1 Eindeutigkeit von Werten (Primary Key)

In der Tabelle „Adresse“ haben wir ein Feld für die ID jedes Datensatzes vorgesehen, damit wir anhand dieser ID jeden Datensatz eindeutig identifizieren können. Den Datensatz mit der ID = 1 haben wir bereits geschrieben. Wenn wir nun jedoch folgendes Statement ausführen:

```
INSERT INTO Adresse SET
ID = 1,
Bundesland = 'Schleswig-Holstein',
PLZ = '24848',
Ort = 'Klein Bennebek',
Strasse = 'Wildanger',
Hausnr = '31a',
Typ = 'B';
```

sehen wir, dass die wir die ID = 1 zweimal belegen können, was dem eigentlichen Sinn dieses Feldes entgegensteht. Um die Datenbank dazu zu bringen, solche Doppeltbelegungen zu verhindern, müssen wir die ID als Primärschlüssel deklarieren. Wir Entfernen die Tabelle nochmal und erzeugen sie erneut, wobei wir die Zeile für die ID mit der Primärschlüseldefinition ergänzen:

```
CREATE TABLE Adresse (
  ID INT PRIMARY KEY,
  …
);
```

Die „NOT NULL“ Direktive benötigen wir nicht mehr, da Primärschlüssel immer automatisch not nullable sind. Wenn wir nun den Datensatz aus dem „Create Table“ Dokument nochmal setzen und danach das oben genannte Statement ausführen, erhalten wir folgende Fehlermeldung:

```

```

Eine zweite Möglichkeit, den Primärschlüssel zu hinterlegen ist, es am Ende des CREATE TABLE Statements anzuhängen:

```
CREATE TABLE Adresse (
  ID INT,
  …
  Typ CHAR(1) NOT NULL DEFAULT "B"
  PRIMARY KEY(ID)
);
```

Es ist auch möglich, mehrere Felder als kombinierten Primärschlüssel zu realisieren. Hierbei ist nur die Option mit der eigenen PRIMARY KEY Zeile möglich:

```
CREATE TABLE Adresse (
  ID1 INT,
  ID2 INT,
  …
  Typ CHAR(1) NOT NULL DEFAULT "B",
  PRIMARY KEY(ID1, ID2)
);
```

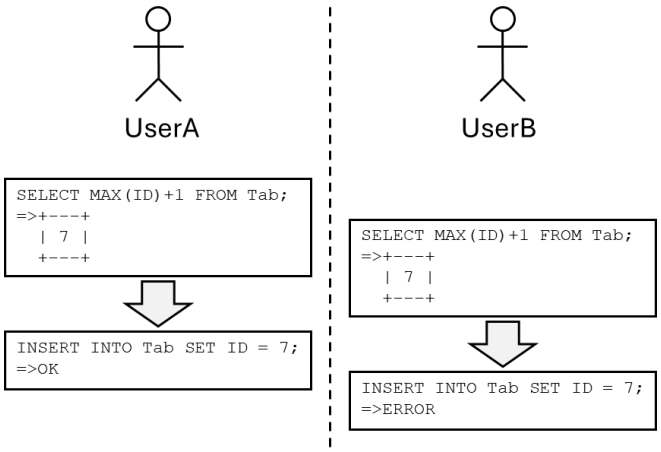
Folgende Kombinationen für ID1/ID2 sind parallel möglich 1/1, 1/2, 2/1, 2/2. Jedoch wäre es nicht möglich 1/1 doppelt zu hinterlegen.

2 AUTO_INCREMET

Wenn wir nun einen neuen Datensatz erstellen müssen wir also sicherstellen, dass die ID des neuen Datensatzes nicht bereits existiert. Man könnte bspw. folgendes Statement nutzen:

```
SELECT MAX(ID) + 1 FROM Adresse;
```

Da Datenbanken aber im Multiluser-Betrieb laufen müssen, könnte folgendes passieren:
User A prüft die nächste mögliche ID und erhält bspw. die 7. User B macht das Gleiche und erhält somit auch die 7. Wenn User A genau nach der Prüfung von User B seinen Datensatz mit der ID 7 schreibt und danach erst User B schreibt, so erhält er einen Duplicate Key Fehler. Dieses Problem würde man „Race Condition“ nennen.



Das ist sehr unpraktisch, weshalb wir eine bessere Lösung benötigen, um eindeutige Keys zu generieren. Hierbei gibt es im Wesentlichen zwei Lösungen.

1. UUID: Es wird ein Wert erzeugt, der weltweit einzigartig ist. MySQL unterstützt hier die UUID (Universal Unique Identifier), oder auch manchmal GUID (Global Unique Identifier) genannt. Hier wird mittels Zufallszahlen, Hardwareinformationen und Zeitstempel ein Wert erzeugt, der weltweit nur einmal existiert.

