Zusammenfassung Datenbanksysteme

Inhalt

[Einführung 3](#_Toc502821962)

[Einsatz der Datenbanken 3](#_Toc502821963)

[Einführung 3](#_Toc502821964)

[Nutzung von Datenbanken 3](#_Toc502821965)

[Einsatz der Datenbanken 3](#_Toc502821966)

[Information 3](#_Toc502821967)

[Daten 3](#_Toc502821968)

[Datenbasis(DB) 3](#_Toc502821969)

[Datenbanksystem / Datenbank(DBS) 4](#_Toc502821970)

[Datenbankmanagementsystem(DBMS) 4](#_Toc502821971)

[Datenbankmodell 4](#_Toc502821972)

[Funktionen DBMS 4](#_Toc502821973)

[Datenmodellierung 6](#_Toc502821974)

[DB-Entwurfsprozess 6](#_Toc502821975)

[Benutzer eines DBMS 7](#_Toc502821976)

[Entity-Relationship-Modell(ER) 7](#_Toc502821977)

[Unified Modelling Language(UML) 7](#_Toc502821978)

[ANSI-3-Ebenen-Modell 9](#_Toc502821979)

[Datenbankenwurf 10](#_Toc502821980)

[Datenmodellierung 10](#_Toc502821981)

[Konzeptionelles Datenmodell 13](#_Toc502821982)

[Relationales Datenmodell 13](#_Toc502821983)

[Datenbankentwurf 14](#_Toc502821984)

[Vom Konzeptionellen zum Logischen DB-Entwurf 14](#_Toc502821985)

[Vom Logischen zum Physischen DB-Entwurf 21](#_Toc502821986)

[Structures Query Language(SQL) 22](#_Toc502821987)

[Data Definition Language(DDL) 22](#_Toc502821988)

[Data Manipulation Language(DML) 25](#_Toc502821989)

[Data Control Language(DCL) 39](#_Toc502821990)

[Transaktionen / Isolation 42](#_Toc502821991)

[Grundsätzliches 42](#_Toc502821992)

[ACID 42](#_Toc502821993)

[Nutzen von Transaktionen 42](#_Toc502821994)

[Transaktionen – Befehle 42](#_Toc502821995)

[Serialisierbarkeit 43](#_Toc502821996)

[Umsetzung der Isolation 45](#_Toc502821997)

[Datenbankarchitektur - Durability / Dauerhaftigkeit 52](#_Toc502821998)

# Einführung

## Einsatz der Datenbanken

### Einführung

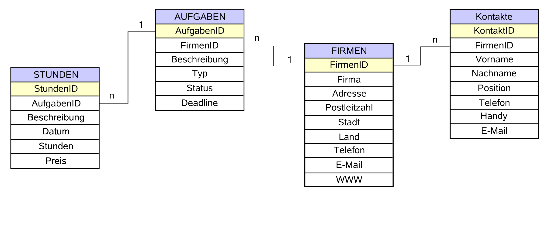
Wieso braucht es Datenbanken? Diese Frage stellen sich wahrscheinlich einige Leute(die diesen Kurs besuchten). Nun Fakt ist, dass ein modernes Leben ohne Datenbanken nicht mehr wegzudenken ist. Die Festplattenkapazität sowie auch die Rechenleistung zeigt ein exponentielles Wachstum. Immer mehr Daten, immer schnellere Rechner, das ist was für uns zählt.  
Natürlich müssen die Daten abgespeichert und verarbeitet werden. Hier kommen die Datenbanken zum Zuge, mittels Ihrem Einsatz können Daten gespeichert, strukturiert und ausgewertet werden.

### Nutzung von Datenbanken

Fast jede (moderne) Applikation benutzt im Hintergrund eine Datenbank, wo sie Daten abspeichern, verwalten und wieder zugreifen kann. Die Genialität hinter einer Datenbank ist, dass sich der Programmierer der Software nicht um die Details kümmern muss, sondern diese Aufgabe einem anderen Techniker zugewiesen werden kann.  
Jeden Tag kommen wir in Kontakt mit Datenbanken:

* Handy Kontakte
* SBB Datenbanken (Fahrplan / Personnenabonnemente)
* Google Abfragen
* viele mehr

### Einsatz der Datenbanken

Wie bereits erwähnt lassen sich in Datenbanken Daten strukturiert verarbeitet. Der Vorteil gegenüber einer simplen Applikation(Beispiel: Java Collections) ist, dass sich eben diese strukturierten Datenstruktur benutzen kann:  
Datenbanken bestehen aus verschiedenen Tabellen mit verschiedenen Spalten, Zeilen und daraus resultierenden Felder. Somit lassen sich abgespeicherte Daten individuell manipulieren und Datensätze miteinander verknüpfen. Datenbanken sind sehr mächtig, da sich komplizierte Strukturen relativ einfach abbilden lassen können, was mit reinen Software nicht oder nur sehr aufwendig realisiert werden könnte.

### Information

Informationen sind vom Empfänger interpretierte Daten, sprich der Computer kann nichts mit diesen Daten anfangen. Informationen sind jedoch für ein Unternehmen immens wichtig: Sie können beispielsweise Produktionsgrundlagen beinhaltet, Steuerung von Maschinen, Kontrolle von Prozesse oder beispielsweise Rezepte eines Produkts.

### Daten

Sind maschinell verarbeitbar, sprich die Computer können diese bearbeiten. Jedoch können die Maschinen die Daten nicht oder nur begrenzt beurteilen(nach wahr/falsch oder wichtig/unwichtig). Daten müssen auf einem physikalischen Datenträger gespeichert werden und haben daher eine «begrenzte» Lebensdauer. Verwaltung von Daten meist aufwendig und pflegeintensiv.

### Datenbasis(DB)

Eine Datenbasis(auch Datenbestand, Datensatz) ist eine strukturierte Ansammlung von Daten. Diese beinhalten in der Regel abwendungsspezifische Informationen.

### Datenbanksystem / Datenbank(DBS)

Ein Datenbanksystem besteht aus einem Datenbankmanagementsystem(DBMS) und einer oder mehreren Datenbasis(DB). Der Begriff Datenbank ist ein Synonym für Datenbanksystem.

### Datenbankmanagementsystem(DBMS)

Das Datenbankmanagementsystem ist zuständig für das Verwalten der verschiedenen Datenbasen. Anwendungen senden Datenanfragen an das Datenbankmanagementsystem, welches die Daten den verschiedenen Applikationen zur Verfügung stellt. Es beinhaltet anwendungsunabhängige Dienste und somit haben Änderungen der Datenorganisation kein Einfluss auf Applikationen. Es beherrscht mächtige Abfragemöglichkeiten und mehrere Sessions(mehrere Applikationen/Clients). Redundanzfreiheit, Datenintegrität, Datenpflege, Datenschutz und -sicherheit können vom DBMS sichergestellt werden:

#### Redundanzfreiheit

Jedes Datenelement soll nur einmal gespeichert werden(pro Datenbasis)

#### Datenintegrität

Die Richtigkeit(auch Qualität und Zuverlässigkeit) der Daten:

* Datenkonsistenz: korrekte und widerspruchsfreie Daten
* Datensicherheit: Sicherung vor physischem Verlust
* Datenschutz: Schutz vor unberechtigtem Zugriff der Daten



Zwei Tabellen mit inkonsistenten(widersprüchlichen) Daten

#### Merkformel

DBMS + (n \*) DB

### Datenbankmodell

Das Datenbankmodell legt das Datenstruktur-Konzept des DBMS fest. Das bedeutet wie sind die Beziehungen von Tabelle, wie werden Daten gespeichert, welche Datentypen gibt es.

### Funktionen DBMS

* Transaktionen
  + Operationen, welche die Konsistenz der Daten garantiert
* Mehrbenutzerbetrieb
  + Erlaubt mehrere Sessions/Clients/Benutzer
* Sicherheit
  + Schutz vor unberechtigtem Zugriff: z.B durch Rechntevergabe
* Backup & Recovery
  + Sicherung/Wiederherstellung der Daten
* Generische Datenstrukturen
  + Definition von Datenstrukturen
  + Datenkatalog(Metadaten, Daten Dictionary)

#### Verschiedene Datenbankmodelle

##### Hierarchisches Datenbankmodell

###### Grundlegendes

* Speicherung der Daten in Hierarchie.
* Beziehungen innerhalb der Daten gespeichert

###### Nachteil

* Komplexe Schema lassen sich nicht abbilden
* Keine Trennung zwischen Beziehung und Daten
* Änderungen verursachen mühsames Anpassen der Hierarchie

###### Beispiel

* XML

##### Netzwerkmodell

* Speicherung in vernetzten Hierarchien
* Beziehungen zusammen mit Daten gespeichert
* Mehrfachbeziehung möglich(komplizierte Modellierung nötig)

###### Nachteil

* Abfragen ist kompliziert
* Unübersichtlich
* Kenntnis komplexe Netzwerk-Datenstrukturen müssen vorhanden sein

###### Beispiel

Praktisch nicht mehr im Einsatz

##### relationales Modell

* Daten in Tabellen(=Relationen) gespeichert
* Flexible, vielseitige Datenabfragen möglich
* Klare Trennung zwischen Anwendung und Daten möglich

###### Nachteil

* Impedance Mismatch(Unverträflichkeit Programmiersprache – Relationale Datenbank)
* Komplexe Abfragen 🡪 Performance Probleme

###### Beispiel

* Alle gängigen Datenbanksysteme

##### Objektrelationale Modelle

* Erweiterungen des relationalen Datenmodells(Typen, Tabellenvererbung etc.)
* Verschachtelungen von Tabellen
* Speicherung von Objekten und Methoden in Datenbank

###### Beispiel

* Oracle

##### Objektorientiertes & NoSQL & NewSQL

* Neuartige Ideen und Funktionalitäten
* Marktanteil sehr gering

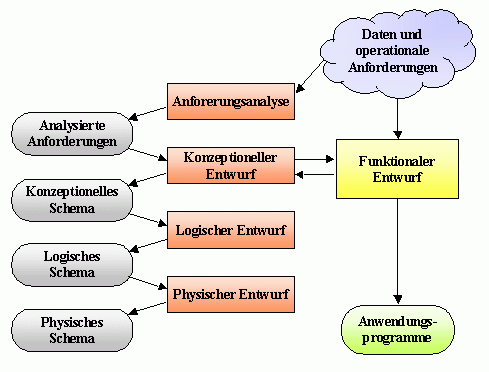
###### Nachteil

* Meistens nur für Spezialanwendung
* Noch nicht sehr verbreitet

# Datenmodellierung

## DB-Entwurfsprozess

1. Anforderungsanalyse 🡪Anforderungsspezifikation
   1. Welche Anforderungen hat das System?
   2. Bestimmung relevanter Informationen durch Systemanalyse
2. Konzeptueller DB-Entwurf 🡪 Konzeptuelles Datenmodell
   1. Überführung der Anforderungsspezifikation in logisches Modell.
   2. Abbildung der Objekte und Beziehungen mittels UML/ER
   3. Unabhängig von eingesetzter DBMS
3. Logischer DB-Entwurf 🡪Logisches Datenmodell
   1. Überführung des logischen Modells(UML) in Datenbankmodell(bei uns relationales Modell)
   2. Bei uns: Modellierung der Datenbasis mittels relationaler Schreibweise
   3. Modell angepasst auf eingesetztes DBMS
4. Physischer Entwurf 🡪 Physischem Datenbankmodell(Defintion der DB-Objekte in SQL)
   1. Abklärungen/Planung hardwaretechnischer Natur
      1. Auf welchem System und wie soll meine DB laufen(Cluster, Loadbalancing …)
      2. Optimierungsfragen(Index, etc.)
      3. Einstellungen / Konfiguration DBMS

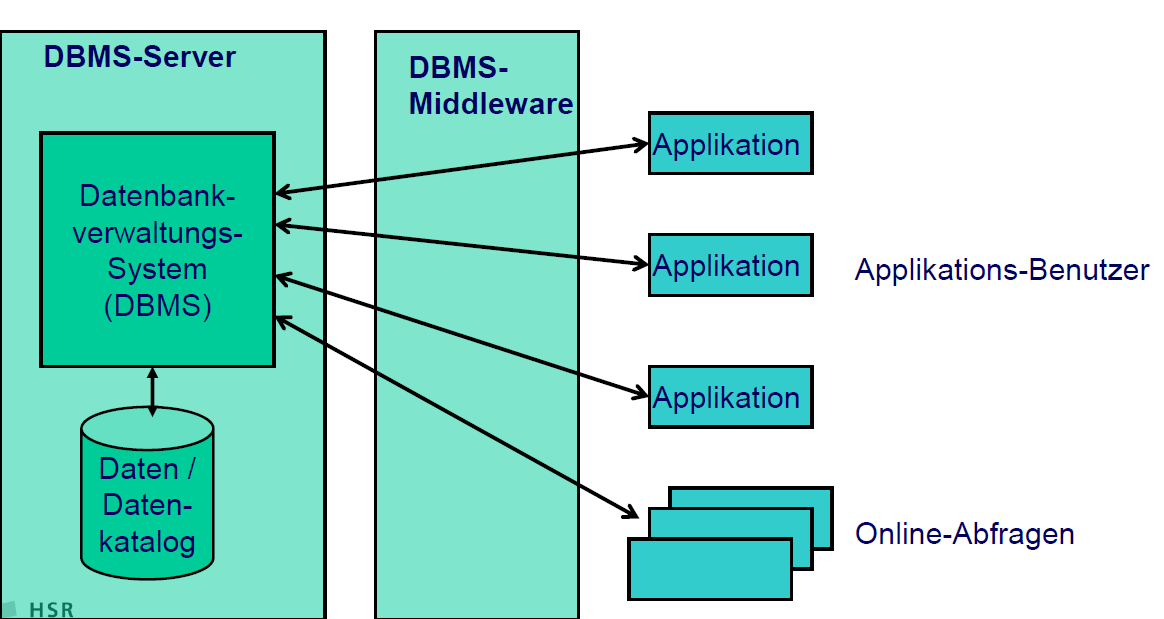


## Benutzer eines DBMS

Es gibt verschieden Benutzer von DBMS:

* Datenbankadministrator(Superuser)
  + Zuständiger für DBMS, Zugriffsrechte, Wartung, etc.
* Fortgeschrittener Benutzer(Experte)
  + Datenbankwisse, kann Queries absetzten
* Applikations-Programmierer
  + Schreibt Software, welche Datenbasis verwendet
* Angelernter Benutzer (Applikations-Benutzer)
  + Benutzt Applikationen

Natürlich sind diese strikte Trennung der Rollen heute nicht mehr so konsequent. Jeder Software Engineer kennt sich auch mit den Abfragen aus, etc.



### Entity-Relationship-Modell(ER)

Das Model zeigt verschiedene Entitätsmengen und deren Assoziationen untereinander. Es kann verwendet werden um das logische Datenmodell zu konstruieren, ist jedoch nicht so mächtig wie das UML.

### Unified Modelling Language(UML)

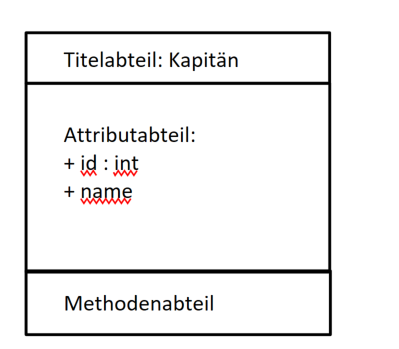
Wird verwendet um das konzeptionelle Modell zu erarbeiten.

### Objektdiagramm(nicht verwendet)

Das Objektdiagramm ist die grafische Darstellung verschiedener Objekte mit deren Attribute.

#### Klassendiagram

Das Klassendiagramm der Unified Modelling Language, kurz UML, wird verwendet, um die verschiedenen Klassen und deren Attribute und Methoden(Operationen) grafisch darzustellen. Ebenfalls lassen sich die Verbindungen(=Assoziationen) der verschiedenen Klassen zeichnen. (Vergleiche dazu «konzeptionelles Datenmodell»)

Klassendiagramme = ER-Diagramme + Operationen (Entity-Relationship-Modell)

Es ist somit eine einfache und übersichtliche Darstellung, bei der man noch nicht an die Datenbank denken muss.

### Vorteil / Nachteil

* Standardisiert
* Unterstützt von vielen Tools
* Noch nicht überall bekannt

#### Bildergebnis für uml notationDie Vererbung(Inheritance)

##### Begriffsdefinition

Haben mehrere Klassen gemeinsame Eigenschafften(Attribute, Operationen, Assoziationen), dann werden diese in einer Oberklasse(Generalisierung) zusammengefasst.

Subklassen(Spezialisierung) erben Eigenschaften der Oberklassen

Die Beziehung zwischen Spezialisierung und Generalisierung nennt man «is-a» oder zu deutsch «ist-ein(e)».

Variante 1:

Eine Tabelle pro Sub- & Superklasse.

#### Attribute

##### Disjoint

Ein Objekt kann genau zu einer Unterklasse gehören.

##### Overlapping

Ein Objekt kann zu mehreren Unterklassen gehören.

##### Complete

Alle Subklassen sind definiert.

##### Incomplete

Noch nicht alle Subklassen sind definiert, es können neue Subklassen hinzukommen.

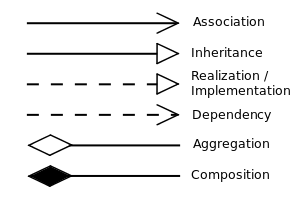
Beispiel:



Disjoint: Ein Angestellter kann nur einem Fachgebiet zugeteilt sein.

Complete: Alle Fachgebiete sind definiert, es kann kein neues Fachgebiet hinzukommen.

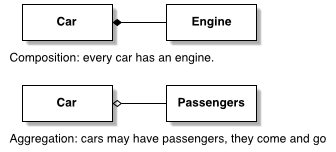
Overlapping: Ein Informatiker kann mehr als eine Funktion annehmen(z.B: Operator und SW-Engineer)

Incomplete: Es sind noch nicht alle Funktionen des Informatikers definiert, es können neue Funktionen definiert werden.

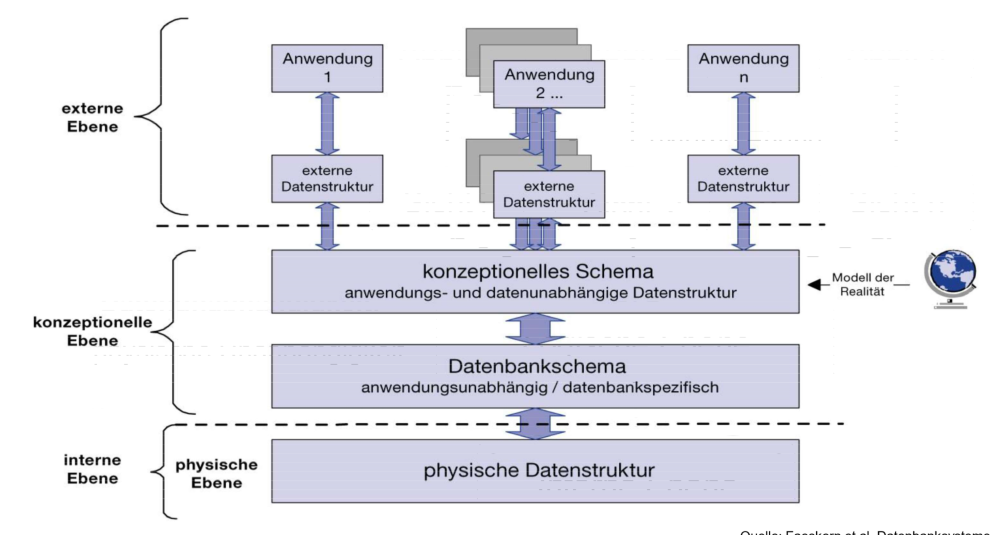
#### Delegation / Komposition / Aggregation

Diese Beziehung bedeutet «a part of» oder zu deutsch «ist ein Teil von». Der Unterschied zwischen Aggregation und Komposition ist folgender:

Die Aggregation kann alleine, dass heisst ohne Aggregationsobjekt, bestehen.

Die Komposition braucht ein Aggregationsobjekt, ohne dieses kann es nicht bestehen.

### ANSI-3-Ebenen-Modell



Mit dem ANSI-3-Ebenen-Modell ist es uns möglich die Daten zu entkoppeln. Somit ist man nicht mehr abhängig von einer Programmiersprache oder einem bestimmten Hardware-Typ. Es kann somit eine Schicht komplett verändert werden ohne das die anderen Schichten angepasst werden müssen.

#### 1. Ebene / externe Ebene

Die externe Ebene ist die 1. Ebene im ANSI-Modell. Sie beinhaltet die Benutzersichten auf die Daten. Es handelt sich um die Applikationen, welche den Benutzern zur Verfügung gestellt werden, um auf die Daten zuzugreifen.

#### 2. Ebene / konzeptionelle Ebene

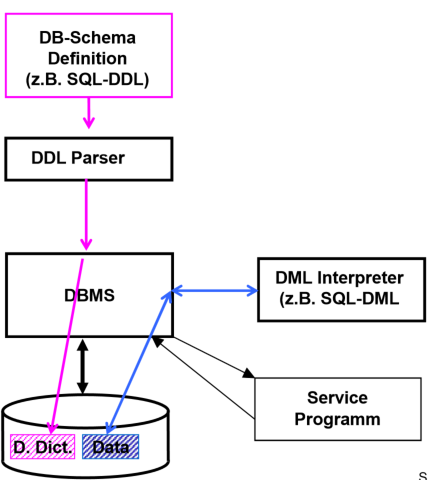
Diese Ebene beinhaltet die logische Darstellung/Struktur der Daten. In einem ersten Schritt wird das Konzeptionelle Schema im UML erstellt. Daraus wird das relationale Datenbankschema in der relationalen Schreibweise gefertigt, welches auf das Modell(relational, objektorientiert, etc.) passt.

3. Ebene / interne Ebene

Die interne Ebene stellt die physikalische Implementierung des DBS dar(Speicherstrukturen). Informationen über Art und Aufbau des phsysichen Systems.  
Wichtige Themen sind hier(Performance, Verfügbarkeit, Datenschutz und -sicherheit, & Optimierungen)

### Datenbankenwurf

Die Datenbank wird mittels DDL(Data Definition Language) erstellt.

### Datenmodellierung

#### Grundlagen konzeptionelles Modell

Es gibt verschiedene wichtige Grundbegriffe der Datenmodellierung.

##### Entität

Ein individuelles Element(auch konkretes Objekt), welches verschiedene Attribute(Merkmale) beinhaltet. Es ist dank den eindeutigen Merkmalen eindeutig identifizierbar von anderen Entitäten der Entitätsmenge.(beispielsweise ein Angestellter ist eine Entität, er kann anhand seiner Personalnummer von anderen Angestellten(Entitäten) unterschieden werden.)

Entitätstyp

Ein Entitätstyp ist definiert durch eine Zusammenfassung von verschiedenen Attributen(Merkmalen).(Beispielsweise ist der Entitätstyp «Angestellter» durch die Attribute «Personalnummer, Name, Geburtstag, Adresse , Lohn und Abteilung» definiert)

##### Entitätsmenge

Menge von Elementen(Entitäten), die gewisse gleiche Eigenschaften haben(z.B Wohnort = Zürich).

