  -- The B entity with ID 3 is already related to A entity with ID 2.
6  UPDATE a SET fkbid = 3 where aid = 4;

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf AB_2_insert_error_2.sql
2  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: ERROR:  duplicate
   ↳ key value violates unique constraint "a_fkbid_key"
3  DETAIL:  Key (fkbid)=(3) already exists.
4  psql:conceptualToRelational/AB_2_insert_error_2.sql:6: STATEMENT:  /*
   ↳ Insert a wrong row into tables for A-|o-----o|-B relationship. */
5  -- Create an error in the relationships between the A and B rows.
6  -- This fails because a B entry is already assigned to an A entry.
7  -- The B entity with ID 3 is already related to A entity with ID 2.
8  UPDATE a SET fkbid = 3 where aid = 4;
9  # psql 16.11 failed with exit code 3.
```



C  D

- Es gibt zwei Entitätstypen C und D.



C  D



- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.

C  D

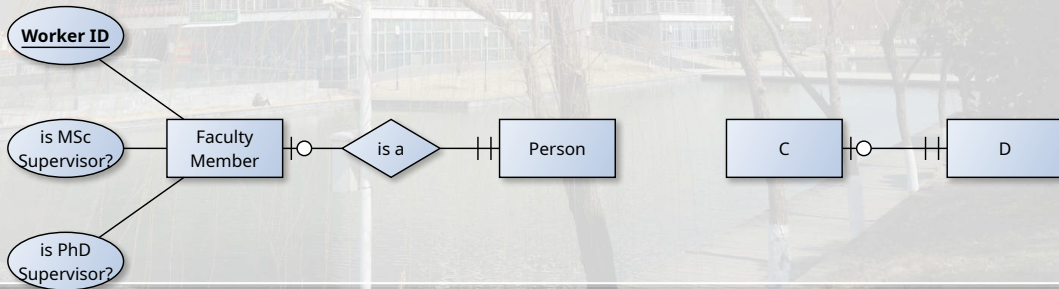


- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.

C  D



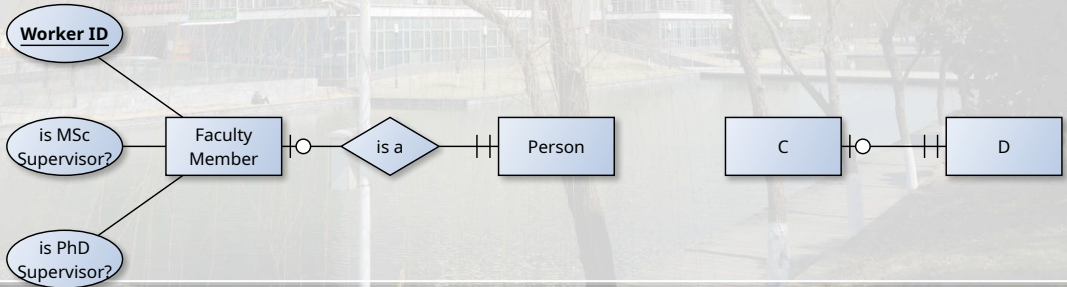
- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.
- Beispiel:



C  D



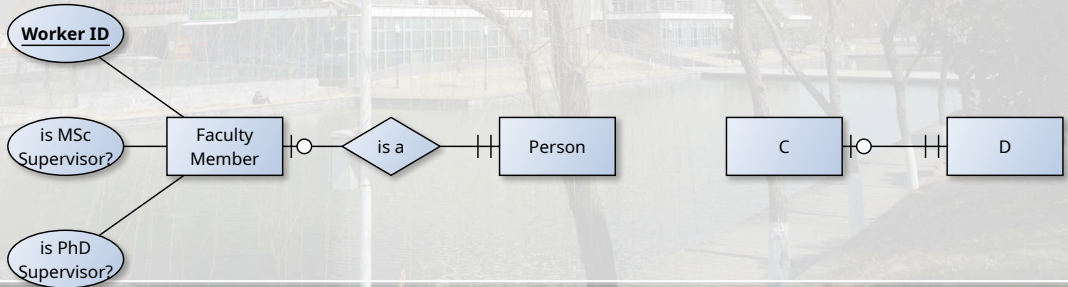
- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.
- Beispiel:
 - Jede Entität vom Typ *Person* kann ein Mitarbeiter (*Faculty*) der Uni sein ... oder auch nicht.



C  D



- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.
- Beispiel:
 - Jede Entität vom Typ *Person* kann ein Mitarbeiter (*Faculty*) der Uni sein ... oder auch nicht.
 - Jede Entität vom Typ *Faculty* muss mit genau einer Entität vom Typ *Person* verbunden sein.



- Eine Lösung mit drei Tabellen ergibt hier gar keinen Sinn.

中国安徽德国中心

合肥德国应用科学学院

- Eine Lösung mit drei Tabellen ergibt hier gar keinen Sinn.
- Wir *wissen*, dass jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein *muss*.

- Eine Lösung mit drei Tabellen ergibt hier gar keinen Sinn.
- Wir *wissen*, dass jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein *muss*.
- Die Lösung ist also so ähnlich wie die zwei-Tabellen-Variante aus dem vorigen Abschnitt.

Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle **C** für die Entitäten vom Typ C.

```
1  /* Create the tables for a C-|o-----||-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
```

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

Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle **c** für die Entitäten vom Typ C.
- Wir brauchen auch eine Tabelle **d** für die Entitäten vom Typ D.

```
1  /* Create the tables for a C-|o-----||-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
```

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

Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle `c` für die Entitäten vom Typ C.
- Wir brauchen auch eine Tabelle `d` für die Entitäten vom Typ D.
- Beide Tabellen sind wie im vorigen Beispiel strukturiert, mit Ersatz-Primärschlüsseln `cid` und `did` sowie mit den Attributen `x` und `y`.

```
1  /* Create the tables for a C-|o-----||-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkidid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkidid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
```

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

Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle `c` für die Entitäten vom Typ C.
- Wir brauchen auch eine Tabelle `d` für die Entitäten vom Typ D.
- Beide Tabellen sind wie im vorigen Beispiel strukturiert, mit Ersatz-Primärschlüsseln `cid` und `did` sowie mit den Attributen `x` und `y`.
- Weil jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein muss, fügen wir die Spalte `fkdid` zur Tabelle `c` hinzu, um Fremdschlüssel von D-Entitäten zu speichern.

```
1  /* Create the tables for a C-to-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
```

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

Zwei-Tabellen Lösung

- Wir brauchen auch eine Tabelle `d` für die Entitäten vom Typ D.
- Beide Tabellen sind wie im vorigen Beispiel strukturiert, mit Ersatz-Primärschlüsseln `cid` und `did` sowie mit den Attributen `x` und `y`.
- Weil jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein muss, fügen wir die Spalte `fkdid` zur Tabelle `c` hinzu, um Fremdschlüssel von D-Entitäten zu speichern.
- Diese Spalte hat daher eine `REFERENCES`-Einschränkung zum Fremdschlüssel, die wir später mit `ALTER TABLE` hinzufügen

```
1  /* Create the tables for a C-to-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
19
20 $ psql "postgres://postgres:XXX@localhost/relationships" -v
21 ↪ ON_ERROR_STOP=1 -ebf CD_tables.sql
22 CREATE TABLE
23 CREATE TABLE
24 ALTER TABLE
25 # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Beide Tabellen sind wie im vorigen Beispiel strukturiert, mit Ersatz-Primärschlüsseln `cid` und `did` sowie mit den Attributen `x` und `y`.
- Weil jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein muss, fügen wir die Spalte `fkdid` zur Tabelle `c` hinzu, um Fremdschlüssel von D-Entitäten zu speichern.
- Diese Spalte hat daher eine `REFERENCES`-Einschränkung zum Fremdschlüssel, die wir später mit `ALTER TABLE` hinzufügen
- **Anders** als vorher muss die Spalte `fkdid` auch `NOT NULL` sein, denn jede Entität vom Typ C *muss* mit einer Entität vom Typ D verbunden sein

```
1  /* Create the tables for a C-to-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18     REFERENCES d (did);
```

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

Zwei-Tabellen Lösung

- Weil jede Entität vom Typ C mit genau einer Entität vom Typ D verbunden sein muss, fügen wir die Spalte `fkdid` zur Tabelle `c` hinzu, um Fremdschlüssel von D-Entitäten zu speichern.
- Diese Spalte hat daher eine `REFERENCES`-Einschränkung zum Fremdschlüssel, die wir später mit `ALTER TABLE` hinzufügen
- **Anders** als vorher muss die Spalte `fkdid` auch `NOT NULL` sein, denn jede Entität vom Typ C *muss* mit einer Entität vom Typ D verbunden sein.
- Die Spalte hat auch eine `UNIQUE`-Einschränkung, denn Entitäten vom Typ D dürfen jeweils höchstens einen Eintrag aus der Tabelle `c`

```
1  /* Create the tables for a C-to-D relationship. */
2
3  -- Table C: Each row in C is related to exactly one row in D.
4  CREATE TABLE c (
5      cid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkdid INT NOT NULL UNIQUE, -- the foreign key to D, see later
7      x CHAR(3) -- example of other attributes
8  );
9
10 -- Table D: Each row in D is related to zero or one row in C.
11 CREATE TABLE d (
12     did INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y CHAR(2) -- example of other attributes
14 );
15
16 -- To table C, we add the foreign key reference constraint to table D.
17 ALTER TABLE c ADD CONSTRAINT c_fkdid_fk FOREIGN KEY (fkdid)
18 REFERENCES d (did);
```

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

Befüllen mit Daten

- Wir können nur Zeilen in die Tabelle **c** einfügen, die auf bereits existierende Zeilen in Tabelle **d** verweisen.

```
1  /* Inserting data into the tables for the C-to-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10                                     (2, '101');
11
12  -- Combine the rows from C and D.
13  SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf CD_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  cid | x | did | y
6  ----+----+----+---
7  1 | 123 | 1 | AB
8  4 | 101 | 2 | CD
9  2 | 456 | 3 | EF
10  3 | 789 | 4 | GH
11  (4 rows)
12  # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nur Zeilen in die Tabelle **c** einfügen, die auf bereits existierende Zeilen in Tabelle **d** verweisen.
- Die Konsequenz ist, dass diese Datensätze zuerst erzeugt werden müssen.

```
1  /* Inserting data into the tables for the C-to-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10                                     (2, '101');
11
12 -- Combine the rows from C and D.
13 SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf CD_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  cid | x | did | y
6  ----+----+----+---
7  1 | 123 | 1 | AB
8  4 | 101 | 2 | CD
9  2 | 456 | 3 | EF
10  3 | 789 | 4 | GH
11  (4 rows)
12
13 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nur Zeilen in die Tabelle **c** einfügen, die auf bereits existierende Zeilen in Tabelle **d** verweisen.
- Die Konsequenz ist, dass diese Datensätze zuerst erzeugt werden müssen.
- Nach dem wir die Datensätze in die Tabelle **d** eingefügt haben, können wir Zeilen in die Tabelle **c** einfügen.

```
1  /* Inserting data into the tables for the C-to-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10                                     (2, '101');
11
12  -- Combine the rows from C and D.
13  SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf CD_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  cid | x | did | y
6  ----+----+----+---
7  1 | 123 | 1 | AB
8  4 | 101 | 2 | CD
9  2 | 456 | 3 | EF
10  3 | 789 | 4 | GH
11  (4 rows)
12  # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nur Zeilen in die Tabelle **c** einfügen, die auf bereits existierende Zeilen in Tabelle **d** verweisen.
- Die Konsequenz ist, dass diese Datensätze zuerst erzeugt werden müssen.
- Nach dem wir die Datensätze in die Tabelle **d** eingefügt haben, können wir Zeilen in die Tabelle **c** einfügen.
- Jede dieser Zeilen muss einen gültigen und einmaligen Fremdschlüssel **fkdid** haben, der einen Primärschlüsselwert **did** in Tabelle **d** referenziert.

```
1  /* Inserting data into the tables for the C-|o-----||-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10                                     (2, '101');
11
12 -- Combine the rows from C and D.
13 SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf CD_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  cid | x | did | y
6  ----+----+----+---
7  1 | 123 | 1 | AB
8  4 | 101 | 2 | CD
9  2 | 456 | 3 | EF
10  3 | 789 | 4 | GH
11  (4 rows)
12  # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir können nur Zeilen in die Tabelle `c` einfügen, die auf bereits existierende Zeilen in Tabelle `d` verweisen.
- Die Konsequenz ist, dass diese Datensätze zuerst erzeugt werden müssen.
- Nach dem wir die Datensätze in die Tabelle `d` eingefügt haben, können wir Zeilen in die Tabelle `c` einfügen.
- Jede dieser Zeilen muss einen gültigen und einmaligen Fremdschlüssel `fkdid` haben, der einen Primärschlüsselwert `did` in Tabelle `d` referenziert.
- Die Daten können über einziges `INNER JOIN` zusammengeführt werden.

```
1  /* Inserting data into the tables for the C-to-D relationship. */
2
3  -- Insert some rows into the table for entity type D.
4  -- We first must create the D elements, because the C rows cannot
5  -- exist without referencing one row in D each.
6  INSERT INTO d (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
7
8  -- Insert some rows into the table for entity type C.
9  INSERT INTO c (fkdid, x) VALUES (1, '123'), (3, '456'), (4, '789'),
10                                     (2, '101');
11
12 -- Combine the rows from C and D.
13 SELECT cid, x, did, y FROM c INNER JOIN d ON c.fkdid = d.did;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf CD_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
5  cid | x | did | y
6  ----+----+----+---
7  1 | 123 | 1 | AB
8  4 | 101 | 2 | CD
9  2 | 456 | 3 | EF
10  3 | 789 | 4 | GH
11  (4 rows)
12  # psql 16.11 succeeded with exit code 0.
```

Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

Table **c**

cid	fkdid	x
1	1	„123“
2	3	„456“
3	4	„789“
4	2	„101“

Table **d**

did	y
1	„AB“
2	„CD“
3	„EF“
4	„GH“

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **c** zu haben, die mit mehr als einer Zeile in Tabelle **d** in Beziehung steht.



Testen der Einschränkungen



- Es ist unmöglich, eine Zeile in Tabelle `c` zu haben, die mit mehr als einer Zeile in Tabelle `d` in Beziehung steht.
- Das Feld `fkdid` kann ja nur genau einen Wert haben.

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **c** zu haben, die mit mehr als einer Zeile in Tabelle **d** in Beziehung steht.
- Das Feld **fkdid** kann ja nur genau einen Wert haben.
- Es ist auch unmöglich, eine Zeile in Tabelle **d** zu haben, die mit mehr als einer Zeile in Tabelle **c** in Beziehung steht.

```
1  /* Insert a wrong row into tables for C-|o-----||-D relationship. */
2
3  -- It is impossible to create a row in C that references a row in D
4  -- which is already referenced by another record.
5  INSERT INTO c (fkdid, x) VALUES (3, '555');

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf CD_insert_error_1.sql
2  psql:conceptualToRelational/CD_insert_error_1.sql:5: ERROR:  duplicate
   ↳ key value violates unique constraint "c_fkdid_key"
3  DETAIL:  Key (fkdid)=(3) already exists.
4  psql:conceptualToRelational/CD_insert_error_1.sql:5: STATEMENT:  /*
   ↳ Insert a wrong row into tables for C-|o-----||-D relationship. */
5  -- It is impossible to create a row in C that references a row in D
6  -- which is already referenced by another record.
7  INSERT INTO c (fkdid, x) VALUES (3, '555');
8  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist unmöglich, eine Zeile in Tabelle **c** zu haben, die mit mehr als einer Zeile in Tabelle **d** in Beziehung steht.
- Das Feld **fkdid** kann ja nur genau einen Wert haben.
- Es ist auch unmöglich, eine Zeile in Tabelle **d** zu haben, die mit mehr als einer Zeile in Tabelle **c** in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte **fkdid** in Tabelle **c** verbietet das.

