



合肥大學
HEFEI UNIVERSITY



Datenbanken

37. Logisches Schema: Beziehungen

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Databases



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

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



Outline



1. Einleitung

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

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

4. $E \vdash \text{---} \circ \lhd F$

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

6. $I \Vdash \text{---} \Vdash J$

7. $K \Vdash \text{---} \circ \lhd L$

8. $M \Vdash \text{---} \vdash \lhd N$

9. $O \rhd \text{---} \circ \lhd P$

10. $Q \rhd \text{---} \vdash \lhd R$

11. $S \rhd \text{---} \vdash \lhd T$

12. 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 $C \text{ } \vdash \bigcirc \text{ } \text{---} \text{ } \vdash D$ -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 $C \text{ } \vdash \bigcirc \text{ } \text{---} \text{ } \vdash D$ -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 $C \text{ } \vdash \bigcirc \text{ } \dashv \vdash D$ -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.
- Das Beziehungsmuster $C \text{ } \vdash \bigcirc \text{ } \dashv \vdash D$ erlaubt, dass eine Entität vom Type D entweder mit einer oder keiner Entität vom Typ C verbunden ist.

Realisierung



- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.

Realisierung



- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \dashv \circ \dashv \vdash H$ erklärt.

Realisierung



- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } \text{---} H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } \text{---} H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“
- Aber wie macht man es, dass jede Entität vom Typ G dann mit mindestens einer Entität vom Typ H verbunden ist?

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } \text{---} H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“
- Aber wie macht man es, dass jede Entität vom Typ G dann mit mindestens einer Entität vom Typ H verbunden ist?
- Das kommt dann nicht vor.

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } \text{---} H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“
- Aber wie macht man es, dass jede Entität vom Typ G dann mit mindestens einer Entität vom Typ H verbunden ist?
- Das kommt dann nicht vor.
- Hier kommt es vor.

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“
- Aber wie macht man es, dass jede Entität vom Typ G dann mit mindestens einer Entität vom Typ H verbunden ist?
- Das kommt dann nicht vor.
- Hier kommt es vor.
- Achtung: **Das wird die hardcore-ste Einheit in der Vorlesung!**

- In der Vorbereitung auf diese Vorlesung habe ich mir einige andere Vorlesungen anguckt.
- In vielen werden konzeptuelle Beziehungstypen wie $G \text{ --- } \bigcirc \text{ --- } H$ erklärt.
- Es wird aber nicht erklärt, wie man das dann mit SQL machen kann.
- Man sagt vielleicht „Da braucht man 2 oder 3 Tabellen dafür.“
- Aber wie macht man es, dass jede Entität vom Typ G dann mit mindestens einer Entität vom Typ H verbunden ist?
- Das kommt dann nicht vor.
- Hier kommt es vor.
- Achtung: Das wird die hardcore-ste Einheit in der Vorlesung!
- Schnallen Sie sich an!

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.

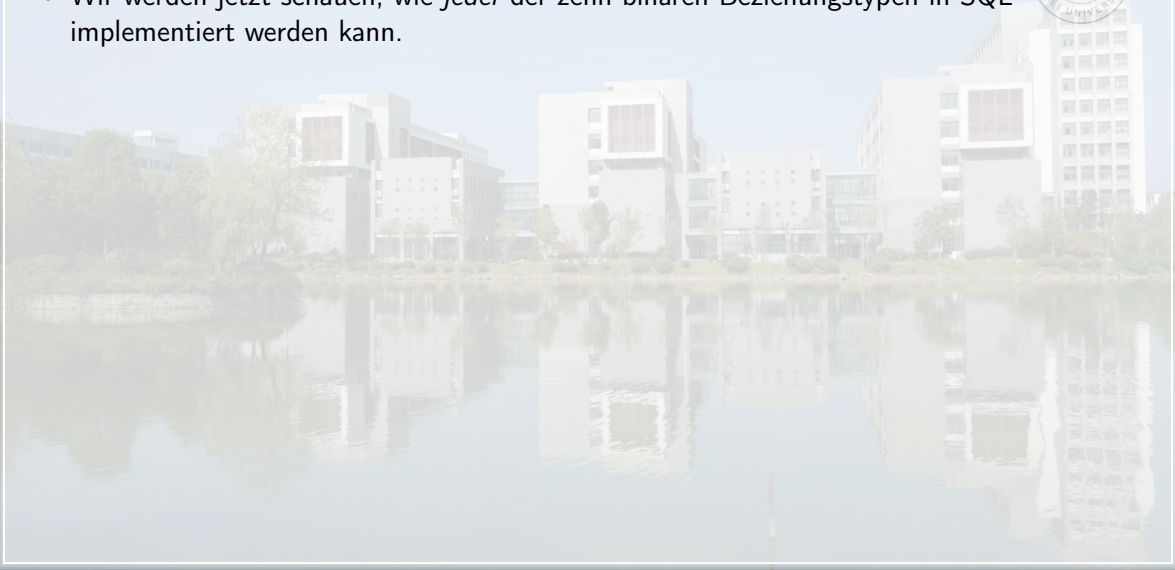
Definition: Referentielle Integrität

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`⁴³ 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`⁴³ 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 + B



A   B

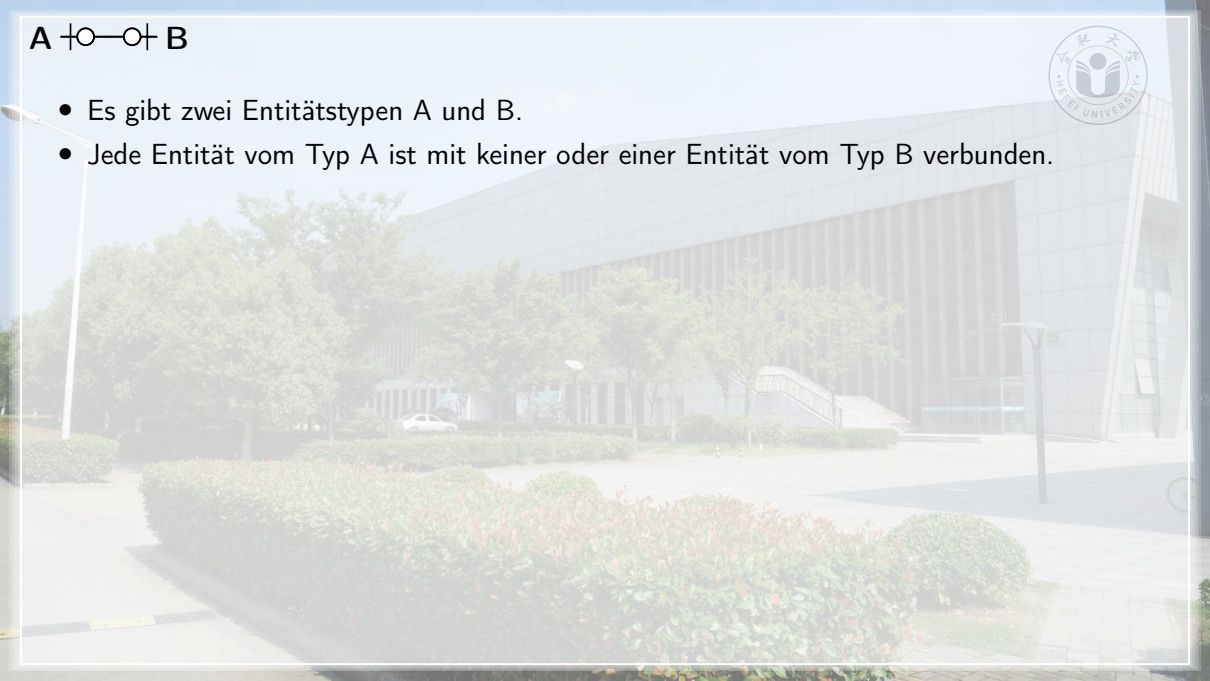
- Es gibt zwei Entitätstypen A und B.



A   B



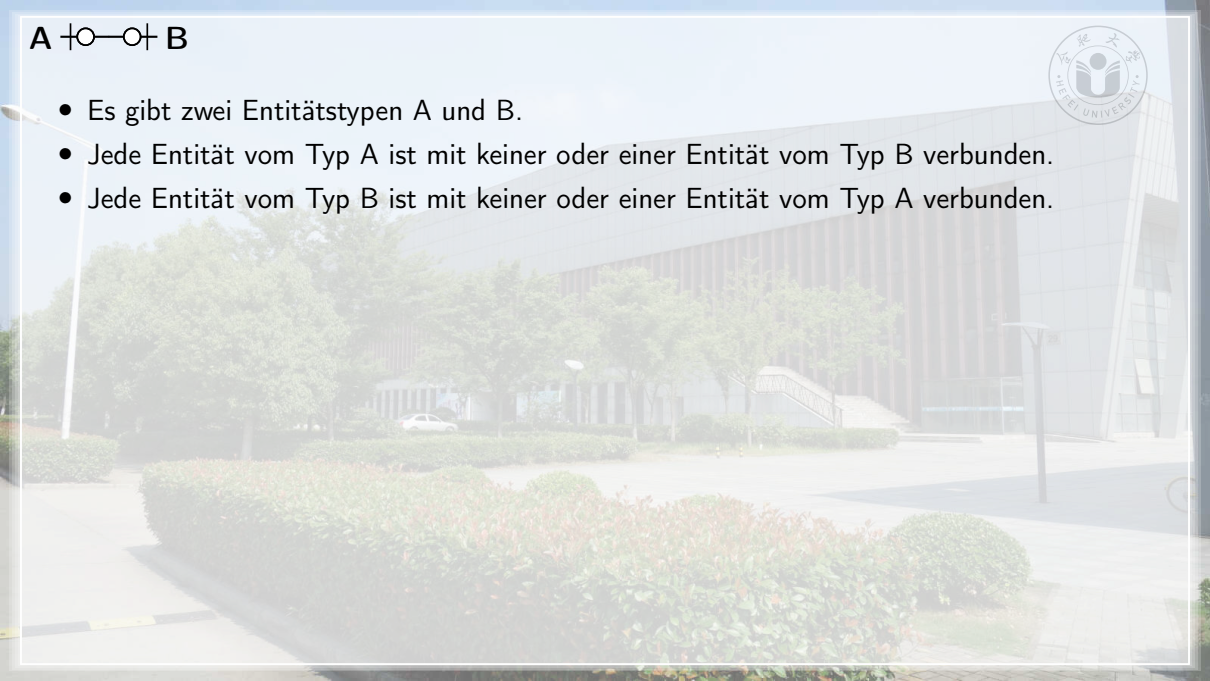
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.



A   B



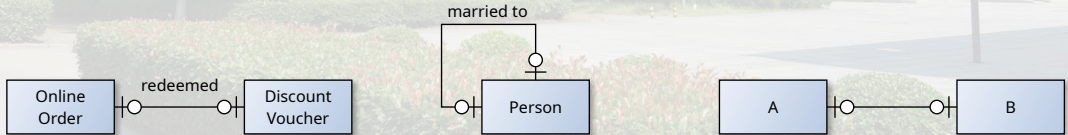
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.



A B



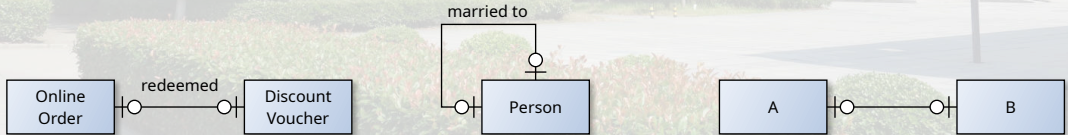
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:



A B



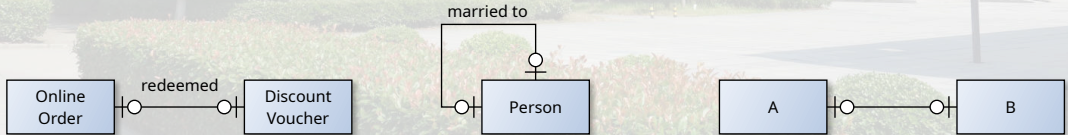
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können.



A B



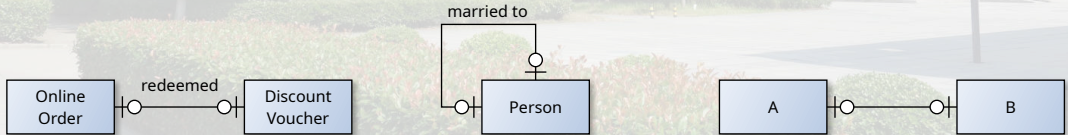
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus,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



A B



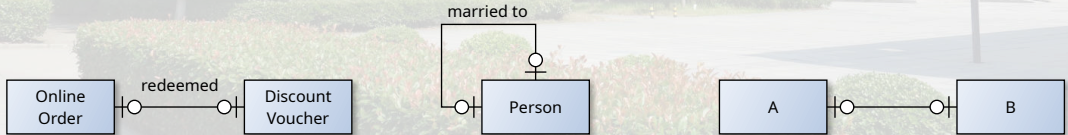
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus,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



A B



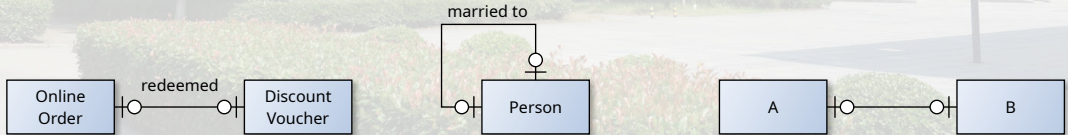
- Es gibt zwei Entitätstypen A und B.
- Jede Entität vom Typ A ist mit keiner oder einer Entität vom Typ B verbunden.
- Jede Entität vom Typ B ist mit keiner oder einer Entität vom Typ A verbunden.
- Beispiele:
 - Wir haben eine Pizzeria bei der Kunden online bestellen können. Wir geben Gutscheine heraus,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



A $\oplus \bigcirc \ominus \oplus$ B: Zwei Realisierungen

- Es gibt zwei Möglichkeiten, das zu realisieren.



A $\oplus$ $\ominus$ B: Zwei Realisierungen

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



A $\oplus$ 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.



A 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.



A $\oplus$ $\ominus$ $\oplus$ 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A $\oplus$ $\ominus$ $\oplus$ 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.

A $\oplus$ $\ominus$ $\oplus$ 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. Wir geben ihr eine weitere Spalte **y**, in der wir einen Text aus zwei Zeichen speichern.

A $\oplus$ $\ominus$ $\oplus$ 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. 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.

A $\oplus$ B: Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁵.

```
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.
```

A $\oplus$ $\ominus$ $\oplus$ B: Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁵.
- 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.
```

A $\oplus$ $\ominus$ B: Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁵.
- 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 );
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.
```

A $\oplus$ $\bigcirc$ $\oplus$ B: Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁵.
- 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 );
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.
```

A B: Mit Drei Tabellen

- Fangen wir mit dem drei-Tabellen-Ansatz an⁶⁵.
- 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 );
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.
```

A B: 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 );
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.
```

A B: 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 );
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.
```

A $\oplus$ B: 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.
```

A B: 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 );
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.
```

A B: 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 );
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.
```

A $\oplus$ $\ominus$ B: 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 );
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.
```

A B: 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, 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.
```

A B: 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, 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.
```

A B: 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, 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.
```

A B: 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, 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.
```

A B: 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, 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.
```

A B: 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, 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.
```

A B: 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.

```
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, 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.
```

A B: 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`.

```
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, 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.
```

A B: 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**.

```
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, 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.
```

A B: Befüllen mit Daten

- 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, 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.
```

A $\oplus$ B: 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

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

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

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

A $\oplus$ B: 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.
```

A $\oplus$ B: Probleme mit der Drei-Tabellen-Methode



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

A $\oplus$ B: 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.

A $\oplus$ $\ominus$ $\oplus$ B: 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.

A $\oplus$ $\ominus$ B: 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).

A $\oplus$ $\ominus$ $\oplus$ B: 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`.

A $\oplus$ $\ominus$ 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...



A $\oplus$ $\ominus$ 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...
- Das probieren wir jetzt aus.



A 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...
- 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.
```

A 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. ...
- 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  ALTER TABLE
5  # psql 16.11 succeeded with exit code 0.
```

A 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. ...
- 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.
```

A B: 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.
```

A $\begin{smallmatrix} \circ \\ \vdash \end{smallmatrix} \begin{smallmatrix} \circ \\ \vdash \end{smallmatrix} B$: 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.
```

A B: 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.
```

A $\oplus$ B: 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.
```

A $\oplus$ B: Zwei-Tabellen Lösung

- 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.
- 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.
```

A $\oplus$ B: 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.
```

A $\oplus$ B: 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-|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.
```

A B: 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-|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.
```

A $\oplus$ B: Zwei-Tabellen Lösung

- Das fügen wir später mit `ALTER TABLE`¹ hinzu (PgModeler macht das ja auch so).
- Sonst müssten wir 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-|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.
```

A B: 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.
```

A B: 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.
```

A B: 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.
```

A B: 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**-Statements⁷⁶.

```
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
8  aid | x | bid | y
9  ----+-----+-----+----
10    1 | 123 | 1 | AB
11    2 | 456 | 3 | EF
12    3 | 789 | 4 | GH
13  (3 rows)
14
15 # psql 16.11 succeeded with exit code 0.
```

A B: 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-Statements**⁷⁶.
- **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.
```

A B: 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**-Statements⁷⁶.
- **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.
```

A $\oplus$ B: 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“

A $\vdash$ $\circ$ $\dashv$ B: 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.



A $\oplus$ $\ominus$ $\oplus$ B: 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).

A B: 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.
```

A B: 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 $\oplus$ D



C $\oplus$ 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 $\oplus$ $\circ$ --- $\parallel$ D

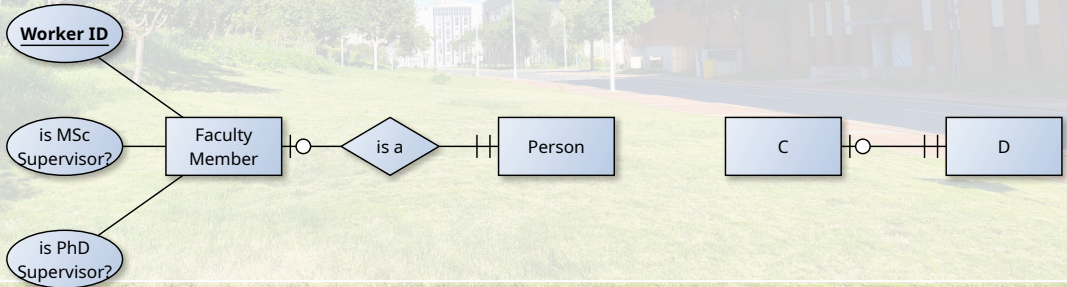


- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.

C  D



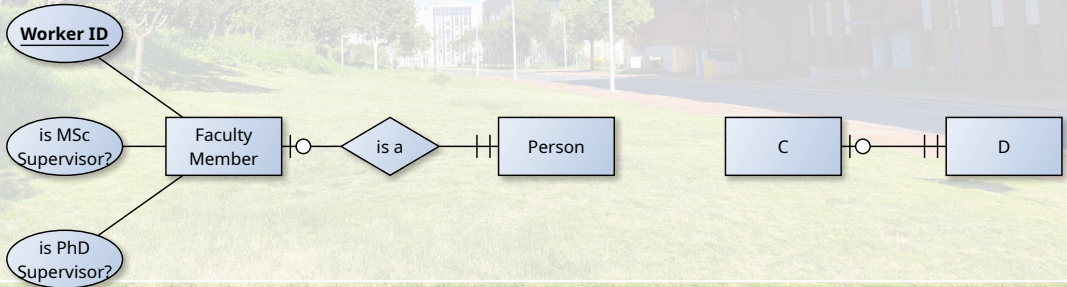
- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.
- Beispiel:



C  D



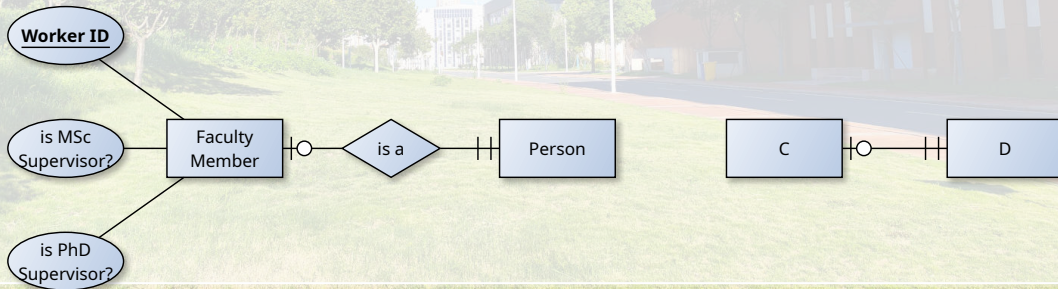
- Es gibt zwei Entitätstypen C und D.
- Jede Entität vom Typ C muss mit genau einer Entität vom Typ D verbunden sein.
- Jede Entität vom Typ D ist mit keiner oder einer Entität vom Typ C verbunden.
- Beispiel:
 - Jede Entität vom Typ *Person* kann ein Mitarbeiter (*Faculty*) der Uni sein ... oder auch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



C D: Lösungen



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

C D: Lösungen



- 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*.

C D: Lösungen



- 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.

C D: Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle  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.
```

C D: Zwei-Tabellen Lösung

- Wir brauchen eine Tabelle  für die Entitäten vom Typ C.
- Wir brauchen auch eine Tabelle  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  CREATE TABLE
5  ALTER TABLE
6  # psql 16.11 succeeded with exit code 0.
```

C D: 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      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.
```

C D: 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-|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.
```

C D: 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-|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.
```

C D: 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.

```
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.
```

C D: Zwei-Tabellen Lösung

- 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** referenzieren.

```
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.
```

C D: 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-|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'), ('IJ');
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 5
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.
```

C D: 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-|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'), ('IJ');
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 5
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.
```

C D: 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-|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'), ('IJ');
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 5
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.
```

C D: 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'), ('IJ');
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 5
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.
```

C D: 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-|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'), ('IJ');
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 5
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.
```

C $\oplus$ $\ominus$ $\dashv$ D: 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“
5	„IJ“

C $\vdash$ $\circ$ $\dashv$ D: Testen der Einschränkungen

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



C D: 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.



C D: 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.
```

C $\oplus$ $\ominus$ $\dashv$ D: 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.
```

C D: 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.
```

C D: 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.
```

C D: 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.
```



$$E \vdash \bigcirc - \bigcirc \leq F$$



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

- 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.

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

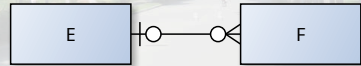
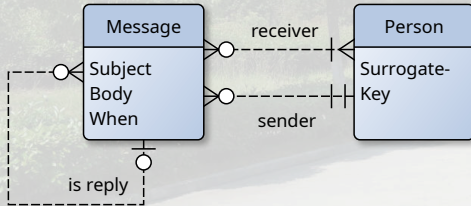


- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.

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



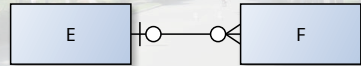
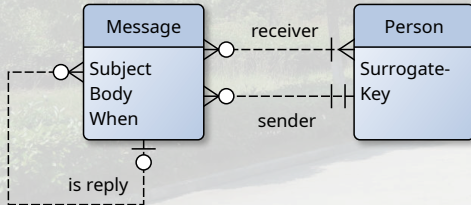
- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:



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



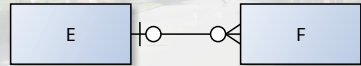
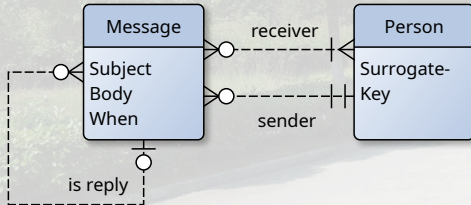
- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:
 - Eine Nachricht kann entweder eine neue Nachricht sein, oder die Antwort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



E $\perp$ ○ — ○ $\times$ F



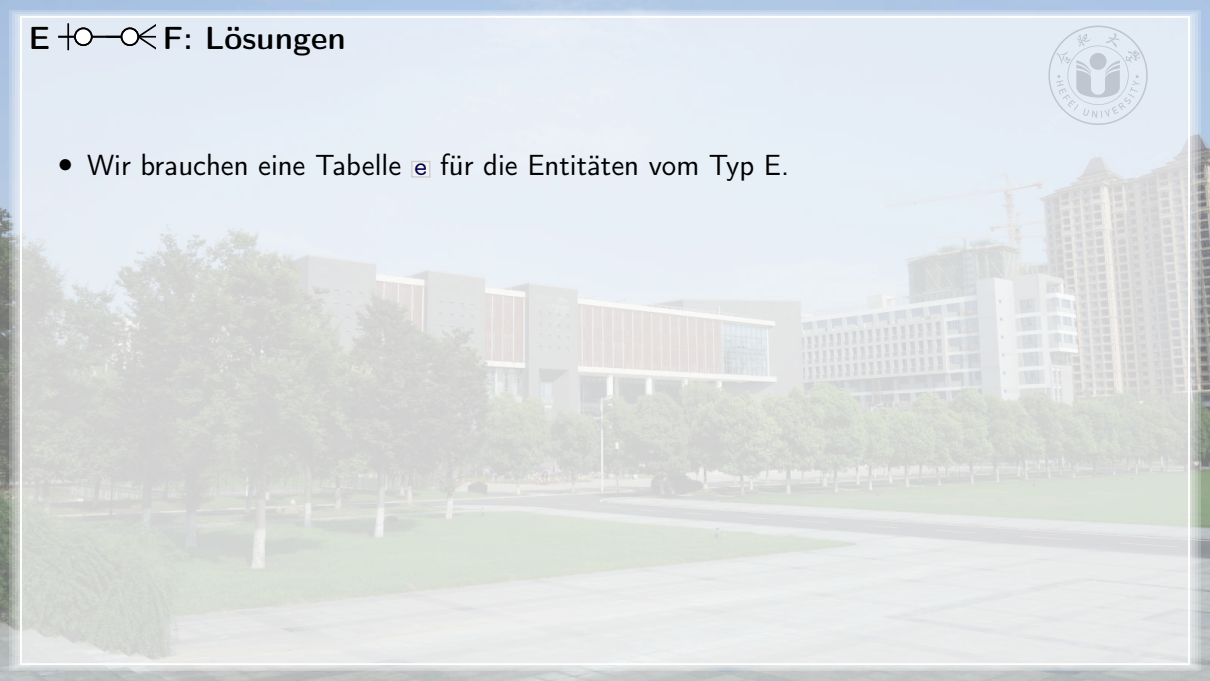
- Es gibt zwei Entitätstypen E und F.
- Jede Entität vom Typ E kann mit keiner, einer, oder mehreren Entitäten vom Typ F verbunden sein.
- Jede Entität vom Typ F ist mit keiner oder einer Entität vom Typ E verbunden.
- Beispiel:
 - Eine Nachricht kann entweder eine neue Nachricht sein, oder die Antwort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



E F: Lösungen



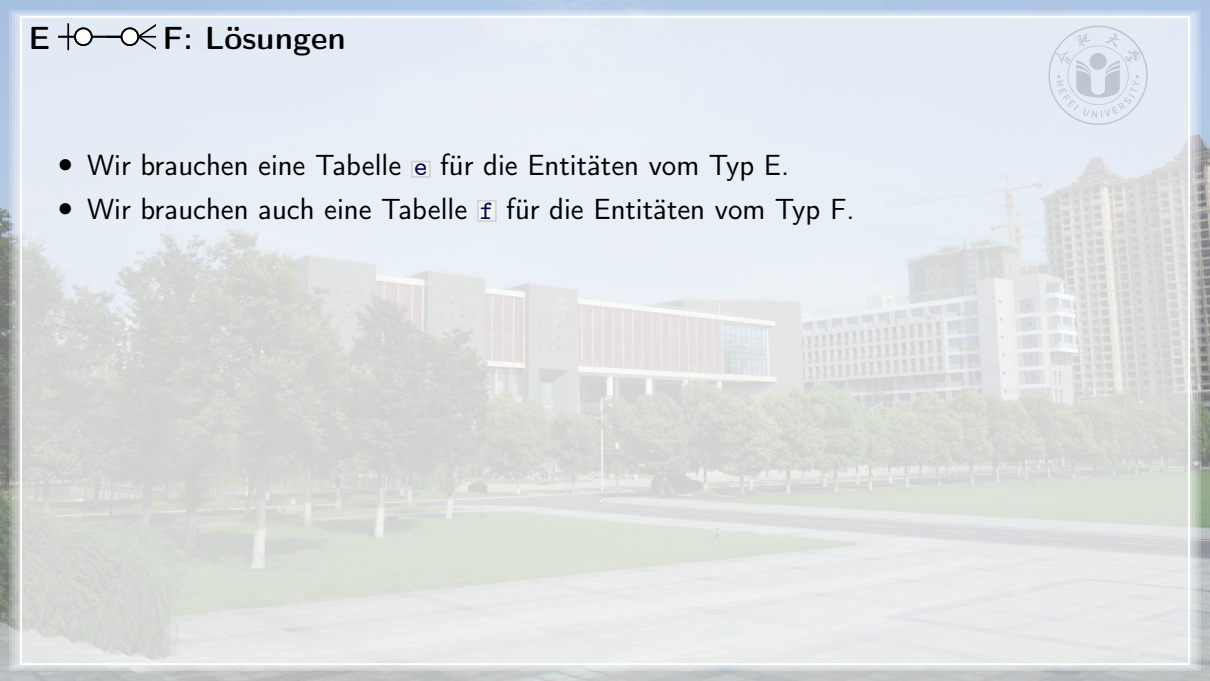
- Wir brauchen eine Tabelle  für die Entitäten vom Typ E.



E F: Lösungen



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



E $\oplus$ $\ominus$ 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**.

E $\oplus$ $\ominus$ 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**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.

E $\oplus$ $\ominus$ 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**.
- Wir fügen auch wieder die Beispielattribute **x** und **y** hinzu.
- Wieder gibt es zwei mögliche Lösungen für dieses Szenario.

E $\oplus$ $\ominus$ 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**.
- 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.

E $\oplus$ 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**.
- 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.

E $\oplus$ --- $\otimes$ 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**.
- 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.

E $\oplus$ o $\ominus$ F: 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.
```

E $\oplus$ o $\ominus$ F: 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.
```

E $\oplus$ o $\ominus$ F: 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.
```

E $\oplus$ o $\ominus$ F: 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 );
```

```
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.
```

E $\oplus$ ○ $\rightarrow$ ○ $\times$ F: 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 );
```

```
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.
```

E $\oplus$ o $\ominus$ $\times$ F: 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.
```

E $\oplus$ o $\ominus$ $\times$ F: 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.
```

E $\oplus$ ○ --- ○ $\leq$ F: 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 **fkeid** 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     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.
```

E ⊕ F: 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.
```

E F: 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.
```

E F: 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.
```

E F: 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.
- Die Daten können dann wieder über zwei `JOIN`s zusammengesetzt werden.

```
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.
```

E $\oplus$ F: 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

$E \vdash \bigcirc \text{---} \bigcirc \vdash F$: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] empfohlen.



E $\oplus$ $\ominus$ $\otimes$ F: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] empfohlen.
- Er hat aber wieder genau die selben Probleme, wie der drei-Tabellen-Ansatz für die Entitätstypen A und B.



E $\oplus$ $\ominus$ $\otimes$ F: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] 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.

$E \text{ --- } F$: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] 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`.

E $\oplus$ $\ominus$ $\otimes$ F: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] 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.

E $\oplus$ $\ominus$ $\otimes$ F: Probleme mit der Drei-Tabellen-Methode



- Dieser Ansatz wird z. B. in [65] 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.

E $\vdash$ $\bigcirc$ $\dashv$ $\bigcirc$ $\dashv$ F: Zwei-Tabellen Lösung

- Probieren wir also eine Lösung mit zwei Tabellen.



E F: 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.
```

- 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-|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.
```

E $\oplus$ $\bigcirc$ $\bigcirc$ $\leq$ F: 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-|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
17 $ psql "postgres://postgres:XXX@localhost/relationships" -v
18     ↪ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
19
20 CREATE TABLE
21 CREATE TABLE
22 # psql 16.11 succeeded with exit code 0.
```

E $\begin{smallmatrix} \text{---} \circ \text{---} \end{smallmatrix}$ $\begin{smallmatrix} \text{---} \circ \text{---} \end{smallmatrix}$ F: 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-|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
17 $ psql "postgres://postgres:XXX@localhost/relationships" -v
18     ↪ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
19
20 CREATE TABLE
21 CREATE TABLE
22 # psql 16.11 succeeded with exit code 0.
```

- ```

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
2      ↪ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.

```

```
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.
```

- ```

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
2      ↪ ON_ERROR_STOP=1 -ebf EF_2_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  # psql 16.11 succeeded with exit code 0.

```

```
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.
```

- ```

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 fkeyid 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.

```

```
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.
```

E F: 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.
```

E F: 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.
```

E F: 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.
```

E $\oplus$ F: 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.

```
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.
```

E F: 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.
```

E $\oplus$ $\ominus$ $\otimes$ F: 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.
```

E F: 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.
```

E F: 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
2      ↪ ON_ERROR_STOP=1 -ebf EF_2_insert_and_select.sql
3  INSERT 0 4
4  INSERT 0 4
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 # psql 16.11 succeeded with exit code 0.
```

E $\bowtie$ F: 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  H

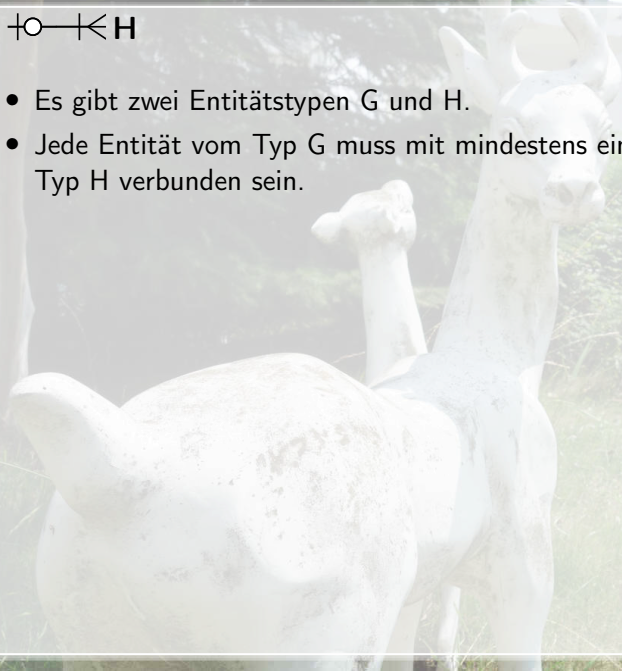
- Es gibt zwei Entitätstypen G und H.



G $\oplus$ $\rightarrow$ 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 $\oplus$ $\rightarrow$ H

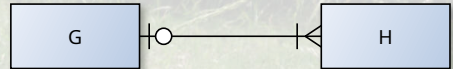


- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.

G  H



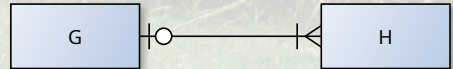
- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.
- Beispiel:



G  H



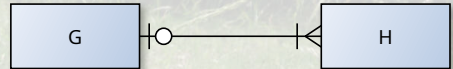
- Es gibt zwei Entitätstypen G und H.
- Jede Entität vom Typ G muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ H verbunden sein.
- Jede Entität vom Typ H ist mit keiner oder einer Entität vom Typ G verbunden.
- Beispiel:
 - In einem Fußballklub trainiert jeder Trainer mehrere Mitglieder.



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.



G $\oplus$ $\dashv$ H: 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
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.
```

G $\oplus$ $\dashv$ H: 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_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.
```

$G \oplus \bigcirc \dashv \vdash H$: 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 $E \oplus \bigcirc \dashv \vdash F$ -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
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.
```

G $\oplus$ $\leftarrow$ H: 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 $\oplus$ $\leftarrow$ 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.
```

G $\oplus$ $\leftarrow$ H: 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 $\oplus$ $\leftarrow$ 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.
```

G $\oplus$ $\rightarrow$ H: Lösung

- Sie haben die Primärschlüssel `gid` und `hid`, sowie die Beispielattribute `x` und `y`.
- Wir können die Situation so ähnlich anpacken wie die E $\oplus$ $\rightarrow$ 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-|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.
```

G $\oplus$ $\dashv$ H: Lösung

- Wir können die Situation so ähnlich anpacken wie die E $\oplus$ $\dashv$ 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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\oplus$ $\rightarrow$ H: 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
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.
```

G $\oplus$ $\rightarrow$ $\leftarrow$ H: 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
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.

```

G $\oplus$ $\leftarrow$ H: 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, 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.
```

G $\rightarrow$ 0 $\leftarrow$ 1 H: 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-|0-----|<-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.
```

G $\rightarrow$ H: 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.
```

G $\oplus$ $\dashv$ $\lhd$ H: 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 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.
- 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.
```

G $\rightarrow$ H: 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->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.
```

G $\oplus$ $\rightarrow$ H: 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.

```
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.
```

G $\oplus$ $\rightarrow$ H: 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
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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G H: 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.
```

G $\oplus$ $\leq$ H: 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
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.
```

G $\oplus$ $\rightarrow$ $\leftarrow$ H: 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, 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.
```

G $\oplus$ $\rightarrow$ H: 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
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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\oplus$ $\leftarrow$ H: 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
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.
```

G $\oplus$ $\rightarrow$ H: 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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\oplus$ $\rightarrow$ $\leftarrow$ H: Lösung

- 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.
- Wir erstellen eine REFERENCES-Einschränkung nicht auf eine einzelne Spalte hid...

```
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.
```

G $\oplus$ $\leftarrow$ H: 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
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.
```

G $\rightarrow$ H: 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-|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.
```

G $\rightarrow$ H: 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->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.
```

G $\rightarrow$ H: 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-|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.
```

G $\rightarrow$ H: 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.
```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\rightarrow$ H: Lösung

- 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.
- Weil Fremdschlüssel-`REFERENCES`-Einschränkungen nur Spalten referenzieren können, die `UNIQUE` sind,

```
1  /* Create the tables for a G->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.
```

G $\rightarrow$ H: 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->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      fkgid  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
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.
```

G H: 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.
```

G $\rightarrow$ H: 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
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.
```

G $\oplus$ $\dashv$ $\lhd$ H: Lösung

- 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
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.
```

G $\oplus$ $\dashv$ H: 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.
```

G $\rightarrow$ H: 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->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.
```

G $\rightarrow$ H: 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_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.
```

$G \nrightarrow \bigcirc \dashv \lhd H$: 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.
```

$G \nrightarrow | \leftarrow H$: 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.
```

G $\oplus$ $\rightarrow$ $\leq$ H: 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.

```

G $\oplus$ $\leftarrow$ H: 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.
```

G $\oplus$ $\leq$ H: 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      fkgid    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.
```

$G \vdash \vdash H$: Testen der Einschränkungen

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



G $\oplus$ $\dashv$ $\dashv$ H: 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.
```

G $\oplus$ $\ominus$ $\dashv$ H: 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.
```

G $\oplus$ $\dashv$ $\lhd$ H: 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? */  
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.
```

G H: 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?