Geben Sie folgendes Statement ein:

```
SELECT UUID();
```

Wie sie sehen, ist dies ein sehr langer, mit Bindestrichen getrennter Hexadezimalwert. Der Vorteil ist, dass dieser Key über mehrere Systeme garantiert eindeutig ist. Der Nachteil ist, dass wir hier Textinformationen (in `CHAR` oder `BINARY`) ablegen müssen, was aus Performance und Platzgründen ungünstiger als `INT` ist.

2. AUTO_INCREMENT: Hier wird pro Tabelle auf dem Primärschlüsselfeld ein Merker erzeugt, der sich den historisch höchsten Wert merkt, der in der Spalte existiert hat – egal ob bereits gelöscht oder noch vorhanden. Wird ein neuer Datensatz erzeugt, ohne dass die Spalte angegeben wurde, wird dieser Wert genommen und um 1 erhöht. Das ist der neue, automatisch erzeugte Schlüssel. Die Datenbank macht also das gleiche wir im obigen Beispiel. Der Unterschied liegt aber darin, dass es in einer atomaren Aktion erfolgt, wodurch wir keine Race Condition erhalten.

Erstellen Sie nun die Adresstabelle neu, diesmal aber mit einem AUTO_INCREMENT Feld:

```
CREATE TABLE Adresse (
  ID INT PRIMARY KEY AUTO_INCREMENT,
  …
);
```

Schreiben Sie nun die beiden Datensätze und prüfen mit SELECT * FROM Adresse; die Werte:

```
INSERT INTO Adresse (Bundesland, PLZ, Ort, Strasse, Hausnr, Typ) VALUES
('Mecklenburg-Vorpommern', '19205', 'Holdorf', 'Clemensstr.', '74a', 'B'),
('Schleswig-Holstein', '24848', 'Klein Bennebek', 'Wildanger', '31a', 'B')
;
```

Mit `SELECT * FROM Adresse;` können sie die Werte prüfen. Welche Werte finden Sie für die IDs vor:

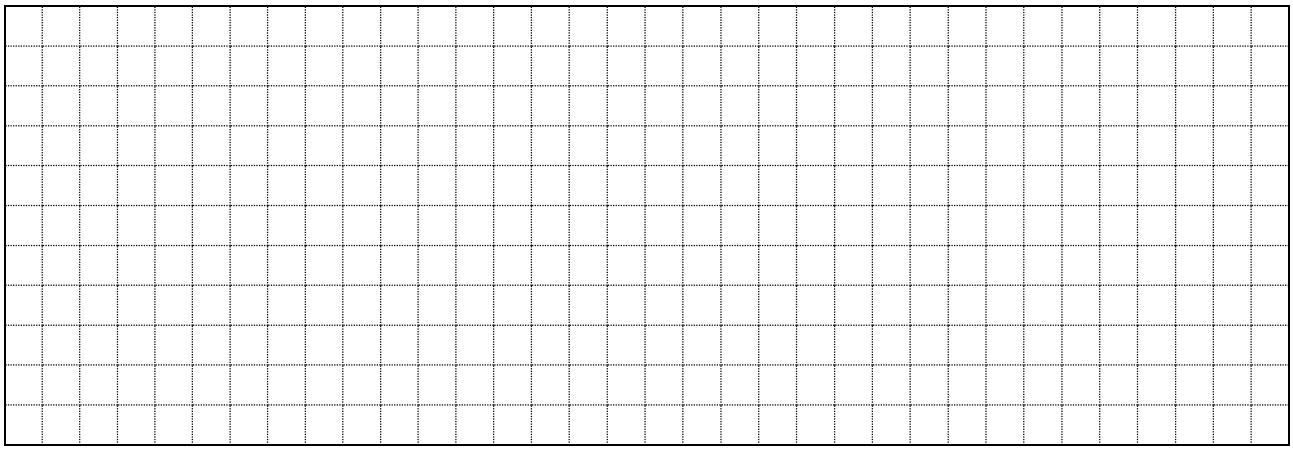
1. Datensatz:		2. Datensatz:	

Löschen sie nun die Werte mit `DELETE FROM Adresse;` schreiben sie sie wieder neu und prüfen sie wieder.