```
1  /* Insert a wrong row into tables for C-|o-----||-D relationship. */
2
3  -- It is impossible to create a row in C that references a row in D
4  -- which is already referenced by another record.
5  INSERT INTO c (fkdid, x) VALUES (3, '555');

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf CD_insert_error_1.sql
2  psql:conceptualToRelational/CD_insert_error_1.sql:5: ERROR:  duplicate
   ↳ key value violates unique constraint "c_fkdid_key"
3  DETAIL:  Key (fkdid)=(3) already exists.
4  psql:conceptualToRelational/CD_insert_error_1.sql:5: STATEMENT:  /*
   ↳ Insert a wrong row into tables for C-|o-----||-D relationship. */
5  -- It is impossible to create a row in C that references a row in D
6  -- which is already referenced by another record.
7  INSERT INTO c (fkdid, x) VALUES (3, '555');
8  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist auch unmöglich, eine Zeile in Tabelle **d** zu haben, die mit mehr als einer Zeile in Tabelle **c** in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte **fkdid** in Tabelle **c** verbietet das.
- Wir können auch keine Zeile in Tabelle **c** einfügen, die keine Zeile in Tabelle **d** referenziert.

```
1  /* Insert a wrong row into tables for C-|o-----||-D relationship. */
2
3  -- It is impossible to create a row in C that references no row in D.
4  INSERT INTO c (x) VALUES ('365');
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf CD_insert_error_2.sql
2  psql:conceptualToRelational/CD_insert_error_2.sql:4: ERROR:  null value
   ↳ in column "fkdid" of relation "c" violates not-null constraint
3  DETAIL:  Failing row contains (6, null, 365).
4  psql:conceptualToRelational/CD_insert_error_2.sql:4: STATEMENT:  /*
   ↳ Insert a wrong row into tables for C-|o-----||-D relationship. */
5  -- It is impossible to create a row in C that references no row in D.
6  INSERT INTO c (x) VALUES ('365');
7  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist auch unmöglich, eine Zeile in Tabelle **d** zu haben, die mit mehr als einer Zeile in Tabelle **c** in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte **fkdid** in Tabelle **c** verbietet das.
- Wir können auch keine Zeile in Tabelle **c** einfügen, die keine Zeile in Tabelle **d** referenziert.
- Das wird durch die **NOT NULL**-Einschränkung verhindert.

```
1  /* Insert a wrong row into tables for C-|o-----||-D relationship. */
2
3  -- It is impossible to create a row in C that references no row in D.
4  INSERT INTO c (x) VALUES ('365');
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf CD_insert_error_2.sql
2  psql:conceptualToRelational/CD_insert_error_2.sql:4: ERROR:  null value
   ↳ in column "fkdid" of relation "c" violates not-null constraint
3  DETAIL:  Failing row contains (6, null, 365).
4  psql:conceptualToRelational/CD_insert_error_2.sql:4: STATEMENT:  /*
   ↳ Insert a wrong row into tables for C-|o-----||-D relationship. */
5  -- It is impossible to create a row in C that references no row in D.
6  INSERT INTO c (x) VALUES ('365');
7  # psql 16.11 failed with exit code 3.
```

Testen der Einschränkungen

- Es ist auch unmöglich, eine Zeile in Tabelle **d** zu haben, die mit mehr als einer Zeile in Tabelle **c** in Beziehung steht.
- Die **UNIQUE**-Einschränkung auf Spalte **fkdid** in Tabelle **c** verbietet das.
- Wir können auch keine Zeile in Tabelle **c** einfügen, die keine Zeile in Tabelle **d** referenziert.
- Das wird durch die **NOT NULL**-Einschränkung verhindert.
- Beachten Sie, dass daher Tabelle **d** niemals weniger Einträge als Tabelle **c** haben kann.

```
1  /* Insert a wrong row into tables for C-|o-----||-D relationship. */
2
3  -- It is impossible to create a row in C that references no row in D.
4  INSERT INTO c (x) VALUES ('365');
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf CD_insert_error_2.sql
2  psql:conceptualToRelational/CD_insert_error_2.sql:4: ERROR:  null value
   ↳ in column "fkdid" of relation "c" violates not-null constraint
3  DETAIL:  Failing row contains (6, null, 365).
4  psql:conceptualToRelational/CD_insert_error_2.sql:4: STATEMENT:  /*
   ↳ Insert a wrong row into tables for C-|o-----||-D relationship. */
5  -- It is impossible to create a row in C that references no row in D.
6  INSERT INTO c (x) VALUES ('365');
7  # psql 16.11 failed with exit code 3.
```



$E \vdash \circ - \circ \vdash F$



$E \text{ --- } \bigcirc \text{ --- } F$

- Es gibt zwei Entitätstypen E und F.



$E \text{ --- } \bigcirc \text{ --- } \bigcirc \text{ --- } F$



- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.

$E \text{ --- } \bigcirc \text{ --- } \bigcirc \text{ --- } F$

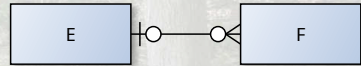
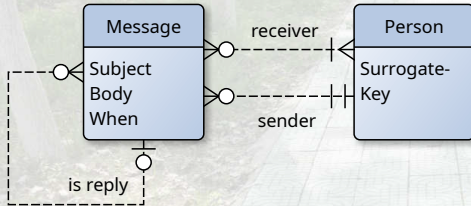


- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.

$E \text{ } \text{---} \bigcirc \text{---} \bigcirc \text{ } F$



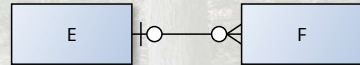
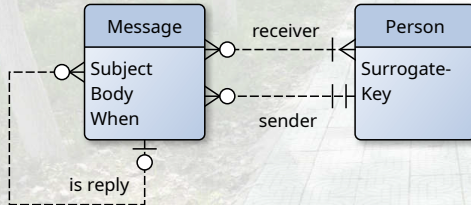
- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:



E $\oplus$ ○ — ○ $\times$ F



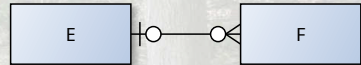
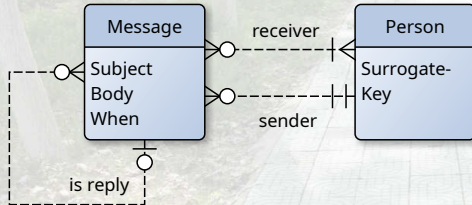
- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:
 - Eine Nachricht kann entweder eine neue Nachricht sein, oder die Antwort auf eine existierende Nachricht.



E $\oplus$ ○ — ○ $\times$ F



- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:
 - Eine Nachricht kann entweder eine neue Nachricht sein, oder die Antwort auf eine existierende Nachricht. Es kann keine, eine, oder beliebig viele Antworten auf jede Nachricht geben.



Lösungen



- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.

Lösungen



- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.

Lösungen



- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.

Lösungen



- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.

- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.
- Wieder gibt es zwei mögliche Lösungen für dieses Szenario.

- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.
- Wieder gibt es zwei mögliche Lösungen für dieses Szenario:
 1. Wir können drei Tabellen nehmen, wobei die dritte Tabelle wieder die „Verbindungen“ speichert.

- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.
- Wieder gibt es zwei mögliche Lösungen für dieses Szenario:
 1. Wir können drei Tabellen nehmen, wobei die dritte Tabelle wieder die „Verbindungen“ speichert oder
 2. wir können zwei Tabellen nehmen, wobei die „Verbindungsdaten“ diesmal in Tabelle **f** gehen müssen.

- Wir brauchen eine Tabelle **e** für die Entitäten vom Typ E.
- Wir brauchen auch eine Tabelle **f** für die Entitäten vom Typ F.
- Beide haben ihre Ersatz-Primärschlüssel **eid** und **fid**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.
- Wieder gibt es zwei mögliche Lösungen für dieses Szenario:
 1. Wir können drei Tabellen nehmen, wobei die dritte Tabelle wieder die „Verbindungen“ speichert oder
 2. wir können zwei Tabellen nehmen, wobei die „Verbindungsdaten“ diesmal in Tabelle **f** gehen müssen.
- Welche Lösung besser ist, hängt wieder mit der Anzahl der verbundenen Datensätze zusammen.

Mit Drei Tabellen

- Wir benutzen eine dritte Tabelle `relate_e_and_f`, die die Entitäten vom Typ E mit denen vom Typ F in Verbindung setzt.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Wir benutzen eine dritte Tabelle `relate_e_and_f`, die die Entitäten vom Typ E mit denen vom Typ F in Verbindung setzt.
- In der Tabelle gibt es zwei Spalten.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Wir benutzen eine dritte Tabelle `relate_e_and_f`, die die Entitäten vom Typ E mit denen vom Typ F in Verbindung setzt.
- In der Tabelle gibt es zwei Spalten.
- Die erste, `fkeid`, speichert die Werte der Primärschlüssel `eid` der E-Entitäten.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Wir benutzen eine dritte Tabelle `relate_e_and_f`, die die Entitäten vom Typ E mit denen vom Typ F in Verbindung setzt.
- In der Tabelle gibt es zwei Spalten.
- Die erste, `fkeid`, speichert die Werte der Primärschlüssel `eid` der E-Entitäten.
- Die zweite, `fkfid`, speichert die Werte der Primärschlüssel `fid` der F-Entitäten.

```
1  /* Create the tables for an E-o----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22 ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- In der Tabelle gibt es zwei Spalten.
- Die erste, `fkeid`, speichert die Werte der Primärschlüssel `eid` der E-Entitäten.
- Die zweite, `fkfid`, speichert die Werte der Primärschlüssel `fid` der F-Entitäten.
- Da wir nur Paare speichern, die existieren, sind beide Spalten `NOT NULL` und haben `REFERENCES`-Einschränkungen auf ihre entsprechenden Fremdschlüssel.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22 ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Die erste, `fkeid`, speichert die Werte der Primärschlüssel `eid` der E-Entitäten.
- Die zweite, `fkfid`, speichert die Werte der Primärschlüssel `fid` der F-Entitäten.
- Da wir nur Paare speichern, die existieren, sind beide Spalten `NOT NULL` und haben `REFERENCES`-Einschränkungen auf ihre entsprechenden Fremdschlüssel.
- Da jede Entität vom Typ F nur mit höchstens einer Entität vom Type E verbunden sein kann, gibt es eine `UNIQUE`-Einschränkung auf Spalte `fkfid`.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3)  -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2)  -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22     ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Die zweite, `fkfid`, speichert die Werte der Primärschlüssel `fid` der F-Entitäten.
- Da wir nur Paare speichern, die existieren, sind beide Spalten `NOT NULL` und haben `REFERENCES`-Einschränkungen auf ihre entsprechenden Fremdschlüssel.
- Da jede Entität vom Typ F nur mit höchstens einer Entität vom Type E verbunden sein kann, gibt es eine `UNIQUE`-Einschränkung auf Spalte `fkfid`.
- Wir machen sie auch zum Primärschlüssel der Tabelle `relate_e_and_f`.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkeid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22 ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Mit Drei Tabellen

- Da jede Entität vom Typ F nur mit höchstens einer Entität vom Type E verbunden sein kann, gibt es eine **UNIQUE**-Einschränkung auf Spalte **fkfid**.
- Wir machen sie auch zum Primärschlüssel der Tabelle **relate_e_and_f**.
- Sie muss der Primärschlüssel sein, denn die Werte in Spalte **fkfid** müssen nicht einmalig sein ... eine Entität vom Typ E kann mit vielen Entitäten vom Typ F verbunden sein.

```
1  /* Create the tables for an E-|o-----o<-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     y   CHAR(2) -- example of other attributes
13 );
14
15 -- The table for managing the relationship between E and F.
16 CREATE TABLE relate_e_and_f (
17     fkfid INT NOT NULL REFERENCES e (eid),
18     fkfid INT NOT NULL UNIQUE PRIMARY KEY REFERENCES f (fid)
19 );
20
21 $ psql "postgres://postgres:XXX@localhost/relationships" -v
22 ↪ ON_ERROR_STOP=1 -ebf EF_1_tables.sql
23 CREATE TABLE
24 CREATE TABLE
25 CREATE TABLE
26 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the E and F rows.
10 INSERT INTO relate_e_and_f (fkeid, fkfid) VALUES (1, 1), (1, 2), (3, 4);
11
12 -- Combine the rows from E and F. This needs two INNER JOINS.
13 SELECT eid, x, fid, y FROM relate_e_and_f
14         INNER JOIN e ON e.eid = relate_e_and_f.fkeid
15         INNER JOIN f ON f.fid = relate_e_and_f.fkfid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  eid | x | fid | y
6  -----+-----+-----+-----
7  1 | 123 | 1 | AB
8  1 | 123 | 2 | CD
9  3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Weil beide Enden der Beziehung optional sind, können wir erstmal die Tabellen `e` und `f` befüllen.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the E and F rows.
10 INSERT INTO relate_e_and_f (fkeid, fkfid) VALUES (1, 1), (1, 2), (3, 4);
11
12 -- Combine the rows from E and F. This needs two INNER JOINS.
13 SELECT eid, x, fid, y FROM relate_e_and_f
14         INNER JOIN e ON e.eid = relate_e_and_f.fkeid
15         INNER JOIN f ON f.fid = relate_e_and_f.fkfid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  eid | x | fid | y
6  ----+-----+-----+----
7    1 | 123 | 1 | AB
8    1 | 123 | 2 | CD
9    3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir füllen nun einige Daten in die Tabellen.
- Weil beide Enden der Beziehung optional sind, können wir erstmal die Tabellen `e` und `f` befüllen.
- Dann erstellen wir die Beziehungen, in dem wir Zeilen in die Tabelle `relate_e_and_f` einfügen.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9  -- Create the relationships between the E and F rows.
10 INSERT INTO relate_e_and_f (fkeid, fkfid) VALUES (1, 1), (1, 2), (3, 4);
11
12 -- Combine the rows from E and F. This needs two INNER JOINS.
13 SELECT eid, x, fid, y FROM relate_e_and_f
14         INNER JOIN e ON e.eid = relate_e_and_f.fkeid
15         INNER JOIN f ON f.fid = relate_e_and_f.fkfid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_1_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  INSERT 0 3
5  eid | x | fid | y
6  ----+-----+-----+----
7    1 | 123 | 1 | AB
8    1 | 123 | 2 | CD
9    3 | 789 | 4 | GH
10 (3 rows)
11
12 # psql 16.11 succeeded with exit code 0.
```

Der Inhalt der Drei Tabellen



- Dies sind die Daten in den drei Tabellen.

Table e	
eid	x
1	„123“
2	„456“
3	„789“
4	„101“

Table f	
fid	y
1	„AB“
2	„CD“
3	„EF“
4	„GH“

Table relate_e_and_f	
fkeid	fkfid
1	1
1	2
3	4

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.
- Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen E und F Entitäten gibt.

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.
- Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen E und F Entitäten gibt.
- Wenn fast alle Entitäten vom Typ F mit einer Entität vom Typ E verbunden sind, dann ist die Tabelle `relate_e_and_f` fast genauso groß wie Tabelle `f`.

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.
- Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen E und F Entitäten gibt.
- Wenn fast alle Entitäten vom Typ F mit einer Entität vom Typ E verbunden sind, dann ist die Tabelle `relate_e_and_f` fast genauso groß wie Tabelle `f`.
- Anstatt einfach die Schlüssel der verbundenen E-Entitäten mit in Tabelle `f` zu speichern, haben wir dann eine weitere Tabelle, die diese Schlüssel *und* die Primärschlüssel von Tabelle `f` speichert.

Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [60] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.
- Der Ansatz hat nur dann Sinn, wenn es relativ wenig Paare von verbundenen E und F Entitäten gibt.
- Wenn fast alle Entitäten vom Typ F mit einer Entität vom Typ E verbunden sind, dann ist die Tabelle `relate_e_and_f` fast genauso groß wie Tabelle `f`.
- Anstatt einfach die Schlüssel der verbundenen E-Entitäten mit in Tabelle `f` zu speichern, haben wir dann eine weitere Tabelle, die diese Schlüssel *und* die Primärschlüssel von Tabelle `f` speichert.
- Und wir brauchen auch zwei `INNER JOIN`s.

Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.



Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.
- Dazu löschen wir erstmal wieder unsere Tabellen.

```
1  /* Drop the tables for the E-o---o<-F relationship. */
2
3  DROP TABLE IF EXISTS relate_e_and_f;
4  DROP TABLE IF EXISTS e;
5  DROP TABLE IF EXISTS f;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_cleanup.sql
2  DROP TABLE
3  DROP TABLE
4  DROP TABLE
5  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.
- Dazu löschen wir erstmal wieder unsere Tabellen.
- Diesmal müssen wir eine Fremdschlüssel-Spalte `fkeid` zur Tabelle `f` hinzufügen.

```
1  /* Create the tables for an E-to-many-to-many relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid   INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y     CHAR(2) -- example of other attributes
14 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.
- Dazu löschen wir erstmal wieder unsere Tabellen.
- Diesmal müssen wir eine Fremdschlüssel-Spalte `fkeid` zur Tabelle `f` hinzufügen.
- Diese Spalte bekommt eine `REFERENCES`-Einschränkung, die auf Spalte `eid` der Tabelle `e` zeigt.

```
1  /* Create the tables for an E-to-many-to-many relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid   INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y     CHAR(2) -- example of other attributes
14 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.
- Dazu löschen wir erstmal wieder unsere Tabellen.
- Diesmal müssen wir eine Fremdschlüssel-Spalte `fkeid` zur Tabelle `f` hinzufügen.
- Diese Spalte bekommt eine `REFERENCES`-Einschränkung, die auf Spalte `eid` der Tabelle `e` zeigt.
- Ein Wert in dieser Spalte darf `NULL` sein, denn nicht alle Zeilen in Tabelle `f` müssen mit Zeilen in Tabelle `e` verbunden sein.

```
1  /* Create the tables for an E-to-many-to-F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y    CHAR(2) -- example of other attributes
14 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Dazu löschen wir erstmal wieder unsere Tabellen.
- Diesmal müssen wir eine Fremdschlüssel-Spalte `fkeid` zur Tabelle `f` hinzufügen.
- Diese Spalte bekommt eine `REFERENCES`-Einschränkung, die auf Spalte `eid` der Tabelle `e` zeigt.
- Ein Wert in dieser Spalte darf `NULL` sein, denn nicht alle Zeilen in Tabelle `f` müssen mit Zeilen in Tabelle `e` verbunden sein.
- Anders als in der $A \text{ } \oplus \text{ } B$ -Situation darf hier keine `UNIQUE`-Einschränkung verwendet werden.

```
1  /* Create the tables for an E-|o-----o->F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid   INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y     CHAR(2) -- example of other attributes
14 );
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Diesmal müssen wir eine Fremdschlüssel-Spalte **fkeid** zur Tabelle **f** hinzufügen.
- Diese Spalte bekommt eine **REFERENCES**-Einschränkung, die auf Spalte **eid** der Tabelle **e** zeigt.
- Ein Wert in dieser Spalte darf **NULL** sein, denn nicht alle Zeilen in Tabelle **f** müssen mit Zeilen in Tabelle **e** verbunden sein.
- Anders als in der $A \oplus B$ -Situation darf hier keine **UNIQUE**-Einschränkung verwendet werden.
- Jede Zeile von Tabelle **e** kann mit mehreren Zeilen in Tabelle **f** verbunden sein.

```
1  /* Create the tables for an E-|o-----o->F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y    CHAR(2) -- example of other attributes
14 );
15
16 $ psql "postgres://postgres:XXX@localhost/relationships" -v
17   ↪ ON_ERROR_STOP=1 --ef EF_2_tables.sql
18
19 CREATE TABLE
20 CREATE TABLE
21 # psql 16.11 succeeded with exit code 0.
```

Zwei-Tabellen Lösung

- Diese Spalte bekommt eine **REFERENCES**-Einschränkung, die auf Spalte **eid** der Tabelle **e** zeigt.
- Ein Wert in dieser Spalte darf **NULL** sein, denn nicht alle Zeilen in Tabelle **f** müssen mit Zeilen in Tabelle **e** verbunden sein.
- Anders als in der A $\rightarrow$ B-Situation darf hier keine **UNIQUE**-Einschränkung verwendet werden.
- Jede Zeile von Tabelle **e** kann mit mehreren Zeilen in Tabelle **f** verbunden sein.
- Die Primärschlüsselwerte von Tabelle **e** dürfen mehrmals als Fremdschlüsselwerte in Spalte **fkeid** in Tabelle **f** auftauchen.

```
1  /* Create the tables for an E-to-many F relationship. */
2
3  -- Table E: Each row in E is related to zero or one or many rows in F.
4  CREATE TABLE e (
5      eid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table F: Each row in F is related to zero or one row in E.
10 CREATE TABLE f (
11     fid   INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkeid INT REFERENCES e (eid), -- the foreign key to E, can be NULL.
13     y     CHAR(2) -- example of other attributes
14 );

```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
2  CREATE TABLE
3  CREATE TABLE
4  # psql 16.11 succeeded with exit code 0.

```

Befüllen mit Daten

- Wir speichern nun Daten in die beiden Tabellen.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                   ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6      1 | 123 | 1 | AB
7      1 | 123 | 2 | CD
8      3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir speichern nun Daten in die beiden Tabellen.
- Wir fügen zuerst Zeilen in die Tabelle `e` ein.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                   ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6      1 | 123 | 1 | AB
7      1 | 123 | 2 | CD
8      3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir speichern nun Daten in die beiden Tabellen.
- Wir fügen zuerst Zeilen in die Tabelle **e** ein.
- Dann fügen wir Zeilen in die Tabelle **f** ein.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                  ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↪ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6      1 | 123 | 1 | AB
7      1 | 123 | 2 | CD
8      3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir speichern nun Daten in die beiden Tabellen.
- Wir fügen zuerst Zeilen in die Tabelle `e` ein.
- Dann fügen wir Zeilen in die Tabelle `f` ein.
- Für jede dieser Zeilen können wir einen gültigen Fremdschlüsselwert `fkeyid` auf eine Zeile in Tabelle `e` (identifiziert vom Primärschlüssel `eid`) angeben.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeyid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                     ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeyid = e.eid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6      1 | 123 | 1 | AB
7      1 | 123 | 2 | CD
8      3 | 789 | 4 | GH
9  (3 rows)
```



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

Befüllen mit Daten

- Wir speichern nun Daten in die beiden Tabellen.
- Wir fügen zuerst Zeilen in die Tabelle **e** ein.
- Dann fügen wir Zeilen in die Tabelle **f** ein.
- Für jede dieser Zeilen können wir einen gültigen Fremdschlüsselwert **fkeid** auf eine Zeile in Tabelle **e** (identifiziert vom Primärschlüssel **eid**) angeben.
- Wenn wir das machen, dann ist die neue Zeile in Tabelle **f** mit einer existierenden Zeile in Tabelle **e** verbunden.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                  ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6  1 | 123 | 1 | AB
7  1 | 123 | 2 | CD
8  3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Wir fügen zuerst Zeilen in die Tabelle **e** ein.
- Dann fügen wir Zeilen in die Tabelle **f** ein.
- Für jede dieser Zeilen können wir einen gültigen Fremdschlüsselwert **fkeid** auf eine Zeile in Tabelle **e** (identifiziert vom Primärschlüssel **eid**) angeben.
- Wenn wir das machen, dann ist die neue Zeile in Tabelle **f** mit einer existierenden Zeile in Tabelle **e** verbunden.
- Wir können auch **NULL** als **fkeid** speichern.

```
1  /* Inserting data into the tables for the E-o-----o-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                  ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6  1 | 123 | 1 | AB
7  1 | 123 | 2 | CD
8  3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Dann fügen wir Zeilen in die Tabelle **f** ein.
- Für jede dieser Zeilen können wir einen gültigen Fremdschlüsselwert **fkeid** auf eine Zeile in Tabelle **e** (identifiziert vom Primärschlüssel **eid**) angeben.
- Wenn wir das machen, dann ist die neue Zeile in Tabelle **f** mit einer existierenden Zeile in Tabelle **e** verbunden.
- Wir können auch **NULL** als **fkeid** speichern.
- Dann ist die Zeile in Tabelle **f** nicht mit einer Zeile in Tabelle **e** verbunden.

```
1  /* Inserting data into the tables for the E-o-----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                  ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6  1 | 123 | 1 | AB
7  1 | 123 | 2 | CD
8  3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Befüllen mit Daten

- Für jede dieser Zeilen können wir einen gültigen Fremdschlüsselwert `fkeid` auf eine Zeile in Tabelle `e` (identifiziert vom Primärschlüssel `eid`) angeben.
- Wenn wir das machen, dann ist die neue Zeile in Tabelle `f` mit einer existierenden Zeile in Tabelle `e` verbunden.
- Wir können auch `NULL` als `fkeid` speichern.
- Dann ist die Zeile in Tabelle `f` nicht mit einer Zeile in Tabelle `e` verbunden.
- Mit einem einzelnen `INNER JOIN` können wir die Daten wieder zusammensetzen.

```
1  /* Inserting data into the tables for the E-o----o<-F relationship. */
2
3  -- Insert some rows into the table for entity type E.
4  INSERT INTO e (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6  -- Insert some rows into the table for entity type F.
7  INSERT INTO f (y, fkeid) VALUES ('AB', 1), ('CD', 1), ('EF', NULL),
8                                ('GH', 3);
9
10 -- Combine the rows from E and F. This needs one INNER JOIN.
11 SELECT eid, x, fid, y FROM e INNER JOIN f ON f.fkeid = e.eid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
2  INSERT 0 4
3  INSERT 0 4
4  eid | x | fid | y
5  ----+-----+-----+----
6  1 | 123 | 1 | AB
7  1 | 123 | 2 | CD
8  3 | 789 | 4 | GH
9  (3 rows)
10
11 # psql 16.11 succeeded with exit code 0.
```

Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

Table e	
eid	x
1	„123“
2	„456“
3	„789“
4	„101“

Table f		
fid	fkeid	y
1	1	„AB“
2	1	„CD“
3	NULL	„EF“
4	3	„GH“



G $\rightarrow$ H



$G \oplus H$

- Es gibt zwei Entitätstypen G und H.



$G \text{ --- } \bigcirc \text{ --- } \text{---} H$



- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.

$G \text{ --- } \bigcirc \text{ --- } \text{---} H$

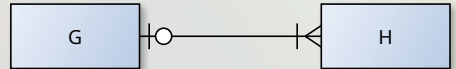
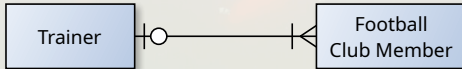


- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.

$G \text{ --- } \bigcirc \text{ --- } \text{---} \text{---} \text{---} H$



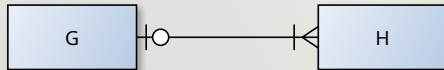
- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.
- Beispiel:



- ```

sequenceDiagram
 participant Trainer
 participant Football Club Member
 Trainer-->>Football Club Member:

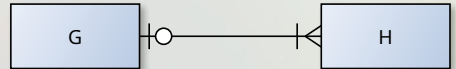
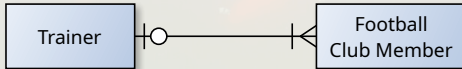
```



G  H



- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.
- Beispiel:
  - In einem Fußballklub trainiert jeder Trainer mehrere Mitglieder. Jedes Mitglied kann von einem Trainer trainiert werden oder auch gar nicht trainiert werden.



# Lösung

- Wir erstellen wieder zwei Tabellen und nennen sie diesmal **g** und **h**.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir erstellen wieder zwei Tabellen und nennen sie diesmal **g** und **h**.
- Sie haben die Primärschlüssel **gid** und **hid**, sowie die Beispielattribute **x** und **y**.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir erstellen wieder zwei Tabellen und nennen sie diesmal **g** und **h**.
- Sie haben die Primärschlüssel **gid** und **hid**, sowie die Beispielattribute **x** und **y**.
- Wir können die Situation so ähnlich anpacken wie die E  $\rightarrow$  F-Situation.

```
1 /* Create the tables for a G-|o----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir erstellen wieder zwei Tabellen und nennen sie diesmal **g** und **h**.
- Sie haben die Primärschlüssel **gid** und **hid**, sowie die Beispielattribute **x** und **y**.
- Wir können die Situation so ähnlich anpacken wie die E  $\rightarrow$  F-Situation.
- Allerdings ist es schwerer, dieses Beziehungsschema zu erzwingen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir erstellen wieder zwei Tabellen und nennen sie diesmal **g** und **h**.
- Sie haben die Primärschlüssel **gid** und **hid**, sowie die Beispielattribute **x** und **y**.
- Wir können die Situation so ähnlich anpacken wie die E  $\rightarrow$  F-Situation.
- Allerdings ist es schwerer, dieses Beziehungsschema zu erzwingen.
- Wir fangen damit an, Tabelle **g** vorzubereiten.

```
1 /* Create the tables for a G-|o----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Sie haben die Primärschlüssel `gid` und `hid`, sowie die Beispielattribute `x` und `y`.
- Wir können die Situation so ähnlich anpacken wie die E  $\rightarrow$  F-Situation.
- Allerdings ist es schwerer, dieses Beziehungsschema zu erzwingen.
- Wir fangen damit an, Tabelle `g` vorzubereiten.
- Jede Entität vom Typ G muss mit mindestens einer Entität vom Typ H in Beziehung stehen.

```
1 /* Create the tables for a G-to-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related to H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir können die Situation so ähnlich anpacken wie die E  $\text{---} \bigcirc \text{---} \bigcirc \text{---} \text{F}$ -Situation.
- Allerdings ist es schwerer, dieses Beziehungsschema zu erzwingen.
- Wir fangen damit an, Tabelle `g` vorzubereiten.
- Jede Entität vom Typ G muss mit mindestens einer Entität vom Typ H in Beziehung stehen.
- Wir lösen das Problem in zwei Schritten.

```
1 /* Create the tables for a G-|o----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Allerdings ist es schwerer, dieses Beziehungsschema zu erzwingen.
- Wir fangen damit an, Tabelle **g** vorzubereiten.
- Jede Entität vom Typ G muss mit mindestens einer Entität vom Typ H in Beziehung stehen.
- Wir lösen das Problem in zwei Schritten.
- Erst erzwingen wir, dass jede Zeile in Tabelle **g** mit einer Zeile in Tabelle **h** verbunden ist.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)

```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.

```

# Lösung

- Wir fangen damit an, Tabelle **g** vorzubereiten.
- Jede Entität vom Typ G muss mit mindestens einer Entität vom Typ H in Beziehung stehen.
- Wir lösen das Problem in zwei Schritten.
- Erst erzwingen wir, dass jede Zeile in Tabelle **g** mit einer Zeile in Tabelle **h** verbunden ist.
- Später erlauben wir, dass sie mit mehr als einer Zeile verbunden ist.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Jede Entität vom Typ G muss mit mindestens einer Entität vom Typ H in Beziehung stehen.
- Wir lösen das Problem in zwei Schritten.
- Erst erzwingen wir, dass jede Zeile in Tabelle `g` mit einer Zeile in Tabelle `h` verbunden ist.
- Später erlauben wir, dass sie mit mehr als einer Zeile verbunden ist.
- Wir fangen also mit dem Hinzufügen eines Attributs `fkhid` an, welches `NOT NULL` sein muss.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir lösen das Problem in zwei Schritten.
- Erst erzwingen wir, dass jede Zeile in Tabelle `g` mit einer Zeile in Tabelle `h` verbunden ist.
- Später erlauben wir, dass sie mit mehr als einer Zeile verbunden ist.
- Wir fangen also mit dem Hinzufügen eines Attributs `fkhid` an, welches `NOT NULL` sein muss.
- Diese Spalte sollte immer auf die Zeile in Tabelle `h` mit dem selben Wert in `hid` zeigen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.