G $\oplus$ $\dashv$ $\dashv$ H: 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 **fk_{hid}** 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.

G $\oplus$ --- $\leftarrow$ H: Testen der Einschränkungen



- Nein, können wir nicht.
- Damit wir eine Zeile in Tabelle g einfügen können, müssen wir den Fremdschlüssel fk_{hid} 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.

G H: 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');

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

G H: 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');

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

$G \vdash \bigcirc \dashv \vdash H$: Wie können wir überhaupt Daten einfügen?



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



$G \vdash \bigcirc \dashv \vdash H$: 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.

$G \oplus \bigcirc \dashv \vdash H$: 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 keine existierende Zeile in Tabelle h referenziert, die noch mit keiner Zeile in Tabelle g verbunden ist.

$G \oplus \bigcirc \dashv \vdash H$: 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 keine 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.

$G \oplus \bigcirc \dashv \vdash H$: 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 keine 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.

G $\oplus$ $\rightarrow$ H: 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 keine 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. ...

$G \oplus \bigcirc \dashv \vdash H$: 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 keine 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 $G \oplus \bigcirc \dashv \vdash H$ implementieren

$G \oplus \bigcirc \dashv \vdash H$: 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 keine 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 $G \oplus \bigcirc \dashv \vdash H$ implementieren
- ... aber wie bekommen wir überhaupt Daten in Tabelle g ?

$G \oplus \bigcirc \dashv \vdash H$: 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 keine 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 $G \oplus \bigcirc \dashv \vdash H$ implementieren
- ... aber wie bekommen wir überhaupt Daten in Tabelle g ?
- Wir müssen dieses komische Henne-Ei-Problem lösen.

G H: Befüllen mit Daten

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

```
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)
```

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

G H: 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 -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.
```

G H: 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
18  # psql 16.11 succeeded with exit code 0.
```

G H: 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  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.
```

G H: 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. . .

G H: 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.
```

G H: 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 --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.
```

G H: 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 Transaktion⁷⁵.

```
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.
```

G H: 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 Transaktion⁷⁵.
- Das heist, dass Einschränkungen zur referentiellen Integrität erst *am Ende* des Kommandos überprüft werden.

```
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
```

G H: 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 Transaktion⁷⁵.
- 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  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.
```

G H: Befüllen mit Daten

- Ein einzelnes Kommando in PostgreSQL ist auch eine Transaktion⁷⁵.
- 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-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.
```

G H: 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)
```

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

G H: 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  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.
```

G H: 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.
```

G H: 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-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.
```

G H: 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.
```

G H: 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-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.
```

G H: 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.
```

G H: 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-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.
```

G H: 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^{59,77}.

```
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.
```

G H: 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^{59,77}.
- Das geht nicht bei allen DBMSen, aber bei MariaDB⁴² und SQLite⁶⁰ geht es 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.
```

G H: Befüllen mit Daten

- In PostgreSQL, das geht mit dem `RETURNING`-Schlüsselwort^{59,77}.
- Das geht nicht bei allen DBMSen, aber bei MariaDB⁴² und SQLite⁶⁰ 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-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.
```

G H: Befüllen mit Daten

- Das geht nicht bei allen DBMSen, aber bei MariaDB⁴² und SQLite⁶⁰ 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  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)
18
19  # psql 16.11 succeeded with exit code 0.
```

G H: 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-|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.
```

G H: 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-|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)
```

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

G H: 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 fkgid = (INSERT INTO g (fkhid, x) VALUES (1, '123') RETURNING gid, fkhid) WHERE h.hid = 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
   ↪ 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.
```

G H: 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 fkgid = (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-|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.
```

G H: 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 fkgid = (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-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.
```

G H: 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 fkgid = (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.
```

G H: Befüllen mit Daten

- Eine Idee wäre es, zu versuchen eine Anfrage wie folgt zu bauen: `UPDATE h SET fkgid = (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
2      ↪ ON_ERROR_STOP=1 --ebf GH_insert_and_select.sql
3
4  INSERT 0 6
5  UPDATE 1
6  UPDATE 1
7  UPDATE 1
8  UPDATE 1
9  UPDATE 1
10
11  gid | x   | hid | y
12  ----+---+----+---
13  3   | 123 | 1   | AB
14  4   | 456 | 3   | EF
15  5   | 789 | 4   | GH
16  3   | 123 | 2   | CD
17  3   | 123 | 5   | IJ
18
19  (5 rows)
20
21  # psql 16.11 succeeded with exit code 0.
```

G H: 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  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)
18
19  # psql 16.11 succeeded with exit code 0.
```

G H: 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.
```

G H: 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
2  ↪ ON_ERROR_STOP=1 -ebf GH_insert_and_select.sql
3
4  INSERT 0 6
5  UPDATE 1
6  UPDATE 1
7  UPDATE 1
8  UPDATE 1
9  UPDATE 1
10
11  gid | x   | hid | y
12  ----+----+----+---
13  3   | 123 | 1   | AB
14  4   | 456 | 3   | EF
15  5   | 789 | 4   | GH
16  3   | 123 | 2   | CD
17  3   | 123 | 5   | IJ
18
19  (5 rows)
20
21  # psql 16.11 succeeded with exit code 0.
```

G H: 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)
16
17  # psql 16.11 succeeded with exit code 0.
```

G H: 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
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.
```

G H: 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-|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.
```

G H: 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-|-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.
```

G H: 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-|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.
```

G H: 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
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.
```

G H: 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-|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.
```

G H: 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.
```

G H: 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
17  # psql 16.11 succeeded with exit code 0.
```

G H: 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-|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
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)
```

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

G H: 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
17 # psql 16.11 succeeded with exit code 0.
```

G H: 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.
```

G H: 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-|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.
```

G H: 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-|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.
```

G H: 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.
```

G H: Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

Table g		
gid	fkhid	x
3	1	„123“
4	3	„456“
5	4	„789“

Table h		
hid	fkgid	y
6	NULL	„KL“
1	3	„AB“
3	4	„EF“
4	5	„GH“
2	3	„CD“
5	3	„IJ“

$G \oplus \ominus \dashv \vdash \Leftarrow H$: 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.
```

$G \oplus \ominus \dashv \vdash \leq H$: 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.
```

$G \oplus \ominus \dashv \vdash \leq H$: 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.
```

G $\oplus$ $\dashv$ $\lhd$ H: 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;
8
9
10 $ psql "postgres://postgres:XXX@localhost/relationships" -v
    ↳ ON_ERROR_STOP=1 -ebf GH_insert_error_4.sql
psql:conceptualToRelational/GH_insert_error_4.sql:7: ERROR:  duplicate
    ↳ key value violates unique constraint "g_fkhid_key"
DETAIL:  Key (fkhid)=(4) already exists.
psql:conceptualToRelational/GH_insert_error_4.sql:7: STATEMENT:  /* Can
    ↳ we insert a G that is related to a H related to another G? */
-- H with id 4 is already related to G with id 3.
-- 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.
```

G H: 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.
```

G $\oplus$ $\ominus$ $\rightarrow$ $\leftarrow$ H: 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**)?

G H: 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.

G H: 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.
```

G H: 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.
```



I ||| J



I ||—|| J

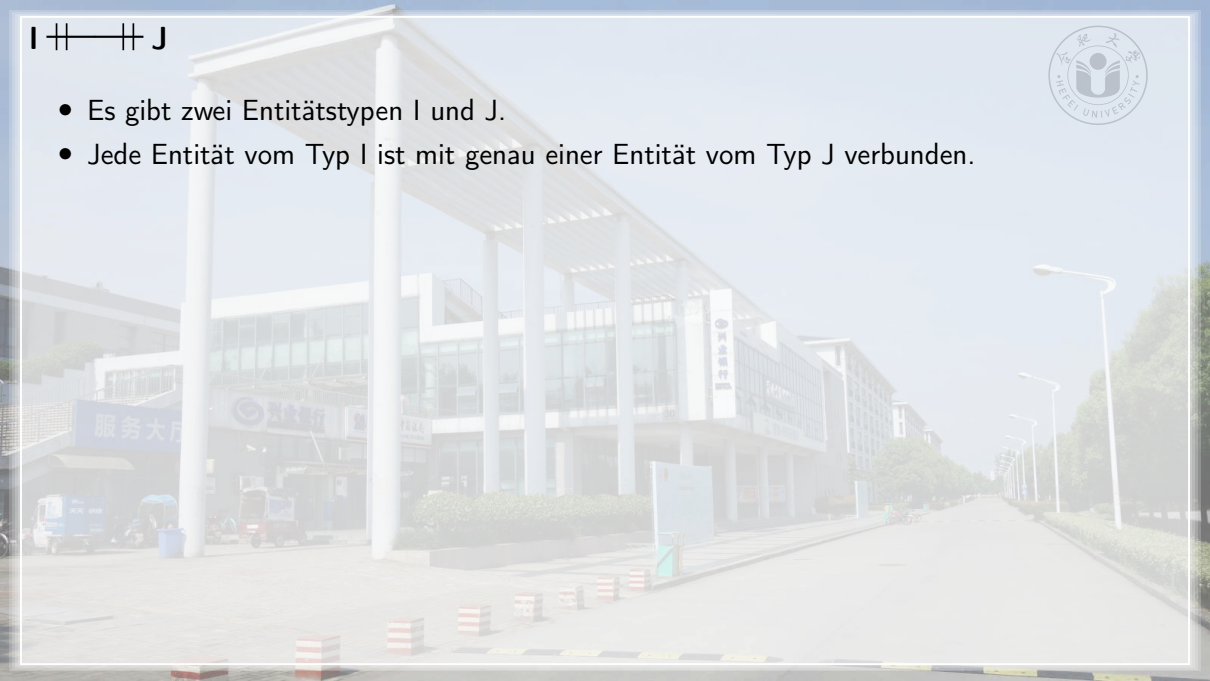
- Es gibt zwei Entitätstypen I und J.



I — J



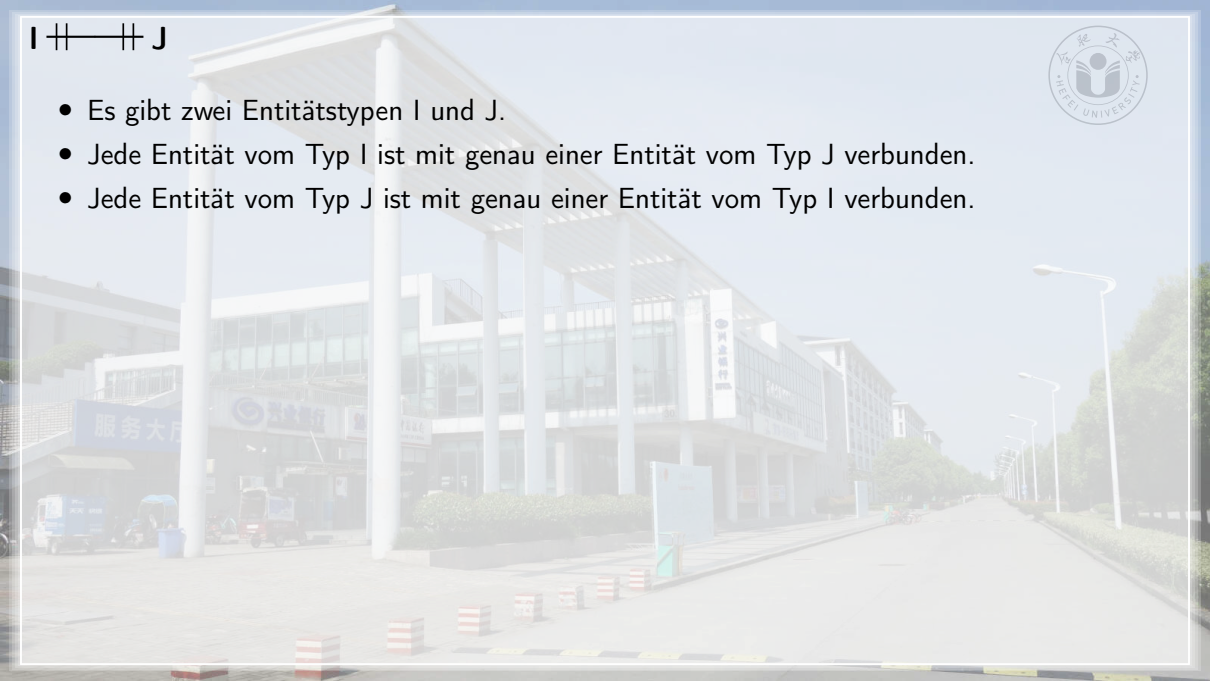
- Es gibt zwei Entitätstypen I und J.
- Jede Entität vom Typ I ist mit genau einer Entität vom Typ J verbunden.



I — J



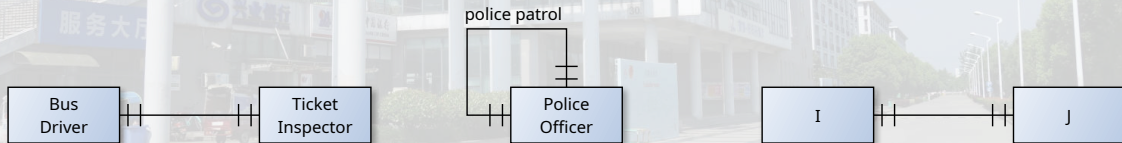
- Es gibt zwei Entitätstypen I und J.
- Jede Entität vom Typ I ist mit genau einer Entität vom Typ J verbunden.
- Jede Entität vom Typ J ist mit genau einer Entität vom Typ I verbunden.



I ||—|| J



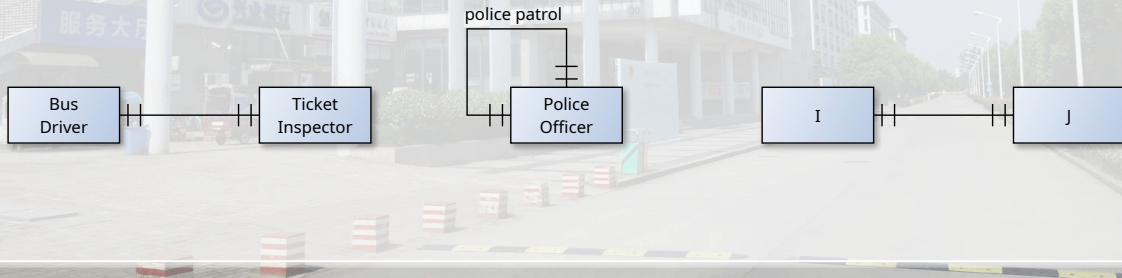
- Es gibt zwei Entitätstypen I und J.
- Jede Entität vom Typ I ist mit genau einer Entität vom Typ J verbunden.
- Jede Entität vom Typ J ist mit genau einer Entität vom Typ I verbunden.
- Beispiel:



I ||—|| J



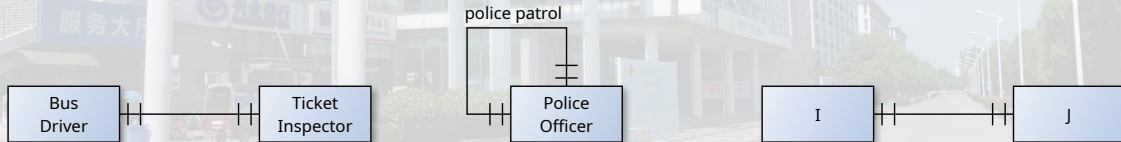
- Es gibt zwei Entitätstypen I und J.
- Jede Entität vom Typ I ist mit genau einer Entität vom Typ J verbunden.
- Jede Entität vom Typ J ist mit genau einer Entität vom Typ I verbunden.
- Beispiel:
 - In klassischen Polizeifilmen gibt es immer Paare von zwei Polizisten, die zusammen auf Streife gehen.



I ||—|| J



- Es gibt zwei Entitätstypen I und J.
- Jede Entität vom Typ I ist mit genau einer Entität vom Typ J verbunden.
- Jede Entität vom Typ J ist mit genau einer Entität vom Typ I verbunden.
- Beispiel:
 - In klassischen Polizeifilmen gibt es immer Paare von zwei Polizisten, die zusammen auf Streife gehen.
 - Wir können uns auch Paare von einer Busfahrer und einem Fahrscheinkontrolleur vorstellen, die zusammen auf einer Buslinie arbeiten.



I || J: Lösung

- In diesem Szenario brauchen wir nur eine einzige Tabelle⁶⁵.

```
1  /* Create the table for an I-||-----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x  CHAR(3), -- example of other attributes of entity type I
7      y  CHAR(2)  -- example of other attributes of entity type J
8  );
```

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

I || J: Lösung

- In diesem Szenario brauchen wir nur eine einzige Tabelle⁶⁵.
- Wir fügen die Attribute der beiden Entitätstypen zusammen und tun sie alle in eine gemeinsame Tabelle.

```
1  /* Create the table for an I-||-----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x  CHAR(3), -- example of other attributes of entity type I
7      y  CHAR(2)  -- example of other attributes of entity type J
8  );
```

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

I — J: Lösung

- In diesem Szenario brauchen wir nur eine einzige Tabelle⁶⁵.
- Wir fügen die Attribute der beiden Entitätstypen zusammen und tun sie alle in eine gemeinsame Tabelle.
- Wenn jedes in Beziehung stehende Paar von I- und J-Entitäten in einer Zeile zusammensteht, dann kann es niemals Probleme mit der referentiellen Integrität geben.

```
1  /* Create the table for an I-||-----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x  CHAR(3), -- example of other attributes of entity type I
7      y  CHAR(2)  -- example of other attributes of entity type J
8  );
```

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

I || J: Lösung

- In diesem Szenario brauchen wir nur eine einzige Tabelle⁶⁵.
- Wir fügen die Attribute der beiden Entitätstypen zusammen und tun sie alle in eine gemeinsame Tabelle.
- Wenn jedes in Beziehung stehende Paar von I- und J-Entitäten in einer Zeile zusammensteht, dann kann es niemals Probleme mit der referentiellen Integrität geben.
- Wir können niemals eine Zeile ohne I-Entität oder eine Zeile ohne J-Entität haben.

```
1  /* Create the table for an I-||-----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x  CHAR(3), -- example of other attributes of entity type I
7      y  CHAR(2)  -- example of other attributes of entity type J
8  );
```

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

I — J: Lösung

- In diesem Szenario brauchen wir nur eine einzige Tabelle⁶⁵.
- Wir fügen die Attribute der beiden Entitätstypen zusammen und tun sie alle in eine gemeinsame Tabelle.
- Wenn jedes in Beziehung stehende Paar von I- und J-Entitäten in einer Zeile zusammensteht, dann kann es niemals Probleme mit der referentiellen Integrität geben.
- Wir können niemals eine Zeile ohne I-Entität oder eine Zeile ohne J-Entität haben.
- Weil das so einfach ist, üben wir diesmal nicht das Einfügen von Daten.

```
1  /* Create the table for an I-||-----||-J relationship. */
2
3  -- Table I_J: Each row holds one entity of type I and one of type J.
4  CREATE TABLE i_j (
5      id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x  CHAR(3), -- example of other attributes of entity type I
7      y  CHAR(2)  -- example of other attributes of entity type J
8  );
```

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



K $\parallel$ $\bigcirc$ L



K $\vdash$ $\circ$ L

- Es gibt zwei Entitätstypen K und L.



K  L



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

K  L

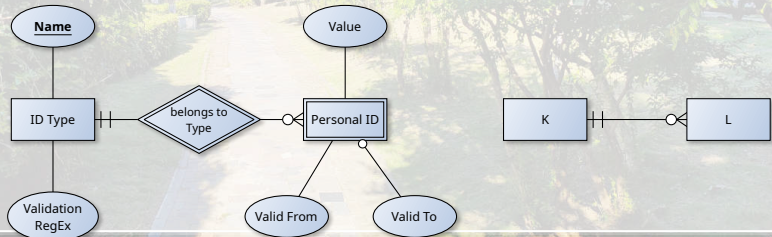


- Es gibt zwei Entitätstypen K und L.
- Jede Entität vom Typ K kann mit keiner, einer, oder mehreren Entitäten vom Typ L verbunden sein.
- Jede Entität vom Typ L ist mit genau einer Entität vom Typ K verbunden.

K ||—○< L



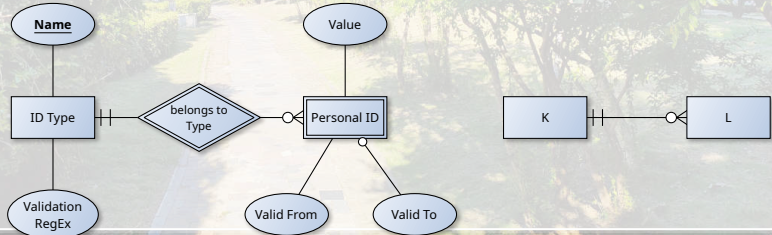
- Es gibt zwei Entitätstypen K und L.
- Jede Entität vom Typ K kann mit keiner, einer, oder mehreren Entitäten vom Typ L verbunden sein.
- Jede Entität vom Typ L ist mit genau einer Entität vom Typ K verbunden.
- Beispiel:



K ||—○< L



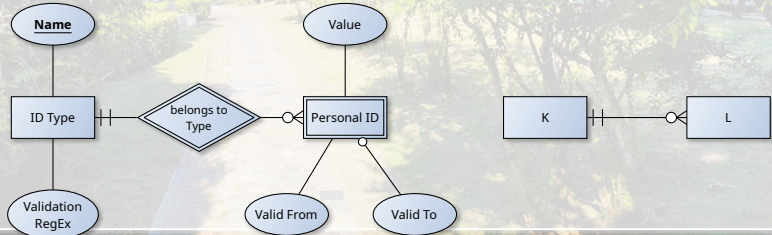
- Es gibt zwei Entitätstypen K und L.
- Jede Entität vom Typ K kann mit keiner, einer, oder mehreren Entitäten vom Typ L verbunden sein.
- Jede Entität vom Typ L ist mit genau einer Entität vom Typ K verbunden.
- Beispiel:
 - Wir erinnern uns, dass es in unserem System viele verschiedene Arten von IDs gibt, z. B. chinesische Ausweisnummern (中国公民身份号码), Reisepässe, Visas, oder sogar Mobiltelefonnummern.



K ||—○< L



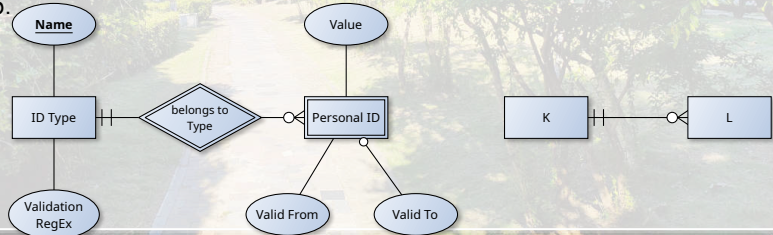
- Es gibt zwei Entitätstypen K und L.
- Jede Entität vom Typ K kann mit keiner, einer, oder mehreren Entitäten vom Typ L verbunden sein.
- Jede Entität vom Typ L ist mit genau einer Entität vom Typ K verbunden.
- Beispiel:
 - Wir erinnern uns, dass es in unserem System viele verschiedene Arten von IDs gibt, z. B. chinesische Ausweisnummern (中国公民身份号码), Reisepässe, Visas, oder sogar Mobiltelefonnummern. Für jeden solchen ID-Typ, können keine, eine, oder viele persönliche ID-Datensätze in der Datenbank gespeichert sein.



K ||—○< L



- Es gibt zwei Entitätstypen K und L.
- Jede Entität vom Typ K kann mit keiner, einer, oder mehreren Entitäten vom Typ L verbunden sein.
- Jede Entität vom Typ L ist mit genau einer Entität vom Typ K verbunden.
- Beispiel:
 - Wir erinnern uns, dass es in unserem System viele verschiedene Arten von IDs gibt, z. B. chinesische Ausweisnummern (中国公民身份号码), Reisepässe, Visas, oder sogar Mobiltelefonnummern. Für jeden solchen ID-Typ, können keine, eine, oder viele persönliche ID-Datensätze in der Datenbank gespeichert sein. Jede persönliche ID gehört jedoch immer zu genau einem ID-Typ.



K L: Lösung



- Wir brauchen eine Tabelle **k** für die Entitäten vom Typ K und eine Tabelle **l** für die Entitäten vom Typ L.

```
1  /* Create the tables for a K-||-----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid   INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y     CHAR(2) -- example of other attributes
14 );
```

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

K L: Lösung



- Wir brauchen eine Tabelle **k** für die Entitäten vom Typ K und eine Tabelle **l** für die Entitäten vom Typ L.
- Wir nennen den Primärschlüssel von Tabelle **k** **kid** und wir fügen wieder unser Beispielattribut **x** hinzu.

```
1  /* Create the tables for a K-||-----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y    CHAR(2) -- example of other attributes
14 );
```

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

K L: Lösung



- Wir brauchen eine Tabelle **k** für die Entitäten vom Typ K und eine Tabelle **l** für die Entitäten vom Typ L.
- Wir nennen den Primärschlüssel von Tabelle **k** **kid** und wir fügen wieder unser Beispielattribut **x** hinzu.
- Der Primärschlüssel der Tabelle **l** ist **lid** und es gibt hier das Beispielattribut **y**.

```
1  /* Create the tables for a K-||----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y    CHAR(2) -- example of other attributes
14 );
```

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

K L: Lösung



- Wir brauchen eine Tabelle **k** für die Entitäten vom Typ K und eine Tabelle **l** für die Entitäten vom Typ L.
- Wir nennen den Primärschlüssel von Tabelle **k** **kid** und wir fügen wieder unser Beispielattribut **x** hinzu.
- Der Primärschlüssel der Tabelle **l** ist **lid** und es gibt hier das Beispielattribut **y**.
- Jede Zeile in Tabelle **l** muss mit genau einer Zeile in Tabelle **k** in Verbindung stehen.

```
1  /* Create the tables for a K-||-----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y    CHAR(2) -- example of other attributes
14 );
```

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

- ```

1 /* Create the tables for a K-||---o<-L relationship. */
2
3 -- Table K: Each row in K is related to zero or one or many rows in L.
4 CREATE TABLE k (
5 kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11 lid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12 fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13 y CHAR(2) -- example of other attributes
14);

```
- ```

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

```

K $\text{||} \text{---} \text{O} \text{<} \text{L}$: Lösung



- Wir nennen den Primärschlüssel von Tabelle **k** **kid** und wir fügen wieder unser Beispielattribut **x** hinzu.
- Der Primärschlüssel der Tabelle **l** ist **lid** und es gibt hier das Beispielattribut **y**.
- Jede Zeile in Tabelle **l** muss mit genau einer Zeile in Tabelle **k** in Verbindung stehen.
- Die Drei-Tabellen-Lösung vom E $\text{+} \text{O} \text{---} \text{O} \text{<} \text{F}$ -Szenario würde hier keinen Sinn ergeben.
- Die dritte Tabelle könnte niemals wenige Zeilen haben – sie würde immer genauso viele Zeilen wie Tabelle **l** haben.

```
1  /* Create the tables for a K-||----O<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid  INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y    CHAR(2) -- example of other attributes
14 );
```

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

K $\begin{array}{c} \text{||} \\ \text{---} \end{array} \begin{array}{c} \text{O} \\ \text{---} \end{array} \text{L: Lösung}$



- Jede Zeile in Tabelle **l** muss mit genau einer Zeile in Tabelle **k** in Verbindung stehen.
- Die Drei-Tabellen-Lösung vom $E \begin{array}{c} \text{+} \\ \text{---} \end{array} \begin{array}{c} \text{O} \\ \text{---} \end{array} \text{F}$ -Szenario würde hier keinen Sinn ergeben.
- Die dritte Tabelle könnte niemals wenige Zeilen haben – sie würde immer genauso viele Zeilen wie Tabelle **l** haben.
- Die vernünftige Lösung ist also, eine Fremdschlüssel-Spalte an Tabelle **l** anhängen, die den Primärschlüssel von Tabelle **k** referenziert.

```
1  /* Create the tables for a K-||----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y   CHAR(2) -- example of other attributes
14 );
```

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

K ||--o< L: Lösung



- Jede Zeile in Tabelle **l** muss mit genau einer Zeile in Tabelle **k** in Verbindung stehen.
- Die Drei-Tabellen-Lösung vom $E \text{ } \text{---} \text{ } \text{---} \text{ } F$ -Szenario würde hier keinen Sinn ergeben.
- Die dritte Tabelle könnte niemals wenige Zeilen haben – sie würde immer genauso viele Zeilen wie Tabelle **l** haben.
- Die vernünftige Lösung ist also, eine Fremdschlüssel-Spalte an Tabelle **l** anhängen, die den Primärschlüssel von Tabelle **k** referenziert.
- Weil die Beziehung zwingend ist, muss die Spalte **NOT NULL** sein.

```
1  /* Create the tables for a K-||----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x   CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y   CHAR(2) -- example of other attributes
14 );
15
16 $ psql "postgres://postgres:XXX@localhost/relationships" -v
17 ↪ ON_ERROR_STOP=1 -ebf KL_tables.sql
18
19 CREATE TABLE
20 CREATE TABLE
21 # psql 16.11 succeeded with exit code 0.
```

K ||--o< L: Lösung



- Die dritte Tabelle könnte niemals wenige Zeilen haben – sie würde immer genauso viele Zeilen wie Tabelle **1** haben.
- Die vernünftige Lösung ist also, eine Fremdschlüssel-Spalte an Tabelle **1** anhängen, die den Primärschlüssel von Tabelle **k** referenziert.
- Weil die Beziehung zwingend ist, muss die Spalte **NOT NULL** sein.
- Sie muss nicht **UNIQUE** sein, denn jede Zeile in Tabelle **k** kann mit beliebig vielen Zeilen in Tabelle **1** verbunden sein.

```
1  /* Create the tables for a K-||----o<-L relationship. */
2
3  -- Table K: Each row in K is related to zero or one or many rows in L.
4  CREATE TABLE k (
5      kid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      x    CHAR(3) -- example of other attributes
7  );
8
9  -- Table L: Each row in L is related to exactly one row in K.
10 CREATE TABLE l (
11     lid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
12     fkkid INT NOT NULL REFERENCES k (kid), -- the foreign key to K.
13     y    CHAR(2) -- example of other attributes
14 );
```

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

K L: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.



```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                   ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



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

K L: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Wir erstellen zuerst die Zeilen für die Tabelle k, denn deren Beziehung zu den Zeilen in Tabelle l ist optional.



```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                   ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



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



K $\oplus$ L: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Wir erstellen zuerst die Zeilen für die Tabelle **k**, denn deren Beziehung zu den Zeilen in Tabelle **l** ist optional.
- Danach können wir Zeilen in Tabelle **l** einfügen.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf KL_insert_and_select.sql
3  INSERT 0 5
4  INSERT 0 4
5
6  kid | x | lid | y
7  ----+-----+----+--
8  1 | 123 | 1 | AB
9  1 | 123 | 2 | CD
10 4 | 101 | 3 | EF
11 3 | 789 | 4 | GH
12 (4 rows)
```



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



K $\oplus$ L: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Wir erstellen zuerst die Zeilen für die Tabelle **k**, denn deren Beziehung zu den Zeilen in Tabelle **l** ist optional.
- Danach können wir Zeilen in Tabelle **l** einfügen.
- Für jede von ihnen müssen wir einen gültigen Wert für die Spalte **fkid** liefern, also einen Fremdschlüssel zur Tabelle **k**.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkid = k.kid;
```



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



K --- L: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Wir erstellen zuerst die Zeilen für die Tabelle **k**, denn deren Beziehung zu den Zeilen in Tabelle **l** ist optional.
- Danach können wir Zeilen in Tabelle **l** einfügen.
- Für jede von ihnen müssen wir einen gültigen Wert für die Spalte **fkid** liefern, also einen Fremdschlüssel zur Tabelle **k**.
- Dieser Wert muss genau zu einem existierenden Wert in Spalte **kid** der Tabelle **k** passen.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkid = k.kid;
```



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

K L: Befüllen mit Daten

- Wir erstellen zuerst die Zeilen für die Tabelle **k**, denn deren Beziehung zu den Zeilen in Tabelle **l** ist optional.
- Danach können wir Zeilen in Tabelle **l** einfügen.
- Für jede von ihnen müssen wir einen gültigen Wert für die Spalte **fkkid** liefern, also einen Fremdschlüssel zur Tabelle **k**.
- Dieser Wert muss genau zu einem existierenden Wert in Spalte **kid** der Tabelle **k** passen.
- Der Grund ist, dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** in Beziehung stehen muss.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



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

K L: Befüllen mit Daten

- Danach können wir Zeilen in Tabelle **l** einfügen.
- Für jede von ihnen müssen wir einen gültigen Wert für die Spalte **fkkid** liefern, also einen Fremdschlüssel zur Tabelle **k**.
- Dieser Wert muss genau zu einem existierenden Wert in Spalte **kid** der Tabelle **k** passen.
- Der Grund ist, dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** in Beziehung stehen muss.
- **NULL** für **fkkid** anzugeben geht daher nicht.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



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

K ||-----o<-L: Befüllen mit Daten

- Für jede von ihnen müssen wir einen gültigen Wert für die Spalte `fkid` liefern, also einen Fremdschlüssel zur Tabelle `k`.
- Dieser Wert muss genau zu einem existierenden Wert in Spalte `kid` der Tabelle `k` passen.
- Der Grund ist, dass jede Zeile in Tabelle `l` mit genau einer Zeile in Tabelle `k` in Beziehung stehen muss.
- `NULL` für `fkid` anzugeben geht daher nicht.
- Es ist unmöglich, eine Zeile in Tabelle `l` ohne einen existierenden Wert des Primärschlüssels `kid` von Tabelle `k` als Fremdschlüssel `fkid` anzugeben.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkid = k.kid;
```

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



K $\text{||} \text{---} \text{O} \text{<} \text{L}$: Befüllen mit Daten

- Der Grund ist, dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** in Beziehung stehen muss.
- **NULL** für **fkid** anzugeben geht daher nicht.
- Es ist unmöglich, eine Zeile in Tabelle **l** ohne einen existierenden Wert des Primärschlüssels **kid** von Tabelle **k** als Fremdschlüssel **fkid** anzugeben.
- In anderen Worten, wegen den **REFERENCES**- und **NOT NULL**-Einschränkungen, muss jede Zeile in Tabelle **l** genau eine Zeile in Tabelle **k** referenzieren.

```
1  /* Inserting data into the tables for the K-||----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkid = k.kid;
```



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

K $\text{||} \text{---} \text{O} \text{<} \text{L}$: Befüllen mit Daten

- `NULL` für `fkkid` anzugeben geht daher nicht.
- Es ist unmöglich, eine Zeile in Tabelle `l` ohne einen existierenden Wert des Primärschlüssels `kid` von Tabelle `k` als Fremdschlüssel `fkkid` anzugeben.
- In anderen Worten, wegen den `REFERENCES`- und `NOT NULL`-Einschränkungen, muss jede Zeile in Tabelle `l` genau eine Zeile in Tabelle `k` referenzieren.
- Sie kann nicht mehr als eine Zeile referenzieren, denn das Attribut `fkkid` kann nur einen einzelnen Wert annehmen.

```
1  /* Inserting data into the tables for the K-||-----O<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkfid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                     ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkfid = k.kid;
```



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

K ||---o< L: Befüllen mit Daten

- Es ist unmöglich, eine Zeile in Tabelle **l** ohne einen existierenden Wert des Primärschlüssels **kid** von Tabelle **k** als Fremdschlüssel **fkid** anzugeben.
- In anderen Worten, wegen den **REFERENCES**- und **NOT NULL**-Einschränkungen, muss jede Zeile in Tabelle **l** genau eine Zeile in Tabelle **k** referenzieren.
- Sie kann nicht mehr als eine Zeile referenzieren, denn das Attribut **fkid** kann nur einen einzelnen Wert annehmen.
- Die Zeilen der Tabelle **k** können von beliebig vielen Zeilen der Tabelle **l** referenziert werden.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkid = k.kid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf KL_insert_and_select.sql
3  INSERT 0 5
4  INSERT 0 4
5
6  kid | x | lid | y
7  ----+-----+----+--
8  1 | 123 | 1 | AB
9  1 | 123 | 2 | CD
10 4 | 101 | 3 | EF
11 3 | 789 | 4 | GH
12 (4 rows)
```



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

K L: Befüllen mit Daten

- In anderen Worten, wegen den REFERENCES- und NOT NULL-Einschränkungen, muss jede Zeile in Tabelle **l** genau eine Zeile in Tabelle **k** referenzieren.
- Sie kann nicht mehr als eine Zeile referenzieren, denn das Attribut **fkkid** kann nur einen einzelnen Wert annehmen.
- Die Zeilen der Tabelle **k** können von beliebig vielen Zeilen der Tabelle **l** referenziert werden: vielleicht von keiner Zeile, vielleicht von einer Zeile, oder vielleicht von hunderten.