##### Attribut

Beschreibung/Name eines bestimmten Merkmales einer Entität.(Beispiel Wohnort)

##### Wertebereich

Der Wertebereich ist ein Bereich zulässiger Werte für eine Eigenschaft einer Entität.(Beispielsweise für das Attribut PLZ ist der Wertebereich 1000-9999)

##### Attributwert

Der Attributwert ist der Wert einer Eigenschaft einer Entität.(Beispielsweise ist «Zürich» ein möglicher Attributwert vom Attribut «Wohnort» von einer Entität)

##### Relation

Eine Relation ist eine Verknüpfung von mehreren Entitäten.(Beispielsweise Fritz Müller(Entität 1) gehört zur Abteilung Entwicklung(Entität 2))

Assoziation

Die Assoziation ist die Verknüpfung von mehreren Entitätstypen.(Beispiel Verbund von Tabelle Angestellter mit Tabelle Abteilung)

#### Grundlagen relationales Modell

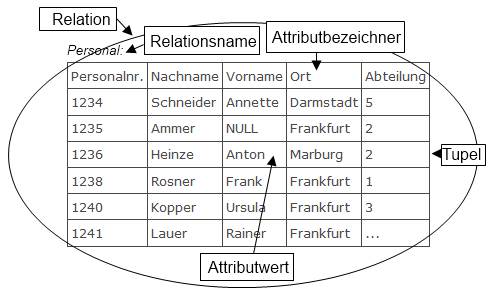
##### Entität / Relationsname

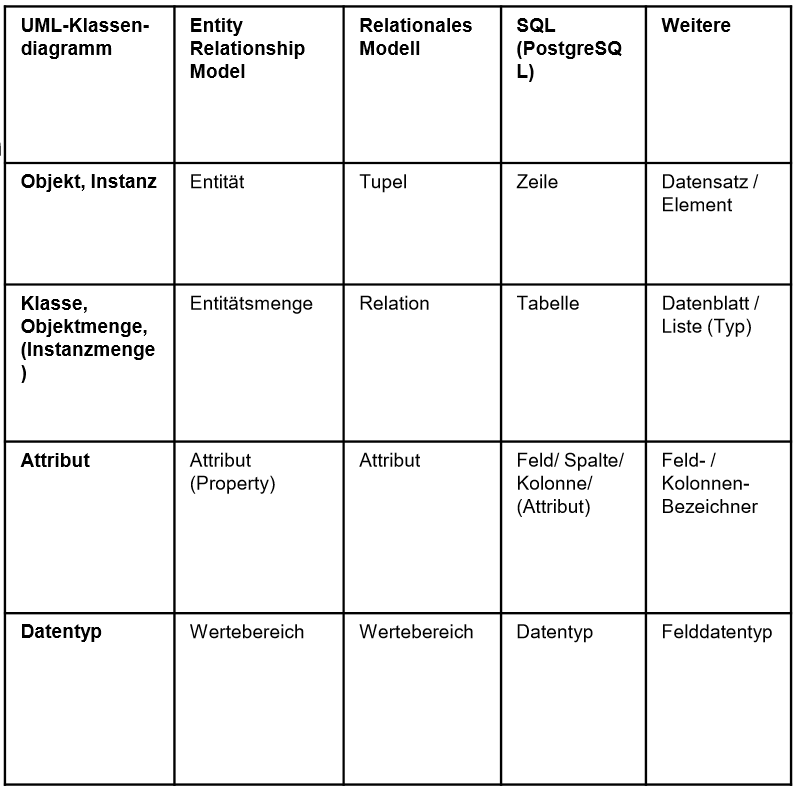
Eine Identität im relationalem Modell ist ein Individuelles Element des Systems(Als beispielsweise ein Angestellter ist eine Entität)

##### Relation

Eine Relation beschreibt Entitätstyp mit der zugehörigen Entitätsmenge(beispielsweise Tabelle Angestellter(Entitätstyp) mit Attributen(PersNr, Name, Lohn) und allen dazugehörigen Datensätze(= Tupel))

##### Tupel

Ein Tupel repräsentiert eine Entität.(Beispielsweise ein Angestellter:(1001, «Marxer», 12000))



### Konzeptionelles Datenmodell

Das Konzeptionelle Datenmodell oder auch konzeptuelles Datenmodell ist eine Darstellung als UML oder ER. Wichtig ist hierbei, dass das Konzeptionelle Datenmodell unabhängig von der eingesetzten Technologie ist.

* Schwierigste Aufgabe im Entwicklungsprozess
* Fehler können gravierende Folgen haben

Es definiert:

* Die Objekte und deren Eigenschaften
* Zusammenhänge und Beziehungen einzelner Objekte
* Legt Konsistenzbedingungen fest

### Relationales Datenmodell

Das Relationale Datenmodell ist die Anpassung vom konzeptionellem Datenmodell auf die Datenbank. Hierbei spielt das eingesetzte Database Managementsystem eine wichtige Rolle.

Wir erarbeiten uns das Relationale Modell mittels der relationalen Schreibweise.

Es basiert dabei auf dem mathematischen Konzept von Relationen(Verbindungen) und benutzt Tabellen zur Darstellung von Daten und Beziehungen.

Nachteil ist dabei, dass es einen Inpedance Mismatch geben kann zwischen DB-typen und Struktur der Programmiersprache.

Regeln für Tabellen

* Jede Tabelle beschreibt alle Entitäten einer Entitätsmenge
* Jede Zeile beschreibt eine Entität, welche einem Tupel entspricht.
* Jede Kolonne(Spalte) entspricht einen Attribut
* Jede Kolonne hat eine eindeutige Überschrift, sie entspricht dem Namen des Attributs
* Die Attribute müssen im Wertebereich liegen
* Reihenfolge der Kolonnen und Zeilen hat keine Bedeutung
* Jede Zeile(Datensatz, Tupel) ist verschieden(eindeutig identifizierbar) von anderen Zeilen
* 🡪 Zu jeder Tabelle existiert mindestens ein Schlüsselkandidat(welcher die Eindeutigkeit eines Tupels garantiert)

Schlüssel / Schlüsselkandidaten

* Ein Attribut oder eine Kombination von Attributen, dass ein Datensatz eindeutig identifizierbar macht.
* Dieser Primärschlüssel wird für Fremdschlüsselbeziehungen verwendet

Eigenschafen

Ein Primärschlüssel besteht grundsätzlich aus einem Attribut und hat folgende Eigenschaften. Manchmal muss man einen Primärschlüssel auch aus einzelnen Attributen zusammensetzten, da es kein einzelnes Attribut gibt, welches eiindeutig ist:

* Eindeutig
* Laufend zuteilbar
* Möglichst kurz, nicht zusammengesetzt
* Invariant(verändert sich nicht)

Manchmal ist es in einer Tabelle nicht möglich, ein Schlüssel zu kreiiren. Dann wird ein Surrogatschlüssel erzeugt. Es Surrogatschlüssel ist ein künstlicher Schlüssel, welcher nicht von Attributen der Tabelle abhängt.

Entitäts-Integrität

* NULL ist ein undefinierten Attribut-Wert. Der Wert eines Schlüssels darf nie NULL sein, da dieser dann nicht eindeutig ist.(Es könnten mehrere Schlüssel den Wert «leer»(NULL) haben.
* NULL und die Zahl 0 ist nicht das gleiche!
* Ob Attributwert NULL sein darf, kann bei Definition des Attributes gewählt werden

Fremdschlüssel

* Falls in einer Tabelle Attribute vorkommen, welche Primärschlüssel einer anderen Tabelle sind, spricht man von Fremdschlüsseln.
* Fremdschlüssel dienen für die Assoziationen zwischen Tabellen
  + Referenzieren auf andere Tabellen
* Fremdschlüssel sind immer in der Tabelle mit n(\*)-Kardinalität zu finden.

Referentielle Integrität

* Besagt, Fremdschlüssel dürfen nur auf Attribut der anderen Tabelle referenzieren, wenn das andere Attribut vorhanden ist
* Dient zur Datenkonsistenz

# Datenbankentwurf

### Vom Konzeptionellen zum Logischen DB-Entwurf

### Grundlagen

Wie wir bereits wissen ist der konzeptionelle Datenbank Entwurf technologieunabhängig. Nun sollen wir aus dem konzeptionellen Modell den logischen DB-Entwurf erstellen. Es gibt einige grundlegende Dinge, welche zu beachten sind:

* Jede Klasse eine Tabelle
* Mindestens ein Primärschlüssel(oder Surrogatschlüssel)
* (Attribute unstrukturierte Werte in 1. NF)

##### Bildschirmausschnitt1:n Beziehungen

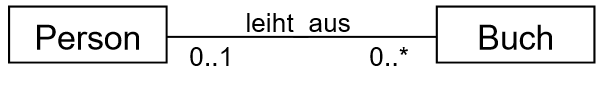
In der \*-Tabelle muss der Primärschlüssel von der 1-Tabelle als Fremdschlüssel vorkommen.

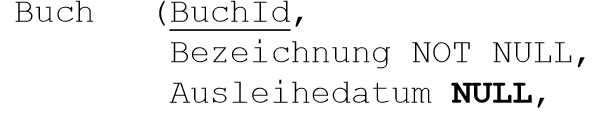
Eigenschafften(NULL, Not NULL) der Assoziation werden in der \*\_Tabelle beim Fremschlüssel mitgegeben.

BildschirmausschnittBildschirmausschnitt

##### Optionale Beziehungen

Eine Beziehung ist dann optional, wenn die die Referenz auch NULL sein darf:

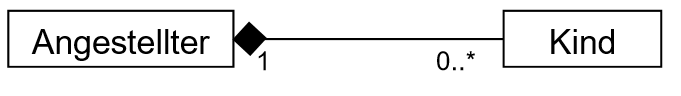


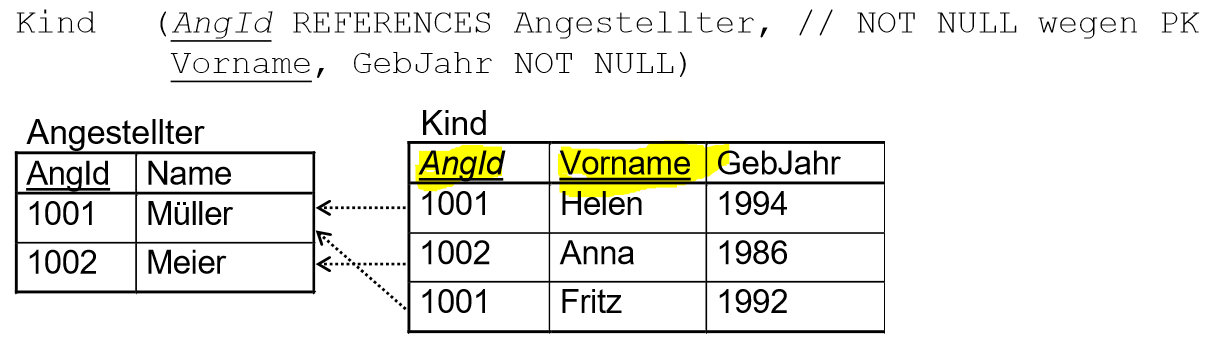
Bildschirmausschnitt

Eine Referenz darf dann NULL sein, wenn es eine 0…1-Tabelle gibt.(Bei einer 1-Tabelle ist Referentielle Integrität gewährleistet)

##### Aggregationen

Wie wir gelernt haben, heisst eine Aggregation, dass eine Tabelle abhängig ist von der anderen und daher nicht alleine überleben kann:

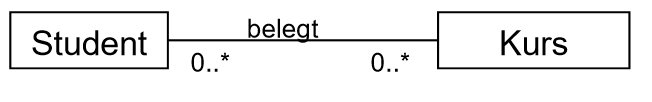


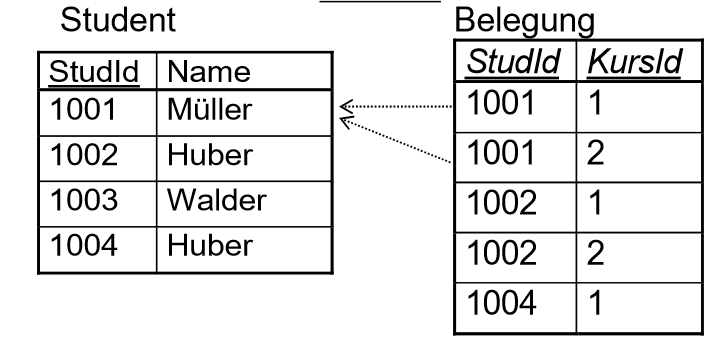


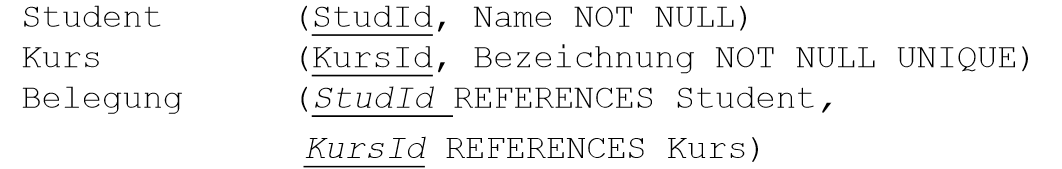
Eine Aggregation beinhaltet immer den Primärschlüssel der abhängigen Tabelle und kann nicht ohne diesen auskommen.(Sofern das Symbol ausgemalt ist, vergleiche dazu [die Komposition](#_Delegation_/_Komposition))

##### n:m Beziehungen

Die Lösung von einer n:m Beziehung ist ein Zwischen Tabelle:







Neu: Ein Student hat mehrere Belegungen und eine Belegung gehört genau zu einem Student. Ein Kurs kommt in mehreren Belegungen vor und eine Belegung beinhaltet genau ein Kurs.

1

0..\*

0..\*

1

Belegung

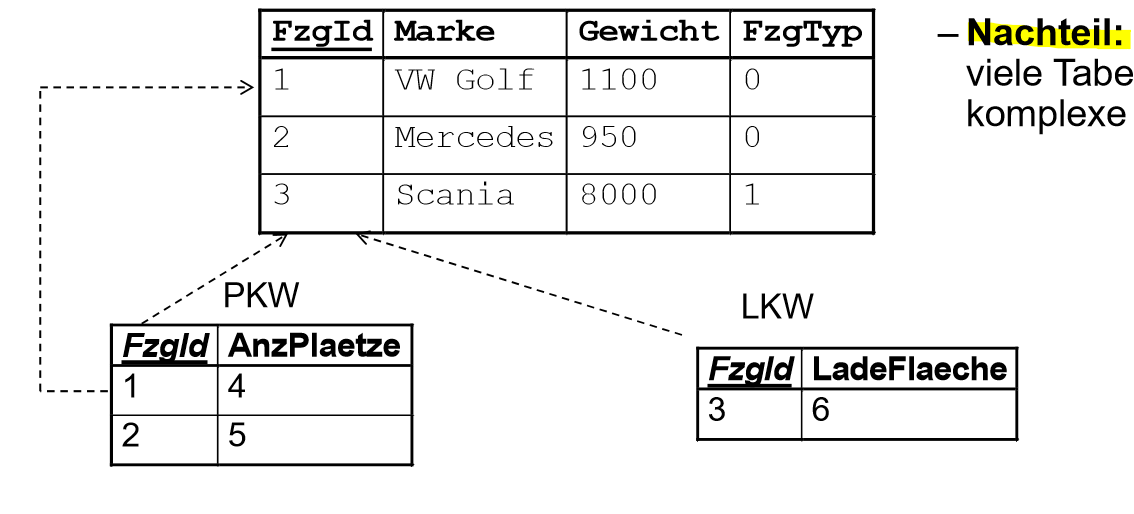
Kurs

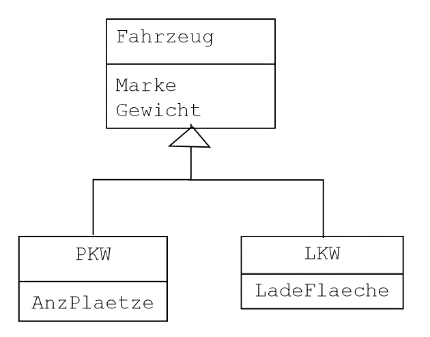
Student

#### Generalisierung

Bei der Generalisierung gibt es verschiedene Arten, wie man das realisieren kann(Beispiel disjoint, kann nur eine Subklasse haben):

##### Variante 1



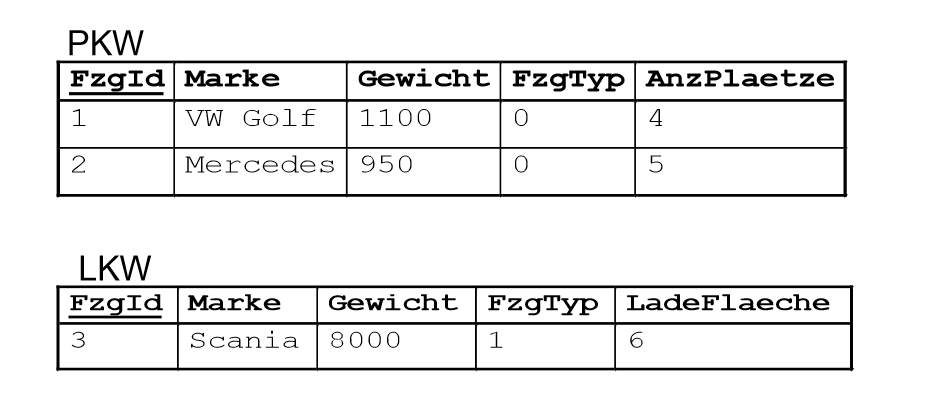


Es wird mit Unterklassen realisiert.

Vorteil: Redundanzfrei, auch bei overlapping

Nachteil: viele Tabellen, komplex

##### Variante 2

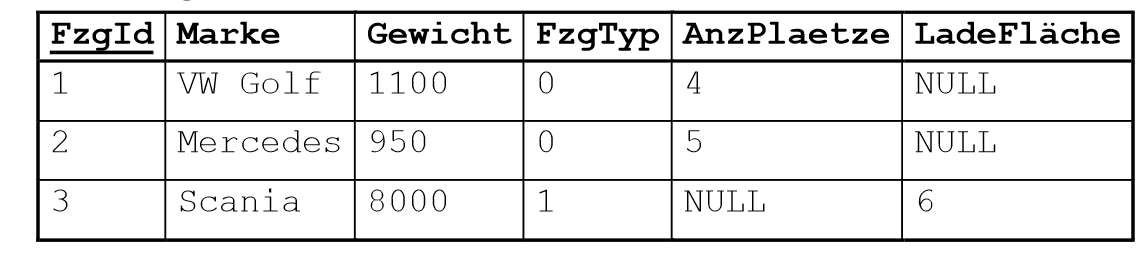
Keine Superklasse:

Vorteil: einfacher Zugriff

Nachteil: Übersicht über alle Fahrzeuge wird komplexer, nur bei disjoint

##### Variante 3

Nur eine Tabelle



Vorteil: Gute Übersicht und einfache Abfragen, auch bei Overlapping

Nachteil: Attributwerte können NULL sein, fehleranfälliger, 3. NF verletzt

#### Schritt 1 Abbildung

In einem ersten Schritt wird der grafische UML-Entwurf auf die Datenbank angepasst. Dies realisieren wir mit der relationalen Schreibweise:

practice( practiceID INTEGER,

day DATE NOT NULL,

*fk\_siteID* REFERENCES site*,*

*fk\_team* REFERENCES team

);

* Tabellenname
* Attributsname mit Datentypen und Bedingungen
* Primärschlüssel unterstrichen
* Fremdschlüssel *kusiv*
* REFERENCES Tabelle

##### Datentypen

* VARCHAR (Anzahl Character)
  + Nachname VARCHAR(20)
* INTEGER
  + Telefonnummer INTEGER
* DECIMAL(10,2)
  + Preis DECIMAL(3,2)
* DATE
  + Datum DATE
* TIME
  + Zeit TIME
* DATETIME
  + Datum DATETIME

##### Bedingungen

* NOT NULL
* UNIQUE

#### Schritt 2 Überprüfen der Normalformen

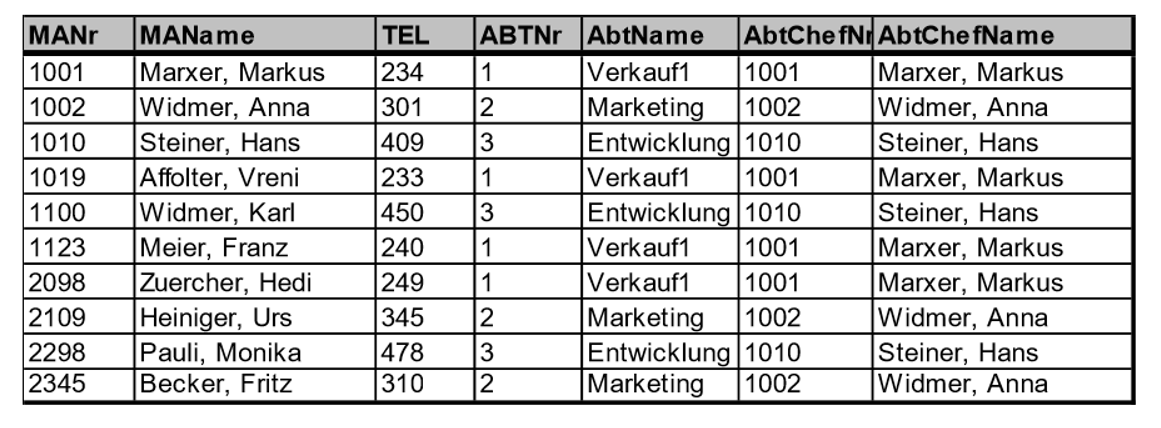
Mit Hilfe der Normalformen lässt sich prüfen, ob ein Datenmodell (praktisch) redundanzfrei ist. Die grösstmögliche Datenintegrität wird mit der 3. Normalform gewährleistet:

Nachdem im 1. Schritt das relationale Datenmodell entworfen wurde, wird im 2. Schritt das Modell auf Redundanzfreiheit geprüft. Diese Überprüfung findet mittels dem Anwenden der Normalformen statt:

* Keine Normalform 🡪 Mehrere Schlüsselkandidaten
* Normalformen(NF) 1, 2, 3 & Boyce-Codd(BCNF) analysieren Abhängigkeit von Nichtschlüsselattributen zu Schlüsselattributen
* Normalform führen zu Redundanzfreiheit und verhindern Anomalien

##### Keine Normalform

In einer Tabelle, welche keine Normalform ausweisst, sind redundante Daten vorhanden. Das bedeutet, gewisse Daten können ohne Informationsverlust weggelassen werden:



In dieser Tabelle sind die Angestellten mit deren Abteilung und deren Chefs aufgelistet. Jedoch sind die Informationen AbtName und AbtChefName überflüssig, da sich diese aus den jeweiligen Nummern herleiten können. Diese Informationen sind redundant und können weggelassen werden.

###### Mutations-Anomalien

Treten in einer Datenbank redundante Daten auf, kann es sogenannte Mutations-Anomalien gegben:

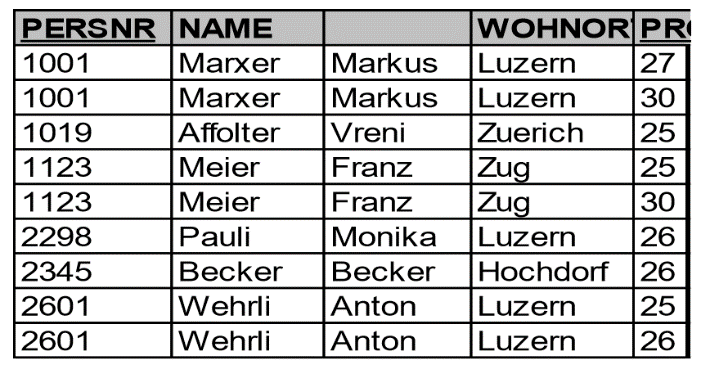
1. Einfügeanomalien
   1. Es braucht einen Abteilungs-MA und -Chef um eine neue Abteilung einzuführen
2. Löschanomalien
   1. Wenn alle MA einer Abteilung gelöscht sind, ist auch die Abteilung verschwunden
3. Änderungsanomalien
   1. Wird der Abteilungsname geändert, so müssen alle Datensätze der Mitarbeiter dieser Abteilung verändert werden

Man merkt sofort es würde ein riesiger Aufwand entstehen und die Datenintegrität wäre schnell nicht mehr garantiert.

##### 1. Normalform

Eine Tabelle ist in der ersten Normalform(1.N), falls die Wertebereiche der Attribute atomar(nicht weiter teilbar – nichts kleineres) sind.

Beispiele:

* Name wird aufgeteilt in Vor- und Nachname
* Adresse wird aufgeteilt in Strasse und Hausnummer PLZ und Wohnort
* Für jedes Projekt ein Datensatz



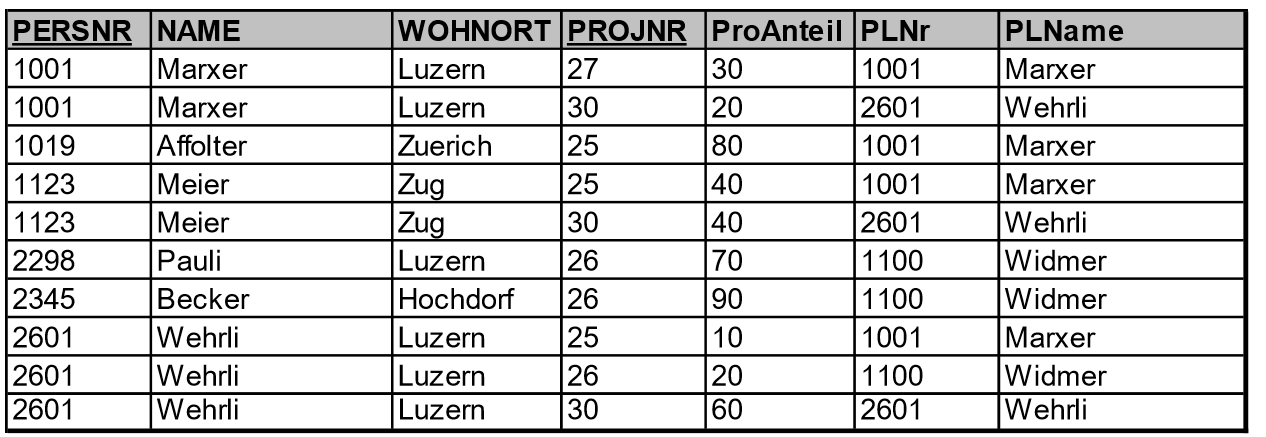
##### 2. Normalform

Eine Tabelle ist in der 2. Normalform, wenn sie in der ersten Normalform ist und wenn jedes Nichtschlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist.

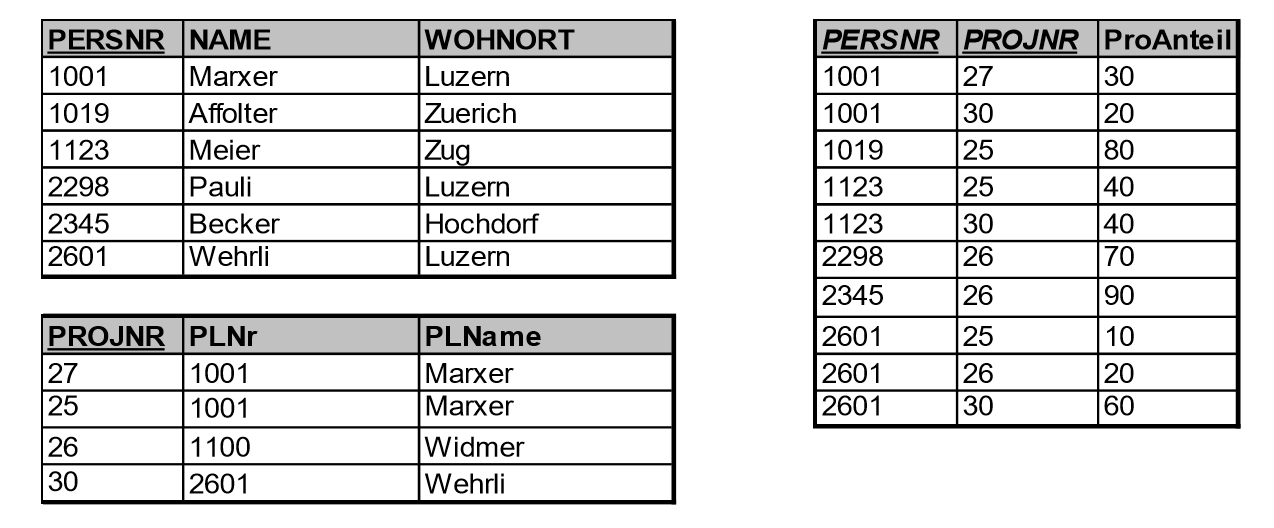
Voll funktional abhängig bedeutet, dass es zu jedem Schlüssel genau ein Wert gehört. Das heisst es gibt pro Schlüsselkandidat genau ein Datensatz.

Besteht der Schlüssel nur aus einem einzigen Attribut ist die Tabelle bereits in der 2. Normalform.

Hat die Tabelle einen zusammengesetzten Schlüssel, muss geschaut werden, dass jedes Nichtschlüsselattribut von beiden abhängig ist.



Diese Tabelle ist in der 1. Normalform, da alle Attribute atomar sind. Um diese Tabelle in die 2. Normalform zu kriegen muss man wie folgt vorgehen:

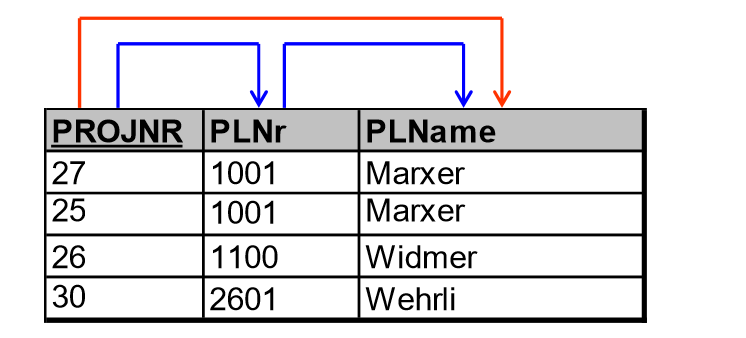
1. Welches sind die Schlüsselkandidaten?
   1. PersNr und ProjNr
2. Welche Attribute sind von PersNr abhängig?
   1. Name und Wohnort und ProAnteil
3. Welche Attribute sind von ProjNr abhängig?
   1. PLNr und PLName und ProAnteil
4. Man erstellt für jeden Schlüsselkandidat eine Tabelle
   1. Personen und Abteilungen
5. Da ProAnteil jedoch von beiden Schlüsselkandidaten abhängig ist, muss es eine eigene Tabelle geben aus einem zusammengesetzten Schlüsselkandidaten:

Mit dieser Lösung sind nun alle Attribute funktional abhängig von ihrem Schlüssel.

##### 3. Normalform

Eine Tabelle ist in der 3. Normalform, wenn sie in der 2. Normalform ist und kein Nichtschlüsselattribut von irgendeinem Schlüssel transitiv abhängig ist.

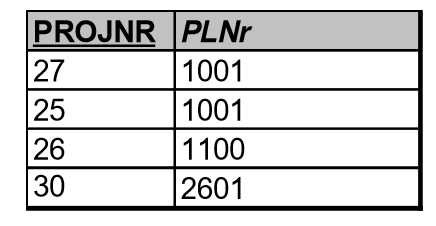
Transitiv abhängig heisst über Umwege abhängig:

Der Projektleiter ist über Umwege von der Projektnummer abhängig. Die PLNr reicht aus, um die ganzen Informationen wie Name, Geburtstag, etc. vom Projektleiter zu erfahren.

PLNr ist funktional abhängig von ProjNr

PLName ist funktional abhängig von PLNr

PLName ist somit transitiv abhängig von ProjNr



PLName muss in dieser Tabelle weggelassen werden und ist in der Tabelle Personen zu finden. PLNr referenziert auf die Tabelle Personen.

##### Boyce-Codd-Normalform(BCNF)

Wenn 3. NF erfüllt und jede Determinate Schlüsselkandidat ist.

##### Weitere Normalformen 4. & 5. Normalform

Wissen, dass es diese gibt.

Denormalisierung

Manchmal ist es nützlich eine Datenbank in die 2. NF zurückzunehmen oder gewollt in der 2. NF zu belassen. Grund dafür ist, dass es je nach Anwendungsfall in der 2. Normalform bessere Performance geben kann.

### Vom Logischen zum Physischen DB-Entwurf

#### Grundlagen

In diesem Kapitel geht es darum, wie man aus dem Logischen zum physischen DB-Entwurf gelangt. Da der logische Entwurf bereits technologieabhängig ist, ist dieser Schritt kleiner als vom konzeptuellen(UML – technologieunabhängig) zum logischen(Relationale Schreibweise).

Das Ziel des physischen Entwurfs ist ein physisches Model mittels SQL. Man erhält also SQL-Skripte, welche die Datenbank mit ihrem Einstellungen, die Tabellen mit Ihren Strukturen, die Benutzer mit ihren Zugriffsrechten, beinhaltet.

1. Datenbank und Rollen erstellen(DDL)
2. Struktur erstellen(DDL)
   1. Relationale Schreibweise zu SQL
3. Daten einpflegen(DML)
4. Referenzen erstellen(DML)
5. VIEWS, QUERIES erstellen(DDL & DML)
6. Berechtigungen und Transaktionen(DCL, TCL)

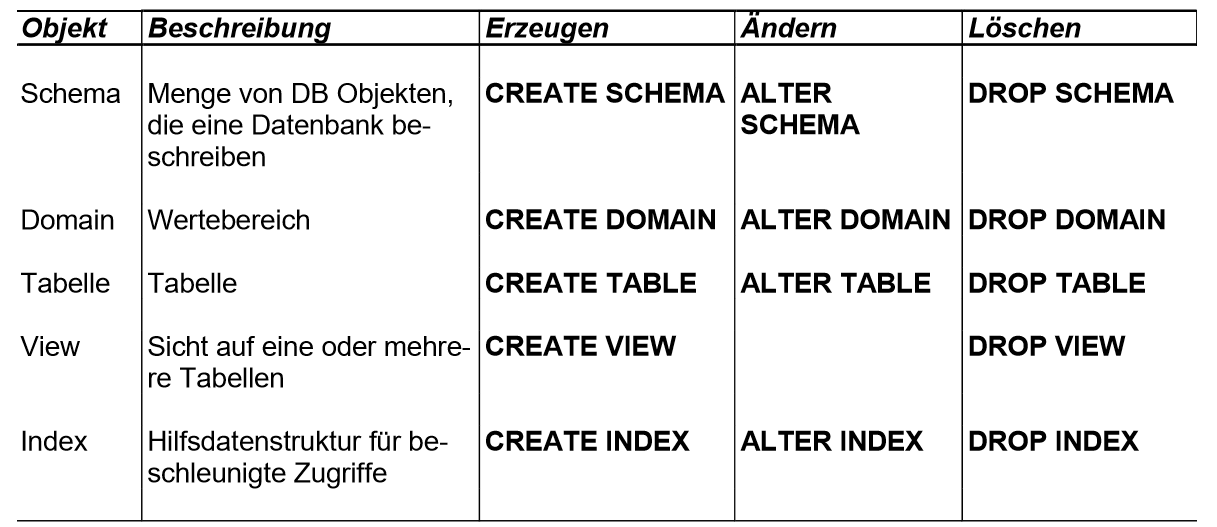
Mehr dazu im nächsten Kapitel

# Structures Query Language(SQL)

SQL bedeutet übrigens Structured Query Language und wird zum Definieren, Manipulieren Löschen und Abfragen von Daten/ Datenbankstrukturen verwendet.

Sie wird unterteilt in 3 Unterkategorien:

## Data Definition Language(DDL)

Wird verwendet um Datenbankobjekte zu definieren, zu ändern und zu löschen.(CREATE, ALTER; DROP)

### CREATE DATABASE

BildschirmausschnittIn einem ersten Schritt geht es um die Erstellung der Datenbank. Diese wird in SQL wie folgt kreiert:

Falls gewollt Besitzer angeben.

### CREATE TABLE / ALTER TABLE

**CREATE TABLE** angestellter(  
 persnr INTEGER PRIMARY KEY,  
 name VARCHAR(80) NOT NULL,  
 abteilung INTEGER REFERENCES abteilung

**);**

* Tabellenname
* Attributsname
* Datentypen/Wertebereich
* Primärschlüssel
* Fremdschlüssel
* REFERENCES Tabelle
* Bedingungen

#### Bedingungen

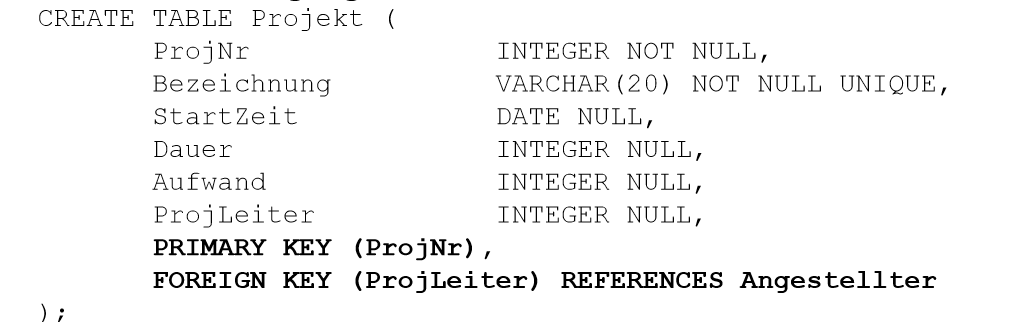
* PRIMARY KEY
* NOT NULL
* UNIQUE
* REFERENCES
* CHECK(ATTRIBUT BETWEEN 1 und 100)
* DEFAULT

##### Hinzufügen von Bedingungen

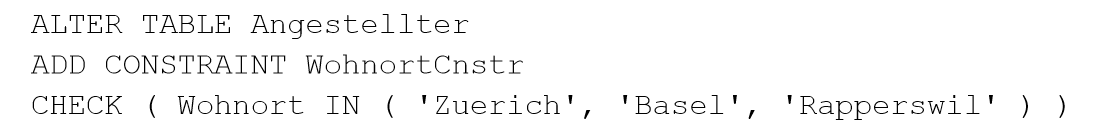
1. Column-Constraints

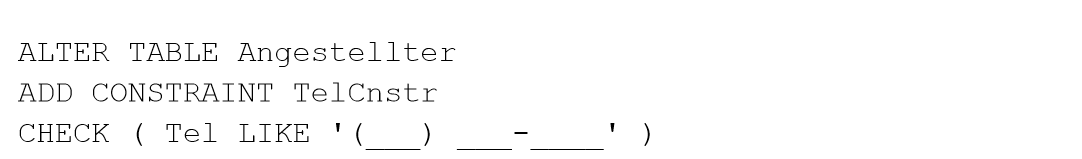


1. Table-Constraints



1. Im Nachhinein(ALTER TABLE)





Tipp: Die Fremdschlüssel nach den Inserts mittels ALTER TABLE hinzufügen, ansonsten könnte es zu Problemen führen.

##### Bedingungen von Fremdschlüssel(Referentielle Integrität)

Damit die Referentielle Integrität nicht verletzt wird, gibt es einige Bedingungen, welche beim Erstellen des Fremdschlüssels mitgegeben werden können:

* ON DELETE CASCADE
  + Wird ein Tupel X gelöscht, werden sämtliche Tupel Y mitgelöscht, welche auf den Tupel X referenziert haben.
* ON DELETE RESTRICT(DEFAULT)
  + Wird ein Tupel X gelöscht und es werden noch Referenzen auf den Tupel X festgestellt, wird die Löschoperation nicht ausgeführt / rückgängig gemacht
* ON DELETE SET NULL
  + Wird ein Tupel X gelöscht, werden bei sämtlichen Tupel Y der Fremdschlüsselwert auf NULL gesetzt, wenn diese auf Tupel X referenzierten
* ON DELETE SET DEFAULT
  + Wird ein Tupel X gelöscht, werden bei sämtlichen Tupel Y der Fremdschlüsselwert auf den DEFAULT gesetzt, wenn diese auf Tupel X referenzierten

##### Bildschirmausschnitt

##### Bedingungen von Primärschlüssel

* ON UPDATE CASCADE
  + Wird ein Primärschlüssel geändert, werden auch die darauf referenzierenden Fremdschlüssel geändert
* ON UPDATE RESTRICT
  + Wird ein Primärschlüssel geändert, auf den referenziert wird, wird die Änderung nicht durchgeführt
* ON UPDATE SET NULL
  + Wird ein Primärschlüssel geändert, werden die darauf referenzierenden Fremdschlüsselwerte auf NULL gesetzt
* ON UPDATE SET DEFAULT
  + Wird ein Primärschlüssel geändert, werden die darauf referenzierenden Fremdschlüsselwerte auf den DEFAULT-Wert gesetzt

### ALTER TABLE

ALTER TABLE distributors ADD COLUMN address varchar(30);

ALTER TABLE distributors DROP COLUMN address RESTRICT;

ALTER TABLE distributors  
 ALTER COLUMN address TYPE varchar(80),  
 ALTER COLUMN name TYPE varchar(100);

ALTER TABLE distributors RENAME COLUMN address TO city;

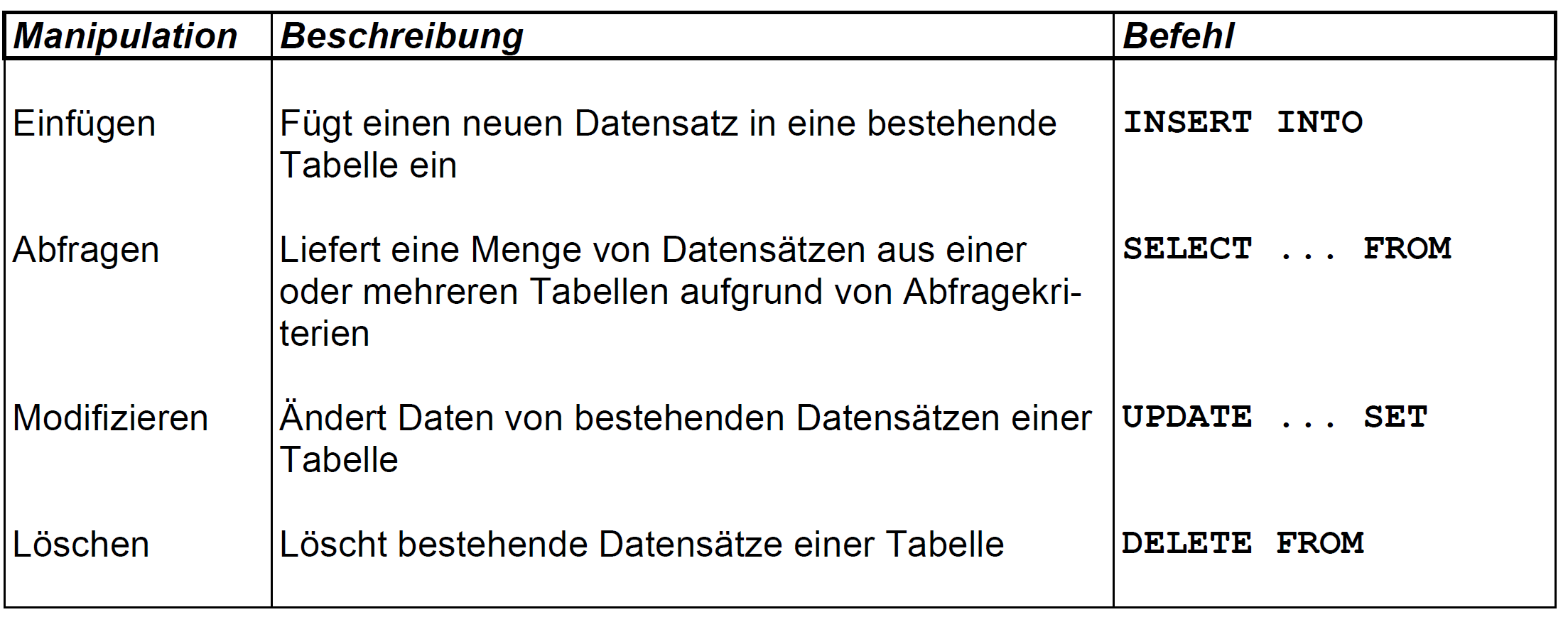
### DROP TABLE

DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]

CASCADE löscht andere Tabellen ebenfalls, welche darauf referenzieren.

RESTRICTED gibt es noch vorhandene Assoziationen, wird die Tabelle nicht gelöscht

## Data Manipulation Language(DML)

Wird verwendet um Daten zu anzuzeigen und zu modifizieren(Select, Insert, Update, Delete).

### Einfügen von Datensätzen(INSERT)

Das Einfügen von Datensätzen geschieht über den Insert Befehl:

**INSERT INTO** angestellter**(**PersNr, Name, AbtNr**) VALUES(1001, ‘Meier’, 1)**



* Tabellenname
* Attributsname
* Attributswerte

### Verändern von Datensätzen(UPDATE)

**UPDATE** angestellter **SET** abtnr **=** 2 **where** abtnr = 1**;**

* Tabellenname
* Attributsname
* Attributswerte(welcher gesetzt werden soll)
* Bedingung(welche erfüllt werden muss, dass neuer Wert gesetzt wird)

Wird die Bedingung vergessen, werden die Änderungen bei allen Datensätze der Tabelle gesetzt!

### Löschen von Datensätzen(DELETE / TRUNCATE)

#### DELETE FROM

**DELETE FROM** angestellter **where** abtnr = 1**;**

* Tabellenname
* Bedingung(die festlegt, welche Datensätze gelöscht werden sollen)

#### TRUNCATE

**TRUNCATE** angestellter (RESTRICT / CASCADE);

* Tabellenname

CASCADE(optional) löscht Datensätze, welche auf zu löschenden Datensatz referenzieren.

RESTRICTED gibt es noch vorhandene Assoziationen, wird der Datensatz nicht gelöscht

### Projektion/ Anzeigen von Datensätzen(Select)

#### Grundlagen

Mit dem Select Befehl können Datensätze ausgegeben werden. Hier gibt es sehr viele Möglichkeiten.

Merke immer die Zauberformel (SFWGO – SELECT, FROM, WHERE, GROUP BY, ORDER BY)

**SELECT** pid **as** Personalnummer, name **from** angestellter **where** name like '%a' OR name ilike '%e' order by pid desc limit 2;

* Tabellenname
* Attributsname(welche ausgegeben werden sollen, \* für alle Attribute)
* Spaltennamen(wie die Spalten heissen sollen, optional)
* Attributswerte(welcher gesetzt werden soll)
* Bedingung(welche erfüllt werden muss)
* Ordnung(nach was geordnet werden soll, standardmässig asc(ascending), hier desc(descending)
* Limitierung(limitiert Anzahl Datensätze, welche ausgegeben werden sollen)



#### Wichtige Bedingungen

Es gibt eine Anzahl von nützlichen Bedingungen:

##### BETWEEN - AND

SELECT name, lohn from angestellter **WHERE** lohn **BETWEEN** 1000 **AND** 10000**;**

##### IN

SELECT name, lohn from angestellter **WHERE** wohnort **IN (‘Luzern’, ’Zürich’. ‘Basel’;**

##### LIKE

SELECT name, lohn from angestellter **WHERE** wohnort **LIKE** ‘Z%’**;**

* % 🡪 beliebige und beliebig viele Character
* \_ 🡪 genau ein beliebiger Character
* (x|y) 🡪x oder y

##### NOT

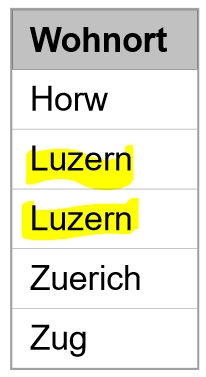
SELECT name, lohn from angestellter **WHERE** wohnort **NOT** **LIKE** ‘Z%’**;**

NOT kann vor jedem LIKE, IN, etc. stehen. Bedeutet ganz einfach nicht.

#### Select DISTINCT

Mit dem Select DISTINCT wird verursacht, dass keine Duplikate auftreten:

**SELECT DISTINCT** wohnort from angestellter;





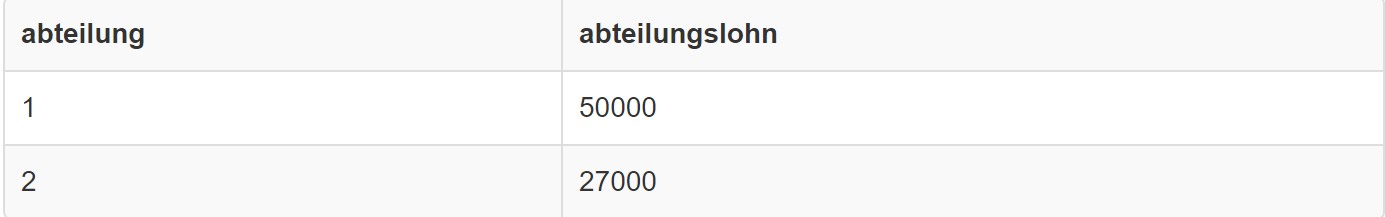
#### Group-Funktion

##### GROUP BY

Mit der Group-Funktion lassen sich Datensätze zusammenfassen:

SELECT abteilung, sum(lohn) as Abteilungslohn from angestellter **GROUP BY** abteilung order by abteilungslohn desc ;

Hier wird die Summe des Lohnes berechnet und dann nach der Abteilung gruppiert. Somit wird der Lohn pro Abteilung ausgegeben:



##### HAVING

Gruppierungsfunktionen können mit dem Attribut «Having» erweitert werden:

SELECT abteilung, avg(lohn)as Durchschnittslohn from angestellter group by abteilung **having** avg(lohn) >= 50000 order by avg(lohn) desc;



##### MIN

Wenn man eine Funktion anwendet, welche ein «group by» voraussetzt kann man dem aus dem Weg gehen, in dem man ein min(Attribut) verwendet:

SELECT **MIN(Wohnort)**, sum(lohn) as WohnortLohn from angestellterwhere wohnort = ‘Luzern’;

Dies ist oft nützlich, wenn man nach mehreren Sachen gruppieren will/muss.

#### Wichtige Funktionen

* Now() 🡪 aktuelles Datum
* SUM()
* COUNT()
* MIN()
* MAX()
* AVG()
* ISNULL()
* ROUND()
* MOD()

Diese wichtigen Funktionen können beim Abfragen von Daten sehr Hilfreich sein.

#### Joins

Um Datensätze anzuzeigen, welche über mehrere Tabellen verbunden sind, braucht man Joins.  
Ein einfaches Beispiel: Bis anhin konnte man immer nur auf die Tabelle angestellter zugreifen und somit nur die AbteilungsID anzeigen. Mittels Joins ist es möglich, statt der ID den Namen der Abteilung anzuzeigen.

SELECT DISTINCT a.abtid as Abteilungsid, ab.name from angestellter a **inner join abteilung ab on a.abtid = ab.abtid**;

* Abkürzungen für Tabellenname(einfach nach Tabellenname angeben)
* Join Bedingung
* Eindeutige Attributs Identifikation (Sobald ein Attribut mit gleichem Namen in zwei Tabellen vorkommt, muss der Tabellenname(oder die Abkürzung) vor dem Attribut stehen, sodass eindeutig ist, welches Atttribut gemeint ist)

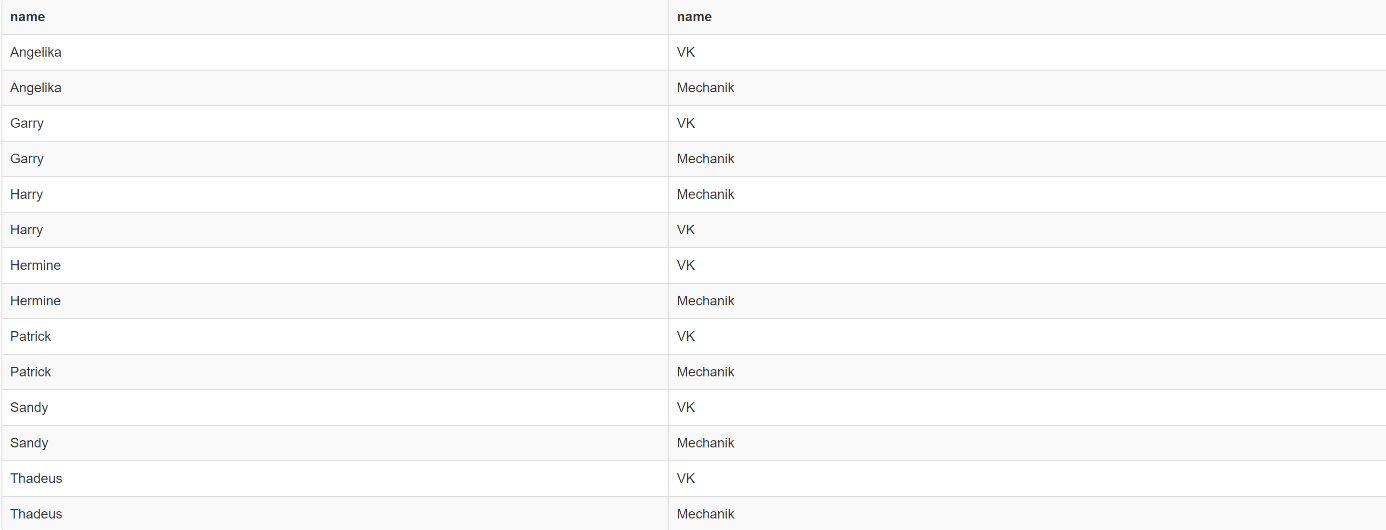
Es wird die Verbindung zwischen den Tabellen gemacht:  
Überall wo der Fremdschlüssel «abtid» der Tabelle Angestellter gleich ist wie der Primärschlüssel der Abteilungstabelle)

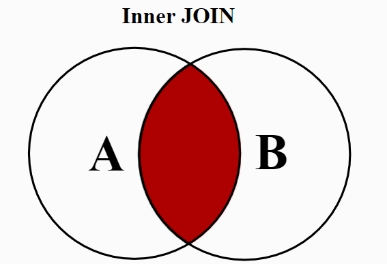


##### Cross Join(Kartesisches Produkt)

Kombination aller Tupel der Relation(Jeder Datensatz in Tabelle Angestellter wird mit jeder

**SELECT** a.name, ab.name from angestellter a **cross join** abteilung ab order by a.name;

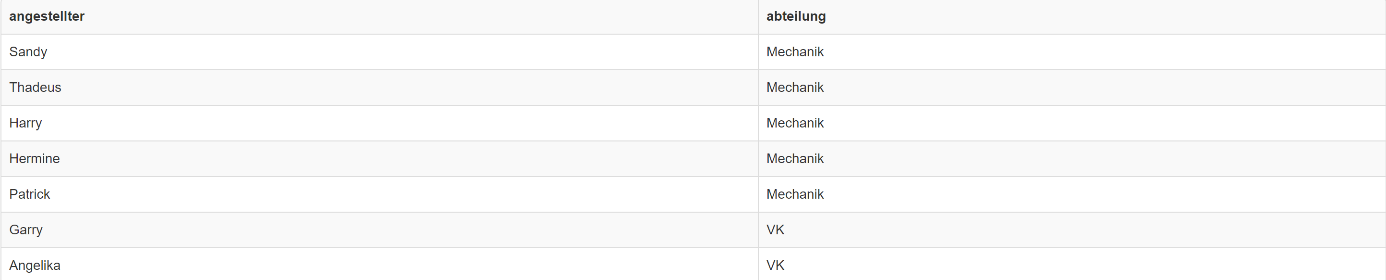


Wird nur selten verwendet. Jedes Element der Menge A wird mit jedem Element der Menge B gejoint.

##### Inner Join

Der wichtigste unter allen Joins. Mit diesem Join lassen sich Filterkriterien setzten wie beispielsweise: Gib mir nur die Zeilen, wo der Fremschlüssel abtid Tabelle A = Primärschlüssel Tabelle B:

SELECT a.name as Angestellter, ab.name as Abteilung from angestellter a **inner join abteilung ab on a.abtid = ab.abtid order by ab.name**;



Dieser Join ist extem mächtig, natürlich kann man auch über mehrere Tabellen «joinen» und Bedingungen, Sortierungen, Gruppierungen, etc. anwenden:

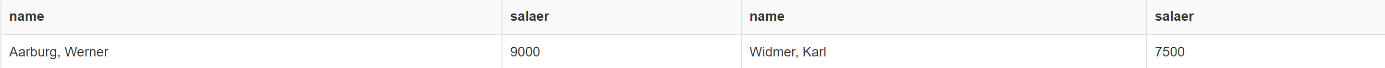
**SELECT** **DISTINCT** a1.name, ab.name **FROM** angestellter a1   
**inner join** projektzuteilung p **on** a1.persnr = p.persnr  
**inner join** projekt pr **on** p.projnr = pr.projnr  
**inner join** angestellter a2 **on** pr.projleiter = a2.persnr  
**inner join** abteilung ab **on** a1.abtnr = ab.abtnr  
**where** (a2.name like 'Marxer%');

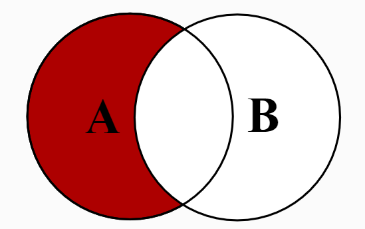
Hier werden alle Mitarbeiter ausgegenen, welche in mindestem einem Projekt mitarbeiten, in welchem der Chef «Marxer» heisst.

##### Self Join

Der Self Join ist ein spezieller Inner Join. Es wird eine Tabelle mit sich selber verknüpft:

**SELECT** a.name, a.salaer, b.name, b.salaer **FROM** angestellter a **inner join** angestellter b **on** a.chef = b.persnr where a.salaer > b.salaer;



Das a und b ist extrem wichtig, so werden die verschiedenen Kriterien auseinander gehalten. «a» repräsentiert hier die Mitarbeitet, «b» der Chef. Es werden also alle MA aufgelistet, welche einen höheren Lohn als ihr Chef haben.

##### Left (Outer) Join

Der Left Outer Join zeigt Datensätze der linken Tabelle an, auch wenn diese die Join Bedingungen nicht erfüllen:

**SELECT** p.bezeichnung, a.name **FROM** projekt p **LEFT OUTER JOIN** angestellter a on a.persnr = p.ProjLeiter;



Der Inner Join zeigt keine Projekte an, welche keinen Projektleiter haben:

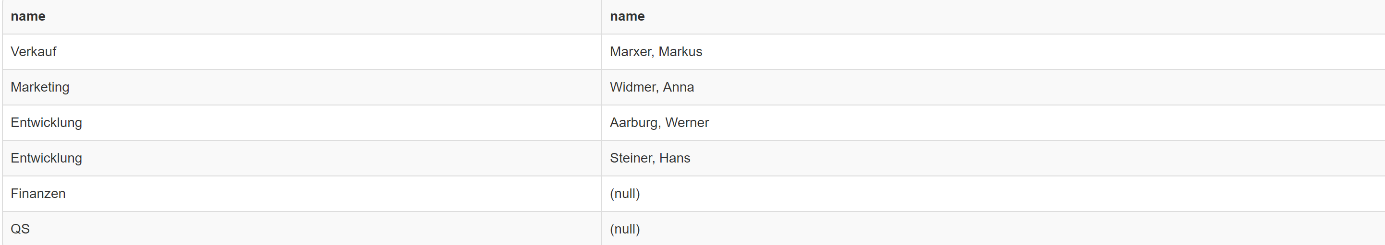


##### BildschirmausschnittRight (Outer) Join

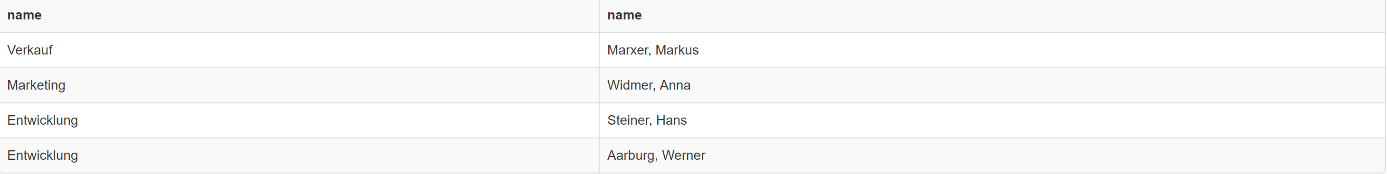
Der Left Outer Join zeigt Datensätze der rechten Tabelle an, auch wenn diese die Join Bedingungen nicht erfüllen:

**SELECT** ab.name, a.name **FROM** abtleitung  
inner join angestellter a  
on abtleitung.AbtChef = a.PersNr  
**right outer join** abteilung ab  
on ab.abtnr = a.abtnr;

Zeigt alle Abteilungen, auch wenn diese keinen Leiter haben:



Hingegen ein Inner Join zeigt nur die Abteilungen, bei welchen die Bedingung erfüllt ist, also ein Chef vorhanden ist:



##### Lateral Join

Mit dem Lateral Join ist es möglich, auf Attribute der Unterabfrage zu greifen:

**SELECT** ab.abtid, ab.name,eintritt.name, eintritt.eintrittsdatum **from** abteilung ab  
**cross join lateral(**  
Select \* from angestellter a  
where a.abtid = ab.abtid  
order by eintrittsdatum asc  
limit 2) **eintritt;**

In der Unterabfrage wird auf die Abteilung ab zugegriffen, was ohne den Lateral Join nicht möglich gewesen wäre. Ebenfalls wird in der Überabfrage auf «eintritt» zugegriffen.

#### Unterabfragen

##### Korrelierte / unkorrelierte

Eine Unterabfrage nennt man korreliert, wenn die Unterabfrage auf Spalten der äusseren Abfrage zugreift. Dies bedingt automatisch, dass die Unterabfrage nur funktionieren kann, wenn die äussere Abfrage vorhanden ist:

**SELECT** a.name, a.salaer **FROM** Angestellter a

WHERE a.salaer > **(select salaer from angestellter b**

**where a.chef = b.persnr)**;

Eine Unterabfrage ist dann unkorreliert, wenn die Abfrage nicht auf Spalten der äusseren Abfrage zugreift. Somit ist die unkorrelierte Abfrage auch selbständig ausführbar.

##### IN

Mit der IN Bedingung kann man prüfen, ob ein Wert = dem Wert in der Unterabfrage ist:

**SELECT** name **FROM** angestellter a **WHERE persnr** **IN** ( select persnr from projektzuteilung);

Es werden alle Mitarbeiter angezeigt, welche einem Projekt zugewiesen sind. Es wird jede PersNr der Tabelle angestellter genommen und geschaut, ob diese in der tabelle Projektzuteilung vorhanden ist, wenn ja wird der Name der Person ausgegeben.

##### EXISTS

Beim EXISTS wird geschaut, ob ein Attribut in den Datensätzen der Unterabfrage vorkommt. Der Unterschied ist, dass hier das Attribut gesucht wird, welches schlussendlich auch ausgegeben wird:

**SELECT** name **FROM** angestellter a **WHERE EXISTS** ( select \* from projektzuteilung p where a.persnr = p.persnr );

Es wird geschaut, ob der Name eines Angestellten in der Menge der Unterabfrage vorkommt. Ist dies der Fall, wird es ausgegeben.

Wichtig: Sobald ein Wert in der Menge vorhanden ist, wird true zurückgegeben. Es kann hier zum beispiel zu Fehler kommen wenn man «Franz» als Nachname und Vorname hat.(Hier jetzt nicht der Fall).

##### ANY

Es wird getestet, ob die Bedingung bei irgendeinem Datensatz der Unterabfrage korrekt ist:

**SELECT** a.name, a.salaer, ab.name **FROM** angestellter a  
inner join abteilung ab on a.abtnr = ab.abtnr   
**where** ab.name ILIKE 'verkauf' AND salaer <= **ANY** (   
select salaer from angestellter a1 inner join abteilung ab1 on a1.abtnr = ab1.abtnr where ab1.name = 'Marketing')  
order by salaer asc;

Es wird geprüft, ob ein Arbeiter des Verkaufs einen kleineren Lohn hat, als irgendein Mitarbeiter der Marketing Abteilung. Sobald ein Mitarbeiter der Marketing Abteilung einen grösseren Lohn hat, wird der Datensatz ausgegeben.

##### ALL

Es wird getestet, ob die Bedingung bei allen Datensätzen der Unterabfrage korrekt ist:

**SELECT** a.name, a.salaer, ab.name **FROM** angestellter a  
inner join abteilung ab on a.abtnr = ab.abtnr   
**where** ab.name ILIKE 'verkauf' AND salaer <= **ALL** (   
select salaer from angestellter a1 inner join abteilung ab1 on a1.abtnr = ab1.abtnr where ab1.name = 'Marketing')  
order by salaer asc;

Ein Datensatz wird ausgegeben, wenn ein Mitarbeitet des Verkaufs einen kleineren Lohn hat als alle Mitarbeiter der Marketing Abteilung.

#### Mengenoperationen

##### UNION

UNION verbindet zwei Mengen ohne Duplikate:  
UNION ALL verbndet zwei Mengen und gibt ebefalls Duplikate aus:

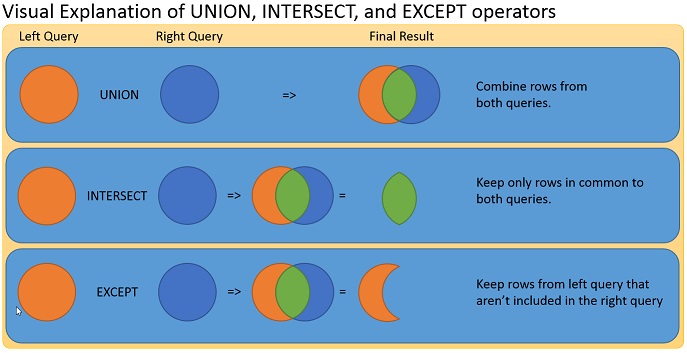
##### Bildergebnis für postgres union**SELECT** name, salaer **FROM** angestellter where name like 'M%'

union

select bezeichnung, dauer from projekt;

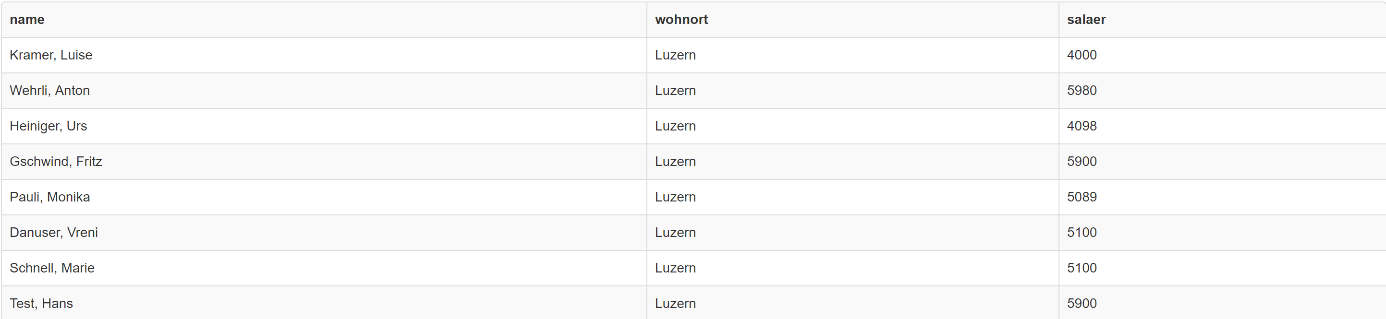
##### BildschirmausschnittVerbindet alle Mitarbeiter, bei welchen der Name mit M beginnt und die Projekte.

##### EXCEPT

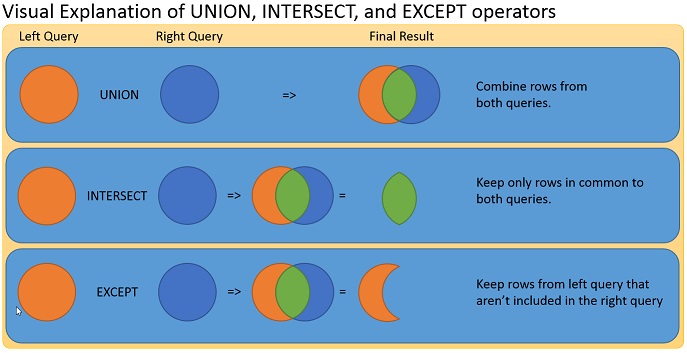
**SELECT** name, wohnort, salaer **FROM** angestellter where wohnort like 'L%'

**EXCEPT**

SELECT name, wohnort, salaer FROM angestellter where salaer > 8000;

Gibt alle MA aus, welche aus einem Dorf kommen, welches mit «L» beginnt und der Salaer nicht grösser als 8000 ist:

##### INTERSECT



**SELECT** name, wohnort, salaer **FROM** angestellter where wohnort like 'L%'

**INTERSECT**

SELECT name, wohnort, salaer FROM angestellter where salaer > 8000;

Gibt nur Mitarbeiter aus, welche in einem Dorf wohnen, welches mit «L» beginnt und einen Lohn grösser 8000 haben:

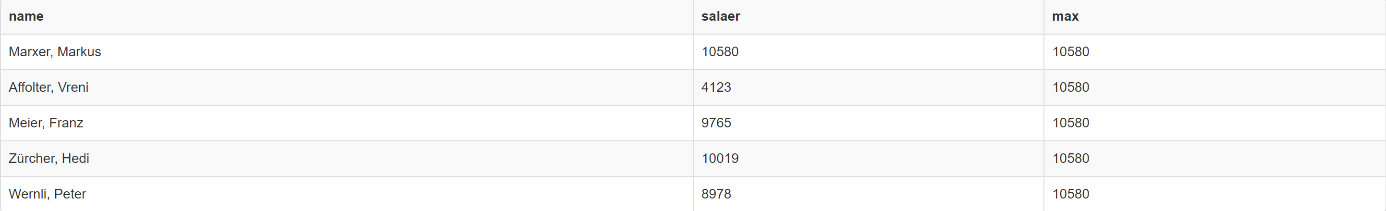


#### Windows Functions

##### Min(), Max(), Avg(), Sum() over

IN einer Spalte wird die Summe, Durchschnitt, etc. ausgerechnet. Und bei jedem Datensatz angezeigt.

**SELECT** name, salaer, max(salaer) **over (partition by abtnr)** **FROM** angestellter where abtnr = 1;

Es wird jeder Mitarbeiter der 1. Abteilung ausgegeben mit Name, Salaer und zusätzlich dem höchsten Salaer der Abteilung(Ohne abtnr=1 jede Abteilung aufgelistet):

Bei diesen funktion darf kein «order by» stehen!

##### Rank() over

Mit der Rank funktion kann eine Rankierung gemacht werden:

**SELECT** select name, salaer, **rank() over** (partition by abtnr order by salaer desc) from angestellter where abtnr = 1;

Hier wird die Option « order » zwingend gebraucht !

#### Common Table Expressions(CTE) / WITH Quries

CTE oder With-Queries sind temporäre Tabellen, welche nur während einem Statement existieren:

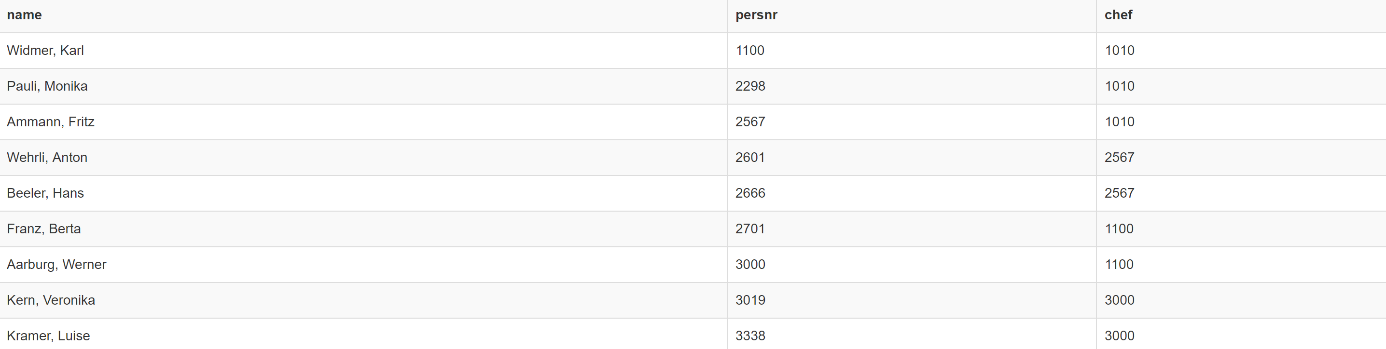
**WITH kleineProjekte AS(**select \* from projekt where dauer between 0 and 200**)** select a.name, kP.bezeichnung from angestellter a inner join **kleineProjekte** kP on kP.projleiter = a.persnr;

##### BildschirmausschnittEs werden der Projektleiter und ihr Projekt ausgegeben, sofern das Projek nicht mehr als 200 Stunden Dauer hat.

##### With Recursive

**WITH RECURSIVE untergebene AS** **(** select name, persnr, chef from angestellter where chef = 1010  
union   
select a.name, a.persnr, a.chef from angestellter a inner join untergebene u on u.persnr = a.chef)  
**SELECT** name, persnr, chef **FROM** untergebene order by 3, 1;

Es werden alle Untergebenen von dem Chef mit Persnr 1010 rekursiv berechnet und ausgegeben. Das bedeutet alle Personen X welche 1010 als Chef haben und alle Personen welche dann wiederrum diese die Untergebenen von den Personen X sind:



Man kann beobachten, die Person mit der Persnr 1010 hat 3 Untergebene, welche wiedr einige Untergebene haben.

#### BildschirmausschnittViews(Sichten)

##### Grundlagen

Views sind Sichten, welche einen Benutzer oder einer Applikation zur Verfügung gestellt werden. Der Anwender muss somit keine tieferen Informationen über die Datenbank(struktur) haben.  
Somit eignen sich Sichten für vielerlei Aspekte:

* Datenkapslung / Datenunabhängigkeit
  + Werden die Strukturen der Datenbank geändert, können die Views in relativ kurzer Zeit angepasst werden. Somit ist es kein grosser Aufwand
* Einfachheit(Vereinfachung von Queries)
  + Mit Views können Abfragen in der Datenbank gespeichert werden
    - Einfache Abfragen dem «Dummy» zur Verfügung stellen
    - Komplizierte Abfragen für DBA-Spezialisten «als Vorlage»
* Sicherheit
  + Sicher erlauben es nur bestimmte Daten anzuzeigen, somit werden irrelevante oder sensitive Daten für gewisse Benutzer verborgen

Eine View ist eine virtuelle Tabelle basierend auf anderen Tabellen oder Views. Views sind gespeicherte Select-Abfragen

##### Erstellen einer View

**CREATE VIEW SalaerProAbteilung AS** SELECT ab.name, sum(salaer)from angestellter a inner join abteilung ab on a.abtnr = ab.abtnr group by ab.abtnr**;**

* View Name
* Select Bedingung

##### Benutzen einer View

**SELECT \* FROM SalaerProAbteilung;**

##### Löschen einer View

**DROP VIEW SalaerProAbteilung;**

*Updatable Views*Änderungs-Operationen(Update, Insert, Delete) auf Views sind in der Regel heikel und nicht empfohlen(jedoch im Postrgres möglich).  
Beispielsweise, man hat eine View bei welcher man die Namen aller MA ausgibt. Wenn nun 2 MA den gleichen Namen haben, kann ein Update auf diese View nicht funktionieren.

Grundsätzlich sind Views updatable, wenn die Abfrage nur über eine Tabelle geht und kein Select Distinct beinhaltet

### Data Control Language(DCL)

### Grundlagen

Wir verwendet um Benutzer und Gruppen zu erstellen und modifizieren und Zugriffsrechte zu erstellen und zu verwalten:

* Eine Rolle gilt für den ganzen Cluster(alle DBs)

### Role erstellen

Eine Rolle kann ein Bentzer oder eine Gruppe sein. Ohne «with login password ‘password’» kein Login möglich.

**CREATE ROLE username WITH LOGIN PASSWORD ‘password’;**

* Benutzername
* Passwort

### Rechtevergabe

Die Rechte können mit dem Befehl «GRANT» vergeben werden:

**GRANT <PERMISSION> on <OBJECT> TO <USER>;**

**GRANT SELECT on testdb to testuser;**

**GRANT SELECT, INSERT on table testtable to testuser;**

#### Objektprivilegien

* SELECT
* INSERT
* UPDATE
* DELETE
* REFERENCES 🡪 Definieren von Fremdschlüsseln gestattet
* TRIGGER 🡪 STORED PROCEDRUES aufrufen
* ALL [PRIVILEGES]

#### Systemprivilegien

* CREATEROLE
  + Darf Role und DB erstellen
* CREATEDB
  + Darf DB erstellen

Systemprivilegien hinzufügen:

* **ALTER ROLE username WITH CREATEROLE;**

Oder beim Erstellen:

**CREATE ROLE username WITH LOGIN PASSWORD ‘password’ WITH CREATEDB;**

### Rechteenzug

**REVOKE <PERMISSION> on <OBJECT> TO <USER>;**

**REVOKE SELECT on testdb to testuser;**

**REVOKE ALL on table testtable to testuser;**

### Role entfernen

**DROP ROLE username;**

### Ändern des Passwortes

**ALTER ROLE username WITH PASSWORD ‘password’;**

### Gruppe erstellen

**CREATE ROLE groupname; --Role ohne Login**

### Mitglieder Gruppe hinzufügen

**GRANT groupname TO username;**

User erhält Rechte der Gruppe

## Transaktionen / Isolation

### Grundsätzliches

In der Datenbanken gibt es eine nicht überallanerkannte weitere Language, die Transaction Control Language, kurz TCL. Mit Hilfe von Transaktionen ist es gewährleistet die Datenintegrität zu optimieren.  
Wird keine Transaktion gestartet, wird jedes Statement als eine Transaktion gewertet(implizite Transaktion).

* Pro Session max. eine Transaktion
* Nach Commit/Rollback werden belegte Ressourcen freigegeben
* Transaktion in Transaktion nicht unterstützt

### ACID

* Atomicity(Atomarität)
  + Operationen werden vollständig oder gar nicht ausgeführt
* Consistency(Konsistenz)
  + Daten sind vor wie nach einer Transaktion konsistent.
* Isolation
  + Eine Transaktion soll so ausgeführt werden, als sei sie von anderen insoliert
* Durability(Dauerhaftigkeit)
  + Änderungen sind nicht flüchtig(persistent) und gehen durch Fehler nicht verloren

### Nutzen von Transaktionen

Mittels den Transaktionen, welche die ACID Kriterien beinhalten soll folgendes erreicht werden:

* Performance
  + Mehrere Transaktionen/Sessions gleichzeitig(Concurrency)
  + Jedoch alle Transaktion isoliert voneinander(Isolation)
* Fehlerertoleranz
  + Fehlgeschlagene Änderungen werden zurückgespielt und wiederholt(Atomicity)
  + Änderungen befinden sich immer auf persistentem Speicher(Durability)
    - Write-Agead-Log

Somit gehen nie Daten verloren oder es gibt nie inkonsistente Daten.

### Transaktionen – Befehle

#### Transaktion beginnen

**BEGIN; --Beginnt Transaktion**

#### Transaktion abschliessen(Änderungen «veröffentlichen»)

**COMMIT; --Beendet Transaktion**

#### Änderungen rückgängig machen

**ROLLBACK; --Spring zum Commit zurück**

#### Mögliche Resultate

Success 🡪Commit

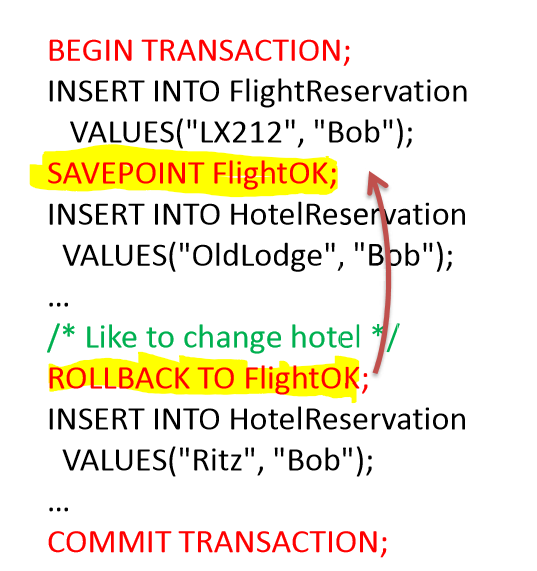
Failure 🡪Rollback

#### Gründe für Abbruch

* Expliziter Abbruch
* Unzulässige Verzahnung(Dead Locks)
* Applikationsabsturz, Systemabsturz, sonstige Fehler(Hardware, etc)

#### Savepoints

Savepoints kann man ansehen als Untertransaktionen. Sie werden nach dem BEGIN gesetzt. Man kann dann in der Transaktion auf die SAVEPOINTS zurückspringen. Dass hat den grossen Vorteil, dass man nicht die ganze Transaktion, sondern nur ein Teil wiederholen muss.



**BEGIN; --Beginnt Transaktion**

**--SQL Statement 1**

**SAVEPOINT SavepointName1**

**--SQL Statement 2**

**ROLLBACK SavepointName1**

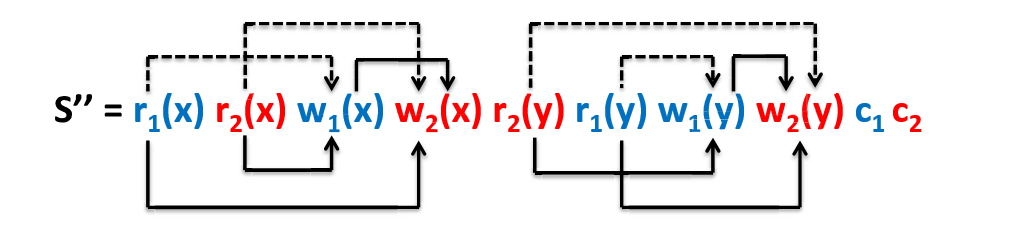
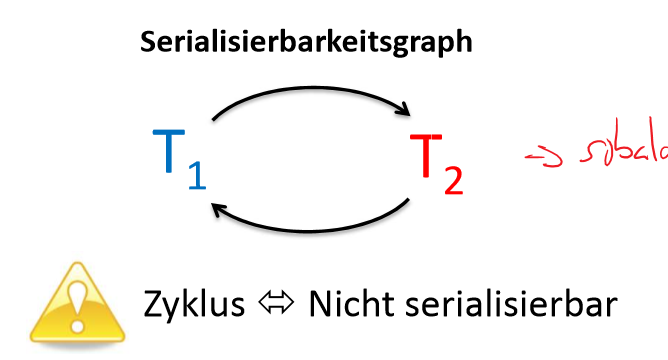
**COMMIT; --Beendet Transaktion**

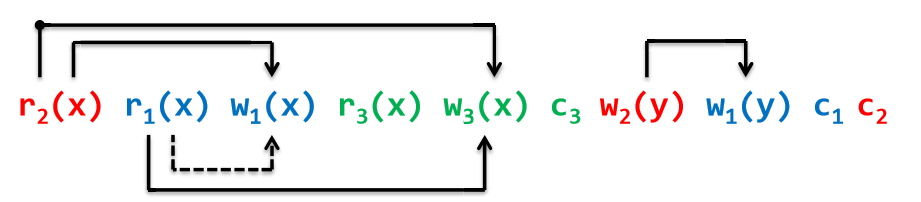
### Serialisierbarkeit

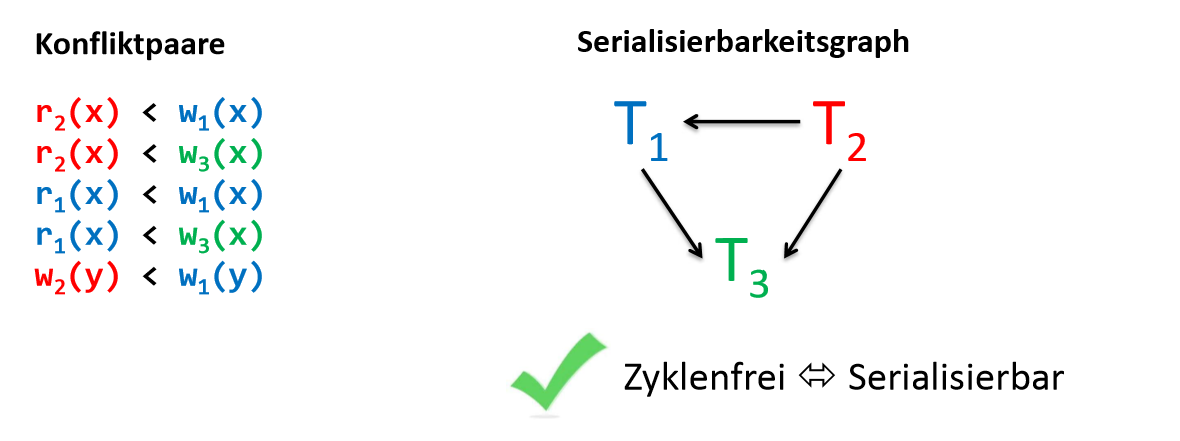
* Jede serielle Ausführung der Transaktionen ist korrekt
* Verzahnte(nebenläufige) Transaktionen auch korrekt, wenn gleiches Resultat wie serielle Ausführung 🡪 Serialisierbarkeit

Verzahnte Transaktionen sind serialisierbar, wenn kein Zyklus(Kreislauf) zwischen den einzelnen Transaktionen auftritt:

*Konfliktpaare*

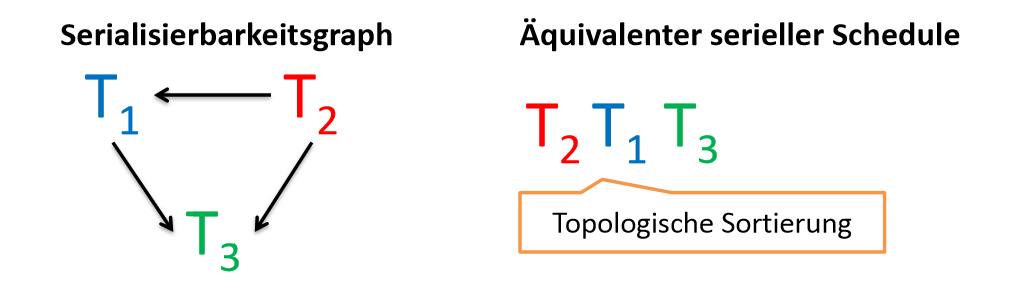
Mit den Konfliktpaaren(read-write und write-write Kombinationen) kann festgestellt werden, ob ein Schedule(mehrere verzahnte Transaktionen) serialisierbar sind:

Sobald sich ein Zyklus bildet, sind die Transaktionen nicht serialisierbar und somit fehlerhaft. Hierbei ist wichtig: Sobald zwei oder mehrere Transaktionen einen Zyklus bilden, ist der Schedule nicht serialisierbar. Es müssen also nicht alle Transaktionen des Schedules im Kreislauf eingebunden sein, damit es fehlschlägt.



*Commit Reihenfolge*

Die Commit Reihenfolge ist ebenfalls von den Konfliktpaaren und kann am Serialisierbarkeitsgraph abgelesen werden:



T2 bekommt keine «Anfragen» darum kann diese zuerst «commiten» dann T1 und dann T3, da T3 erst die Daten freigeben kann, wenn Sie T1 bereits freigegeben hat.

*Serialisierbarkeitsgraph*

* Theoretisches Mittel um serialisierbarkeit von Schedule zu ermitteln
* Wird in Praxis(von DBMS) nicht verwendet, in Praxis Synchronisationsprotokolle

### Umsetzung der Isolation

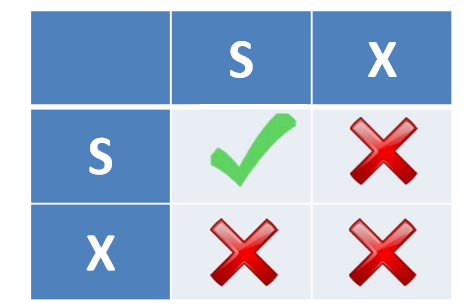
#### Isolationsverfahren

* Pessimistisches Verfahren(Locking)
  + Transaktion sperrt benutze Daten
  + Bei hoher Konfliktwahrscheinlichkeit gut
* Optimistisches Verfahren
  + Konfliktbehebung im Nachhinein
    - Vermeidet Konflikte nicht, sondern erkennt und behebt sie nur
  + Bei tiefer Konfliktwahrscheinlichkeit gut
  + Snapshot Isolation(SI) oder Multi-Version-Concurrency Control(MVCC)

#### Locking

Die Grundidee vom Locking ist, dass gebrauchte DB-Objekte gesperrt werden, so soll verhindert werden, dass Dateninkonsistenz auftritt.

Man unterscheidet zwischen zwei verschiedenen Modi:

* Exklusive Lock(X)
  + Sperren bei Lese- oder Schreibzugriffen
  + Nur eine Transaktion kann (xlock) auf Objekt haben
* Shared Lock(S)
  + Nur für Lese-Zugriffe
  + Mehrere Tranaktionen könne (slock) auf Objekt haben

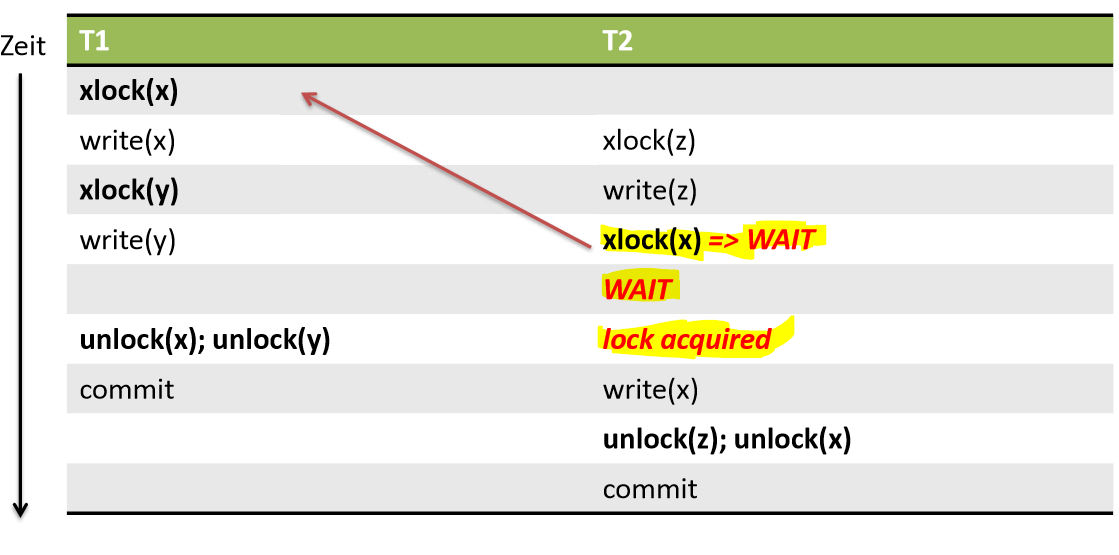
##### Granularität

Die Locks könenn auf folgende Objekte gelten

* Table
* Table Ranges
* Row
* Item

Je kleiner das DB-Objekt, desto mehr Transaktionen können parallel Arbeiten. Postgres auf Row-Ebene.