```

# Lösung

- Erst erzwingen wir, dass jede Zeile in Tabelle **g** mit einer Zeile in Tabelle **h** verbunden ist.
- Später erlauben wir, dass sie mit mehr als einer Zeile verbunden ist.
- Wir fangen also mit dem Hinzufügen eines Attributs **fkhid** an, welches **NOT NULL** sein muss.
- Diese Spalte sollte immer auf die Zeile in Tabelle **h** mit dem selben Wert in **hid** zeigen.
- Später setzen wir eine passende **REFERENCE**-Einschränkung über **ALTER TABLE** (noch können wir als nicht, es gibt die Tabelle **h** ja noch nicht).

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir fangen also mit dem Hinzufügen eines Attributs `fkhid` an, welches **NOT NULL** sein muss.
- Diese Spalte sollte immer auf die Zeile in Tabelle `h` mit dem selben Wert in `hid` zeigen.
- Später setzen wir eine passende **REFERENCE**-Einschränkung über **ALTER TABLE** (noch können wir es nicht, es gibt die Tabelle `h` ja noch nicht).
- Stellen Sie sich erstmal vor, wir hätten das schon gemacht und dass das Attribut immer eine Zeile in Tabelle `h` referenziert.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- Diese Spalte sollte immer auf die Zeile in Tabelle `h` mit dem selben Wert in `hid` zeigen.
- Später setzen wir eine passende **REFERENCE**-Einschränkung über **ALTER TABLE** (noch können wir als nicht, es gibt die Tabelle `h` ja noch nicht).
- Stellen Sie sich erstmal vor, wir hätten das schon gemacht und dass das Attribut immer eine Zeile in Tabelle `h` referenziert.
- Über dieses Attribut `fkhid` soll immer die „erste“ H-Entität referenzieren, zu der die Zeile in Tabelle `g` in Beziehung steht.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Später setzen wir eine passende **REFERENCE**-Einschränkung über **ALTER TABLE** (noch können wir das nicht, es gibt die Tabelle **h** ja noch nicht).
- Stellen Sie sich erstmal vor, wir hätten das schon gemacht und dass das Attribut immer eine Zeile in Tabelle **h** referenziert.
- Über dieses Attribut **fkhid** soll immer die „erste“ H-Entität referenzieren, zu der die Zeile in Tabelle **g** in Beziehung steht.
- Wir machen diese Spalte **UNIQUE**, denn keine Zeile in Tabelle **h** kann mit mehr als einer Zeile in Tabelle **g** verbunden sein.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Später setzen wir eine passende **REFERENCE**-Einschränkung über **ALTER TABLE** (noch können wir das nicht, es gibt die Tabelle **h** ja noch nicht).
- Stellen Sie sich erstmal vor, wir hätten das schon gemacht und dass das Attribut immer eine Zeile in Tabelle **h** referenziert.
- Über dieses Attribut **fkhid** soll immer die „erste“ H-Entität referenzieren, zu der die Zeile in Tabelle **g** in Beziehung steht.
- Wir machen diese Spalte **UNIQUE**, denn keine Zeile in Tabelle **h** kann mit mehr als einer Zeile in Tabelle **g** verbunden sein.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Über dieses Attribut `fkhid` soll immer die „erste“ H-Entität referenzieren, zu der die Zeile in Tabelle `g` in Beziehung steht.
- Wir machen diese Spalte `UNIQUE`, denn keine Zeile in Tabelle `h` kann mit mehr als einer Zeile in Tabelle `g` verbunden sein.
- Deshalb kann kein Primärschlüsselwert `hid` mehr als einmal in der Spalte `fkhid` auftauchen.
- Nun bearbeiten wir die Tabelle für den Entitätstyp H.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.

```

# Lösung

- Wir machen diese Spalte **UNIQUE**, denn keine Zeile in Tabelle **h** kann mit mehr als einer Zeile in Tabelle **g** verbunden sein.
- Deshalb kann kein Primärschlüsselwert **hid** mehr als einmal in der Spalte **fkhid** auftauchen.
- Nun bearbeiten wir die Tabelle für den Entitätstyp H.
- Weil jede Entität vom Typ H mit keiner oder einer Entität vom Typ G in Verbindung steht, fügen wir die Spalte **fkgid** zur Tabelle **h** hinzu.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- Wir machen diese Spalte **UNIQUE**, denn keine Zeile in Tabelle **h** kann mit mehr als einer Zeile in Tabelle **g** verbunden sein.
- Deshalb kann kein Primärschlüsselwert **hid** mehr als einmal in der Spalte **fkhid** auftauchen.
- Nun bearbeiten wir die Tabelle für den Entitätstyp H.
- Weil jede Entität vom Typ H mit keiner oder einer Entität vom Typ G in Verbindung steht, fügen wir die Spalte **fkgid** zur Tabelle **h** hinzu.
- Die Werte in dieser Spalte können **NULL** sein.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Deshalb kann kein Primärschlüsselwert `hid` mehr als einmal in der Spalte `fkhid` auftauchen.
- Nun bearbeiten wir die Tabelle für den Entitätstyp H.
- Weil jede Entität vom Typ H mit keiner oder einer Entität vom Typ G in Verbindung steht, fügen wir die Spalte `fkgid` zur Tabelle `h` hinzu.
- Die Werte in dieser Spalte können `NULL` sein.
- In diesem Fall ist die entsprechende Zeile in `h` mit keiner Entität vom Typ G verbunden.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- Nun bearbeiten wir die Tabelle für den Entitätstyp H.
- Weil jede Entität vom Typ H mit keiner oder einer Entität vom Typ G in Verbindung steht, fügen wir die Spalte `fkgid` zur Tabelle `h` hinzu.
- Die Werte in dieser Spalte können `NULL` sein.
- In diesem Fall ist die entsprechende Zeile in `h` mit keiner Entität vom Typ G verbunden.
- Wenn der Wert nicht `NULL` ist, dann muss es ein passender Fremdschlüsselverweis auf eine Zeile in Tabelle `g` sein.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkfid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Die Werte in dieser Spalte können **NULL** sein.
- In diesem Fall ist die entsprechende Zeile in **h** mit keiner Entität vom Typ **G** verbunden.
- Wenn der Wert nicht **NULL** ist, dann muss es ein passender Fremdschlüsselverweis auf eine Zeile in Tabelle **g** sein.
- Die Werte in dieser Spalte brauchen nicht **UNIQUE** zu sein, weil mehrere Entitäten vom Typ **H** mit der selben Entität vom Typ **G** in Verbindung stehen können.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- In diesem Fall ist die entsprechende Zeile in **h** mit keiner Entität vom Typ G verbunden.
- Wenn der Wert nicht **NULL** ist, dann muss es ein passender Fremdschlüsselverweis auf eine Zeile in Tabelle **g** sein.
- Die Werte in dieser Spalte brauchen nicht **UNIQUE** zu sein, weil mehrere Entitäten vom Typ H mit der selben Entität vom Typ G in Verbindung stehen können.
- Jetzt haben wir sichergestellt, dass jede Entität vom Typ H mit keiner oder genau einer Entität vom Typ G verbunden ist.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Die Werte in dieser Spalte brauchen nicht **UNIQUE** zu sein, weil mehrere Entitäten vom Typ H mit der selben Entität vom Typ G in Verbindung stehen können.
- Jetzt haben wir sichergestellt, dass jede Entität vom Typ H mit keiner oder genau einer Entität vom Typ G verbunden ist.
- Wir könnten jetzt eine Einschränkung **g\_fkhid\_fk** als **FOREIGN KEY (fkhid) REFERENCES h (hid)** zur Tabelle **g** mit **ALTER TABLE g ADD CONSTRAINT...** hinzufügen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Jetzt haben wir sichergestellt, dass jede Entität vom Typ H mit keiner oder genau einer Entität vom Typ G verbunden ist.
- Wir könnten jetzt eine Einschränkung `g_fkhid_fk` als `FOREIGN KEY (fkhid) REFERENCES h (hid)` zur Tabelle `g` mit `ALTER TABLE g ADD CONSTRAINT...` hinzufügen.
- Das würde erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden sein muss.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir könnten jetzt eine Einschränkung `g_fkhid_fk` als `FOREIGN KEY (fkhid) REFERENCES h (hid)` zur Tabelle `g` mit `ALTER TABLE g ADD CONSTRAINT...` hinzufügen.
- Das würde erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden sein muss.
- Damit stellen wir aber *nicht* sicher, dass wenn eine Entität vom Typ G mit einer Entität vom Typ H verbunden ist (als ihre „erste“ H-Entität) ... dass diese H-Entität auch mit der G-Entität verbunden ist.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir könnten jetzt eine Einschränkung `g_fkhid_fk` als `FOREIGN KEY (fkhid) REFERENCES h (hid)` zur Tabelle `g` mit `ALTER TABLE g ADD CONSTRAINT...` hinzufügen.
- Das würde erzwingen, dass jede Entität vom Typ `G` mit mindestens einer Entität vom Typ `H` verbunden sein muss.
- Damit stellen wir aber *nicht* sicher, dass wenn eine Entität vom Typ `G` mit einer Entität vom Typ `H` verbunden ist (als ihre „erste“ `H`-Entität) ... dass diese `H`-Entität auch mit der `G`-Entität verbunden ist.
- Wir können das auf eine etwas komische Art sicherstellen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir könnten jetzt eine Einschränkung `g_fkhid_fk` als `FOREIGN KEY (fkhid) REFERENCES h (hid)` zur Tabelle `g` mit `ALTER TABLE g ADD CONSTRAINT...` hinzufügen.
- Das würde erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden sein muss.
- Damit stellen wir aber *nicht* sicher, dass wenn eine Entität vom Typ G mit einer Entität vom Typ H verbunden ist (als ihre „erste“ H-Entität) ... dass diese H-Entität auch mit der G-Entität verbunden ist.
- Wir können das auf eine etwas komische Art sicherstellen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Damit stellen wir aber *nicht* sicher, dass wenn eine Entität vom Typ G mit einer Entität vom Typ H verbunden ist (als ihre „erste“ H-Entität) ... dass diese H-Entität auch mit der G-Entität verbunden ist.
- Wir können das auf eine etwas komische Art sicherstellen.
- Wir erstellen eine **REFERENCES**-Einschränkung nicht auf eine einzelne Spalte **hid**...
- ...sondern wir erzwingen, dass das Paar (**fkhid**, **gid**) in Tabelle **g** genauso als Paar (**hid**, **fkgid**) in Tabelle **h** auftauchen muss!

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir können das auf eine etwas komische Art sicherstellen.
- Wir erstellen eine REFERENCES-Einschränkung nicht auf eine einzelne Spalte hid...
- ...sondern wir erzwingen, dass das Paar (fkhid, gid) in Tabelle g genauso als Paar (hid, fkgid) in Tabelle h auftauchen muss!
- Wir wissen, dass jeder Wert von gid nur einmal in Tabelle g existiert.

```
1 /* Create the tables for a G-to-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related to >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir können das auf eine etwas komische Art sicherstellen.
- Wir erstellen eine REFERENCES-Einschränkung nicht auf eine einzelne Spalte hid...
- ...sondern wir erzwingen, dass das Paar (fkhid, gid) in Tabelle g genauso als Paar (hid, fkgid) in Tabelle h auftauchen muss!
- Wir wissen, dass jeder Wert von gid nur einmal in Tabelle g existiert.
- Wir wissen auch, dass jeder Wert von hid nur einmal in Tabelle h existiert.

```
1 /* Create the tables for a G-to-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related to H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

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

# Lösung

- ...sondern wir erzwingen, dass das Paar (fkhid, gid) in Tabelle g genauso als Paar (hid, fkgid) in Tabelle h auftauchen muss!
- Wir wissen, dass jeder Wert von gid nur einmal in Tabelle g existiert.
- Wir wissen auch, dass jeder Wert von hid nur einmal in Tabelle h existiert.
- Der erste Teil des Spaltenpaares in der Einschränkung – fkhid oder von der anderen Seite kommend, hid – selektiert also immer eine eindeutige Zeile in Tabelle h.