```
1  /* Inserting data into the tables for the K-||-----o<-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



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

K ||--o L: Befüllen mit Daten

- Sie kann nicht mehr als eine Zeile referenzieren, denn das Attribut `fkkid` kann nur einen einzelnen Wert annehmen.
- Die Zeilen der Tabelle `k` können von beliebig vielen Zeilen der Tabelle `l` referenziert werden: vielleicht von keiner Zeile, vielleicht von einer Zeile, oder vielleicht von hunderten.
- Das ist genau die Bedeutung der „optional-viele“-Beziehung, die auf den Entitätstyp `L` zeigt.

```
1  /* Inserting data into the tables for the K-||-----o-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf KL_insert_and_select.sql
3  INSERT 0 5
4  INSERT 0 4
5
6  kid | x | lid | y
7  ----+-----+----+---
8  1 | 123 | 1 | AB
9  1 | 123 | 2 | CD
10 4 | 101 | 3 | EF
11 3 | 789 | 4 | GH
12
13 (4 rows)
14
15 # psql 16.11 succeeded with exit code 0.
```

K ||--o L: Befüllen mit Daten

- Sie kann nicht mehr als eine Zeile referenzieren, denn das Attribut `fkkid` kann nur einen einzelnen Wert annehmen.
- Die Zeilen der Tabelle `k` können von beliebig vielen Zeilen der Tabelle `l` referenziert werden: vielleicht von keiner Zeile, vielleicht von einer Zeile, oder vielleicht von hunderten.
- Das ist genau die Bedeutung der „optional-viele“-Beziehung, die auf den Entitätstyp `L` zeigt.
- Wir können die Daten wieder mit einem einzelnen `INNER JOIN` zusammenziehen.

```
1  /* Inserting data into the tables for the K-||-----o-L relationship. */
2
3  -- Insert some rows into the table for entity type K.
4  INSERT INTO k (x) VALUES ('123'), ('456'), ('789'), ('101'), ('202');
5
6  -- Insert some rows into the table for entity type L, referencing K.
7  INSERT INTO l (y, fkkid) VALUES ('AB', 1), ('CD', 1), ('EF', 4),
8                                  ('GH', 3);
9
10 -- Combine the rows from K and L.
11 SELECT kid, x, lid, y FROM l INNER JOIN k ON l.fkkid = k.kid;
```



```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2      ↪ ON_ERROR_STOP=1 -ebf KL_insert_and_select.sql
3  INSERT 0 5
4  INSERT 0 4
5
6  kid | x | lid | y
7  ----+-----+----+--
8  1 | 123 | 1 | AB
9  1 | 123 | 2 | CD
10 4 | 101 | 3 | EF
11 3 | 789 | 4 | GH
12 (4 rows)
```



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

K $\parallel$ $\circ$ L: Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

Table **k**

kid	x
1	„123“
2	„456“
3	„789“
4	„101“
5	„202“

Table **l**

lid	fkkid	y
1	1	„AB“
2	1	„CD“
3	4	„EF“
4	3	„GH“



$$M \equiv \neg N$$



M $\vdash$ $\dashv$ N

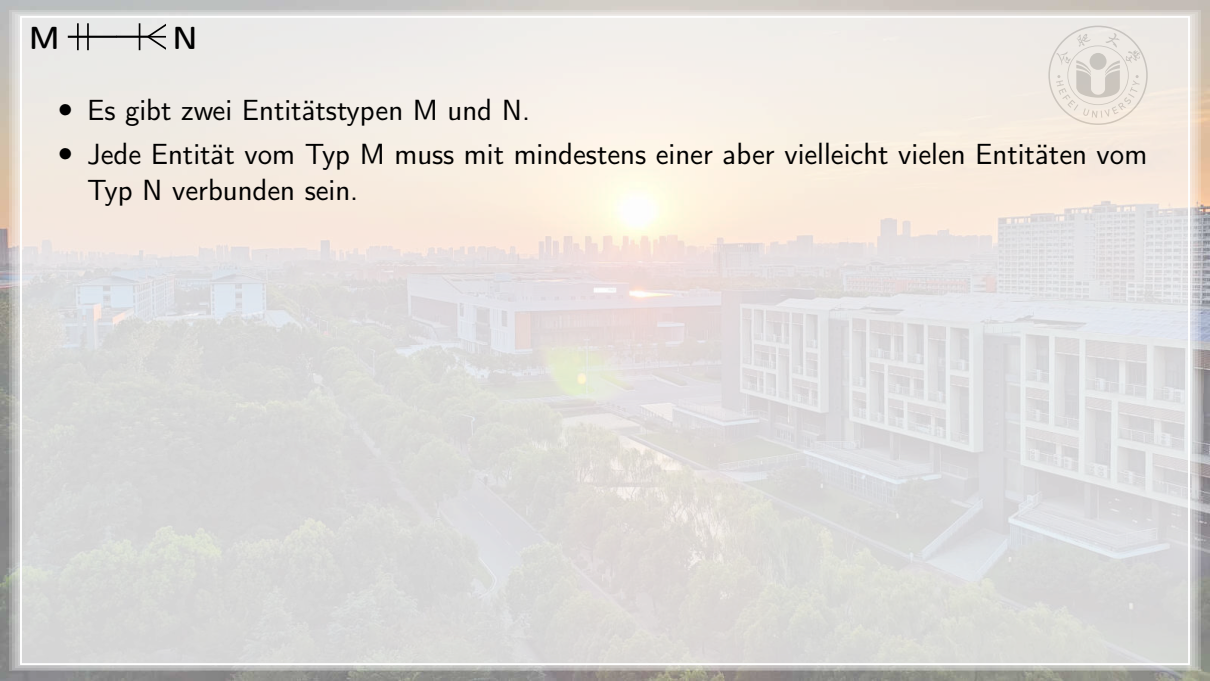
- Es gibt zwei Entitätstypen M und N.



M ---|---|---| N



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



M  N

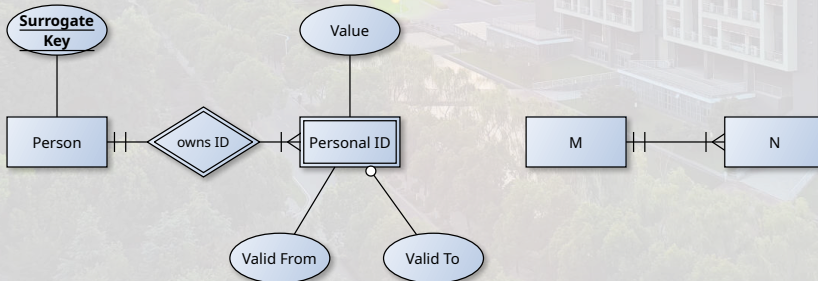


- Es gibt zwei Entitätstypen M und N.
- Jede Entität vom Typ M muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ N verbunden sein.
- Jede Entität vom Typ N ist immer mit genau einer Entität vom Typ M verbunden.

$M \text{ } \text{||} \text{---} \text{||} \text{ } N$



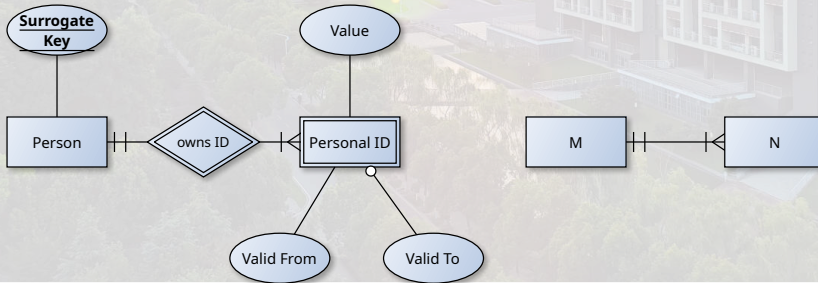
- Es gibt zwei Entitätstypen M und N.
- Jede Entität vom Typ M muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ N verbunden sein.
- Jede Entität vom Typ N ist immer mit genau einer Entität vom Typ M verbunden.
- Beispiel:



$M \text{ } \text{||} \text{---} \text{||} \text{ } N$



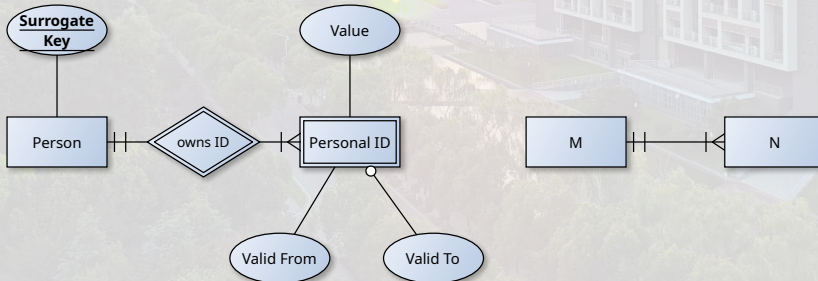
- Es gibt zwei Entitätstypen M und N.
- Jede Entität vom Typ M muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ N verbunden sein.
- Jede Entität vom Typ N ist immer mit genau einer Entität vom Typ M verbunden.
- Beispiel:
 - Jede Person hat mindestens eine persönliche ID.



$M \text{ } \text{---} \text{ } N$



- Es gibt zwei Entitätstypen M und N.
- Jede Entität vom Typ M muss mit mindestens einer aber vielleicht vielen Entitäten vom Typ N verbunden sein.
- Jede Entität vom Typ N ist immer mit genau einer Entität vom Typ M verbunden.
- Beispiel:
 - Jede Person hat mindestens eine persönliche ID. Jede persönliche ID gehört zu genau einer Person.



M N: Lösung

- Die Einschränkungen für die referentielle Integrität in diesem Szenario zu implementieren ist tricky.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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

- Aber machbar.

```

1  /* Create the tables for a M-||-><N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence
9      fknid    INT NOT NULL UNIQUE,    -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);

```

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

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \text{N}$: Lösung

- Die Einschränkungen für die referentielle Integrität in diesem Szenario zu implementieren ist tricky.
- Aber machbar.
- Ich hatte mit G $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \text{H}$ -Szenario zuerst Probleme, aber am Ende haben wir es hinbekommen.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Die Einschränkungen für die referentielle Integrität in diesem Szenario zu implementieren ist tricky.
- Aber machbar.
- Ich hatte mit G  H-Szenario zuerst Probleme, aber am Ende haben wir es hinbekommen.
- Jetzt können wir ähnlich vorgehen.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)              -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Die Einschränkungen für die referentielle Integrität in diesem Szenario zu implementieren ist tricky.
- Aber machbar.
- Ich hatte mit G  H-Szenario zuerst Probleme, aber am Ende haben wir es hinbekommen.
- Jetzt können wir ähnlich vorgehen.
- Wir haben eine zusätzliche Einschränkung: Alle Entitäten vom Typ N müssen jeweils mit einer Entität vom Typ M verbunden sein.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x        CHAR(3)                -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                -- example of other attributes
18     UNIQUE (nid, fknid)  -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Aber machbar.
- Ich hatte mit G  H-Szenario zuerst Probleme, aber am Ende haben wir es hinbekommen.
- Jetzt können wir ähnlich vorgehen.
- Wir haben eine zusätzliche Einschränkung: Alle Entitäten vom Typ N müssen jeweils mit einer Entität vom Typ M verbunden sein.
- Diese zusätzliche Einschränkung ist die selbe, die wir damals verwendet haben ... nur „andersherum“.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Ich hatte mit G  H-Szenario zuerst Probleme, aber am Ende haben wir es hinbekommen.
- Jetzt können wir ähnlich vorgehen.
- Wir haben eine zusätzliche Einschränkung: Alle Entitäten vom Typ N müssen jeweils mit einer Entität vom Typ M verbunden sein.
- Diese zusätzliche Einschränkung ist die selbe, die wir damals verwendet haben ... nur „andersherum“.
- Wenn wir das G  H-Szenario verstehen, dann verstehen wir auch die Tabellenstruktur hier.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x        CHAR(3)  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\vdash$ $\vdash$ N: Lösung

- Jetzt können wir ähnlich vorgehen.
- Wir haben eine zusätzliche Einschränkung: Alle Entitäten vom Typ N müssen jeweils mit einer Entität vom Typ M verbunden sein.
- Diese zusätzliche Einschränkung ist die selbe, die wir damals verwendet haben ... nur „andersherum“.
- Wenn wir das $G \oplus \bigcirc \dashv \vdash H$ -Szenario verstehen, dann verstehen wir auch die Tabellenstruktur hier.
- Naja, fast.

```

1  /* Create the tables for a M-||---|<N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fknid)                -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);

```

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

M N: Lösung

- Wir haben eine zusätzliche Einschränkung: Alle Entitäten vom Typ N müssen jeweils mit einer Entität vom Typ M verbunden sein.
- Diese zusätzliche Einschränkung ist die selbe, die wir damals verwendet haben ... nur „andersherum“.
- Wenn wir das G  H-Szenario verstehen, dann verstehen wir auch die Tabellenstruktur hier.
- Naja, fast.
- Wir erstellen eine Tabelle `m` für die Entitäten vom Typ M.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 2  CREATE SEQUENCE
31 3  CREATE TABLE
32 4  CREATE TABLE
33 5  ALTER TABLE
34 6  # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Diese zusätzliche Einschränkung ist die selbe, die wir damals verwendet haben ... nur „andersherum“.
- Wenn wir das G  H-Szenario verstehen, dann verstehen wir auch die Tabellenstruktur hier.
- Naja, fast.
- Wir erstellen eine Tabelle `m` für die Entitäten vom Typ M.
- Sie hat den Primärschlüssel `mid` und eine zusätzliche Datenspalte `x`.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Wenn wir das G  H-Szenario verstehen, dann verstehen wir auch die Tabellenstruktur hier.
- Naja, fast.
- Wir erstellen eine Tabelle `m` für die Entitäten vom Typ M.
- Sie hat den Primärschlüssel `mid` und eine zusätzliche Datenspalte `x`.
- Wir brauchen auch das Attribut `fknid`, welches wir später zum Referenzieren einer Zeile in Tabelle `n` verwenden.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 2  CREATE SEQUENCE
31 3  CREATE TABLE
32 4  CREATE TABLE
33 5  ALTER TABLE
34 6  # psql 16.11 succeeded with exit code 0.
```

```

1  /* Create the tables for a M-||---||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                -- example of other attributes
18     UNIQUE (nid, fkmid)              -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);

```

```

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

```

M |||<N: Lösung

- Wir erstellen eine Tabelle `m` für die Entitäten vom Typ M.
- Sie hat den Primärschlüssel `mid` und eine zusätzliche Datenspalte `x`.
- Wir brauchen auch das Attribut `fknid`, welches wir später zum Referenzieren einer Zeile in Tabelle `n` verwenden.
- Weil jede Entität vom Typ M mit mindestens einer Entität vom Typ N verbunden sein muss, ist diese Spalte `NOT NULL`.
- Weil jede Entität vom Typ N mit genau einer Entität vom Typ M (und nicht mehr als einer) verbunden sein muss, machen wir die Spalte auch `UNIQUE`.

```
1  /* Create the tables for a M-|||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||-< N: Lösung

- Sie hat den Primärschlüssel `mid` und eine zusätzliche Datenspalte `x`.
- Wir brauchen auch das Attribut `fknid`, welches wir später zum Referenzieren einer Zeile in Tabelle `n` verwenden.
- Weil jede Entität vom Typ M mit mindestens einer Entität vom Typ N verbunden sein muss, ist diese Spalte `NOT NULL`.
- Weil jede Entität vom Typ N mit genau einer Entität vom Typ M (und nicht mehr als einer) verbunden sein muss, machen wir die Spalte auch `UNIQUE`.
- Jetzt machen wir die Tabelle `n` zum Speichern der Entitäten vom Typ N.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x       CHAR(3)  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fkfid, mid)
24 REFERENCES n (nid, fkfid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M ||-|<N: Lösung

- Weil jede Entität vom Typ M mit mindestens einer Entität vom Typ N verbunden sein muss, ist diese Spalte **NOT NULL**.
- Weil jede Entität vom Typ N mit genau einer Entität vom Typ M (und nicht mehr als einer) verbunden sein muss, machen wir die Spalte auch **UNIQUE**.
- Jetzt machen wir die Tabelle **n** zum Speichern der Entitäten vom Typ N.
- Der Primärschlüssel ist **nid** und die zusätzliche Datenspalte ist **y**.
- Weil jede Zeile in Tabelle **n** mit genau einer Zeile in Tabelle **m** verbunden sein muss, fügen wir auch eine Spalte **fkmid** an diese Tabelle an.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)              -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||-|<N: Lösung

- Weil jede Entität vom Typ M mit mindestens einer Entität vom Typ N verbunden sein muss, ist diese Spalte **NOT NULL**.
- Weil jede Entität vom Typ N mit genau einer Entität vom Typ M (und nicht mehr als einer) verbunden sein muss, machen wir die Spalte auch **UNIQUE**.
- Jetzt machen wir die Tabelle **n** zum Speichern der Entitäten vom Typ N.
- Der Primärschlüssel ist **nid** und die zusätzliche Datenspalte ist **y**.
- Weil jede Zeile in Tabelle **n** mit genau einer Zeile in Tabelle **m** verbunden sein muss, fügen wir auch eine Spalte **fkmid** an diese Tabelle an.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)              -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M $\vdash$ $\vdash$ N: Lösung

- Weil jede Entität vom Typ N mit genau einer Entität vom Typ M (und nicht mehr als einer) verbunden sein muss, machen wir die Spalte auch **UNIQUE**.
- Jetzt machen wir die Tabelle **n** zum Speichern der Entitäten vom Typ N.
- Der Primärschlüssel ist **nid** und die zusätzliche Datenspalte ist **y**.
- Weil jede Zeile in Tabelle **n** mit genau einer Zeile in Tabelle **m** verbunden sein muss, fügen wir auch eine Spalte **fkmid** an diese Tabelle an.
- Diese Spalte **REFERENCES** den Primärschlüssel **mid** der Tabelle **m**.

```

1  /* Create the tables for a M-|||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x        CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                -- example of other attributes
18     UNIQUE (nid, fkmid)              -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

M $\vdash$ $\vdash$ N: Lösung

- Jetzt machen wir die Tabelle `n` zum Speichern der Entitäten vom Typ N.
- Der Primärschlüssel ist `nid` und die zusätzliche Datenspalte ist `y`.
- Weil jede Zeile in Tabelle `n` mit genau einer Zeile in Tabelle `m` verbunden sein muss, fügen wir auch eine Spalte `fkmid` an diese Tabelle an.
- Diese Spalte `REFERENCES` den Primärschlüssel `mid` der Tabelle `m`.
- Der Unterschied zum G $\rightarrow$ H-Szenario ist, dass `fkmid` der Tabelle `n` nun `NOT NULL` sein muss.

```

1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)                -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fkmid);

```

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

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leftarrow \text{N}$: Lösung

- Der Primärschlüssel ist `nid` und die zusätzliche Datenspalte ist `y`.
- Weil jede Zeile in Tabelle `n` mit genau einer Zeile in Tabelle `m` verbunden sein muss, fügen wir auch eine Spalte `fkmid` an diese Tabelle an.
- Diese Spalte `REFERENCES` den Primärschlüssel `mid` der Tabelle `m`.
- Der Unterschied zum G $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leftarrow \text{H}$ -Szenario ist, dass `fkmid` der Tabelle `n` nun `NOT NULL` sein muss.
- Damals konnte der Fremdschlüssel `fkgid` der Tabelle `h` noch `NULL` sein.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkmid  INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 2  CREATE SEQUENCE
30 3  CREATE TABLE
31 4  CREATE TABLE
32 5  ALTER TABLE
33 6  # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Diese Spalte **REFERENCES** den Primärschlüssel **mid** der Tabelle **m**.
- Der Unterschied zum G  H-Szenario ist, dass **fkmid** der Tabelle **n** nun **NOT NULL** sein muss.
- Damals konnte der Fremdschlüssel **fkgid** der Tabelle **h** noch **NULL** sein.
- Diese Änderung ist notwendig, weil jede Entität vom Typ N mit einer Entität vom Typ M verbunden sein *muss*, wohingegen damals jede Entität vom Typ H mit keiner oder einer Entität vom Typ G verbunden sein *konnte*.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)                -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 --eof MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M N: Lösung

- Der Unterschied zum G  H-Szenario ist, dass `fkmid` der Tabelle `n` nun **NOT NULL** sein muss.
- Damals konnte der Fremdschlüssel `fkgid` der Tabelle `h` noch **NULL** sein.
- Diese Änderung ist notwendig, weil jede Entität vom Typ N mit einer Entität vom Typ M verbunden sein *muss*, wohingegen damals jede Entität vom Typ H mit keiner oder einer Entität vom Typ G verbunden sein *konnte*.
- Davon abgesehen können wir nun die Einschränkung `m_fkfid_mid_fk` zur Tabelle `m` hinzufügen, die genauso funktioniert wie damals `g_fkfid_gid_fk`.

```
1  /* Create the tables for a M-||-----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkfid    INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)              -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkfid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fkfid) -- needed for m_mid_fkfid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkfid_mid_fk FOREIGN KEY (fkfid, mid)
24     REFERENCES n (nid, fkfid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M N: Befüllen mit Daten

- OK, füllen wir Daten in die Tabellen.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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

- OK, füllen wir Daten in die Tabellen.
- Es stellt sich heraus, dass dieses einzelne neue `NOT NULL`, das wir auf Spalte `fkmid` von Tabelle `n` definieren, das Einfügen der Daten viel schwieriger macht.

```

1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x        CHAR(3)                   -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)                -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

- G $\vdash$ O $\dashv$ H-Szenario schon schwierig ...

```

1  /* Create the tables for a M-|---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)                -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leq N$: Befüllen mit Daten

- OK, füllen wir Daten in die Tabellen.
- Es stellt sich heraus, dass dieses einzelne neue **NOT NULL**, das wir auf Spalte **fkmid** von Tabelle **n** definieren, das Einfügen der Daten viel schwieriger macht.
- Gut, es war auch im G $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leq H$ -Szenario schon schwierig ... aber jetzt ist es noch ärgerlicher geworden.
- Das „G-Ende“ der Beziehung war damals optional.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

- ```

1 /* Create the tables for a M-|- relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fkmid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fkmid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24 REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

- Dann konnten wir die Datensätze für die Tabelle mit den Entitäten vom Typ G erstellen und die referentielle Integrität in Ordnung bringen.

```

1 /* Create the tables for a M-||---||<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fkmid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fkmid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24 REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

- ```

1  /* Create the tables for a M-||---||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)               -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

- Dann konnten wir die Datensätze für die Tabelle mit den Entitäten vom Typ G erstellen und die referentielle Integrität in Ordnung bringen.
- Egal wie wir uns das jetzt angucken: Wir haben kein Glück.
- Jede Entität vom Typ N *muss* mit genau einer existierenden Entität vom Typ M verbunden sein.

```

1  /* Create the tables for a M-|||<---||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fknid)                -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

- ```

1 /* Create the tables for a M-|->N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fkmid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fkmid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24 REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.

```



# M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ $\leftarrow$ N: Befüllen mit Daten

- Jede Entität vom Typ N *muss* mit genau einer existierenden Entität vom Typ M verbunden sein.
- Jede Entität vom Typ M *muss* mit wenigstens einer existierenden Entität vom Typ N verbunden sein.
- Das ist ein viel schlimmeres Problem, denn wir müssen den Primärschlüssel `nid` einer Zeile der Tabelle `n` kennen, um eine neue Zeile in Tabelle `m` anzulegen.
- Und wir müssen den Primärschlüssel `mid` einer Zeile in Tabelle `m` kennen, um eine neue Zeile in Tabelle `n` zu speichern.

```
1 /* Create the tables for a M-||----|<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fknid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fkmid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

```

1 /* Create the tables for a M-1|-...|<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fknid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fknid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fknid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

# M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ $\leftarrow$ N: Befüllen mit Daten

- Jede Entität vom Typ M *muss* mit wenigstens einer existierenden Entität vom Typ N verbunden sein.
- Das ist ein viel schlimmeres Problem, denn wir müssen den Primärschlüssel `nid` einer Zeile der Tabelle `n` kennen, um eine neue Zeile in Tabelle `m` anzulegen.
- Und wir müssen den Primärschlüssel `mid` einer Zeile in Tabelle `m` kennen, um eine neue Zeile in Tabelle `n` zu speichern.
- Es ist klar, dass wir wieder CTEs brauchen werden.
- Die einzige Lösung, die ich mir überlegen konnte, ist folgende Idee:

```
1 /* Create the tables for a M-||----|<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fknid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fknid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

# M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array}$ N: Befüllen mit Daten

- Und wir müssen den Primärschlüssel `mid` einer Zeile in Tabelle `m` kennen, um eine neue Zeile in Tabelle `n` zu speichern.
- Es ist klar, dass wir wieder CTEs brauchen werden.
- Die einzige Lösung, die ich mir überlegen konnte, ist folgende Idee:
- Wir müssen, irgendwie, den Wert des Primärschlüssels, der für die nächste Zeile in Tabelle `m` verwendet werden wird herausfinden, *bevor* die Zeile erzeugt wird.

```
1 /* Create the tables for a M-||----|<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fknid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fknid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

# M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array}$ N: Befüllen mit Daten

- Es ist klar, dass wir wieder CTEs brauchen werden.
- Die einzige Lösung, die ich mir überlegen konnte, ist folgende Idee:
- Wir müssen, irgendwie, den Wert des Primärschlüssels, der für die nächste Zeile in Tabelle `m` verwendet werden wird herausfinden, *bevor* die Zeile erzeugt wird.
- Wenn wir den Primärschlüsselwert kennen, dann können wir ihn als `fkmid` verwenden, wenn wir eine Zeile in Tabelle `n` einfügen.

```
1 /* Create the tables for a M-||----|<-N relationship. */
2
3 -- The sequence for the primary keys of the rows in m.
4 CREATE SEQUENCE sqmid AS INT;
5
6 -- Table M: Each row in M is related to one or multiple rows in N.
7 CREATE TABLE m (
8 mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9 fknid INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10 x CHAR(3) -- example of other attributes
11);
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15 nid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16 fkmid INT NOT NULL REFERENCES m (mid),
17 y CHAR(2), -- example of other attributes
18 UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19);
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24 REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

- ```

1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x        CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

-

```

1  /* Create the tables for a M-1|-...|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x        CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leq N$: Befüllen mit Daten

- Wenn wir den Primärschlüsselwert kennen, dann können wir ihn als `fkmid` verwenden, wenn wir eine Zeile in Tabelle `n` einfügen.
- So bekommen wir den Primärschlüsselwert dieser neuen Zeile in Tabelle `n`.
- Dann verwenden wir diesen Primärschlüsselwert als `fkmid` und fügen die Zeile in Tabelle `m` ein.
- Das kann man machen, wenn wir den Primärschlüssel für Tabelle `m` expliziter erstellen als früher.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ N: Befüllen mit Daten

- Wenn wir den Primärschlüsselwert kennen, dann können wir ihn als `fkmid` verwenden, wenn wir eine Zeile in Tabelle `n` einfügen.
- So bekommen wir den Primärschlüsselwert dieser neuen Zeile in Tabelle `n`.
- Dann verwenden wir diesen Primärschlüsselwert als `fkmid` und fügen die Zeile in Tabelle `m` ein.
- Das kann man machen, wenn wir den Primärschlüssel für Tabelle `m` expliziter erstellen als früher.
- Bisher haben wir immer sowas wie
`mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY.`

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkmid  INT NOT NULL UNIQUE,                      -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)                                   -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),                                  -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

```

1  /* Create the tables for a M-||---||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)               -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.

```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ N: Befüllen mit Daten

- Dann verwenden wir diesen Primärschlüsselwert als `fknid` und fügen die Zeile in Tabelle `m` ein.
- Das kann man machen, wenn wir den Primärschlüssel für Tabelle `m` expliziter erstellen als früher.
- Bisher haben wir immer sowas wie `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- Das bedeutet, dass die Spalte ein Primärschlüssel vom Typ `INT` ist.
- Sie wird `GENERATED BY DEFAULT`, was bedeutet, dass ihre Werte automatisch generiert werden es sei denn, wir geben sie explizit an³³.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||--<N: Befüllen mit Daten

- Bisher haben wir immer sowas wie `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- Das bedeutet, dass die Spalte ein Primärschlüssel vom Typ `INT` ist.
- Sie wird `GENERATED BY DEFAULT`, was bedeutet, dass ihre Werte automatisch generiert werden *es sei denn, wir geben sie explizit an*³³.
- Wenn wir eine Zeile in die Tabelle einfügen, dann könnten wir tatsächlich einen Wert für `mid` angeben, welcher dann verwendet wird, oder wir geben keinen Wert an, dann wird einer automatisch erzeugt.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Das bedeutet, dass die Spalte ein Primärschlüssel vom Typ `INT` ist.
- Sie wird `GENERATED BY DEFAULT`, was bedeutet, dass ihre Werte automatisch generiert werden *es sei denn, wir geben sie explizit an*³³.
- Wenn wir eine Zeile in die Tabelle einfügen, dann könnten wir tatsächlich einen Wert für `mid` angeben, welcher dann verwendet wird, oder wir geben keinen Wert an, dann wird einer automatisch erzeugt.
- Das `AS IDENTITY` bedeutet, dass wenn ein Wert erzeugt wird, dann von einer impliziten Sequenz⁴⁰.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Sie wird **GENERATED BY DEFAULT**, was bedeutet, dass ihre Werte automatisch generiert werden *es sei denn, wir geben sie explizit an*³³.
- Wenn wir eine Zeile in die Tabelle einfügen, dann könnten wir tatsächlich einen Wert für **mid** angeben, welcher dann verwendet wird, oder wir geben keinen Wert an, dann wird einer automatisch erzeugt.
- Das **AS IDENTITY** bedeutet, dass wenn ein Wert erzeugt wird, dann von einer impliziten Sequenz⁴⁰.
- Anstatt eine implizite Sequenz ohne Name zu verwenden, können wir auch eine Sequenz selbst erzeugen.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ $\leftarrow$ N: Befüllen mit Daten

- Wenn wir eine Zeile in die Tabelle einfügen, dann könnten wir tatsächlich einen Wert für `mid` angeben, welcher dann verwendet wird, oder wir geben keinen Wert an, dann wird einer automatisch erzeugt.
- Das `AS IDENTITY` bedeutet, dass wenn ein Wert erzeugt wird, dann von einer impliziten Sequenz⁴⁰.
- Anstatt eine implizite Sequenz ohne Name zu verwenden, können wir auch eine Sequenz selbst erzeugen.
- Das geht mit dem Kommando `CREATE SEQUENCE`²³.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fkmid  INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)                -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),                -- example of other attributes
18     UNIQUE (nid, fkmid)  -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array}$ N: Befüllen mit Daten

- Das `AS IDENTITY` bedeutet, dass wenn ein Wert erzeugt wird, dann von einer impliziten Sequenz⁴⁰.
- Anstatt eine implizite Sequenz ohne Name zu verwenden, können wir auch eine Sequenz selbst erzeugen.
- Das geht mit dem Kommando `CREATE SEQUENCE`²³.
- Wir erstellen also die Sequenz `sqmid` um die Primärschlüsselwerte für Tabelle `m` zu erzeugen mit `CREATE SEQUENCE sqmid AS INT;`.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,                      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                                   -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                                  -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M ||--< N: Befüllen mit Daten

- Das `AS IDENTITY` bedeutet, dass wenn ein Wert erzeugt wird, dann von einer impliziten Sequenz⁴⁰.
- Anstatt eine implizite Sequenz ohne Name zu verwenden, können wir auch eine Sequenz selbst erzeugen.
- Das geht mit dem Kommando `CREATE SEQUENCE`²³.
- Wir erstellen also die Sequenz `sqmid` um die Primärschlüsselwerte für Tabelle `m` zu erzeugen mit `CREATE SEQUENCE sqmid AS INT;`.
- Diese Sequenz produziert streng monoton ansteigende Werte.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ N: Befüllen mit Daten

- Anstatt eine implizite Sequenz ohne Name zu verwenden, können wir auch eine Sequenz selbst erzeugen.
- Das geht mit dem Kommando `CREATE SEQUENCE`²³.
- Wir erstellen also die Sequenz `sqmid` um die Primärschlüsselwerte für Tabelle `m` zu erzeugen mit `CREATE SEQUENCE sqmid AS INT;`.
- Diese Sequenz produziert streng monoton ansteigende Werte.
- Der nächste Wert von `sqmid` kann atomisch mit `NEXTVAL('sqmid')` erzeugt werden⁶².

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Das geht mit dem Kommando `CREATE SEQUENCE`²³.
- Wir erstellen also die Sequenz `sqmid` um die Primärschlüsselwerte für Tabelle `m` zu erzeugen mit `CREATE SEQUENCE sqmid AS INT;`.
- Diese Sequenz produziert streng monoton ansteigende Werte.
- Der nächste Wert von `sqmid` kann atomisch mit `NEXTVAL('sqmid')` erzeugt werden⁶².
- Wir definieren nun den Primärschlüssel der Tabelle `m` als `mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY`.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leq N$: Befüllen mit Daten

- Diese Sequenz produziert streng monoton ansteigende Werte.
- Der nächste Wert von `sqmid` kann atomisch mit `NEXTVAL('sqmid')` erzeugt werden⁶².
- Wir definieren nun den Primärschlüssel der Tabelle `m` als `mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY`.
- Diese Definition besagt, dass der Wert des Primärschlüssels `mid` beim Erstellen einer Zeile in Tabelle `m` angegeben werden kann.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

- Diese Sequenz produziert streng monoton ansteigende Werte.
- Der nächste Wert von `sqmid` kann atomisch mit `NEXTVAL('sqmid')` erzeugt werden⁶².
- Wir definieren nun den Primärschlüssel der Tabelle `m` als `mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY`.
- Diese Definition besagt, dass der Wert des Primärschlüssels `mid` beim Erstellen einer Zeile in Tabelle `m` angegeben werden *kann*.
- Wenn er *nicht* angegeben wird, dann wird ein `DEFAULT`-Wert benutzt²⁸.