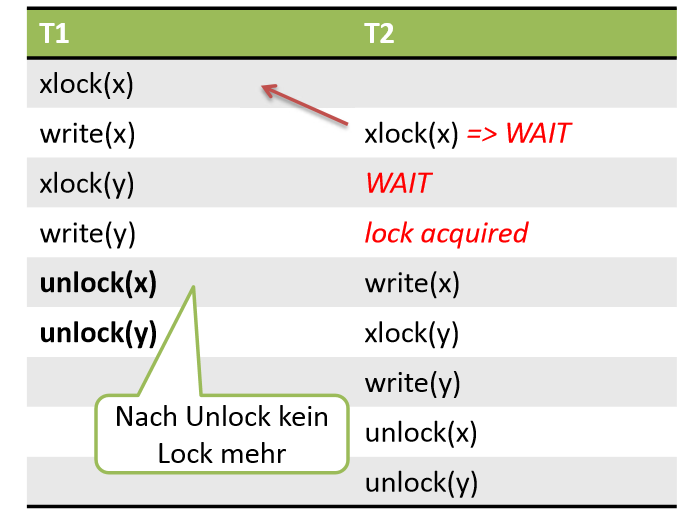
##### Beispiel



Es muss gewartet werden, bis das DB-Objekt x freigegeben wird.

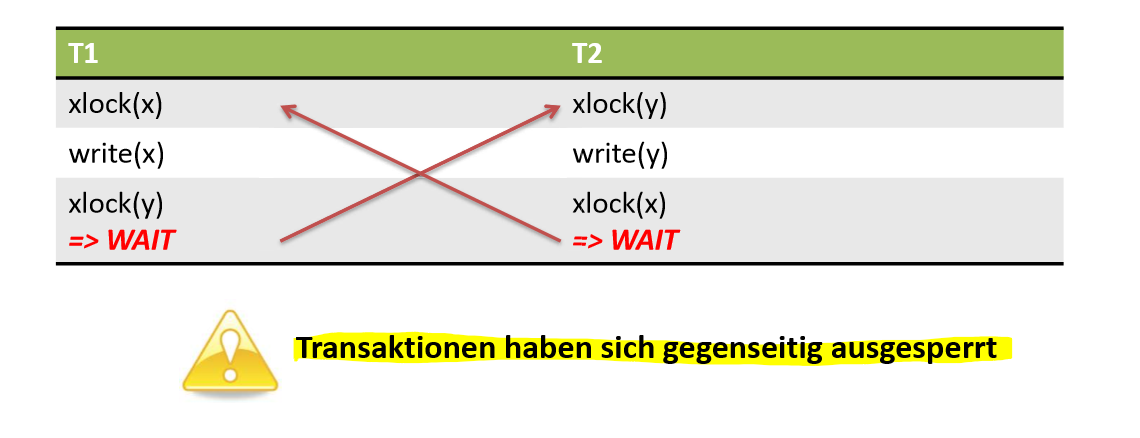
Wichtig: Locking garantiert noch keine Serialisierbarkeit!

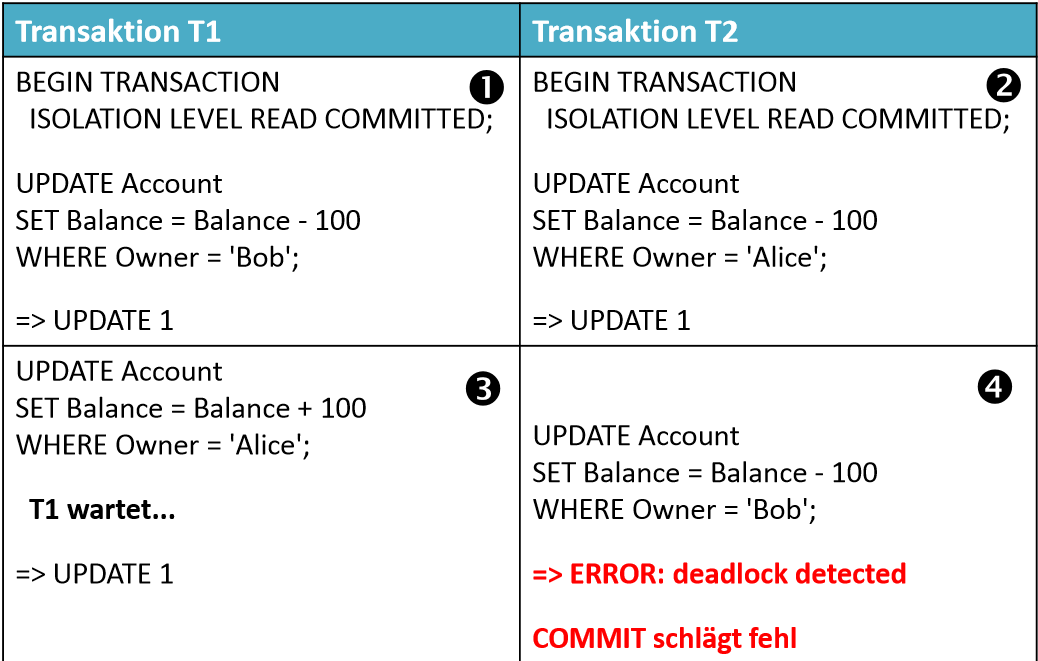
##### Two-Phase Locking(2PL)

* Phase 1
  + Growing Phase 🡪Jedes Objekt vor Zugriff gesperrt
* Phase 2
  + Shrinking Phase 🡪 Sobald Transaktion einen Lock freigegeben, kein weiterer mehr

Wichtig: 2PL garantiert Serialisierbarkeit, jedoch Deadlocks und Cascading Rollbacks möglich.

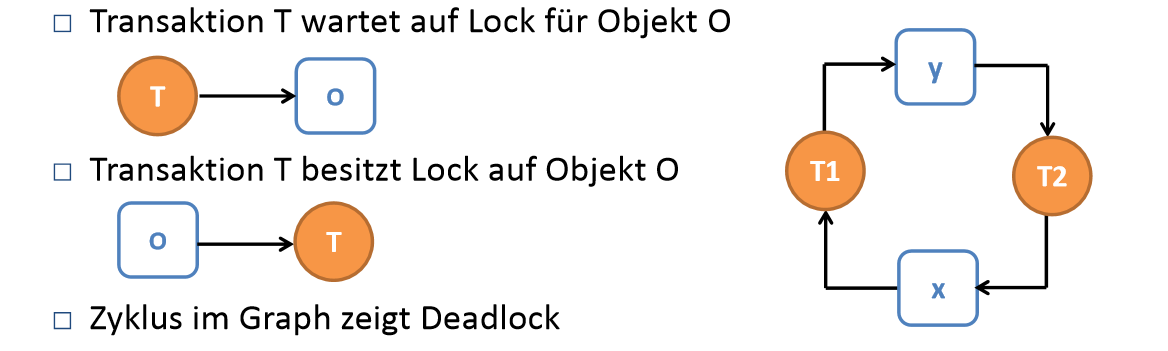
##### Deadlocks

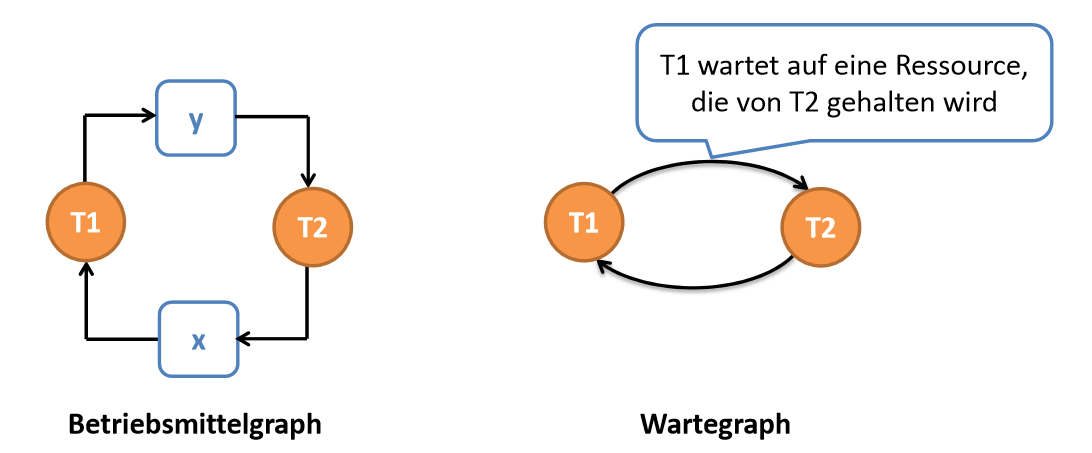
Deadlocks entstehen, wenn zwei Transaktionen sich gegenseitig ausgesperrt haben. Konkret bedeutet das, dass beide Transaktionen ein Objekt gesperrt haben, welches die andere Transaktion braucht. Somit warten beide Transaktion auf die Freigabe der jeweiligen Objekte und laufen in einen Timeout:

Spezialfall: Es kann auch sein, dass ein Deadlock mit slocks auftreten.

Deadlock Erkennung / Betriebsmittelgraph

Die Deadlock Erkennung erfolgt über den Betriebsmittelgraph, aus welchem der Wartegrapg resultiert. Hier gilt wieder, wenn der Wartegraph einen Zyklus beinhaltet, gibt es einen Deadlock.



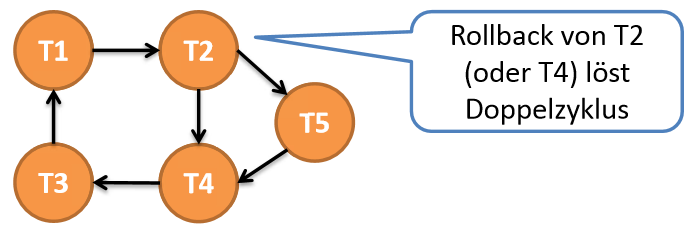


##### Deadlock Behebung

* Opfer bestimmen
* Timeout definieren

###### Variante 1 Opfersuche

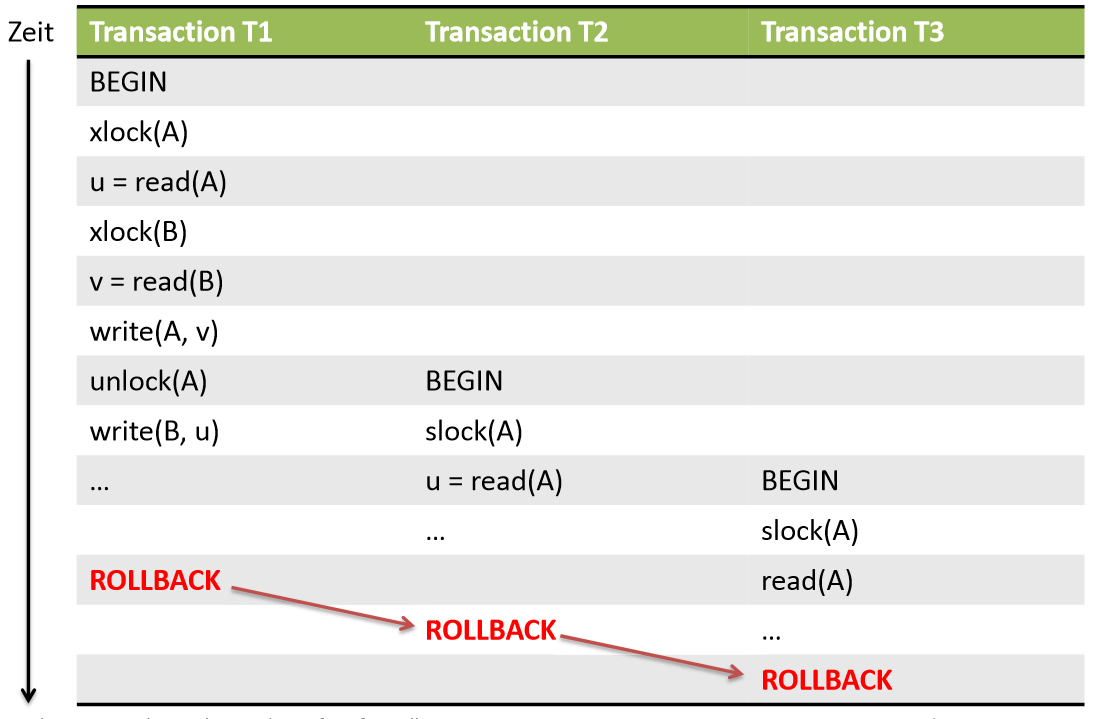
1. Scheduler erkennt Zyklus im Wartegraph
2. Scheduler wählt Transaktion im Zyklus und bricht diese ab(Opferauswahl)
3. Eventuell mehrere Rollbacks(Cascaded Rollbacks)

Es gibt verschiedene Arten von Opfer-Auswahl:

* Transaktion mit wenigsten Sperren
* Transaktion mit meisten Sperren
* Nicht immer dieselbe Transaktion
* Wenn Mehrfache Zyklen, wenn möglich Transaktion, welche möglichst viele Zyklen auflöst

Cascading Rollbacks

Bei 2PL ist es möglich, dass mehrere Rollbacks hintereinander ausgeführt werden müssen. Dieser Vorgang nennt man Cascading Rollbacks:



Eine Transaktion «reisst andere mit»

##### Strict Two-Phase Locking(S2PL)

* Alle gehaltenen Sperren werden erst nach Ende der Transaktion freigegeben
* Keine Cascading Rollbacks
* Deadlocks möglich

#### Isolation Level

In Postgres(und in anderen DBMS) gibt es verschiedene Isolation Level. Diese definiere, wie stark eine Transaktion von anderen Transaktion isoliert sein soll. Es gibt es folgende verschiedene:

* READ UNCOMMITED
  + Lesezugriffe sind nicht Synchronisiert(keine Read-Locks)
  + Diesen IL kann man im Postgres einstellen, wird jedoch nicht verwendet🡪sinnlos
* READ COMMITED(DEFAULT)
  + Lesezugriffe sind temporär synchronisiert
* REPEATABLE READ
  + Einzelne zugegriffene Rows sind synchronisiert
* SERIALIZABLE
  + Vollständige Isolation

Die strengste Isolation ist meistens nicht erwünscht und in der Praxis nicht zu finden.

##### Befehle

Anzeigen des Isolation Levels

**SHOW TRANSACTION ISOLATION LEVEL;**

*Setzten des Isolation Levels*

**SET TRANSACTION ISOLATION LEVEL <LEVEL>**

* READ
* REPEATABLE READ
* SERIALIZABLE

Oder beim Beginn:

**BEGIN TRANSACTION ISOLATION LEVEL <LEVEL>**

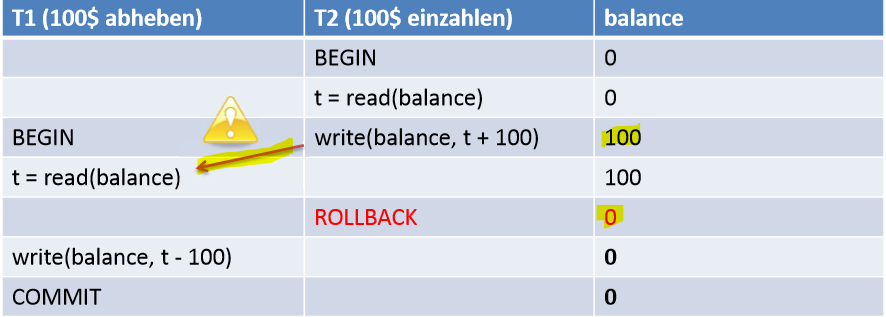
Oder pro Session:

**SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;<LEVEL>**

##### Mögliche Fehlertypen

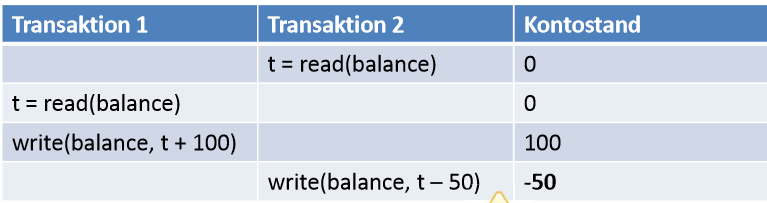


###### Dirty Read

Eine Transaktion 1 liest Daten von einer nicht abgeschlossenen Tranasktion 2:

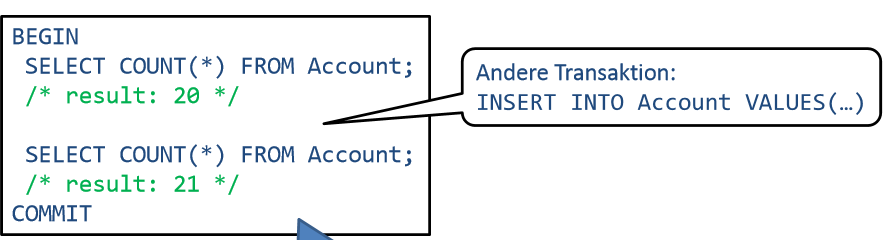
Kann nur in «read uncommited» vorkommen(also in Postgres gar nicht)

###### Fuzzy Read

Gelesene Daten Ändern sich durch parallel Transaktion. Das heisst die andere Transaktion verändert Objekte in der Zwischenzeit:

Kann nur in «read commited» vorkommen, also beim Default von Postgres gibt’s Fuzzy Reads.

Phantom Read

Neue oder gelöschte Zeilen werden erkannt(Tritt in Postgres wegen SI nicht auf):

##### Beispiele Isolation Level

###### BildschirmausschnittRead Commited

###### BildschirmausschnittRepeatable Read

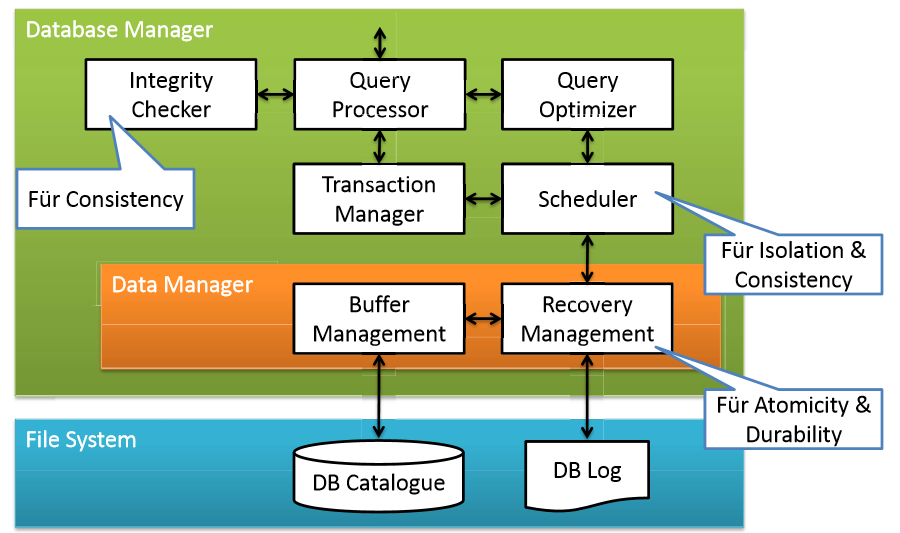
Snapshot Isolation

* Effizient
* In vielen DBMS(Postgres, Oracle, MSSQL)
* Keine Serialisierbarkeit!
* Nicht korrekt bei SERIALIZABLE und REPEATABLE READ

Isolationsverfahren Überblick

## Datenbankarchitektur - Durability / Dauerhaftigkeit

### Grundlagen

Einen kurzen Überblick, wie die Datenbank aufgebaut ist:

* Transaction Manager
  + Koordiniert Transaktionen
* Recovery Manager
  + Ist zuständig für das zurückführen der Daten in einen konsistenten Zustand, falls ein Fehler in einer Transaktion auftritt. Dazu wird ein Log-File benutzt, welches jede Operation dokumentiert
* Buffer Manager
  + Zuständig für die Zwischenspeicherung der Daten, zwischen Haupt und Sekundärspeicher

### Pufferverwaltung

* Transaktion schreibt Änderung in Puffer
* Hintergrundprozess schreibt Änderungen von Puffer in Sekundärspeicher
* Alle Änderungen vermerkt im Log-File somit jederzeit nachvollzierbar was geändert wurde(auch nach Systemausfall)

### Fehlerklassifikation

* Transaktionsfehler
  + Fehler in Applikation oder Transaktionsabbruch(Deadlocks, expliziter Rollback, etc).
  + Transaktion(en) rückgangig machen
  + Millisekunden
* Fehler im Hauptspeicher
  + Fehler DB-System, HW, Stromausfall
  + Verlust von Änderungen, welche noch im Puffer waren(nicht auf Sekundärspeicher)
  + Alle Alle aktiven Transaktionen zurücksetzten, alle noch nicht geschriebenen Transaktionen wiederholen
  + Minuten
* Fehler im sekundär Speicher
  + Disk-Defekt, RAID-Fehler, Natureinflüsse
  + Verlust von allen Daten auf sekundär Speicher
  + Backup Recovery
  + Stunden

### Write-Ahead-Log

Im Write-Ahead Log befinden sich alle Änderungen der Transaktionen.(auch Transaction Log genannt). Somit können alle Transaktionen zurückgesetzt oder wiederholt werden nach einem Absturz und somit ist die Datenintegrität garantiert.  
Das WAL arbeitet mittels Checkpoints, welche das System setzt. Bei einem Absturz müssen die Transaktionen vom letzten Checkpoint bis zum Absturz analysiert werden(undo, redo).

Perfomance Optimierung

### Indexe

#### Grundlagen

Index-Strukturen sind selbständige Datenstrukturen, welche redundant zu einer Tabelle sind. Mittels ihnen ist es möglich, den Lesezugriff aus Kosten von Insert, Update und Delete-Operationen zu beschleunigen.

#### Wieso auf Kosten Modifizierungsoperationen?

Eigentlich ganz einfach: Die Modifizierungsoperationen müssen nicht mehr nur in einer sondern in zwei Strukturen Anpassungen durchführen. Jede Anpassung muss in der Tabellen- sowie in der Indexstruktur vorgenommen werden.

##### Zwei Möglichkeiten

* Heap – doppelt-verkettete Liste
* Suchbaum – Indexeinträge(Defaut)

##### Kriterien

Folgende Aspekte spielen eine wesentliche Rolle, damit man den richtigen Index auswählt:

* DBMS
* Tabellengrösse
* Gibt es oft Mutationen
* Welche Attribute (häufig) benötigt
* Werteberieche der Attribute

##### Index-Kandidaten

Werte die:

* Schlüssel(Normalfall)
* An Joins teilnehmen
* In Gleichheits- oder Bereichsbedingungen vorkommen

##### Relevante Operationen

Operationen, welche für die Indexe relevant sind:

* Scan(Durchsuchen)
  + Seiten, welche Sätze enthalten, werden gelesen
* Equality Search(Suchen in Abhängigkeit eines Gleichheitstests)
  + Seiten, welche Gleichheitsbedingungen erfüllen, werden gelesen
* Range Search(Bereichssuche)
  + Suchen nach einem Bereich
* Insert / Delete
  + Lesen und Schreiben einer oder mehrerer Seiten erforderlich

##### Index Algorithmen

* ISAM
  + Index-Sequential Access Method
  + Erster / Älteste Index-Methode
  + Einfügen/Suchen schnell
  + Aktualisieren schlecht
  + Index mit logischen Adressen(Daten über Indexspalte aufsteigend sortiert)
* B-Tree
  + Balanced Tree
  + Abbildungen von Baum-Knoten auf Seiten im Hintergrundspeicher(Mehrwegsuchbaum)
  + Einfügen/Löschen abhängig von Höhe des Baums
* Hash
  + Hashwerte werden verwendet
  + Vorteil: Schnelle Suche
  + Nachteil: Overflows
* Bitmap
  + Ordnen Attributwerten Bitwerte zu
* Mehrdimensionale Indexe
  + R-Tree

#### Index Befehle

##### Index erstellen

**CREATE INDEX <IndexName> ON <Table(attr)> [USING indexmethod(col)];**

##### Index löschen

**DROP INDEX <IndexName>;**

#### Index in PostrgeSQL

##### Index-Algorithmen

* B-Tree(Default)
* Hash
  + B-Tree in fast allen Aspekten besser
* BRIN
  + Block Range Index
* GiST
* GIN

##### Index-Varianten

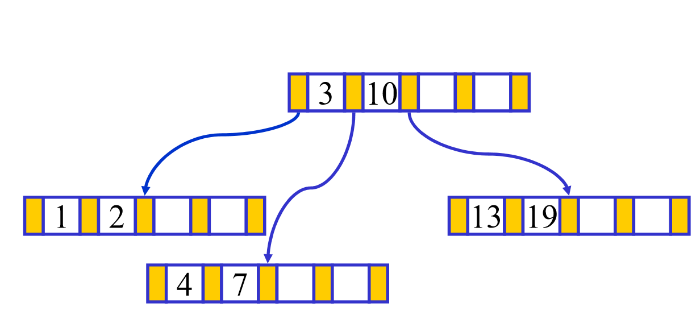
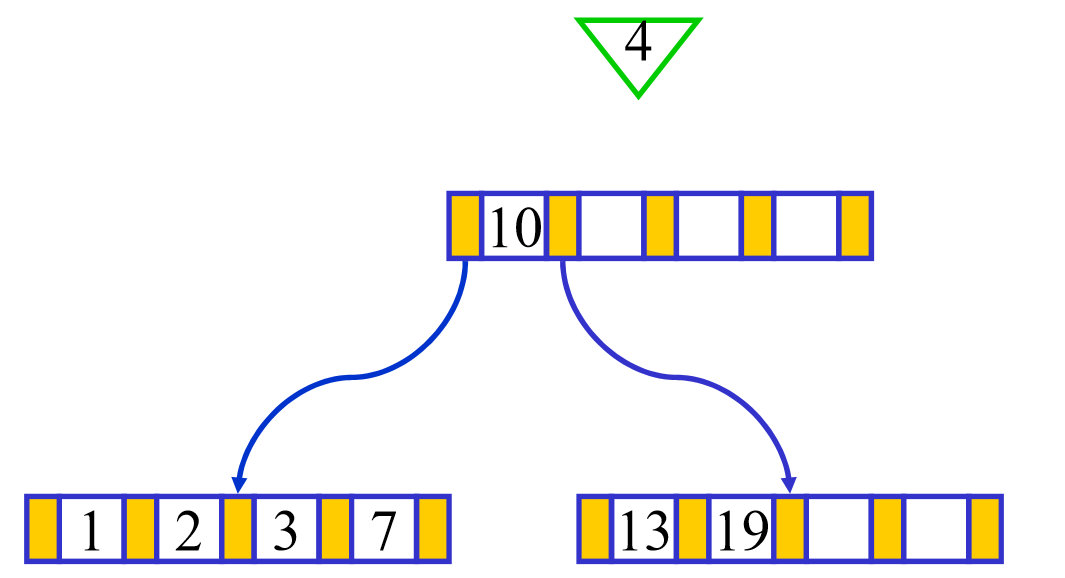
* Clustered Index
  + Sortierter Index
* Primärer/Sekundär Index
  + Primärindex 🡪 Primärschlüssel
  + Sekundärindex 🡪 alle anderen Index
* Zusammengesetzter Index
  + Index über mehrere Kolonnen
  + Tabelle mit zusammengesetztem Schlüssel
  + ACHTUNG! Reihenfolge spielt eine Rolle(siehe Übungen)
* Partieller Index
  + Index mit Where Klausel(Beispiel alle Konten, welche nicht Status eingefroren haben)
* Funktionaler Index
  + Index mit Funktionen

#### Funktionsweise B-Tree

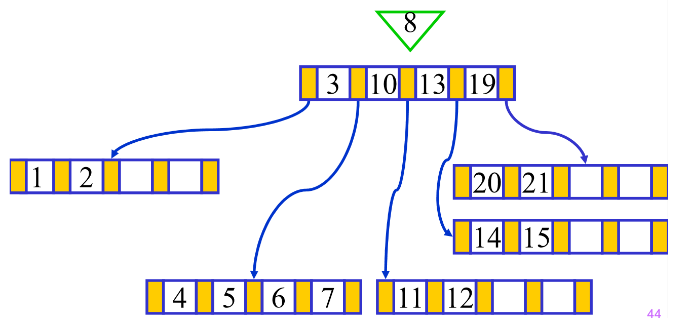
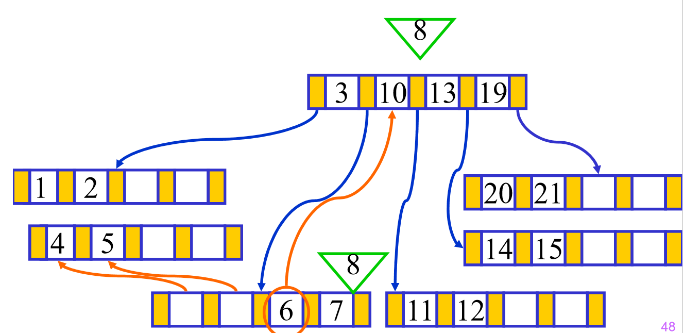
##### Einfügen eines Schlüssels

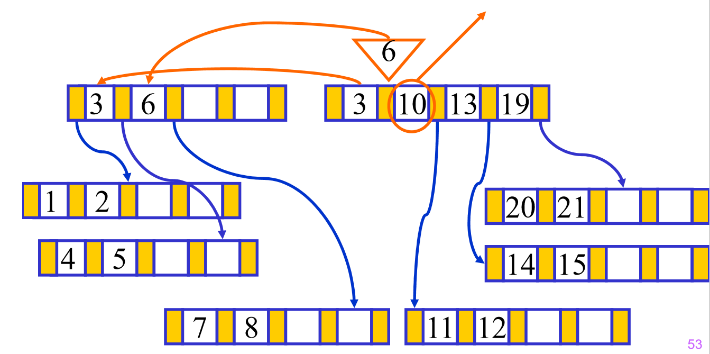
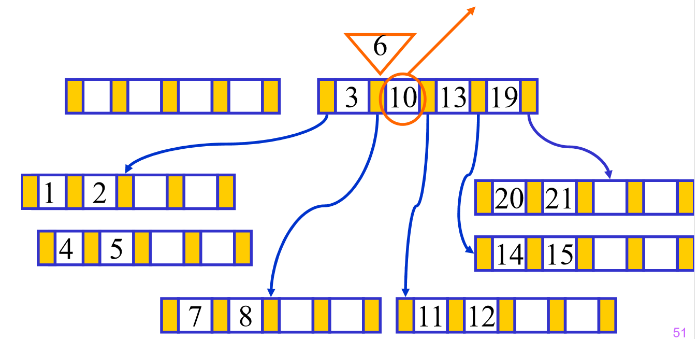
1. Führe eine Suche nach dem Schlüssel durch, diese endet an der Einfügestelle
2. Füge Schlüssel ein.
3. Falls Knoten überfüllt, teile ihn:
   1. Erstelle neuen Knoten mit allen Einträgen des überfüllten Knotens, welche Rechts des mittleren Schlüssels sind
   2. Füge den mittleren Knoten beim Vaterknoten ein
4. Falls Vaterknoten überfüllt
   1. Falls Wurzel, neue Wurzel anlegen
   2. Wiederhole Schritt 2 mit Vaterknoten

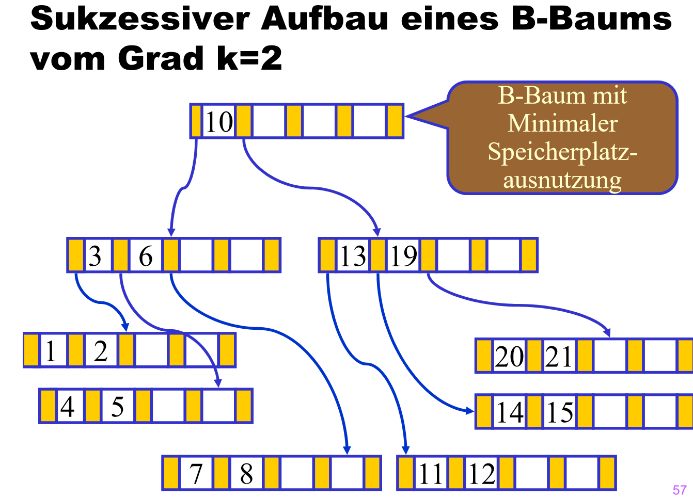
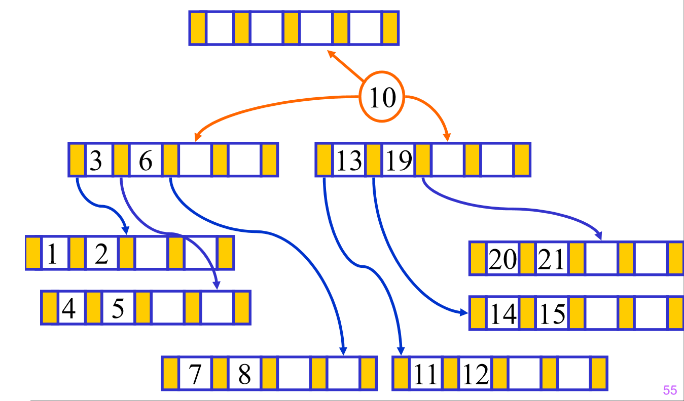
Schritt 3:



Schritt 4:





##### Löschen eines Schlüssels

Das Löschen eines Schlüssels ist das umgekehrte Einfügen. Das wird aber in Datenbanken eher selten angewendet.

#### Logische Optimierungen

* Heuristiken
  + (Zwischen)Ergebnisse so klein wie möglich halten
  + Optimierte Abfragen schreiben
    - Nur benötigte Daten berechnen/laden, etc.
    - So wenige Arbeitsschritte wie ,möglich
  + Schema ok=

#### Physische Optimierungen

* Optimale Algorithmen(Indexe, Scan, etc.) für Anwendungszweck wählen
* Konfiguration der HW anpassen
* Systemarchitektur die richtige(IOPS, Cache, RAID, etc.)

## Java Database Connector(JDBC)

### Bildschirmausschnitt

### Grundlagen

JDBC ist eine API, welches uns ermöglicht per Java auf tabellarische Daten zuzugreifen(vor allem relationale Datenbanken). Es werden viele DBMS unterstützt.

#### Features

* Verbindung zu Data Source(z.B DB)
* Queries und Änderungsbefehle an DB senden
* Resultate empfangen und in Java zugreifen

#### Ähnliche Produkte

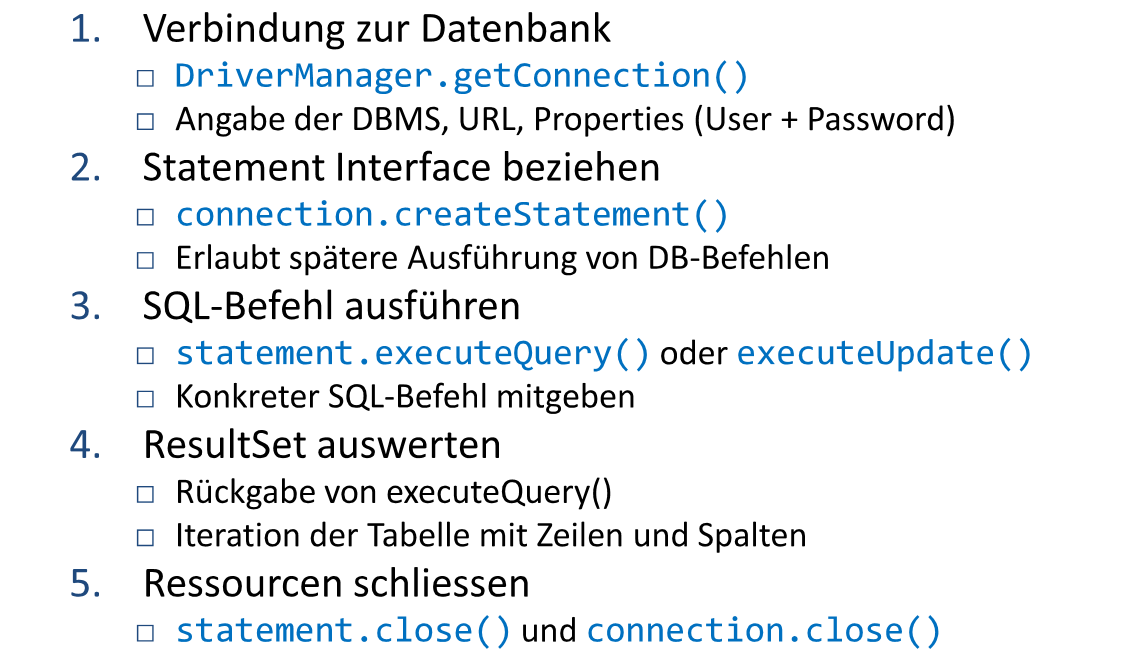
Es gibt ähnliche Produkte, das bekannteste neben JDBC ist wahrscheinlich das Produkt ODBC(Open Database Connectivity), welches auf C basiert. Es ist von Microsft entwickelt worden und wird von Unix unterstützt. Es ist eine Schnittstelle für Relationale DBMS.

#### BildschirmausschnittArchitektur

Wer JDBC einsetzt, kann grundsätzlich zwischen vier verschiedenen Treiber auswählen:

1. JDBC-ODBC Bridge, ODBC auf Client installiert, ODBC-Quellen vorinstalliert
2. Halber Java-Treiber, native DB-Library, auf Client installiert
3. Reiner Java-Treiber, ein Access-Server im Netz vermittelt
4. Reiner Java-Treiber, direkter DB-Zugriff, keine Middleware(verwenden wir)

### Vorgehensweise

Die Vorgehensweise im Java ist in der Regel die gleiche:

Verbindungsaufbau / Connection

Mit einem Connection Objekt wird die Verbindung zum Datenbank Server gemacht.

##### Connection Objekt

**Connection connection = DriverManager.getConnection(database, user, password)**

##### Database URL String

Format: "jdbc:driver://host:port/dbname":

**final String user = "jdbctut";  
final String password = "jdbctut";  
final String database = "jdbc:postgresql://localhost/jdbctut";**

##### Exception Handling

Damit die Exception aufgefangen werden, ist es sinnvoll, die Connection in ein «try with ressources» zu stecken. Dann wird das Connection ebenfalls am Schluss automatisch geschlossen:

**try (**Connection conn = DriverManager.getConnection(database, user, password)**)  
{  
…  
} catch (SQLException ex){  
System.err.println("SQLException: " + ex.getMessage());  
}**

##### Sicherheitsaspekte

In den oben gezeigten Verbindungen werden alle Daten im Klartext übermittelt. Aus Sicherheitsaspekten ist es sinnvoll SSL anzufordern:

**Properties prop = new Properties();  
properties.setProperty("user", username);  
properties.setProperty("password", password);  
properties.setProperty("ssl", "true");  
try (Connection conn = DriverManager.getConnection("jdbc:postgresql:mydb", prop))**

## Ausführen von SQL-Befehlen /Statements

Um SQL Befehle im Java abzusetzen, braucht es ein Statement.

##### Statement Objekt

**try (Statement statement = connection.createStatement()){ … }**

Statement Typen

* **Statement** (Basis-Interface)
  + Für beliebige SQL-Befehle
* **PreparedStatement** 
  + Für serverseitig vorkompilierte Statements
* **CallableStatement** 
  + Helper zum Aufruf von Stored Procedures

Methoden

Beim Statement kann man verschiedene Methoden verwenden:

###### executeQuery

Wird für SQL-Anweisungen verwendet, welche eine Menge von Rows liefert(SELECT). Wird in ein RuleSeT gespeichert, welches eine Tabelle repräsentiert:

**ResultSet Statement.executeQuery(String sql);**

**ResultSet rs = statement.executeQuery(«Select \* from angestellter»);**

###### executeUpdate

Wird für SQL Anweisungen verwendet, welche Daten modifiziert(Insert, Update, Delete). Ebenfalls könne damit auch DDL Anweisungen ausgefügrt werden(Create, Alter, Drop). Rückgabe ist ein Integer, welcher mitteilt, wie viele Datensätze bearbeitet wurden(0 bei DDL):

**int executeUpdate(String sql)**

**int result = statement.executeUpdate(«UPDATE angestellter SET lohn =» +lohn + «where name = ‘» +person + «’»);**

Beachte: Strins müssen in Hochkommas(‘) stehen.

###### execute

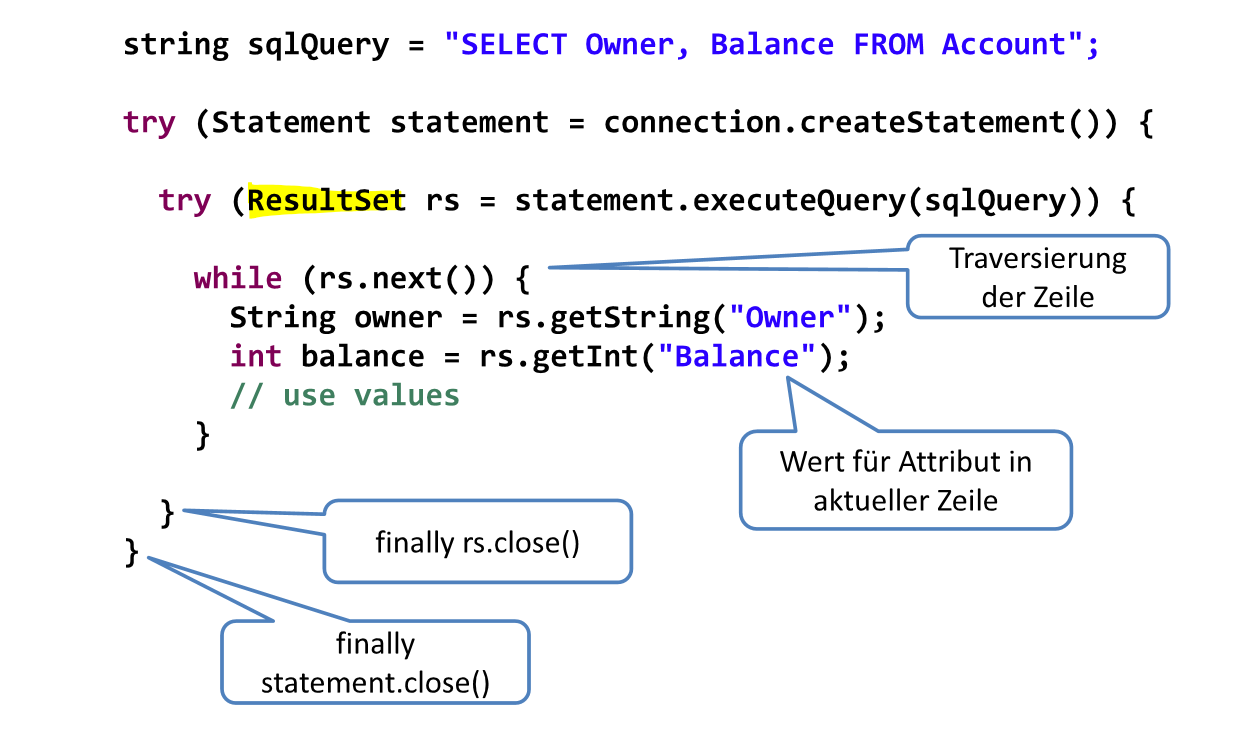
Für beliebige SQL-Anweisungen. Mehr als ein Resultset möglich, gibt true or false zurück:

**Boolean execute()**

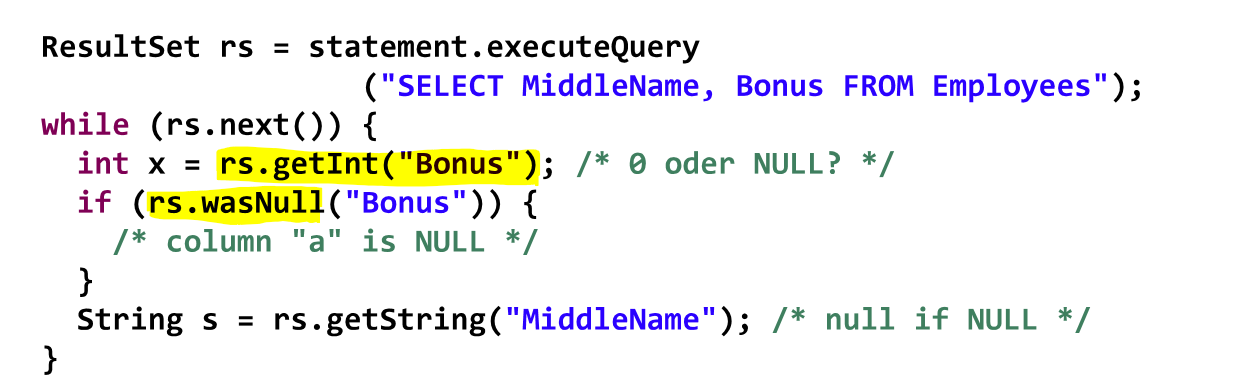
**Boolean**

**boolean = statement.execute(«Drop Database angproj;»);**

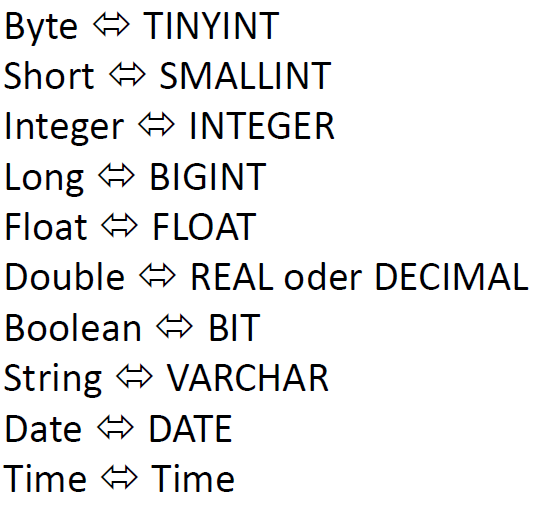
Beispiel

**

Null-Werte

**

Data Type Mapping



Ist ein Wert NULL, so gibt es beim Boolean false und beim int, Double, etc. 0 zurück.

SQL Injections

Auch im Java können SQL-Injections auftreten:

**int result = statement.executeUpdate(«UPDATE angestellter SET salaer = 1000" +"where name like ‘" +name);**

**Eingabe Name:**

**%'; DROP TABLE angestellter; --**

Zuerst werden der Salaer bei allen angepasst, dann die Tabelle gelöscht und dann der Restliche Befehl(«where…») auskommentiert.

Wie kann man sich dagegen schützen:

* Parameter(Eingabe) prüfen
* String Delimiter (') escapen => '' (Doppel-Apostroph)
* Non String prüfen
* Vorkompilierte Statments, etc.

Vorkompilierte Statements

Mit vorkompilierten Statements werden Platzhalter für Paramenter eingefügt:

##### Argumente setzten

**statement.set<type>(<rownumber>, <value>);**

**statement.setInt(1, 12);  
statement.setString(2, "Bob");  
statement.execute(); --Änderungen ausführen**

Transaktionen JDBC

##### Auto Commit Mode

Per Default ist der Auto Commit Modus aktiviert. Das bedeutet, dass jede einzelne SQL-Anweisung eine Transaktion ist. Am Ende der SQL-Anweisung gibt es ein Commit.

##### Explizite Transaktionen

###### Vorgehen

Will man selber über die Transaktionen herrschen, kann man dies wie folgt tun:

1. Auto-Commit ausschalten:

**connection.setAutoCommit(false);**

1. Isolation Level Setzen:

**connection.setTransactionIsolation(int level)**

1. Erste SQL-Anweisung eingeben(Startet automatisch Transaktion)
2. Commit / Rollback implementieren:

**connection.commit();  
connection.rollback();**

###### BildschirmausschnittBeispiel

###### Isolation Level

**connection.setTransactionIsolation(int level)**

Konstanten:

* TRANSACTION\_SERIALIZABLE
* TRANSACTION\_REPEATABLE\_READ
* TRANSACTION\_READ\_COMMITTED
* TRANSACTION\_READ\_UNCOMMITTED

Lesen des aktuellen Levels:

**int level = connection.getTransactionIsolation()**

##### Metadaten

Mittels Metadaten kann man wichtige Info über die DB oder das Ruleset(die Tabelle) erhalten:

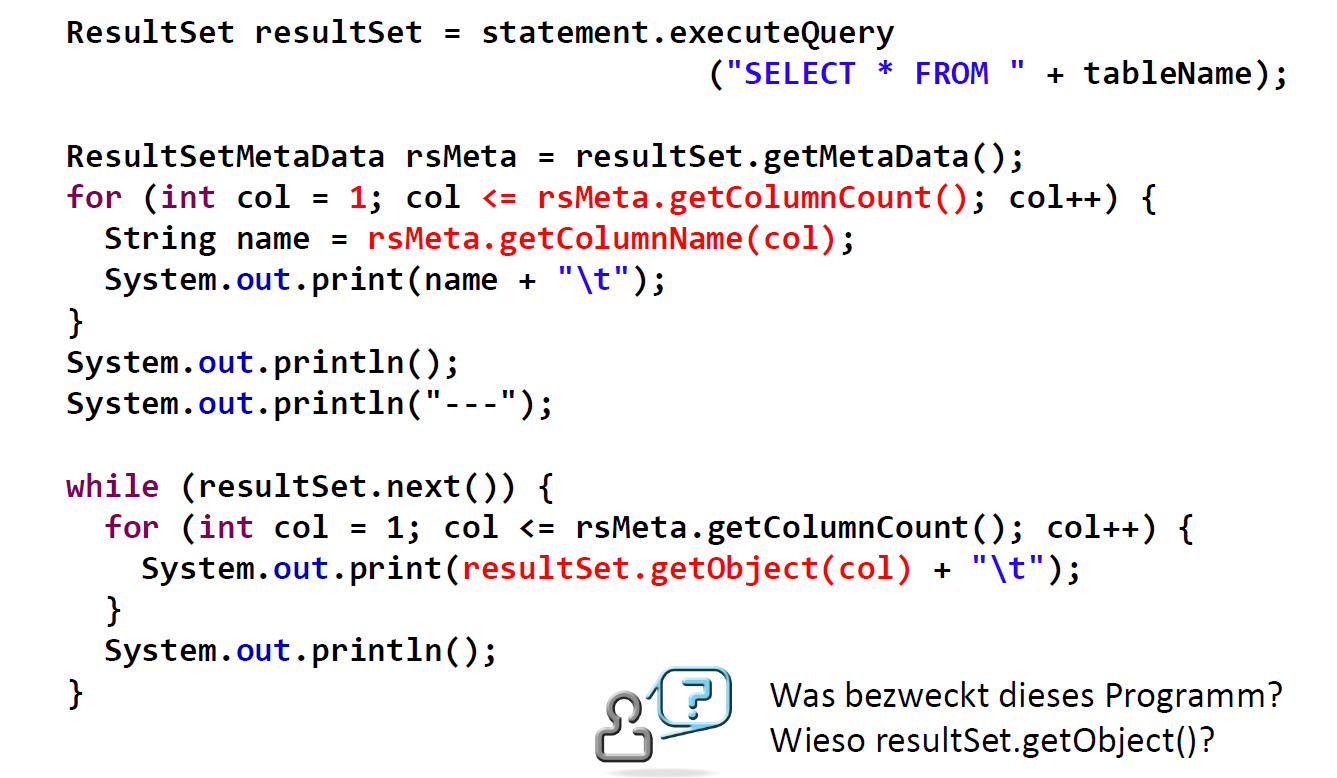
###### Informationen über DB

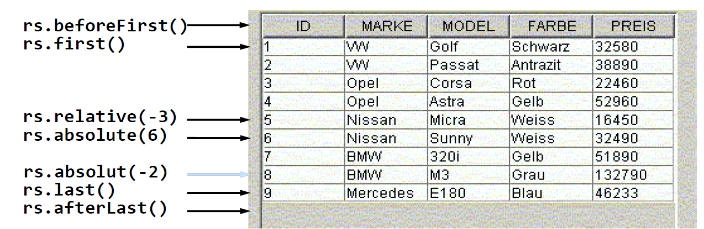
**connection.getMetaData()**

Informationen über RuleSet(Tabelle)

**resultSet.getMetaData()**

* Tabellen Info
  + Name
  + Anzahl Spalten
* Spalten Info
  + Name
  + Datentypen
  + NULLABLE

Beispiel

**Scrollable ResultSet

* Navigation vor- und rückwärts
* Sprünge

##### Scroll-Richtung

**setFetchDirection(int dir)**

* FETCH\_FORWARD
* FETCH\_REVERSE
* FETCH\_UNKNOWN

Updatable ResultSet

* Updates direkt am Ruleset vornehmen
* ResultSet darf nur eine Tabelle beinhalten
* Primary Key muss in Abfrage vorkommen

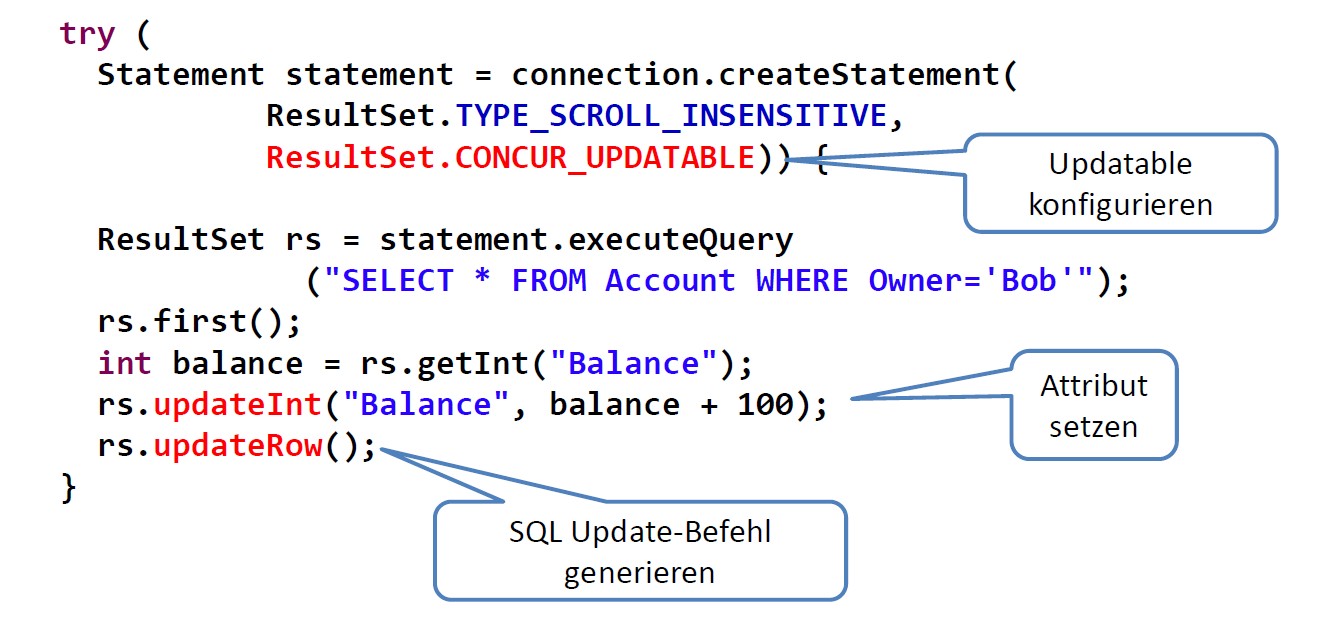
Optionen createStatement()

**Statement stat = conn.createStatement(ResultSet.<Option1>, ResultSet<Option2>**

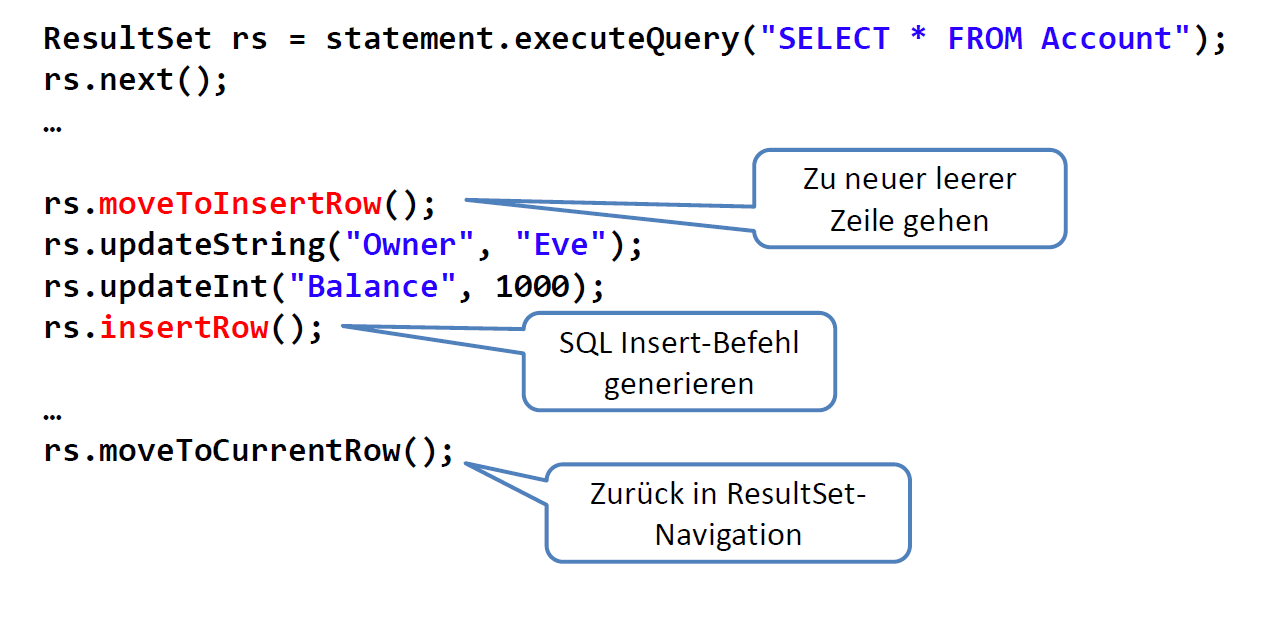
* TYPE\_SCROLL\_INSENSITIVE
  + kann scrollen, Gelesene Rows bleiben in der Regel unverändert
* TYPE\_SCROLL\_SENSITIVE
  + kann scrollen(in alle Richtungen), Gelesene Rows können sich verändern
  + externe Changes und Caching ab
* CONCUR\_READ\_ONLY
* CONCUR\_UPDATABLE

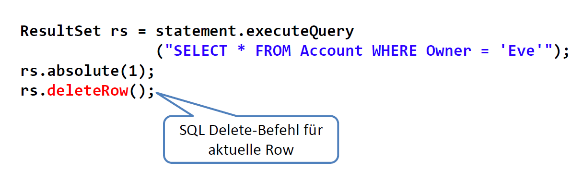
Beispiele

Updaten



Einfügen





#### *Batch Updates*

Batch Updates sind eine Möglichkeit um eine Serie von Anweisungen zu sammeln und auf einmal an die DB zu senden.

* INSERT, UPDATE, DELETE
* CREATE-, DROP-, ALTER TABLE

##### Vorgehen

1. Auto Commit ausschalten
2. Statement erzeugen
3. Statements auf Batch legen:

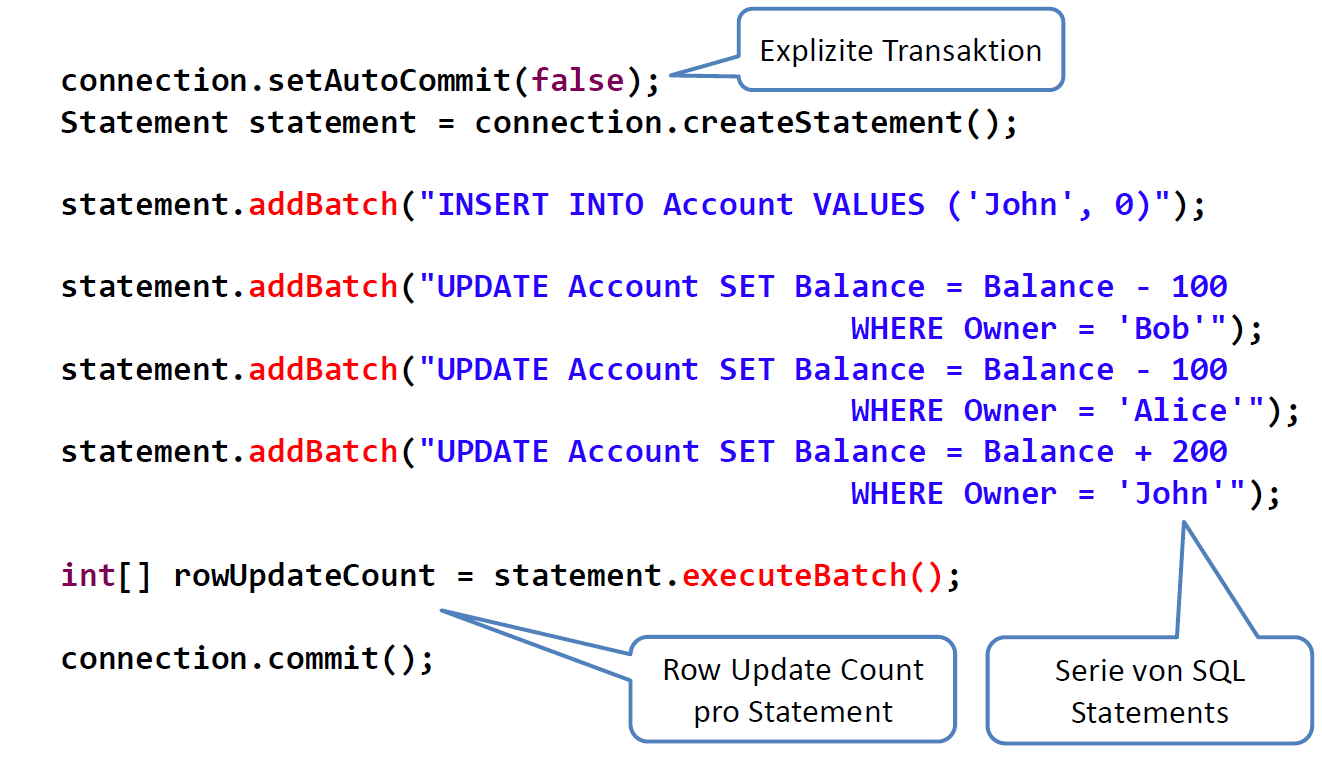
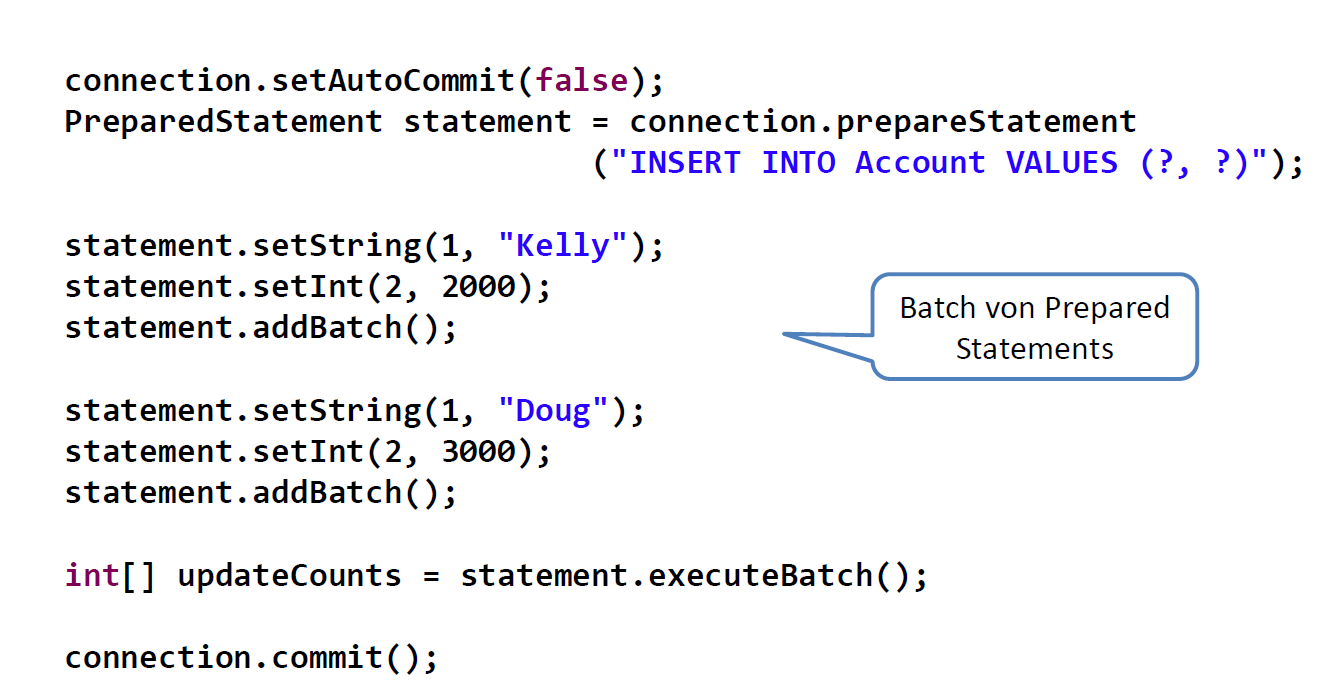
**statement.addBatch("UPDATE Account SET Balance = Balance - 100 WHERE Owner = 'Bob'");**

**Prepared Statements**

**statement.setString(1, "Kelly");  
statement.setInt(2, 2000);  
statement.addBatch();**

1. Batch ausführen

**statement.executeBatch();**

# Datenbank-Sicherheit und Allgemeine Infos

* Identifizierung und Authentisierung von Benutzern
* Überprüfung von Benutzerprivilegien(Autorisierung)
* Kontrolle Benutzung, System Ressourcen
* Kommunikations-Sicherheit(Verschlüsselung)

DBMS ist vor allem Zugriffskontrolle zuständig:

* Authentifizierung
  + Benutzer/Applikation muss sich beim DBMS Authentifizieren(Name + PW)
* Autorisierung
  + DBMS überprüft, ob der Benutzer berechtig ist eine Aktion durchzuführen

## SQL Injections

Wie man sich dagegen wehren kann:

* Spezielle Benutzer mit limitierten Rechnen(kein SuperUser!)
* Input auf Typ prüfen
* Input Escapen
* Stored procedures

## Datentypen

* Boolean
* SMALLINT(2B), INTEGER(4B), BIGINT(8B)
* REAL, FLOAT8 (8B)
* DECIMAL(6,2) Festkommazahlen
* VARCHAR(size)
* DATE
* TIME

## Postgres und psql

In dieser Vorlesung haben wir vor allem PostgreSQL als DBMS eingesetzt und PSQL Zeilenkommandotool.

## Der Nullwert

Wichtig zu wissen. Vergleichsoperationen mit einem Null Wert geben im Postgres den Wert UNKNOWN zurück.

## Schema und Owner

Einer Datenbank kann ein Schema zugeordnet werden. Ein Schema ist ein Namensraum unterhalb der Datenbank, dem Datenbankobjekte(Tabellen, Views, Triggers, Stored Procedures, etc.) zugeordnet werden kann. Standard Schema heisst «public».

Nach dem Datenbank name kann der Datenbank der Owner mitgegeben werden. Diese Role hat dann die Rechte um DB-Objekte zu erstellen, modifizieren und löschen.

Bildschirmausschnitt

Es lohnt sich auf jeden Fall immer eine/n role/user zu erstellen, welcher nachher Besitzer der Datenbank ist. Dann kann man sich mit diesem User anmelden und muss nicht den SuperUser verwenden.

Es wird empfohlen, bevor die Datenbank erstellt wird, die Datenbank zu löschen, dass man die Datenbank «sauber» erstellen kann.

## DB Programmierungen

Es gibt verschiede Möglichkeiten, wie man programmieren mit einer Datenbank verbinden kann:

* Serverseitig
  + (Stored Procedures)
  + Prozedurale SQL Sprachen
    - PL/SQL(Oracle), Transact SQL(MSSQL), PL/pgSQL(Postgres)
    - Java, Phyton, .Net
* Clientseitig
  + Database Connectivity
  + Applikation via SQL Befehle auf DB zu
  + JDBC
  + ODBC oder ähnliches