1. Datensatz:		2. Datensatz:	

3 Fremdschlüssel

Zwischen dem Kunden und der Adresse haben wir eine 1:n Beziehung. Zeichnen sie das Datenmodell:



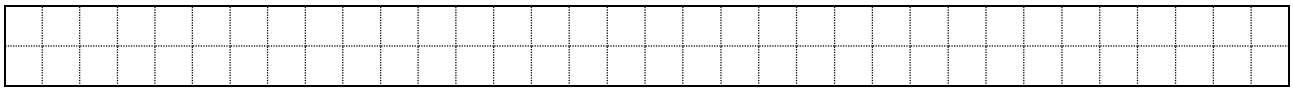
Wir benötigen also in der Adresstabelle eine Referenz auf den zugehörigen Kunden, was wir mit der KID realisiert haben. Wenn wir allerdings dieses Feld in der Tabelle ohne weitere Vorkehrungen anlegen:

```
CREATE TABLE Adresse (
  ID INT PRIMARY KEY AUTO_INCREMENT,
  …
  KID INT,
  …
);
```

werden wir ein potentielles Problem bekommen. Geben wir beispielsweise folgenden Datensatz in die Adresstabelle ein:

```
INSERT INTO Adresse (Bundesland, PLZ, Ort, Strasse, Hausnr, KID, Typ)
 VALUES
('Sachsen', '01936', 'Königsbrück', 'Rosenheimer Str.', '10', 500, 'B');
```

so existiert der Kunde mit der ID = 500 nicht in der Kundentabelle. Dies würden wir als eine Verletzung der



bezeichnen. Wenn ein Feld einer Tabelle auf ein Primärschlüsselfeld einer anderen Tabelle verweist, muss der Wert der verweisenden Tabelle als Primärschlüsselwert in der anderen Tabelle existieren! Verletzungen der referenziellen Integrität können durch folgende Aktionen entstehen:

- Fehlerhaftes INSERT in der referenzierenden Tabelle
- Fehlerhaftes UPDATE in der referenzierenden oder referenzierten Tabelle
- Löschen eines Datensatzes (oder der ganzen Tabelle) in der referenzierten Tabelle

Um dies nun zu unterbinden, können wir das verweisende Feld als „Fremdschlüssel“ oder „Foreign Key“ definieren. Dazu müssen wir aber erst die Kundentabelle mit einem Primärschlüssel versehen. Erzeugen Sie also die Kundentabelle aus dem Dokument „AEuP_DB1_FirstTable_Schueler“ neu, setzen die ID aber als Primärschlüssel um. Dann schreiben sie die ersten beiden Adressdatensätze neu. Jetzt erzeugen sie die Adresstabelle neu, mit einem Foreign Key Feld:

```
CREATE TABLE Adresse (
  ID INT PRIMARY KEY AUTO_INCREMENT,
  …
  KID INT,
  FOREIGN KEY (KID) REFERENCES Kunde(ID)
);
```

Versuchen sie nun den Datensatz mit der KundenID = 500 nochmal zu schreiben.

Welche Fehlermeldung erhalten sie:

Durch den Verweis der Adresstabelle auf die Kundentabelle ist nun jedoch folgendes bei den DDL-Skripten zu beachten:

Bei der Erzeugung unserer Tabellen muss:

Bei der Löschung unserer Tabellen muss:

Nun gibt es noch eine Möglichkeit, Änderungen oder Löschungen bei der referenzierten Tabelle auf die referenzierende Tabelle zu kaskadieren. Erstellen sie die Adresstabelle neu, mit folgender Ergänzung:

```
CREATE TABLE Adresse (
  ID INT PRIMARY KEY AUTO_INCREMENT,
  …
  KID INT,
  FOREIGN KEY (KID) REFERENCES Kunde(ID)
  ON UPDATE CASCADE
  ON DELETE CASCADE
);
```

Danach schreiben sie folgende beiden Datensätze in der Adresstabelle neu (unter der Voraussetzung, dass die beiden Kundendatensätze von Toni Kraft und Lio Karger existieren – wenn nicht, erzeugen Sie sie neu).

```
INSERT INTO Adresse (ID, Bundesland, PLZ, Ort, Strasse, Hausnr, KID, Typ)
VALUES
(1, 'Mecklenburg-Vorpommern', '19205', 'Holdorf', 'Clemensstr.', '74a',
 1, 'B'),
(2, 'Schleswig-Holstein', '24848', 'Klein Bennebek', 'Wildanger', '31a',
 2, 'B');
```

Ändern Sie nun in der Kundentabelle die ID des ersten Datensatzes:

```
UPDATE Kunde SET ID = 500 WHERE ID = 1;
```

Löschen Sie anschließend den zweiten Datensatz in der Kundentabelle:

```
DELETE FROM Kunde WHERE ID = 2;
```

Was beobachten sie in der Adresstabelle beim ersten Datensatz:

Was beobachten sie in der Adresstabelle beim zweiten Datensatz: