合肥大學
HEFEI UNIVERSITY

Datenbanken

43. Logisches Schema: 3. Normalform

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-11-28

Databases

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

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

Outline

Einleitung

Einleitung

- Die 3. Normalform (EN: *third normal form*) (3NF) behandelt die Beziehungen zwischen nicht-Schlüssel-Attributen[13,16,18,23,31].

Einleitung

- Die 3. Normalform (EN: *third normal form*) (3NF) behandelt die Beziehungen zwischen nicht-Schlüssel-Attributen[13,16,18,23,31].
- Die 3NF ist verletzt, wenn ein nicht-Schlüssel-Attribut einen Fakt über ein anderes nicht-Schlüssel-Attribut darstellt[31].

Einleitung

- Die 3. Normalform (EN: *third normal form*) (3NF) behandelt die Beziehungen zwischen nicht-Schlüssel-Attributen[13,16,18,23,31].
- Die 3NF ist verletzt, wenn ein nicht-Schlüssel-Attribut einen Fakt über ein anderes nicht-Schlüssel-Attribut darstellt[31].
- Eine Relation R ist in der 3NF, wenn kein nicht-Schlüssel-Attribut *transitiv* von einem Schlüssel-Attribut abhängt.

Einleitung

- Die 3. Normalform (EN: *third normal form*) (3NF) behandelt die Beziehungen zwischen nicht-Schlüssel-Attributen[13,16,18,23,31].
- Die 3NF ist verletzt, wenn ein nicht-Schlüssel-Attribut einen Fakt über ein anderes nicht-Schlüssel-Attribut darstellt[31].
- Eine Relation R ist in der 3NF, wenn kein nicht-Schlüssel-Attribut *transitiv* von einem Schlüssel-Attribut abhängt.

Definition: Transitive Functionale Abhängigkeit

A, B, und C seien drei verschiede Attribute (oder Mengen von Attributen) der Realtion R, also gelten $A \subseteq \Sigma(R)$, $B \subseteq \Sigma(R)$ und $C \subseteq \Sigma(R)$. Die funktionale Abhängigkeit $A \rightarrow C$ ist eine *transitive Abhängigkeit* dann und nur dann wenn A→B und B→C beide wahr sind, aber B→A *nicht* wahr ist.

Einleitung

- Eine Relation R ist in der 3NF, wenn kein nicht-Schlüssel-Attribut *transitiv* von einem Schlüssel-Attribut abhängt.

Definition: Transitive Functionale Abhängigkeit

A, B, und C seien drei verschiede Attribute (oder Mengen von Attributen) der Realtion R, also gelten $A \subseteq \Sigma(R)$, $B \subseteq \Sigma(R)$ und $C \subseteq \Sigma(R)$. Die funktionale Abhängigkeit $A \rightarrow C$ ist eine *transitive Abhängigkeit* dann und nur dann wenn A→B und B→C beide wahr sind, aber B→A *nicht* wahr ist.

- Formal kann die 3NF so definiert werden[47]:

Einleitung

Definition: Transitive Functionale Abhängigkeit

A, B, und C seien drei verschiede Attribute (oder Mengen von Attributen) der Realtion R, also gelten $A \subseteq \Sigma(R)$, $B \subseteq \Sigma(R)$ und $C \subseteq \Sigma(R)$. Die funktionale Abhängigkeit $A \rightarrow C$ ist eine *transitive Abhängigkeit* dann und nur dann wenn A→B und B→C beide wahr sind, aber B→A *nicht* wahr ist.

- Formal kann die 3NF so definiert werden[47]:

Definition: 3. Normalform

Eine Relation R ist in der 3NF wenn sie in der 2NF ist und jedes Attribute $a \in \Sigma(R)$ das transitiv von einem Schlüssel $X \subseteq \Sigma(R)$ abhängt – für das also $X \rightarrow Y \rightarrow a$ mit $Y \subseteq \Sigma(R)$ gilt – ist es wahr, das entweder Y einen Schlüssel beinhaltet, a Teil des Primärschlüssels ist, oder $a \in X$.

Einleitung

Definition: Transitive Functionale Abhängigkeit

A, B, und C seien drei verschiede Attribute (oder Mengen von Attributen) der Realtion R, also gelten $A \subseteq \Sigma(R)$, $B \subseteq \Sigma(R)$ und $C \subseteq \Sigma(R)$. Die funktionale Abhängigkeit $A \to C$ ist eine *transitive Abhängigkeit* dann und nur dann wenn A→B und B→C beide wahr sind, aber B→A *nicht* wahr ist.

Definition: 3. Normalform

Eine Relation R ist in der 3NF wenn sie in der 2NF ist und jedes Attribute $a \in \Sigma(R)$ das transitiv von einem Schlüssel $X \subseteq \Sigma(R)$ abhängt – für das also $X \to Y \to a$ mit $Y \subseteq \Sigma(R)$ gilt – ist es wahr, das entweder Y einen Schlüssel beinhaltet, a Teil des Primärschlüssels ist, oder $a \in X$.

- Eine Tabelle ist in der 3NF, wenn sie in der 2NF ist und alle Attribute, die nicht Teil eines Schlüsselkandidats sind, direkt vom Primärschlüssel abhängen.

Einleitung

- Eine Relation R ist in der 3NF, wenn kein nicht-Schlüssel-Attribut *transitiv* von einem Schlüssel-Attribut abhängt.

Definition: 3. Normalform

Eine Relation R ist in der 3NF wenn sie in der 2NF ist und jedes Attribute $a \in \Sigma(R)$ das transitiv von einem Schlüssel $X \subseteq \Sigma(R)$ abhängt – für das also $X \to Y \to a$ mit $Y \subseteq \Sigma(R)$ gilt – ist es wahr, das entweder Y einen Schlüssel beinhaltet, a Teil des Primärschlüssels ist, oder $a \in X$.

- Eine Tabelle ist in der 3NF, wenn sie in der 2NF ist und alle Attribute, die nicht Teil eines Schlüsselkandidats sind, direkt vom Primärschlüssel abhängen.
- Jedes nicht-Primärschlüssel-Attribut ist nicht transitiv von jedem Schlüsselkandidat der Tabelle äbhängig.

Beispiel

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank.

Beispiel

- OK, diese Definition ist wieder auf den ersten Blick nicht so klar.
- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`.

Beispiel

- Schauen wir sie uns mit einem Beispiel an.
- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`.

Beispiel

- Stellen Sie sich vor, dass wir gerade am Design unserer Lehre-Management-Platform arbeiten.
- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile.

Beispiel

- Ein Kollege schlägt vor, dass es eine gute Idee wäre, auch die Kontaktinformation der Elten der Studenten zu speichern.
- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.

Beispiel

- Es kann ja durchaus Situationen geben, wo man diese Informationen gerne hätte.
- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten.

Beispiel

- Vielleicht hat ja ein Student einen Unfall.
- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten:
  - `student_id` $\rightarrow$ `student_name`.

Beispiel

- Vielleicht kommt ein Student nicht mehr zu Vorlesungen und kann nirgendwo gefunden werden.
- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten:
  - `student_id` $\rightarrow$ `student_name`,
  - `student_id` $\rightarrow$ `parent_name`.

Beispiel

- Dann würde man gerne die Eltern anrufen können.
- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten:
  - `student_id` → `student_name`,
  - `student_id` → `parent_name`,
  - `student_id` → `parent_mobile`.

Beispiel

- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten:
  - `student_id` → `student_name`,
  - `student_id` → `parent_name`,
  - `student_id` → `parent_mobile` und
  - `parent_mobile` → `parent_name`.

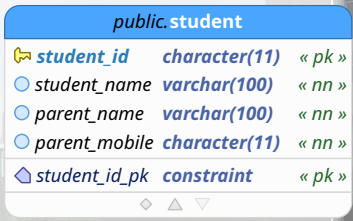
Beispiel

- Stellen Sie sich vor, wir speichern die folgenden Informationen für jeden Studenten in unserer Datenbank:
  - die von der Uni zugewiesene Studenten-ID `student_id`,
  - den Name `student_name`,
  - den Name `parent_name` eines der Elternteile und
  - die Mobiltelefonnummer `parent_mobile` dieses Elternteils.
- Wir erkennen sofort folgende funktionalen Abhängigkeiten:
  - `student_id` → `student_name`,
  - `student_id` → `parent_name`,
  - `student_id` → `parent_mobile` und
  - `parent_mobile` → `parent_name`.
- Wie gehen wir damit um?

Verletzung der 3. Normalform

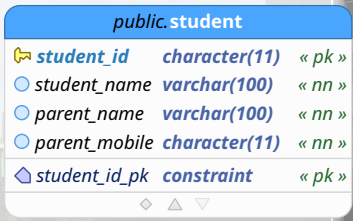
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.



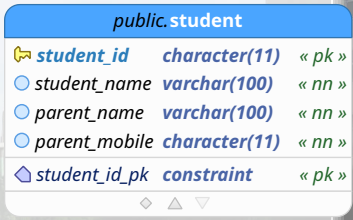
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.



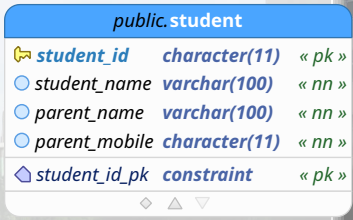
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.



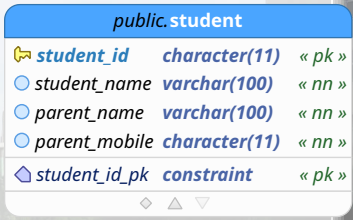
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.



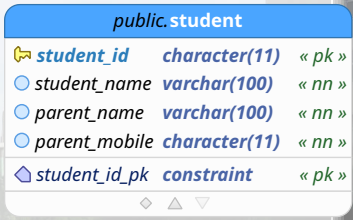
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.



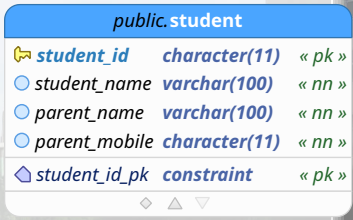
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.



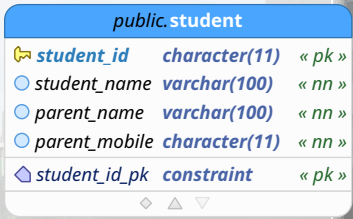
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.



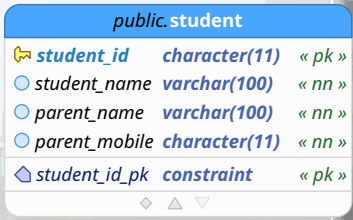
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.



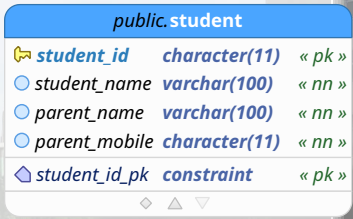
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.



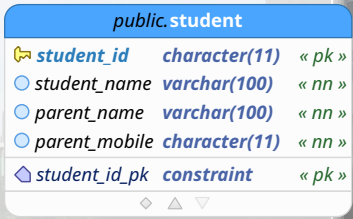
Verletzung der 3. Normalform

- Unser erster Ansatz ist, alle Daten in eine Tabelle zu tun.
- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.



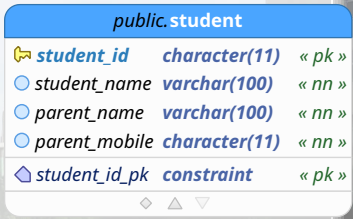
Verletzung der 3. Normalform

- Die Tabelle `student` speichert alle Informationen über die Studenten und ihre Eltern.
- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.



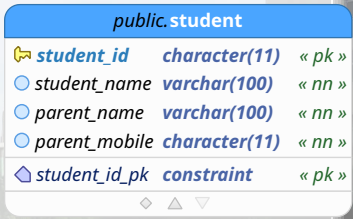
Verletzung der 3. Normalform

- Die Tabelle hat vier Spalten.
- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.



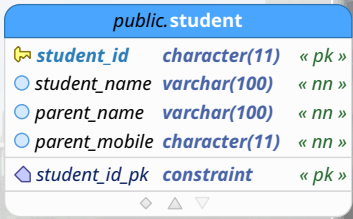
Verletzung der 3. Normalform

- Die erste speichert die Studenten-ID als Primärschlüssel.
- Die zweite speichert den Namen des Studenten.
- Dieser ist kein Schlüssel und auch nicht zwingend einzigartig.
- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile`$\rightarrow$`parent_name`.



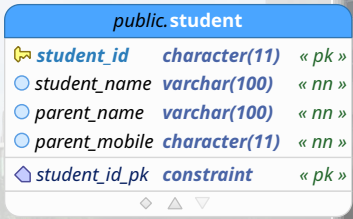
Verletzung der 3. Normalform

- Dann speichern wir noch den name der Eltern und deren Mobiltelefonnummer.
- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile` $\rightarrow$ `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.



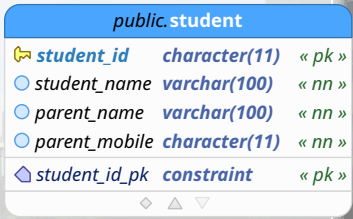
Verletzung der 3. Normalform

- Somit haben wir also die Notfallkontaktinformation.
- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.



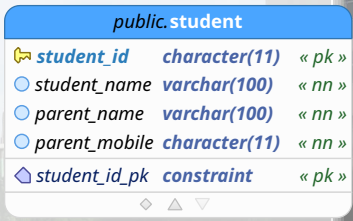
Verletzung der 3. Normalform

- Diese Tabelle ist in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id` → `parent_mobile` geschrieben werden.



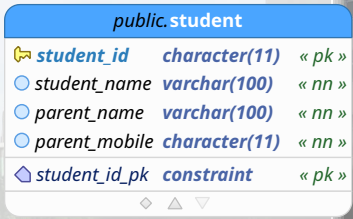
Verletzung der 3. Normalform

- Sie ist auch in der 2NF, weil es keinen zusammengesetzten Schlüssel gibt.
- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile`→`parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id`→`parent_mobile` geschrieben werden.
- Die Kette `student_id`→`parent_mobile`→`parent_name` existiert.



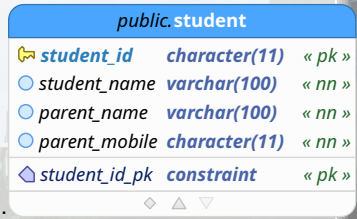
Verletzung der 3. Normalform

- Sie verletzt aber die 3NF.
- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id` → `parent_mobile` geschrieben werden.
- Die Kette `student_id` → `parent_mobile` → `parent_name` existiert.
- Es gilt natürlich nicht das `parent_mobile` → `student_id`.



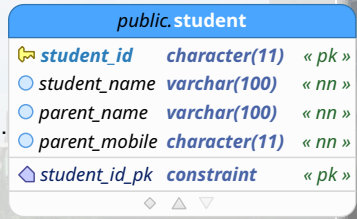
Verletzung der 3. Normalform

- Das Attribut `parent_name` ist funktional vom Attribut `parent_mobile` abhängig.
- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id` → `parent_mobile` geschrieben werden.
- Die Kette `student_id` → `parent_mobile` → `parent_name` existiert.
- Es gilt natürlich nicht das `parent_mobile` → `student_id`.
- Ein Elternteil könnte ja mehrere Kinder haben, die an unserer Uni studieren.



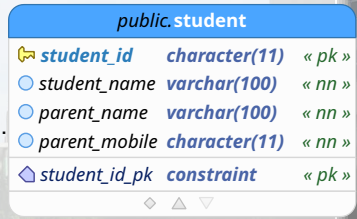
Verletzung der 3. Normalform

- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id` → `parent_mobile` geschrieben werden.
- Die Kette `student_id` → `parent_mobile` → `parent_name` existiert.
- Es gilt natürlich nicht das `parent_mobile` → `student_id`.
- Ein Elternteil könnte ja mehrere Kinder haben, die an unserer Uni studieren.
- Daher ist die Beziehung `student_id` → `parent_name` transitiv.



Verletzung der 3. Normalform

- Das schreiben wir als `parent_mobile` → `parent_name`.
- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id` → `parent_mobile` geschrieben werden.
- Die Kette `student_id` → `parent_mobile` → `parent_name` existiert.
- Es gilt natürlich nicht das `parent_mobile` → `student_id`.
- Ein Elternteil könnte ja mehrere Kinder haben, die an unserer Uni studieren.
- Daher ist die Beziehung `student_id` → `parent_name` transitiv.
- Und weil sie in Tabelle `student` auftaucht, verletzt das die 3NF.



Verletzung der 3. Normalform

- Eine Mobiltelefonnummer gehört zu einer einzelnen Person, deshalb kann es niemals mehr als einen Namen zu einer Mobiltelefonnummer geben.
- Die Mobiltelefonnummer des Elternteils hängt funktional vom Primärschlüssel `student_id` ab.
- Das kann als `student_id`→`parent_mobile` geschrieben werden.
- Die Kette `student_id`→`parent_mobile`→`parent_name` existiert.
- Es gilt natürlich nicht das `parent_mobile`→`student_id`.
- Ein Elternteil könnte ja mehrere Kinder haben, die an unserer Uni studieren.
- Daher ist die Beziehung `student_id`→`parent_name` transitiv.
- Und weil sie in Tabelle `student` auftaucht, verletzt das die 3NF.
- Erforschen wir mal, welche Konsequenzen das hat.

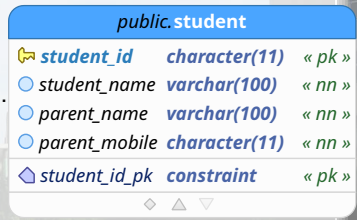


Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
   ↪ -ebf 03_public_student_table_5103.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
   ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Der Primärschlüssel `student_id` ist eine Zeichenkette, die immer die Länge 11 hat, also ein `CHARACTER(11)`, weil die Studenten-IDs in unserer Uni nun mal so sind.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf 03_public_student_table_5103.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Der Primärschlüssel `student_id` ist eine Zeichenkette, die immer die Länge 11 hat, also ein `CHARACTER(11)`, weil die Studenten-IDs in unserer Uni nun mal so sind.
- Die Namen der Studenten, gespeichert als `student_name`, sind variabel-lange Zeichenketten mit maximal 100 Zeichen, also vom Typ `VARCHAR(100)`.

```
1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      student_name varchar(100) NOT NULL,
6      parent_name varchar(100) NOT NULL,
7      parent_mobile character(11) NOT NULL,
8      CONSTRAINT student_id_pk PRIMARY KEY (student_id)
9  );
10 -- ddl-end --
11 ALTER TABLE public.student OWNER TO postgres;
12 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
       ↪ -ebf 03_public_student_table_5103.sql
2  psql: error: connection to server at "localhost" (::1), port 5432 failed
       ↪ : FATAL:  database "violation" does not exist
3  # psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Der Primärschlüssel `student_id` ist eine Zeichenkette, die immer die Länge 11 hat, also ein `CHARACTER(11)`, weil die Studenten-IDs in unserer Uni nun mal so sind.
- Die Namen der Studenten, gespeichert als `student_name`, sind variabel-lange Zeichenketten mit maximal 100 Zeichen, also vom Typ `VARCHAR(100)`.
- Das selbe gilt für die Namen `parent_name` der Eltern.

```
1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      student_name varchar(100) NOT NULL,
6      parent_name varchar(100) NOT NULL,
7      parent_mobile character(11) NOT NULL,
8      CONSTRAINT student_id_pk PRIMARY KEY (student_id)
9  );
10 -- ddl-end --
11 ALTER TABLE public.student OWNER TO postgres;
12 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
       ↪ -ebf 03_public_student_table_5103.sql
2  psql: error: connection to server at "localhost" (::1), port 5432 failed
       ↪ : FATAL:  database "violation" does not exist
3  # psql 16.11 failed with exit code 2.
```

Tabelle student

- Der Primärschlüssel `student_id` ist eine Zeichenkette, die immer die Länge 11 hat, also ein `CHARACTER(11)`, weil die Studenten-IDs in unserer Uni nun mal so sind.
- Die Namen der Studenten, gespeichert als `student_name`, sind variabel-lange Zeichenketten mit maximal 100 Zeichen, also vom Typ `VARCHAR(100)`.
- Das selbe gilt für die Namen `parent_name` der Eltern.
- Die Mobiltelefonnummer der Eltern, `parent_mobile`, wird wieder als Zeichenkette der festen Länge 11 gespeichert.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf 03_public_student_table_5103.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Die Namen der Studenten, gespeichert als `student_name`, sind variabel-lange Zeichenketten mit maximal 100 Zeichen, also vom Typ `VARCHAR(100)`.
- Das selbe gilt für die Namen `parent_name` der Eltern.
- Die Mobiltelefonnummer der Eltern, `parent_mobile`, wird wieder als Zeichenkette der festen Länge 11 gespeichert.
- Keine der vier Spalten ist optional, daher sind sie alle mit `NOT NULL` markiert.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf 03_public_student_table_5103.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir die Tabelle `student` mit Daten!

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',    'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',    'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir die Tabelle `student` mit Daten!
- Wir speichern vier Studentendatensätze.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir die Tabelle `student` mit Daten!
- Wir speichern vier Studentendatensätze.
- Es gibt die vier Studenten Herr Bibbo, Herr Bebbo, Herr Bibboto und Frau Bibboba.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir die Tabelle `student` mit Daten!
- Wir speichern vier Studentendatensätze.
- Es gibt die vier Studenten Herr Bibbo, Herr Bebbo, Herr Bibboto und Frau Bibboba.
- Ihre IDs sind unwichtig.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir die Tabelle `student` mit Daten!
- Wir speichern vier Studentendatensätze.
- Es gibt die vier Studenten Herr Bibbo, Herr Bebbo, Herr Bibboto und Frau Bibboba.
- Ihre IDs sind unwichtig.
- Was wichtig ist, ist das Herr Bibbo, Herr Bibboto und Frau Bibboba Geschwister sind!

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Wir speichern vier Studentendatensätze.
- Es gibt die vier Studenten Herr Bibbo, Herr Bebbo, Herr Bibboto und Frau Bibboba.
- Ihre IDs sind unwichtig.
- Was wichtig ist, ist das Herr Bibbo, Herr Bibboto und Frau Bibboba Geschwister sind!
- Ihr stolzer Vater ist Herr Böddö.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Es gibt die vier Studenten Herr Bibbo, Herr Bebbo, Herr Bibboto und Frau Bibboba.
- Ihre IDs sind unwichtig.
- Was wichtig ist, ist das Herr Bibbo, Herr Bibboto und Frau Bibboba Geschwister sind!
- Ihr stolzer Vater ist Herr Böddö.
- Nun sind wir aber eben in China und der Sekretär, der die Daten eingeben sollte, wusste nicht, wie er den Buchstabe [ö] eingeben kann.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Ihre IDs sind unwichtig.
- Was wichtig ist, ist das Herr Bibbo, Herr Bibboto und Frau Bibboba Geschwister sind!
- Ihr stolzer Vater ist Herr Böddö.
- Nun sind wir aber eben in China und der Sekretär, der die Daten eingeben sollte, wusste nicht, wie er den Buchstabe [ö] eingeben kann.
- Deshalb hat er sich entschlossen, den Vater der drei Studenten kruzerhand „Herr Boddo“ zu nennen.

```
/** Insert data into the database. */

-- Insert several student + parent records.
INSERT INTO student (student_id, student_name, parent_name,
                     parent_mobile) VALUES
    ('1234567890', 'Bibbo',   'Boddo', '55544466677'),
    ('1234567891', 'Bebbo',   'Balla', '77788811122'),
    ('1234567892', 'Bibboto', 'Boddo', '55544466677'),
    ('1234567894', 'Bibboba', 'Boddo', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Ein paar Tage später hat er dann herausgefunden, wie man ein ö eingibt.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Ein paar Tage später hat er dann herausgefunden, wie man ein ö eingibt.
- Er hat einen deutschen Kollegen gebeten, ihm den Buchstaben über WeChat zu schicken, so dass er ihn copy-pasten kann.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Ein paar Tage später hat er dann herausgefunden, wie man ein ö eingibt.
- Er hat einen deutschen Kollegen gebeten, ihm den Buchstaben über WeChat zu schicken, so dass er ihn copy-pasten kann.
- Um die Daten zu korrigieren, hat er das Skript rechts geschrieben.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Ein paar Tage später hat er dann herausgefunden, wie man ein ö eingibt.
- Er hat einen deutschen Kollegen gebeten, ihm den Buchstaben über WeChat zu schicken, so dass er ihn copy-pasten kann.
- Um die Daten zu korrigieren, hat er das Skript rechts geschrieben.
- Es benutzt die Mobiltelefonnummer 555 444 666 77 von Herrn Böddö um ihn in der Tabelle zu finden.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Ein paar Tage später hat er dann herausgefunden, wie man ein ö eingibt.
- Er hat einen deutschen Kollegen gebeten, ihm den Buchstaben über WeChat zu schicken, so dass er ihn copy-pasten kann.
- Um die Daten zu korrigieren, hat er das Skript rechts geschrieben.
- Es benutzt die Mobiltelefonnummer 555 444 666 77 von Herrn Böddö um ihn in der Tabelle zu finden.
- Das wird erstmal mit einer SELECT-Anfrage ausprobiert.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Er hat einen deutschen Kollegen gebeten, ihm den Buchstaben über WeChat zu schicken, so dass er ihn copy-pasten kann.
- Um die Daten zu korrigieren, hat er das Skript rechts geschrieben.
- Es benutzt die Mobiltelefonnummer 555 444 666 77 von Herrn Böddö um ihn in der Tabelle zu finden.
- Das wird erstmal mit einer `SELECT`-Anfrage ausprobiert.
- Diese liefert wirklich die drei Student/Eltern-Paare zurück.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Um die Daten zu korrigieren, hat er das Skript rechts geschrieben.
- Es benutzt die Mobiltelefonnummer 555 444 666 77 von Herrn Böddö um ihn in der Tabelle zu finden.
- Das wird erstmal mit einer SELECT-Anfrage ausprobiert.
- Diese liefert wirklich die drei Student/Eltern-Paare zurück.
- Jetzt können wir also ein UPDATE-Kommando ausführen.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Es benutzt die Mobiltelefonnummer 555 444 666 77 von Herrn Böddö um ihn in der Tabelle zu finden.
- Das wird erstmal mit einer `SELECT`-Anfrage ausprobiert.
- Diese liefert wirklich die drei Student/Eltern-Paare zurück.
- Jetzt können wir also ein `UPDATE`-Kommando ausführen.
- Der `parent_name` wird auf `Böddö` gesetzt, und zwar für jeden DAtensatz mit `parent_mobile = '55544466677'`.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Das wird erstmal mit einer `SELECT`-Anfrage ausprobiert.
- Diese liefert wirklich die drei Student/Eltern-Paare zurück.
- Jetzt können wir also ein `UPDATE`-Kommando ausführen.
- Der `parent_name` wird auf `Böddö` gesetzt, und zwar für jeden DAtensatz mit `parent_mobile = '55544466677'`.
- Die geänderten Datensätze holen wir uns über das `RETURNING`-Statement.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Diese liefert wirklich die drei Student/Eltern-Paare zurück.
- Jetzt können wir also ein `UPDATE`-Kommando ausführen.
- Der `parent_name` wird auf `Böddö` gesetzt, und zwar für jeden DAtensatz mit `parent_mobile = '55544466677'`.
- Die geänderten Datensätze holen wir uns über das `RETURNING`-Statement.
- Wir sehen, dass die richtigen drei Zeilen geändert wurden.

```
1  /** Find the parent Boddo and change his name to Böddö. */
2
3  -- Get the names of all students whose parent has mobile 55544466677.
4  SELECT student_name, parent_name FROM student
5      WHERE parent_mobile = '55544466677';
6
7  -- Change the name of the parent with mobile 55544466677 to Böddö.
8  UPDATE student SET parent_name = 'Böddö'
9      WHERE parent_mobile = '55544466677'
10     RETURNING student_name, parent_name;
```

```
1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
       ↪ -ebf update.sql
2  psql: error: connection to server at "localhost" (::1), port 5432 failed
       ↪ : FATAL:  database "violation" does not exist
3  # psql 16.11 failed with exit code 2.
```