```

1  /* Create the tables for a M-||---||<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE,      -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)                  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2),                  -- example of other attributes
18     UNIQUE (nid, fkmid)                -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

M ||---|<-N: Befüllen mit Daten

- Der nächste Wert von `sqmid` kann atomisch mit `NEXTVAL('sqmid')` erzeugt werden⁶².
- Wir definieren nun den Primärschlüssel der Tabelle `m` als `mid INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY`.
- Diese Definition besagt, dass der Wert des Primärschlüssels `mid` beim Erstellen einer Zeile in Tabelle `m` angegeben werden *kann*.
- Wenn er *nicht* angegeben wird, dann wird ein `DEFAULT`-Wert benutzt²⁸.
- Der Default-Wert wird berechnet, in dem der Ausdruck `NEXTVAL('sqmid')` ausgerechnet wird.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid    INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x         CHAR(3)             -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
    ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Wir definieren nun den Primärschlüssel der Tabelle `m` als `mid` `INT` `DEFAULT NEXTVAL('sqmid')` `PRIMARY KEY`.
- Diese Definition besagt, dass der Wert des Primärschlüssels `mid` beim Erstellen einer Zeile in Tabelle `m` angegeben werden *kann*.
- Wenn er *nicht* angegeben wird, dann wird ein `DEFAULT`-Wert benutzt²⁸.
- Der Default-Wert wird berechnet, in dem der Ausdruck `NEXTVAL('sqmid')` ausgerechnet wird.
- Er wird eine der immer-ansteigenden Ausgaben der Sequenz `sqmid` sein.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fkmid);
25
26 $ psql "postgres://postgres:XXX@localhost/relationships" -v
27     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
28
29 CREATE SEQUENCE
30 CREATE TABLE
31 CREATE TABLE
32 ALTER TABLE
33 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Diese Definition besagt, dass der Wert des Primärschlüssels `mid` beim Erstellen einer Zeile in Tabelle `m` angegeben werden *kann*.
- Wenn er *nicht* angegeben wird, dann wird ein `DEFAULT`-Wert benutzt²⁸.
- Der Default-Wert wird berechnet, in dem der Ausdruck `NEXTVAL('sqmid')` ausgerechnet wird.
- Er wird eine der immer-ansteigenden Ausgaben der Sequenz `sqmid` sein.
- Aus mechanischer Sicht funktioniert das im Grunde genauso wie `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ $\leq$ N: Befüllen mit Daten

- Wenn er *nicht* angegeben wird, dann wird ein **DEFAULT**-Wert benutzt²⁸.
- Der Default-Wert wird berechnet, in dem der Ausdruck **NEXTVAL('sqmid')** ausgerechnet wird.
- Er wird eine der immer-ansteigenden Ausgaben der Sequenz **sqmid** sein.
- Aus mechanischer Sicht funktioniert das im Grunde genauso wie **mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY**.
- Der Unterschied ist, dass wir jetzt auch selber Werte für **mid** erzeugen können, *ohne* dass dafür eine neue Zeile in Tabelle **m** erstellt werden muss.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ N: Befüllen mit Daten

- Er wird eine der immer-ansteigenden Ausgaben der Sequenz `sqmid` sein.
- Aus mechanischer Sicht funktioniert das im Grunde genauso wie `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- Der Unterschied ist, dass wir jetzt auch selber Werte für `mid` erzeugen können, *ohne* dass dafür eine neue Zeile in Tabelle `m` erstellt werden muss.
- Natürlich können wir dann `NEXTVAL('sqmid')` in jeden SQL-Anfrage verwenden – `SELECT`, `INSERT`, `UPDATE` – was immer wir wollen.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M ||---|<-N: Befüllen mit Daten

- Aus mechanischer Sicht funktioniert das im Grunde genauso wie `mid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY`.
- Der Unterschied ist, dass wir jetzt auch selber Werte für `mid` erzeugen können, *ohne* dass dafür eine neue Zeile in Tabelle `m` erstellt werden muss.
- Natürlich können wir dann `NEXTVAL('sqmid')` in jeden SQL-Anfrage verwenden – `SELECT`, `INSERT`, `UPDATE` – was immer wir wollen.
- Und wann immer `NEXTVAL('sqmid')` aufgerufen wird, macht die Sequenz `sqmid` einen Schritt.

```
1  /* Create the tables for a M-||---|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY,  -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE,  -- UNIQUE: Each N is related to one M.
10     x      CHAR(3)  -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2),  -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28 ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{---} \\ \text{---} \end{array}$ $\leftarrow$ N: Befüllen mit Daten

- Der Unterschied ist, dass wir jetzt auch selber Werte für `mid` erzeugen können, *ohne* dass dafür eine neue Zeile in Tabelle `m` erstellt werden muss.
- Natürlich können wir dann `NEXTVAL('sqmid')` in jeden SQL-Anfrage verwenden – `SELECT`, `INSERT`, `UPDATE` – was immer wir wollen.
- Und wann immer `NEXTVAL('sqmid')` aufgerufen wird, macht die Sequenz `sqmid` einen Schritt.
- Sie wird niemals den selben Wert zweimal liefern (naja, es sei denn, wir rufen das 2^{64} Mal auf...).

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid    INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fknid  INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x      CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fknid  INT NOT NULL REFERENCES m (mid),
17     y      CHAR(2), -- example of other attributes
18     UNIQUE (nid, fknid) -- needed for m_mid_fknid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fknid_mid_fk FOREIGN KEY (fknid, mid)
24     REFERENCES n (nid, fknid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34 # psql 16.11 succeeded with exit code 0.
```

M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array}$ N: Befüllen mit Daten

- Und wann immer `NEXTVAL('sqmid')` aufgerufen wird, macht die Sequenz `sqmid` einen Schritt.
- Sie wird niemals den selben Wert zweimal liefern (naja, es sei denn, wir rufen das 2^{64} Mal auf. ...).
- Wir können also gültige Primärschlüsselwerte für Tabelle `m` erzeugen, mit diesen als Fremdschlüssel eine Zeile in Tabelle `n` generieren und den Wert dann für den Primärschlüssel in einer neuen Zeile in Tabelle `m` verwenden.

```
1  /* Create the tables for a M-||----|<-N relationship. */
2
3  -- The sequence for the primary keys of the rows in m.
4  CREATE SEQUENCE sqmid AS INT;
5
6  -- Table M: Each row in M is related to one or multiple rows in N.
7  CREATE TABLE m (
8      mid      INT DEFAULT NEXTVAL('sqmid') PRIMARY KEY, -- Use ID sequence!
9      fkmid    INT NOT NULL UNIQUE, -- UNIQUE: Each N is related to one M.
10     x         CHAR(3) -- example of other attributes
11 );
12
13 -- Table N: Each row in N is related exactly one row in M.
14 CREATE TABLE n (
15     nid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkmid    INT NOT NULL REFERENCES m (mid),
17     y        CHAR(2), -- example of other attributes
18     UNIQUE (nid, fkmid) -- needed for m_mid_fkmid_fk
19 );
20
21 -- To table M, we add the foreign key reference constraint towards
22 -- table N. This enforces the mandatory-many part of the relationship.
23 ALTER TABLE m ADD CONSTRAINT m_fkmid_mid_fk FOREIGN KEY (fkmid, mid)
24     REFERENCES n (nid, fkmid);
25
26
27 $ psql "postgres://postgres:XXX@localhost/relationships" -v
28     ↪ ON_ERROR_STOP=1 -ebf MN_tables.sql
29
30 CREATE SEQUENCE
31 CREATE TABLE
32 CREATE TABLE
33 ALTER TABLE
34
35 # psql 16.11 succeeded with exit code 0.
```

M ||---|<N: Befüllen mit Daten

- Und wann immer `NEXTVAL('sqmid')` aufgerufen wird, macht die Sequenz `sqmid` einen Schritt.
- Sie wird niemals den selben Wert zweimal liefern (naja, es sei denn, wir rufen das 2^{64} Mal auf. . .).
- Wir können also gültige Primärschlüsselwerte für Tabelle `m` erzeugen, mit diesen als Fremdschlüssel eine Zeile in Tabelle `n` generieren und den Wert dann für den Primärschlüssel in einer neuen Zeile in Tabelle `m` verwenden.
- Wir fügen ein Paar Zeilen in Tabellen `m` und `n` wie folgt ein:

```
1  /* Inserting data into the tables for the M-||----|<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7       new_n AS (INSERT INTO n (y, fkmid)
8                  SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16       new_n AS (INSERT INTO n (y, fkmid)
17                  SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22       new_n AS (INSERT INTO n (y, fkmid)
23                  SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```



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



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

M ||---|<N: Befüllen mit Daten

- Wir können also gülte Primärschlüsselwerte für Tabelle **m** erzeugen, mit diesen als Fremdschlüssel eine Zeile in Tabelle **n** generieren und den Wert dann für den Primärschlüssel in einer neuen Zeile in Tabelle **m** verwenden.
- Wir fügen ein Paar Zeilen in Tabellen **m** und **n** wie folgt ein:
- Zuerst erstellen wir den Primärschlüsselwert für die Zeile, die wir in Tabelle **m** einfügen wollen als CTE der nur `NEXTVAL('sqmid')` aufruft und sich das Ergebnis merkt, also via `m_id AS (SELECT NEXTVAL('sqmid') AS new_mid)`.

```
1  /* Inserting data into the tables for the M-||----|<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7       new_n AS (INSERT INTO n (y, fkmid)
8                 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9       INSERT INTO m (mid, x, fknid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16       new_n AS (INSERT INTO n (y, fkmid)
17                 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18       INSERT INTO m (mid, x, fknid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22       new_n AS (INSERT INTO n (y, fkmid)
23                 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24       INSERT INTO m (mid, x, fknid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

M $\dashv$ $\dashv$ $\dashv$ N: Befüllen mit Daten

- Wir fügen ein Paar Zeilen in Tabellen `m` und `n` wie folgt ein:

- Zuerst erstellen wir den Primärschlüsselwert für die Zeile, die wir in Tabelle `m` einfügen wollen als CTE der nur `NEXTVAL('sqmid')` aufruft und sich das Ergebnis merkt, also via `m_id AS (SELECT NEXTVAL('sqmid') AS new_mid)`.

- Dann fügen wir eine Zeile in Tabelle `n` ein – auch als CTE – in dem wir `new_n AS (INSERT INTO n (y, fkmid) SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)`.

```
1  /* Inserting data into the tables for the M-||-----||<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid,
7               new_n AS (INSERT INTO n (y, fkmid)
8                           SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9               INSERT INTO m (mid, x, fkid) SELECT fkmid, '123', nid FROM new_n;
10
11  -- Create a new row in N referencing an existing row in M.
12  INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14  -- Insert into M and N at the same time.
15  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid,
16               new_n AS (INSERT INTO n (y, fkmid)
17                           SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18               INSERT INTO m (mid, x, fkid) SELECT fkmid, '456', nid FROM new_n;
19
20  -- Insert into M and N at the same time.
21  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid,
22               new_n AS (INSERT INTO n (y, fkmid)
23                           SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24               INSERT INTO m (mid, x, fkid) SELECT fkmid, '789', nid FROM new_n;
25
26  -- Create a new row in N referencing an existing row in M.
27  INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29  -- Combine the rows from M and N.
30  SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

M ||---|<N: Befüllen mit Daten

- Dann fügen wir eine Zeile in Tabelle `n` ein – auch als CTE – in dem wir `new_n AS (INSERT INTO n (y, fkmid) SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)`.
- Dieser CTE gibt uns den Primärschlüsselwert `nid` der neuen Zeile in Tabelle `n` und auch den vorhin erstellten Wert `new_mid` für den Primärschlüssel der neuen Zeile die wir in Tabelle `m` einfügen wollen (als `fkmid`).

```
1  /* Inserting data into the tables for the M-||----|<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7       new_n AS (INSERT INTO n (y, fkmid)
8                 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16       new_n AS (INSERT INTO n (y, fkmid)
17                 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22       new_n AS (INSERT INTO n (y, fkmid)
23                 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
31
32
33 $ psql "postgres://postgres:XXX@localhost/relationships" -v
34 ↪ ON_ERROR_STOP=1 --ebf MN_insert_and_select.sql
35
36 INSERT 0 1
37 INSERT 0 1
38 INSERT 0 1
39 INSERT 0 1
40 INSERT 0 1
41
42 mid | x  | nid | y
43 ----+----+----+--
44 1 | 123 | 1 | AB
45 1 | 123 | 2 | CD
46 2 | 456 | 3 | EF
47 3 | 789 | 4 | GH
48 1 | 123 | 5 | IJ
49 (5 rows)
50
51 # psql 16.11 succeeded with exit code 0.
```

M ||---|<N: Befüllen mit Daten

- Dieser CTE gibt uns den Primärschlüsselwert `nid` der neuen Zeile in Tabelle `n` und auch den vorhin erstellten Wert `new_mid` für den Primärschlüssel der neuen Zeile die wir in Tabelle `m` einfügen wollen (als `fkmid`).
- Wir benutzen beide Werte um letztendlich die Zeile in Tabelle `m` einzufügen, in dem wir `INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;` aufrufen.

```
1  /* Inserting data into the tables for the M-||----|<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7       new_n AS (INSERT INTO n (y, fkmid)
8                  SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16       new_n AS (INSERT INTO n (y, fkmid)
17                  SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22       new_n AS (INSERT INTO n (y, fkmid)
23                  SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

M --- N: Befüllen mit Daten

- Dieser CTE gibt uns den Primärschlüsselwert `nid` der neuen Zeile in Tabelle `n` und auch den vorhin erstellten Wert `new_mid` für den Primärschlüssel der neuen Zeile die wir in Tabelle `m` einfügen wollen (als `fkmid`).
- Wir benutzen beide Werte um letztendlich die Zeile in Tabelle `m` einzufügen, in dem wir `INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;` aufrufen.
- Danach können wir eine weitere Zeile an Tabelle `n` anhängen und diese mit einer existierenden Zeile in Tabelle `m` verbinden.

```
1  /* Inserting data into the tables for the M-||-----||<-n relationship. */
2
3  -- Insert into M and N at the same time.
4  -- This creates M entry with id 1, and N entry with id 1, and
5  -- relationship (1, 1).
6  WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7       new_n AS (INSERT INTO n (y, fkmid)
8                  SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16       new_n AS (INSERT INTO n (y, fkmid)
17                  SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22       new_n AS (INSERT INTO n (y, fkmid)
23                  SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24       INSERT INTO m (mid, x, fkmid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

M $\dashv$ $\dashv$ $\dashv$ N: Befüllen mit Daten

- Wir benutzen beide Werte um letztendlich die Zeile in Tabelle `m` einzufügen, in dem wir

```
INSERT INTO m (mid, x, fkfid)
SELECT fkfid, '123', nid
FROM new_n; aufrufen.
```

- Danach können wir eine weitere Zeile an Tabelle `n` anhängen und diese mit einer existierenden Zeile in Tabelle `m` verbinden.

- Das ist dann einfach, über
- ```
INSERT INTO n (y, fkfid)VALUES
('CD', 1);.
```

```
1 /* Inserting data into the tables for the M-||-----||<-n relationship. */
2
3 -- Insert into M and N at the same time.
4 -- This creates M entry with id 1, and N entry with id 1, and
5 -- relationship (1, 1).
6 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7 new_n AS (INSERT INTO n (y, fkfid)
8 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkfid)
9 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkfid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16 new_n AS (INSERT INTO n (y, fkfid)
17 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkfid)
18 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22 new_n AS (INSERT INTO n (y, fkfid)
23 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkfid)
24 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkfid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkfid = m.mid;
```

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

# M $\dashv$ N: Befüllen mit Daten

- Wir benutzen beide Werte um letztendlich die Zeile in Tabelle `m` einzufügen, in dem wir  
`INSERT INTO m (mid, x, fkfid)`  
`SELECT fkfid, '123', nid`  
`FROM new_n;` aufrufen.
- Danach können wir eine weitere Zeile an Tabelle `n` anhängen und diese mit einer existierenden Zeile in Tabelle `m` verbinden.
- Das ist dann einfach, über  
`INSERT INTO n (y, fkfid) VALUES ('CD', 1);`
- Wenn wir neue Zeilen in Tabelle `m` einfügen wollen, brauchen wir aber immer zwei CTEs.

```
1 /* Inserting data into the tables for the M-||-----||<-n relationship. */
2
3 -- Insert into M and N at the same time.
4 -- This creates M entry with id 1, and N entry with id 1, and
5 -- relationship (1, 1).
6 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7 new_n AS (INSERT INTO n (y, fkfid)
8 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkfid)
9 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkfid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16 new_n AS (INSERT INTO n (y, fkfid)
17 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkfid)
18 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22 new_n AS (INSERT INTO n (y, fkfid)
23 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkfid)
24 INSERT INTO m (mid, x, fkfid) SELECT fkfid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkfid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkfid = m.mid;
```

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

# M $\dashv$ $\dashv$ $\dashv$ N: Befüllen mit Daten

- Danach können wir eine weitere Zeile an Tabelle **n** anhängen und diese mit einer existierenden Zeile in Tabelle **m** verbinden.

- Das ist dann einfach, über

```
INSERT INTO n (y, fkmid)VALUES
('CD', 1);
```

- Wenn wir neue Zeilen in Tabelle **m** einfügen wollen, brauchen wir aber immer zwei CTEs.

- Dieser Ansatz beruht auf mehreren PostgreSQL-spezifischen Kommandos, **RETURNING**, **NEXTVAL** und das Erstellen von Sequenzen.

```
1 /* Inserting data into the tables for the M-||-----|<-n relationship. */
2
3 -- Insert into M and N at the same time.
4 -- This creates M entry with id 1, and N entry with id 1, and
5 -- relationship (1, 1).
6 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7 new_n AS (INSERT INTO n (y, fkmid)
8 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9 INSERT INTO m (mid, x, fkid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16 new_n AS (INSERT INTO n (y, fkmid)
17 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18 INSERT INTO m (mid, x, fkid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22 new_n AS (INSERT INTO n (y, fkmid)
23 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24 INSERT INTO m (mid, x, fkid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
31
32
33 $ psql "postgres://postgres:XXX@localhost/relationships" -v
34 ↪ ON_ERROR_STOP=1 --ebf MN_insert_and_select.sql
35
36 INSERT 0 1
37 INSERT 0 1
38 INSERT 0 1
39 INSERT 0 1
40 INSERT 0 1
41
42 mid | x | nid | y
43 ----+----+----+---
44 1 | 123 | 1 | AB
45 1 | 123 | 2 | CD
46 2 | 456 | 3 | EF
47 3 | 789 | 4 | GH
48 1 | 123 | 5 | IJ
49 (5 rows)
50
51 # psql 16.11 succeeded with exit code 0.
```

# M $\text{---}$ N: Befüllen mit Daten

- Das ist dann einfach, über

```
INSERT INTO n (y, fkmid) VALUES ('CD', 1);
```

- Wenn wir neue Zeilen in Tabelle `m` einfügen wollen, brauchen wir aber immer zwei CTEs.
- Dieser Ansatz beruht auf mehreren PostgreSQL-spezifischen Kommandos, `RETURNING`, `NEXTVAL` und das Erstellen von Sequenzen.
- Auf anderen DBMSen geht es dann wohl etwas anders, aber bestimmt ähnlich.

```
1 /* Inserting data into the tables for the M-->N relationship. */
2
3 -- Insert into M and N at the same time.
4 -- This creates M entry with id 1, and N entry with id 1, and
5 -- relationship (1, 1).
6 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7 new_n AS (INSERT INTO n (y, fkmid)
8 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9 INSERT INTO m (mid, x, fknid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16 new_n AS (INSERT INTO n (y, fkmid)
17 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18 INSERT INTO m (mid, x, fknid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22 new_n AS (INSERT INTO n (y, fkmid)
23 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24 INSERT INTO m (mid, x, fknid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

# M ||---|<N: Befüllen mit Daten

- Das ist dann einfach, über

```
INSERT INTO n (y, fkmid) VALUES ('CD', 1);
```

- Wenn wir neue Zeilen in Tabelle `m` einfügen wollen, brauchen wir aber immer zwei CTEs.
- Dieser Ansatz beruht auf mehreren PostgreSQL-spezifischen Kommandos, `RETURNING`, `NEXTVAL` und das Erstellen von Sequenzen.
- Auf anderen DBMSen geht es dann wohl etwas anders, aber bestimmt ähnlich.
- Die Daten lassen sich einfach über ein `INNER JOIN` wieder zusammensetzen.

```
1 /* Inserting data into the tables for the M-||----|<-n relationship. */
2
3 -- Insert into M and N at the same time.
4 -- This creates M entry with id 1, and N entry with id 1, and
5 -- relationship (1, 1).
6 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
7 new_n AS (INSERT INTO n (y, fkmid)
8 SELECT 'AB', new_mid FROM m_id RETURNING nid, fkmid)
9 INSERT INTO m (mid, x, fknid) SELECT fkmid, '123', nid FROM new_n;
10
11 -- Create a new row in N referencing an existing row in M.
12 INSERT INTO n (y, fkmid) VALUES ('CD', 1);
13
14 -- Insert into M and N at the same time.
15 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
16 new_n AS (INSERT INTO n (y, fkmid)
17 SELECT 'EF', new_mid FROM m_id RETURNING nid, fkmid)
18 INSERT INTO m (mid, x, fknid) SELECT fkmid, '456', nid FROM new_n;
19
20 -- Insert into M and N at the same time.
21 WITH m_id AS (SELECT NEXTVAL('sqmid') AS new_mid),
22 new_n AS (INSERT INTO n (y, fkmid)
23 SELECT 'GH', new_mid FROM m_id RETURNING nid, fkmid)
24 INSERT INTO m (mid, x, fknid) SELECT fkmid, '789', nid FROM new_n;
25
26 -- Create a new row in N referencing an existing row in M.
27 INSERT INTO n (y, fkmid) VALUES ('IJ', 1);
28
29 -- Combine the rows from M and N.
30 SELECT mid, x, nid, y FROM n INNER JOIN m ON n.fkmid = m.mid;
```

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

# M $\parallel$ $\rightarrow$ $\leftarrow$ N: Der Inhalt der Zwei Tabellen



- Dies sind die Daten in den zwei Tabellen.

| Table <u>m</u> |       |       |
|----------------|-------|-------|
| mid            | fknid | x     |
| 1              | 1     | „123“ |
| 2              | 3     | „456“ |
| 3              | 4     | „789“ |

| Table <u>n</u> |       |      |
|----------------|-------|------|
| nid            | fknid | y    |
| 1              | 1     | „AB“ |
| 2              | 1     | „CD“ |
| 3              | 2     | „EF“ |
| 4              | 3     | „GH“ |
| 5              | 1     | „IJ“ |

# M $\vdash$ $\dashv$ $\vdash$ N: Testen der Einschränkungen

- Probieren wir die Einschränkungen aus.



# M $\begin{array}{c} \text{+} \text{---} \text{+} \\ \text{+} \text{---} \text{+} \end{array}$ N: Testen der Einschränkungen



- Probieren wir die Einschränkungen aus.
- Können wir eine Zeile in Tabelle `m` erstellen, die mit keiner Zeile in Tabelle `n` verbunden ist?

```
1 /* Can we create a row in M unrelated to any row in N? */
2
3 INSERT INTO m (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_1.sql
2 psql:conceptualToRelational/MN_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "m" violates not-null constraint
3 DETAIL: Failing row contains (4, null, 777).
4 psql:conceptualToRelational/MN_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in M unrelated to any row in N? */
5 INSERT INTO m (x) VALUES ('777');
6 # psql 16.11 failed with exit code 3.
```

# M $\begin{array}{c} \text{+} \text{---} \text{+} \\ \text{+} \text{---} \text{+} \end{array}$ N: Testen der Einschränkungen



- Probieren wir die Einschränkungen aus.
- Können wir eine Zeile in Tabelle `m` erstellen, die mit keiner Zeile in Tabelle `n` verbunden ist?
- Nein.

```
1 /* Can we create a row in M unrelated to any row in N? */
2
3 INSERT INTO m (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_1.sql
2 psql:conceptualToRelational/MN_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "m" violates not-null constraint
3 DETAIL: Failing row contains (4, null, 777).
4 psql:conceptualToRelational/MN_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in M unrelated to any row in N? */
5 INSERT INTO m (x) VALUES ('777');
6 # psql 16.11 failed with exit code 3.
```

# M $\nmid$ $\nmid$ $\nmid$ N: Testen der Einschränkungen



- Probieren wir die Einschränkungen aus.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **n** erstellen, die mit keiner Zeile in Tabelle **m** verbunden ist?

```
1 /* Can we create a row in M unrelated to any row in N? */
2
3 INSERT INTO m (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_1.sql
2 psql:conceptualToRelational/MN_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "m" violates not-null constraint
3 DETAIL: Failing row contains (4, null, 777).
4 psql:conceptualToRelational/MN_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in M unrelated to any row in N? */
5 INSERT INTO m (x) VALUES ('777');
6 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we create a row in N unrelated to any row in M? */
2
3 INSERT INTO n (y) VALUES ('ZZ');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_2.sql
2 psql:conceptualToRelational/MN_insert_error_2.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "n" violates not-null constraint
3 DETAIL: Failing row contains (6, null, ZZ).
4 psql:conceptualToRelational/MN_insert_error_2.sql:3: STATEMENT: /* Can
 ↳ we create a row in N unrelated to any row in M? */
5 INSERT INTO n (y) VALUES ('ZZ');
6 # psql 16.11 failed with exit code 3.
```

# M $\nmid$ $\nmid$ $\nmid$ N: Testen der Einschränkungen



- Probieren wir die Einschränkungen aus.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **n** erstellen, die mit keiner Zeile in Tabelle **m** verbunden ist?
- Nein.

```
1 /* Can we create a row in M unrelated to any row in N? */
2
3 INSERT INTO m (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_1.sql
2 psql:conceptualToRelational/MN_insert_error_1.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "m" violates not-null constraint
3 DETAIL: Failing row contains (4, null, 777).
4 psql:conceptualToRelational/MN_insert_error_1.sql:3: STATEMENT: /* Can
 ↳ we create a row in M unrelated to any row in N? */
5 INSERT INTO m (x) VALUES ('777');
6 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we create a row in N unrelated to any row in M? */
2
3 INSERT INTO n (y) VALUES ('ZZ');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_2.sql
2 psql:conceptualToRelational/MN_insert_error_2.sql:3: ERROR: null value
 ↳ in column "fkaid" of relation "n" violates not-null constraint
3 DETAIL: Failing row contains (6, null, ZZ).
4 psql:conceptualToRelational/MN_insert_error_2.sql:3: STATEMENT: /* Can
 ↳ we create a row in N unrelated to any row in M? */
5 INSERT INTO n (y) VALUES ('ZZ');
6 # psql 16.11 failed with exit code 3.
```

```

/* Can we insert a M that is related to a N related to another M? */

-- N with id 4 is already related to M with id 3.
-- Can we make our new M row point to it as its "primary N" anyway?
INSERT INTO m (fknid, x) VALUES (4, '888');
```

- ```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_3.sql
2 psql:conceptualToRelational/MN_insert_error_3.sql:5: ERROR:  duplicate
   ↳ key value violates unique constraint "m_fknid_key"
3 DETAIL:  Key (fknid)=(4) already exists.
4 psql:conceptualToRelational/MN_insert_error_3.sql:5: STATEMENT:  /* Can
   ↳ we insert a M that is related to a N related to another M? */
   -- N with id 4 is already related to M with id 3.
5 -- Can we make our new M row point to it as its "primary N" anyway?
6 INSERT INTO m (fknid, x) VALUES (4, '888');
7 # psql 16.11 failed with exit code 3.

```

$M \dashv\dashv N$: Testen der Einschränkung

- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **n** erstellen, die mit keiner Zeile in Tabelle **m** verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „erstes M“ verbunden ist?
- Nein.

```

1  /* Can we insert a M that is related to a N related to another M? */
2  -- N with id 4 is already related to M with id 3.
3  -- Can we make our new M row point to it as its "primary N" anyway?
4  INSERT INTO m (fknid, x) VALUES (4, '888');
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

```

$M \Vdash N$: Testen der Einschränkung

- Können wir eine Zeile in Tabelle **n** erstellen, die mit keiner Zeile in Tabelle **m** verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „erstes M“ verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „sekundäres M“ verbunden ist?

```

/* Can we insert a M that is related to a N related to another M? */

-- N with id 4 is already related to M with id 3.
-- Can we make our new M row point to it as its "primary N" anyway?
INSERT INTO m (fknid, x) VALUES (4, '888');

```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2   ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_3.sql
3 psql:conceptualToRelational/MN_insert_error_3.sql:5: ERROR:  duplicate
4   ↳ key value violates unique constraint "m_fknid_key"
5 DETAIL:  Key (fknid)=(4) already exists.
6 psql:conceptualToRelational/MN_insert_error_3.sql:5: STATEMENT:  /* Can
7   ↳ we insert a M that is related to a N related to another M? */
8 -- N with id 4 is already related to M with id 3.
9 -- Can we make our new M row point to it as its "primary N" anyway?
10 INSERT INTO m (fknid, x) VALUES (4, '888');
11 # psql 16.11 failed with exit code 3.
```

```

1  /* Can we insert a M that is related to a N related to another N? */
2
3  -- N with id 4 is already related to M with id 3.
4  -- Can we make it point to the new M row instead?
5  WITH m_new AS (INSERT INTO m (fknid, x) VALUES (4, '555'))
6
7      RETURNING mid, fknid)
8
9      UPDATE n SET fknid = m_new.mid FROM m_new WHERE nid = fknid;

```

```

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
2   ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_4.sql
3 psql:conceptualToRelational/MN_insert_error_4.sql:7: ERROR:  duplicate
4   ↳ key value violates unique constraint "m_fkfid_key"
5 DETAIL:  Key (fkfid)=(4) already exists.
6 psql:conceptualToRelational/MN_insert_error_4.sql:7: STATEMENT:  /* Can
7   ↳ we insert a M that is related to a N related to another M? */
8 -- N with id 4 is already related to M with id 3.
9 -- Can we make it point to the new M row instead?
10 WITH m_new AS (INSERT INTO m (fkfid, x) VALUES (4, '555')
11   RETURNING mid, fkfid)
12   UPDATE n SET fkfid = m_new.mid FROM m_new WHERE mid = fkfid;
13 # psql 16.11 failed with exit code 3.

```

M $\dashv$ $\dashv$ $\dashv$ N: Testen der Einschränkungen

- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „erstes M“ verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „sekundäres M“ verbunden ist?
- Nein.

```
1 /* Can we insert a M that is related to a N related to another M? */
2
3 -- N with id 4 is already related to M with id 3.
4 -- Can we make our new M row point to it as its "primary N" anyway?
5 INSERT INTO m (fknid, x) VALUES (4, '888');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_3.sql
2 psql:conceptualToRelational/MN_insert_error_3.sql:5: ERROR:  duplicate
   ↳ key value violates unique constraint "m_fknid_key"
3 DETAIL:  Key (fknid)=(4) already exists.
4 psql:conceptualToRelational/MN_insert_error_3.sql:5: STATEMENT:  /* Can
   ↳ we insert a M that is related to a N related to another M? */
5 -- N with id 4 is already related to M with id 3.
6 -- Can we make our new M row point to it as its "primary N" anyway?
7 INSERT INTO m (fknid, x) VALUES (4, '888');
8 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we insert a M that is related to a N related to another M? */
2
3 -- N with id 4 is already related to M with id 3.
4 -- Can we make it point to the new M row instead?
5 WITH m_new AS (INSERT INTO m (fknid, x) VALUES (4, '555')
6                RETURNING mid, fknid)
7 UPDATE n SET fknid = m_new.mid FROM m_new WHERE nid = fknid;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_4.sql
2 psql:conceptualToRelational/MN_insert_error_4.sql:7: ERROR:  duplicate
   ↳ key value violates unique constraint "m_fknid_key"
3 DETAIL:  Key (fknid)=(4) already exists.
4 psql:conceptualToRelational/MN_insert_error_4.sql:7: STATEMENT:  /* Can
   ↳ we insert a M that is related to a N related to another M? */
5 -- N with id 4 is already related to M with id 3.
6 -- Can we make it point to the new M row instead?
7 WITH m_new AS (INSERT INTO m (fknid, x) VALUES (4, '555')
8                RETURNING mid, fknid)
9 UPDATE n SET fknid = m_new.mid FROM m_new WHERE nid = fknid;
10 # psql 16.11 failed with exit code 3.
```

- ```

1 /* Can we change the relationship of a N away from its primary M? */
2
3 -- N with id 1 is used as "primary N" for M with ID 1.
4 -- Can we make it point to another M?
5 UPDATE n SET fkmid = 3 WHERE nid = 1;
6
7
8
9
10
11 $ psql "postgres://postgres:XXX@localhost/relationships" -v
12 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_5.sql
13 psql:conceptualToRelational/MN_insert_error_5.sql:5: ERROR: update or
14 ↳ delete on table "n" violates foreign key constraint "
15 ↳ m_fkaid_mid_fk" on table "m"
16 DETAIL: Key (nid, fkaid)=(1, 1) is still referenced from table "m".
17 psql:conceptualToRelational/MN_insert_error_5.sql:5: STATEMENT: /* Can
18 ↳ we change the relationship of a N away from its primary M? */
19 -- N with id 1 is used as "primary N" for M with ID 1.
20 -- Can we make it point to another M?
21 UPDATE n SET fkaid = 3 WHERE nid = 1;
22 # psql 16.11 failed with exit code 3.

```

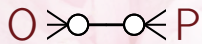
# M $\begin{array}{c} \text{+} \text{---} \text{+} \end{array} \leq \text{N}$ : Testen der Einschränkungen



- Nein.
- Können wir eine Zeile in Tabelle **m** erstellen, die mit keiner Zeile in Tabelle **n** verbunden ist, die bereits mit einer anderen Zeile in Tabelle **m** als „sekundäres M“ verbunden ist?
- Nein.
- Können wir eine Zeile in Tabelle **n** mit einer anderen Zeile in Tabelle **m** verbinden, in dem wir ihr „erstes M“ umhängen?
- Nein.