```
1 /* Create the tables for a G-to-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related to >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir wissen, dass jeder Wert von `gid` nur einmal in Tabelle `g` existiert.
- Wir wissen auch, dass jeder Wert von `hid` nur einmal in Tabelle `h` existiert.
- Der erste Teil des Spaltenpaares in der Einschränkung – `fkhid` oder von der anderen Seite kommend, `hid` – selektiert also immer eine eindeutige Zeile in Tabelle `h`.
- Es kann niemals eine andere Zeile mit dem selben `hid`-Wert geben, denn das ist der Primärschlüssel der Tabelle `h`.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir wissen auch, dass jeder Wert von `hid` nur einmal in Tabelle `h` existiert.
- Der erste Teil des Spaltenpaares in der Einschränkung – `fkhid` oder von der anderen Seite kommend, `hid` – selektiert also immer eine eindeutige Zeile in Tabelle `h`.
- Es kann niemals eine andere Zeile mit dem selben `hid`-Wert geben, denn das ist der Primärschlüssel der Tabelle `h`.
- Das zweite Element des Paares – also `gid` oder von der anderen Seite kommend, `fkgid`) – erzwingt darum, dass diese einzelne Zeile in Tabelle `h` den passenden Wert `gid` in `fkgid` gespeichert hat.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Wir wissen auch, dass jeder Wert von `hid` nur einmal in Tabelle `h` existiert.
- Der erste Teil des Spaltenpaares in der Einschränkung – `fkhid` oder von der anderen Seite kommend, `hid` – selektiert also immer eine eindeutige Zeile in Tabelle `h`.
- Es kann niemals eine andere Zeile mit dem selben `hid`-Wert geben, denn das ist der Primärschlüssel der Tabelle `h`.
- Das zweite Element des Paares – also `gid` oder von der anderen Seite kommend, `fkgid`) – erzwingt darum, dass diese einzelne Zeile in Tabelle `h` den passenden Wert `gid` in `fkgid` gespeichert hat.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Es kann niemals eine andere Zeile mit dem selben `hid`-Wert geben, denn das ist der Primärschlüssel der Tabelle `h`.
- Das zweite Element des Paares – also `gid` oder von der anderen Seite kommend, `fkgid`) – erzwingt darum, dass diese einzelne Zeile in Tabelle `h` den passenden Wert `gid` in `fkgid` gespeichert hat.
- Weil Fremdschlüssel-**REFERENCES**-Einschränkungen nur Spalten referenzieren können, die **UNIQUE** sind, müssen wir die Einschränkung **UNIQUE (hid, fkgid)** zur Tabelle `h` hinzufügen.
- Überlegen wir uns das noch einmal.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Weil Fremdschlüssel-**REFERENCES**-Einschränkungen nur Spalten referenzieren können, die **UNIQUE** sind, müssen wir die Einschränkung **UNIQUE (hid, fkgid)** zur Tabelle **h** hinzufügen.
- Überlegen wir uns das noch einmal.
- Wir haben die Einschränkung **g\_fkhid\_gid\_fk** als **FOREIGN KEY (fkhid, gid) REFERENCES h (hid, fkgid)**.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Überlegen wir uns das noch einmal.
- Wir haben die Einschränkung `g_fkhid_gid_fk` als `FOREIGN KEY (fkhid, gid) REFERENCES h (hid, fkgid)`.
- Auf der Seite der Tabelle `g` schaut diese Einschränkung auf eine Zeile und nimmt den Wert des Fremdschlüssels `fkhid` zur Tabelle `h` zusammen mit dem eigenen Primärschlüssel `gid` als ein Tuple `(fkhid, gid)`.

```
1 /* Create the tables for a G-|o----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Überlegen wir uns das noch einmal.
- Wir haben die Einschränkung `g_fkhid_gid_fk` als `FOREIGN KEY (fkhid, gid) REFERENCES h (hid, fkgid)`.
- Auf der Seite der Tabelle `g` schaut diese Einschränkung auf eine Zeile und nimmt den Wert des Fremdschlüssels `fkhid` zur Tabelle `h` zusammen mit dem eigenen Primärschlüssel `gid` als ein Tuple `(fkhid, gid)`.
- Auf Seiten der Tabelle `h` muss es ein passendes Tupel mit dem Primärschlüssel `hid` und dem dazugehörigen Wert des Fremdschlüssels `fkgid` geben.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Auf der Seite der Tabelle `g` schaut diese Einschränkung auf eine Zeile und nimmt den Wert des Fremdschlüssels `fkhid` zur Tabelle `h` zusammen mit dem eigenen Primärschlüssel `gid` als ein Tuple `(fkhid, gid)`.
- Auf Seiten der Tabelle `h` muss es ein passendes Tupel mit dem Primärschlüssel `hid` und dem dazugehörigen Wert des Fremdschlüssels `fkgid` geben.
- Es wird also erzwungen, dass die Paare `(fkhi, gid) == (hid, fkgid)` immer passen.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- Auf der Seite der Tabelle `g` schaut diese Einschränkung auf eine Zeile und nimmt den Wert des Fremdschlüssels `fkhid` zur Tabelle `h` zusammen mit dem eigenen Primärschlüssel `gid` als ein Tuple `(fkhid, gid)`.
- Auf Seiten der Tabelle `h` muss es ein passendes Tupel mit dem Primärschlüssel `hid` und dem dazugehörigen Wert des Fremdschlüssels `fkgid` geben.
- Es wird also erzwungen, dass die Paare `(fkhi, gid) == (hid, fkgid)` immer passen.
- Natürlich sind die Primärschlüsselwerte beider Tabellen immer einzigartig.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Auf Seiten der Tabelle `h` muss es ein passendes Tupel mit dem Primärschlüssel `hid` und dem dazugehörigen Wert des Fremdschlüssels `fkgid` geben.
- Es wird also erzwungen, dass die Paare `(fkhi, gid) == (hid, fkgid)` immer passen.
- Natürlich sind die Primärschlüsselwerte beider Tabellen immer einzigartig.
- Sagen wir, Tabelle `g` hat Primärschlüsselwert `gid=u` und Fremdschlüssel `fkhid=v`.

```
1 /* Create the tables for a G-|o----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Es wird also erzwungen, dass die Paare  $(fkhi, gid) == (hid, fkgid)$  immer passen.
- Natürlich sind die Primärschlüsselwerte beider Tabellen immer einzigartig.
- Sagen wir, Tabelle `g` hat Primärschlüsselwert `gid=u` und Fremdschlüssel `fkhid=v`.
- Dann muss die Zeile in Tabelle `h` mit Primärschlüsselwert `hid=v` den Fremdschlüssel `fkgid=u` haben.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Es wird also erzwungen, dass die Paare  $(fkhi, gid) == (hid, fkgid)$  immer passen.
- Natürlich sind die Primärschlüsselwerte beider Tabellen immer einzigartig.
- Sagen wir, Tabelle `g` hat Primärschlüsselwert `gid=u` und Fremdschlüssel `fkhid=v`.
- Dann muss die Zeile in Tabelle `h` mit Primärschlüsselwert `hid=v` den Fremdschlüssel `fkgid=u` haben.
- Natürlich kann es auch andere Zeilen in Tabelle `h` geben, die auch den Fremdschlüssel `fkgid=u` haben.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Natürlich sind die Primärschlüsselwerte beider Tabellen immer einzigartig.
- Sagen wir, Tabelle `g` hat Primärschlüsselwert `gid=u` und Fremdschlüssel `fkhid=v`.
- Dann muss die Zeile in Tabelle `h` mit Primärschlüsselwert `hid=v` den Fremdschlüssel `fkgid=u` haben.
- Natürlich kann es auch andere Zeilen in Tabelle `h` geben, die auch den Fremdschlüssel `fkgid=u` haben.
- Das ist OK, denn mehrere Entitäten vom Typ `H` können die selbe Entität vom Typ `G` referenzieren.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)
```

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

# Lösung

- Sagen wir, Tabelle **g** hat Primärschlüsselwert **gid=u** und Fremdschlüssel **fkhid=v**.
- Dann muss die Zeile in Tabelle **h** mit Primärschlüsselwert **hid=v** den Fremdschlüssel **fkgid=u** haben.
- Natürlich kann es auch andere Zeilen in Tabelle **h** geben, die auch den Fremdschlüssel **fkgid=u** haben.
- Das ist OK, denn mehrere Entitäten vom Typ H können die selbe Entität vom Typ G referenzieren.
- Jetzt haben wir also diese Beziehung exakt implementiert.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkfid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkfid_gid_fk FOREIGN KEY (fkfid, gid)
22 REFERENCES h (hid, fkgid)

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_tables.sql
2 CREATE TABLE
3 CREATE TABLE
4 ALTER TABLE
5 # psql 16.11 succeeded with exit code 0.
```

# Lösung

- Dann muss die Zeile in Tabelle `h` mit Primärschlüsselwert `hid=v` den Fremdschlüssel `fkgid=u` haben.
- Natürlich kann es auch andere Zeilen in Tabelle `h` geben, die auch den Fremdschlüssel `fkgid=u` haben.
- Das ist OK, denn mehrere Entitäten vom Typ `H` können die selbe Entität vom Typ `G` referenzieren.
- Jetzt haben wir also diese Beziehung exakt implementiert.
- Schauen wir uns mal an, was wir gemacht haben.

```
1 /* Create the tables for a G-|o-----|<-H relationship. */
2
3 -- Table G: Each row in G is related to one or multiple rows in H.
4 -- We force that that row in H is also related to our row in G via
5 -- a foreign key constraint g_h_fk.
6 CREATE TABLE g (
7 gid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
8 fkhid INT NOT NULL UNIQUE, -- row is related >= 1 H.
9 x CHAR(3) -- example of other attributes
10);
11
12 -- Table H: Each row in H is related to zero or one rows in G.
13 CREATE TABLE h (
14 hid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
15 fkgid INT REFERENCES g (gid), -- can be related to 0 or 1 G
16 y CHAR(2), -- example of attributes
17 UNIQUE (hid, fkgid) -- needed for g_fkhid_gid_fk
18);
19
20 -- To table G, we add the foreign key reference constraint towards H.
21 ALTER TABLE g ADD CONSTRAINT g_fkhid_gid_fk FOREIGN KEY (fkhid, gid)
22 REFERENCES h (hid, fkgid)

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

# Testen der Einschränkungen

- Probieren wir mal aus, was wir da gemacht haben.



# Testen der Einschränkungen

- Probieren wir mal aus, was wir da gemacht haben.
- Können wir eine Zeile in Tabelle `g` einfügen, die nicht mit einer Zeile in Tabelle `h` in Beziehung steht?

```
1 /* Can we create a row in G unrelated to any row in H? */
```

```
2
3 INSERT INTO g (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
```

```
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_1.sql
```

```
2 psql:conceptualToRelational/GH_insert_error_1.sql:3: ERROR: null value
```

```
 ↳ in column "fkhid" of relation "g" violates not-null constraint
```

```
3 DETAIL: Failing row contains (1, null, 777).
```

```
4 psql:conceptualToRelational/GH_insert_error_1.sql:3: STATEMENT: /* Can
```

```
 ↳ we create a row in G unrelated to any row in H? */
```

```
5 INSERT INTO g (x) VALUES ('777');
```

```
6 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Probieren wir mal aus, was wir da gemacht haben.
- Können wir eine Zeile in Tabelle `g` einfügen, die nicht mit einer Zeile in Tabelle `h` in Beziehung steht?
- Nein, können wir nicht.

```
1 /* Can we create a row in G unrelated to any row in H? */
```

```
2
3 INSERT INTO g (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_1.sql
```

```
2 psql:conceptualToRelational/GH_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkhid" of relation "g" violates not-null constraint
```

```
3 DETAIL: Failing row contains (1, null, 777).
```

```
4 psql:conceptualToRelational/GH_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in G unrelated to any row in H? */
```

```
5 INSERT INTO g (x) VALUES ('777');
```

```
6 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Probieren wir mal aus, was wir da gemacht haben.
- Können wir eine Zeile in Tabelle `g` einfügen, die nicht mit einer Zeile in Tabelle `h` in Beziehung steht?
- Nein, können wir nicht.
- Damit wir eine Zeile in Tabelle `g` einfügen können, müssen wir den Fremdschlüssel `fkhid` angeben und die entsprechende Zeile in Tabelle `h` muss es geben.

```
1 /* Can we create a row in G unrelated to any row in H? */
```

```
3 INSERT INTO g (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_1.sql
```

```
2 psql:conceptualToRelational/GH_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkhid" of relation "g" violates not-null constraint
3 DETAIL: Failing row contains (1, null, 777).
```

```
4 psql:conceptualToRelational/GH_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in G unrelated to any row in H? */
```

```
5 INSERT INTO g (x) VALUES ('777');
```

```
6 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen



- Probieren wir mal aus, was wir da gemacht haben.
- Können wir eine Zeile in Tabelle `g` einfügen, die nicht mit einer Zeile in Tabelle `h` in Beziehung steht?
- Nein, können wir nicht.
- Damit wir eine Zeile in Tabelle `g` einfügen können, müssen wir den Fremdschlüssel `fkhid` angeben und die entsprechende Zeile in Tabelle `h` muss es geben.
- Können wir Zeilen in Tabelle `h` einfügen, die nicht mit Zeilen in Tabelle `g` in Beziehung stehen?

# Testen der Einschränkungen



- Können wir eine Zeile in Tabelle `g` einfügen, die nicht mit einer Zeile in Tabelle `h` in Beziehung steht?
- Nein, können wir nicht.
- Damit wir eine Zeile in Tabelle `g` einfügen können, müssen wir den Fremdschlüssel `fkhid` angeben und die entsprechende Zeile in Tabelle `h` muss es geben.
- Können wir Zeilen in Tabelle `h` einfügen, die nicht mit Zeilen in Tabelle `g` in Beziehung stehen?
- Ja, das würde gehen. Und das ist OK.

# Testen der Einschränkungen



- Nein, können wir nicht.
- Damit wir eine Zeile in Tabelle `g` einfügen können, müssen wir den Fremdschlüssel `fkhid` angeben und die entsprechende Zeile in Tabelle `h` muss es geben.
- Können wir Zeilen in Tabelle `h` einfügen, die nicht mit Zeilen in Tabelle `g` in Beziehung stehen?
- Ja, das würde gehen. Und das ist OK.
- Entitäten vom Typ H sind mit entweder einer oder keiner Entität vom Typ G verbunden.

# Testen der Einschränkungen

- Damit wir eine Zeile in Tabelle **g** einfügen können, müssen wir den Fremdschlüssel **fkhid** angeben und die entsprechende Zeile in Tabelle **h** muss es geben.
- Können wir Zeilen in Tabelle **h** einfügen, die nicht mit Zeilen in Tabelle **g** in Beziehung stehen?
- Ja, das würde gehen. Und das ist OK.
- Entitäten vom Typ H sind mit entweder einer oder keiner Entität vom Typ G verbunden.
- Aber können wir eine Zeile in Tabelle **g** einfügen, die eine Zeile in Tabelle **h** referenziert, die *nicht* mit einer Entität vom Typ G verbunden ist?

```
1 /* Can we create a row in G related to a row in H unrelated to any G? */
2
3 INSERT INTO g (fkhid, x) VALUES (6, '888');
4
5 $ psql "postgres://postgres:XXX@localhost/relationships" -v
6 ↪ ON_ERROR_STOP=1 -ebf GH_insert_error_2.sql
7 psql:conceptualToRelational/GH_insert_error_2.sql:3: ERROR: insert or
8 ↪ update on table "g" violates foreign key constraint "
9 ↪ g_fkhid_gid_fk"
10 DETAIL: Key (fkhid, gid)=(6, 2) is not present in table "h".
11 psql:conceptualToRelational/GH_insert_error_2.sql:3: STATEMENT: /* Can
12 ↪ we create a row in G related to a row in H unrelated to any G? */
13 INSERT INTO g (fkhid, x) VALUES (6, '888');
14 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Können wir Zeilen in Tabelle `h` einfügen, die nicht mit Zeilen in Tabelle `g` in Beziehung stehen?
- Ja, das würde gehen. Und das ist OK.
- Entitäten vom Typ H sind mit entweder einer oder keiner Entität vom Typ G verbunden.
- Aber können wir eine Zeile in Tabelle `g` einfügen, die eine Zeile in Tabelle `h` referenziert, die *nicht* mit einer Entität vom Typ G verbunden ist?
- Nein, denn die Einschränkung `g_fkhid_gid_fk` verlangt, dass die Zeile in Tabelle `h` zurück auf die Zeile in Tabelle `g` verweist.

```
1 /* Can we create a row in G related to a row in H unrelated to any G? */
2
3 INSERT INTO g (fkhid, x) VALUES (6, '888');
4
5 $ psql "postgres://postgres:XXX@localhost/relationships" -v
6 ↪ ON_ERROR_STOP=1 -ebf GH_insert_error_2.sql
7 psql:conceptualToRelational/GH_insert_error_2.sql:3: ERROR: insert or
8 ↪ update on table "g" violates foreign key constraint "
9 ↪ g_fkhid_gid_fk"
10 DETAIL: Key (fkhid, gid)=(6, 2) is not present in table "h".
11 psql:conceptualToRelational/GH_insert_error_2.sql:3: STATEMENT: /* Can
12 ↪ we create a row in G related to a row in H unrelated to any G? */
13 INSERT INTO g (fkhid, x) VALUES (6, '888');
14 # psql 16.11 failed with exit code 3.
```

# Wie können wir überhaupt Daten einfügen?

- Aber wie können wir überhaupt Zeilen in Tabelle `g` einfügen?



# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle `g` einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle `h` referenziert.

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.
- Was sie nicht tut.

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.
- Was sie nicht tut.
- Nun haben wir also unsere beiden Tabellen erzeugt und ihre referenzielle Integrität mit starken Einschränkungen geschützt. ...

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.
- Was sie nicht tut.
- Nun haben wir also unsere beiden Tabellen erzeugt und ihre referenzielle Integrität mit starken Einschränkungen geschützt. . .
- ... und die *müssen* ja auch so sein, denn wir wollen ja gerade  $G \nrightarrow \bigcirc \dashv \vdash H$  implementieren

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.
- Was sie nicht tut.
- Nun haben wir also unsere beiden Tabellen erzeugt und ihre referenzielle Integrität mit starken Einschränkungen geschützt. ...
- ... und die *müssen* ja auch so sein, denn wir wollen ja gerade  $G \dashv \circ \dashv \vdash H$  implementieren
- ... aber wie bekommen wir überhaupt Daten in Tabelle **g**?

# Wie können wir überhaupt Daten einfügen?



- Aber wie können wir überhaupt Zeilen in Tabelle **g** einfügen?
- Wie können keine Zeile einfügen, die keine existierende Zeile in Tabelle **h** referenziert.
- Wir können auch keine Zeile einfügen, die gar eine existierende Zeile in Tabelle **h** referenziert, die noch mit keiner Zeile in Tabelle **g** verbunden ist.
- Und wenn wir eine Zeile in Tabelle **g** einfügen wollen würden, die eine Zeile in Tabelle **h** referenziert welche bereits eine andere Zeile in Tabelle **q** referenziert ... dann müsste diese andere Zeile in Tabelle **g** erstmal existieren.
- Was sie nicht tut.
- Nun haben wir also unsere beiden Tabellen erzeugt und ihre referenzielle Integrität mit starken Einschränkungen geschützt. ...
- ... und die *müssen* ja auch so sein, denn wir wollen ja gerade  $G \nrightarrow H$  implementieren
- ... aber wie bekommen wir überhaupt Daten in Tabelle **g**?
- Wir müssen dieses komische Henne-Ei-Problem lösen.

# Befüllen mit Daten

- Wir fangen damit an, erstmal Zeilen in die Tabelle `h` zu schreiben.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir fangen damit an, erstmal Zeilen in die Tabelle `h` zu schreiben.
- Da die Entitäten vom Typ H nicht unbedingt mit denen vom Typ G in Beziehung stehen müssen, brauchen wir uns um die Einschränkungen keine Gedanken zu machen.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --bf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir fangen damit an, erstmal Zeilen in die Tabelle `h` zu schreiben.
- Da die Entitäten vom Typ H nicht unbedingt mit denen vom Typ G in Beziehung stehen müssen, brauchen wir uns um die Einschränkungen keine Gedanken zu machen.
- Wenn wir aber eine Entität vom Typ G in die Tabelle `g` einfügen, dann müssen wir **gleichzeitig** eine Beziehung zu einer H-Entität in der Tabelle `h` erstellen.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9 gid | x | hid | y
10 ---+---+---+---
11 3 | 123 | 1 | AB
12 4 | 456 | 3 | EF
13 5 | 789 | 4 | GH
14 3 | 123 | 2 | CD
15 3 | 123 | 5 | IJ
16 (5 rows)
```

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

# Befüllen mit Daten

- Wir fangen damit an, erstmal Zeilen in die Tabelle `h` zu schreiben.
- Da die Entitäten vom Typ H nicht unbedingt mit denen vom Typ G in Beziehung stehen müssen, brauchen wir uns um die Einschränkungen keine Gedanken zu machen.
- Wenn wir aber eine Entität vom Typ G in die Tabelle `g` einfügen, dann müssen wir **gleichzeitig** eine Beziehung zu einer H-Entität in der Tabelle `h` erstellen.
- Das klingt unmöglich, geht aber problemlos.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9 gid | x | hid | y
10 ---+---+---+---
11 3 | 123 | 1 | AB
12 4 | 456 | 3 | EF
13 5 | 789 | 4 | GH
14 3 | 123 | 2 | CD
15 3 | 123 | 5 | IJ
16 (5 rows)
```

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

# Befüllen mit Daten



- Da die Entitäten vom Typ H nicht unbedingt mit denen vom Typ G in Beziehung stehen müssen, brauchen wir uns um die Einschränkungen keine Gedanken zu machen.
- Wenn wir aber eine Entität vom Typ G in die Tabelle `g` einfügen, dann müssen wir **gleichzeitig** eine Beziehung zu einer H-Entität in der Tabelle `h` erstellen.
- Das klingt unmöglich, geht aber problemlos, weil:

*A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction...*

# Befüllen mit Daten

- Da die Entitäten vom Typ H nicht unbedingt mit denen vom Typ G in Beziehung stehen müssen, brauchen wir uns um die Einschränkungen keine Gedanken zu machen.
- Wenn wir aber eine Entität vom Typ G in die Tabelle `g` einfügen, dann müssen wir **gleichzeitig** eine Beziehung zu einer H-Entität in der Tabelle `h` erstellen.
- Das klingt unmöglich, geht aber problemlos, weil:
- In PostgreSQL und wahrscheinlich vielen anderen DBMSen werden Einschränkungen zur referentiellen Integrität am Ende einer Transaktion überprüft.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+---+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wenn wir aber eine Entität vom Typ G in die Tabelle `g` einfügen, dann müssen wir **gleichzeitig** eine Beziehung zu einer H-Entität in der Tabelle `h` erstellen.
- Das klingt unmöglich, geht aber problemlos, weil:
- In PostgreSQL und wahrscheinlich vielen anderen DBMSen werden Einschränkungen zur referentiellen Integrität am Ende einer Transaktion überprüft.
- Eine Transaktion kann mehrere Kommandos gruppieren, die dann entweder alle zusammen erfolgreich sind oder alle zusammen fehlschlagen (wobei dann die DB nicht verändert wird).

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --eof GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das klingt unmöglich, geht aber problemlos, weil:
- In PostgreSQL und wahrscheinlich vielen anderen DBMSen werden Einschränkungen zur referentiellen Integrität am Ende einer Transaktion überprüft.
- Eine Transaktion kann mehrere Kommandos gruppieren, die dann entweder alle zusammen erfolgreich sind oder alle zusammen fehlschlagen (wobei dann die DB nicht verändert wird).
- Ein einzelnes Kommando in PostgreSQL ist auch eine Transaktion<sup>70</sup>.

```
1 /* Inserting data into the tables for the G-o----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- In PostgreSQL und wahrscheinlich vielen anderen DBMSen werden Einschränkungen zur referentiellen Integrität am Ende einer Transaktion überprüft.
- Eine Transaktion kann mehrere Kommandos gruppieren, die dann entweder alle zusammen erfolgreich sind oder alle zusammen fehlschlagen (wobei dann die DB nicht verändert wird).
- Ein einzelnes Kommando in PostgreSQL ist auch eine Transaktion<sup>70</sup>.
- Das heist, dass Einschränkungen zur referentiellen Integrität erst *am Ende* des Kommandos überprüft werden.

```
1 /* Inserting data into the tables for the G-|-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Eine Transaktion kann mehrere Kommandos gruppieren, die dann entweder alle zusammen erfolgreich sind oder alle zusammen fehlschlagen (wobei dann die DB nicht verändert wird).
- Ein einzelnes Kommando in PostgreSQL ist auch eine Transaktion<sup>70</sup>.
- Das heist, dass Einschränkungen zur referentiellen Integrität erst *am Ende* des Kommandos überprüft werden.
- Die Änderungen werden dann zur Datenbank committed, wenn alle Einschränkungen „passen“ und ansonsten zurückgerollt.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

```
16 # psql 16.11 succeeded with exit code 0.
17
```

# Befüllen mit Daten

- Ein einzelnes Kommando in PostgreSQL ist auch eine Transaktion<sup>70</sup>.
- Das heist, dass Einschränkungen zur referentiellen Integrität erst *am Ende* des Kommandos überprüft werden.
- Die Änderungen werden dann zur Datenbank committed, wenn alle Einschränkungen „passen“ und ansonsten zurückgerollt.
- Um nun eine Zeile in Tabelle `g` einzufügen, dann müssen wir auch einen Datensatz in Tabelle `h` verändern, der aktuell noch auf keine Zeile in Tabelle `g` verweist.

```
1 /* Inserting data into the tables for the G-I relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das heist, dass Einschränkungen zur referentiellen Integrität erst *am Ende* des Kommandos überprüft werden.
- Die Änderungen werden dann zur Datenbank committed, wenn alle Einschränkungen „passen“ und ansonsten zurückgerollt.
- Um nun eine Zeile in Tabelle `g` einzufügen, dann müssen wir auch einen Datensatz in Tabelle `h` verändern, der aktuell noch auf keine Zeile in Tabelle `g` verweist.
- Wir müssen diese Zeile in Tabelle `h` dann mit der neuen Zeile in Tabelle `g` verbinden.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Die Änderungen werden dann zur Datenbank committed, wenn alle Einschränkungen „passen“ und ansonsten zurückgerollt.
- Um nun eine Zeile in Tabelle `g` einzufügen, dann müssen wir auch einen Datensatz in Tabelle `h` verändern, der aktuell noch auf keine Zeile in Tabelle `g` verweist.
- Wir müssen diese Zeile in Tabelle `h` dann mit der neuen Zeile in Tabelle `g` verbinden.
- Wenn wir die Einfügung und die Veränderung in ein einziges SQL-Kommando packen können, dann wird es einfach.

```
1 /* Inserting data into the tables for the G-o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Um nun eine Zeile in Tabelle `g` einzufügen, dann müssen wir auch einen Datensatz in Tabelle `h` verändern, der aktuell noch auf keine Zeile in Tabelle `g` verweist.
- Wir müssen diese Zeile in Tabelle `h` dann mit der neuen Zeile in Tabelle `g` verbinden.
- Wenn wir die Einfügung und die Veränderung in ein einziges SQL-Kommando packen können, dann wird es einfach.
- Sonst müssen wir eben lernen, wie man explizit Transaktionen verwendet. Das geht auch.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir müssen diese Zeile in Tabelle **h** dann mit der neuen Zeile in Tabelle **g** verbinden.
- Wenn wir die Einfügung und die Veränderung in ein einziges SQL-Kommando packen können, dann wird es einfach.
- Sonst müssen wir eben lernen, wie man explizit Transaktionen verwendet. Das geht auch.
- Um eine den **fkgid**-Wert einer Zeile in Tabelle **h** zu verändern, müssen wir den Wert des Primärschlüssels **gid** der neuen Zeile in Tabelle **g** kennen.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Wenn wir die Einfügung und die Veränderung in ein einziges SQL-Kommando packen können, dann wird es einfach.
- Sonst müssen wir eben lernen, wie man explizit Transaktionen verwendet. Das geht auch.
- Um eine den `fkgid`-Wert einer Zeile in Tabelle `h` zu verändern, müssen wir den Wert des Primärschlüssels `gid` der neuen Zeile in Tabelle `g` kennen.
- Wir haben den Primärschlüssel `gid` der Tabelle `g` als `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY` deklariert.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Sonst müssen wir eben lernen, wie man explizit Transaktionen verwendet. Das geht auch.
- Um eine den `fkgid`-Wert einer Zeile in Tabelle `h` zu verändern, müssen wir den Wert des Primärschlüssels `gid` der neuen Zeile in Tabelle `g` kennen.
- Wir haben den Primärschlüssel `gid` der Tabelle `g` als `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY` deklariert.
- Das bedeutet, dass der Wert dieses Schlüssels erst erzeugt wird, wenn die Zeile generiert wird.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Um eine den `fkgid`-Wert einer Zeile in Tabelle `h` zu verändern, müssen wir den Wert des Primärschlüssels `gid` der neuen Zeile in Tabelle `g` kennen.
- Wir haben den Primärschlüssel `gid` der Tabelle `g` als `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY` deklariert.
- Das bedeutet, dass der Wert dieses Schlüssels erst erzeugt wird, wenn die Zeile generiert wird.
- Das heist wiederum, dass wir den Wert nicht kennen können, bevor wir die Zeile in Tabelle `g` einfügen.

```
1 /* Inserting data into the tables for the G-|-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir haben den Primärschlüssel `gid` der Tabelle `g` als `INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY` deklariert.
- Das bedeutet, dass der Wert dieses Schlüssels erst erzeugt wird, wenn die Zeile generiert wird.
- Das heist wiederum, dass wir den Wert nicht kennen können, bevor wir die Zeile in Tabelle `g` einfügen.
- Zum Glück ist das ein sehr normales Problem: „Was, wenn wir Daten in eine Tabelle mit einem automatisch generierten Primärschlüssel einfügen und wir dann den Schlüssel wissen müssen?“

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

```
16 # psql 16.11 succeeded with exit code 0.
17
```

# Befüllen mit Daten

- Das bedeutet, dass der Wert dieses Schlüssels erst erzeugt wird, wenn die Zeile generiert wird.
- Das heist wiederum, dass wir den Wert nicht kennen können, bevor wir die Zeile in Tabelle `g` einfügen.
- Zum Glück ist das ein sehr normales Problem: „Was, wenn wir Daten in eine Tabelle mit einem automatisch generierten Primärschlüssel einfügen und wir dann den Schlüssel wissen müssen?“
- In PostgreSQL, das geht mit dem `RETURNING`-Schlüsselwort<sup>55,72</sup>.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das heist wiederum, dass wir den Wert nicht kennen können, bevor wir die Zeile in Tabelle `g` einfügen.
- Zum Glück ist das ein sehr normales Problem: „Was, wenn wir Daten in eine Tabelle mit einem automatisch generierten Primärschlüssel einfügen und wir dann den Schlüssel wissen müssen?“
- In PostgreSQL, das geht mit dem `RETURNING`-Schlüsselwort<sup>55,72</sup>.
- Das geht nicht bei allen DBMSen, aber bei MariaDB<sup>38</sup> und SQLite<sup>56</sup> geht es auch.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- In PostgreSQL, das geht mit dem **RETURNING**-Schlüsselwort<sup>55,72</sup>.
- Das geht nicht bei allen DBMSen, aber bei MariaDB<sup>38</sup> und SQLite<sup>56</sup> geht es auch.
- Mit **INSERT INTO g (fkhid, x) VALUES (1, '123')** **RETURNING gid, fkhid** würden wir eine Zeile in Tabelle **g** einfügen, wo der Wert des Attributs **x** gleich **'123'** ist und der Wert des Attributs **fkhid** gleich **1**.

```
1 /* Inserting data into the tables for the G-I relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --eof GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+---+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das geht nicht bei allen DBMSen, aber bei MariaDB<sup>38</sup> und SQLite<sup>56</sup> geht es auch.
- Mit `INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid` würden wir eine Zeile in Tabelle `g` einfügen, wo der Wert des Attributs `x` gleich `'123'` ist und der Wert des Attributs `fkhid` gleich `1`.
- Das Kommando würde den Wert des automatisch erzeugten Primärschlüssels `gid` zurückliefern und auch den Wert von `fkhid`, welcher in diesem Fall `1` wären.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Mit `INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid` würden wir eine Zeile in Tabelle `g` einfügen, wo der Wert des Attributs `x` gleich `'123'` ist und der Wert des Attributs `fkhid` gleich `1`.
- Das Kommando würde den Wert des automatisch erzeugten Primärschlüssels `gid` zurückliefern und auch den Wert von `fkhid`, welcher in diesem Fall `1` wären.
- Das Kommando würde natürlich fehlschlagen, denn es verletzt die Einschränkung `g_fkhid_gid_fk`.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das Kommando würde den Wert des automatisch erzeugten Primärschlüssels `gid` zurückliefern und auch den Wert von `fkhid`, welcher in diesem Fall `1` wären.
- Das Kommando würde natürlich fehlschlagen, denn es verletzt die Einschränkung `g_fkhid_gid_fk`.
- Aber wir sind schon mal einen Schritt weiter.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das Kommando würde den Wert des automatisch erzeugten Primärschlüssels `gid` zurückliefern und auch den Wert von `fkhid`, welcher in diesem Fall `1` wären.
- Das Kommando würde natürlich fehlschlagen, denn es verletzt die Einschränkung `g_fkhid_gid_fk`.
- Aber wir sind schon mal einen Schritt weiter.
- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET gid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = fkhid;`

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkgid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkgid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkgid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Das Kommando würde natürlich fehlschlagen, denn es verletzt die Einschränkung `g_fkhid_gid_fk`.
- Aber wir sind schon mal einen Schritt weiter.
- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET gid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = fkhid;`
- Das funktioniert aber nicht.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Das Kommando würde natürlich fehlschlagen, denn es verletzt die Einschränkung `g_fkhid_gid_fk`.
- Aber wir sind schon mal einen Schritt weiter.
- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET gid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = fkhid;`
- Das funktioniert aber nicht.
- Was aber geht ist ein sogenannter common table expression (CTE).

```
1 /* Inserting data into the tables for the G-to-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Aber wir sind schon mal einen Schritt weiter.
- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET gid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = fkhid;`
- Das funktioniert aber nicht.
- Was aber geht ist ein sogenannter common table expression (CTE).
- Ein CTE erlaubt es uns, einem Unterausdruck einer Anfrage einen Namen zuzuweisen.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET gid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = fkhid;`
- Das funktioniert aber nicht.
- Was aber geht ist ein sogenannter common table expression (CTE).
- Ein CTE erlaubt es uns, einem Unterausdruck einer Anfrage einen Namen zuzuweisen.
- Dann funktioniert der Unterausdruck in etwa wie eine temporäre Tabelle.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

```
16 # psql 16.11 succeeded with exit code 0.
17
```

# Befüllen mit Daten

- Das funktioniert aber nicht.
- Was aber geht ist ein sogenannter common table expression (CTE).
- Ein CTE erlaubt es uns, einem Unterausdruck einer Anfrage einen Namen zuzuweisen.
- Dann funktioniert der Unterausdruck in etwa wie eine temporäre Tabelle.
- Der Unterausdruck wird genau einmal ausgeführt und kann wie eine read-only Tabelle in den anderen Teilen der Anfrage verwendet werden.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Ein CTE erlaubt es uns, einem Unterausdruck einer Anfrage einen Namen zuzuweisen.
- Dann funktioniert der Unterausdruck in etwa wie eine temporäre Tabelle.
- Der Unterausdruck wird genau einmal ausgeführt und kann wie eine read-only Tabelle in den anderen Teilen der Anfrage verwendet werden.
- Mit `WITH cats AS`  
`(SELECT age, name FROM animals`  
`WHERE type='cat')` würden wir das Ergebnis der Anfrage  
`SELECT age, name FROM animals`  
`WHERE type='cat'` dem CTE `cats` zu weisen.

```
1 /* Inserting data into the tables for the G-o----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --eof GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Der Unterausdruck wird genau einmal ausgeführt und kann wie eine read-only Tabelle in den anderen Teilen der Anfrage verwendet werden.
- Mit `WITH cats AS`  
`(SELECT age, name FROM animals`  
`WHERE type='cat')` würden wir das Ergebnis der Anfrage  
`SELECT age, name FROM animals`  
`WHERE type='cat'` dem CTE `cats` zu weisen.
- Wir könnten dann `cats` verwenden, so als wäre es eine read-only Tabelle, und z. B. mit `SELECT name FROM cats;` die Katzennamen abfragen.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

```
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Mit `WITH cats AS (SELECT age, name FROM animals WHERE type='cat')` würden wir das Ergebnis der Anfrage `SELECT age, name FROM animals WHERE type='cat'` dem CTE `cats` zu weisen.
- Wir könnten dann `cats` verwenden, so als wäre es eine read-only Tabelle, und z. B. mit `SELECT name FROM cats;` die Katzennamen abfragen.
- Es ist ein wenig so wie ein `VIEW`, aber es ist Teil eines einzelnen SQL-Kommandos.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Mit `WITH cats AS`  
`(SELECT age, name FROM animals`  
`WHERE type='cat')` würden wir das  
Ergebnis der Anfrage  
`SELECT age, name FROM animals`  
`WHERE type='cat'` dem CTE `cats`  
zu weisen.
- Wir könnten dann `cats` verwenden, so  
als wäre es eine read-only Tabelle, und  
z. B. mit `SELECT name FROM cats;`  
die Katzennamen abfragen.
- Es ist ein wenig so wie ein `VIEW`, aber  
es ist Teil eines einzelnen  
SQL-Kommandos.
- Nun setzen wir alles zusammen.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 --eof GH_insert_and_select.sql
2 INSERT 0 6
3 UPDATE 1
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir könnten dann `cats` verwenden, so als wäre es eine read-only Tabelle, und z. B. mit `SELECT name FROM cats;` die Katzennamen abfragen.
- Es ist ein wenig so wie ein `VIEW`, aber es ist Teil eines einzelnen SQL-Kommandos.
- Nun setzen wir alles zusammen.
- Nehmen wir an, wir haben bereits eine Zeile in Tabelle `h` mit Primärschlüssel 1 eingefügt.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9 gid | x | hid | y
10 -----+-----
11 3 | 123 | 1 | AB
12 4 | 456 | 3 | EF
13 5 | 789 | 4 | GH
14 3 | 123 | 2 | CD
15 3 | 123 | 5 | IJ
16 (5 rows)
17
18 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir könnten dann `cats` verwenden, so als wäre es eine read-only Tabelle, und z. B. mit `SELECT name FROM cats;` die Katzennamen abfragen.
- Es ist ein wenig so wie ein `VIEW`, aber es ist Teil eines einzelnen SQL-Kommandos.
- Nun setzen wir alles zusammen.
- Nehmen wir an, wir haben bereits eine Zeile in Tabelle `h` mit Primärschlüssel 1 eingefügt.
- Wir können das jederzeit, denn diese Zeilen müssen nicht mit den Zeilen in Tabelle `g` verbunden sein.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9 gid | x | hid | y
10 -----+-----+-----+-----
11 3 | 123 | 1 | AB
12 4 | 456 | 3 | EF
13 5 | 789 | 4 | GH
14 3 | 123 | 2 | CD
15 3 | 123 | 5 | IJ
16 (5 rows)
17
18 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Es ist ein wenig so wie ein **VIEW**, aber es ist Teil eines einzelnen SQL-Kommandos.
- Nun setzen wir alles zusammen.
- Nehmen wir an, wir haben bereits eine Zeile in Tabelle **h** mit Primärschlüssel 1 eingefügt.
- Wir können das Jederzeit, denn diese Zeilen müssen nicht mit den Zeilen in Tabelle **g** verbunden sein.
- Dann tun wir das Kommando zum Einfügen einer Zeile in Tabelle **g** in einen CTE **g\_new** in dem wir schreiben **WITH g\_new AS (INSERT INTO g (fkhid, x)VALUES (1, '123') RETURNING id, fkhid).**

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Nun setzen wir alles zusammen.
- Nehmen wir an, wir haben bereits eine Zeile in Tabelle `h` mit Primärschlüssel 1 eingefügt.
- Wir können das Jederzeit, denn diese Zeilen müssen nicht mit den Zeilen in Tabelle `g` verbunden sein.
- Dann tun wir das Kommando zum Einfügen einer Zeile in Tabelle `g` in einen CTE `g_new` in dem wir schreiben `WITH g_new AS (INSERT INTO g (fkhid, x)VALUES (1, '123') RETURNING id, fkhid)`.
- Dieser CTE fügt eine Zeile in Tabelle `g` ein, die die Zeile in Tabelle `h` mit Primärschlüssel 1 referenziert.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir können das jederzeit, denn diese Zeilen müssen nicht mit den Zeilen in Tabelle `g` verbunden sein.
- Dann tun wir das Kommando zum Einfügen einer Zeile in Tabelle `g` in einen CTE `g_new` in dem wir schreiben `WITH g_new AS (INSERT INTO g (fkhid, x)VALUES (1, '123') RETURNING id, fkhid)`.
- Dieser CTE fügt eine Zeile in Tabelle `g` ein, die die Zeile in Tabelle `h` mit Primärschlüssel 1 referenziert.
- Er liefert auch den Primärschlüssel der neuen Zeile in Tabelle `g` als `gid` zurück, ebenso den Fremdschlüssel `fkhid` (mit Wert 1).

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Dann tun wir das Kommando zum Einfügen einer Zeile in Tabelle `g` in einen CTE `g_new` in dem wir schreiben `WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING id, fkhid)`.
- Dieser CTE fügt eine Zeile in Tabelle `g` ein, die die Zeile in Tabelle `h` mit Primärschlüssel 1 referenziert.
- Er liefert auch den Primärschlüssel der neuen Zeile in Tabelle `g` als `gid` zurück, ebenso den Fremdschlüssel `fkhid` (mit Wert 1).
- Dann benutzen wir diesen CTE um die Zeile mit Primärschlüssel `hid = fkhid` in Tabelle `h` upzudaten.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Dieser CTE fügt eine Zeile in Tabelle `g` ein, die die Zeile in Tabelle `h` mit Primärschlüssel 1 referenziert.
- Er liefert auch den Primärschlüssel der neuen Zeile in Tabelle `g` als `gid` zurück, ebenso den Fremdschlüssel `fkhid` (mit Wert 1).
- Dann benutzen wir diesen CTE um die Zeile mit Primärschlüssel `hid = fkhid` in Tabelle `h` upzudaten.
- Wir machen `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+-----+----
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Dieser CTE fügt eine Zeile in Tabelle `g` ein, die die Zeile in Tabelle `h` mit Primärschlüssel 1 referenziert.
- Er liefert auch den Primärschlüssel der neuen Zeile in Tabelle `g` als `gid` zurück, ebenso den Fremdschlüssel `fkhid` (mit Wert 1).
- Dann benutzen wir diesen CTE um die Zeile mit Primärschlüssel `hid = fkhid` in Tabelle `h` upzudaten.
- Wir machen `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`
- Dadurch zeigt der Fremdschlüssel, der in dieser Zeile gespeichert ist, auf unsere neue Zeile in Tabelle `g`.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 UPDATE 1
9
10 gid | x | hid | y
11 ----+---+----+---
12 3 | 123 | 1 | AB
13 4 | 456 | 3 | EF
14 5 | 789 | 4 | GH
15 3 | 123 | 2 | CD
16 3 | 123 | 5 | IJ
17 (5 rows)
```

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

# Befüllen mit Daten

- Dann benutzen wir diesen CTE um die Zeile mit Primärschlüssel `hid = fkhid` in Tabelle `h` upzudaten.
- Wir machen `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`
- Dadurch zeigt der Fremdschlüssel, der in dieser Zeile gespeichert ist, auf unsere neue Zeile in Tabelle `g`.
- Weil beide Unterausdrücke Teil des selben Kommandos sind, wird die referentielle Integrität – also die `REFERENCES`-Einschränkung – erst am Ende geprüft, wenn das `;` erreicht wird.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

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

# Befüllen mit Daten

- Dann benutzen wir diesen CTE um die Zeile mit Primärschlüssel `hid = fkhid` in Tabelle `h` upzudaten.
- Wir machen `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`
- Dadurch zeigt der Fremdschlüssel, der in dieser Zeile gespeichert ist, auf unsere neue Zeile in Tabelle `g`.
- Weil beide Unterausdrücke Teil des selben Kommandos sind, wird die referentielle Integrität – also die `REFERENCES`-Einschränkung – erst am Ende geprüft, wenn das `;` erreicht wird.
- Zu dieser Zeit stimmt es wieder und die referentielle Integrität ist intakt.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Wir machen `UPDATE h SET fkgid = g_new.gid FROM g_new WHERE h.gid = g_new.fkhid;`
- Dadurch zeigt der Fremdschlüssel, der in dieser Zeile gespeichert ist, auf unsere neue Zeile in Tabelle `g`.
- Weil beide Unterausdrücke Teil des selben Kommandos sind, wird die referentielle Integrität – also die `REFERENCES`-Einschränkung – erst am Ende geprüft, wenn das `;` erreicht wird.
- Zu dieser Zeit stimmt es wieder und die referentielle Integrität ist intakt.
- Wir haben eine Zeile in Tabelle `g` eingefügt und die passende Zeile in Tabelle `h` referenziert sie auch.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
```

```
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
```

```
16 # psql 16.11 succeeded with exit code 0.
17
```

# Befüllen mit Daten

- Dadurch zeigt der Fremdschlüssel, der in dieser Zeile gespeichert ist, auf unsere neue Zeile in Tabelle `g`.
- Weil beide Unterausdrücke Teil des selben Kommandos sind, wird die referentielle Integrität – also die **REFERENCES**-Einschränkung – erst am Ende geprüft, wenn das `;` erreicht wird.
- Zu dieser Zeit stimmt es wieder und die referentielle Integrität ist intakt.
- Wir haben eine Zeile in Tabelle `g` eingefügt und die passende Zeile in Tabelle `h` referenziert sie auch.
- Es ist dann viel einfacher, existierende Zeilen in `g` mit neuen Zeilen in `h` zu verbinden.

```
1 /* Inserting data into the tables for the G-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Befüllen mit Daten

- Weil beide Unterausdrücke Teil des selben Kommandos sind, wird die referentielle Integrität – also die **REFERENCES**-Einschränkung – erst am Ende geprüft, wenn das `;` erreicht wird.
- Zu dieser Zeit stimmt es wieder und die referentielle Integrität ist intakt.
- Wir haben eine Zeile in Tabelle `g` eingefügt und die passende Zeile in Tabelle `h` referenziert sie auch.
- Es ist dann viel einfacher, existierende Zeilen in `g` mit neuen Zeilen in `h` zu verbinden.
- Das geht mit einem einfachen **UPDATE** Statement auf Tabelle `h`.

