Datenbanken, SQL

&

Python

eine Einführung
Version 0.3

Zusammengestellt von Markus Signitzer
September 2023
HTL Anichstraße

Inhaltsverzeichnis

1 Vorwort

Hallo Leute!
Anbei ein Versuch von mir die grundlegenden Aspekte von relationalen Datenbanken und SQL am Beispiel einer MariaDB zu erklären. Absolut kein Anspruch auf Vollständigkeit!
Update 2023 Version 0.3 hat ein neues Kapitel in dem erklärt wird wie man mit Python auf eine Datenbank zugreift.
Meine Quellen sind:

- https://de.wikipedia.org/wiki/Relationale_Datenbank
- https://www.datenbank-grundlagen.de
- offizielles MySQL-Referenzhandbuch
- Unterlagen vom freundlichen Prof. Albert Greinöcker

Viel Spaß!
Markus Signitzer

2 LAMP-Installation

Eine Voraussetzung für das Arbeiten mit SQL, HTML und PHP auf einem Linux System ist eine LAMP Installation. LAMP steht für Linux Apache MySQL PHP.
Wobei seit MySQL von Oracel übernommen wurde vermehrt MariaDB verwendet wird. Zum Installieren:

```
sudo apt-get install apache2 libapache2-mod-php php php-gd mariadb-server phpmyadmin
```

Wobei wir phpmyadmin aus Komfortgründen auch noch mit-installiern.
Wenn bei der Installation die Frage nach einer automatischen Datenbank-Konfiguration mit Hilfe von **dbconfig-common** gestellt wird kann man mit JA antworten.
Es kann dann ein password erstellt werden, mit dem sich phpmyadmin bei der Datenbank anmelden kann.

2.1 Apache Konfiguration

- **Nur auf Anfragen vom localhost reagieren:**

  ```
  sudo nano /etc/apache2/ports.conf
  ```

  Der ursprüngliche Eintrag Listen 80 auf Listen 127.0.0.1:80 ändern.

- **Dateizugriff auf /var/www/html herstellen:**
  Der Ordner /var/www/html ist der, auf den http://localhost/ verweist, und in dem die eigenen Webdateien liegen werden.
  Der Benutzer von Apache (www-data) und auch wir selber brauchen deshalb Schreibrechte für /var/www/html (gehört momentan noch root weil es bei der installtion mit sudo angelegt wurde).
  Dies kann man aber ändern, indem man zunächst der Gruppe www-data beitritt:

  ```
  sudo adduser <Benutzer> www-data && newgrp www-data
  ```

  Der Platzhalter `<Benutzer>` ist durch den eigenen Benutzernamen zu ersetzen. Anschließend werden Besitzer und Gruppe von /var/www/html von root auf www-data geändert und der Gruppe www-data Schreibrechte auf dieses Verzeichnis gewährt, damit die eigenen Produktionsdateien dorthin kopiert werden können:

  ```
  sudo chown www-data:www-data /var/www/html && sudo chmod 775 /var/www/html
  ```

- **phpmyadmin auf apache2 zuslassen:**

  ```
  sudo nano /etc/apache2/apache2.conf
  ```

  und folgende Zeile ganz am Ende vom File hinzufügen:

  ```
  include /etc/phpmyadmin/apache.conf
  ```

- **phpmyadmin user einrichten:**
  +++++ ACHTUNG +++++
  Wenn man während der Installation die Hilfe des **dbconfig-common** Scripts verwendet hat sind die folgenden Punkte nicht nötig!
  Man kann in diesem Fall überprüfen ob ein User für phpmyadmin angelegt wurde und ob er die entsprechenden Rechte hat:
  Zuerst in die Datenbank einsteigen mittels:
  sudo mysql -u root -p
  (momentan kein root PW also einfach ENTER drücken bei der Passwortabfrage)
  dann das folgende SQL-Statement eingeben:
  SELECT User FROM mysql.user;
  sollte einen phpmyadmin user zeigen und
  show grants for 'phpmyadmin'@'localhost';
  sollte die Wörter GRANT ALL PRIVILEGES enthalten.
  Aus dem Datenbank-Interface kann man dann mittels **quit** aussteigen.
  ++++++++++++++++++++
  phpmyadmin lässt keinen root user login mehr zu, deshalb muss man einen eigenen User für phpmyadmin einrichten. Dazu mariaDB starten mit: sudo mysql -u root -p
  Info: da noch kein root password für die Datenbank konfiguriert wurde, könne wir bei der Passwortabfrage einfach ENTER drücken.

  ```
  CREATE USER 'masi'@'localhost' IDENTIFIED BY 'password123';
  GRANT ALL PRIVILEGES ON *.* TO 'masi'@'localhost' WITH GRANT OPTION;
  FLUSH PRIVILEGES;
  ```

- Anschließend den Apache-Server neu starten:

  ```
  sudo service apache2 restart
  ```

2.1.1 Installation Testen

Um zu überprüfen ob die Installation erfolgreich war genügen ein paar einfache Tests.

- **Apache testen:**
  Einfach einen Webbrowser öffnen und in der URL-Zeile `http://localhost` eingeben. Man sollte eine Apache2 Default Page sehen in der auch Informationen zu den Apache Konfigurationsfiles sind.

- **PHP testen:**
  Ein kleines php file erstellen, welches phpinfo() aufruft:

  ```
  nano /var/www/html/myPHPinfo.php
  ```

  und folgenden Text eintragen:

  ```
  <?php
  phpinfo();
  ?>
  ```

  Anschließend im Browser aufrufen mittels: localhost/myPHPinfo.php

- **MySQL testen:** In der Bash mittels: sudo mysql -u root -p den MySQL Zugang testen (momentan kein root PW also einfach ENTER drücken bei der Passwortabfrage)

- **phpmyadmin testen:** Im Browser die URL localhost/phpmyadmin/ aufrufen und mit dem oben angelegten user (masi oder phpmyadmin) einloggen.

2.2 OPTIONAL: MariaDB Installation absichern

Das der mariaDB root user kein PW hinterlegt hat ist für unsere Testzwecke okay, aber für einen richtigen Webserver nicht akzeptable. Um die DB abzusichern und ein PW für den root user anzulegen können noch folgende Schritte ausgeführt werden:

```
sudo mysql -u root -p
```

Einloggen noch ohne PW, einfach ENTER drücken bei der PW-Abfrage.

```
use mysql;
update user set plugin='' where User ='root';
flush privileges;
exit
```

Restart der DB

```
sudo systemctl restart mariadb
```

Abschliessend mit Hilfe eines Skriptes die DB-Installation absichern (Bei allen Fragen mit Y (Yes) antworten und ein neues root PW vergeben)!

```
sudo mysql_secure_installation
```

3 Datenbanksystem (DBS)

Ein DBS setzt sich aus 2 Teilen zusammen:

1. Datenbank-Managementsystem (DBMS)
   - Verwaltet die Datenbanken und deren Zugriff darauf
   - Komplexe Software, die von den Inneren Vorgängen der DB abstrahiert und den Zugriff somit moglichst leicht gestaltet
   - Zur Verwaltung wird meist eine standardisierte Sprache (z.B. SQL (Structured Query Language)) verwendet
   - Es werden in der Regel mehrere Datenbanken verwaltet

2. Datenbank(en) (DB)
   Ein logisch zusammengehöriger Datenbestand (z.B. Personaldaten, Messwerte von Wetterstationen, Produkte, ...)

3.1 Wichtige Prinzipien für Datenbanksysteme

- Daten werden **persistent** (über einen längeren Zeitraum) gespeichert
- Eine **effiziente** Verwaltung wird angestrebt
  -Schnelle Ausführung
  -Einfache Handhabung
- Die Datenbank muss immer in einem **konsistenten** Zustand sein!
  Transaktionsprinzip: Zusammengehörende Aktionen werden entweder ganz oder gar nicht durchgeführt (Bsp.: Banküberweisung)
- Vermeidung von **Redundanz**
- **Datenschutz**! Nur Berechtigte sollen auf die für sie vorgesehenen Daten in für sie vorgesehener Form (lesen, schreiben) zugreifen.
- **Datensicherheit** Es sollen keine Daten verloren gehen

4 MySQL vs MariaDB

MySQL und MariaDB sind **Datenbankmanagementsystem**.

MySQL: wurde ursprünglich von von einer schwedischen Firma (MySQL AB: David Axmark, Allan Larsson, und Michael "Monty" Widenius) programmiert und in 1995 released. 2008 kaufte Sun Microsystems MySQL AB, und in 2010 wurde Sun Microsystems von Oracle gekauft → MySQL wird momentan von Oracle gewartet und weiterentwickelt.

MariaDB: Da Oracle aber selbst schon eine eigene (ClosedSource) Datenbanklösung hatte, befürchteten viele Entwickler in der OpenSource-Community, das Oracle MySQL "sterben lassen" könnte. Aus diesem Grund gründete Michael "Monty" Widenius zusammen mit anderen **MariaDB als Fork von MySQL**.

Seit diesem Zeitpunkt existieren beide Systeme und pushen sich gegenseitig mit neuen Entwicklungen. MariaDB verwendet die gleiche SQL Syntax wie MySQL und ist (bis jetzt) zu 100% kompatible.
Verwendet werden beide, z.B MySQL von GitHub, NASA, Tesla, Netfilx und MariaDB von Google, Wikipedia, Craigslist und vielen Linux-Distros.

p.s. My und Maria sind die Vornamen von Widenius Töchtern :)

Weitere bekannte DBMS: SQLite, Oracle, DB2 (IBM), Access, MS SQL Server, PostgreSQL, Sybase, SAP MaxDB, ...

5 Relationale Datenbanken

Das relationale Datenbankmodell ist heutzutage das verbreitetste Modell in der Datenbankentwicklung. Im Mittelpunkt steht hier die namensgebende Relation, die mathematisch gesehen nichts anderes als eine

Beschreibung für eine Tabelle ist. Durch die relationale Algebra können Operationen auf diese Relation durchgeführt werden. Erstmals vorgeschlagen wurde es im Jahr 1970 vom britischen Mathematiker und Datenbanktheoretiker Edgar F. Codd. Und auch wenn es einige Kritikpunkte beinhaltet, ist es dennoch bis heute der etablierte Standard für Datenbanken.

Eine relationale Datenbank besteht aus einer oder mehreren **Tabellen (Relationen)**, in denen Daten abgespeichert werden können. Und zwar in der Form, dass jede **Zeile (Tupel)** einen eigenen **Datensatz (Record)** darstellt. Die **Spalten** jeder Zeilen sind **Attribute**, also Eigenschaften der abgespeicherten Daten. Eine **Domäne (Domain)** ist dabei der Wertebereich, den die Attribute annehmen könne. Durch ein Relationenschema lassen sich sowohl die Anzahl als auch die Typen der Attributen einer Relation festlegen.
Siehe Abbildung 1 auf Seite 7.

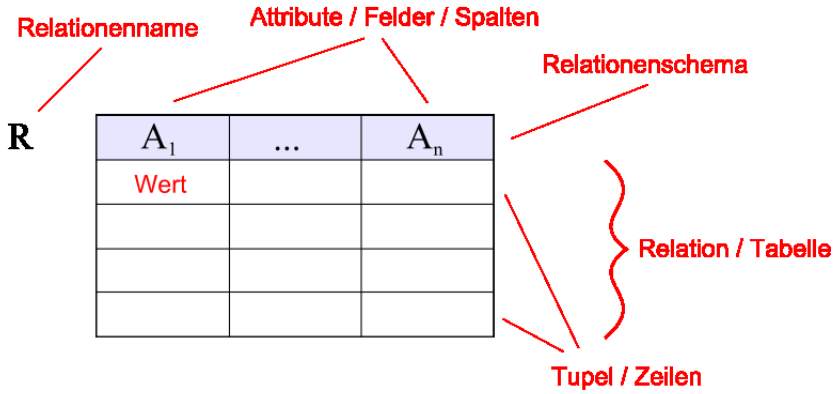


Abbildung 1: Begriffe relationaler Datenbanken, ©Wikipedia

Ein Datensatz muss eindeutig identifizierbar sein! Dies wird mittels eines oder mehreren **Schlüssel (Keys)** sichergestellt. Über den Schlüssel soll aber nicht nur die Identifizierung des Datensatzs sichergestellt werden, eine weitere Anforderung ist, dass der Schlüssel auch **minimal (einzigartig)** ist. Innerhalb der Tabelle ist dies dann der **Primärschlüssel**.Existieren Verknüpfungen (Beziehungen) mit anderen Tabellen, wird dieser Primärschlüssel in der anderen Tabelle als **Fremdschlüssel** bezeichnet.
Siehe Abbildung 2 auf Seite 8.

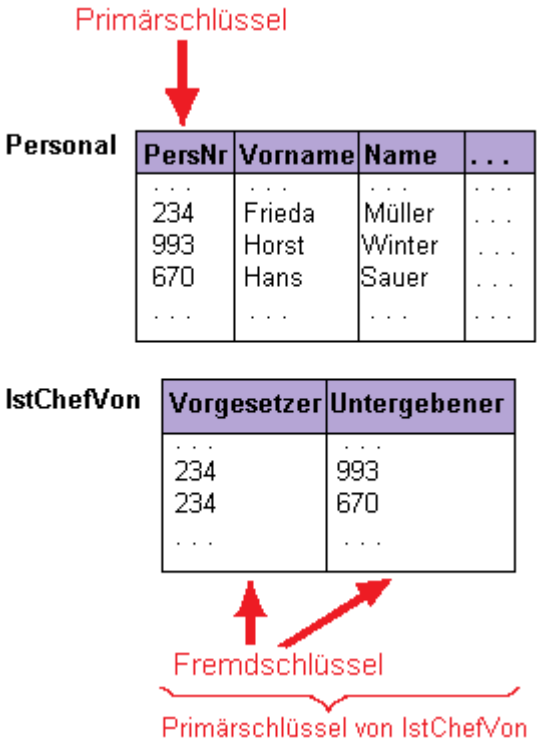


Abbildung 2: Primärschlüssel vs Fremdschlüssel, ©Wikipedia

5.1 Schritte zur Erstellung einer Datenbank

1. **Realität**
   In einer Datenbanklösung soll meist ein Ausschnitt aus der Realität abgebildet werden.
   Ziel ist es, gleich wie bei der Softwareerstellung, die Realität so abzubilden, wie es für das technische Vorhaben am Sinnvollsten erscheint.

2. **Spezifikation**
   Es wird eine Abstraktion der Realität vorgenommen.
   Hier sollen die Anforderungen informal beschrieben werde.

3. **Konzeptionelles** Datenmodell
   Mit Modellierungswerkzeugen soll die Spezifikation standardisiert und anschaulich dargestellt werden.
   Es wird festgelegt welche Daten abgelegt werden sollen und wie diese zueinander in Beziehung stehen.
   Mögliche Modellierungsarten: **Entity Relationship** (ER) - Modell oder UML (Unified Markup Language)

4. **Logisches** Datenmodell
   Das Modell wird um datentechnische Angaben erweitert (Feldnamen, Datentypen).
   Das Modell wird in der dem konkreten DBMS-Typ vorgegebenen Form abgelegt. In einem relationalen DBMS wären das z.B. Tabellen.

5. **Physisches** Datenmodell
   Konkrete Umsetzung in einem DBMS-System
   Die Form wie die Daten tatsächlich am Datenträger gespeichert werden

6. **Inbetriebnahme und Wartung**

5.2 Datennmodellierung

Ein Ausschnitt aus der Realität soll abstrahiert (entsprechend) und standardisiert (idealerweise grafisch) dargestellt werden.

ER (Entity Relationship) -Modell
ist das meist verwendete Modell. Aber es kann auch **UML** (Unified Modeling Language) verwendet werden, wobei in diesem Fall die Klassendiagramme 'umfunktioniert' werden: Entities werden zu Klassen, Attribute direkt in die Klasse geschrieben und Relationships werden zu qualifizierten Assoziationen.

Sehr einfaches Werkzeug für die Unterstützung der Modellierung:
Dia (Windows: http://dia-installer.de/; oder https://projects.gnome.org/dia/)

5.2.1 ER-Modell

Grundidee:

1. **Entity-Menge** (Klasse) bzw. **Entity** (Objekt):
   - Hier sollen gleichartige Exemplare (Dinge, Personen, Begriffe) zusammengefasst werden.
   - Darstellung als Rechteck mit dem Namen der Entitätsmenge.
2. **Attribute**
   - Beschreiben Eigenschaften, die allen Entitäten gemeinsam ist
   - Werden als Oval oder Kreis dargestellt und mit der Entität verbunden
   - Attribute können ein Primärschlüssel (ein pro Entity eindeutiger Wert) sein. Dieser wird unterstrichen.
3. **Relationships**
   - Stellen die Beziehungen zwischen den Entitäten dar.
   - Werden als Raute dargestellt. - Haben eine **Kardinalität** z.B. 1:1; 1:n; n:m
   Kardinalität besagt, wie viele Entities eines beteiligten Entitätstyps mit wie vielen Entities der anderen beteiligten Entitätstypen in Beziehung treten können.

Siehe Abbildung 3 auf Seite 9 für Beispiele:

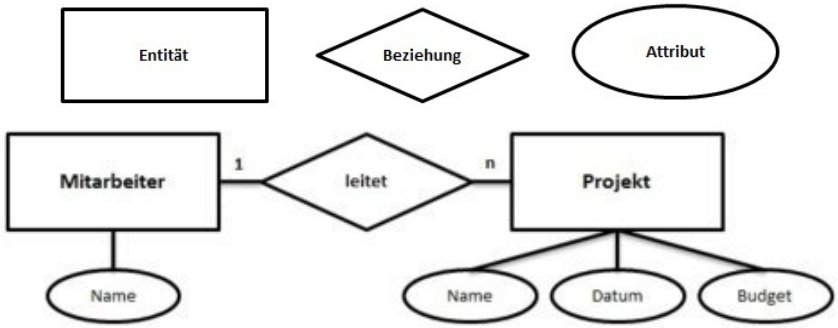


Abbildung 3: Einfaches ER-Modell, ©http://www.datenbanken-verstehen.de

5.3 Vorgehensweise beim ER-Modell

Beispielslides von Albert Greinöcker:

ER-Modell in ein relationales Modell überführen

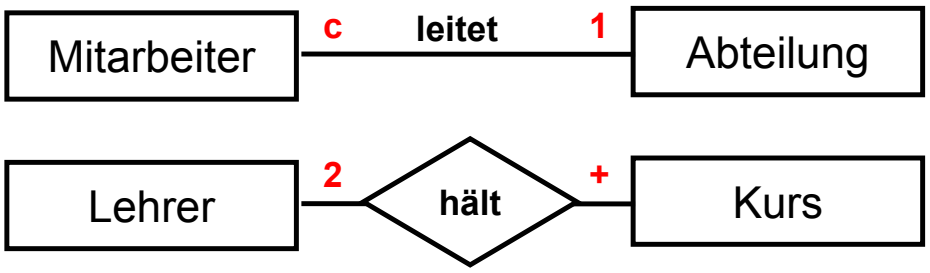
BINF

Vorgehen bei der ER-Modellierung

1. Problemrahmen abstecken
   - Aufgabenstellung lesen
2. Festlegen der Entities
   - Substantive raussuchen (Auto, Wähler)
3. Festlegen der Beziehungstypen
   - Verben raussuchen (hat, wählt) => können Beziehungen werden
4. Festlegen der Kardinalitäten
   - Mengenangaben raussuchen
5. Festlegen der Attribute, Wertebereiche und Schlüssel

Kardinalitäten – erweiterte Notation

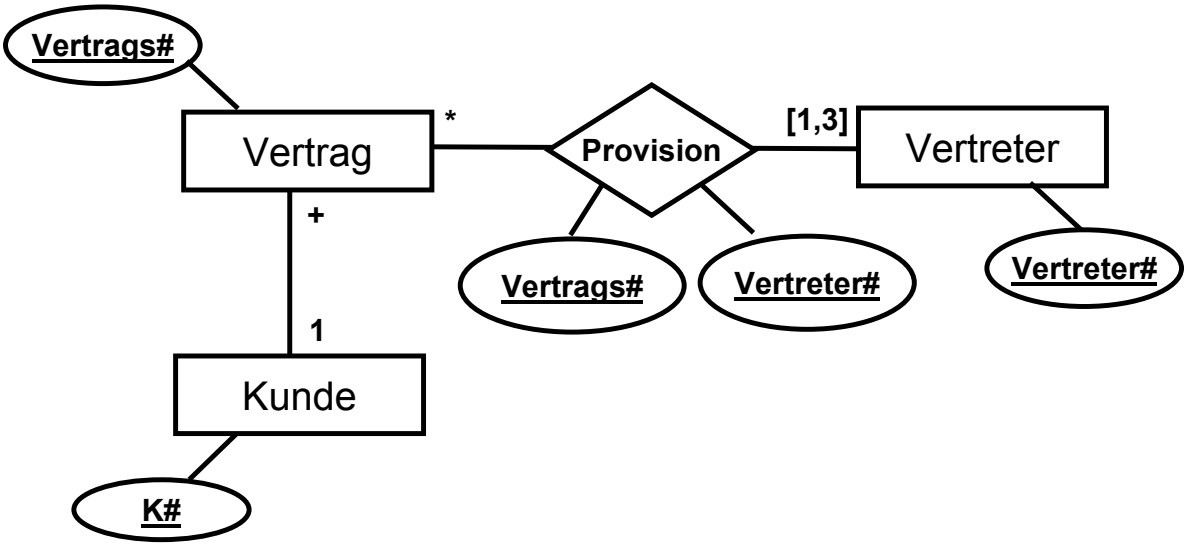
- **k : genau k-mal. Z.B. 3**
- **[n,m] : mindestens n, maximal m. Z.B. [1,5]**
- *** : Null oder mehr**
- **+ : 1 oder mehr**
- **c : Null oder 1**



Aufgabe Lebensversicherungen

- Ein Versicherungsunternehmen beabsichtigt, zur Verwaltung der Lebensversicherungen ein relationales Datenbanksystem einzusetzen. Sie werden mit der Erstellung des Datenmodells beauftragt. Gehen Sie von folgenden (stark vereinfachten) Annahmen aus:
  - Die Datenbank soll Informationen über Kunden, Versicherungs-verträge und Vertreter speichern.
  - Versicherungsverträge weisen unterschiedliche Vertragssummen und Laufzeiten auf und sind entweder vom Typ K (Kapitallebens-versicherung), R (Risikolebensversicherung) oder S (sonstige Lebensversicherung).
  - Am Abschluß eines Versicherungsvertrages können bis zu drei Vertreter beteiligt sein.
  - Jeder Vertreter bekommt eine Provision als Prozentsatz der vereinbarten Vertragssumme, wobei die Provision für jeden beteiligten Vertreter und je Vertrag unterschiedlich sein kann.
- Zeichnen Sie das ERD und bestimmen Sie die Primärschlüssel

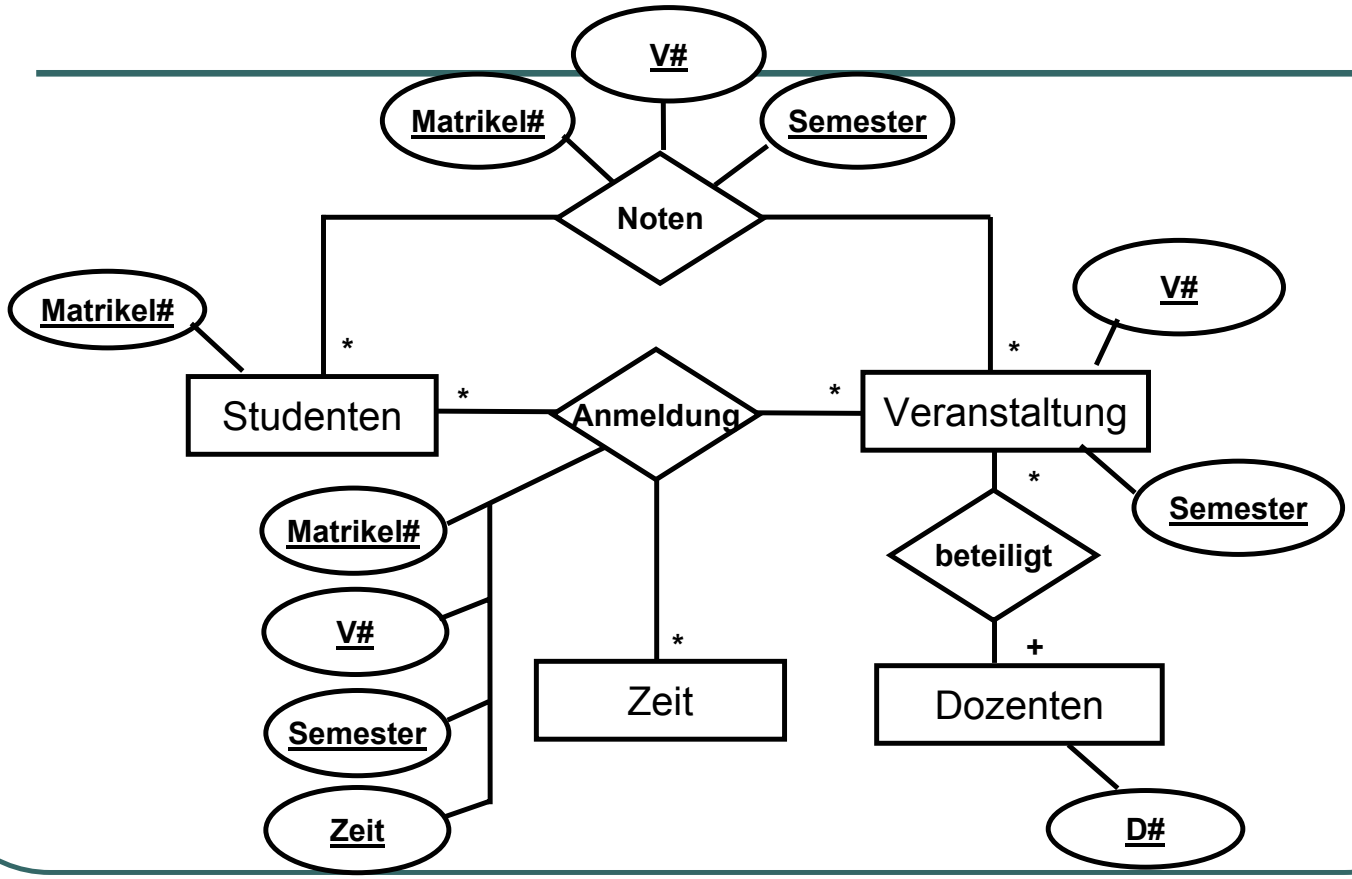
Aufgabe - Lebensversicherungen



Aufgabe 9 - Klausuranmeldung

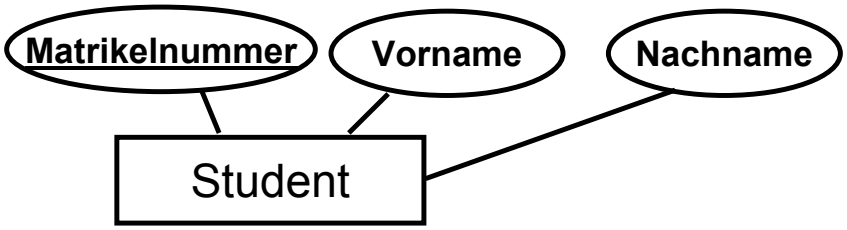
- Ein Fachbereich der Uni Berlin beabsichtigt die Anmeldungen seiner Grundstudiumsklausuren computergestützt durchzuführen. Dazu wird jedem Studienanfänger ein Passwort zugeteilt. Für jeden Studenten werden folgende personenbezogenen Informationen benötigt: Name, Vorname, Matrikelnummer, Geburtsdatum, Adresse, Geschlecht und Telefonnummer. Eine Grundstudiumsveranstaltung wird durch die semesterunabhängige Veranstaltungsnummer, Semesterbezeichnung, die Semesterwochenstundenanzahl, einer Bezeichnung und einer Liste der beteiligten Dozenten beschrieben.
- Bei der Klausuranmeldung gibt ein Student Matrikelnummer und Passwort ein und wählt aus einer Liste die Veranstaltungen, für die er sich anmelden will. Aus rechtlichen Gründen muß der Zeitpunkt (Datum, Uhrzeit) der Anmeldung erfaßt werden. Sehen Sie auch vor, daß das Resultat der Klausur gespeichert werden kann. Berücksichtigen Sie, daß mehrfache An- und Abmeldungen für eine bestimmte Klausur zulässig sein sollen.
- Zeichnen Sie das ERD und bestimmen Sie die Primärschlüssel

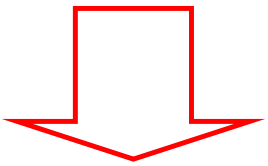
Aufgabe - Klausuranmeldung



Überführung von Objekttypen ins Relationenmodell

- Jeder Objekttyp wird in einen Relationstyp überführt
- Empfehlung: Relationstypen sollten in der Pluralform bezeichnet werden (z.B. Bestellungen, Studenten).

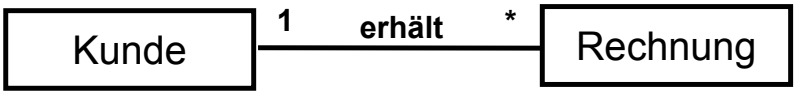


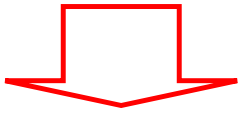


Studenten (Matrikelnummer, Vorname, Nachname)

Überführung von 1:n und c:n – Beziehungstypen

- 1:n - Beziehungen (1:* 1:+ 1:c 1:k 1:[n,m])
  - Verknüpfung der Relationstypen durch Fremdschlüssel im Relationstyp mit der Kardinalität 1
- c:n - Beziehungen (c:* c:+ c:k c:[n,m])
  - Verknüpfung der Relationstypen durch Fremdschlüssel im Relationstyp mit der Kardinalität m
- Beispiel





Fremdschlüssel

Kunden (<u>Kundennummer</u>, Vorname, Nachname, ...)

Rechnungen (<u>Rechnungssnummer</u>, Datum, Kundennummer, ..)

Beispiel: 1:n und c:n – Beziehungstypen

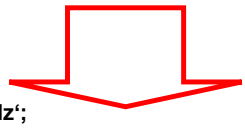
Rechnungen

Rechnungsnummer	Datum	Kundennummer
12454	**1.1.2000**	**3003587**
65432	**13.5.2000**	**3003587**
87342	**24.8.2000**	**3110020**

Kunden

Kundennummer	Vorname	Nachname
3003587	**Christian**	**Schulz**
3072456	**Martin**	**Seger**
3110020	**Julia**	**Maier**

```
SELECT Nachname, Rechnungsnummer, Datum
FROM Kunden, Rechnungen
WHERE Kunden.Kundennummer = Rechnungen.Kundennummer AND Nachname = 'Schulz';
```

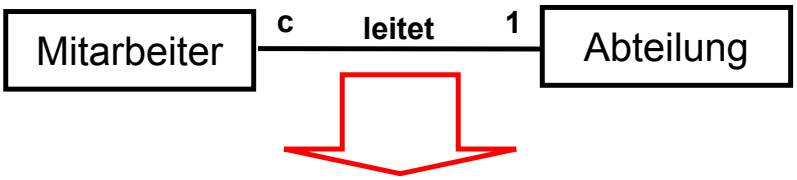


Ergebnis

Nachname	Rechungsnummer	Datum
Schulz	**12454**	**1.1.2000**
Schulz	**65432**	**13.5.2000**

Überführung von 1:1,c:1 und c:c – Beziehungstypen

- 1:1 – Beziehungen
  - Verknüpfung der Relationstypen durch Fremdschlüssel in einem der beiden Relationstypen
  - oder beide Objekttypen werden zu einem Relationstypen zusammengefasst
- c:c - Beziehungen
  - Verknüpfung der Relationstypen durch Fremdschlüssel in einem der beiden Relationstypen. Idealerweise den Fremdschlüssel in den Relationstyp für den weniger Datensätze erwartet werden einfügen.
- Beispiel
  - Annahme: Es gibt auch Abteilungen ohne Leiter



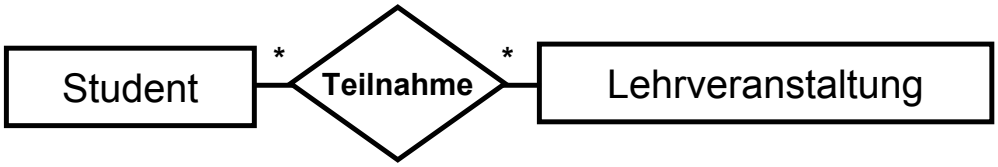
Fremdschlüssel

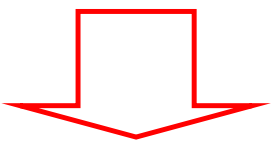
Mitarbeiter(<u>Mitarbeiternummer</u>, Vorname, Nachname, ...)

Abteilungen (<u>Abteilungsnummer</u>, Abteilungsname, Leiter, ..)

Überführung von n:m Beziehungstypen

- n:m - Beziehungen (*:* +:+ *:+ [n,m]:[n,m])
  - Es muss ein eigener Relationstyp für den Beziehungstyp gebildet werden.
  - Der Primärschlüssel dieses Relationstyps wird aus den beiden Primärschlüsseln der anderen Relationstypen zusammengesetzt.





Studenten (<u>Matrikelnummer</u>, Vorname, Nachname, ...)

Lehrveranstaltungen (<u>Veranstaltungsnummer</u>, Termin, Raum, ..)

Teilnahme (<u>Matrikelnummer</u>, <u>Veranstaltungsnummer</u>)

Beispiel: n:m – Beziehungstypen

Lehrveranstaltungen

Veranstaltungsnummer	Termin	Raum
12454	**Di, 10-12**	**102**
65432	**Mi, 14-16**	**OR3**

Studenten

Matrikelnummer	Vorname	Nachname
3003587	**Christian**	**Schulz**
3072456	**Martin**	**Seger**

Teilnahme

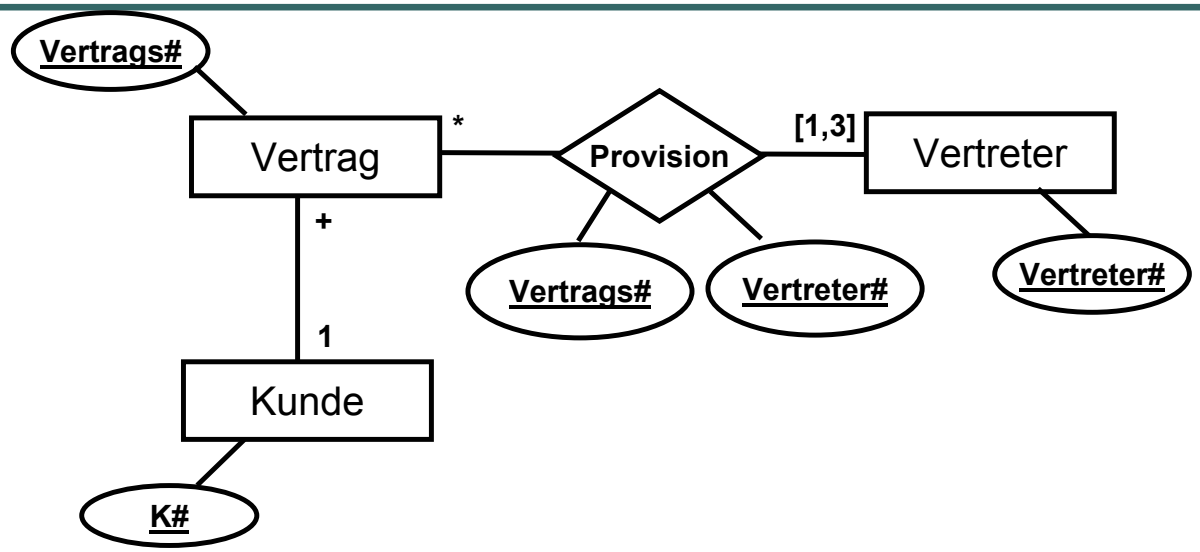
Veranstaltungsnummer	Matrikelnummer
12454	**3072456**
65432	**3072456**

```
SELECT Nachname, Termin, Raum
FROM Studenten, Lehrveranstaltungen, Teilnahme
WHERE Studenten.Matrikelnummer = Teilnahme. Matrikelnummer
      AND Lehrveranstaltungen.Veranstaltungsnummer = Teilnahme.Veranstaltungsnummer
      AND Nachname = 'Seger';
```

Ergebnis

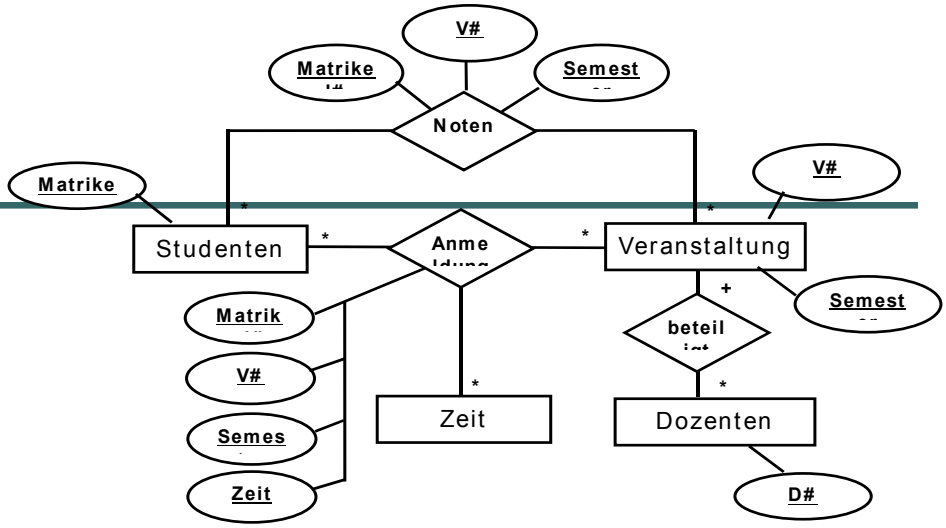
Nachname	Termin	Raum
Seger	**Di, 10-12**	**102**
Seger	**Mi, 14-16**	**OR3**

Aufgabe Lösung



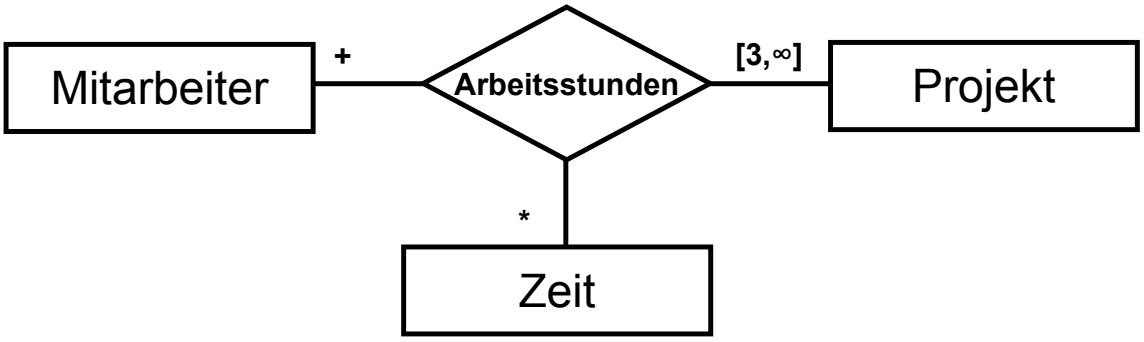
- Kunden (K#, K-Name, K-Straße, ...)
- Vertreter (Vertreter#, V-Name, V-Straße, ...)
- Vertraege (Vertrags#, Typ, Summe, Abschlussdatum, Laufzeit, K#)
- Provisionen (Vertrags#, Vertreter#, Provision)

Lösung



- Studenten (Matrikel#, Name, Vorname, geboren, Adresse, Geschlecht, Telefon, Passwort)
- Veranstaltungen (V#, Semester, Sws, Bezeichnung)
- Dozenten (D#, Name, Vorname, Telefon, Adresse)
- beteiligt (V#, Semester, d#)
- Anmeldungen (Matrikel#, Datum, Uhrzeit, V#, Semester, Abmeldungsdatum)
- Noten (Matrikel#, V#, Semester, Note)

Aufgabe Lösung



- Relationenmodell:
  - Mitarbeiter (M-Nr, Name, Vorname, Adresse, ...)
  - Projekte (P-Nr, P-Name, Beginn, Ende, Budget)
  - Arbeitsstunden (M-Nr, P-Nr, Jahr, Kalenderwoche, Stunden)

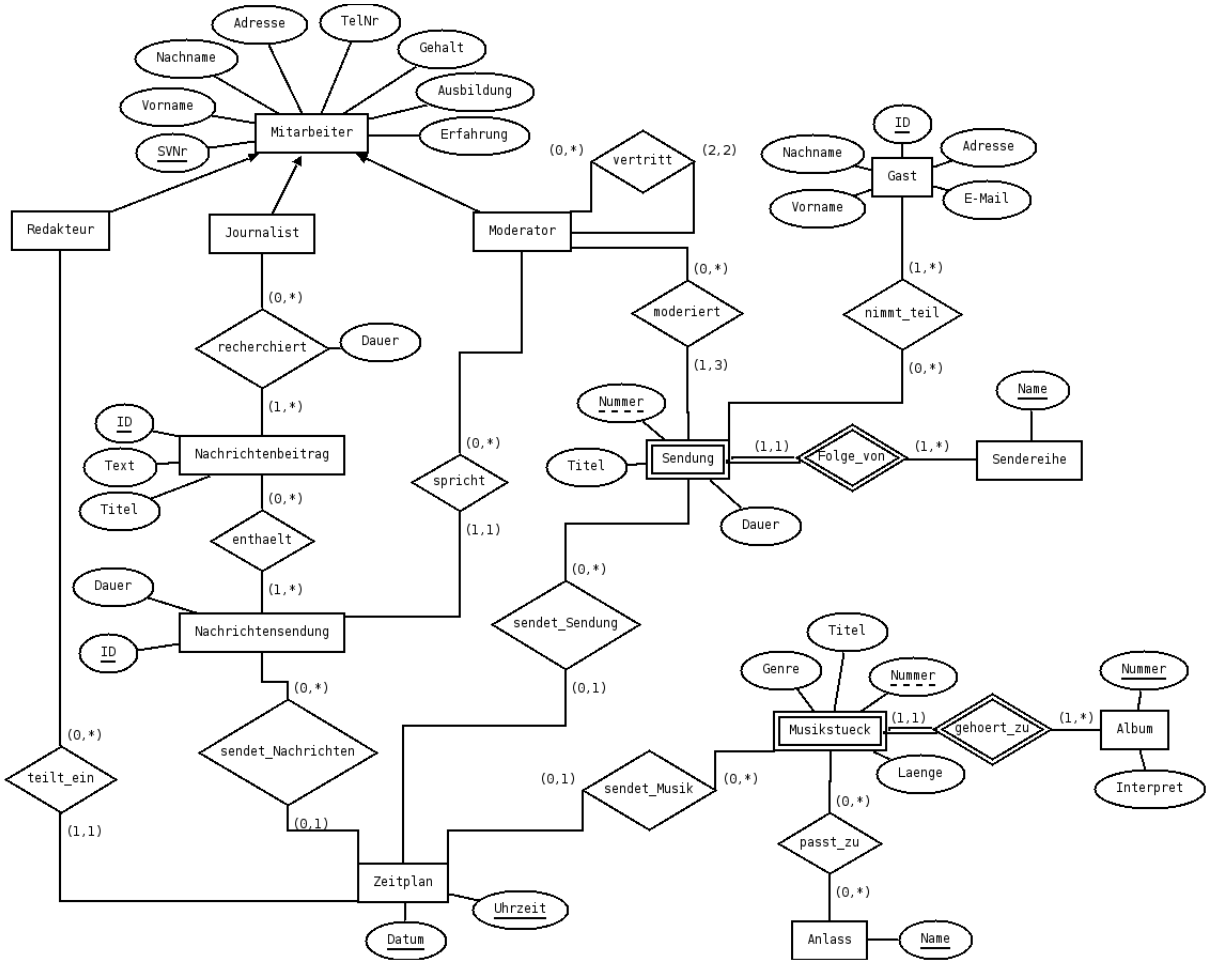


Abbildung 4: Ein nicht so einfaches ER-Modell (Nachrichtensendung), ©TU-Wien

6 Normalisierung

Nach dem mit Hilfe des ER-Modell die zu modellierende Realität in Tabellen (Relations) gebracht wurde, muss nun noch kontrolliert werden ob keine Redundanzen vorhanden sind. Redundanzen in den Daten könnten dazu führen, dass sie eventuell bei Updates nur teilweise und unvollständig geändert werden.
Dies kann dazu führen, dass sie obsolet oder widersprüchlich werden. Man spricht von auftretenden Anomalien. Zudem belegt mehrfache Speicherung derselben Daten unnötig Speicherplatz. Um Redundanz zu verhindern, **normalisiert** man solche Tabellen.

> Unter Normalisierung eines relationalen Datenschemas (Tabellenstruktur) versteht man die Aufteilung von Attributen (Tabellenspalten) in mehrere Relationen (Tabellen) gemäß den Normalisierungsregeln (s. u.), so dass eine Form entsteht, die keine vermeidbaren Redundanzen mehr enthält.

Es gibt eigentlich fünf Normalformen, aber für uns sind die ersten drei ausreichend.

6.1 Erste Normalform (1NF)

> Die **Erste Normalform (1NF)** ist dann gegeben, wenn **alle Informationen** in einer Tabelle **atomar** vorliegen.

Dies bedeutet, dass jede Information innerhalb einer Tabelle eine eigene Tabellenspalte bekommt und zusammenhängende Informationen, wie zum Beispiel die Postleitzahl (PLZ) und der Ort, nicht in einer Tabellenspalte vorliegen dürfen.

Beispiel: Relation Freifächer:

SNr	Name	Addresse	Klasse	FreiFNr	Beschreibung	Zeit [h]
1	Max Moor	Study-Road 1, 9874 Bigtown	5BHEL	1,2	Robotic, Spanish	2,2
2	Steve Platte	Nr.12, 5698 Sleepy	5BHEL	1	Robotic	2
3	Denis Zack	Nr:13 Top:7, 9874 Bigtown	5BHEL	2,3	Spanish, Boxing	2,3
4	Tim Coat	Nr.1, 6666 Thecave	5BHEL	1,3	Robotic, Boxing	2,3

Zur Bildung der ersten Normalform müssen die nicht atomaren Attribute umgewandelt werden. Dies kann durch Einfügen zusätzlicher Zeilen, Spalten oder neuer Relationen erfolgen.
Relation Freifächer in der ersten Normalform:

SNr	Vorname	Nachname	Addresse	PLZ	Ort	Klasse	FreiFNr	Beschreibung	Zeit [h]
1	Max	Moor	Study-Road 1	9874	Bigtown	5BHEL	1	Robotic	2
1	Max	Moor	Study-Road 1	9874	Bigtown	5BHEL	2	Spanish	2
2	Steve	Platte	Nr.12	5698	Sleepy	5BHEL	1	Robotic	2
3	Denis	Zack	Nr:13 Top:7	9874	Bigtown	5BHEL	2	Spanish	2
3	Denis	Zack	Nr:13 Top:7	9874	Bigtown	5BHEL	3	Boxing	3
4	Tim	Coat	Nr.1	6666	Thecave	5BHEL	1	Robotic	2
4	Tim	Coat	Nr.1	6666	Thecave	5BHEL	3	Boxing	3

Die Redundanz nimmt zu. Der bisherige Primärschlüssel verliert seine Eindeutigkeit, es muss ein Neuer gefunden werden.
Festlegung: SNr und FreiFNr sind Schlüsselkandidaten.

6.2 Zweite Normalform (2NF)

Ein Relationstyp (Tabelle) befindet sich genau dann in der **zweiten Normalform (2NF)**, wenn er sich in der ersten Normalform (1NF) befindet und jedes Nichtschlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist.

Die Spalten, die von einem Schlüsselkandidaten nicht vollständig funktional abhängig sind, werden in einer Untertabelle ausgelagert. Der Teil des Schlüsselkandidaten, von dem eine ausgelagerten Spalte funktional abhängig ist, wird Primärschlüssel der neuen Tabelle. In der zweiten Normalform werden auch die ersten Beziehungen in Datenbanken festgelegt.
Es bietet sich an unsere Relation Freifächer in eine Relation Schüler und eine Relation Freifach auzuteilen und die Beziehung der beiden (welcher Schüler besucht welches Freifach für wieviele Stunden) in der Relation Freifachbelegung darzustellen:

Relation Schüler: (Primärschlüssel SNr)

SNr	Vorname	Nachname	Addresse	PLZ	Ort	Klasse
1	Max	Moor	Study-Road 1	9874	Bigtown	5BHEL
2	Steve	Platte	Nr.12	5698	Sleepy	5BHEL
3	Denis	Zack	Nr:13 Top:7	9874	Bigtown	5BHEL
4	Tim	Coat	Nr.1	6666	Thecave	5BHEL

Relation Freifach: (Primärschlüssel FreiFNr)

FreiFNr	Beschreibung
1	Robotic
2	Spanish
3	Boxing

Relation Freifachbelegung:

SNr	FreiFNr	Zeit [h]
1	1	2
1	2	2
2	1	2
3	2	2
3	3	3
4	1	2
4	3	3

6.3 Dritte Normalform (3NF)

Ein Relationstyp befindet sich genau dann in der **dritten Normalform (3NF)**, wenn er sich in der zweiten Normalform (2NF) befindet und kein Nichtschlüsselattribut transitiv von einem Kandidatenschlüssel abhängt.

Die Dritte Normalform ist das Ziel einer erfolgreichen Normalisierung in einem relationalen Datenbankmodell. Sie verhindert einerseits Anomalien und Redundanzen in Datensätzen und andererseits bietet sie genügend Performance für SQL-Abfragen. Die Dritte Normalform ist oft ausreichend, um eine gute Balance aus Redundanz, Performance und Flexibilität für eine Datenbank zu gewährleisten.
In unserem Beispiel ist in der Relation Schüler das Attribut „Ort“ abhängig vom Attribut „PLZ“, und nicht vom Primärschlüssel SNr!
Die transitiv abhängigen Spalten werden in eine weitere Untertabelle ausgelagert, da sie nicht direkt vom Schlüsselkandidaten abhängen, sondern nur indirekt.

Relation Schüler: (Primärschlüssel SNr)

SNr	Vorname	Nachname	Addresse	PLZ	Klasse
1	Max	Moor	Study-Road 1	9874	5BHEL
2	Steve	Platte	Nr.12	5698	5BHEL
3	Denis	Zack	Nr:13 Top:7	9874	5BHEL
4	Tim	Coat	Nr.1	6666	5BHEL

Relation Polstleitzahl: (Primärschlüssel PLZ)

PLZ	Ort
9874	Bigtown
5698	Sleepy
6666	Thecave

Relation Freifach: (Primärschlüssel FreiFNr)

FreiFNr	Beschreibung
1	Robotic
2	Spanish
3	Boxing

Relation Freifachbelegung:

SNr	FreiFNr	Zeit [h]
1	1	2
1	2	2
2	1	2
3	2	2
3	3	3
4	1	2
4	3	3

7 SQL-Commandline-Tutorial

Voraussetzung ist eine funktionierende lokale MySQL-Server oder MariaDB-Server Installation mit einem PW-gesicherten Root-User.

7.1 Anmeldung

Um eine Verbindung mit dem Server herzustellen, muss beim Aufruf von mysql normalerweise einen MySQL-Benutzernamen und in aller Regel auch ein Passwort angeben werden. Wird der Server auf einem anderen System als demjenigen ausgeführt, an dem man angmeldent ist, dann muss auch einen Hostnamen angeben werden.

```
shell> mysql -h host-ip -u userName -p
Enter password: ***********

oder

shell> mysql -u userName -p
Enter password: ***********
```

Wenn man erfolgreich eingelogt ist bekommt je nach Version folgenden prompt:

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 31
Server version: 10.1.34-MariaDB-0ubuntu0.18.04.1 Ubuntu 18.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Um sich wieder auszuloggen gibt man einfach **quit** ein.

7.2 Anfragen eingeben

Dieser Abschnitt erläutert die Grundprinzipien der Befehlseingabe mithilfe verschiedener Abfragen.
Zunächst ein einfacher Befehl, der den Server bittet, seine Versionsnummer und das aktuelle Datum anzugeben:

```
MariaDB [(none)]> SELECT VERSION(), CURRENT_DATE;
+---------------------------------+--------------+
| VERSION()                       | CURRENT_DATE |
+---------------------------------+--------------+
| 10.1.34-MariaDB-0ubuntu0.18.04.1 | 2019-01-05   |
+---------------------------------+--------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```

Diese Abfrage veranschaulicht mehrere Aspekte von mysql:

- Ein Befehl besteht normalerweise aus einer SQL-Anweisung gefolgt von einem Semikolon. (Es gibt eine Reihe von Ausnahmen, bei denen das Semikolon weggelassen werden kann. Ein Beispiel ist das bereits weiter oben erwähnte QUIT; weitere werden folgen.)
- Wenn Sie einen Befehl absetzen, sendet mysql (oder natürlich MariaDB, aber von nun an wird im Skript immer mysql verwendet!) ihn zur Ausführung an den Server und zeigt die Ergebnisse an. Darauf folgt wieder eine neue Eingabeaufforderung `MariaDB [(none)]>`, mit der angezeigt wird, dass nun ein neuer Befehl eingegeben werden kann.

- mysql zeigt die Abfrageausgabe in Tabellenform (d. h. als Zeilen und Spalten) an. Die erste Zeile enthält die Spaltenüberschriften. Alle nachfolgenden Zeilen sind Abfrageergebnisse. Normalerweise sind Spaltenüberschriften die Namen der Spalten, die aus den Datenbanktabellen abgerufen werden. Wenn Sie den Wert eines Ausdrucks statt einer Tabellenspalte (wie im obigen Beispiel) abrufen, beschriftet mysql die Spalte mit dem Ausdruck selbst.
- mysql zeigt an, wie viele Datensätze (Zeilen) zurückgegeben wurden und wie lange die Ausführung der Abfrage dauerte; hierdurch können Sie grob auf die Serverleistung schließen. Die Werte sind allerdings nicht sehr genau, denn sie geben nur eine normale Zeit statt der Prozessor- oder Systemzeit an, die zudem durch Faktoren wie der Serverauslastung und der Netzwerklatenz beeinflusst wird. (Aus Gründen der Übersichtlichkeit haben wir die Zeile „rows in set“ in einigen der in diesem Kapitel aufgeführten Beispiele weggelassen.)

Schlüsselwörter können in beliebiger Groß-/Kleinschreibung angegeben werden. Die folgenden Abfragen sind gleichwertig:

```
MariaDB [(none)]> SELECT VERSION(), CURRENT_DATE;
MariaDB [(none)]> select version(), current_date;
MariaDB [(none)]> SeLeCt vErSIoN(), CURRent_dATe;
```

Aus Gründen der Lesebarkeit und der Einheitlichkeit werden Schlüsselwörter in diesem Text immer in Grossbuchstaben geschrieben!

Man kann auch mehrere Abfragen in eine Zeile schreiben. Dazu wird einfache jede Abfrage mit einem Semikolon abgeschlossen z.B.:

```
MariaDB [(none)]> SELECT VERSION(); SELECT NOW();
+---------------------------------+
| VERSION()                       |
+---------------------------------+
| 10.1.34-MariaDB-0ubuntu0.18.04.1 |
+---------------------------------+
1 row in set (0.00 sec)

+---------------------+
| NOW()               |
+---------------------+
| 2019-01-05 20:25:38 |
+---------------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```

Ein Befehl muss nicht vollständig innerhalb einer einzelnen Zeile angegeben werden; insofern stellen auch längere Befehle über mehrere Zeilen kein Problem dar. mysql ermittelt das Ende der Anweisung anhand des schließenden Semikolons und nicht auf der Basis der Eingabezeile. (Anders gesagt, akzeptiert mysql frei formatierte Eingaben: Alle Eingabezeilen werden gesammelt, die Ausführung erfolgt aber erst, nachdem das Semikolon erkannt wurde.)
Hier ist eine einfache mehrzeilige Anweisung:

```
MariaDB [(none)]> SELECT
    -> USER()
    -> ,
    -> CURRENT_DATE;
+----------------+--------------+
| USER()         | CURRENT_DATE |
+----------------+--------------+
| root@localhost | 2019-01-05   |
+----------------+--------------+
1 row in set (0.00 sec)

MariaDB [(none)]>
```

Achtung: Die Eingabeaufforderung stellt von `MariaDB [(none)]>` auf `->` um, nachdem die erste Zeile einer mehrzeiligen Abfrage eingegeben wurde. Auf diese Weise zeigt mysql an, dass noch keine vollständige Anweisung erkannt wurde und weitere Eingaben erwartet werden.

Wenn einen Befehl, der gerade eingeben wurde, doch nicht ausgeführt werden soll, kann man ihn durch Eingabe von `\c` abbrechen:

```
MariaDB [(none)]> SELECT
    -> USER()
    -> \c
MariaDB [(none)]>
```

Die Eingabeaufforderungen `'>` und `">` erscheinen bei der Erfassung von Strings (MySQL erwartet also die Vervollständigung eines Strings). In MySQL können Sie Strings entweder in `'` oder `"` setzen (z.B. `'hello'` oder `"goodbye"`). mysql erlaubt die Eingabe von Strings, die sich über mehrere Zeilen erstrecken. Achtung: Wenn dies unabsichtlich passiert, z.B. wenn einfach ein Anführungszeichen vergessen wurde, kehrt mysql erst wieder zu Befehlseingabe zurück, wenn das fehlende Anführungszeichen gegeben wurde - dies kann etwas verwirrend sein, da selbst `\c` und verb—quit— nicht funktionieren solange das fehlende Anführungszeichen nicht eingegeben wird! Ein Beispiel:

```
MariaDB [(none)]> SELECT * FROM my_table WHERE name = 'Markus AND age < 50;
    '> \c
    '> quit
    '> '
    -> \c
MariaDB [(none)]>
```

Die Eingabeaufforderung `` `> `` ähnelt `'>` und `">`, gibt aber an, dass Sie mit einem Backtick einen Bezeichner begonnen, aber noch nicht beendet haben.

7.3 Eine Datenbank erzeugen und einem User zuweisen

Verwenden Sie die SHOW-Anweisung, um zu ermitteln, welche Datenbanken derzeit auf dem Server vorhanden sind:

```
MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
3 rows in set (0.00 sec)
```

Die Liste der Datenbanken sieht auf Ihrem Computer wahrscheinlich etwas anders aus, aber die Datenbanken mysql ist höchstwahrscheinlich vorhanden, da sie die Benutzerberechtigungen beschreibt.

7.3.1 Erzeugen und zuweisen

Als root user kann man nun eine neue Datenbank (nennen wir sie menagerie - weil sie alle Haustiere enthalten soll) erzeugen
(mittels `CREATE DATABASE menagerie;` und auch gleich einem User zuweisen (wenn der User neu angelegt wird auch gleich mit einem Passwort ausstatten):

```
MariaDB [(none)]> CREATE DATABASE menagerie;
Query OK, 1 row affected (0.00 sec)

MariaDB [(none)]> GRANT ALL PRIVILEGES ON menagerie.* TO 'markus'@'localhost'
    -> IDENTIFIED BY 'markus123' WITH GRANT OPTION;
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| performance_schema |
+--------------------+
4 rows in set (0.00 sec)
```

Wenn man sich nun als der neue User *markus* anmeldet und ein SHOW DATABASES eingibt sieht man nun die menagerie Datenbank (nicht aber mysql und performance_schema, die sind dem root user vorbehalten):

```
shell> mysql -u markus -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 33
Server version: 10.1.34-MariaDB-0ubuntu0.18.04.1 Ubuntu 18.04

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| menagerie          |
+--------------------+
2 rows in set (0.00 sec)
```

Achtung: Unter Unix sind die Database-Namen case-sensitive!!!

7.4 Mit der Datanbank arbeiten

Wenn Sie eine Datenbank erstellen, wird diese nicht automatisch für die Verwendung ausgewählt; Sie müssen dies ausdrücklich tun. Um menagerie also zur aktuellen Datenbank zu machen, verwenden Sie folgenden Befehl:

```
MariaDB [(none)]> USE menagerie
Database changed
MariaDB [menagerie]>
```

7.4.1 Eine Tabelle erzeugen

Das Erstellen einer Datenbank ist ganz einfach. Noch aber ist die Datenbank leer, wie Sie mit SHOW TABLES nachprüfen können:

```
MariaDB [menagerie]> SHOW TABLES;
Empty set (0.00 sec)

MariaDB [menagerie]>
```

Das Layout Ihrer Tabelle legen Sie mit einer CREATE TABLE-Anweisung fest:

```
MariaDB [menagerie]> CREATE TABLE haustier (name VARCHAR(20), besitzer VARCHAR(20),
    -> gattung VARCHAR(20), geschlecht CHAR(1), geburtstag DATE, todestag DATE);
Query OK, 0 rows affected (0.05 sec)

MariaDB [menagerie]>
```

VARCHAR ist eine geeignete Wahl für die Spalten name, besitzer und gattung, denn die Werte dieser Spalten können in der Länge variieren. Die Längen in diesen Spaltendefinitionen müssen weder identisch sein noch alle den Wert 20 haben. Sie können eine beliebige Länge zwischen 1 und 65535 eingeben – je nachdem, was Ihnen am sinnvollsten erscheint. Haben Sie jedoch eine falsche Entscheidung getroffen und es stellt sich später heraus, dass Sie ein längeres Feld benötigen, dann bietet MySQL hierfür die ALTER TABLE-Anweisung an.
Zur Auswahl des Geschlechts Ihrer Tiere bieten sich mehrere Wertetypen an, z. B. 'm' und 'w' oder auch 'männlich' und 'weiblich'. Am einfachsten ist die Verwendung der Einzelzeichen 'm' und 'w'.
Der Datentyp DATE ist offensichtlich am geeignetsten für die Spalten geburtstag und todestag.
Wenn Sie eine Tabelle erstellt haben, sollte die Anweisung SHOW TABLES eine entsprechende Ausgabe erzeugen und um zu überprüfen, dass Ihre Tabelle wie gewünscht erstellt worden ist, verwenden Sie eine DESCRIBE-Anweisung::

```
MariaDB [menagerie]> SHOW TABLES; DESCRIBE haustier;
+---------------------+
| Tables_in_menagerie |
+---------------------+
| haustier            |
+---------------------+
1 row in set (0.00 sec)

+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| name       | varchar(20) | YES  |     | NULL    |       |
| besitzer   | varchar(20) | YES  |     | NULL    |       |
| gattung    | varchar(20) | YES  |     | NULL    |       |
| geschlecht | char(1)     | YES  |     | NULL    |       |
| geburtstag | date        | YES  |     | NULL    |       |
| todestag   | date        | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

Weitere Informationen zu MySQL-Datentypen finden Sie online auf:
SQL Datentypen

7.4.2 Die Tabellen mit Daten füllen

Nachdem Sie Ihre Tabelle erstellt haben, müssen Sie sie mit Daten füllen. Zu diesem Zweck sind die Anweisungen LOAD DATA und INSERT vorhanden. Beachten Sie dabei, dass MySQL Datumsangaben im Format 'YYYY-MM-DD' erwartet!

LOAD DATA erlaubt es Einträge aus einer Datei zu laden:
Erstellen Sie also eine Textdatei namens pet.txt mit einem Datensatz pro Zeile, bei dem die einzelnen Werte durch Komma voneinander getrennt sind, also eigentlich eine CSV Datei - wobei das Trennzeichen zwischen den Feldern spezifiert werden kann; die Werte müssen dabei in der Reihenfolge eingegeben werden, in der Sie in der CREATE TABLE-Anweisung aufgeführt wurden. Bei fehlenden Werten (z. B. wenn das Geschlecht nicht bekannt oder das Ableben des Tieres noch nicht erfolgt ist) können Sie NULL-Werte eingeben. Diese geben Sie als `\N` (Backslash, großes N) in Ihre Textdatei ein.
Die Datei könnte zum Beispiel so aussehen:

```
shell> cat pet.txt
```

```
Fluffy,Sepp,Katze,w,2001-02-03,\N
Wuffy,Franz,Hund,m,2002-04-05,2010-10-10
Beiser,Franz,Hund,m,2008-09-12,\N
Schnurrly,Otto,Katze,w,2009-12-12,\N
Zwitscherly,Otto,Vogel,m,2009-03-12,2012-09-09
```

Um die Textdatei pet.txt in die Tabelle haustier zu laden, genügt die folgende Anweisung - Achtung auf die Spezifizierung des Trennzeichens mittels `FIELDS TERMINATED BY ','`

```
MariaDB [menagerie]> LOAD DATA LOCAL INFILE '/path_to_the_File/pet.txt' INTO TABLE haustier
    -> FIELDS TERMINATED BY ',';
Query OK, 5 rows affected (0.00 sec)
Records: 5  Deleted: 0  Skipped: 0  Warnings: 0

MariaDB [menagerie]> select * from haustier;
+-------------+----------+---------+------------+------------+------------+
| name        | besitzer | gattung | geschlecht | geburtstag | todestag   |
+-------------+----------+---------+------------+------------+------------+
| Fluffy      | Sepp     | Katze   | w          | 2001-02-03 | NULL       |
| Wuffy       | Franz    | Hund    | m          | 2002-04-05 | 2010-10-10 |
| Beiser      | Franz    | Hund    | m          | 2008-09-12 | NULL       |
| Schnurrly   | Otto     | Katze   | w          | 2009-12-12 | NULL       |
| Zwitscherly | Otto     | Vogel   | m          | 2009-03-12 | 2012-09-09 |
+-------------+----------+---------+------------+------------+------------+
5 rows in set (0.00 sec)
```

INSERT verwendet man um der Tabelle einzelne Einträge hinzuzufügen. In ihrer einfachsten Form geben Sie Werte für jede Spalte in der Reihenfolge an, in der die Spalten in der CREATE TABLE-Anweisung aufgeführt waren. Nehmen wir etwa an, Luise bekommt einen neuen Hamster namens „Fluffkugel“. Nun könnten Sie den erforderlichen neuen Datensatz wie folgt mithilfe einer INSERT-Anweisung angeben:

```
MariaDB [menagerie]> INSERT INTO haustier
    -> VALUES ('Fluffkugel','Luise','Hamster','w','2010-04-30',NULL);
Query OK, 1 row affected (0.00 sec)

MariaDB [menagerie]> select * from haustier;
+-------------+----------+---------+------------+------------+------------+
| name        | besitzer | gattung | geschlecht | geburtstag | todestag   |
+-------------+----------+---------+------------+------------+------------+
| Fluffy      | Sepp     | Katze   | w          | 2001-02-03 | NULL       |
| Wuffy       | Franz    | Hund    | m          | 2002-04-05 | 2010-10-10 |
| Beiser      | Franz    | Hund    | m          | 2008-09-12 | NULL       |
| Schnurrly   | Otto     | Katze   | w          | 2009-12-12 | NULL       |
| Zwitscherly | Otto     | Vogel   | m          | 2009-03-12 | 2012-09-09 |
| Fluffkugel  | Luise    | Hamster | w          | 2010-04-30 | NULL       |
+-------------+----------+---------+------------+------------+------------+
6 rows in set (0.00 sec)
```

String- und Datenwerte werden hier in Anführungszeichen gesetzt. Mit der INSERT-Anweisung können Sie NULL auch direkt eingeben, um einen fehlenden Wert darzustellen.

Dies waren nur zwei kurze Beispiele. Für mehr Information zu loading und inserting data bitte unter https://mariadb.com/kb/en/library/inserting-loading-data/ nachsehen!

7.4.3 Informationen aus einer Tabelle abfragen

Mit der SELECT-Anweisung können Sie Daten aus einer Tabelle abrufen. Die allgemeine Form dieser Anweisung sieht wie folgt aus:

```
SELECT what_to_select
FROM which_table
WHERE conditions_to_satisfy;
```

Wobei wir `SELECT * FROM haustier` oben schon verwendet haben um den ganzen Tabelleninhalt von haustier anzuzeigen, um zu kontrollieren ob die LOAD DATA und INSERT Befehle funktioniert haben.

WHERE wird verwendet um nun eine bestimmte Bedingung zu definieren. Zum Beispiel wollen wir alle Katzen der Tabelle anzeigen:

```
MariaDB [menagerie]> SELECT * FROM haustier WHERE gattung = 'Katze';
+-----------+----------+---------+------------+------------+----------+
| name      | besitzer | gattung | geschlecht | geburtstag | todestag |
+-----------+----------+---------+------------+------------+----------+
| Fluffy    | Sepp     | Katze   | w          | 2001-02-03 | NULL     |
| Schnurrly | Otto     | Katze   | w          | 2009-12-12 | NULL     |
+-----------+----------+---------+------------+------------+----------+
2 rows in set (0.00 sec)
```

Diese Bedingungen können auch durch **logische Operatoren** (AND und OR) verknüpft werden, bzw. müssen sie auch nicht immer exakt (=) sein, sondern es können auch ($<, >, <=, >=, !=$) verwendete werden. Ein etwas komplizierteres Beispiel:

```
MariaDB [menagerie]> SELECT * FROM haustier WHERE ((gattung = 'Katze' OR gattung = 'vogel')
    -> AND geschlecht = 'm') OR (gattung = 'Hund' AND todestag != 'NULL');
+-------------+----------+---------+------------+------------+------------+
| name        | besitzer | gattung | geschlecht | geburtstag | todestag   |
+-------------+----------+---------+------------+------------+------------+
| Wuffy       | Franz    | Hund    | m          | 2002-04-05 | 2010-10-10 |
| Zwitscherly | Otto     | Vogel   | m          | 2009-03-12 | 2012-09-09 |
+-------------+----------+---------+------------+------------+------------+
2 rows in set, 1 warning (0.00 sec)
```

Zwischeninfo: Warum sind bei manchen Statements warnings und wie kann man sie anzeigen?
Sofort nach der query kann mit SHOW WARNINGS; die Information zum Warning angezeigt werden. Alternativ kann man die interaktive mysql session mit der Option –show-warnings starten.

Im konkreten Beispiel ist die Warnung:

```
MariaDB [menagerie]> SHOW WARNINGS;
+---------+------+-----------------------------------+
| Level   | Code | Message                           |
+---------+------+-----------------------------------+
| Warning | 1292 | Incorrect datetime value: 'NULL' |
+---------+------+-----------------------------------+
1 row in set (0.00 sec)
```

Das Problem ist `todestag != 'NULL'` weil der != Operator nur auf Werte angewendet werden kann, NULL aber eigenlich kein Wert ist! Die bessere Syntax ist daher: `todestag IS NOT NULL`

```
MariaDB [menagerie]> SELECT * FROM haustier WHERE ((gattung = 'Katze' OR gattung = 'vogel')
    -> AND geschlecht = 'm') OR (gattung = 'Hund' AND todestag IS NOT NULL);
+-------------+----------+---------+------------+------------+------------+
| name        | besitzer | gattung | geschlecht | geburtstag | todestag   |
+-------------+----------+---------+------------+------------+------------+
| Wuffy       | Franz    | Hund    | m          | 2002-04-05 | 2010-10-10 |
| Zwitscherly | Otto     | Vogel   | m          | 2009-03-12 | 2012-09-09 |
+-------------+----------+---------+------------+------------+------------+
2 rows in set (0.00 sec)
```

Gleiches Ergebniss aber ohen warnings!

Wenn nur bestimmte Spalten interessant sind:

```
MariaDB [menagerie]> SELECT name, geburtstag FROM haustier;
+-------------+------------+
| name        | geburtstag |
+-------------+------------+
| Fluffy      | 2001-02-03 |
| Wuffy       | 2002-04-05 |
| Beiser      | 2008-09-12 |
| Schnurrly   | 2009-12-12 |
| Zwitscherly | 2009-03-12 |
| Fluffkugel  | 2010-04-30 |
+-------------+------------+
6 rows in set (0.00 sec)
```

Um herauszufinden welche Tierarten vertreten sind, und um mehrfaches Vorkommen zu ignorieren kann das Schlüsselwort DISTINCT verwendet werden:

```
SELECT gattung FROM haustier;
+---------+
| gattung |
+---------+
| Katze   |
| Hund    |
| Hund    |
| Katze   |
| Vogel   |
| Hamster |
+---------+
6 rows in set (0.00 sec)
```

```
SELECT DISTINCT gattung FROM haustier;
+---------+
| gattung |
+---------+
| Katze   |
| Hund    |
| Vogel   |
| Hamster |
+---------+
4 rows in set (0.00 sec)
```

Abfrageergebnisse **sortieren** kann man mit dem ORDER BY Schlüsselwort:

```
MariaDB [menagerie]> SELECT name, geburtstag FROM haustier ORDER BY geburtstag;
+-------------+------------+
| name        | geburtstag |
+-------------+------------+
| Fluffy      | 2001-02-03 |
| Wuffy       | 2002-04-05 |
| Beiser      | 2008-09-12 |
| Zwitscherly | 2009-03-12 |
| Schnurrly   | 2009-12-12 |
| Fluffkugel  | 2010-04-30 |
+-------------+------------+
6 rows in set (0.00 sec)
```

Die Standardreihenfolge bei der Sortierung ist aufsteigend, d. h., die kleinsten Werte werden zuerst aufgeführt. Um in umgekehrter (absteigender) Reihenfolge zu sortieren, fügen Sie das Schlüsselwort DESC zum Namen der Spalte zu, nach der die Sortierung erfolgt.
Sie können auch nach mehreren Spalten und diese jeweils mit eigener Reihenfolge sortieren. Um beispielsweise nach der Tierart in aufsteigender, dann dem Geburtsdatum innerhalb der Tierart in absteigender Reihenfolge (d. h. das jüngste Tier zuerst nennend) zu sortieren, verwenden Sie die folgende Abfrage:

```
MariaDB [menagerie]> SELECT name, gattung, geburtstag FROM haustier
    -> ORDER BY gattung, geburtstag DESC;
+-------------+---------+------------+
| name        | gattung | geburtstag |
+-------------+---------+------------+
| Fluffkugel  | Hamster | 2010-04-30 |
| Beiser      | Hund    | 2008-09-12 |
| Wuffy       | Hund    | 2002-04-05 |
| Schnurrly   | Katze   | 2009-12-12 |
| Fluffy      | Katze   | 2001-02-03 |
| Zwitscherly | Vogel   | 2009-03-12 |
+-------------+---------+------------+
6 rows in set (0.00 sec)
```

7.4.4 Änderungen an Datensätzen durchführen

Wenn z.B. bei der Eingabe ein Fehler passiert ist, kann man einzelne Datensätze mittels der UPDATE Anweisung ändern:

```
MariaDB [menagerie]> UPDATE haustier SET geburtstag = '1492-10-21' WHERE name = 'Wuffy';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

7.4.5 Datumsberechnungen

MySQL bietet eine Anzahl von Funktionen, mit denen Sie datumsbezogene Berechnungen durchführen können, um etwa Altersangaben zu ermitteln oder Teile aus Datumsangaben zu extrahieren. Um zu bestimmen, wie alt Ihre Haustiere jeweils sind, berechnen Sie die Differenz im Jahresbestandteil des aktuellen und des Geburtsdatums und ziehen den Wert 1 ab, sofern das aktuelle Datum im Kalenderjahr vor dem Geburtsdatum liegt. Die folgende Abfrage zeigt für jedes Haustier das Geburtsdatum, das aktuelle Datum und das Alter in Jahren an, wobei wir nach dem Alter sortieren:

```
MariaDB [menagerie]> SELECT name, geburtstag, CURDATE(),
    -> (YEAR(CURDATE())-YEAR(geburtstag))
    -> - (RIGHT(CURDATE(),5)<RIGHT(geburtstag,5))
    -> AS tieralter
    -> FROM haustier ORDER BY tieralter;
+-------------+------------+------------+-----------+
| name        | geburtstag | CURDATE()  | tieralter |
+-------------+------------+------------+-----------+
| Fluffkugel  | 2010-04-30 | 2019-01-06 |         8 |
| Schnurrly   | 2009-12-12 | 2019-01-06 |         9 |
| Zwitscherly | 2009-03-12 | 2019-01-06 |         9 |
| Beiser      | 2008-09-12 | 2019-01-06 |        10 |
| Fluffy      | 2001-02-03 | 2019-01-06 |        17 |
| Wuffy       | 1492-10-21 | 2019-01-06 |       526 |
+-------------+------------+------------+-----------+
6 rows in set (0.00 sec)
```

Hierbei extrahiert YEAR() die Jahreszahl aus dem Datum, während mit RIGHT() die fünf Zeichen ganz rechts im Datum (MM-DD, also der Monat und der Tag) ermittelt werden. Der Teil des Ausdrucks, der die Werte MM-DD vergleicht, ist entweder 1 oder 0. Hiermit wird von der Jahresdifferenz ggf. ein Jahr abgezogen, sofern CURDATE() (das aktuelle Datum) im Kalenderjahr vor geburtstag liegt. Der gesamte Ausdruck wirkt ein wenig unübersichtlich, weswegen ein Alias (tieralter) verwendet wird, um die Beschriftung der Ausgabespalte sinnvoller zu gestalten.

Was aber, wenn Sie nun wissen wollen, welche Tiere im nächsten Monat Geburtstag haben? Für diese Art der Berechnung sind Jahr und Tag irrelevant: Sie müssen lediglich den Monat aus der Spalte geburtstag

extrahieren. MySQL bietet mehrere Funktionen zur Extraktion von Datumsbestandteilen, z. B. YEAR(), MONTH() und DAYOFMONTH(). MONTH() ist für unsere Belange die passende Funktion. DATE_ADD() erlaubt Ihnen das Addieren eines bestimmten Zeitintervalls für ein gegebenes Datum. Wenn Sie einen Monat zum aktuellen Wert von CURDATE() hinzuaddieren wollen, extrahieren Sie den Monatsbestandteil mit MONTH(). Das Ergebnis ist der Monat, in dem Sie nach Geburtstagen suchen müssen:

```
SELECT name, geburtstag FROM haustier
-> WHERE MONTH(geburtstag) = MONTH(DATE_ADD(CURDATE(),INTERVAL 1 MONTH));
+--------+------------+
| name   | geburtstag |
+--------+------------+
| Fluffy | 2001-02-03 |
+--------+------------+
1 row in set (0.00 sec)
```

7.4.6 Zeilen zählen

Datenbanken werden häufig zur Beantwortung der Frage „Wie häufig taucht ein bestimmter Datentyp in einer Tabelle auf?“ verwendet, oder „Wie viele Datensätze sind in der Tabelle haustier vorhanden?“.
COUNT(*) zählt die Anzahl der Datensätze, d. h., die Abfrage zur Zählung Ihrer Tiere sieht wie folgt aus:

```
SELECT COUNT(*) FROM haustier;
+----------+
| COUNT(*) |
+----------+
|        6 |
+----------+
1 row in set (0.00 sec)
```

Oder mann kann ermitteln, wie viele Tiere jeder Besitzer hat:

```
SELECT besitzer, COUNT(*) FROM haustier GROUP BY besitzer;
+----------+----------+
| besitzer | COUNT(*) |
+----------+----------+
| Franz    |        2 |
| Luise    |        1 |
| Otto     |        2 |
| Sepp     |        1 |
+----------+----------+
4 rows in set (0.00 sec)
```

Beachten Sie, dass GROUP BY zur Gruppierung aller Datensätze für jeden Besitzer besitzer verwendet wird.
COUNT() und GROUP BY sind praktisch, wenn Sie Ihre Daten auf verschiedene Arten charakterisieren wollen. Die folgenden Beispiele zeigen verschiedene Wege, statistische Daten zu den Tieren zu erheben.
Anzahl der Exemplare je Tierart:

```
SELECT gattung, COUNT(*) FROM haustier GROUP BY gattung;
+---------+----------+
| gattung | COUNT(*) |
+---------+----------+
| Hamster |        1 |
| Hund    |        2 |
| Katze   |        2 |
| Vogel   |        1 |
+---------+----------+
4 rows in set (0.00 sec)
```

Anzahl der Tiere je Kombination von Tierart und Geschlecht:

```
SELECT gattung, geschlecht, COUNT(*) FROM haustier GROUP BY gattung, geschlecht;
+---------+------------+----------+
| gattung | geschlecht | COUNT(*) |
+---------+------------+----------+
| Hamster | w          |        1 |
| Hund    | m          |        2 |
| Katze   | w          |        2 |
| Vogel   | m          |        1 |
+---------+------------+----------+
4 rows in set (0.00 sec)
```

Wenn Sie COUNT() verwenden, müssen Sie nicht die gesamte Tabelle abrufen. Die obige Abfrage beispielsweise sieht wie folgt aus, wenn Sie nur für Hunde und Katzen durchgeführt wird:

```
SELECT gattung, geschlecht, COUNT(*) FROM haustier
    -> WHERE gattung = 'Hund' OR gattung = 'Katze'
    -> GROUP BY gattung, geschlecht;
+---------+------------+----------+
| gattung | geschlecht | COUNT(*) |
+---------+------------+----------+
| Hund    | m          |        2 |
| Katze   | w          |        2 |
+---------+------------+----------+
2 rows in set (0.00 sec)
```

7.4.7 Joins

Als SQL-JOIN (auf Deutsch: Verbund) bezeichnet man eine Operation in relationalen Datenbanken, die Abfragen über mehrere Datenbanktabellen ermöglicht. JOINs führen Daten zusammen, die in unterschiedlichen Tabellen gespeichert sind, und geben diese in gefilterter Form in einer Ergebnistabelle aus.
Es folgt ein Beispiel von Wikipedia, das mit folgenden zwei Tabellen (Abteilung und Angestellter) arbeitet:

```
+-------------+---------------+
| AbteilungID | AbteilungName |
+-------------+---------------+
|          31 | Verkauf       |
|          33 | Technik       |
|          34 | Büro          |
|          35 | Marketing     |
+-------------+---------------+
```

```
+-----------+-------------+
| Nachname  | AbteilungID |
+-----------+-------------+
| Müller    |          31 |
| Schmidt   |          33 |
| Schneider |          33 |
| Fischer   |          34 |
| Weber     |          34 |
| Meyer     |        NULL |
+-----------+-------------+
```

7.4.7.1 Cross Join - Ein CROSS JOIN gibt das kartesische (kreuz) Produkt der Tabellenzeilen eines Joins wieder. Mit anderen Worten: Ein neuer Datensatz entsteht durch die Verknüpfung jeder Zeile der einen Tabelle mit jeder Zeile der anderen Tabelle:
Das kartesische Produkt A x B der beiden Mengen $A = x, y, z$ und $B = 1, 2, 3$ ist:
A x B = (x,1), (x,2), (x,3), (y,1), (y,2), (y,3), (z,1), (z,2), (z,3)

Angewandt auf die zwei Tabellen ergibt es:

```
MariaDB [joinTesting]> SELECT * FROM Angestellter CROSS JOIN Abteilung;
+-----------+-------------+-------------+---------------+
| Nachname  | AbteilungID | AbteilungID | AbteilungName |
+-----------+-------------+-------------+---------------+
| Müller    |          31 |          31 | Verkauf       |
| Müller    |          31 |          33 | Technik       |
| Müller    |          31 |          34 | Büro          |
| Müller    |          31 |          35 | Marketing     |
| Schmidt   |          33 |          31 | Verkauf       |
| Schmidt   |          33 |          33 | Technik       |
| Schmidt   |          33 |          34 | Büro          |
| Schmidt   |          33 |          35 | Marketing     |
| Schneider |          33 |          31 | Verkauf       |
| Schneider |          33 |          33 | Technik       |
| Schneider |          33 |          34 | Büro          |
| Schneider |          33 |          35 | Marketing     |
| Fischer   |          34 |          31 | Verkauf       |
| Fischer   |          34 |          33 | Technik       |
| Fischer   |          34 |          34 | Büro          |
| Fischer   |          34 |          35 | Marketing     |
| Weber     |          34 |          31 | Verkauf       |
| Weber     |          34 |          33 | Technik       |
| Weber     |          34 |          34 | Büro          |
| Weber     |          34 |          35 | Marketing     |
| Meyer     |        NULL |          31 | Verkauf       |
| Meyer     |        NULL |          33 | Technik       |
| Meyer     |        NULL |          34 | Büro          |
| Meyer     |        NULL |          35 | Marketing     |
+-----------+-------------+-------------+---------------+
```

7.4.7.2 Inner Join - Bei einem INNER JOIN handelt es sich um eine gefilterte Form des CROSS JOINs, bei der in der Ergebnismenge nur die Tupel beider Ausgangstabellen zusammengeführt werden, die die vom Anwender definierte Selektionsbedingung erfüllen.

Für einen INNER JOIN müssen in jeder Zeile der zwei zu joinenden Tabellen übereinstimmende Spalten-Werte vorhanden sein.
Dargestellt in einem Mengendiagramm:

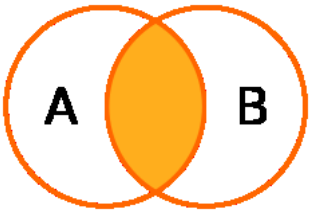


Abbildung 5: Inner Join ©Wikipedia

Angewandt auf das Angestellten - Abteilung - Beispiel können wir z.B. herausfinden welcher Angestellte in welcher Abteilung arbeitet:
Expliziter Inner Join:

```
MariaDB [joinTesting]> SELECT Angestellter.Nachname, Angestellter.AbteilungID, Abteilung.AbteilungNam
    -> FROM Angestellter
    -> INNER JOIN Abteilung ON
    -> Angestellter.AbteilungID = Abteilung.AbteilungID;
+-----------+-------------+---------------+
| Nachname  | AbteilungID | AbteilungName |
+-----------+-------------+---------------+
| Müller    |          31 | Verkauf       |
| Schmidt   |          33 | Technik       |
| Schneider |          33 | Technik       |
| Fischer   |          34 | Büro          |
| Weber     |          34 | Büro          |
+-----------+-------------+---------------+
```

Das ganze geht auch implizit mit den schon besprochenen SELECT FROM WHERE Statements z.B.:

```
MariaDB [joinTesting]> SELECT *
    -> FROM Angestellter, Abteilung
    -> WHERE Angestellter.AbteilungID = Abteilung.AbteilungID;
+-----------+-------------+-------------+---------------+
| Nachname  | AbteilungID | AbteilungID | AbteilungName |
+-----------+-------------+-------------+---------------+
| Müller    |          31 |          31 | Verkauf       |
| Schmidt   |          33 |          33 | Technik       |
| Schneider |          33 |          33 | Technik       |
| Fischer   |          34 |          34 | Büro          |
| Weber     |          34 |          34 | Büro          |
+-----------+-------------+-------------+---------------+
```

Auffallen sollte bei beiden Möglichkeiten, das Herr Meyer und die Abteilung 35 Marketing fehlen. Die zeigt deutlich, dass ein INNER JOIN nur bei Datenbanken zulässig ist, die referentielle Integrität erzwingen oder in Fällen, in denen die Join-Spalten mit Sicherheit keine NULL-Werte enthalten.

7.4.7.3 Left Outer Join - LEFT JOIN; Die gejointe Tabelle enthält jede Zeile, selbst wenn diese keine Paare bildet. Je nachdem welche Zeilen der Tabellen erhalten bleiben, ist ein OUTER JOINs ein LEFT OUTER-Join, RIGHT OUTER-Joins oder ein FULL OUTER-Join. Die Ausdrücke LEFT, RIGHT and FULL stehen für die rechte, linke oder beide Seite(n) des JOIN-Schlüsselworts.

Das Ergebnis eines LEFT OUTER JOINs (oder vereinfacht LEFT JOIN) der Tabellen A und B enthält alle Zeilen der linken ("LEFT") Tabelle A, selbst wenn die Join-Bedingung auf keine Spalte der rechten ("RIGHT") Tabelle B zutrifft.
Falls die ON-Bedingung für eine vorgegebene Zeile der Spalte A keine Zeile für die Tabelle B liefert, generiert der Join trotzdem die Zeile der Spalte A. Die entsprechenden Einträge für die Spalten der Tabelle B sind jedoch mit dem NULL-Wert ausgegeben. Ein LEFT OUTER JOIN gibt alle Werte eines INNER JOINs wieder zusätzlich aller Werte einer LEFT-Tabelle, welche nicht der RIGHT-Tabelle entsprechen und alle Zeilen mit NULL-Werten (also leeren Werten) der linken Spalte.

Dargestellt in einem Mengendiagramm:

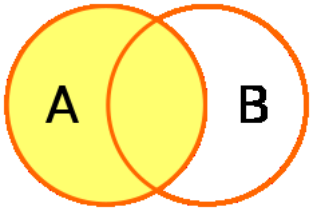


Abbildung 6: Left Outer Join ©Wikipedia

```
MariaDB [joinTesting]> SELECT *
    -> FROM Angestellter
    -> LEFT OUTER JOIN Abteilung ON Angestellter.AbteilungID = Abteilung.AbteilungID;
+-----------+-------------+-------------+---------------+
| Nachname  | AbteilungID | AbteilungID | AbteilungName |
+-----------+-------------+-------------+---------------+
| Müller    |          31 |          31 | Verkauf       |
| Schmidt   |          33 |          33 | Technik       |
| Schneider |          33 |          33 | Technik       |
| Fischer   |          34 |          34 | Büro          |
| Weber     |          34 |          34 | Büro          |
| Meyer     |        NULL |        NULL | NULL          |
+-----------+-------------+-------------+---------------+
```

7.4.7.4 Right Outer Join - RIGHT JOIN; ähnelt stark einem LEFT OUTER JOIN mit umgekehrter Abarbeitung der Tabellenreihenfolge. Jede Zeile der rechten ("RIGHT") Tabelle B kommt in der gejointen Tabelle mindestens einmal vor. Im Falle keiner passenden Zeile der linken ("LEFT") Tabelle A erscheint der NULL-Wert in den Spalten von A für die Zeilen, welche keine Entsprechung in B haben.

Dargestellt in einem Mengendiagramm:

Ein RIGHT OUTER JOIN gibt alle Werte der rechten ("RIGHT") Tabelle und passenden Werte der linken ("LEFT") Tabelle wieder (im Falle keiner passendenen Join-Prädikate NULL). Daher können wir zum Beispiel jeden Angestellten und seine Abteilung finden, aber dennoch Abteilungen anzeigen lassen, die keine Angestellten besitzen. Weiter unten ist ein Beispiel eines RIGHT OUTER JOINs (das OUTER-Schlüsselwort ist optional):

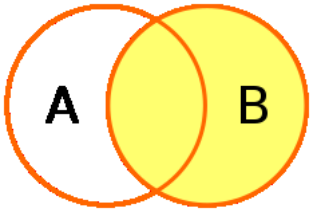


Abbildung 7: Right Outer Join ©Wikipedia

```
MariaDB [joinTesting]> SELECT *
    -> FROM Angestellter
    -> RIGHT OUTER JOIN Abteilung ON Angestellter.AbteilungID = Abteilung.AbteilungID;
+-----------+-------------+-------------+---------------+
| Nachname  | AbteilungID | AbteilungID | AbteilungName |
+-----------+-------------+-------------+---------------+
| Müller    |          31 |          31 | Verkauf       |
| Schmidt   |          33 |          33 | Technik       |
| Schneider |          33 |          33 | Technik       |
| Fischer   |          34 |          34 | Büro          |
| Weber     |          34 |          34 | Büro          |
| NULL      |        NULL |          35 | Marketing     |
+-----------+-------------+-------------+---------------+
```

RIGHT und LEFT OUTER JOIN sind funktional äquivalent. Keiner von beidem besitzt eine Funktionalität, welche der andere nicht hat. Sie können sich also durch eine Vertauschung der Tabellenreihenfolge gegenseitig ersetzen.

7.4.7.5 Full Outer Join ; Ein FULL OUTER JOIN verbindet die Resultate eines LEFT und eines RIGHT OUTER JOINs. Nicht passende Zeileneinträge eines FULL OUTER JOINs erhalten NULL-Werte für jede Spalte der Tabelle, denen eine passende Zeile fehlt. Passende Zeilen erzeugen eine einzelne Zeile im Ergebnis-Set, welche die Spalten beider Tabellen erhält. Zum Beispiel können wir dadurch jeden Angestellten anzeigen, der einer Abteilung zugewiesen ist, und jede Abteilung, die einen Angestellten hat, aber zusätzlich auch Angestellte, die nicht Teil einer Abteilung sind, oder Abteilungen, die keine Angestellten haben.

Beispiel für einen FULL OUTER JOIN (das OUTER-Schlüsselwort ist optional):

```
SELECT * FROM Angestellter FULL OUTER JOIN Abteilung ON
    -> Angestellter.AbteilungID = Abteilung.AbteilungID;
```

Dargestellt in einem Mengendiagramm:

Manche Datenbanksysteme (darunter MySQL und MariaDB) unterstützen die Funktionalität des FULL OUTER JOINs nicht direkt. Sie können diesen aber durch den Gebrauch eines INNER JOINs und eines UNION ALL imitieren, indem sie die alleinstehenden Zeilen der rechten ("RIGHT") und der linken ("LEFT") Tabelle sowie den INNER JOIN mit einem UNION ALL verbinden.
Das gleiche Beispiel sieht dann wie folgt aus:

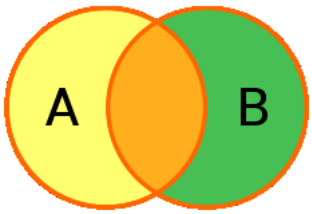


Abbildung 8: Full Outer Join ©Wikipedia

```
MariaDB [joinTesting]> SELECT * FROM Abteilung LEFT JOIN Angestellter ON Abteilung.AbteilungID = Ange
    -> UNION ALL SELECT * FROM Abteilung RIGHT JOIN Angestellter ON
    -> Abteilung.AbteilungID = Angestellter.AbteilungID WHERE Abteilung.AbteilungID IS NULL;
+-------------+--------------+-----------+-------------+
| AbteilungID | AbteilungName | Nachname  | AbteilungID |
+-------------+--------------+-----------+-------------+
|          31 | Verkauf       | Müller    |          31 |
|          33 | Technik       | Schmidt   |          33 |
|          33 | Technik       | Schneider |          33 |
|          34 | Büro          | Fischer   |          34 |
|          34 | Büro          | Weber     |          34 |
|          35 | Marketing     | NULL      |        NULL |
|        NULL | NULL          | Meyer     |        NULL |
+-------------+--------------+-----------+-------------+
```

8 SQL Injection

SQL-Injection bezeichnet das Ausnutzen einer Sicherheitslücke in Zusammenhang mit SQL-Datenbanken, bei der versucht wird über manipulierte Benutzereingaben Datenbankbefehle direkt an die Datenbank zu senden. Damit können eventuell Daten ausgespäht, verändert, hinzugefügt oder gelöscht werden.

SQL-Injection ist nur dann möglich, wenn der Userinput direkt zum SQL-Interpreter gelangt!

8.1 SQL-Injection Beispiel

Angenommen es existiert folgender Code, eingebettet in eine Webseite mit einem Eingabefeld für die UserID:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

txtSQL wird im Anschluss direkt an den SQL-Interpreter übergeben, ohne das die Variable *txtUserId* überprüft wird.

8.1.1 Always true

Der naive Programmierer hat angenommen, dass die User der Webseite nur IDs (integer) in Eingabefeld eingeben, aber was passiert wenn folgendes eingegeben wird:
txtUserId = "42198 OR 1=1"
Dann schaut das SQL-Statement plötzlich folgendermaßen aus:

```
txtSQL = "SELECT * FROM Users WHERE UserId = 42198 OR 1=1";
```

Nun es mag möglich sein, dass ein User mit der ID=42198 existiert aber wenn nicht wird das 1=1 als True interpretiert und man bekommt ALLE Einträge der Users Tabelle!

8.1.2 Batched SQL-Statements

Wenn mehrere SQL-Statements in einer Zeile (getrennt durch ;) an den Interpreter übergeben werden, spricht man von SQl-batch. Dieser wird vom Interpreter akzepiert und abgearbeitet. Leider kann dies durch unüberprüfte eingaben ebenfalls leicht ausgenützt werden:
txtUserId = "42198; DROP TABLE Users"
Dann schaut das SQL-Statement plötzlich folgendermaßen aus:

```
txtSQL = "SELECT * FROM Users WHERE UserId = 42198; DROP TABLE Users";
```

Und das wars dann mit der Users-Tabelle ;-)

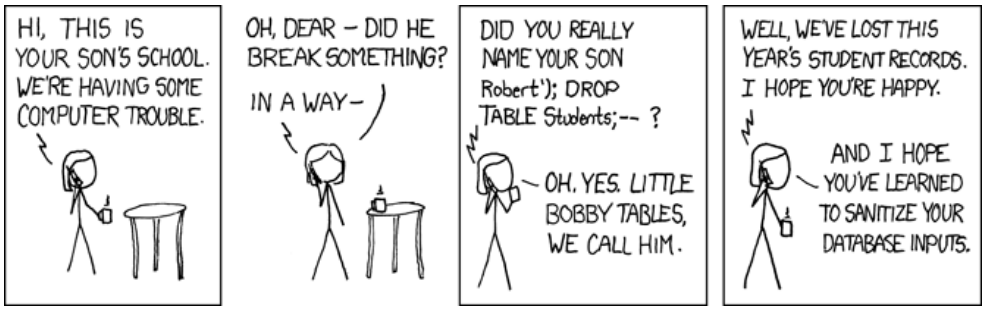


Abbildung 9: Funny ©https://xkcd.com/327/

8.2 Prepared-Statements

Prepared Statements sind eine Möglichkeit um SQL-Injectin zu vermeiden. Die Grundidee ist, ein SQL-statement als eine Art Vorlage an das DBMS zu senden, in dem zwar zu erwartende Parameter definiert sind, die eigentlichen Werte aber noch fehlen.
Diese fehlenden Parameterwerte werden seperat an das DBMS übermittelt und falls durch SQL-Injection verfälschte Werte das ursprüngliche Prepared Statement verändern würden, wird es nicht ausgeführt.
Beispielablauf:

- Zuerst wird as prepared Statement erzeugt, wobei unspezifizierte Parameter mit einem ? gekennzeichnet sind (je nach verwendeter Programmiersprache eventuell andere Zeichen) und dem DBMS übermittelt:

  ```
  INSERT INTO Mitarbeiter (ie, vorname, nachname, abteilung) VALUES (?,?,?,?);
  ```

- Das DBMS übersetzt bzw. parsed das Statement, führt es aber noch nicht aus.
- Später übergibt (binds) die Applikation dann die Werte für die Parameter im Prepared Statement und die Anweisung wird ausgeführt, sofern sich die ursprüngliche Syntax des Statements nicht verändert hat - SQL-Prevention!

Beispiel-Code in Java:

```
PreparedStatement ps = connection.prepareStatement(
    "SELECT user, password FROM tbl_user WHERE (user=?)"); // Statement wird erzeugt
ps.setString(1, username); // Parameter werden übergeben
ResultSet rs = ps.executeQuery(); //Statement wird ausgeführt.
```

Prepared Statements bieten nicht nur Sicherheit gegenüber SQL-Injection, sondern können eventuell auch ein Performance Vorteil sein. Wenn z.B. in der App eine Schleife vorkommt ist es schneller ein Prepared Statement zu machen, und in der Schleife nur die Werte zu übergeben - dadurch wird das statement nur einmal vom SQL-Interpreter übersetzt!
Umgekehrt kann es ein Performance Nachteil sein, wenn ein Statement nur einmal ausgeführt wird, weil man zweimal mit dem Server kommunizieren muss, einmal um das Prepared Statement zu übergeben und dann nocheinmal um die Werte nachzureichen.

9 Python und Datenbanken

Im folgenden wird mit Hilfe von Beispielcodes erklärt wie man mittels Python mit einer Datenbank interagiert.
Es gibt mehrer Python Module, welche eine Kommunikation mit SQL-Datenbanken ermöglichen. In diesem Skript wird das Modul:
mysql-connector-python
verwendet.

9.1 Installation des mysql-connector-moduls

Das Modul kann via pip installiert werden:

```
pip install mysql-connector-python
```

9.2 Code-Beispiele und Tutorials

Es gibt auf der Homepage mysql-connector-python
Code-Beispiele: coding-examples
und ein Tutorial: tutorial
Ein weiteres einfaches Beispiel eines Zahlenratespiels, das die Highscore in einer Datenbank speichert:

```
'''
Ein erstes Python Programm um mit einer
Datenbank zu kommunizieren.
Kleines Zahlenrate-Spiel mit fixen Regeln und
Usernamen - User Score wird in die Datenbank
geschrieben.

Das Beispiel geht davon aus, das auf der gleichen Maschine eine
MariaDB Datenbank läuft für die es einen User "markus" mit einem
Password "markus" gibt.

notwendige Installationen unter Linux:
sudo apt install python3-mysqldb

notwendige Installation des Mysql-Connector Moduls (für alle Betriebssystem
mittels pip)
pip3 install mysql-connector-python

no (c) Markus
September 2023
'''

import random
import mysql.connector as mariadb

#keep the db-setup seperate from the rest of the code!
def database_setup():
    #optional add the database name below to auto connect
    mydb = mariadb.connect(
        host="localhost",
        user='markus',
        password='markus'
    )

    #create a cursor object - it will be used to interact with the database
    mycursor = mydb.cursor()

    #prepare the database for the guessing game i.e. create the database
    mycursor.execute("CREATE DATABASE IF NOT EXISTS guessing_game_high_score")
    #check if it worked?
    #mycursor.execute("SHOW DATABASES")
    #for x in mycursor:
    #    print(x)

    mycursor.execute("USE guessing_game_high_score")
```

```
    #create the table to store the highscore
    mycursor.execute("""CREATE TABLE IF NOT EXISTS highscore
                    (name VARCHAR(100) PRIMARY KEY,
                     guesses_nr INT)""")
    #check if it worked?
    #mycursor.execute("SHOW TABLES")
    #for x in mycursor:
    #    print(x)

    #add some dummy user to the table (attention: %s is necessary for the courser)
    #IGNORE means that entries which already exist will not be overwritten
    sql = "INSERT IGNORE INTO highscore (name, guesses_nr) VALUES (%s,%s)"
    val =[
        ('Sepp', 101),
        ('Franz', 111),
        ('Ignazius', 121)
        ]

    #now we use executemany - one sql statement with multiple values
    #think of it like a loop which calls the sql statement with three
    #different values
    mycursor.executemany(sql,val)

    #commit makes sure the data gets written to the database (could still
    #be in a buffer)
    mydb.commit()

    #check if it worked?
    #mycursor.execute("SELECT * FROM highscore")
    #for x in mycursor:
    #    print(x)

    #return the coursor for use in the main programm
    return mycursor

# function to show the current highscore in a nice format
def show_highscore(mycursor):
    print()
    mycursor.execute("SELECT * FROM highscore ORDER BY guesses_nr")
    print("##########################################")
    print("###### The current highscore list  ######")
    print("##########################################")
    print("#{0:^19}#{1:^19}#".format("Name","Number of guesses"))
    print("##########################################")
    for name, guesses_nr in mycursor:
        print("#{0:^19}#{1:^19}#".format(name, guesses_nr))
    print("##########################################")
    print()

# function to update the highscore
def update_highscore(mycursor, name, number):
    sql = "INSERT IGNORE INTO highscore (name, guesses_nr) VALUES (%s,%s)"
    val = (name, number)
    mycursor.execute(sql,val)

    sql = "SELECT guesses_nr FROM highscore WHERE name = '{}'".format(name)
    mycursor.execute(sql)
    for guesses_nr in mycursor:
        old_score = guesses_nr[0]
        print("did it work? guesses_nr={}".format(old_score))
        if (old_score > number):
            sql = "UPDATE highscore SET guesses_nr = '{}' WHERE name = '{}'".format(number,name)
            mycursor.execute(sql)
    mydb.commit()
```

```
########## main gussing game program starts here ##########
mycoursor = database_setup()
print("### Welcome to the guessing game! ###")

show_highscore(mycursor)

print("The secret number lies between 0 - 20")
name = input("Please enter your name:")
print("Okay {}, lets start!".format(name))
counter = 1
secret = random.randint(0,20)
while(True):
    number = int(input("Please enter your {}. guess:".format(counter)))
    if (number < secret):
        print("My number is greater!")
    elif (number > secret):
        print("My number is smaller!")
    else:
        print("Well done {}! You got it in just {} tries!".format(name, counter))
        break
    counter += 1

update_highscore(mycursor, name, counter)
show_highscore(mycursor)
```