Daten Ändern

- Jetzt können wir also ein `UPDATE`-Kommando ausführen.
- Der `parent_name` wird auf `Böddö` gesetzt, und zwar für jeden DAtensatz mit `parent_mobile = '55544466677'`.
- Die geänderten Datensätze holen wir uns über das `RETURNING`-Statement.
- Wir sehen, dass die richtigen drei Zeilen geändert wurden.
- Um eine Information zu ändern, mussten wir drei Zeilen anfassen.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Der `parent_name` wird auf `Böddö` gesetzt, und zwar für jeden DAtensatz mit `parent_mobile = '55544466677'`.
- Die geänderten Datensätze holen wir uns über das `RETURNING`-Statement.
- Wir sehen, dass die richtigen drei Zeilen geändert wurden.
- Um eine Information zu ändern, mussten wir drei Zeilen anfassen.
- Das ist ein klassisches Beispiel einer Update-Anomalie.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die geänderten Datensätze holen wir uns über das `RETURNING`-Statement.
- Wir sehen, dass die richtigen drei Zeilen geändert wurden.
- Um eine Information zu ändern, mussten wir drei Zeilen anfassen.
- Das ist ein klassisches Beispiel einer Update-Anomalie.
- Wenn wir die Datensätze von Herr Bibbo, Herr Bibboto und Frau Bibboba löschen würden, dann würden auch die Daten über ihren Vater verschwinden.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Wir sehen, dass die richtigen drei Zeilen geändert wurden.
- Um eine Information zu ändern, mussten wir drei Zeilen anfassen.
- Das ist ein klassisches Beispiel einer Update-Anomalie.
- Wenn wir die Datensätze von Herr Bibbo, Herr Bibboto und Frau Bibboba löschen würden, dann würden auch die Daten über ihren Vater verschwinden.
- Schlimmer: Wenn wir die Daten eines Elternteils aus der Datenbank löschen wöllten, dann würden wir automatisch alle Daten von dessen Studenten-Kindern mit löschen.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Um eine Information zu ändern, mussten wir drei Zeilen anfassen.
- Das ist ein klassisches Beispiel einer Update-Anomalie.
- Wenn wir die Datensätze von Herr Bibbo, Herr Bibboto und Frau Bibboba löschen würden, dann würden auch die Daten über ihren Vater verschwinden.
- Schlimmer: Wenn wir die Daten eines Elternteils aus der Datenbank löschen wöllten, dann würden wir automatisch alle Daten von dessen Studenten-Kindern mit löschen.
- Das wäre also eine Lösch-Anomalie.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Das ist ein klassisches Beispiel einer Update-Anomalie.
- Wenn wir die Datensätze von Herr Bibbo, Herr Bibboto und Frau Bibboba löschen würden, dann würden auch die Daten über ihren Vater verschwinden.
- Schlimmer: Wenn wir die Daten eines Elternteils aus der Datenbank löschen wölltten, dann würden wir automatisch alle Daten von dessen Studenten-Kindern mit löschen.
- Das wäre also eine Lösch-Anomalie.
- Andersherum können wir keine Daten eines Studenten eingeben, ohne auch die Daten von einem Elternteil einzugeben, und andersherum.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Wenn wir die Datensätze von Herr Bibbo, Herr Bibboto und Frau Bibboba löschen würden, dann würden auch die Daten über ihren Vater verschwinden.
- Schlimmer: Wenn wir die Daten eines Elternteils aus der Datenbank löschen wöllten, dann würden wir automatisch alle Daten von dessen Studenten-Kindern mit löschen.
- Das wäre also eine Lösch-Anomalie.
- Andersherum können wir keine Daten eines Studenten eingeben, ohne auch die Daten von einem Elternteil einzugeben, und andersherum.
- Das ist eine Einfüge-Anomalie.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Schlimmer: Wenn wir die Daten eines Elternteils aus der Datenbank löschen wöllten, dann würden wir automatisch alle Daten von dessen Studenten-Kindern mit löschen.
- Das wäre also eine Lösch-Anomalie.
- Andersherum können wir keine Daten eines Studenten eingeben, ohne auch die Daten von einem Elternteil einzugeben, und andersherum.
- Das ist eine Einfüge-Anomalie.
- Und natürlich haben wir wieder Redundanz.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Das wäre also eine Lösch-Anomalie.
- Andersherum können wir keine Daten eines Studenten eingeben, ohne auch die Daten von einem Elternteil einzugeben, und andersherum.
- Das ist eine Einfüge-Anomalie.
- Und natürlich haben wir wieder Redundanz.
- Die Daten von Herrn Böddö werden dreimal gespeichert.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Andersherum können wir keine Daten eines Studenten eingeben, ohne auch die Daten von einem Elternteil einzugeben, und andersherum.
- Das ist eine Einfüge-Anomalie.
- Und natürlich haben wir wieder Redundanz.
- Die Daten von Herrn Böddö werden dreimal gespeichert.
- Diese Redundanz löst im Grunde die Update-Anomalie aus.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Das ist eine Einfüge-Anomalie.
- Und natürlich haben wir wieder Redundanz.
- Die Daten von Herrn Böddö werden dreimal gespeichert.
- Diese Redundanz löst im Grunde die Update-Anomalie aus.
- Und sie zwingt uns dazu, die selbe Information mehrfach einzugeben.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Und natürlich haben wir wieder Redundanz.
- Die Daten von Herrn Böddö werden dreimal gespeichert.
- Diese Redundanz löst im Grunde die Update-Anomalie aus.
- Und sie zwingt uns dazu, die selbe Information mehrfach einzugeben.
- Dadurch steigt die Wahrscheinlichkeit von Fehlern und Inkonsistenzen.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Daten von Herrn Böddö werden dreimal gespeichert.
- Diese Redundanz löst im Grunde die Update-Anomalie aus.
- Und sie zwingt uns dazu, die selbe Information mehrfach einzugeben.
- Dadurch steigt die Wahrscheinlichkeit von Fehlern und Inkonsistenzen.
- Es hätte ja sein können, dass die Daten der Studenten von verschiedenen Lehrern eingegeben werden.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Diese Redundanz löst im Grunde die Update-Anomalie aus.
- Und sie zwingt uns dazu, die selbe Information mehrfach einzugeben.
- Dadurch steigt die Wahrscheinlichkeit von Fehlern und Inkonsistenzen.
- Es hätte ja sein können, dass die Daten der Studenten von verschiedenen Lehrern eingegeben werden.
- Vielleicht weiß ja ein Lehrer, wie man ein [ö] eingibt, und ein ander weiß es nicht und nimmt stattdessen [o].

```
/** Find the parent Boddo and change his name to Böddö. */

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    WHERE parent_mobile = '55544466677';

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE student SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING student_name, parent_name;
```

```
$ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
    ↪ -ebf update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "violation" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Und sie zwingt uns dazu, die selbe Information mehrfach einzugeben.
- Dadurch steigt die Wahrscheinlichkeit von Fehlern und Inkonsistenzen.
- Es hätte ja sein können, dass die Daten der Studenten von verschiedenen Lehrern eingegeben werden.
- Vielleicht weiß ja ein Lehrer, wie man ein ö eingibt, und ein ander weiß es nicht und nimmt stattdessen o.
- Und schon sind unsere Daten inkonsistent, denn wir hätten den selben Elternteil unter zwei verschiedenen Namen im System.

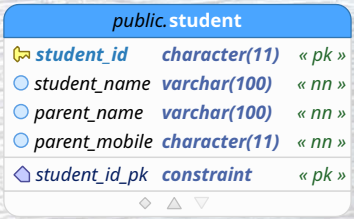
```
1  /** Find the parent Boddo and change his name to Böddö. */
2
3  -- Get the names of all students whose parent has mobile 55544466677.
4  SELECT student_name, parent_name FROM student
5      WHERE parent_mobile = '55544466677';
6
7  -- Change the name of the parent with mobile 55544466677 to Böddö.
8  UPDATE student SET parent_name = 'Böddö'
9      WHERE parent_mobile = '55544466677'
10     RETURNING student_name, parent_name;
```