```
1 /* Inserting data into the tables for the G-|o-----|<-H relationship. */
2
3 -- Insert some rows into the table for entity type H.
4 -- Not specifying 'g' leave the references G as NULL for now.
5 INSERT INTO h (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into G and relate to H. We do this three times.
8 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (1, '123')
9 RETURNING gid, fkhid)
10 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
11
12 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (3, '456')
13 RETURNING gid, fkhid)
14 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
15
16 WITH new_g AS (INSERT INTO g (fkhid, x) VALUES (4, '789')
17 RETURNING gid, fkhid)
18 UPDATE h SET fkgid = new_g.gid FROM new_g WHERE h.hid = new_g.fkhid;
19
20 -- Link one H row to another G row. (We do this twice.)
21 UPDATE h SET fkgid = 3 WHERE hid = 2;
22 UPDATE h SET fkgid = 3 WHERE hid = 5;
23
24 -- Combine the rows from G and H.
25 SELECT gid, x, hid, y FROM h INNER JOIN g ON g.gid = h.fkgid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2 ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3 INSERT 0 6
4 UPDATE 1
5 UPDATE 1
6 UPDATE 1
7 UPDATE 1
8 gid | x | hid | y
9 ----+-----+----+---
10 3 | 123 | 1 | AB
11 4 | 456 | 3 | EF
12 5 | 789 | 4 | GH
13 3 | 123 | 2 | CD
14 3 | 123 | 5 | IJ
15 (5 rows)
16
17 # psql 16.11 succeeded with exit code 0.
```

# Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

| Table <b>g</b> |       |       |
|----------------|-------|-------|
| gid            | fkhid | x     |
| 3              | 1     | „123“ |
| 4              | 3     | „456“ |
| 5              | 4     | „789“ |

| Table <b>h</b> |       |      |
|----------------|-------|------|
| hid            | fkgid | y    |
| 6              | NULL  | „KL“ |
| 1              | 3     | „AB“ |
| 3              | 4     | „EF“ |
| 4              | 5     | „GH“ |
| 2              | 3     | „CD“ |
| 5              | 3     | „IJ“ |

# Testen der Einschränkungen

- Können wir eine zweite Zeile in Tabelle `g` auf eine Zeile in Tabelle `h` zeigen lassen, die schon mit einer anderen Zeile in Tabelle `g` verbunden ist?

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make our new G row point to it as its "primary H" anyway?
5 INSERT INTO g (fkhid, x) VALUES (4, '999');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_3.sql
2 psql:conceptualToRelational/GH_insert_error_3.sql:5: ERROR: duplicate
 ↳ key value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_3.sql:5: STATEMENT: /* Can
 ↳ we insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make our new G row point to it as its "primary H" anyway?
7 INSERT INTO g (fkhid, x) VALUES (4, '999');
8 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Können wir eine zweite Zeile in Tabelle `g` auf eine Zeile in Tabelle `h` zeigen lassen, die schon mit einer anderen Zeile in Tabelle `g` verbunden ist?
- Unser Modell erlaubt das nicht.

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make our new G row point to it as its "primary H" anyway?
5 INSERT INTO g (fkhid, x) VALUES (4, '999');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_3.sql
2 psql:conceptualToRelational/GH_insert_error_3.sql:5: ERROR: duplicate
 ↳ key value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_3.sql:5: STATEMENT: /* Can
 ↳ we insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make our new G row point to it as its "primary H" anyway?
7 INSERT INTO g (fkhid, x) VALUES (4, '999');
8 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Können wir eine zweite Zeile in Tabelle `g` auf eine Zeile in Tabelle `h` zeigen lassen, die schon mit einer anderen Zeile in Tabelle `g` verbunden ist?
- Unser Modell erlaubt das nicht.
- Und es geht auch nicht.

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make our new G row point to it as its "primary H" anyway?
5 INSERT INTO g (fkhid, x) VALUES (4, '999');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_3.sql
2 psql:conceptualToRelational/GH_insert_error_3.sql:5: ERROR: duplicate
 ↳ key value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_3.sql:5: STATEMENT: /* Can
 ↳ we insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make our new G row point to it as its "primary H" anyway?
7 INSERT INTO g (fkhid, x) VALUES (4, '999');
8 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Können wir eine zweite Zeile in Tabelle `g` auf eine Zeile in Tabelle `h` zeigen lassen, die schon mit einer anderen Zeile in Tabelle `g` verbunden ist?
- Unser Modell erlaubt das nicht.
- Und es geht auch nicht.
- Können wir eine Zeile in Tabelle `h` auf eine andere Zeile in Tabelle `g` verweisen lassen?

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make it point to the new G row instead?
5 WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
6 RETURNING gid, fkhid)
7 UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_4.sql
2 psql:conceptualToRelational/GH_insert_error_4.sql:7: ERROR: duplicate
 ↳ key value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_4.sql:7: STATEMENT: /* Can
 ↳ we insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make it point to the new G row instead?
7 WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
8 RETURNING gid, fkhid)
9 UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;
10 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen

- Können wir eine zweite Zeile in Tabelle `g` auf eine Zeile in Tabelle `h` zeigen lassen, die schon mit einer anderen Zeile in Tabelle `g` verbunden ist?
- Unser Modell erlaubt das nicht.
- Und es geht auch nicht.
- Können wir eine Zeile in Tabelle `h` auf eine andere Zeile in Tabelle `g` verweisen lassen?
- Nein, das geht auch nicht.

```
1 /* Can we insert a G that is related to a H related to another G? */
2
3 -- H with id 4 is already related to G with id 3.
4 -- Can we make it point to the new G row instead?
5 WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
6 RETURNING gid, fkhid)
7 UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_4.sql
2 psql:conceptualToRelational/GH_insert_error_4.sql:7: ERROR: duplicate
 ↳ key value violates unique constraint "g_fkhid_key"
3 DETAIL: Key (fkhid)=(4) already exists.
4 psql:conceptualToRelational/GH_insert_error_4.sql:7: STATEMENT: /* Can
 ↳ we insert a G that is related to a H related to another G? */
5 -- H with id 4 is already related to G with id 3.
6 -- Can we make it point to the new G row instead?
7 WITH g_new AS (INSERT INTO g (fkhid, x) VALUES (4, '555')
8 RETURNING gid, fkhid)
9 UPDATE h SET fkgid = g_new.gid FROM g_new WHERE hid = fkhid;
10 # psql 16.11 failed with exit code 3.
```

# Testen der Einschränkungen



- Unser Modell erlaubt das nicht.
- Und es geht auch nicht.
- Können wir eine Zeile in Tabelle **h** auf eine andere Zeile in Tabelle **g** verweisen lassen?
- Nein, das geht auch nicht.
- Können wir eine Zeile in Tabelle **h**, die aktuell mit einer Zeile in Tabelle **g** verbunden ist, auf eine andere Zeile zeigen lassen (via **UPDATE**)?

# Testen der Einschränkungen



- Und es geht auch nicht.
- Können wir eine Zeile in Tabelle `h` auf eine andere Zeile in Tabelle `g` verweisen lassen?
- Nein, das geht auch nicht.
- Können wir eine Zeile in Tabelle `h`, die aktuell mit einer Zeile in Tabelle `g` verbunden ist, auf eine andere Zeile zeigen lassen (via `UPDATE`)?
- Nein, geht auch nicht.

# Testen der Einschränkungen

- Können wir eine Zeile in Tabelle `h` auf eine andere Zeile in Tabelle `g` verweisen lassen?
- Nein, das geht auch nicht.
- Können wir eine Zeile in Tabelle `h`, die aktuell mit einer Zeile in Tabelle `g` verbunden ist, auf eine andere Zeile zeigen lassen (via `UPDATE`)?
- Nein, geht auch nicht.
- Unsere Einschränkungen sind ziemlich stark und schützen unsere Daten.

```
1 /* Can we change the relationship of a H away from its primary G? */
2
3 -- H with id 1 is used as "primary H" for G with ID 1.
4 -- Can we make it point to another G?
5 UPDATE h SET fkgid = 3 WHERE hid = 1;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_5.sql
2 UPDATE 1
3 # psql 16.11 succeeded with exit code 0.
```

# Testen der Einschränkungen

- Können wir eine Zeile in Tabelle `h` auf eine andere Zeile in Tabelle `g` verweisen lassen?
- Nein, das geht auch nicht.
- Können wir eine Zeile in Tabelle `h`, die aktuell mit einer Zeile in Tabelle `g` verbunden ist, auf eine andere Zeile zeigen lassen (via `UPDATE`)?
- Nein, geht auch nicht.
- Unsere Einschränkungen sind ziemlich stark und schützen unsere Daten.

```
1 /* Can we change the relationship of a H away from its primary G? */
2
3 -- H with id 1 is used as "primary H" for G with ID 1.
4 -- Can we make it point to another G?
5 UPDATE h SET fkgid = 3 WHERE hid = 1;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↪ ON_ERROR_STOP=1 -ebf GH_insert_error_5.sql
2 UPDATE 1
3 # psql 16.11 succeeded with exit code 0.
```



# Zusammenfassung



- Fertig





谢谢你们！  
Thank you!  
Vielen Dank!



# References I



- [1] "[ALTER TABLE](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. URL: <https://www.postgresql.org/docs/17/sql-altertable.html> (besucht am 2025-11-18) (siehe S. 85–98).
- [2] 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. 333).
- [3] 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<sup>4</sup> (siehe S. 323, 333).
- [4] 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<sup>3</sup> (siehe S. 323, 333).
- [5] 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. 332).
- [6] 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. 333, 334).
- [7] 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. 333).
- [8] Daniel Bartholomew. *MariaDB and MySQL Common Table Expressions and Window Functions Revealed*. New York, NY, USA: Apress Media, LLC, Nov. 2017. ISBN: 978-1-4842-3120-3 (siehe S. 332).
- [9] 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. 334).
- [10] 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. 332).
- [11] 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. 333).

# References II



- [12] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 333).
- [13] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 332).
- [14] 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. 332).
- [15] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 333, 334).
- [16] 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. 334).
- [17] 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](http://bit.csc.lsu.edu/%7Echen/pdf/Chen_Pioneers.pdf) (besucht am 2025-03-06) (siehe S. 332).
- [18] 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. 324, 332).
- [19] 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<sup>18</sup> for a more comprehensive introduction. (Siehe S. 332).
- [20] 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. 334).

# References III



- [21] 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. 333).
- [22] "Constraints". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.5. URL: <https://www.postgresql.org/docs/17/ddl-constraints.html> (besucht am 2025-02-28) (siehe S. 19–26).
- [23] "[CREATE TABLE](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. URL: <https://www.postgresql.org/docs/17/sql-createtable.html> (besucht am 2025-04-21) (siehe S. 266).
- [24] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 334).
- [25] 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. 334).
- [26] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 334).
- [27] 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. 334).
- [28] 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. 333).
- [29] 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. 332, 333).

# References IV



- [30] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy und Jignesh M. Patel. "SQLite: Past, Present, and Future". *Proceedings of the VLDB Endowment (PVLDB)* 15(12):3535–3547, Aug. 2022. Irvine, CA, USA: Very Large Data Bases Endowment Inc. ISSN: **2150-8097**. doi:[10.14778/3554821.3554842](https://doi.org/10.14778/3554821.3554842). URL: <https://www.vldb.org/pvldb/vol15/p3535-gaffney.pdf> (besucht am 2025-01-12). All papers in this issue were presented at the *48th International Conference on Very Large Data Bases (VLDB 2022)*, 9 5-9, 2022, hybrid/Sydney, NSW, Australia (siehe S. **334**).
- [31] 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. **333**).
- [32] 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. **333**).
- [33] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: **978-1-0981-0894-6** (siehe S. **333**).
- [34] 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. **333, 334**).
- [35] D. Richard Hipp u. a. "Well-Known Users of SQLite". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2. Jan. 2023. URL: <https://www.sqlite.org/famous.html> (besucht am 2025-01-12) (siehe S. **334**).
- [36] 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](https://doi.org/10.1007/978-3-031-35122-8) (siehe S. **333**).
- [37] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO\\_IEC\\_9075-1\\_2023\\_ed\\_6\\_-\\_id\\_76583\\_Publication\\_PDF\\_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. **334**).

# References V



- [38] “[INSERT...RETURNING](#)”. In: *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/docs/server/reference/sql-statements/data-manipulation/inserting-loading-data/insertreturning> (besucht am 2025-07-06) (siehe S. 267–284).
- [39] “Joined Tables”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 7.2.1.1. URL: <https://www.postgresql.org/docs/17/queries-table-expressions.html#QUERIES-JOIN> (besucht am 2025-03-01) (siehe S. 19–26).
- [40] 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. 332).
- [41] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 334).
- [42] 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. 333).
- [43] 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. 332).
- [44] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 333).
- [45] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (besucht am 2025-04-24) (siehe S. 333).
- [46] 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. 334).
- [47] 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. 332).

# References VI



- [48] 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. 333).
- [49] 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. 332).
- [50] *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).
- [51] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 333).
- [52] 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. 332).
- [53] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: **978-1-78398-154-0** (siehe S. 333).
- [54] 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. 333).
- [55] "Returning Data from Modified Rows". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 6.4. URL: <https://www.postgresql.org/docs/17/dml-returning.html> (besucht am 2025-04-21) (siehe S. 267–283).
- [56] "**RETURNING**". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 8. Mai 2024. URL: [https://sqlite.org/lang\\_returning.html](https://sqlite.org/lang_returning.html) (besucht am 2025-04-24) (siehe S. 267–284).
- [57] 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. 332).
- [58] 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](https://dbs.academy.lv/lection/dbs_LS04EN_erd.pdf) (besucht am 2025-03-29) (siehe S. 332).

# References VII



- [59] Yuriy Shamshin. *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. URL: <https://dbs.academy.lv> (besucht am 2025-01-11).
- [60] 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](https://dbs.academy.lv/lection/dbs_LS06EN_er2rm.pdf) (besucht am 2025-04-20) (siehe S. 54–58, 167–172).
- [61] 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. 333).
- [62] 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. 333).
- [63] *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2025. URL: <https://sqlite.org> (besucht am 2025-04-24) (siehe S. 334).
- [64] "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. 334).
- [65] 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. 329, 334).
- [66] 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<sup>65</sup> (siehe S. 334).
- [67] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 333, 334).
- [68] 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. 333).

# References VIII



- [69] 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. 333).
- [70] "Transactions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 3.4. URL: <https://www.postgresql.org/docs/17/tutorial-transactions.html> (besucht am 2025-04-21) (siehe S. 267–272).
- [71] "UPDATE". 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-update.html> (besucht am 2025-03-07) (siehe S. 100–104).
- [72] "What does standard SQL or any of the non-PostgreSQL SQLs do instead of RETURNING?" In: *Database Administrators*. Hrsg. von user210271. New York, NY, USA: Stack Exchange Inc., 7.–8. Juni 2020. URL: <https://dba.stackexchange.com/questions/268664> (besucht am 2025-04-23) (siehe S. 267–283).
- [73] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 333).
- [74] 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. 332, 333).
- [75] 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. 333).
- [76] 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. 332).
- [77] *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. 333).

# References IX



- [78] 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. **333**).
- [79] Marianne Winslett und Vanessa Braganholo. "Richard Hipp Speaks Out on SQLite". *ACM SIGMOD Record* 48(2):39–46, Juni 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: **0163-5808**. doi:**10.1145/3377330.3377338** (siehe S. **334**).
- [80] "**WITH** Queries (Common Table Expressions)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 7.8. URL: <https://www.postgresql.org/docs/17/queries-with.html> (besucht am 2025-04-21) (siehe S. **332**).
- [81] 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. **332**).
- [82] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: **978-1-78439-687-9** (siehe S. **332**).

# 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<sup>13,47,82</sup>. 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<sup>10,43,49,52,57</sup>. 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.

**CTE** *Common Table Expressions* are Structured Query Language (SQL) constructs for simplifying complex queries by allowing us to break them into smaller parts which are evaluated only once and, hence, can be reused. CTEs act as temporary named result sets created during query execution and discarded after query completion<sup>8,29,80</sup>.

**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*<sup>74</sup>.

**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<sup>81</sup>.

**ERD** Entity relationship diagrams show the relationships between objects, e.g., between the tables in a DB and how they reference each other<sup>5,14,17–19,40,58,76</sup>

**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<sup>15,34</sup>.
- Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows<sup>6,33,61,69,73</sup>. 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<sup>3,4,7,28,45,53</sup>. See <https://mariadb.org> for more information.
- Microsoft Windows** is a commercial proprietary operating system<sup>12</sup>. 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<sup>11,28,54,68,78</sup>. 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<sup>2</sup>. Learn more at <https://pgmodeler.io>.
- PostgreSQL** An open source object-relational DBMS<sup>29,48,51,68</sup>. See <https://postgresql.org> for more information.
- psql** is the client program used to access the PostgreSQL DBMS server.
- Python** The Python programming language<sup>36,42,44,75</sup>, i.e., what you will learn about in our book<sup>75</sup>. Learn more at <https://python.org>.
- relational database** A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other<sup>21,31,32,62,67,74,77</sup>.

# 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 carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers<sup>15</sup> 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“<sup>41</sup>.

**SQL** The *Structured Query Language* is basically a programming language for querying and manipulating relational databases<sup>16,24–26,37,46,64–67</sup>. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference<sup>64</sup>.

**SQLite** is an relational DBMS which runs as in-process library that works directly on files as opposed to the client-server architecture used by other common DBMSes. It is the most wide-spread SQL-based DB in use today, installed in nearly every smartphone, computer, web browser, television, and automobile<sup>16,30,35,79</sup>. Learn more at <https://sqlite.org><sup>63</sup>.

**terminal** A terminal is a text-based window where you can enter commands and execute them<sup>6,20</sup>. 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<sup>20,34</sup>. 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<sup>9,27</sup>