```
1 /* Can we change the relationship of a N away from its primary M? */
2
3 -- N with id 1 is used as "primary N" for M with ID 1.
4 -- Can we make it point to another M?
5 UPDATE n SET fkmid = 3 WHERE nid = 1;
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf MN_insert_error_5.sql
2 psql:conceptualToRelational/MN_insert_error_5.sql:5: ERROR: update or
 ↳ delete on table "n" violates foreign key constraint "
 ↳ m_fkmid_mid_fk" on table "m"
3 DETAIL: Key (nid, fkmid)=(1, 1) is still referenced from table "m".
4 psql:conceptualToRelational/MN_insert_error_5.sql:5: STATEMENT: /* Can
 ↳ we change the relationship of a N away from its primary M? */
5 -- N with id 1 is used as "primary N" for M with ID 1.
6 -- Can we make it point to another M?
7 UPDATE n SET fkmid = 3 WHERE nid = 1;
8 # psql 16.11 failed with exit code 3.
```



O  $\bowtie$  O — O  $\bowtie$  P

- Es gibt zwei Entitätstypen O und P.



O  $\bowtie$  O  $\text{---}$  O  $\bowtie$  P

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



O  $\bowtie$  O  $\text{---}$  O  $\bowtie$  P

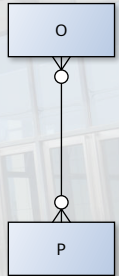
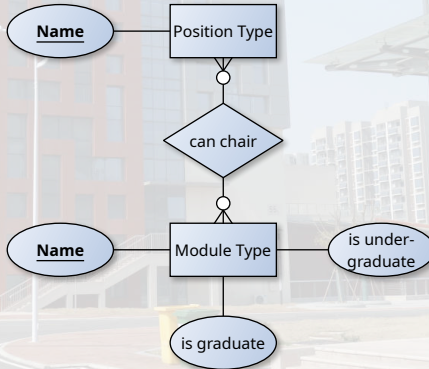


- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.

O  $\bowtie$  O  $\text{---}$  O  $\bowtie$  P



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



# O $\times$ O $\times$ P



- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.
- Beispiel:
  - Wir hatten das, als wir die Beziehung zwischen Modul-Typ und Position diskutiert hatten. Jede Position gibt einem Mitarbeiter die Berechtigung, keinen, einen, oder mehrere Modultypen anzuleiten.

- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.
- Beispiel:
  - Wir hatten das, als wir die Beziehung zwischen Modul-Typ und Position diskutiert hatten. Jede Position gibt einem Mitarbeiter die Berechtigung, keinen, einen, oder mehrere Modultypen anzuleiten. Jeder Modultyp kann von Mitarbeitern keiner, einer, oder mehrerer Positionen geleitet werden.

- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.
- Beispiel:
  - Wir hatten das, als wir die Beziehung zwischen Modul-Typ und Position diskutiert hatten. Jede Position gibt einem Mitarbeiter die Berechtigung, keinen, einen, oder mehrere Modultypen anzuleiten. Jeder Modultyp kann von Mitarbeitern keiner, einer, oder mehrerer Positionen geleitet werden. Pflichtmodule erfordern vielleicht einen Full Professor als Leiter, wohingegen Wahlmodule von Full Professoren, Assistenzprofessoren und Lecturern geleitet werden können.

- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.
- Beispiel:
  - Wir hatten das, als wir die Beziehung zwischen Modul-Typ und Position diskutiert hatten. Jede Position gibt einem Mitarbeiter die Berechtigung, keinen, einen, oder mehrere Modultypen anzuleiten. Jeder Modultyp kann von Mitarbeitern keiner, einer, oder mehrerer Positionen geleitet werden. Pflichtmodule erfordern vielleicht einen Full Professor als Leiter, wohingegen Wahlmodule von Full Professoren, Assistenzprofessoren und Lecturern geleitet werden können. OK, eine Situation wo ein Modultyp von niemandem geleitet werden kann ist sicherlich komisch ... vielleicht sollten wir unser Modell ändern...

- Es gibt zwei Entitätstypen O und P.
- Jede Entität vom Typ O kann mit keiner, einer, oder mehreren Entitäten vom Typ P verbunden sein.
- Jede Entität vom Typ P kann mit keiner, einer, oder mehreren Entitäten vom Typ O verbunden sein.
- Beispiel:
  - Wir hatten das, als wir die Beziehung zwischen Modul-Typ und Position diskutiert hatten. Jede Position gibt einem Mitarbeiter die Berechtigung, keinen, einen, oder mehrere Modultypen anzuleiten. Jeder Modultyp kann von Mitarbeitern keiner, einer, oder mehrerer Positionen geleitet werden. Pflichtmodule erfordern vielleicht einen Full Professor als Leiter, wohingegen Wahlmodule von Full Professoren, Assistenzprofessoren und Lecturern geleitet werden können. OK, eine Situation wo ein Modultyp von niemandem geleitet werden kann ist sicherlich komisch ... vielleicht sollten wir unser Modell ändern...
  - Aber wir hatten ja auch noch mehr Beispiele in Einheit 31...

## O $\times$ O $\times$ P: Lösung

- Wir brauchen wieder eine Tabelle  $\circ$  für die Entitäten vom Typ O.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\times$ o $\times$ P: Lösung

- Wir brauchen wieder eine Tabelle **o** für die Entitäten vom Typ O.
- Der Primärschlüssel wird **oid** und es gibt wieder ein Beispielattribut **x**.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O P: Lösung

- Wir brauchen wieder eine Tabelle  für die Entitäten vom Typ O.
- Der Primärschlüssel wird  und es gibt wieder ein Beispielattribut .
- Wir brauchen auch eine Tabelle  für die Entitäten vom Typ P.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpид INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpид) -- Primary key includes both columns.
20);
```

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

## O $\times$ o $\times$ P: Lösung

- Wir brauchen wieder eine Tabelle **o** für die Entitäten vom Typ O.
- Der Primärschlüssel wird **oid** und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **p** für die Entitäten vom Typ P.
- Diese bekommt den Primärschlüssel **pid** und das Beispielattribut **y**.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\times$ o $\times$ P: Lösung

- Wir brauchen wieder eine Tabelle **o** für die Entitäten vom Typ O.
- Der Primärschlüssel wird **oid** und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **p** für die Entitäten vom Typ P.
- Diese bekommt den Primärschlüssel **pid** und das Beispielattribut **y**.
- Weil jede Zeile in Tabelle **o** mit mehreren Zeilen in Tabelle **p** verbunden sein kann **und** andersherum, können wir das Problem **nicht** mehr mit zwei Tabellen lösen.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\times$ o $\times$ P: Lösung

- Der Primärschlüssel wird `oid` und es gibt wieder ein Beispielattribut `x`.
- Wir brauchen auch eine Tabelle `p` für die Entitäten vom Typ P.
- Diese bekommt den Primärschlüssel `pid` und das Beispielattribut `y`.
- Weil jede Zeile in Tabelle `o` mit mehreren Zeilen in Tabelle `p` verbunden sein kann **und** andersherum, können wir das Problem **nicht** mehr mit zwei Tabellen lösen.
- Es geht nicht anders: Diesmal brauchen wir drei Tabellen.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Wir brauchen auch eine Tabelle **p** für die Entitäten vom Typ P.
- Diese bekommt den Primärschlüssel **pid** und das Beispielattribut **y**.
- Weil jede Zeile in Tabelle **o** mit mehreren Zeilen in Tabelle **p** verbunden sein kann **und** andersherum, können wir das Problem **nicht** mehr mit zwei Tabellen lösen.
- Es geht nicht anders: Diesmal brauchen wir drei Tabellen.
- Der Drei-Tabellen-Ansatz wird sehr ähnlich zu dem in A  $\oplus$  o  $\oplus$  B aussehen, nur mit anderen **UNIQUE**-Einschränkungen.

```
1 /* Create the tables for an O->o<--P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Weil jede Zeile in Tabelle **o** mit mehreren Zeilen in Tabelle **p** verbunden sein kann **und** andersherum, können wir das Problem **nicht** mehr mit zwei Tabellen lösen.
- Es geht nicht anders: Diesmal brauchen wir drei Tabellen.
- Der Drei-Tabellen-Ansatz wird sehr ähnlich zu dem in A  $\oplus$  o  $\oplus$  B aussehen, nur mit anderen **UNIQUE**-Einschränkungen.
- Wir erstellen also auch eine Tabelle **relate\_o\_and\_p** mit zwei Spalten, **fkoid** und **fkpid**.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);

```

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

```

## O $\times$ O $\times$ P: Lösung

- Es geht nicht anders: Diesmal brauchen wir drei Tabellen.
- Der Drei-Tabellen-Ansatz wird sehr ähnlich zu dem in A  $\oplus$  O  $\oplus$  B aussehen, nur mit anderen **UNIQUE**-Einschränkungen.
- Wir erstellen also auch eine Tabelle **relate\_o\_and\_p** mit zwei Spalten, **fkoid** und **fkpid**.
- Diese sind Fremdschlüssel, die jeweils auf die Primärschlüssel **oid** und **pid** der Tabellen **o** und **p** zeigen.

```
1 /* Create the tables for an O->o-----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
21
22 $ psql "postgres://postgres:XXX@localhost/relationships" -v
23 ↪ ON_ERROR_STOP=1 -ebf OP_tables.sql
24 CREATE TABLE
25 CREATE TABLE
26 CREATE TABLE
27 # psql 16.11 succeeded with exit code 0.
```

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Es geht nicht anders: Diesmal brauchen wir drei Tabellen.
- Der Drei-Tabellen-Ansatz wird sehr ähnlich zu dem in A  $\oplus$  o  $\oplus$  B aussehen, nur mit anderen **UNIQUE**-Einschränkungen.
- Wir erstellen also auch eine Tabelle **relate\_o\_and\_p** mit zwei Spalten, **fkoid** und **fkpid**.
- Diese sind Fremdschlüssel, die jeweils auf die Primärschlüssel **oid** und **pid** der Tabellen **o** und **p** zeigen.
- Das sichern wir mit **REFERENCES**-Einschränkungen ab.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
21
22 $ psql "postgres://postgres:XXX@localhost/relationships" -v
23 ↪ ON_ERROR_STOP=1 -ebf OP_tables.sql
24 CREATE TABLE
25 CREATE TABLE
26 CREATE TABLE
27 # psql 16.11 succeeded with exit code 0.
```

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Der Drei-Tabellen-Ansatz wird sehr ähnlich zu dem in A  $\oplus$  o  $\oplus$  B aussehen, nur mit anderen **UNIQUE**-Einschränkungen.
- Wir erstellen also auch eine Tabelle **relate\_o\_and\_p** mit zwei Spalten, **fkoid** und **fkpid**.
- Diese sind Fremdschlüssel, die jeweils auf die Primärschlüssel **oid** und **pid** der Tabellen **o** und **p** zeigen.
- Das sichern wir mit **REFERENCES**-Einschränkungen ab.
- Beide Spalten sind **NOT NULL**, weil wir nur gültige O-P-Paare speichern.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);

```

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

```

## O $\times$ o $\times$ P: Lösung

- Wir erstellen also auch eine Tabelle `relate_o_and_p` mit zwei Spalten, `fkoid` und `fkpid`.
- Diese sind Fremdschlüssel, die jeweils auf die Primärschlüssel `oid` und `pid` der Tabellen `o` und `p` zeigen.
- Das sichern wir mit `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind `NOT NULL`, weil wir nur gültige O-P-Paare speichern.
- Keine der Spalten ist `UNIQUE`, denn jede Zeile in Tabelle `o` kann mit mehreren Zeilen in Tabelle `p` verbunden sein und umgekehrt.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
```

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

## O $\times$ O $\times$ P: Lösung

- Diese sind Fremdschlüssel, die jeweils auf die Primärschlüssel `oid` und `pid` der Tabellen `o` und `p` zeigen.
- Das sichern wir mit `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind `NOT NULL`, weil wir nur gültige O-P-Paare speichern.
- Keine der Spalten ist `UNIQUE`, denn jede Zeile in Tabelle `o` kann mit mehreren Zeilen in Tabelle `p` verbunden sein und umgekehrt.
- In den vorherigen Drei-Tabellen-Lösungen hatten wir immer mindestens eine der Fremdschlüssel-Spalten als `UNIQUE` festgelegt.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
```

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

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Das sichern wir mit **REFERENCES**-Einschränkungen ab.
- Beide Spalten sind **NOT NULL**, weil wir nur gültige O-P-Paare speichern.
- Keine der Spalten ist **UNIQUE**, denn jede Zeile in Tabelle **o** kann mit mehreren Zeilen in Tabelle **p** verbunden sein und umgekehrt.
- In den vorherigen Drei-Tabellen-Lösungen hatten wir immer mindestens eine der Fremdschlüssel-Spalten als **UNIQUE** festgelegt.
- Dadurch war sichergestellt, dass jedes Paar von Fremdschlüsselwerten nur höchstens einmal auftreten kann.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpoid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpoid) -- Primary key includes both columns.
20);
21
22 $ psql "postgres://postgres:XXX@localhost/relationships" -v
23 ↪ ON_ERROR_STOP=1 -ebf OP_tables.sql
24 CREATE TABLE
25 CREATE TABLE
26 CREATE TABLE
27 # psql 16.11 succeeded with exit code 0.
```

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Keine der Spalten ist **UNIQUE**, denn jede Zeile in Tabelle o kann mit mehreren Zeilen in Tabelle p verbunden sein und umgekehrt.
- In den vorherigen Drei-Tabellen-Lösungen hatten wir immer mindestens eine der Fremdschlüssel-Spalten als **UNIQUE** festgelegt.
- Dadurch war sichergestellt, dass jedes Paar von Fremdschlüsselwerten nur höchstens einmal auftreten kann.
- Weil wir dieses Mal keine **UNIQUE**-Spalte mehr haben können, wäre es möglich, dass das selbe Paar von (fkoid, fkpid) mehrfach auftreten können.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);

```

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

```

## O $\times$ o $\times$ P: Lösung

- Keine der Spalten ist **UNIQUE**, denn jede Zeile in Tabelle o kann mit mehreren Zeilen in Tabelle p verbunden sein und umgekehrt.
- In den vorherigen Drei-Tabellen-Lösungen hatten wir immer mindestens eine der Fremdschlüssel-Spalten als **UNIQUE** festgelegt.
- Dadurch war sichergestellt, dass jedes Paar von Fremdschlüsselwerten nur höchstens einmal auftreten kann.
- Weil wir dieses Mal keine **UNIQUE**-Spalte mehr haben können, wäre es möglich, dass das selbe Paar von (fkoid, fkpid) mehrfach auftreten können.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);

```

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

```

## O $\times$ O $\times$ P: Lösung

- Dadurch war sichergestellt, dass jedes Paar von Fremdschlüsselwerten nur höchstens einmal auftreten kann.
- Weil wir dieses Mal keine **UNIQUE**-Spalte mehr haben können, wäre es möglich, dass das selbe Paar von (fkoid, fkpid) mehrfach auftreten können.
- Das können wir natürlich nicht erlauben.
- Wir müssen sicherstellen, dass jedes Paar (fkoid, fkpid) höchstens einmal in der Tabelle auftauchen kann.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
```

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

## O P: Lösung

- Dadurch war sichergestellt, dass jedes Paar von Fremdschlüsselwerten nur höchstens einmal auftreten kann.
- Weil wir dieses Mal keine **UNIQUE**-Spalte mehr haben können, wäre es möglich, dass das selbe Paar von (**fkoid**, **fkpid**) mehrfach auftreten können.
- Das können wir natürlich nicht erlauben.
- Wir müssen sicherstellen, dass jedes Paar (**fkoid**, **fkpid**) höchstens einmal in der Tabelle auftauchen kann.
- Zwei Zeilen in den Tabellen **o** und **p** dürfen, natürlich, höchstens einmal miteinander in Beziehung stehen.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
```

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

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Weil wir dieses Mal keine **UNIQUE**-Spalte mehr haben können, wäre es möglich, dass das selbe Paar von (fkoid, fkpid) mehrfach auftreten können.
- Das können wir natürlich nicht erlauben.
- Wir müssen sicherstellen, dass jedes Paar (fkoid, fkpid) höchstens einmal in der Tabelle auftauchen kann.
- Zwei Zeilen in den Tabellen o und p dürfen, natürlich, höchstens einmal miteinander in Beziehung stehen.
- Wir könnten das mit der Einschränkung **UNIQUE** (fkoid, fkpid) erzwingen.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
21
22 $ psql "postgres://postgres:XXX@localhost/relationships" -v
23 ↪ ON_ERROR_STOP=1 -ebf OP_tables.sql
24 CREATE TABLE
25 CREATE TABLE
26 CREATE TABLE
27 # psql 16.11 succeeded with exit code 0.
```

## O $\times$ O $\times$ P: Lösung

- Das können wir natürlich nicht erlauben.
- Wir müssen sicherstellen, dass jedes Paar (fkoid, fkpid) höchstens einmal in der Tabelle auftauchen kann.
- Zwei Zeilen in den Tabellen o und p dürfen, natürlich, höchstens einmal miteinander in Beziehung stehen.
- Wir könnten das mit der Einschränkung UNIQUE (fkoid, fkpid) erzwingen.
- Die richtige Lösung hier wäre es wahrscheinlich, direkt PRIMARY KEY (fkoid, fkpid) zu benutzen.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);

```

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

```

## O $\times$ o $\times$ P: Lösung

- Wir müssen sicherstellen, dass jedes Paar (fkoid, fkpid) höchstens einmal in der Tabelle auftauchen kann.
- Zwei Zeilen in den Tabellen o und p dürfen, natürlich, höchstens einmal miteinander in Beziehung stehen.
- Wir könnten das mit der Einschränkung UNIQUE (fkoid, fkpid) erzwingen.
- Die richtige Lösung hier wäre es wahrscheinlich, direkt PRIMARY KEY (fkoid, fkpid) zu benutzen.
- Unsere Tabelle braucht einen Primärschlüssel und der einzig mögliche Primärschlüssel sind ja Paare von (fkoid, fkpid).

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
```

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

## O $\rightarrow$ o $\leftarrow$ P: Lösung

- Zwei Zeilen in den Tabellen **o** und **p** dürfen, natürlich, höchstens einmal miteinander in Beziehung stehen.
- Wir könnten das mit der Einschränkung **UNIQUE (fkoid, fkpid)** erzwingen.
- Die richtige Lösung hier wäre es wahrscheinlich, direkt **PRIMARY KEY (fkoid, fkpid)** zu benutzen.
- Unsere Tabelle braucht einen Primärschlüssel und der einzig mögliche Primärschlüssel sind ja Paare von **(fkoid, fkpid)**.
- Primärschlüssel sind sowieso einzigartig, also deckt diese Lösung die Einzigartigkeit gleich mit ab.

```
1 /* Create the tables for an O->o----o<-P relationship. */
2
3 -- Table O: Each row in O is related to zero or one or many rows in P.
4 CREATE TABLE o (
5 oid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6 x CHAR(3) -- example of other attributes
7);
8
9 -- Table P: Each row in P is related to zero or one or many rows in O.
10 CREATE TABLE p (
11 pid 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 O and P.
16 CREATE TABLE relate_o_and_p (
17 fkoid INT NOT NULL REFERENCES o (oid),
18 fkpid INT NOT NULL REFERENCES p (pid),
19 PRIMARY KEY (fkoid, fkpid) -- Primary key includes both columns.
20);
```

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

# O P: Befüllen mit Daten

- Weil beide Beziehungsenden optional sind, können wir Datensätze in die beiden Datentabellen in beliebiger Reihenfolge eingeben.

```
1 /* Inserting data into the tables for the O->o-----o<-P relationship. */
2
3 -- Insert some rows into the table for entity type O.
4 INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6 -- Insert some rows into the table for entity type P.
7 INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9 -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpId) VALUES
11 (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15 INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16 INNER JOIN p ON p.pid = relate_o_and_p.fkpId;
```

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

# O P: Befüllen mit Daten

- Weil beide Beziehungsenden optional sind, können wir Datensätze in die beiden Datentabellen in beliebiger Reihenfolge eingeben.
- Dann erstellen wir die Beziehungen zwischen ihnen.

```
1 /* Inserting data into the tables for the O->o-----o<-P relationship. */
2
3 -- Insert some rows into the table for entity type O.
4 INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6 -- Insert some rows into the table for entity type P.
7 INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9 -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpId) VALUES
11 (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15 INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16 INNER JOIN p ON p.pid = relate_o_and_p.fkpId;
```

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

# O P: Befüllen mit Daten

- Weil beide Beziehungsenden optional sind, können wir Datensätze in die beiden Datentabellen in beliebiger Reihenfolge eingeben.
- Dann erstellen wir die Beziehungen zwischen ihnen.
- Wir fügen also erstmal Zeilen in die Tabellen **o** und **p** ein.

```
1 /* Inserting data into the tables for the O->o-----o<-P relationship. */
2
3 -- Insert some rows into the table for entity type O.
4 INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6 -- Insert some rows into the table for entity type P.
7 INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9 -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpId) VALUES
11 (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15 INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16 INNER JOIN p ON p.pid = relate_o_and_p.fkpId;
```

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

# O P: Befüllen mit Daten

- Weil beide Beziehungsenden optional sind, können wir Datensätze in die beiden Datentabellen in beliebiger Reihenfolge eingeben.
- Dann erstellen wir die Beziehungen zwischen ihnen.
- Wir fügen also erstmal Zeilen in die Tabellen `o` und `p` ein.
- Danach erstellen wir die Beziehungen, in dem wir Zeilen in Tabelle `relate_o_and_p` einfügen.

```
1 /* Inserting data into the tables for the O->o-----o<-P relationship. */
2
3 -- Insert some rows into the table for entity type O.
4 INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6 -- Insert some rows into the table for entity type P.
7 INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9 -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpId) VALUES
11 (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15 INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16 INNER JOIN p ON p.pid = relate_o_and_p.fkpId;
```

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

# O P: Befüllen mit Daten

- Weil beide Beziehungsenden optional sind, können wir Datensätze in die beiden Datentabellen in beliebiger Reihenfolge eingeben.
- Dann erstellen wir die Beziehungen zwischen ihnen.
- Wir fügen also erstmal Zeilen in die Tabellen `o` und `p` ein.
- Danach erstellen wir die Beziehungen, in dem wir Zeilen in Tabelle `relate_o_and_p` einfügen.
- Um die Daten wieder zusammenzuführen brauchen wir zwei `INNER JOINs`.

```
1 /* Inserting data into the tables for the O->o-----o<-P relationship. */
2
3 -- Insert some rows into the table for entity type O.
4 INSERT INTO o (x) VALUES ('123'), ('456'), ('789'), ('101');
5
6 -- Insert some rows into the table for entity type P.
7 INSERT INTO p (y) VALUES ('AB'), ('CD'), ('EF'), ('GH');
8
9 -- Create some relationships between O and P.
10 INSERT INTO relate_o_and_p (fkoid, fkpId) VALUES
11 (1, 1), (1, 2), (2, 2), (4, 1), (3, 2);
12
13 -- Combine the rows from O and P.
14 SELECT oid, x, pid, y FROM relate_o_and_p
15 INNER JOIN o ON o.oid = relate_o_and_p.fkoid
16 INNER JOIN p ON p.pid = relate_o_and_p.fkpId;
```

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

# O $\bowtie$ O $\bowtie$ P: Der Inhalt der Drei Tabellen



- Dies sind die Daten in den drei Tabellen.

| Table o |       |
|---------|-------|
| oid     | x     |
| 1       | „123“ |
| 2       | „456“ |
| 3       | „789“ |
| 4       | „101“ |

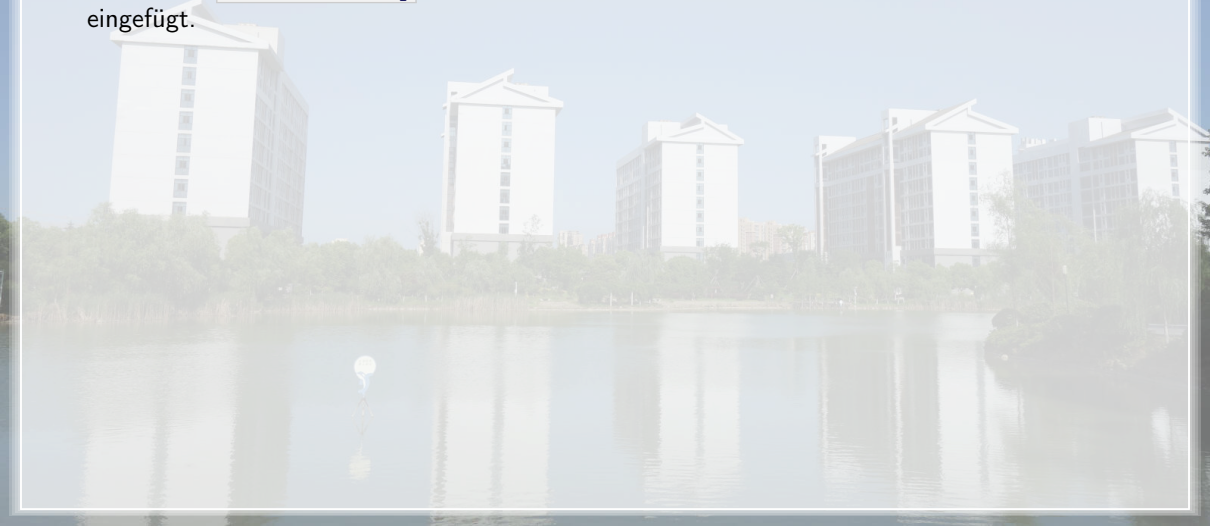
| Table p |      |
|---------|------|
| pid     | y    |
| 1       | „AB“ |
| 2       | „CD“ |
| 3       | „EF“ |
| 4       | „GH“ |

| Table relate_o_and_p |       |
|----------------------|-------|
| fkoid                | fkpid |
| 1                    | 1     |
| 1                    | 2     |
| 2                    | 2     |
| 4                    | 1     |
| 3                    | 2     |

# $O \bowtie O \rightarrow O \bowtie P$ : Testen der Einschränkungen



- Vorhin haben wir die Zeile (1, 1) in die Tabelle `relate_o_and_p` eingefügt.



# O $\bowtie$ O $\text{---}$ O $\bowtie$ P: Testen der Einschränkungen



- Vorhin haben wir die Zeile (1, 1) in die Tabelle `relate_o_and_p` eingefügt.
- Das legt fest, dass die Zeile mit Primärschlüssel `oid = 1` in Tabelle `o` mit der Zeile mit Primärschlüssel `pid = 1` in Tabelle `p` in Beziehung steht.

# O P: Testen der Einschränkungen



- Vorhin haben wir die Zeile (1, 1) in die Tabelle `relate_o_and_p` eingefügt.
- Das legt fest, dass die Zeile mit Primärschlüssel `oid = 1` in Tabelle `o` mit der Zeile mit Primärschlüssel `pid = 1` in Tabelle `p` in Beziehung steht.
- Wir versuchen nun, die selbe Zeile nochmal in `relate_o_and_p` einzufügen.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpид) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpид)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpид) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O — P: Testen der Einschränkungen



- Vorhin haben wir die Zeile (1, 1) in die Tabelle `relate_o_and_p` eingefügt.
- Das legt fest, dass die Zeile mit Primärschlüssel `oid = 1` in Tabelle `o` mit der Zeile mit Primärschlüssel `pid = 1` in Tabelle `p` in Beziehung steht.
- Wir versuchen nun, die selbe Zeile nochmal in `relate_o_and_p` einzufügen.
- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpид) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpид)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpид) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O — P: Testen der Einschränkungen



- Das legt fest, dass die Zeile mit Primärschlüssel `oid = 1` in Tabelle `o` mit der Zeile mit Primärschlüssel `pid = 1` in Tabelle `p` in Beziehung steht.
- Wir versuchen nun, die selbe Zeile nochmal in `relate_o_and_p` einzufügen.
- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.
- Natürlich würde es keinen Sinn ergeben, der selben Person die selbe Adresse zweimal gleichzeitig zuzuweisen.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpid) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpid)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpid) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O P: Testen der Einschränkungen

- Wir versuchen nun, die selbe Zeile nochmal in `relate_o_and_p` einzufügen.
- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.
- Natürlich würde es keinen Sinn ergeben, der selben Person die selbe Adresse zweimal gleichzeitig zuzuweisen.
- Das würde auch verletzen, das Zeilen einmalig sein müssen – eine grundlegende Anforderung des relationalen Datenmodells.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkp_id) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkp_id)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkp_id) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O P: Testen der Einschränkungen

- Wir versuchen nun, die selbe Zeile nochmal in `relate_o_and_p` einzufügen.
- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.
- Natürlich würde es keinen Sinn ergeben, der selben Person die selbe Adresse zweimal gleichzeitig zuzuweisen.
- Das würde auch verletzen, das Zeilen einmalig sein müssen – eine grundlegende Anforderung des relationalen Datenmodells.
- Und dann würden die Zeilen ja auch doppelt als Ergebnis der `INNER JOINs` von vorhin auftauchen.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpoid)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O P: Testen der Einschränkungen

- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.
- Natürlich würde es keinen Sinn ergeben, der selben Person die selbe Adresse zweimal gleichzeitig zuzuweisen.
- Das würde auch verletzen, das Zeilen einmalig sein müssen – eine grundlegende Anforderung des relationalen Datenmodells.
- Und dann würden die Zeilen ja auch doppelt als Ergebnis der `INNER JOINs` von vornhin auftauchen.
- Das sollte also nicht gehen.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpoid)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O P: Testen der Einschränkungen

- Das sollte natürlich nicht gehen, weil zwei Zeilen nicht zweimal miteinander in Beziehung stehen dürfen.
- Natürlich würde es keinen Sinn ergeben, der selben Person die selbe Adresse zweimal gleichzeitig zuzuweisen.
- Das würde auch verletzen, das Zeilen einmalig sein müssen – eine grundlegende Anforderung des relationalen Datenmodells.
- Und dann würden die Zeilen ja auch doppelt als Ergebnis der `INNER JOINs` von vornhin auftauchen.
- Das sollte also nicht gehen.
- Und es geht auch nicht.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpoid)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```

# O P: Testen der Einschränkungen

- Das würde auch verletzen, das Zeilen einmalig sein müssen – eine grundlegende Anforderung des relationalen Datenmodells.
- Und dann würden die Zeilen ja auch doppelt als Ergebnis der **INNER JOINs** von vorhin auftauchen.
- Das sollte also nicht gehen.
- Und es geht auch nicht.
- Dank der **PRIMARY KEY**-Einschränkung, die wir für Tabelle **relate\_o\_and\_p** definiert hatten, schlägt die Einfügung fehl.

```
1 /* Try to create a relationship that already exists. */
2
3 -- The relationship (1, 1) already exists, so this will fail.
4 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);

1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
 ↳ ON_ERROR_STOP=1 -ebf OP_insert_error.sql
2 psql:conceptualToRelational/OP_insert_error.sql:4: ERROR: duplicate key
 ↳ value violates unique constraint "relate_o_and_p_pkey"
3 DETAIL: Key (fkoid, fkpoid)=(1, 1) already exists.
4 psql:conceptualToRelational/OP_insert_error.sql:4: STATEMENT: /* Try to
 ↳ create a relationship that already exists. */
5 -- The relationship (1, 1) already exists, so this will fail.
6 INSERT INTO relate_o_and_p (fkoid, fkpoid) VALUES (1, 1);
7 # psql 16.11 failed with exit code 3.
```



$Q \supseteq \bigcirc \text{---} \vdash R$



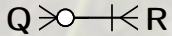
$$Q \multimap \bigcirc \text{---} | \multimap R$$

- Es gibt zwei Entitätstypen Q und R.

Q  $\Rightarrow$   $\circ$   $\dashv$  R



- Es gibt zwei Entitätstypen Q und R.
- Jede Entität vom Typ Q muss mit mindestens einer Entität vom Typ R verbunden sein, kann aber auch mit mehreren verbunden sein.

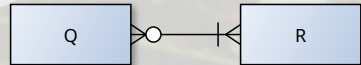
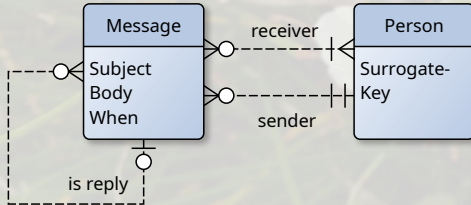


- Es gibt zwei Entitätstypen Q und R.
- Jede Entität vom Typ Q muss mit mindestens einer Entität vom Typ R verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ R kann mit keiner, einer, oder mehreren Entitäten vom Typ Q verbunden sein.

Q  $\bowtie$  ○  $\text{---}$   $\perp$  R



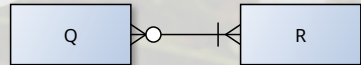
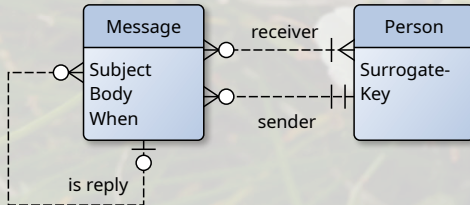
- Es gibt zwei Entitätstypen Q und R.
- Jede Entität vom Typ Q muss mit mindestens einer Entität vom Typ R verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ R kann mit keiner, einer, oder mehreren Entitäten vom Typ Q verbunden sein.
- Beispiel:



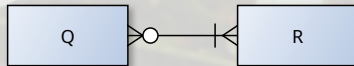
Q  $\Rightarrow$  ○  $\text{---}$   $\vdash$  R



- Es gibt zwei Entitätstypen Q und R.
- Jede Entität vom Typ Q muss mit mindestens einer Entität vom Typ R verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ R kann mit keiner, einer, oder mehreren Entitäten vom Typ Q verbunden sein.
- Beispiel:
  - Eine Nachricht muss mindestens einen Empfänger haben, könnte aber auch an mehrere Empfänger versendet werden.



- Q und R.
- uss mit mindestens einer Entität  
en verbunden sein.
- nn mit keiner, einer, oder mehrere
- ndestens einen Empfänger haben, l  
rden. Jede Person kann keine, eine
- 
- ```
graph LR; er[er] --- Person[Person];
```



Q $\rightarrow$ o $\leftarrow$ R: Lösung

- Bauen wir zuerst die Tabellen, die wir brauchen.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ O $\leftarrow$ R: Lösung

- Bauen wir zuerst die Tabellen, die wir brauchen.
- Dann kümmern wir uns darum, die referentielle Integrität mit passenden Einschränkungen zu sichern.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ O $\leftarrow$ R: Lösung

- Bauen wir zuerst die Tabellen, die wir brauchen.
- Dann kümmern wir uns darum, die referentielle Integrität mit passenden Einschränkungen zu sichern.
- Wir brauchen eine Tabelle **q** für die Entitäten vom Typ Q.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36
37 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ O $\leftarrow$ R: Lösung

- Bauen wir zuerst die Tabellen, die wir brauchen.
- Dann kümmern wir uns darum, die referentielle Integrität mit passenden Einschränkungen zu sichern.
- Wir brauchen eine Tabelle **q** für die Entitäten vom Typ Q.
- Der Primärschlüssel für diese Tabelle soll **qid** sein und es gibt wieder ein Beispielattribut **x**.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36
37 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Bauen wir zuerst die Tabellen, die wir brauchen.
- Dann kümmern wir uns darum, die referentielle Integrität mit passenden Einschränkungen zu sichern.
- Wir brauchen eine Tabelle **q** für die Entitäten vom Typ Q.
- Der Primärschlüssel für diese Tabelle soll **qid** sein und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **r** für die Entitäten vom Typ R.

```
1  /* Create the tables for a Q->R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ o $\leftarrow$ R: Lösung

- Dann kümmern wir uns darum, die referentielle Integrität mit passenden Einschränkungen zu sichern.
- Wir brauchen eine Tabelle **q** für die Entitäten vom Typ Q.
- Der Primärschlüssel für diese Tabelle soll **qid** sein und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **r** für die Entitäten vom Typ R.
- Diese bekommt den Primärschlüssel **rid** und das Beispielattribut **y**.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL, -- later used to reference R via relate_q_and_r
7      x      CHAR(3)       -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2) -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q R: Lösung

- Wir brauchen eine Tabelle **q** für die Entitäten vom Typ Q.
- Der Primärschlüssel für diese Tabelle soll **qid** sein und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **r** für die Entitäten vom Typ R.
- Diese bekommt den Primärschlüssel **rid** und das Beispielattribut **y**.
- Weil beide Enden der Beziehung es erlauben, dass eine Zeile mit mehreren Zeilen in der jeweils anderen Tabelle verbunden ist, brauchen wir definitiv wieder drei Tabellen.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29   ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Der Primärschlüssel für diese Tabelle soll `qid` sein und es gibt wieder ein Beispielattribut `x`.
- Wir brauchen auch eine Tabelle `r` für die Entitäten vom Typ R.
- Diese bekommt den Primärschlüssel `rid` und das Beispielattribut `y`.
- Weil beide Enden der Beziehung es erlauben, dass eine Zeile mit mehreren Zeilen in der jeweils anderen Tabelle verbunden ist, brauchen wir definitiv wieder drei Tabellen.
- Wir erstellen also die Tabelle `relate_q_and_r` um die Beziehungen zu managen.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Diese bekommt den Primärschlüssel `rid` und das Beispielattribut `y`.
- Weil beide Enden der Beziehung es erlauben, dass eine Zeile mit mehreren Zeilen in der jeweils anderen Tabelle verbunden ist, brauchen wir definitiv wieder drei Tabellen.
- Wir erstellen also die Tabelle `relate_q_and_r` um die Beziehungen zu managen.
- Diese Tabelle hat zwei Spalten, `fkqid` und `fkrid`, die Fremdschlüssel sind und jeweils auf die Primärschlüssel `qid` und `rid` der Tabellen `q` und `r` zeigen.

```
1  /* Create the tables for a Q->-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
```

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

Q $\multimap$ R: Lösung

- Weil beide Enden der Beziehung es erlauben, dass eine Zeile mit mehreren Zeilen in der jeweils anderen Tabelle verbunden ist, brauchen wir definitiv wieder drei Tabellen.
- Wir erstellen also die Tabelle `relate_q_and_r` um die Beziehungen zu managen.
- Diese Tabelle hat zwei Spalten, `fkqid` und `fkrid`, die Fremdschlüssel sind und jeweils auf die Primärschlüssel `qid` und `rid` der Tabellen `q` und `r` zeigen.
- Das sichern wir durch `REFERENCES`-Einschränkungen ab.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y      CHAR(2)          -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid  INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q R: Lösung

- Weil beide Enden der Beziehung es erlauben, dass eine Zeile mit mehreren Zeilen in der jeweils anderen Tabelle verbunden ist, brauchen wir definitiv wieder drei Tabellen.
- Wir erstellen also die Tabelle `relate_q_and_r` um die Beziehungen zu managen.
- Diese Tabelle hat zwei Spalten, `fkqid` und `fkrid`, die Fremdschlüssel sind und jeweils auf die Primärschlüssel `qid` und `rid` der Tabellen `q` und `r` zeigen.
- Das sichern wir durch `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind `NOT NULL`.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y      CHAR(2)          -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid  INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Wir erstellen also die Tabelle `relate_q_and_r` um die Beziehungen zu managen.
- Diese Tabelle hat zwei Spalten, `fkqid` und `fkrid`, die Fremdschlüssel sind und jeweils auf die Primärschlüssel `qid` und `rid` der Tabellen `q` und `r` zeigen.
- Das sichern wir durch `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind `NOT NULL`.
- Genau wie im vorigen Szenario kann jedes Paar `(fkqid, fkrid)` nur einmal in der Tabelle auftauchen.

```
1  /* Create the tables for a Q->-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ O $\leftarrow$ R: Lösung

- Diese Tabelle hat zwei Spalten, **fkqid** und **fkrid**, die Fremdschlüssel sind und jeweils auf die Primärschlüssel **qid** und **rid** der Tabellen **q** und **r** zeigen.
- Das sichern wir durch **REFERENCES**-Einschränkungen ab.
- Beide Spalten sind **NOT NULL**.
- Genau wie im vorigen Szenario kann jedes Paar (**fkqid**, **fkrid**) nur einmal in der Tabelle auftauchen.
- Das ist weil zwei Zeilen in den Tabellen **q** und **r** natürlich nur jeweils einmal miteinander in Beziehung stehen können.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)          -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
```

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

Q $\rightarrow$ R: Lösung

- Das sichern wir durch **REFERENCES**-Einschränkungen ab.
- Beide Spalten sind **NOT NULL**.
- Genau wie im vorigen Szenario kann jedes Paar (fkqid, fkrid) nur einmal in der Tabelle auftauchen.
- Das ist weil zwei Zeilen in den Tabellen **q** und **r** natürlich nur jeweils einmal miteinander in Beziehung stehen können.
- Das sichern wir also wieder über die Einschränkung **PRIMARY KEY** (fkqid, fkrid) ab.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
```

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

Q $\supseteq$ O --- R: Lösung

- Beide Spalten sind **NOT NULL**.
- Genau wie im vorigen Szenario kann jedes Paar (fkqid, fkrid) nur einmal in der Tabelle auftauchen.
- Das ist weil zwei Zeilen in den Tabellen **q** und **r** natürlich nur jeweils einmal miteinander in Beziehung stehen können.
- Das sichern wir also wieder über die Einschränkung **PRIMARY KEY (fkqid, fkrid)** ab.
- Im Vergleich zu O $\supseteq$ O --- P müssen wir das Problem lösen, dass wir erzwingen müssen, dass jede Zeile in Tabelle **q** mit mindestens einer Zeile in Tabelle **r** verbunden sein muss.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid     INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid   INT NOT NULL, -- later used to reference R via relate_q_and_r
7      x       CHAR(3)       -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid     INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y       CHAR(2) -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
```

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

Q $\bowtie$ R: Lösung

- Das ist weil zwei Zeilen in den Tabellen **q** und **r** natürlich nur jeweils einmal miteinander in Beziehung stehen können.
- Das sichern wir also wieder über die Einschränkung **PRIMARY KEY (fkqid, fkrid)** ab.
- Im Vergleich zu $O \bowtie P$ müssen wir das Problem lösen, dass wir erzwingen müssen, dass jede Zeile in Tabelle **q** mit mindestens einer Zeile in Tabelle **r** verbunden sein muss.
- Als wir $G \bowtie H$ behandelt haben, hatten wir ein ähnliches Problem.

```
1  /* Create the tables for a Q->O----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL, -- later used to reference R via relate_q_and_r
7      x      CHAR(3)       -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2) -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\supseteq$ O --- R: Lösung

- Das sichern wir also wieder über die Einschränkung
`PRIMARY KEY (fkqid, fkrid)` ab.
- Im Vergleich zu O $\supseteq$ O --- P müssen wir das Problem lösen, dass wir erzwingen müssen, dass jede Zeile in Tabelle `q` mit mindestens einer Zeile in Tabelle `r` verbunden sein muss.
- Als wir G --- O --- H behandelt haben, hatten wir ein ähnliches Problem.
- „Wie können wir erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden ist?“

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);

```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
3  CREATE TABLE
4  CREATE TABLE
5  CREATE TABLE
6  ALTER TABLE
7
8  # psql 16.11 succeeded with exit code 0.

```

Q $\multimap$ O --- R: Lösung

- Im Vergleich zu O $\multimap$ O --- P müssen wir das Problem lösen, dass wir erzwingen müssen, dass jede Zeile in Tabelle **q** mit mindestens einer Zeile in Tabelle **r** verbunden sein muss.
- Als wir G --- O --- H behandelt haben, hatten wir ein ähnliches Problem.
- „Wie können wir erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden ist?“
- Das Hauptproblem war nicht, zu erzwingen, dass eine neue Zeile **g** eine Zeile in Tabelle **h** referenziert ... sondern sicherzustellen, dass die Zeile in Tabelle **h** die selbe, neue Zeile in Tabelle **g** zurückreferenziert.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ o $\leftarrow$ R: Lösung

- „Wie können wir erzwingen, dass jede Entität vom Typ G mit mindestens einer Entität vom Typ H verbunden ist?“
- Das Hauptproblem war nicht, zu erzwingen, dass eine neue Zeile **g** eine Zeile in Tabelle **h** referenziert ... sondern sicherzustellen, dass die Zeile in Tabelle **h** die selbe, neue Zeile in Tabelle **g** zurückreferenziert.
- Wir haben das gelöst, in dem wir eine Einschränkung erstellt hatten, die erzwingt dass der Primärschlüssel dieser Zeile in Tabelle **h** zusammen mit dem Fremdschlüssel in Tabelle **g** gleich dem Fremdschlüssel in Tabelle **g** mit ihrem Primärschlüssel waren.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid     INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid   INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x       CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26 REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29   ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\multimap$ R: Lösung

- Das Hauptproblem war nicht, zu erzwingen, dass eine neue Zeile **g** eine Zeile in Tabelle **h** referenziert ... sondern sicherzustellen, dass die Zeile in Tabelle **h** die selbe, neue Zeile in Tabelle **g** zurückreferenziert.
- Wir haben das gelöst, in dem wir eine Einschränkung erstellt hatten, die erzwingt dass der Primärschlüssel dieser Zeile in Tabelle **h** zusammen mit dem Fremdschlüssel in Tabelle **g** gleich dem Fremdschlüssel in Tabelle **g** mit ihrem Primärschlüssel waren.
- Nun wird unsere Beziehung aber in Tabelle **relate_q_and_r** gemanaged.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30   ↳ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Wir haben das gelöst, in dem wir eine Einschränkung erstellt hatten, die erzwingt dass der Primärschlüssel dieser Zeile in Tabelle `h` zusammen mit dem Fremdschlüssel in Tabelle `g` gleich dem Fremdschlüssel in Tabelle `g` mit ihrem Primärschlüssel waren.
- Nun wird unsere Beziehung aber in Tabelle `relate_q_and_r` gemanaged.
- Was wir machen müssen ist also, sicherzustellen, dass wenn wir eine Zeile in Tabelle `q` einfügen, gleichzeitig eine Zeile in Tabelle `relate_q_and_r` eingefügt wird, die diese Zeile mit einer Zeile in Tabelle `r` verbindet.

```
1  /* Create the tables for a Q->O-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ o $\leftarrow$ R: Lösung

- Nun wird unsere Beziehung aber in Tabelle `relate_q_and_r` gemanaged.
- Was wir machen müssen ist also, sicherzustellen, dass wenn wir eine Zeile in Tabelle `q` einfügen, gleichzeitig eine Zeile in Tabelle `relate_q_and_r` eingefügt wird, die diese Zeile mit einer Zeile in Tabelle `r` verbindet.
- Wir speichern daher den „ersten R“-Fremdschlüssel auf eine Zeile in Tabelle `r` als Spalte `fkrid` in Tabelle `q`.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28
29 $ psql "postgres://postgres:XXX@localhost/relationships" -v
30 ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
31
32 CREATE TABLE
33 CREATE TABLE
34 CREATE TABLE
35 ALTER TABLE
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ o $\leftarrow$ R: Lösung

- Nun wird unsere Beziehung aber in Tabelle `relate_q_and_r` gemanaged.
- Was wir machen müssen ist also, sicherzustellen, dass wenn wir eine Zeile in Tabelle `q` einfügen, gleichzeitig eine Zeile in Tabelle `relate_q_and_r` eingefügt wird, die diese Zeile mit einer Zeile in Tabelle `r` verbindet.
- Wir speichern daher den „ersten R“-Fremdschlüssel auf eine Zeile in Tabelle `r` als Spalte `fkrid` in Tabelle `q`.
- Dann fügen wir die Einschränkung `q_qid_fkrid_fk` zur Tabelle `q` hinzu.

```
1  /* Create the tables for a Q->o-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29   ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q $\rightarrow$ R: Lösung

- Was wir machen müssen ist also, sicherzustellen, dass wenn wir eine Zeile in Tabelle `q` einfügen, gleichzeitig eine Zeile in Tabelle `relate_q_and_r` eingefügt wird, die diese Zeile mit einer Zeile in Tabelle `r` verbindet.
- Wir speichern daher den „ersten R“-Fremdschlüssel auf eine Zeile in Tabelle `r` als Spalte `fkrid` in Tabelle `q`.
- Dann fügen wir die Einschränkung `q_qid_fkrid_fk` zur Tabelle `q` hinzu.
- Diese erzwingt dass `FOREIGN KEY (qid, fkrid) REFERENCES relate_q_and_r (fkqid, fkrid)`.

```
1  /* Create the tables for a Q->-----|<-R relationship. */
2
3  -- Table Q: Each row in Q is related to one or multiple rows in R.
4  CREATE TABLE q (
5      qid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
6      fkrid  INT NOT NULL,    -- later used to reference R via relate_q_and_r
7      x      CHAR(3)         -- example for other attributes
8  );
9
10 -- Table R: Each row in R is related to zero, one, or many rows in Q.
11 CREATE TABLE r (
12     rid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
13     y   CHAR(2)    -- example for other attributes
14 );
15
16 -- The table for managing the relationship between Q and R.
17 CREATE TABLE relate_q_and_r (
18     fkqid INT NOT NULL REFERENCES q (qid),
19     fkrid  INT NOT NULL REFERENCES r (rid),
20     PRIMARY KEY (fkqid, fkrid) -- Primary key includes both columns.
21 );
22
23 -- To table Q, we add the foreign key reference constraint towards
24 -- table relate_q_and_r. This enforces that one relation must exist.
25 ALTER TABLE q ADD CONSTRAINT q_qid_fkrid_fk FOREIGN KEY (qid, fkrid)
26     REFERENCES relate_q_and_r (fkqid, fkrid);
27
28 $ psql "postgres://postgres:XXX@localhost/relationships" -v
29     ↪ ON_ERROR_STOP=1 -ebf QR_tables.sql
30
31 CREATE TABLE
32 CREATE TABLE
33 CREATE TABLE
34 ALTER TABLE
35
36 # psql 16.11 succeeded with exit code 0.
```

Q R: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.

```
1 /* Inserting data into the tables for the Q->R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10 RETURNING qid, fkrid)
11 INSERT INTO relate_q_and_r (fkqid, fkrid)
12 SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17 RETURNING qid, fkrid)
18 INSERT INTO relate_q_and_r (fkqid, fkrid)
19 SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24 RETURNING qid, fkrid)
25 INSERT INTO relate_q_and_r (fkqid, fkrid)
26 SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33 INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34 INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Für die Entitäten vom Typ R ist die Verbindung zu Entiten vom Typ Q optional.

```
1 /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10 RETURNING qid, fkrid)
11 INSERT INTO relate_q_and_r (fkqid, fkrid)
12 SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17 RETURNING qid, fkrid)
18 INSERT INTO relate_q_and_r (fkqid, fkrid)
19 SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24 RETURNING qid, fkrid)
25 INSERT INTO relate_q_and_r (fkqid, fkrid)
26 SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33 INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34 INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Für die Entitäten vom Typ R ist die Verbindung zu Entiten vom Typ Q optional.
- Befüllen wir also zuerst Tabelle **r**.

```
1 /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10 RETURNING qid, fkrid)
11 INSERT INTO relate_q_and_r (fkqid, fkrid)
12 SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17 RETURNING qid, fkrid)
18 INSERT INTO relate_q_and_r (fkqid, fkrid)
19 SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24 RETURNING qid, fkrid)
25 INSERT INTO relate_q_and_r (fkqid, fkrid)
26 SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33 INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34 INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Für die Entitäten vom Typ R ist die Verbindung zu Entiten vom Typ Q optional.
- Befüllen wir also zuerst Tabelle `r`.
- Wenn wir nun Zeilen in Tabelle `q` einfügen wollen, müssen wir gleichzeitig Zeilen in Tabelle `relate_q_and_r` einfügen.

```
1 /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10 RETURNING qid, fkrid)
11 INSERT INTO relate_q_and_r (fkqid, fkrid)
12 SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17 RETURNING qid, fkrid)
18 INSERT INTO relate_q_and_r (fkqid, fkrid)
19 SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24 RETURNING qid, fkrid)
25 INSERT INTO relate_q_and_r (fkqid, fkrid)
26 SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33 INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34 INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Für die Entitäten vom Typ R ist die Verbindung zu Entiten vom Typ Q optional.
- Befüllen wir also zuerst Tabelle `r`.
- Wenn wir nun Zeilen in Tabelle `q` einfügen wollen, müssen wir gleichzeitig Zeilen in Tabelle `relate_q_and_r` einfügen.
- Wenn wir das machen, dann müssen wir den Primärschlüsselwert der neuen Zeile für Tabelle `q` als Fremdschlüssel angeben.

```
1  /* Inserting data into the tables for the Q->o----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Für die Entitäten vom Typ R ist die Verbindung zu Entiten vom Typ Q optional.
- Befüllen wir also zuerst Tabelle `r`.
- Wenn wir nun Zeilen in Tabelle `q` einfügen wollen, müssen wir gleichzeitig Zeilen in Tabelle `relate_q_and_r` einfügen.
- Wenn wir das machen, dann müssen wir den Primärschlüsselwert der neuen Zeile für Tabelle `q` als Fremdschlüssel angeben.
- Wir benutzen also wieder eine CTE, die wir `q_new` nennen.

```
1  /* Inserting data into the tables for the Q->o----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q $\bowtie$ R: Befüllen mit Daten

- Befüllen wir also zuerst Tabelle **r**.
- Wenn wir nun Zeilen in Tabelle **q** einfügen wollen, müssen wir gleichzeitig Zeilen in Tabelle **relate_q_and_r** einfügen.
- Wenn wir das machen, dann müssen wir den Primärschlüsselwert der neuen Zeile für Tabelle **q** als Fremdschlüssel angeben.
- Wir benutzen also wieder eine CTE, die wir **q_new** nennen.
- Wir definieren ihn als **INSERT INTO q (x, fkrid) VALUES ('123', 1) RETURNING qid, fkrid.**

```
1 /* Inserting data into the tables for the Q->R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                 RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                 RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                 RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Wenn wir das machen, dann müssen wir den Primärschlüsselwert der neuen Zeile für Tabelle **q** als Fremdschlüssel angeben.
- Wir benutzen also wieder eine CTE, die wir **q_new** nennen.
- Wir definieren ihn als **INSERT INTO q (x, fkrid) VALUES ('123', 1) RETURNING qid, fkrid.**
- Wir wollen also eine Zeile in Tabelle **q** einfügen, deren **x** Attribut den Wert **'123'** hat und die in Beziehung steht mit der Zeile in Tabelle **r**, die den Primärschlüsselwert **1** hat.

```
1  /* Inserting data into the tables for the Q->R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Wir benutzen also wieder eine CTE, die wir `q_new` nennen.
- Wir definieren ihn als `INSERT INTO q (x, fkrid) VALUES ('123', 1) RETURNING qid, fkrid`.
- Wir wollen also eine Zeile in Tabelle `q` einfügen, deren `x` Attribut den Wert `'123'` hat und die in Beziehung steht mit der Zeile in Tabelle `r`, die den Primärschlüsselwert `1` hat.
- Wenn diese Einfügung ausgeführt wird, gibt sie uns den Primärschlüsselwert der neuen Zeile als `qid` und den Fremdschlüsselwert auf die Zeile in Tabelle `r` als `fkrid` (mit Wert `1`) zurück.

```
1  /* Inserting data into the tables for the Q->R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q $\bowtie$ R: Befüllen mit Daten

- Wir wollen also eine Zeile in Tabelle `q` einfügen, deren `x` Attribut den Wert `'123'` hat und die in Beziehung steht mit der Zeile in Tabelle `r`, die den Primärschlüsselwert `1` hat.
- Wenn diese Einfügung ausgeführt wird, gibt sie uns den Primärschlüsselwert der neuen Zeile als `qid` und den Fremdschlüsselwert auf die Zeile in Tabelle `r` als `fkrid` (mit Wert `1`) zurück.
- Diesen CTE benutzen wir dann, um eine neue Zeile in Tabelle `relate_q_and_r` einzufügen.

```

1  /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;

```

```

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

```



Q R: Befüllen mit Daten

- Wenn diese Einfügung ausgeführt wird, gibt sie uns den Primärschlüsselwert der neuen Zeile als `qid` und den Fremdschlüsselwert auf die Zeile in Tabelle `r` als `fkrid` (mit Wert 1) zurück.
- Diesen CTE benutzen wir dann, um eine neue Zeile in Tabelle `relate_q_and_r` einzufügen.
- Wir schreiben `INSERT INTO relate_q_and_r (fkqid, fkrid) SELECT qid, fkrid FROM q_new;`

```
1 /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                 RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                 RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                 RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Wenn diese Einfügung ausgeführt wird, gibt sie uns den Primärschlüsselwert der neuen Zeile als `qid` und den Fremdschlüsselwert auf die Zeile in Tabelle `r` als `fkrid` (mit Wert 1) zurück.
- Diesen CTE benutzen wir dann, um eine neue Zeile in Tabelle `relate_q_and_r` einzufügen.
- Wir schreiben `INSERT INTO relate_q_and_r (fkqid, fkrid) SELECT qid, fkrid FROM q_new;`
- Das ist recht einfach.

```
1 /* Inserting data into the tables for the Q->R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                 RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                 RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                 RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Diesen CTE benutzen wir dann, um eine neue Zeile in Tabelle `relate_q_and_r` einzufügen.
- Wir schreiben `INSERT INTO relate_q_and_r (fkqid, fkrid)`
`SELECT qid, fkrid FROM q_new;`
- Das ist recht einfach.
- Die neue Zeile in Tabelle `relate_o_and_p` bekommt den Primärschlüsselwert, der auf die neue Zeile in Tabelle `q` zeigt.

```
1 /* Inserting data into the tables for the Q->o-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                 RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                 RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                 RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Diesen CTE benutzen wir dann, um eine neue Zeile in Tabelle `relate_q_and_r` einzufügen.
- Wir schreiben `INSERT INTO relate_q_and_r (fkqid, fkrid)`
`SELECT qid, fkrid FROM q_new;`
- Das ist recht einfach.
- Die neue Zeile in Tabelle `relate_o_and_p` bekommt den Primärschlüsselwert, der auf die neue Zeile in Tabelle `q` zeigt.
- Zusätzlich bekommt sie den Primärschlüsselwert der Zeile in Tabelle `r`, die diese referenziert.

```
1 /* Inserting data into the tables for the Q->O-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q $\bowtie$ R: Befüllen mit Daten

- Wir schreiben `INSERT INTO relate_q_and_r (fkqid, fkrid) SELECT qid, fkrid FROM q_new;`.
- Das ist recht einfach.
- Die neue Zeile in Tabelle `relate_o_and_p` bekommt den Primärschlüsselwert, der auf die neue Zeile in Tabelle `q` zeigt.
- Zusätzlich bekommt sie den Primärschlüsselwert der Zeile in Tabelle `r`, die diese referenziert.
- Beide Einfügungen werden zusammen als ein Kommando ausgeführt und die referentielle Integrität wird an dessen Ende überprüft.

```
1 /* Inserting data into the tables for the Q->o----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Das ist recht einfach.
- Die neue Zeile in Tabelle `relate_o_and_p` bekommt den Primärschlüsselwert, der auf die neue Zeile in Tabelle `q` zeigt.
- Zusätzlich bekommt sie den Primärschlüsselwert der Zeile in Tabelle `r`, die diese referenziert.
- Beide Einfügungen werden zusammen als ein Kommando ausgeführt und die referentielle Integrität wird an dessen Ende überprüft.
- Und die passt natürlich.

```
1  /* Inserting data into the tables for the Q->o-----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Die neue Zeile in Tabelle `relate_o_and_p` bekommt den Primärschlüsselwert, der auf die neue Zeile in Tabelle `q` zeigt.
- Zusätzlich bekommt sie den Primärschlüsselwert der Zeile in Tabelle `r`, die diese referenziert.
- Beide Einfügungen werden zusammen als ein Kommando ausgeführt und die referentielle Integrität wird an dessen Ende überprüft.
- Und die passt natürlich.
- Wir fügen ein paar Zeilen in die Tabellen `q` und `relate_q_and_r` auf diese Art ein.

```
1 /* Inserting data into the tables for the Q->o-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10 RETURNING qid, fkrid)
11 INSERT INTO relate_q_and_r (fkqid, fkrid)
12 SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17 RETURNING qid, fkrid)
18 INSERT INTO relate_q_and_r (fkqid, fkrid)
19 SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24 RETURNING qid, fkrid)
25 INSERT INTO relate_q_and_r (fkqid, fkrid)
26 SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33 INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34 INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Zusätzlich bekommt sie den Primärschlüsselwert der Zeile in Tabelle **r**, die diese referenziert.
- Beide Einfügungen werden zusammen als ein Kommando ausgeführt und die referentielle Integrität wird an dessen Ende überprüft.
- Und die passt natürlich.
- Wir fügen ein paar Zeilen in die Tabellen **q** und **relate_q_and_r** auf diese Art ein.
- Danach können wir weitere Verbindungen als Zeilen in Tabelle **relate_q_and_r** einfügen.

```
1 /* Inserting data into the tables for the Q->o-----|<-R relationship. */
2
3 -- Insert some rows into the table for entity type R first.
4 -- We can only create rows in Q related to existing R entities.
5 INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7 -- Insert into Q and relate_q_and_r at the same time.
8 -- This creates Q entry with id 1 and relationship (1, 1).
9 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                 RETURNING qid, fkrid)
11     INSERT INTO relate_q_and_r (fkqid, fkrid)
12         SELECT qid, fkrid FROM q_new;
13
14 -- Insert into Q and relate_q_and_r at the same time.
15 -- This creates Q entry with id 2 and relationship (2, 4).
16 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                 RETURNING qid, fkrid)
18     INSERT INTO relate_q_and_r (fkqid, fkrid)
19         SELECT qid, fkrid FROM q_new;
20
21 -- Insert into Q and relate_q_and_r at the same time.
22 -- This creates Q entry with id 2 and relationship (3, 1).
23 WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                 RETURNING qid, fkrid)
25     INSERT INTO relate_q_and_r (fkqid, fkrid)
26         SELECT qid, fkrid FROM q_new;
27
28 -- We can now insert additional relationships
29 INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31 -- Combine the rows from Q and R. This needs two INNER JOINS.
32 SELECT qid, x, rid, y FROM relate_q_and_r
33     INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34     INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q R: Befüllen mit Daten

- Beide Einfügungen werden zusammen als ein Kommando ausgeführt und die referentielle Integrität wird an dessen Ende überprüft.
- Und die passt natürlich.
- Wir fügen ein paar Zeilen in die Tabellen `q` und `relate_q_and_r` auf diese Art ein.
- Danach können wir weitere Verbindungen als Zeilen in Tabelle `relate_q_and_r` einfügen.
- Die Daten können wir wieder mit zwei `INNER JOIN`s zusammensetzen.

```
1  /* Inserting data into the tables for the Q->o-----|<-R relationship. */
2
3  -- Insert some rows into the table for entity type R first.
4  -- We can only create rows in Q related to existing R entities.
5  INSERT INTO r (y) VALUES ('AB'), ('CD'), ('EF'), ('GH'), ('IJ'), ('KL');
6
7  -- Insert into Q and relate_q_and_r at the same time.
8  -- This creates Q entry with id 1 and relationship (1, 1).
9  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('123', 1)
10                  RETURNING qid, fkrid)
11    INSERT INTO relate_q_and_r (fkqid, fkrid)
12      SELECT qid, fkrid FROM q_new;
13
14  -- Insert into Q and relate_q_and_r at the same time.
15  -- This creates Q entry with id 2 and relationship (2, 4).
16  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('456', 4)
17                  RETURNING qid, fkrid)
18    INSERT INTO relate_q_and_r (fkqid, fkrid)
19      SELECT qid, fkrid FROM q_new;
20
21  -- Insert into Q and relate_q_and_r at the same time.
22  -- This creates Q entry with id 2 and relationship (3, 1).
23  WITH q_new AS (INSERT INTO q (x, fkrid) VALUES ('789', 1)
24                  RETURNING qid, fkrid)
25    INSERT INTO relate_q_and_r (fkqid, fkrid)
26      SELECT qid, fkrid FROM q_new;
27
28  -- We can now insert additional relationships
29  INSERT INTO relate_q_and_r VALUES (1, 2), (2, 3), (2, 5);
30
31  -- Combine the rows from Q and R. This needs two INNER JOINS.
32  SELECT qid, x, rid, y FROM relate_q_and_r
33    INNER JOIN q ON q.qid = relate_q_and_r.fkqid
34    INNER JOIN r ON r.rid = relate_q_and_r.fkrid;
```

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



Q $\bowtie$ R: Der Inhalt der Drei Tabellen



- Dies sind die Daten in den drei Tabellen.

Table **q**

qid	fkrid	x
1	1	„123“
2	4	„456“
3	1	„789“

Table **r**

rid	y
1	„AB“
2	„CD“
3	„EF“
4	„GH“
5	„IJ“
6	„KL“

Table **relate_q_and_r**

fkqid	fkrid
1	1
2	4
3	1
1	2
2	3
2	5

$Q \rhd \bigcirc \dashv \vdash R$: Testen der Einschränkung

- Wir können keine Zeile in Tabelle **q** einfügen, die nicht einen Fremdschlüssel auf eine Zeile in Tabelle **r** hat.

```
1 /* Can we create a row in Q unrelated to any row in R? */
```

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

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v  
  ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_1.sql
```

```
2 psql:conceptualToRelational/QR_insert_error_1.sql:3: ERROR: null value  
  ↳ in column "fkrid" of relation "q" violates not-null constraint
```

```
3 DETAIL: Failing row contains (4, null, 777).
```

```
4 psql:conceptualToRelational/QR_insert_error_1.sql:3: STATEMENT: /* Can  
  ↳ we create a row in Q unrelated to any row in R? */
```

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

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

Q R: Testen der Einschränkungen

- Wir können keine Zeile in Tabelle **q** einfügen, die nicht einen Fremdschlüssel auf eine Zeile in Tabelle **r** hat.
- Wir können auch keine Zeile in Tabelle **q** einfügen, die so einen Fremdschlüssel bereitstellt, für den es aber keine passende Zeile in Tabelle **relate_q_and_r** gibt.

```
1 /* Can we create a row in Q unrelated to any row in R? */
```

```
2 INSERT INTO q (x) VALUES ('777');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_1.sql
2 psql:conceptualToRelational/QR_insert_error_1.sql:3: ERROR: null value
   ↳ in column "fkrid" of relation "q" violates not-null constraint
3 DETAIL: Failing row contains (4, null, 777).
4 psql:conceptualToRelational/QR_insert_error_1.sql:3: STATEMENT: /* Can
   ↳ we create a row in Q unrelated to any row in R? */
5 INSERT INTO q (x) VALUES ('777');
6 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we create a row in Q ignoring `relate_q_and_r` table? */
```

```
2
3 INSERT INTO q (fkrid, x) VALUES (6, '888');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_2.sql
2 psql:conceptualToRelational/QR_insert_error_2.sql:3: ERROR: insert or
   ↳ update on table "q" violates foreign key constraint "
   ↳ q_qid_fkrid_fk"
3 DETAIL: Key (qid, fkrid)=(5, 6) is not present in table "relate_q_and_r"
   ↳ ".
4 psql:conceptualToRelational/QR_insert_error_2.sql:3: STATEMENT: /* Can
   ↳ we create a row in Q ignoring `relate_q_and_r` table? */
5 INSERT INTO q (fkrid, x) VALUES (6, '888');
6 # psql 16.11 failed with exit code 3.
```

Q $\rightarrow$ R: Testen der Einschränkung

- Wir können keine Zeile in Tabelle **q** einfügen, die nicht einen Fremdschlüssel auf eine Zeile in Tabelle **r** hat.
- Wir können auch keine Zeile in Tabelle **q** einfügen, die so einen Fremdschlüssel bereitstellt, für den es aber keine passende Zeile in Tabelle **relate_q_and_r** gibt.
- Wir können auch nicht zuerst eine Zeile in Tabelle **relate_q_and_r** einfügen und danach die passende Zeile in Tabelle **q** erzeugen, selbst wenn wir den Primärschlüssel dieser zweiten Zeile vorher kennen würden.

```
1 /* Can we create the row in `relate_q_and_r` first? */
2
3 INSERT INTO relate_q_and_r VALUES (10, 4);
4 INSERT INTO q (qid, fkrid, x) VALUES (10, 4, '999');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_3.sql
2 psql:conceptualToRelational/QR_insert_error_3.sql:3: ERROR: insert or
   ↳ update on table "relate_q_and_r" violates foreign key constraint "
   ↳ relate_q_and_r_fkqid_fk"
3 DETAIL: Key (fkqid)=(10) is not present in table "q".
4 psql:conceptualToRelational/QR_insert_error_3.sql:3: STATEMENT: /* Can
   ↳ we create the row in `relate_q_and_r` first? */
5 INSERT INTO relate_q_and_r VALUES (10, 4);
6 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we create a row in Q ignoring `relate_q_and_r` table? */
2
3 INSERT INTO q (fkrid, x) VALUES (6, '888');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_2.sql
2 psql:conceptualToRelational/QR_insert_error_2.sql:3: ERROR: insert or
   ↳ update on table "q" violates foreign key constraint "
   ↳ q_qid_fkrid_fk"
3 DETAIL: Key (qid, fkrid)=(5, 6) is not present in table "relate_q_and_r"
   ↳ ".
4 psql:conceptualToRelational/QR_insert_error_2.sql:3: STATEMENT: /* Can
   ↳ we create a row in Q ignoring `relate_q_and_r` table? */
5 INSERT INTO q (fkrid, x) VALUES (6, '888');
6 # psql 16.11 failed with exit code 3.
```

Q R: Testen der Einschränkungen

- Wir können keine Zeile in Tabelle `q` einfügen, die nicht einen Fremdschlüssel auf eine Zeile in Tabelle `r` hat.
- Wir können auch keine Zeile in Tabelle `q` einfügen, die so einen Fremdschlüssel bereitstellt, für den es aber keine passende Zeile in Tabelle `relate_q_and_r` gibt.
- Wir können auch nicht zuerst eine Zeile in Tabelle `relate_q_and_r` einfügen und danach die passende Zeile in Tabelle `q` erzeugen, selbst wenn wir den Primärschlüssel dieser zweiten Zeile vorher kennen würden.
- Die Einschränkungen beschützen die referentielle Integrität also sehr gut.

```
1 /* Can we create the row in `relate_q_and_r` first? */
2
3 INSERT INTO relate_q_and_r VALUES (10, 4);
4 INSERT INTO q (qid, fkrid, x) VALUES (10, 4, '999');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_3.sql
2 psql:conceptualToRelational/QR_insert_error_3.sql:3: ERROR: insert or
   ↳ update on table "relate_q_and_r" violates foreign key constraint "
   ↳ relate_q_and_r_fkqid_fkey"
3 DETAIL: Key (fkqid)=(10) is not present in table "q".
4 psql:conceptualToRelational/QR_insert_error_3.sql:3: STATEMENT: /* Can
   ↳ we create the row in `relate_q_and_r` first? */
5 INSERT INTO relate_q_and_r VALUES (10, 4);
6 # psql 16.11 failed with exit code 3.
```

```
1 /* Can we create a row in Q ignoring `relate_q_and_r` table? */
2
3 INSERT INTO q (fkrid, x) VALUES (6, '888');
```

```
1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf QR_insert_error_2.sql
2 psql:conceptualToRelational/QR_insert_error_2.sql:3: ERROR: insert or
   ↳ update on table "q" violates foreign key constraint "
   ↳ q_qid_fkrid_fk"
3 DETAIL: Key (qid, fkrid)=(5, 6) is not present in table "relate_q_and_r"
   ↳ ".
4 psql:conceptualToRelational/QR_insert_error_2.sql:3: STATEMENT: /* Can
   ↳ we create a row in Q ignoring `relate_q_and_r` table? */
5 INSERT INTO q (fkrid, x) VALUES (6, '888');
6 # psql 16.11 failed with exit code 3.
```



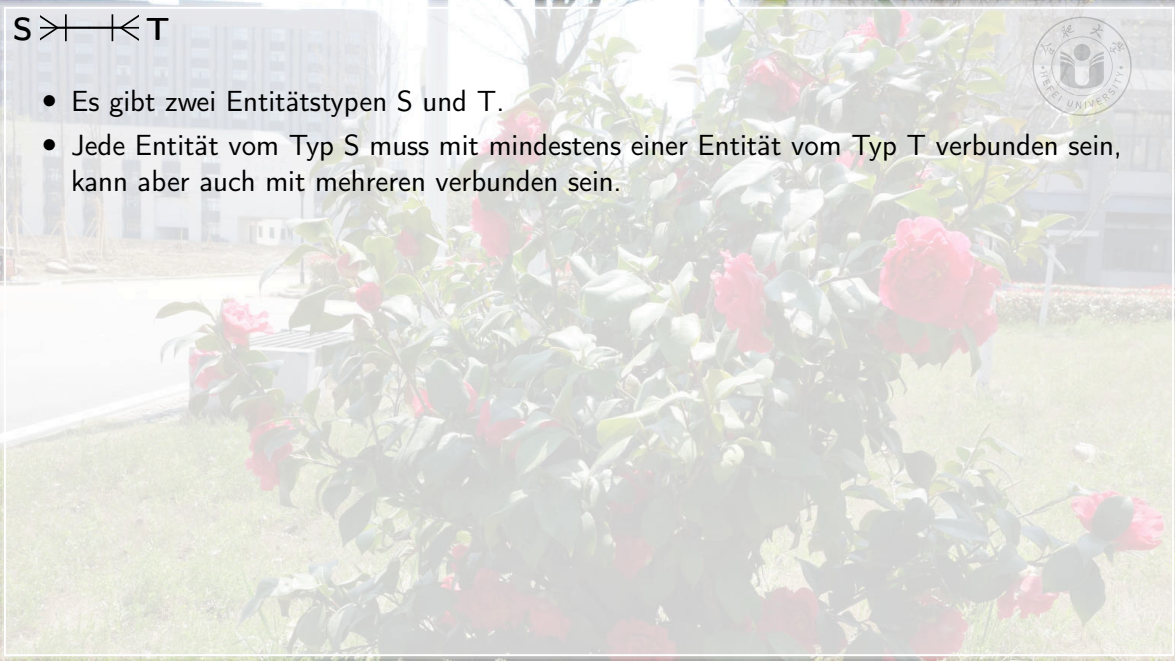
S $\gg$ | $\ll$ T



S $\bowtie$ T

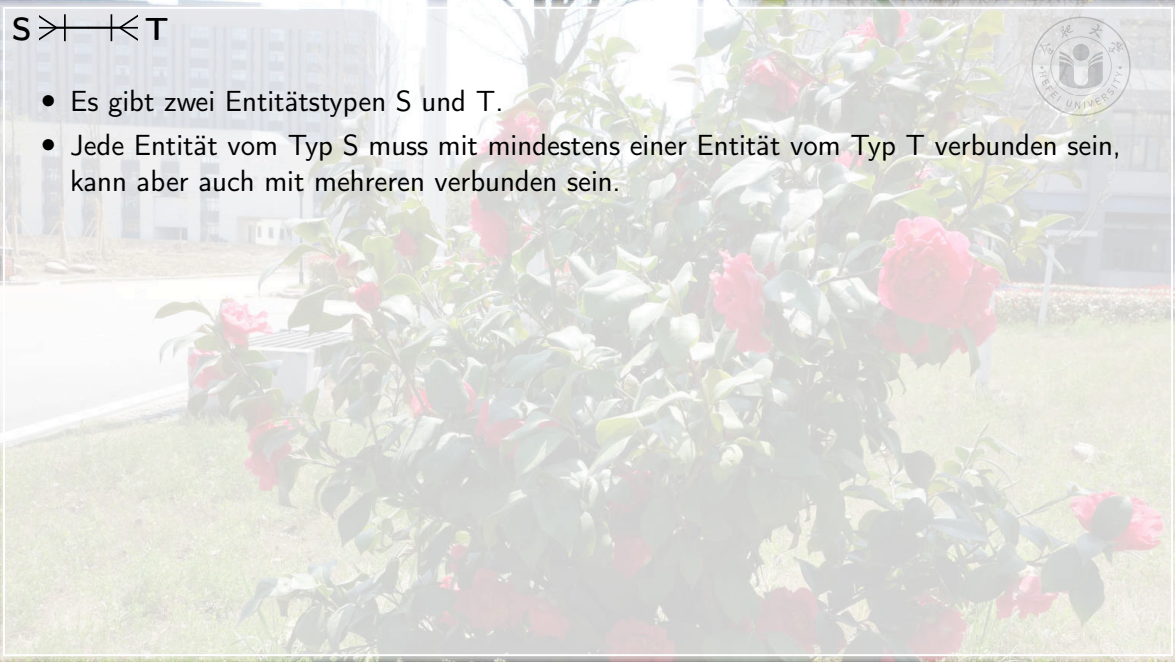
- Es gibt zwei Entitätstypen S und T.

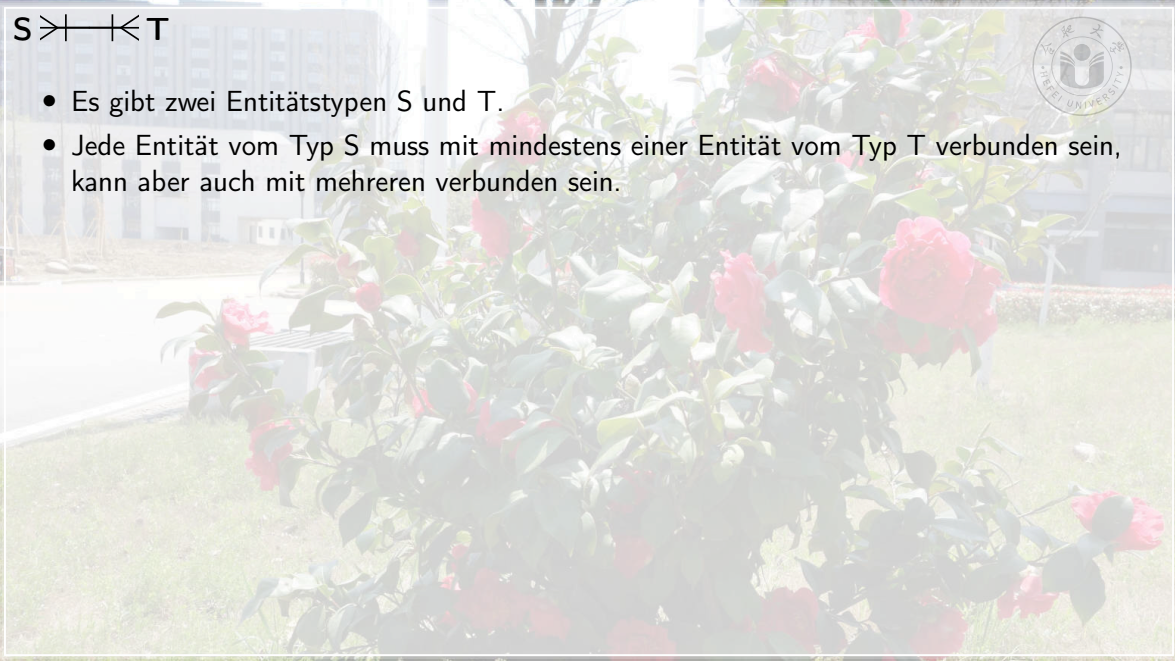




S $\rhd$ | $\lhd$ T

- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.



- 
- S $\rhd$ | $\lhd$ T
- Es gibt zwei Entitätstypen S und T.
 - Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- 

S $\Rightarrow$ $\vdash$ $\vdash$ $\vdash$ $\vdash$ T

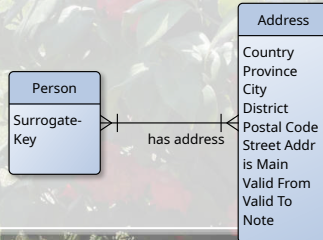
- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ T muss mit mindestens einer Entität vom Typ S verbunden sein, kann aber auch mit mehreren verbunden sein.



S $\gg$ --- $\ll$ T



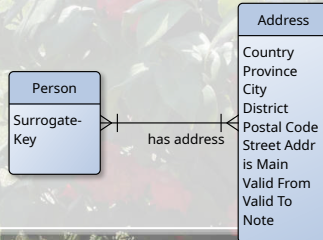
- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ T muss mit mindestens einer Entität vom Typ S verbunden sein, kann aber auch mit mehreren verbunden sein.
- Beispiel:



S $\gg$ --- $\ll$ T



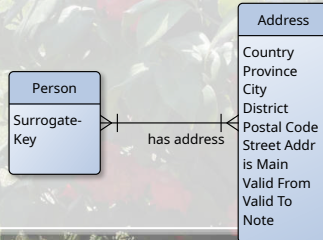
- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ T muss mit mindestens einer Entität vom Typ S verbunden sein, kann aber auch mit mehreren verbunden sein.
- Beispiel:
 - Wir kennen das aus der Beziehung von Personen und Adressen.



S $\gg$ $\ll$ T



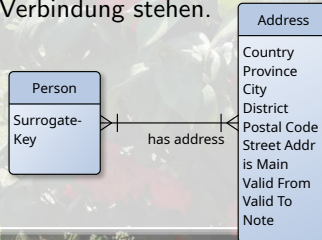
- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ T muss mit mindestens einer Entität vom Typ S verbunden sein, kann aber auch mit mehreren verbunden sein.
- Beispiel:
 - Wir kennen das aus der Beziehung von Personen und Adressen. Jede Person muss mindestens eine Adresse haben, kann aber auch mehrere haben.



S $\Rightarrow$ | $\Leftarrow$ T



- Es gibt zwei Entitätstypen S und T.
- Jede Entität vom Typ S muss mit mindestens einer Entität vom Typ T verbunden sein, kann aber auch mit mehreren verbunden sein.
- Jede Entität vom Typ T muss mit mindestens einer Entität vom Typ S verbunden sein, kann aber auch mit mehreren verbunden sein.
- Beispiel:
 - Wir kennen das aus der Beziehung von Personen und Adressen. Jede Person muss mindestens eine Adresse haben, kann aber auch mehrere haben. Jede Adresse muss mit mindestens einer Person in Verbindung stehen (sonst würden wir sie ja nicht speichern), kann aber auch mit mehreren Personen in Verbindung stehen.



S ↔ T: Lösung

- Fangen wir also zuerst wieder mit den grundlegenden Tabellen an.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Fangen wir also zuerst wieder mit den grundlegenden Tabellen an.
- Wir brauchen eine Tabelle **s** für die Entitäten vom Typ S.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S ⇨ ⇨ ⇨ T: Lösung

- Fangen wir also zuerst wieder mit den grundlegenden Tabellen an.
- Wir brauchen eine Tabelle **s** für die Entitäten vom Typ S.
- Der Primärschlüssel ist diesmal **sid** und es gibt wieder ein Beispielattribut **x**.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     ktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, ktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, ktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, ktid);
```

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

S T: Lösung

- Fangen wir also zuerst wieder mit den grundlegenden Tabellen an.
- Wir brauchen eine Tabelle **s** für die Entitäten vom Typ S.
- Der Primärschlüssel ist diesmal **sid** und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **t** für die Entitäten vom Typ T.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Fangen wir also zuerst wieder mit den grundlegenden Tabellen an.
- Wir brauchen eine Tabelle **s** für die Entitäten vom Typ S.
- Der Primärschlüssel ist diesmal **sid** und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **t** für die Entitäten vom Typ T.
- Sie bekommt Primärschlüssel **tid** und Beispielattribut **y**.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Wir brauchen eine Tabelle **s** für die Entitäten vom Typ S.
- Der Primärschlüssel ist diesmal **sid** und es gibt wieder ein Beispielattribut **x**.
- Wir brauchen auch eine Tabelle **t** für die Entitäten vom Typ T.
- Sie bekommt Primärschlüssel **tid** und Beispielattribut **y**.
- Wir brauchen auch eine dritte Tabelle um die Beziehungen zu speichern.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Der Primärschlüssel ist diesmal `sid` und es gibt wieder ein Beispielattribut `x`.
- Wir brauchen auch eine Tabelle `t` für die Entitäten vom Typ T.
- Sie bekommt Primärschlüssel `tid` und Beispielattribut `y`.
- Wir brauchen auch eine dritte Tabelle um die Beziehungen zu speichern.
- Wir nennen sie `relate_s_and_t`.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Wir brauchen auch eine Tabelle **t** für die Entitäten vom Typ T.
- Sie bekommt Primärschlüssel **tid** und Beispielattribut **y**.
- Wir brauchen auch eine dritte Tabelle um die Beziehungen zu speichern.
- Wir nennen sie **relate_s_and_t**.
- Diese Tabelle die zwei Spalten **fksid** und **fktid** die Fremdschlüssel auf die Primärschlüssel **sid** und **tid** der Tabellen **s** und **t** sind.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Sie bekommt Primärschlüssel `tid` und Beispielattribut `y`.
- Wir brauchen auch eine dritte Tabelle um die Beziehungen zu speichern.
- Wir nennen sie `relate_s_and_t`.
- Diese Tabelle die zwei Spalten `fksid` und `fktid` die Fremdschlüssel auf die Primärschlüssel `sid` und `tid` der Tabellen `s` und `t` sind.
- Das sichern wir wieder über entsprechende `REFERENCES`-Einschränkungen ab.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S ↔ T: Lösung

- Wir brauchen auch eine dritte Tabelle um die Beziehungen zu speichern.
- Wir nennen sie `relate_s_and_t`.
- Diese Tabelle die zwei Spalten `fksid` und `fktid` die Fremdschlüssel auf die Primärschlüssel `sid` und `tid` der Tabellen `s` und `t` sind.
- Das sichern wir wieder über entsprechende `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind als `NOT NULL` markiert, denn keiner der Werte kann weggelassen werden.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\rightarrow$ | $\leftarrow$ T: Lösung

- Wir nennen sie `relate_s_and_t`.
- Diese Tabelle die zwei Spalten `fksid` und `fktid` die Fremdschlüssel auf die Primärschlüssel `sid` und `tid` der Tabellen `s` und `t` sind.
- Das sichern wir wieder über entsprechende `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind als `NOT NULL` markiert, denn keiner der Werte kann weggelassen werden.
- Wie im Fall von $O \rightarrow O \leftarrow P$, darf jedes Paar `(fksid, fktid)` nur einmal in der Tabelle auftauchen.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\rightarrow$ | $\leftarrow$ T: Lösung

- Diese Tabelle die zwei Spalten `fksid` und `fktid` die Fremdschlüssel auf die Primärschlüssel `sid` und `tid` der Tabellen `s` und `t` sind.
- Das sichern wir wieder über entsprechende `REFERENCES`-Einschränkungen ab.
- Beide Spalten sind als `NOT NULL` markiert, denn keiner der Werte kann weggelassen werden.
- Wie im Fall von $O \rightarrow \bigcirc \leftarrow P$, darf jedes Paar `(fksid, fktid)` nur einmal in der Tabelle auftauchen.
- Zwei Zeilen der Tabellen `s` und `t` können ja nur einmal verbunden sein.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Das sichern wir wieder über entsprechende **REFERENCES**-Einschränkungen ab.
- Beide Spalten sind als **NOT NULL** markiert, denn keiner der Werte kann weggelassen werden.
- Wie im Fall von $O \rightarrow O \leftarrow P$, darf jedes Paar (**fksid**, **fktid**) nur einmal in der Tabelle auftauchen.
- Zwei Zeilen der Tabellen **s** und **t** können ja nur einmal verbunden sein.
- Das implementieren wir über die Einschränkung **PRIMARY KEY (fksid, fktid)**.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Beide Spalten sind als **NOT NULL** markiert, denn keiner der Werte kann weggelassen werden.
- Wie im Fall von $O \rightarrow \text{---} | \leftarrow \text{---} P$, darf jedes Paar **(fksid, fktid)** nur einmal in der Tabelle auftauchen.
- Zwei Zeilen der Tabellen **s** und **t** können ja nur einmal verbunden sein.
- Das implementieren wir über die Einschränkung **PRIMARY KEY (fksid, fktid)**.
- Wie beim Beziehungstyp $M \text{---} | \leftarrow \text{---} N$ sind beide Beziehungsenden zwingend.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid      INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid    INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x        CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid    INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y        CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\bowtie$ T: Lösung

- Wie im Fall von $O \bowtie P$, darf jedes Paar `(fkssid, fktid)` nur einmal in der Tabelle auftauchen.
- Zwei Zeilen der Tabellen `s` und `t` können ja nur einmal verbunden sein.
- Das implementieren wir über die Einschränkung `PRIMARY KEY (fkssid, fktid)`.
- Wie beim Beziehungstyp $M \bowtie N$ sind beide Beziehungsenden zwingend.
- Wir können keine Zeile in Tabelle `s` erstellen, ohne diese mit einer Zeile in Tabelle `t` zu verbinden.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkssid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fkssid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fkssid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fkssid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fkssid_tid_fk FOREIGN KEY (fkssid, tid)
32     REFERENCES relate_s_and_t (fkssid, fktid);
```

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

S T: Lösung

- Zwei Zeilen der Tabellen **s** und **t** können ja nur einmal verbunden sein.
- Das implementieren wir über die Einschränkung **PRIMARY KEY (fksid, fktid)**.
- Wie beim Beziehungstyp M  N sind beide Beziehungsenden zwingend.
- Wir können keine Zeile in Tabelle **s** erstellen, ohne diese mit einer Zeile in Tabelle **t** zu verbinden.
- Wir können auch keine Zeile in Tabelle **t** erstellen, ohne diese mit einer Zeile in Tabelle **s** zu verbinden.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

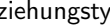
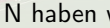
S $\rightrightarrows$ $\leftarrow$ T: Lösung

- Das implementieren wir über die Einschränkung `PRIMARY KEY (fksid, fktid)`.
- Wie beim Beziehungstyp $M \rightrightarrows \leftarrow N$ sind beide Beziehungsenden zwingend.
- Wir können keine Zeile in Tabelle `s` erstellen, ohne diese mit einer Zeile in Tabelle `t` zu verbinden.
- Wir können auch keine Zeile in Tabelle `t` erstellen, ohne diese mit einer Zeile in Tabelle `s` zu verbinden.
- Bei $M \rightrightarrows \leftarrow N$ haben wir dieses Henne-Ei-Problem so gelöst, dass wir eine Sequenz erstellt haben, von der wir die Primärschlüsselwerte für Tabelle `m` errechnet haben.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Wie beim Beziehungstyp M  N sind beide Beziehungsenden zwingend.
- Wir können keine Zeile in Tabelle **s** erstellen, ohne diese mit einer Zeile in Tabelle **t** zu verbinden.
- Wir können auch keine Zeile in Tabelle **t** erstellen, ohne diese mit einer Zeile in Tabelle **s** zu verbinden.
- Bei M  N haben wir dieses Henne-Ei-Problem so gelöst, dass wir eine Sequenz erstellt haben, von der wir die Primärschlüsselwerte für Tabelle **m** errechnet haben.
- Wir benutzten diese Sequenz um zuerst immer einen Primärschlüsselwert für Tabelle **m** zu erzeugen.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

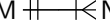
S $\rightarrow$ | $\leftarrow$ T: Lösung

- Wir können auch keine Zeile in Tabelle **t** erstellen, ohne diese mit einer Zeile in Tabelle **s** zu verbinden.
- Bei M $\rightarrow$ | $\leftarrow$ N haben wir dieses Henne-Ei-Problem so gelöst, dass wir eine Sequenz erstellt haben, von der wir die Primärschlüsselwerte für Tabelle **m** errechnet haben.
- Wir benutzen diese Sequenz um zuerst immer einen Primärschlüsselwert für Tabelle **m** zu erzeugen.
- Dann benutzen wir diesen Primärschlüsselwert als Fremdschlüsselwert für Tabelle **n**.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Lösung

- Bei M  N haben wir dieses Henne-Ei-Problem so gelöst, dass wir eine Sequenz erstellt haben, von der wir die Primärschlüsselwerte für Tabelle **m** errechnet haben.
- Wir benutzten diese Sequenz um zuerst immer einen Primärschlüsselwert für Tabelle **m** zu erzeugen.
- Dann benutzen wir diesen Primärschlüsselwert als Fremdschlüsselwert für Tabelle **n**.
- Danach benutzen wir den Primärschlüsselwert dieser neuen Zeile zusammen mit dem für Tabelle **m** um letztendlich die Zeile in Tabelle **m** einzufügen.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\rightrightarrows$ T: Lösung

- Wir benutzten diese Sequenz um zuerst immer einen Primärschlüsselwert für Tabelle **m** zu erzeugen.
- Dann benutzen wir diesen Primärschlüsselwert als Fremdschlüsselwert für Tabelle **n**.
- Danach benutzten wir den Primärschlüsselwert dieser neuen Zeile zusammen mit dem für Tabelle **m** um letztendlich die Zeile in Tabelle **m** einzufügen.
- Wir machen das diesmal ähnlich.

```

1  /* Create the tables for a S->|---|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid      INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid    INT NOT NULL,    -- later used to reference T via relate_s_and_t
10     x        CHAR(3)         -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid    INT NOT NULL,    -- later used to reference S via relate_s_and_t
17     y        CHAR(2)         -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid    INT NOT NULL REFERENCES s (sid),
23     fktid    INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);

```

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

S <--> T: Lösung

- Wir benutzen diese Sequenz um zuerst immer einen Primärschlüsselwert für Tabelle `m` zu erzeugen.
- Dann benutzen wir diesen Primärschlüsselwert als Fremdschlüsselwert für Tabelle `n`.
- Danach benutzen wir den Primärschlüsselwert dieser neuen Zeile zusammen mit dem für Tabelle `m` um letztendlich die Zeile in Tabelle `m` einzufügen.
- Wir machen das diesmal ähnlich.
- Wir führen also erstmal
`CREATE SEQUENCE sqsid AS INT;`
aus.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S <--> T: Lösung

- Dann benutzen wir diesen Primärschlüsselwert als Fremdschlüsselwert für Tabelle `n`.
- Danach benutzen wir den Primärschlüsselwert dieser neuen Zeile zusammen mit dem für Tabelle `m` um letztendlich die Zeile in Tabelle `m` einzufügen.
- Wir machen das diesmal ähnlich.
- Wir führen also erstmal `CREATE SEQUENCE sqsid AS INT;` aus.
- Der Primärschlüssel für Tabelle `s` wird dann definiert als `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY.`

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S <--> T: Lösung

- Danach benutzen wir den Primärschlüsselwert dieser neuen Zeile zusammen mit dem für Tabelle **m** um letztendlich die Zeile in Tabelle **m** einzufügen.
- Wir machen das diesmal ähnlich.
- Wir führen also erstmal `CREATE SEQUENCE sqsid AS INT;` aus.
- Der Primärschlüssel für Tabelle **s** wird dann definiert als `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY.`
- Beide Tabellen **s** und **t** brauchen eine Spalte um einen Primärschlüsselwert der jeweils anderen Tabelle zu referenzieren.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S ⇨ ⇨ ⇨ T: Lösung

- Wir machen das diesmal ähnlich.
- Wir führen also erstmal
`CREATE SEQUENCE sqsid AS INT;`
aus.
- Der Primärschlüssel für Tabelle `s` wird dann definiert als `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY`.
- Beide Tabellen `s` und `t` brauchen eine Spalte um einen Primärschlüsselwert der jeweils anderen Tabelle zu referenzieren.
- Für Tabelle `s`, ist das die Spalte `fktid`.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid  INT NOT NULL REFERENCES s (sid),
23     fktid  INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S ↔ T: Lösung

- Wir führen also erstmal

```
CREATE SEQUENCE sqsid AS INT;
```

aus.

- Der Primärschlüssel für Tabelle **s** wird dann definiert als `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY`.
- Beide Tabellen **s** und **t** brauchen eine Spalte um einen Primärschlüsselwert der jeweils anderen Tabelle zu referenzieren.
- Für Tabelle **s**, ist das die Spalte `fktid`.
- Für Tabelle **t**, ist das die Spalte `fksid`.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL,    -- later used to reference T via relate_s_and_t
10     x      CHAR(3)          -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL,    -- later used to reference S via relate_s_and_t
17     y      CHAR(2)          -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid  INT NOT NULL REFERENCES s (sid),
23     fktid  INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S <--> T: Lösung

- Der Primärschlüssel für Tabelle **s** wird dann definiert als `sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY`.
- Beide Tabellen **s** und **t** brauchen eine Spalte um einen Primärschlüsselwert der jeweils anderen Tabelle zu referenzieren.
- Für Tabelle **s**, ist das die Spalte `fktid`.
- Für Tabelle **t**, ist das die Spalte `fksid`.
- Diese sind dazu da, um sicherzustellen, dass jede Zeile in Tabelle **s** definitiv mit mindestens einer Zeile in Tabelle **t** verbunden ist.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid  INT NOT NULL REFERENCES s (sid),
23     fktid  INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S <--> T: Lösung

- Beide Tabellen **s** und **t** brauchen eine Spalte um einen Primärschlüsselwert der jeweils anderen Tabelle zu referenzieren.
- Für Tabelle **s**, ist das die Spalte **fktid**.
- Für Tabelle **t**, ist das die Spalte **fksid**.
- Diese sind dazu da, um sicherzustellen, dass jede Zeile in Tabelle **s** definitiv mit mindestens einer Zeile in Tabelle **t** verbunden ist.
- Und umgekehrt.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\rightrightarrows$ T: Lösung

- Für Tabelle `s`, ist das die Spalte `fktid`.
- Für Tabelle `t`, ist das die Spalte `fksid`.
- Diese sind dazu da, um sicherzustellen, dass jede Zeile in Tabelle `s` definitiv mit mindestens einer Zeile in Tabelle `t` verbunden ist.
- Und umgekehrt.
- Natürlich können die Zeilen mit mehreren Zeilen in Verbindung stehen.

```

1  /* Create the tables for a S->T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid      INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid    INT NOT NULL,    -- later used to reference T via relate_s_and_t
10     x        CHAR(3)          -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid    INT NOT NULL,    -- later used to reference S via relate_s_and_t
17     y        CHAR(2)          -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid    INT NOT NULL REFERENCES s (sid),
23     fktid    INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);

```

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

S ⇨ ⇨ ⇨ T: Lösung

- Für Tabelle **t**, ist das die Spalte **fksid**.
- Diese sind dazu da, um sicherzustellen, dass jede Zeile in Tabelle **s** definitiv mit mindestens einer Zeile in Tabelle **t** verbunden ist.
- Und umgekehrt.
- Natürlich können die Zeilen mit mehreren Zeilen in Verbindung stehen.
- Dafür haben wir ja Tabelle **relate_s_and_t**.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\rightrightarrows$ T: Lösung

- Diese sind dazu da, um sicherzustellen, dass jede Zeile in Tabelle `s` definitiv mit mindestens einer Zeile in Tabelle `t` verbunden ist.
- Und umgekehrt.
- Natürlich können die Zeilen mit mehreren Zeilen in Verbindung stehen.
- Dafür haben wir ja Tabelle `relate_s_and_t`.
- Jetzt müssen wir sicherstellen, dass es für jede Zeile in Tabelle `s` mit den Werten `(sid, fktid)` eine entsprechende Zeile `(fksid, fktid)` in Tabelle `relate_s_and_t` gibt.

```

1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid      INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x        CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y        CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);

```

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

S <--> T: Lösung

- Und umgekehrt.
- Natürlich können die Zeilen mit mehreren Zeilen in Verbindung stehen.
- Dafür haben wir ja Tabelle `relate_s_and_t`.
- Jetzt müssen wir sicherstellen, dass es für jede Zeile in Tabelle `s` mit den Werten `(sid, fktid)` eine entsprechende Zeile `(fksid, fktid)` in Tabelle `relate_s_and_t` gibt.
- Das geht mit der Einschränkung `s_sid_fktid_fk`, die wir in Tabelle `s` als `FOREIGN KEY (sid, fktid) REFERENCES relate_s_and_t (fksid, fktid)` macht.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid    INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid  INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x      CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid    INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid  INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y      CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30     REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32     REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\bowtie$ T: Lösung

- Dafür haben wir ja Tabelle `relate_s_and_t`.
- Jetzt müssen wir sicherstellen, dass es für jede Zeile in Tabelle `s` mit den Werten `(sid, fktid)` eine entsprechende Zeile `(fksid, fktid)` in Tabelle `relate_s_and_t` gibt.
- Das geht mit der Einschränkung `s_sid_fktid_fk`, die wir in Tabelle `s` als `FOREIGN KEY (sid, fktid) REFERENCES relate_s_and_t (fksid, fktid)` macht.
- Das ist der gleiche Ansatz, den wir damals für `Q  $\bowtie$  R` verwendet haben.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S $\bowtie$ T: Lösung

- Das geht mit der Einschränkung `s_sid_fktid_fk`, die wir in Tabelle `s` als `FOREIGN KEY (sid, fktid)` definieren, der `REFERENCES relate_s_and_t (fkssid, fktid)` macht.
- Das ist der gleiche Ansatz, den wir damals für `Q  $\bowtie$  R` verwendet haben.
- Der Unterschied ist, dass wir das diesmal auch genauso für Tabelle `t` machen müssen, denn beide Beziehungsenden sind zwingend.

```
1  /* Create the tables for a S->|----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkssid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fkssid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fkssid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fkssid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fkssid_tid_fk FOREIGN KEY (fkssid, tid)
32 REFERENCES relate_s_and_t (fkssid, fktid);
```

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

S $\bowtie$ T: Lösung

- Das geht mit der Einschränkung `s_sid_fktid_fk`, die wir in Tabelle `s` als `FOREIGN KEY (sid, fktid) REFERENCES relate_s_and_t (fkfid, fktid)` macht.
- Das ist der gleiche Ansatz, den wir damals für `Q  $\bowtie$  R` verwendet haben.
- Der Unterschied ist, dass wir das diesmal auch genauso für Tabelle `t` machen müssen, denn beide Beziehungsenden sind zwingend.
- Mit anderen Worten, wir fügen die Einschränkung `t_fkfid_tid_fk` zur Tabelle `t` hinzu.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x CHAR(3) -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fkfid INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y CHAR(2) -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fkfid INT NOT NULL REFERENCES s (sid),
23     fktid INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fkfid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fkfid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fkfid_tid_fk FOREIGN KEY (fkfid, tid)
32 REFERENCES relate_s_and_t (fkfid, fktid);
```

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

S $\rightarrow$ | $\leftarrow$ T: Lösung

- Das ist der gleiche Ansatz, den wir damals für $Q \rightarrow \bigcirc \leftarrow R$ verwendet haben.
- Der Unterschied ist, dass wir das diesmal auch genauso für Tabelle `t` machen müssen, denn beide Beziehungsenden sind zwingend.
- Mit anderen Worten, wir fügen die Einschränkung `t_fksid_tid_fk` zur Tabelle `t` hinzu.
- Diese stellt sicher, dass es für jedes Tupel `(fksid, tid)` in dieser Tabelle auch ein passendes Tupel `(fksid, fktid)` in `relate_s_and_t` gibt.

```
1  /* Create the tables for a S->|-----|<-T relationship. */
2
3  -- The sequence for the primary keys of the rows in s.
4  CREATE SEQUENCE sqsid AS INT;
5
6  -- Table S: Each row in S is related to one or multiple rows in T.
7  CREATE TABLE s (
8      sid      INT DEFAULT NEXTVAL('sqsid') PRIMARY KEY,
9      fktid    INT NOT NULL, -- later used to reference T via relate_s_and_t
10     x        CHAR(3)      -- example for other attributes
11 );
12
13 -- Table T: Each row in T is related to one or multiple rows in S.
14 CREATE TABLE t (
15     tid      INT GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
16     fksid    INT NOT NULL, -- later used to reference S via relate_s_and_t
17     y        CHAR(2)      -- example for other attributes
18 );
19
20 -- The table for managing the relationship between S and T.
21 CREATE TABLE relate_s_and_t (
22     fksid    INT NOT NULL REFERENCES s (sid),
23     fktid    INT NOT NULL REFERENCES t (tid),
24     PRIMARY KEY (fksid, fktid) -- Primary key includes both columns.
25 );
26
27 -- To tables S and T, we add foreign key constraints that enforce that
28 -- the corresponding rows in relate_s_and_t exist.
29 ALTER TABLE s ADD CONSTRAINT s_sid_fktid_fk FOREIGN KEY (sid, fktid)
30 REFERENCES relate_s_and_t (fksid, fktid);
31 ALTER TABLE t ADD CONSTRAINT t_fksid_tid_fk FOREIGN KEY (fksid, tid)
32 REFERENCES relate_s_and_t (fksid, fktid);
```

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

S T: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Anfänglich sind beide Tabellen leer.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S T: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Anfänglich sind beide Tabellen leer.
- Wir müssen also **drei** Zeilen auf einmal erstellen.

```
1  /* Insert into the tables for the S->1-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Anfänglich sind beide Tabellen leer.
- Wir müssen also **drei** Zeilen auf einmal erstellen.
- Wir müssen eine Zeile in Tabelle **S** erstellen.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Fügen wir nun Daten in die Tabellen ein.
- Anfänglich sind beide Tabellen leer.
- Wir müssen also **drei** Zeilen auf einmal erstellen.
- Wir müssen eine Zeile in Tabelle **S** erstellen.
- Wir müssen diese sofort mit einer neuen Zeile in Tabelle **T** verbinden.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Anfänglich sind beide Tabellen leer.
- Wir müssen also **drei** Zeilen auf einmal erstellen.
- Wir müssen eine Zeile in Tabelle **s** erstellen.
- Wir müssen diese sofort mit einer neuen Zeile in Tabelle **t** verbinden.
- Und diese Beziehung muss auch gleich als neue Zeile in Tabelle **relate_s_and_t** auftauchen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15        INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21        INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27        INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S <--> T: Befüllen mit Daten

- Wir müssen also **drei** Zeilen auf einmal erstellen.
- Wir müssen eine Zeile in Tabelle **s** erstellen.
- Wir müssen diese sofort mit einer neuen Zeile in Tabelle **t** verbinden.
- *Und* diese Beziehung muss auch gleich als neue Zeile in Tabelle **relate_s_and_t** auftauchen.
- Dank CTEs geht das.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir müssen eine Zeile in Tabelle `s` erstellen.
- Wir müssen diese sofort mit einer neuen Zeile in Tabelle `t` verbinden.
- Und diese Beziehung muss auch gleich als neue Zeile in Tabelle `relate_s_and_t` auftauchen.
- Dank CTEs geht das.
- Zuerst erstellen wir einen Primärschlüsselwert für Tabelle `s` über den CTE `s_id`.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir müssen diese sofort mit einer neuen Zeile in Tabelle `t` verbinden.
- Und diese Beziehung muss auch gleich als neue Zeile in Tabelle `relate_s_and_t` auftauchen.
- Dank CTEs geht das.
- Zuerst erstellen wir einen Primärschlüsselwert für Tabelle `s` über den CTE `s_id`.
- Dieser CTE macht corresponding to `SELECT NEXTVAL('sqsid') AS new_sid`.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Und diese Beziehung muss auch gleich als neue Zeile in Tabelle `relate_s_and_t` auftauchen.
- Dank CTEs geht das.
- Zuerst erstellen wir einen Primärschlüsselwert für Tabelle `s` über den CTE `s_id`.
- Dieser CTE macht corresponding to `SELECT NEXTVAL('sqsid') AS new_sid`
- Wir benutzen diesen neuen Primärschlüsselwert um eine Zeile in Tabelle `t` einzufügen.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S <--> T: Befüllen mit Daten

- Dank CTEs geht das.
- Zuerst erstellen wir einen Primärschlüsselwert für Tabelle `s` über den CTE `s_id`.
- Dieser CTE macht corresponding to `SELECT NEXTVAL('sqsid') AS new_sid`.
- Wir benutzen diesen neuen Primärschlüsselwert um eine Zeile in Tabelle `t` einzufügen.
- Dafür machen wir `INSERT INTO t (y, fksid) SELECT 'AB', new_sid FROM s_id`.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Zuerst erstellen wir einen Primärschlüsselwert für Tabelle `s` über den CTE `s_id`.
- Dieser CTE macht corresponding to `SELECT NEXTVAL('sqsid') AS new_sid`.
- Wir benutzen diesen neuen Primärschlüsselwert um eine Zeile in Tabelle `t` einzufügen.
- Dafür machen wir `INSERT INTO t (y, fksid) SELECT 'AB', new_sid FROM s_id`.
- Natürlich wird das wieder ein CTE, den wir `new_t` nennen und der auch noch `RETURNING tid, fksid` macht.

```
1  /* Insert into the tables for the S->I-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir benutzen diesen neuen Primärschlüsselwert um eine Zeile in Tabelle `t` einzufügen.
- Dafür machen wir `INSERT INTO t (y, fksid) SELECT 'AB', new_sid FROM s_id`.
- Natürlich wird das wieder ein CTE, den wir `new_t` nennen und der auch noch `RETURNING tid, fksid` macht.
- Dieser CTE gibt uns den Primärschlüsselwert der neuen Zeile in `t` und den Primärschlüsselwert `fksid`, den wir für die neue Zeile in Table `s` allokiert haben.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S >|< T: Befüllen mit Daten

- Wir benutzen diesen neuen Primärschlüsselwert um eine Zeile in Tabelle `t` einzufügen.
- Dafür machen wir `INSERT INTO t (y, fksid) SELECT 'AB', new_sid FROM s_id`.
- Natürlich wird das wieder ein CTE, den wir `new_t` nennen und der auch noch `RETURNING tid, fksid` macht.
- Dieser CTE gibt uns den Primärschlüsselwert der neuen Zeile in `t` und den Primärschlüsselwert `fksid`, den wir für die neue Zeile in Table `s` allokiert haben.
- Und jetzt erstellen wir diese Zeile.

```
1  /* Insert into the tables for the S->|<----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

```
1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
2  ↪ ON_ERROR_STOP=1 -ebf ST_insert_and_select.sql
3
4  INSERT 0 1
5  INSERT 0 1
6  INSERT 0 1
7  INSERT 0 1
8  INSERT 0 1
9
10  sid | x   | tid | y
11  ----+---+----+---
12  1   | 123 | 1   | AB
13  2   | 456 | 1   | AB
14  2   | 456 | 2   | CD
15  2   | 456 | 3   | EF
16  1   | 123 | 3   | EF
17
18  (5 rows)
19
20 # psql 16.11 succeeded with exit code 0.
```



S $\bowtie$ T: Befüllen mit Daten

- Natürlich wird das wieder ein CTE, den wir `new_t` nennen und der auch noch `RETURNING tid, fksid` macht.
- Dieser CTE gibt uns den Primärschlüsselwert der neuen Zeile in `t` und den Primärschlüsselwert `fksid`, den wir für die neue Zeile in Table `s` allokiert haben.
- Und jetzt erstellen wir diese Zeile.
- Wir führen `INSERT INTO s (sid, x, fktid)`
`SELECT fksid, '123', tid`
`FROM new_t` aus.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                   SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                   SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16          SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22          SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28          SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S <--> T: Befüllen mit Daten

- Natürlich wird das wieder ein CTE, den wir `new_t` nennen und der auch noch `RETURNING tid, fksid` macht.
- Dieser CTE gibt uns den Primärschlüsselwert der neuen Zeile in `t` und den Primärschlüsselwert `fksid`, den wir für die neue Zeile in Table `s` allokiert haben.
- Und jetzt erstellen wir diese Zeile.
- Wir führen `INSERT INTO s (sid, x, fktid)`
`SELECT fksid, '123', tid`
`FROM new_t` aus.
- Wir benutzen also den vorher erstellten Primärschlüsselwert.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH new_s AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM new_s RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                  RETURNING sid, fktid)
15          INSERT INTO relate_s_and_t (fksid, fktid)
16             SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                  RETURNING tid, fksid)
21          INSERT INTO relate_s_and_t (fksid, fktid)
22             SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                  RETURNING tid, fksid)
27          INSERT INTO relate_s_and_t (fksid, fktid)
28             SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S >| <| T: Befüllen mit Daten

- Dieser CTE gibt uns den Primärschlüsselwert der neuen Zeile in `t` und den Primärschlüsselwert `fksid`, den wir für die neue Zeile in Table `s` allokiert haben.
- Und jetzt erstellen wir diese Zeile.
- Wir führen `INSERT INTO s (sid, x, fktid)`
`SELECT fksid, '123', tid`
`FROM new_t` aus.
- Wir benutzen also den vorher erstellten Primärschlüsselwert.
- Wir benutzen auch den Primärschlüsselwert der neuen Zeile in Tabelle `t` als Fremdschlüsselwert.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15          INSERT INTO relate_s_and_t (fksid, fktid)
16             SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21          INSERT INTO relate_s_and_t (fksid, fktid)
22             SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27          INSERT INTO relate_s_and_t (fksid, fktid)
28             SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S <--> T: Befüllen mit Daten

- Und jetzt erstellen wir diese Zeile.
- Wir führen `INSERT INTO s`
`(sid, x, fktid)`
`SELECT fksid, '123', tid`
`FROM new_t` aus.
- Wir benutzen also den vorher erstellten Primärschlüsselwert.
- Wir benutzen auch den Primärschlüsselwert der neuen Zeile in Tabelle `t` als Fremdschlüsselwert.
- Das ist dann unser dritter CTE, den wir `new_s` nennen.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15        INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21        INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27        INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir führen `INSERT INTO s (sid, x, fktid)`
`SELECT fksid, '123', tid`
`FROM new_t` aus.
- Wir benutzen also den vorher erstellten Primärschlüsselwert.
- Wir benutzen auch den Primärschlüsselwert der neuen Zeile in Tabelle `t` als Fremdschlüsselwert.
- Das ist dann unser dritter CTE, den wir `new_s` nennen.
- Er liefert beide Schlüsselwerte über `RETURNING sid, fktid` zurück.