```
1  $ psql "postgres://postgres:XXX@localhost/violation" -v ON_ERROR_STOP=1
       ↪ -ebf update.sql
2  psql: error: connection to server at "localhost" (::1), port 5432 failed
       ↪ : FATAL:  database "violation" does not exist
3  # psql 16.11 failed with exit code 2.
```

Lösung: Wiederherstellung der 3. Normalform

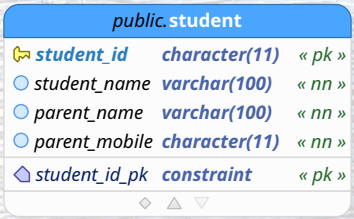
Lösung

- Lösen wir das Problem.



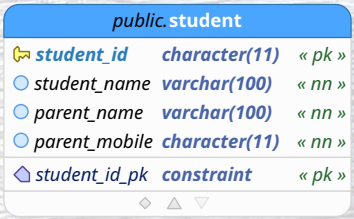
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.



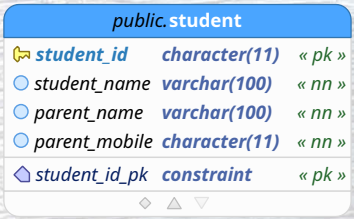
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.



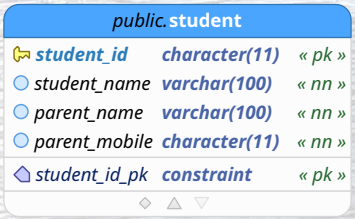
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängigkeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.



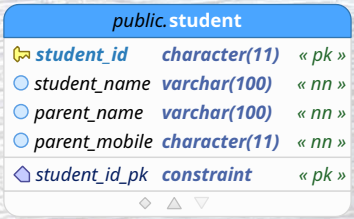
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängigkeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.



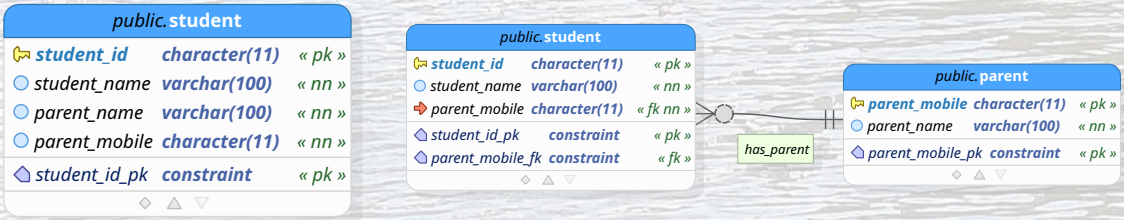
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängigkeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.



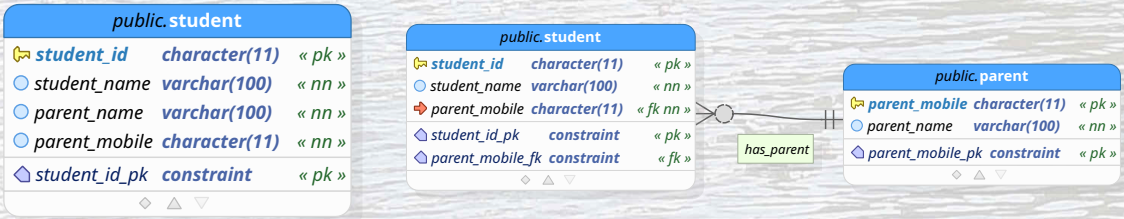
Lösung

- Lösen wir das Problem.
- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängigkeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.



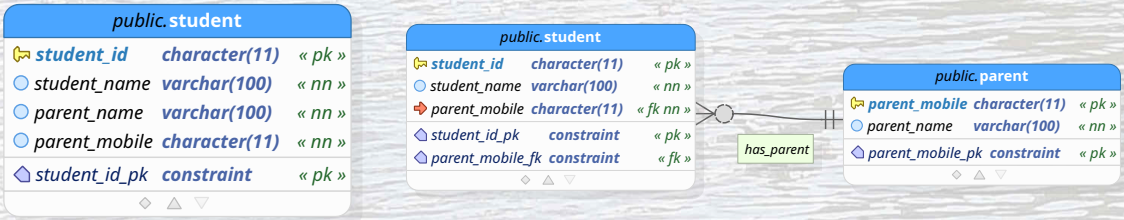
Lösung

- Wir müssen die Daten in die 3NF bringen.
- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängigkeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.



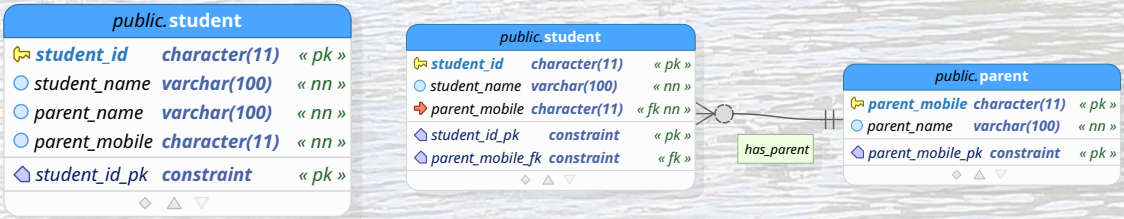
Lösung

- Dafür müssen wir die transitive funktionale Abhängigkeit `student_id` → `parent_mobile` → `parent_name` weg bekommen.
- Diese Abhängikeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.



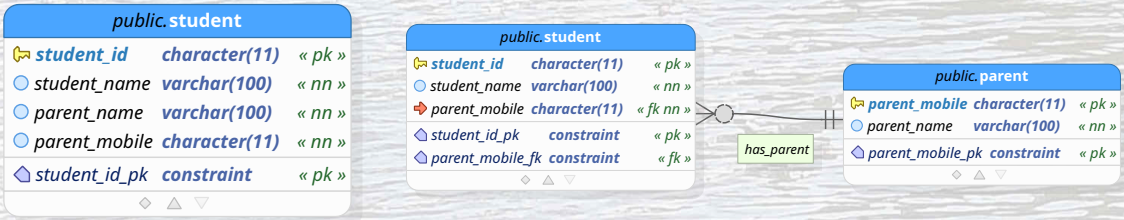
Lösung

- Diese Abhängikeit stört, weil `parent_mobile` kein Schlüssel der Tabelle `student` ist, aber `parent_name` von `parent_mobile` bestimmt wird.
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.
- Also ziehen wir diese Daten aus der Tabelle `student` heraus unt tun sie in die neue Tabelle `parent`.



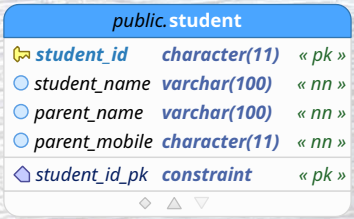
Lösung

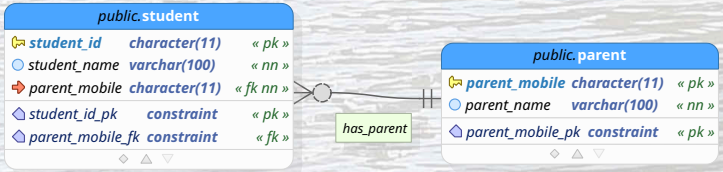
- So lange diese drei Spalten in der selben Tabelle sind, bleibt die 3NF verletzt.
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.
- Also ziehen wir diese Daten aus der Tabelle `student` heraus unt tun sie in die neue Tabelle `parent`.
- Als Primärschlüssel verwenden wir `parent_mobile`.



Lösung

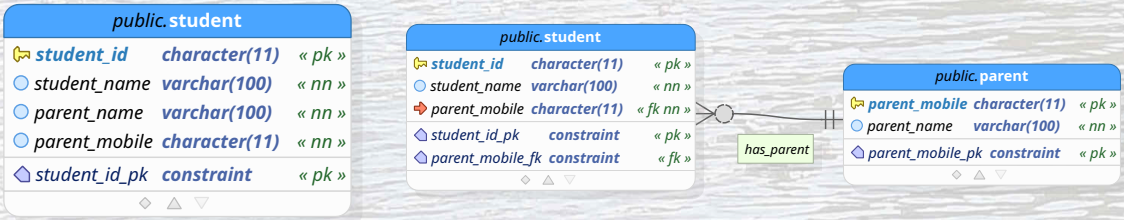
- Also müssen sie in unterschiedliche Tabellen.
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.
- Also ziehen wir diese Daten aus der Tabelle `student` heraus unt tun sie in die neue Tabelle `parent`.
- Als Primärschlüssel verwenden wir `parent_mobile`.
- Dieser Wert ist einmalig und identifiziert einen Elternteil.





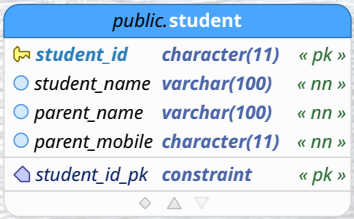
Lösung

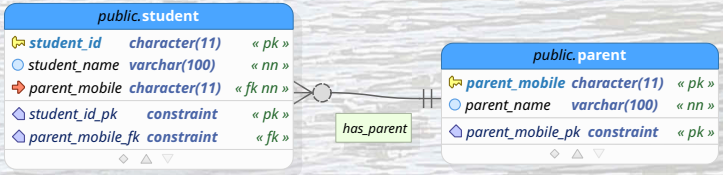
- Hier zeichnen wir ein logisches Modell, dass die 3NF nicht mehr verletzt.
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.
- Also ziehen wir diese Daten aus der Tabelle `student` heraus unt tun sie in die neue Tabelle `parent`.
- Als Primärschlüssel verwenden wir `parent_mobile`.
- Dieser Wert ist einmalig und identifiziert einen Elternteil.
- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.



Lösung

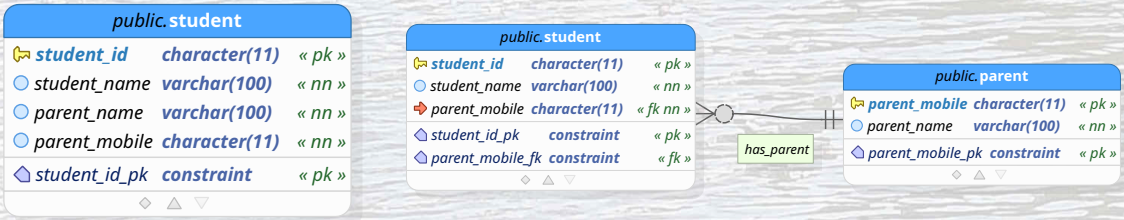
- Anders als in dem ursprünglichen Design benutzen wir zwei Tabellen.
- Der Name der Eltern hängt von deren Mobiltelefonnummer ab.
- Also ziehen wir diese Daten aus der Tabelle `student` heraus unt tun sie in die neue Tabelle `parent`.
- Als Primärschlüssel verwenden wir `parent_mobile`.
- Dieser Wert ist einmalig und identifiziert einen Elternteil.
- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.
- Die veränderte Tabelle `student` benutzt nun das Attribut `parent_mobile` als Fremdschlüssel.





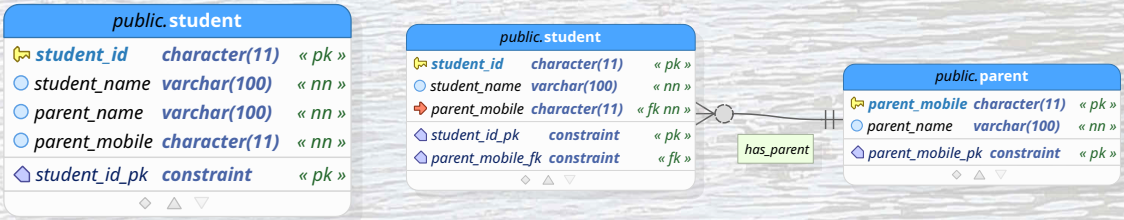
Lösung

- Als Primärschlüssel verwenden wir `parent_mobile`.
- Dieser Wert ist einmalig und identifiziert einen Elternteil.
- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.
- Die veränderte Tabelle `student` benutzt nun das Attribut `parent_mobile` als Fremdschlüssel.
- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.



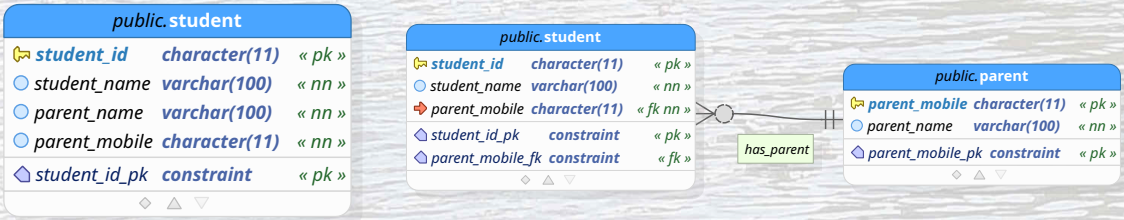
Lösung

- Dieser Wert ist einmalig und identifiziert einen Elternteil.
- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.
- Die veränderte Tabelle `student` benutzt nun das Attribut `parent_mobile` als Fremdschlüssel.
- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.
- Wir speichern den Name der Eltern nicht mehr in der Tabelle `student`.



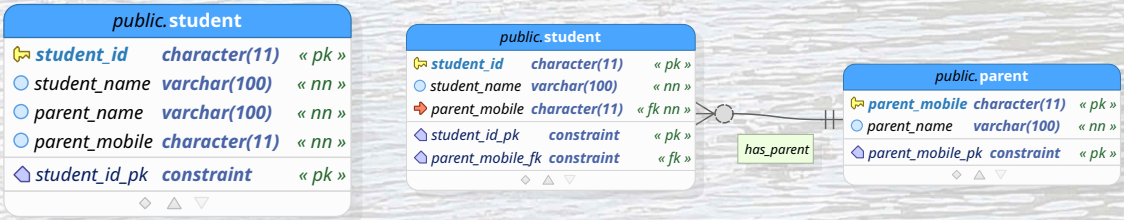
Lösung

- Dieser Wert ist einmalig und identifiziert einen Elternteil.
- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.
- Die veränderte Tabelle `student` benutzt nun das Attribut `parent_mobile` als Fremdschlüssel.
- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.
- Wir speichern den Name der Eltern nicht mehr in der Tabelle `student`.



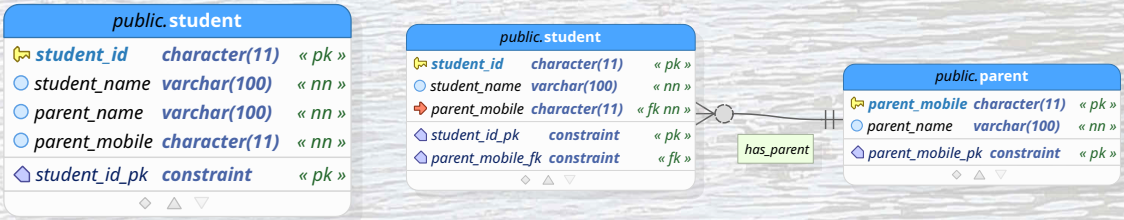
Lösung

- Die zweite Spalte dieser Tabelle ist `name` – und die hängt natürlich vom Primärschlüssel ab.
- Die veränderte Tabelle `student` benutzt nun das Attribut `parent_mobile` als Fremdschlüssel.
- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.
- Wir speichern den Name der Eltern nicht mehr in der Tabelle `student`.
- Beide Tabellen sind in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.



Lösung

- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.
- Wir speichern den Name der Eltern nicht mehr in der Tabelle `student`.
- Beide Tabellen sind in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie sind auch in der 2NF, weil es keinen zusammengesetzte Schlüssel gibt und daher kein Attribut von einer echten Teilmenge eines Schlüssels abhängen kann.



Lösung

- Dieser muss `NOT NULL` sein, was heist, dass es jede Zeile der Tabelle `student` mit genau einer Zeile in der Tabelle `parent` verbunden ist.
- Wir speichern den Name der Eltern nicht mehr in der Tabelle `student`.
- Beide Tabellen sind in der 1NF, weil es weder zusammengesetzte Attribute noch wiederholte Gruppen gibt.
- Sie sind auch in der 2NF, weil es keinen zusammengesetzte Schlüssel gibt und daher kein Attribut von einer echten Teilmenge eines Schlüssels abhängen kann.
- Sie befolgen auch die 3NF, weil es keine transitiven funktionalen Abhängigkeiten mehr gibt.

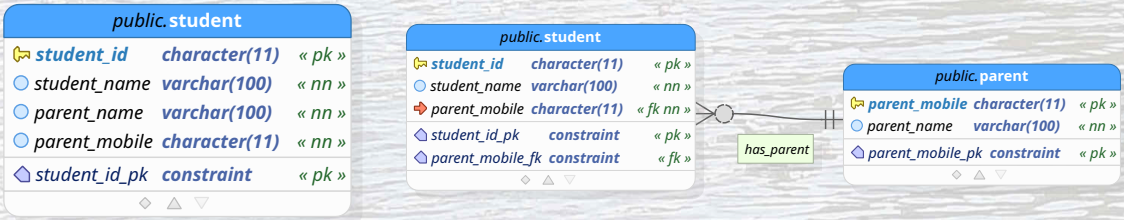


Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 03_public_student_table_5071.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Diese Tabelle hat die drei Spalten `student_id`, `student_name` und `parent_mobile` behalten.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 03_public_student_table_5071.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Diese Tabelle hat die drei Spalten `student_id`, `student_name` und `parent_mobile` behalten.
- Die Spalte `parent_name` gibt es jetzt nicht mehr.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 03_public_student_table_5071.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Diese Tabelle hat die drei Spalten `student_id`, `student_name` und `parent_mobile` behalten.
- Die Spalte `parent_name` gibt es jetzt nicht mehr.
- Der Primärschlüssel ist immer noch `student_id`.

```
1  -- object: public.student | type: TABLE --
2  -- DROP TABLE IF EXISTS public.student CASCADE;
3  CREATE TABLE public.student (
4      student_id character(11) NOT NULL,
5      student_name varchar(100) NOT NULL,
6      parent_mobile character(11) NOT NULL,
7      CONSTRAINT student_id_pk PRIMARY KEY (student_id)
8  );
9  -- ddl-end --
10 ALTER TABLE public.student OWNER TO postgres;
11 -- ddl-end --
```

```
1  $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
       ↪ 03_public_student_table_5071.sql
2  psql: error: connection to server at "localhost" (::1), port 5432 failed
       ↪ : FATAL:  database "fixed" does not exist
3  # psql 16.11 failed with exit code 2.
```

Tabelle student

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `student`.
- Diese Tabelle hat die drei Spalten `student_id`, `student_name` und `parent_mobile` behalten.
- Die Spalte `parent_name` gibt es jetzt nicht mehr.
- Der Primärschlüssel ist immer noch `student_id`.
- Die Spalte `parent_mobile` wird dann ein Fremdschlüssel auf die Tabelle mit den Eltern-Datensätzen.

```
-- object: public.student | type: TABLE --
-- DROP TABLE IF EXISTS public.student CASCADE;
CREATE TABLE public.student (
    student_id character(11) NOT NULL,
    student_name varchar(100) NOT NULL,
    parent_mobile character(11) NOT NULL,
    CONSTRAINT student_id_pk PRIMARY KEY (student_id)
);
-- ddl-end --
ALTER TABLE public.student OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 03_public_student_table_5071.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle parent

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `parent`.

```
-- object: public.parent | type: TABLE --
-- DROP TABLE IF EXISTS public.parent CASCADE;
CREATE TABLE public.parent (
    parent_mobile character(11) NOT NULL,
    parent_name varchar(100) NOT NULL,
    CONSTRAINT parent_mobile_pk PRIMARY KEY (parent_mobile)
);
-- ddl-end --
ALTER TABLE public.parent OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 04_public_parent_table_5077.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle parent

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `parent`.
- Diese Tabelle hat nur zwei Spalten.

```
-- object: public.parent | type: TABLE --
-- DROP TABLE IF EXISTS public.parent CASCADE;
CREATE TABLE public.parent (
    parent_mobile character(11) NOT NULL,
    parent_name varchar(100) NOT NULL,
    CONSTRAINT parent_mobile_pk PRIMARY KEY (parent_mobile)
);
-- ddl-end --
ALTER TABLE public.parent OWNER TO postgres;
-- ddl-end --
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 04_public_parent_table_5077.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Tabelle parent

- Hier ist das auto-generierte Skript zum Erstellen der Tabelle `parent`.
- Diese Tabelle hat nur zwei Spalten.
- 1. den Primärschlüssel `parent_mobile` und 2. den Name des Elternteils.

```
1  -- object: public.parent | type: TABLE --
2  -- DROP TABLE IF EXISTS public.parent CASCADE;
3  CREATE TABLE public.parent (
4      parent_mobile character(11) NOT NULL,
5      parent_name varchar(100) NOT NULL,
6      CONSTRAINT parent_mobile_pk PRIMARY KEY (parent_mobile)
7  );
8  -- ddl-end --
9  ALTER TABLE public.parent OWNER TO postgres;
10 -- ddl-end --
```

```
1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 04_public_parent_table_5077.sql
2 psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
3 # psql 16.11 failed with exit code 2.
```

Fremdschlüssel-Einschränkung

- Hier ist das auto-generierte Skript zum Hinzufügen der Fremdschlüssel-Einschränkung zur Tabelle `student`, die erzwingt, dass jede Zeile in Tabelle `student` mit genau einer Zeile in Tabelle `parent` verbunden ist.

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

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ 05_public_student_parent_mobile_fk_constraint_5081.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Fremdschlüssel-Einschränkung

- Hier ist das auto-generierte Skript zum Hinzufügen der Fremdschlüssel-Einschränkung zur Tabelle `student`, die erzwingt, dass jede Zeile in Tabelle `student` mit genau einer Zeile in Tabelle `parent` verbunden ist.
- Dieser Schritt macht aus der Spalte `parent_mobile` der Tabelle `student` erst einen Fremdschlüssel.

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

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
   ↪ 05_public_student_parent_mobile_fk_constraint_5081.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
   ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir also die Tabellen mit den selben Daten wie vorhin.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir also die Tabellen mit den selben Daten wie vorhin.
- Wir fangen damit an, dass wir die beiden Elterndatensätze in die Tabelle `parent` eingeben.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir also die Tabellen mit den selben Daten wie vorhin.
- Wir fangen damit an, dass wir die beiden Elterndatensätze in die Tabelle `parent` eingeben.
- die Mobiltelefonnummern von Frau Balla, der Mutter von Herrn Bebbo, und von Herrn Böddö, dem Vater der anderen drei Studenten, werden gespeichert.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Füllen wir also die Tabellen mit den selben Daten wie vorhin.
- Wir fangen damit an, dass wir die beiden Elterndatensätze in die Tabelle `parent` eingeben.
- die Mobiltelefonnummern von Frau Balla, der Mutter von Herrn Bebbo, und von Herrn Böddö, dem Vater der anderen drei Studenten, werden gespeichert.
- Wie letztes mal wusste der Sekretär nicht, wie man ein ö schreibt, und nennt daher Herrn Böddö wieder erstmal Herr Boddo.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Wir fangen damit an, dass wir die beiden Elterndatensätze in die Tabelle `parent` eingeben.
- die Mobiltelefonnummern von Frau Balla, der Mutter von Herrn Bebbo, und von Herrn Böddö, dem Vater der anderen drei Studenten, werden gespeichert.
- Wie letztes mal wusste der Sekretär nicht, wie man ein `ö` schreibt, und nennt daher Herrn Böddö wieder erstmal Herr Boddo.
- Dann fügen wir die vier Studentendatensätze für Herrn Bibbo, Herrn Bebbo, Herrn Bibboto and Frau Bibboba ein.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- die Mobiltelefonnummern von Frau Balla, der Mutter von Herrn Bebbo, und von Herrn Böddö, dem Vater der anderen drei Studenten, werden gespeichert.
- Wie letztes mal wusste der Sekretär nicht, wie man ein ö schreibt, und nennt daher Herrn Böddö wieder erstmal Herr Boddo.
- Dann fügen wir die vier Studentendatensätze für Herrn Bibbo, Herrn Bebbo, Herrn Bibboto and Frau Bibboba ein.
- Hier geben wir nicht mehr die Namen ihrer Eltern an.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Einfügen