```
1  /* Insert into the tables for the S->I-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                   SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                   SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir benutzen also den vorher erstellten Primärschlüsselwert.
- Wir benutzen auch den Primärschlüsselwert der neuen Zeile in Tabelle `t` als Fremdschlüsselwert.
- Das ist dann unser dritter CTE, den wir `new_s` nennen.
- Er liefert beide Schlüsselwerte über `RETURNING sid, fktid` zurück.
- Zuletzt können wir nun eine Zeile in `relate_s_and_t` mit `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` einfügen.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\rightarrow$ T: Befüllen mit Daten

- Wir benutzen auch den Primärschlüsselwert der neuen Zeile in Tabelle `t` als Fremdschlüsselwert.
- Das ist dann unser dritter CTE, den wir `new_s` nennen.
- Er liefert beide Schlüsselwerte über `RETURNING sid, fktid` zurück.
- Zuletzt können wir nun eine Zeile in `relate_s_and_t` mit `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` einfügen.
- Dank der CTEs konnten wir also jeweils eine Zeile in jede der drei Tabellen einfügen.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Das ist dann unser dritter CTE, den wir `new_s` nennen.
- Er liefert beide Schlüsselwerte über `RETURNING sid, fktid` zurück.
- Zuletzt können wir nun eine Zeile in `relate_s_and_t` mit `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` einfügen.
- Dank der CTEs konnten wir also jeweils eine Zeile in jede der drei Tabellen einfügen.
- All das war ein einziges SQL-Kommando.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Er liefert beide Schlüsselwerte über `RETURNING sid, fktid` zurück.
- Zuletzt können wir nun eine Zeile in `relate_s_and_t` mit `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` einfügen.
- Dank der CTEs konnten wir also jeweils eine Zeile in jede der drei Tabellen einfügen.
- All das war ein einziges SQL-Kommando.
- Die referentielle Integrität wird an dessen Ende geprüft (und ist OK).

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15        INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21        INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27        INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Zuletzt können wir nun eine Zeile in `relate_s_and_t` mit `INSERT INTO relate_s_and_t (fkssid, fktid)` `SELECT sid, fktid FROM new_s` einfügen.
- Dank der CTEs konnten wir also jeweils eine Zeile in jede der drei Tabellen einfügen.
- All das war ein einziges SQL-Kommando.
- Die referentielle Integrität wird an dessen Ende geprüft (und ist OK).
- Nun gibt es existierende Datensätze in den Tabellen `s` und `t`.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fkssid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fkssid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fkssid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fkssid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fkssid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fkssid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fkssid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fkssid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Dank der CTEs konnten wir also jeweils eine Zeile in jede der drei Tabellen einfügen.
- All das war ein einziges SQL-Kommando.
- Die referentielle Integrität wird an dessen Ende geprüft (und ist OK).
- Nun gibt es existierende Datensätze in den Tabellen **s** und **t**.
- Es ist dann einfacher, eine neue Zeile in Tabelle **s** zu erstellen, die mit einer existierenden Zeile in Tabelle **t** verbunden ist.

```
1  /* Insert into the tables for the S->T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S T: Befüllen mit Daten

- All das war ein einziges SQL-Kommando.
- Die referentielle Interitt wird an dessen Ende geprft (und ist OK).
- Nun gibt es existierende Datenstze in den Tabellen **s** und **t**.
- Es ist dann einfacher, eine neue Zeile in Tabelle **s** zu erstellen, die mit einer existierenden Zeile in Tabelle **t** verbunden ist.
- Alles was wir machen mssen ist eine Zeile in Tabelle **s** einzufgen und gleichzeitig eine Zeile in **relate_s_and_t**.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Die referentielle Integrität wird an dessen Ende geprüft (und ist OK).
- Nun gibt es existierende Datensätze in den Tabellen `s` und `t`.
- Es ist dann einfacher, eine neue Zeile in Tabelle `s` zu erstellen, die mit einer existierenden Zeile in Tabelle `t` verbunden ist.
- Alles was wir machen müssen ist eine Zeile in Tabelle `s` einzufügen und gleichzeitig eine Zeile in `relate_s_and_t`.
- Wir brauchen nur einen CTE, den wir `new_s` nennen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
37
38
39 1 $ psql "postgres://postgres:XXX@localhost/relationships" -v
40   ↪ ON_ERROR_STOP=1 -ebf ST_insert_and_select.sql
41
42 2 INSERT 0 1
43 3 INSERT 0 1
44 4 INSERT 0 1
45 5 INSERT 0 1
46 6 INSERT 0 1
47
48 7  sid | x  | tid | y
49 8  -----
50 9  1 | 123 | 1 | AB
51 10 2 | 456 | 1 | AB
52 11 2 | 456 | 2 | CD
53 12 2 | 456 | 3 | EF
54 13 1 | 123 | 3 | EF
55 14 (5 rows)
56
57 16 # psql 16.11 succeeded with exit code 0.
```



S >|< T: Befüllen mit Daten

- Nun gibt es existierende Datensätze in den Tabellen `s` und `t`.
- Es ist dann einfacher, eine neue Zeile in Tabelle `s` zu erstellen, die mit einer existierenden Zeile in Tabelle `t` verbunden ist.
- Alles was wir machen müssen ist eine Zeile in Tabelle `s` einzufügen und gleichzeitig eine Zeile in `relate_s_and_t`.
- Wir brauchen nur einen CTE, den wir `new_s` nennen.
- Der macht `INSERT INTO s (x, fktid) VALUES ('456', 1) RETURNING sid, fktid`.

```
1  /* Insert into the tables for the S->|< T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Alles was wir machen müssen ist eine Zeile in Tabelle `s` einzufügen und gleichzeitig eine Zeile in `relate_s_and_t`.
- Wir brauchen nur einen CTE, den wir `new_s` nennen.
- Der macht `INSERT INTO s (x, fktid) VALUES ('456', 1) RETURNING sid, fktid`.
- Als Ergebnis bekommen wir den Primärschlüsselwert `sid` der neuen Zeile in Tabelle `s` und den Fremdschlüsselwert `fktid` zur Zeile in Tabelle `t`.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir brauchen nur einen CTE, den wir `new_s` nennen.
- Der macht `INSERT INTO s (x, fktid) VALUES ('456', 1) RETURNING sid, fktid`.
- Als Ergebnis bekommen wir den Primärschlüsselwert `sid` der neuen Zeile in Tabelle `s` und den Fremdschlüsselwert `fktid` zur Zeile in Tabelle `t`.
- `fktid` ist der Wert, den wir angegeben haben, als wir die Zeile in Tabelle `s` erstellt haben – also `1`.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH new_s AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM new_s RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Der macht `INSERT INTO s`
`(x, fktid) VALUES ('456', 1)`
`RETURNING sid, fktid.`
- Als Ergebnis bekommen wir den Primärschlüsselwert `sid` der neuen Zeile in Tabelle `s` und den Fremdschlüsselwert `fktid` zur Zeile in Tabelle `t`.
- `fktid` ist der Wert, den wir angegeben haben, als wir die Zeile in Tabelle `s` erstellt haben – also 1.
- Damit können wir dann `INSERT INTO relate_s_and_t`
`(fksid, fktid)`
`SELECT sid, fktid FROM new_s` machen und sind fertig.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15          INSERT INTO relate_s_and_t (fksid, fktid)
16             SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21          INSERT INTO relate_s_and_t (fksid, fktid)
22             SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27          INSERT INTO relate_s_and_t (fksid, fktid)
28             SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Als Ergebnis bekommen wir den Primärschlüsselwert `sid` der neuen Zeile in Tabelle `s` und den Fremdschlüsselwert `fktid` zur Zeile in Tabelle `t`.
- `fktid` ist der Wert, den wir angegeben haben, als wir die Zeile in Tabelle `s` erstellt haben – also `1`.
- Damit können wir dann
`INSERT INTO relate_s_and_t (fksid, fktid)`
`SELECT sid, fktid FROM new_s`
machen und sind fertig.
- Da der Beziehungstyp symmetrisch ist, können wir genau das gleiche mit Tabelle `t` machen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16         SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22         SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28         SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
37
38
39 $ psql "postgres://postgres:XXX@localhost/relationships" -v
40 ↪ ON_ERROR_STOP=1 -ebf ST_insert_and_select.sql
41
42 INSERT 0 1
43 INSERT 0 1
44 INSERT 0 1
45 INSERT 0 1
46 INSERT 0 1
47
48  sid | x   | tid | y
49  ----+---+----+---
50   1 | 123 |  1 | AB
51   2 | 456 |  1 | AB
52   2 | 456 |  2 | CD
53   2 | 456 |  3 | EF
54   1 | 123 |  3 | EF
55 (5 rows)
56
57 # psql 16.11 succeeded with exit code 0.
```



S $\bowtie$ T: Befüllen mit Daten

- `fktid` ist der Wert, den wir angegeben haben, als wir die Zeile in Tabelle `s` erstellt haben – also 1.
- Damit können wir dann `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` machen und sind fertig.
- Da der Beziehungstyp symmetrisch ist, können wir genau das gleiche mit Tabelle `t` machen.
- Wir können eine Zeile in Tabelle `t` einfügen und diese auf eine existierende Zeile in Tabelle `s` zeigen lassen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                    SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                    SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                  RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16          SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                  RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22          SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                  RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28          SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\times$ T: Befüllen mit Daten

- `fktid` ist der Wert, den wir angegeben haben, als wir die Zeile in Tabelle `s` erstellt haben – also 1.
- Damit können wir dann `INSERT INTO relate_s_and_t (fksid, fktid)` `SELECT sid, fktid FROM new_s` machen und sind fertig.
- Da der Beziehungstyp symmetrisch ist, können wir genau das gleiche mit Tabelle `t` machen.
- Wir können eine Zeile in Tabelle `t` einfügen und diese auf eine existierende Zeile in Tabelle `s` zeigen lassen.
- Wir würden genauso vorgehen und einen CTE benutzen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH new_s AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                SELECT 'AB', new_sid FROM new_s RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15        INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21        INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27        INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Damit können wir dann

```
INSERT INTO relate_s_and_t  
(fksid, fktid)
```

```
SELECT sid, fktid FROM new_s
```

machen und sind fertig.

- Da der Beziehungstyp symmetrisch ist, können wir genau das gleiche mit Tabelle **t** machen.

- Wir können eine Zeile in Tabelle **t** einfügen und diese auf eine existierende Zeile in Tabelle **s** zeigen lassen.

- Wir würden genauso vorgehen und einen CTE benutzen.

- Zu guter Letzt können wir auch einfach zwei existierende Zeilen der Tabellen **s** und **t** in Beziehung setzen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10          SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                  RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16          SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                  RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22          SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                  RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28          SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35       INNER JOIN s ON s.sid = relate_s_and_t.fksid
36       INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Da der Beziehungstyp symmetrisch ist, können wir genau das gleiche mit Tabelle **t** machen.
- Wir können eine Zeile in Tabelle **t** einfügen und diese auf eine existierende Zeile in Tabelle **s** zeigen lassen.
- Wir würden genauso vorgehen und einen CTE benutzen.
- Zu guter Letzt können wir auch einfach zwei existierende Zeilen der Tabellen **s** und **t** in Beziehung setzen.
- Dafür müssen wir nur eine neue Zeile in Tabelle **relate_s_and_t** erstellen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                RETURNING sid, fktid)
15        INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                RETURNING tid, fksid)
21        INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                RETURNING tid, fksid)
27        INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir können eine Zeile in Tabelle `t` einfügen und diese auf eine existierende Zeile in Tabelle `s` zeigen lassen.
- Wir würden genauso vorgehen und einen CTE benutzen.
- Zu guter Letzt können wir auch einfach zwei existierende Zeilen der Tabellen `s` und `t` in Beziehung setzen.
- Dafür müssen wir nur eine neue Zeile in Tabelle `relate_s_and_t` erstellen.
- Das geht in etwa so:
`INSERT INTO relate_s_and_t`
`VALUES (1, 3);`

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                   SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                   SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10        SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15       INSERT INTO relate_s_and_t (fksid, fktid)
16        SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21       INSERT INTO relate_s_and_t (fksid, fktid)
22        SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27       INSERT INTO relate_s_and_t (fksid, fktid)
28        SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Befüllen mit Daten

- Wir würden genauso vorgehen und einen CTE benutzen.
- Zu guter Letzt können wir auch einfach zwei existierende Zeilen der Tabellen `s` und `t` in Beziehung setzen.
- Dafür müssen wir nur eine neue Zeile in Tabelle `relate_s_and_t` erstellen.
- Das geht in etwa so:
`INSERT INTO relate_s_and_t`
`VALUES (1, 3);`
- Die Daten können wir wieder über zwei `INNER JOIN`s zusammenführen.

```
1  /* Insert into the tables for the S->|-----|<-T relationship. */
2
3  -- Create a pair of new and related S and T entities.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'AB', new_sid FROM s_id RETURNING tid, fksid),
7       new_s AS (INSERT INTO s (sid, x, fktid)
8                  SELECT fksid, '123', tid FROM new_t RETURNING sid, fktid)
9       INSERT INTO relate_s_and_t (fksid, fktid)
10         SELECT sid, fktid FROM new_s;
11
12 -- Create a new S record and relate it to an existing T entity.
13 WITH new_s AS (INSERT INTO s (x, fktid) VALUES ('456', 1)
14                 RETURNING sid, fktid)
15         INSERT INTO relate_s_and_t (fksid, fktid)
16           SELECT sid, fktid FROM new_s;
17
18 -- Create a new T entity and relate it to an existing S entity.
19 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('CD', 2)
20                 RETURNING tid, fksid)
21         INSERT INTO relate_s_and_t (fksid, fktid)
22           SELECT fksid, tid FROM new_t;
23
24 -- Create a new T entity and relate it to an existing S entity.
25 WITH new_t AS (INSERT INTO t (y, fksid) VALUES ('EF', 2)
26                 RETURNING tid, fksid)
27         INSERT INTO relate_s_and_t (fksid, fktid)
28           SELECT fksid, tid FROM new_t;
29
30 -- We can now insert additional relationships.
31 INSERT INTO relate_s_and_t VALUES (1, 3);
32
33 -- Combine the rows from S and T. This needs two INNER JOINS.
34 SELECT sid, x, tid, y FROM relate_s_and_t
35        INNER JOIN s ON s.sid = relate_s_and_t.fksid
36        INNER JOIN t ON t.tid = relate_s_and_t.fktid;
```

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



S $\bowtie$ T: Der Inhalt der Drei Tabellen



- Dies sind die Daten in den drei Tabellen.

Table **s**

sid	fktid	x
1	1	„123“
2	1	„456“

Table **t**

tid	fksid	y
1	1	„AB“
2	2	„CD“
3	2	„EF“

Table **relate_s_and_t**

fksid	fktid
1	1
2	1
2	2
2	3
1	3

$S \succcurlyeq \mid \mid \mid \preccurlyeq T$: Testen der Einschränkungen



- Es ist unmöglich, Zeilen in Tabelle s (oder t) einzufügen, die nicht mit Zeilen in t (oder s) verbunden sind.



S $\bowtie$ T: Testen der Einschränkungen



- Es ist unmöglich, Zeilen in Tabelle **s** (oder **t**) einzufügen, die nicht mit Zeilen in **t** (oder **s**) verbunden sind.
- Wir müssen auch die Tabelle **relate_s_and_t** immer mit updaten.

```
1  /* Trying to insert rows S and T without using the relationship table */
2
3  -- Create a pair of new related S and T entities, ignore relate_s_and_t.
4  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
5       new_t AS (INSERT INTO t (y, fksid)
6                  SELECT 'GH', new_sid FROM s_id RETURNING tid, fksid)
7  INSERT INTO s (sid, x, fktid) SELECT fksid, '555', tid FROM new_t;

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf ST_insert_error_1.sql
2  psql:conceptualToRelational/ST_insert_error_1.sql:7: ERROR:  insert or
   ↳ update on table "t" violates foreign key constraint "
   ↳ t_fksid_tid_fk"
3  DETAIL:  Key (fksid, tid)=(3, 4) is not present in table "relate_s_and_t"
   ↳ ".
4  psql:conceptualToRelational/ST_insert_error_1.sql:7: STATEMENT:  /*
   ↳ Trying to insert rows S and T without using the relationship table
   ↳ */
5  -- Create a pair of new related S and T entities, ignore relate_s_and_t
6  WITH s_id AS (SELECT NEXTVAL('sqsid') AS new_sid),
7       new_t AS (INSERT INTO t (y, fksid)
8                  SELECT 'GH', new_sid FROM s_id RETURNING tid, fksid)
9  INSERT INTO s (sid, x, fktid) SELECT fksid, '555', tid FROM new_t;
10 # psql 16.11 failed with exit code 3.
```

S $\bowtie$ T: Testen der Einschränkungen



- Es ist unmöglich, Zeilen in Tabelle **s** (oder **t**) einzufügen, die nicht mit Zeilen in **t** (oder **s**) verbunden sind.
- Wir müssen auch die Tabelle **relate_s_and_t** immer mit updaten.

```
1  /* Trying to insert rows S and T without using the relationship table */
2
3  -- Create relate a new S entity to an existing T entity without updating
4  -- relate_s_and_t.
5  INSERT INTO s (fktid, x) VALUES (3, '777');

1  $ psql "postgres://postgres:XXX@localhost/relationships" -v
   ↳ ON_ERROR_STOP=1 -ebf ST_insert_error_2.sql
2  psql:conceptualToRelational/ST_insert_error_2.sql:5: ERROR:  insert or
   ↳ update on table "s" violates foreign key constraint "
   ↳ s_sid_fktid_fk"
3  DETAIL:  Key (sid, fktid)=(4, 3) is not present in table "relate_s_and_t"
   ↳ ".
4  psql:conceptualToRelational/ST_insert_error_2.sql:5: STATEMENT:  /*
   ↳ Trying to insert rows S and T without using the relationship table
   ↳ */
5  -- Create relate a new S entity to an existing T entity without updating
6  -- relate_s_and_t.
7  INSERT INTO s (fktid, x) VALUES (3, '777');
8  # psql 16.11 failed with exit code 3.
```

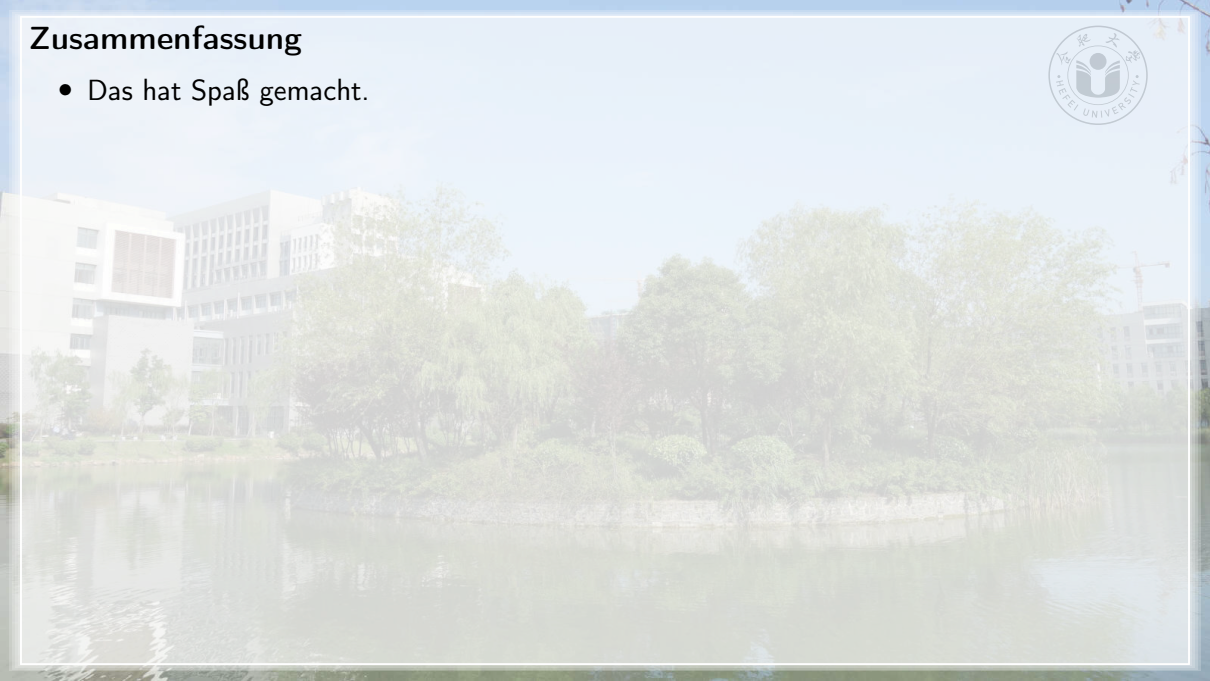


Zusammenfassung



Zusammenfassung

- Das hat Spaß gemacht.



Zusammenfassung

- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.



Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.

Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.

Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{⋈} \text{ } \text{---} \text{ } \text{O} \text{ } \text{<} \text{ } L$ implementieren wollen, dann können wir das tun.

Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{⋈} \text{ } \text{—} \text{ } \text{○} \text{ } \text{⋈} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle **k** für die Entitäten vom Type K und eine Tabelle **l** für die Entitäten vom Typ L.

Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{⧻} \text{ } \text{⊗} \text{ } \text{⋈} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle **k** für die Entitäten vom Type K und eine Tabelle **l** für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** verbunden ist.

Zusammenfassung



- Das hat Spaß gemacht.
- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{⧻} \text{ } \text{⧻} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle **k** für die Entitäten vom Type K und eine Tabelle **l** für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle **l** gibt, die nicht mit genau einer Zeile in Tabelle **k** verbunden ist.

Zusammenfassung



- Wir haben uns jedes einzelne mögliche binäre Beziehungsschema angeschaut, dass mit der Krähenfußnotation ausgedrückt werden kann.
- Wir haben herausgefunden, dass jedes davon wirklich im relationalen Datenmodell implementiert werden kann.
- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{HAW} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle **k** für die Entitäten vom Type K und eine Tabelle **l** für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen dass jede Zeile in Tabelle **l** mit genau einer Zeile in Tabelle **k** verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle **l** gibt, die nicht mit genau einer Zeile in Tabelle **k** verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle **k** mit beliebig vielen Zeilen in Tabelle **l** verbunden sein können.

清华大学
Tsinghua University

- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \overline{\parallel} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle k für die Entitäten vom Type K und eine Tabelle l für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen dass jede Zeile in Tabelle l mit genau einer Zeile in Tabelle k verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle l gibt, die nicht mit genau einer Zeile in Tabelle k verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle k mit beliebig vielen Zeilen in Tabelle l verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle l mit einer Zeile in Tabelle k verbunden ist, dass dann auch die entsprechende Zeile in Tabelle k mit einer Zeile in Tabelle l verbunden ist und umgekehrt.

清华大学
Tsinghua University

- Und mit *implemented*, wir meinen sowohl die Modalität als auch die Kardinalität der Beziehungsenden.
- Wenn wir z. B. die Beziehungsform $K \text{ } \text{---} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle k für die Entitäten vom Type K und eine Tabelle l für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen dass jede Zeile in Tabelle l mit genau einer Zeile in Tabelle k verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle l gibt, die nicht mit genau einer Zeile in Tabelle k verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle k mit beliebig vielen Zeilen in Tabelle l verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle l mit einer Zeile in Tabelle k verbunden ist, dass dann auch die entsprechende Zeile in Tabelle k mit einer Zeile in Tabelle l verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.

nen wir a

- Wenn wir z. B. die Beziehungsform $K \text{ } \text{---} \text{ } L$ implementieren wollen, dann können wir das tun.
- Wir können eine Tabelle k für die Entitäten vom Typ K und eine Tabelle l für die Entitäten vom Typ L .
- Dann können wir Einschränkungen erstellen, die erzwingen, dass jede Zeile in Tabelle l mit genau einer Zeile in Tabelle k verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle l gibt, die nicht mit genau einer Zeile in Tabelle k verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle k mit beliebig vielen Zeilen in Tabelle l verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle l mit einer Zeile in Tabelle k verbunden ist, dass dann auch die entsprechende Zeile in Tabelle k mit einer Zeile in Tabelle l verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.

Zusammenfassung



- Wir können eine Tabelle k für die Entitäten vom Type K und eine Tabelle l für die Entitäten vom Typ L.
- Dann können wir Einschränkungen erstellen, die erzwingen das jede Zeile in Tabelle l mit genau einer Zeile in Tabelle k verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle l gibt, die nicht mit genau einer Zeile in Tabelle k verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle k mit beliebig vielen Zeilen in Tabelle l verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle l mit einer Zeile in Tabelle k verbunden ist, dass dann auch die entsprechende Zeile in Tabelle k mit einer Zeile in Tabelle l verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.

Zusammenfassung



- Dann können wir Einschränkungen erstellen, die erzwingen das jede Zeile in Tabelle **1** mit genau einer Zeile in Tabelle **k** verbunden ist.
- Wir können erzwingen, dass es niemals eine Zeile in Tabelle **1** gibt, die nicht mit genau einer Zeile in Tabelle **k** verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle **k** mit beliebig vielen Zeilen in Tabelle **1** verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle **1** mit einer Zeile in Tabelle **k** verbunden ist, dass dann auch die entsprechende Zeile in Tabelle **k** mit einer Zeile in Tabelle **1** verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.

Zusammenfassung



- Wir können erzwingen, dass es niemals eine Zeile in Tabelle `l` gibt, die nicht mit genau einer Zeile in Tabelle `k` verbunden ist.
- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle `k` mit beliebig vielen Zeilen in Tabelle `l` verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle `l` mit einer Zeile in Tabelle `k` verbunden ist, dass dann auch die entsprechende Zeile in Tabelle `k` mit einer Zeile in Tabelle `l` verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).

Zusammenfassung



- Gleichzeitig können wir sicherstellen, dass die Zeilen in Tabelle `k` mit beliebig vielen Zeilen in Tabelle `l` verbunden sein können.
- Wir erzwingen auch, dass wenn eine Zeile in Tabelle `l` mit einer Zeile in Tabelle `k` verbunden ist, dass dann auch die entsprechende Zeile in Tabelle `k` mit einer Zeile in Tabelle `l` verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderes solches Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.

Zusammenfassung



- Wir erzwingen auch, dass wenn eine Zeile in Tabelle `l` mit einer Zeile in Tabelle `k` verbunden ist, dass dann auch die entsprechende Zeile in Tabelle `k` mit einer Zeile in Tabelle `l` verbunden ist und umgekehrt.
- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderer solches Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.
- Es ist also oftmals schwer.

Zusammenfassung



- Wenn wir diese Bedingungen erfüllen, dann hat unsere Datenbank *referentielle Integrität*.
- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderer solches Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.
- Es ist also oftmals schwer.
- Und wir haben immer nur zwei Entitätstypen angeguckt.

- Wir haben auch gelernt, dass das nicht immer einfach ist.
- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderer solcher Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.
- Es ist also oftmals schwer.
- Und wir haben immer nur zwei Entitätstypen angeguckt.
- Stellen Sie sich vor, wir haben die Beziehung $M \text{ } \begin{array}{c} \text{+} \\ \text{---} \\ \text{+} \end{array} \text{ } \begin{array}{c} \text{+} \\ \text{---} \\ \text{+} \end{array} \text{ } N$ und gleichzeitig $N \text{ } \begin{array}{c} \text{+} \\ \text{---} \\ \text{+} \end{array} \text{ } \begin{array}{c} \text{+} \\ \text{---} \\ \text{+} \end{array} \text{ } X$...

Zusammenfassung



- Besonders wenn Beziehungsenden zwingend sind und dann besonders wenn beide Enden zwingend sind ... dann wird es schwer.
- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderer solches Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.
- Es ist also oftmals schwer.
- Und wir haben immer nur zwei Entitätstypen angeguckt.
- Stellen Sie sich vor, wir haben die Beziehung $M \text{ } \overline{\text{H}} \text{ } \text{H} \text{ } \text{N}$ und gleichzeitig $N \text{ } \text{H} \text{ } \text{H} \text{ } \text{X}$...
- Deshalb ändert man manchmal die Beziehungstypen eben doch, wenn man vom konzeptuellen zum logischen Schema übergeht.

- Wir haben es mit zusätzlichen SQL-Kommandos wie CTEs hinbekommen.
- Manchmal brauchten wir PostgreSQL-spezifische Werkzeuge, wie `RETURNING`⁵⁹ (das aber auch bei MariaDB⁴² und SQLite⁶⁰ funktioniert).
- Ein anderer solches Werkzeug waren Sequenzen und die `NEXTVAL`-Funktion^{23,62}.
- Es ist also oftmals schwer.
- Und wir haben immer nur zwei Entitätstypen angeguckt.
- Stellen Sie sich vor, wir haben die Beziehung $M \text{ } \overline{\text{---}} \text{ } \overline{\text{---}} \text{ } N$ und gleichzeitig $N \text{ } \overline{\text{---}} \text{ } \overline{\text{---}} \text{ } X$...
- Deshalb ändert man manchmal die Beziehungstypen eben doch, wenn man vom konzeptuellen zum logischen Schema übergeht.
- Oder man nimmt komplexere Werkzeuge, wie Stored Procedures und Transaktionen (über die wir hier nicht sprechen).

Aufräumen



- Das hat trotzdem alles viel Spaß gemacht.

```
1  /* Drop the database for our examples */  
2  
3  -- If the database 'relationships' exists, we delete it.  
4  DROP DATABASE IF EXISTS relationships;
```

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

Aufräumen



- Das hat trotzdem alles viel Spaß gemacht.
- Räumen wir nun auf.

```
1  /* Drop the database for our examples */
2
3  -- If the database 'relationships' exists, we delete it.
4  DROP DATABASE IF EXISTS relationships;
```

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



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. 94–107).
- [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. 688).
- [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 also⁴ (siehe S. 678, 688).
- [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 also³ (siehe S. 678, 688).
- [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. 687).
- [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. 688, 689).
- [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. 688).
- [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. 687).
- [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. 689).
- [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. 687).
- [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. 688).

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. 688).
- [13] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 687).
- [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. 687).
- [15] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 688, 689).
- [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. 689).
- [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 (besucht am 2025-03-06) (siehe S. 687).
- [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. 679, 687).
- [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. See¹⁸ for a more comprehensive introduction. (Siehe S. 687).
- [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. 689).

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. 688).
- [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. 28–35).
- [23] "[CREATE SEQUENCE](#)". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. URL: <https://www.postgresql.org/docs/current/sql-createsequence.html> (besucht am 2025-04-22) (siehe S. 402–430, 654–674).
- [24] "[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. 276).
- [25] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 689).
- [26] 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. 689).
- [27] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 689).
- [28] "Default Values". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.2. URL: <https://www.postgresql.org/docs/current/ddl-default.html> (besucht am 2025-04-23) (siehe S. 402–436).
- [29] 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. 689).
- [30] 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. 688).
- [31] 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. 687, 688).

References IV



- [32] 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. 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. 689).
- [33] "Generated Columns". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.4. URL: <https://www.postgresql.org/docs/17/ddl-generated-columns.html> (besucht am 2025-04-23) (siehe S. 402–425).
- [34] 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. 688).
- [35] 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. 688).
- [36] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 688).
- [37] 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. 688, 689).
- [38] 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. 689).
- [39] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 688).
- [40] "Identity Columns". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 5.3. URL: <https://www.postgresql.org/docs/17/ddl-identity-columns.html> (besucht am 2025-02-27) (siehe S. 402–428).

References V



- [41] *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) and 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. 689).
- [42] “**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. 277–294, 654–674).
- [43] “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. 28–35).
- [44] 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. 687).
- [45] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 689).
- [46] 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. 688).
- [47] 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. 687).
- [48] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 688).
- [49] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (besucht am 2025-04-24) (siehe S. 688).

References VI



- [50] 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. **689**).
- [51] 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. **687**).
- [52] 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. **688**).
- [53] 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. **687**).
- [54] *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).
- [55] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. **688**).
- [56] 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. **687**).
- [57] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: **978-1-78398-154-0** (siehe S. **688**).
- [58] 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. **688**).
- [59] "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. **277–293**, **654–674**).

References VII



- [60] “RETURNING”. In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 8. Mai 2024. URL: https://sqlite.org/lang_returning.html (besucht am 2025-04-24) (siehe S. 277–294, 654–674).
- [61] 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. 687).
- [62] “Sequence Manipulation Functions”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.17. URL: <https://www.postgresql.org/docs/17/functions-sequence.html> (besucht am 2025-04-23) (siehe S. 402–433, 654–674).
- [63] Yuriy Shamshin. “Conceptual Database Model. Entity Relationship Diagram (ERD)”. In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 04. URL: https://dbs.academy.lv/lection/dbs_LS04EN_erd.pdf (besucht am 2025-03-29) (siehe S. 687).
- [64] Yuriy Shamshin. *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. URL: <https://dbs.academy.lv> (besucht am 2025-01-11).
- [65] Yuriy Shamshin. “Mapping ER Diagrams to Relation Data Model”. In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 06. URL: https://dbs.academy.lv/lection/dbs_LS06EN_er2rm.pdf (besucht am 2025-04-20) (siehe S. 63–67, 177–182, 337–341).
- [66] 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. 688).
- [67] 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. 688).
- [68] *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2025. URL: <https://sqlite.org> (besucht am 2025-04-24) (siehe S. 689).
- [69] “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. 689).

References VIII



- [70] 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. 685, 689).
- [71] 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. Burghann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of⁷⁰ (siehe S. 689).
- [72] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 688, 689).
- [73] 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. 688).
- [74] 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. 688).
- [75] "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. 277–282).
- [76] "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. 109–113).
- [77] "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. 277–293).
- [78] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 688).

References IX



- [79] 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. 687, 688).
- [80] 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. 688).
- [81] 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. 687).
- [82] *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. 688).
- [83] 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. 688).
- [84] 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. 689).
- [85] "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. 687).
- [86] 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. 687).
- [87] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 687).

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^{13,51,87}. 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^{10,47,53,56,61}. 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^{8,31,85}.

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*⁷⁹.

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 DB⁸⁶.

ERD Entity relationship diagrams show the relationships between objects, e.g., between the tables in a DB and how they reference each other^{5,14,17–19,44,63,81}

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^{15,37}.
- Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows^{6,36,66,74,78}. 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^{3,4,7,30,49,57}. See <https://mariadb.org> for more information.
- Microsoft Windows** is a commercial proprietary operating system¹². 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^{11,30,58,73,83}. 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 notation². Learn more at <https://pgmodeler.io>.
- PostgreSQL** An open source object-relational DBMS^{31,52,55,73}. See <https://postgresql.org> for more information.
- psql** is the client program used to access the PostgreSQL DBMS server.
- Python** The Python programming language^{39,46,48,80}, i.e., what you will learn about in our book⁸⁰. 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^{21,34,35,67,72,79,82}.

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 servers¹⁵ 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“⁴⁵.

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases^{16,25–27,41,50,69–72}. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference⁶⁹.

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^{16,32,38,84}. Learn more at <https://sqlite.org>⁶⁸.

terminal A terminal is a text-based window where you can enter commands and execute them^{6,20}. 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  + , dann Schreiben von `cmd`, dann Druck auf . Under Ubuntu Linux,  +  +  opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux^{20,37}. 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^{9,29}