- Wie letztes mal wusste der Sekretär nicht, wie man ein ö schreibt, und nennt daher Herrn Böddö wieder erstmal Herr Boddo.
- Dann fügen wir die vier Studentendatensätze für Herrn Bibbo, Herrn Bebbo, Herrn Bibboto and Frau Bibboba ein.
- Hier geben wir nicht mehr die Namen ihrer Eltern an.
- Nur die Mobiltelefonnummern, die auf die entsprechenden Datensätze in Tabelle `parent` verweisen, werden benötigt.

```
/** Insert data into the database. */

-- Insert several parent records.
INSERT INTO parent (parent_name, parent_mobile) VALUES
    ('Boddo', '55544466677'),
    ('Balla', '77788811122');

-- Insert several student records that are linked to parent records.
INSERT INTO student (student_id, student_name, parent_mobile) VALUES
    ('1234567890', 'Bibbo', '55544466677'),
    ('1234567891', 'Bebbo', '77788811122'),
    ('1234567892', 'Bibboto', '55544466677'),
    ('1234567894', 'Bibboba', '55544466677');
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ insert.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Auf den ersten Blick sieht das Design komplizierter aus.

```sql
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Auf den ersten Blick sieht das Design komplizierter aus.
- Wir haben nun zwei Tabellen anstatt von einer.

```
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Auf den ersten Blick sieht das Design komplizierter aus.
- Wir haben nun zwei Tabellen anstatt von einer.
- Wollen wir die Namen der Studenten und Eltern zusammen haben, dann reicht ein einfaches `SELECT` nicht mehr aus.

```sql
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Auf den ersten Blick sieht das Design komplizierter aus.
- Wir haben nun zwei Tabellen anstatt von einer.
- Wollen wir die Namen der Studenten und Eltern zusammen haben, dann reicht ein einfaches `SELECT` nicht mehr aus.
- Stattdessen müssen wir die Daten mit einem `INNER JOIN` zusammensetzen.

```sql
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Auf den ersten Blick sieht das Design komplizierter aus.
- Wir haben nun zwei Tabellen anstatt von einer.
- Wollen wir die Namen der Studenten und Eltern zusammen haben, dann reicht ein einfaches `SELECT` nicht mehr aus.
- Stattdessen müssen wir die Daten mit einem `INNER JOIN` zusammensetzen.
- Der Vorteil ist die reduzierte Redundanz.

```sql
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Wir haben nun zwei Tabellen anstatt von einer.
- Wollen wir die Namen der Studenten und Eltern zusammen haben, dann reicht ein einfaches `SELECT` nicht mehr aus.
- Stattdessen müssen wir die Daten mit einem `INNER JOIN` zusammensetzen.
- Der Vorteil ist die reduzierte Redundanz.
- Die Namen jedes Elternteils müssen nur noch einmal eingegeben werden.

```sql
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Wollen wir die Namen der Studenten und Eltern zusammen haben, dann reicht ein einfaches `SELECT` nicht mehr aus.
- Stattdessen müssen wir die Daten mit einem `INNER JOIN` zusammensetzen.
- Der Vorteil ist die reduzierte Redundanz.
- Die Namen jedes Elternteils müssen nur noch einmal eingegeben werden.
- Sie könnten jetzt sagen: OK, aber wir müssen dafür die Mobiltelefonnummern zweimal eingeben.

```
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Stattdessen müssen wir die Daten mit einem `INNER JOIN` zusammensetzen.
- Der Vorteil ist die reduzierte Redundanz.
- Die Namen jedes Elternteils müssen nur noch einmal eingegeben werden.
- Sie könnten jetzt sagen: OK, aber wir müssen dafür die Mobiltelefonnummern zweimal eingeben.
- Das stimmt in diesem Fall, ist aber auch nur deshalb überhaupt erwähnenswert, weil wir im Grunde nur eine einzige Spalte „aussortiert“ haben.

```
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Der Vorteil ist die reduzierte Redundanz.
- Die Namen jedes Elternteils müssen nur noch einmal eingegeben werden.
- Sie könnten jetzt sagen: OK, aber wir müssen dafür die Mobiltelefonnummern zweimal eingeben.
- Das stimmt in diesem Fall, ist aber auch nur deshalb überhaupt erwähnenswert, weil wir im Grunde nur eine einzige Spalte „aussortiert“ haben.
- Oftmals betrifft das mehr als eine Spalte.

```
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Die Namen jedes Elternteils müssen nur noch einmal eingegeben werden.
- Sie könnten jetzt sagen: OK, aber wir müssen dafür die Mobiltelefonnummern zweimal eingeben.
- Das stimmt in diesem Fall, ist aber auch nur deshalb überhaupt erwähnenswert, weil wir im Grunde nur eine einzige Spalte „aussortiert“ haben.
- Oftmals betrifft das mehr als eine Spalte.
- Stellen Sie sich vor, dass wir noch die Adresse und Ausweisnummer der Eltern usw. gespeichert hätten.

```
/** Get the names of all parents of all the students. */

SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪: FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Auslesen

- Sie könnten jetzt sagen: OK, aber wir müssen dafür die Mobiltelefonnummern zweimal eingeben.
- Das stimmt in diesem Fall, ist aber auch nur deshalb überhaupt erwähnenswert, weil wir im Grunde nur eine einzige Spalte „aussortiert“ haben.
- Oftmals betrifft das mehr als eine Spalte.
- Stellen Sie sich vor, dass wir noch die Adresse und Ausweisnummer der Eltern usw. gespeichert hätten.
- Eine Reduzierung der Redundanz ist schon ein gutes Argument.

```
1 /** Get the names of all parents of all the students. */
2
3 SELECT student_name, parent_name FROM student
4     INNER JOIN parent ON student.parent_mobile = parent.parent_mobile;
```

```
1 $ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ select.sql
2 psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
3 # psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Nützlichkeit des neuen Designs wird klarer, wenn wir die Daten ändern müssen.


```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Nützlichkeit des neuen Designs wird klarer, wenn wir die Daten ändern müssen.
- Wir im ursprünglichen Beispiel hat der Sachbearbeiter nach ein paar Tagen herausgefunden, wie er ein ö machen kann.

```sql
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Nützlichkeit des neuen Designs wird klarer, wenn wir die Daten ändern müssen.
- Wir im ursprünglichen Beispiel hat der Sachbearbeiter nach ein paar Tagen herausgefunden, wie er ein ö machen kann.
- Die Daten können korrigiert werden.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Nützlichkeit des neuen Designs wird klarer, wenn wir die Daten ändern müssen.
- Wir im ursprünglichen Beispiel hat der Sachbearbeiter nach ein paar Tagen herausgefunden, wie er ein `ö` machen kann.
- Die Daten können korrigiert werden.
- Das `UPDATE`-Kommando wird nun auf die Tabelle `parent` angewandt.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Nützlichkeit des neuen Designs wird klarer, wenn wir die Daten ändern müssen.
- Wir im ursprünglichen Beispiel hat der Sachbearbeiter nach ein paar Tagen herausgefunden, wie er ein `ö` machen kann.
- Die Daten können korrigiert werden.
- Das `UPDATE`-Kommando wird nun auf die Tabelle `parent` angewandt.
- Es berührt nur eine Zeile, wie Sie im Ergebnis des `RETURNING`-Statements sehen.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Wir im ursprünglichen Beispiel hat der Sachbearbeiter nach ein paar Tagen herausgefunden, wie er ein ö machen kann.
- Die Daten können korrigiert werden.
- Das `UPDATE`-Kommando wird nun auf die Tabelle `parent` angewandt.
- Es berührt nur eine Zeile, wie Sie im Ergebnis des `RETURNING`-Statements sehen.
- Wir prüfen nun, ob die Namen des Elternteils von Herrn Bibbo, Herrn Bibboto und Frau Bibboba geändert wurden.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Die Daten können korrigiert werden.
- Das `UPDATE`-Kommando wird nun auf die Tabelle `parent` angewandt.
- Es berührt nur eine Zeile, wie Sie im Ergebnis des `RETURNING`-Statements sehen.
- Wir prüfen nun, ob die Namen des Elternteils von Herrn Bibbo, Herrn Bibboto und Frau Bibboba geändert wurden.
- Sie wurden.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Daten Ändern

- Das `UPDATE`-Kommando wird nun auf die Tabelle `parent` angewandt.
- Es berührt nur eine Zeile, wie Sie im Ergebnis des `RETURNING`-Statements sehen.
- Wir prüfen nun, ob die Namen des Elternteils von Herrn Bibbo, Herrn Bibboto und Frau Bibboba geändert wurden.
- Sie wurden.
- Die Update-Anomalie ist weg.

```
/** Find the parent Boddo and change his name to Böddö. */

-- Change the name of the parent with mobile 55544466677 to Böddö.
UPDATE parent SET parent_name = 'Böddö'
    WHERE parent_mobile = '55544466677'
    RETURNING parent_name, parent_mobile;

-- Get the names of all students whose parent has mobile 55544466677.
SELECT student_name, parent_name FROM student
    INNER JOIN parent ON student.parent_mobile = parent.parent_mobile
    WHERE student.parent_mobile = '55544466677';
```

```
$ psql "postgres://postgres:XXX@localhost/fixed" -v ON_ERROR_STOP=1 -ebf
    ↪ update.sql
psql: error: connection to server at "localhost" (::1), port 5432 failed
    ↪ : FATAL:  database "fixed" does not exist
# psql 16.11 failed with exit code 2.
```

Zusammenfassung

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \to B \to A$ habe.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.

Zusammenfassung

- Die 3NF ist wieder eine Methode, um Redundanz zu vermeiden.
- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.
- Wir könnten B als Primärschlüssel der neuen Tabelle benutzen und die entsprechenden Werte von A darin speichern.

Zusammenfassung

- Diesmal geht es um Redundanz, die entsteht wenn einige Spalten einer Tabelle von anderen Spalten abhängen, die keine Schlüssel sind.
- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.
- Wir könnten B als Primärschlüssel der neuen Tabelle benutzen und die entsprechenden Werte von A darin speichern.
- Die alten B-Spalten in der ursprünglichen Tabelle werden zum Fremdschlüssel.

Zusammenfassung

- Normalerweise sollte alle Information in einer Zeile einer Tabelle Information über den Schlüssel K sein.
- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.
- Wir könnten B als Primärschlüssel der neuen Tabelle benutzen und die entsprechenden Werte von A darin speichern.
- Die alten B-Spalten in der ursprünglichen Tabelle werden zum Fremdschlüssel.
- Auf diese Art haben wir die Redundanz der Daten reduziert und die Klarheit und Sauberkeit erhöht.

Zusammenfassung

- Sagen wir, dass es eine funktionale Abhängigkeit der Spalten A auf die Spalten B gibt und das diese transitiv auf den Schlüssel ist, also das wir $K \rightarrow B \rightarrow A$ habe.
- Dann gilt für jede Zeile mit dem Wert β von B, der selbe Wert α von A gespeichert wird.
- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.
- Wir könnten B als Primärschlüssel der neuen Tabelle benutzen und die entsprechenden Werte von A darin speichern.
- Die alten B-Spalten in der ursprünglichen Tabelle werden zum Fremdschlüssel.
- Auf diese Art haben wir die Redundanz der Daten reduziert und die Klarheit und Sauberkeit erhöht.
- Der Nachteil ist, dass wir ein `INNER JOIN` brauchen, wenn wir die Daten wieder zusammensetzen wollen um den Wert von A für ein bestimmtes K zu bekommen.

Zusammenfassung

- Der Wert von B bestimmt den Wert von A.
- Wenn der selbe Wert von B in mehreren Zeilen auftaucht, dann speichern wir auch den selben Wert von A mehrmals.
- Was wir nicht wirklich machen müssen, denn wir wissen den Wert von B, also kennen wir auch den von A.
- Um also die 3NF wieder herzustellen, können wir die Tabelle in zwei teilen.
- Wir könnten B als Primärschlüssel der neuen Tabelle benutzen und die entsprechenden Werte von A darin speichern.
- Die alten B-Spalten in der ursprünglichen Tabelle werden zum Fremdschlüssel.
- Auf diese Art haben wir die Redundanz der Daten reduziert und die Klarheit und Sauberkeit erhöht.
- Der Nachteil ist, dass wir ein `INNER JOIN` brauchen, wenn wir die Daten wieder zusammensetzen wollen um den Wert von A für ein bestimmtes K zu bekommen.
- Trotzdem ist das oft eine gute Idee.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[2] (siehe S. 168, 176).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[1] (siehe S. 168, 176).

[3] 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. 176, 177).

[4] 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. 176).

[5] 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. 177).

[6] 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. 175).

[7] 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. 176).

[8] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 176).

[9] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 175).

[10] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 176, 177).

[11] 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. 177).

References II

[12] 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. 177).

[13] Edgar Frank „Ted“ Codd. “Normalized Data Base Structure: A Brief Tutorial”. In: *ACM SIGFIDET Workshop on Data Description, Access, and Control*. 11.–12. Nov. 1971, San Diego, CA, USA. Hrsg. von Edgar Frank „Ted“ Codd und Albert L. Dean Jr. New York, NY, USA: Association for Computing Machinery (ACM), 1971, S. 1–17. ISBN: 978-1-4503-7300-5. doi:10.1145/1734714.1734716. See also[14] (siehe S. 5–8, 175).

[14] Edgar Frank „Ted“ Codd. *Normalized Data Base Structure: A Brief Tutorial*. IBM Research Report RJ935. San Jose, CA, USA: IBM Research Laboratory, 1971. URL: https://www.fsmwarden.com/Codd/Normalized%20data%20base%20structure_%20a%20brief%20tutorial(1971,%20nov).pdf (besucht am 2025-05-04) (siehe S. 169).

[15] 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. 175, 177).

[16] Edgar Frank „Ted“ Codd. *Further Normalization of the Data Base Relational Model*. IBM Research Report RJ909. San Jose, CA, USA: IBM Research Laboratory, 31. Aug. 1971. URL: https://forum.thethirdmanifesto.com/wp-content/uploads/asgarosforum/987737/00-efc-further-normalization.pdf (besucht am 2025-05-04). Reprinted in and presented at[46] (siehe S. 5–8, 175).

[17] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 177).

[18] Christopher J. Date. *An Introduction to Database Systems*. 8. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Juli 2003. ISBN: 978-0-321-19784-9 (siehe S. 5–8, 175, 176).

[19] 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. 177).

References III

[20] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 177).

[21] 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. 177).

[22] 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. 176).

[23] Ramez Elmasri und Shamkant Navathe. *Fundamentals of Database Systems*. 7. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Juni 2015. ISBN: 978-0-13-397077-7 (siehe S. 5–8, 175, 176).

[24] 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. 176).

[25] 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. 177).

[26] 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. 177).

[27] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 176).

[28] 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. 176, 177).

[29] 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. 176).

References IV

[30] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 177).

[31] William (Bill) Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory". *Communications of the ACM (CACM)* 26(2):120–125, Sep. 1982–Feb. 1983. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/358024.358054. URL: https://www.cs.dartmouth.edu/~cs61/Resources/Papers/CACM%20Kent%20Five%20Normal%20Forms.pdf (besucht am 2025-05-03) (siehe S. 5–8, 175, 176).

[32] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 177).

[33] 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. 176).

[34] 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. 175).

[35] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 176).

[36] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 176).

[37] 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. 177).

[38] 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. 175).

References V

[39] 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. 176).

[40] 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. 175).

[41] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 176).

[42] 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. 175).

[43] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 176).

[44] 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. 176).

[45] 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. 175).

[46] Randall Rustin, Hrsg. *Data Base Systems: Courant Computer Science Symposium 6*. 24.–25. Mai 1971, New York, NY, USA. Englewood Cliffs, NJ, USA: Prentice-Hall, 1972. ISBN: 978-0-13-196741-0 (siehe S. 169).

[47] Heinz Schweppe und Manuel Scholz. "Schema Definition with SQL / DDL (II)". In: *Einführung in die Datenbanksysteme. Datenbanken für die Bioinformatik*. Berlin, Germany: Freie Universität Berlin, Apr.–Okt. 2005. Kap. 4. URL: https://www.inf.fu-berlin.de/lehre/SS05/19517-V/FolienEtc/dbs05-06-DDLSQL-2-2.pdf (besucht am 2025-05-15) (siehe S. 5–10, 177).

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

References VI

[49] Yuriy Shamshin. "Normalization". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 07a. URL: https://dbs.academy.lv/lection/dbs_LS07ENa_normalization.pdf (besucht am 2025-05-03) (siehe S. 176).

[50] Yuriy Shamshin. "RDM Normalization. Data Anomalies. Functional Dependency. Normal Forms.". In: *Databases*. Riga, Latvia: ISMA University of Applied Sciences, Mai 2024. Kap. 07. URL: https://dbs.academy.lv/lection/dbs_LS07EN_normalization.pdf (besucht am 2025-05-03) (siehe S. 176).

[51] 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. 176).

[52] 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. 177).

[53] "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. 177).

[54] 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. 173, 177).

[55] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of[54] (siehe S. 177).

[56] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 177).

[57] 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. 176).

References VII

[58] 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. 176).

[59] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 176).

[60] 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. 175, 177).

[61] 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. 176).

[62] *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. 177).

[63] 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. 176).

[64] 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. 175).

[65] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 175).

Glossary (in English) I

1NF The first normal form (NF) in relational databases (DBs)[15,18,23,31].

2NF The second normal form (NF) in relational DBs[13,16,18,23,31].

3NF The third normal form (NF) in relational DBs[13,16,18,23,31].

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[9,38,65]. 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[6,34,40,42,45]. 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 DB software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[60].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[64].

DBS A *database system* is the combination of a DB and a the corresponding DBMS, i.e., basically, an installation of a DBMS on a computer together with one or multiple DBs. DBS = DB + DBMS.

FD A *functional dependency* exists between two groups of attributes Y and X if the values of X determine the values of Y. This is written as $X \to Y$.

Glossary (in English) II

IT	information technology
LAMP Stack	A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[10,28].
Linux	is the leading open source operating system, i.e., a free alternative for Microsoft Windows[3,27,51,58,59]. 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[1,2,4,22,36,43]. See https://mariadb.org for more information.
Microsoft Windows	is a commercial proprietary operating system[8]. 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[7,22,44,57,63]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.
NF	The *normal forms* define guidelines for the design of relational DBs with the goal to avoid redundancy and to prevent inconsistencies and anomalies[18,23,31,49,50]. There are several normal forms, first normal form (1NF), second normal form (2NF), third normal form (3NF), and so on, each more restrictive than the other.
PostgreSQL	An open source object-relational DBMS[24,39,41,57]. See https://postgresql.org for more information.
psql	is the client program used to access the PostgreSQL DBMS server.
Python	The Python programming language[29,33,35,61], i.e., what you will learn about in our book[61]. Learn more at https://python.org.

Glossary (in English) III

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[15,25,26,52,56,60,62].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers[10] 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“[32].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[11,17,19,20,30,37,53–56]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[53].

terminal A terminal is a text-based window where you can enter commands and execute them[3,12]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux[12,28]. 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.

WeChat 微信, produced by Tencent (腾讯公司) is the most-used messenger application in China. It integrates payment capabilities, video and voice chats, as well as social plugins and location-based services. See also https://weixin.qq.com/.

WWW World Wide Web[5,21]

$\Sigma(R)$ the relation schema of relation R[47].