Datenbanken – Zusammenfassung

Einführung, relationales Datenmodell

Begriffe

- **Zeichen:** entstammen einem **Zeichenvorrat**, der beliebig angeordnet sein kann.
- **Daten:** Es handelt sich hier um bereits **strukturierte Zeichen**. Das können Zeichen aus Alphabete, Zahlen, Satzzeichen sowie Piktogramme sein. Sie befinden sich immer auf einem **Datenträger**, (z.B. Blatt Papier).
- **Information:** Erhält man, wenn diese **Daten** in einen **Kontext** gestellt werden.
- **Wissen:** Wenn es zur **Verknüpfung** von Informationen und zur **intellektuellen Einbettung** kommt.
- **Beispiel:** Zeichen: «0», «1», «.»; Daten: «50», «Hans»; Information: «50 Km/h», «Vorname: Hans»
- Wissen: «Dieses Auto fährt mit 50 Stundenkilometer»

Strukturierte Daten (Synonym: Formatierte Daten)

- Daten, die eine **reguläre**, **fest** vorgegebene **Struktur** besitzen.
- Mehrere «gleichartige Datensätze» mit identischem Aufbau. Gut «tabellarisch» darstellbar.

Unstrukturierte Daten (Synonyme: Unformatierte oder formatfreie Daten)

- Daten, für deren Elemente keine bestimmte Anordnung bzw. Struktur vorgeschrieben ist.
- Sie haben keine explizite Struktur (aber ev. eine implizite, z.B. Grammatikregeln für Text).
- **Beispiele:** Texte (können auch strukturiert sein, z.B. gewisse Formulare), Bilder, Filme, Audiodaten, …

Semi-strukturierte Daten

- Daten, die teilweise **irreguläre** bzw. **unvollständige Strukturen** aufweisen, die sich öfters ändern können.
- Das Schema kann evtl. aus zusätzlicher Information (sog. „markup“) (re)konstruiert werden.

Datenarten ↔ Technologien

- Unterschiedliche Datenarten verlangen nach unterschiedlichen Technologien:
  - Strukturierte Daten: Relationale Datenbanken
  - Semi-Strukturierte Daten: XML, JSON, …
  - Unstrukturierte Daten: z.B. Information Retrieval

Persistenz (Partnerbegriff: Volatilität)

- Zentrales Ziele bei der Verarbeitung von Daten in elektronischer Form ist die Sicherstellung der Persistenz.
- Persistenz (lat. persistere «beharren») bezeichnet die Fähigkeit, Daten über lange Zeit (insbesondere über die Laufzeit eines Programmes hinaus) bereitzuhalten. Dies benötigt nichtflüchtige Speichermedien.

Datenverwaltung mittels Dateisystemen

- Speicherung von Daten in Dateien (verwaltet vom Betriebssystem)
- Anwendungen/Programme lesen/schreiben Daten direkt
- **Vorteile:**
  - Einfach, auf Anwendung angepasst, effizient implementierbar
  - Anwendung muss keine Rücksicht nehmen auf „andere“
  - Proprietäre (nicht allgemein anerkannten Standards entsprechend) Formate möglich
- **Nachteile:**
  - Probleme bei Mehrfachverwendung der Daten für unterschiedliche Zwecke
  - Datenstrukturänderung bedeutet i.d.R. Programmänderung
  - Gleichzeitiger Zugriff aufwändig zu realisieren
  - Abgestufte Zugriffsrechte aufwändig zu realisieren
  - Daten werden oft mehrfach gespeichert
  - Datenaustausch, -integration komplex

  ➔ Logische und physische Daten**ab**hängigkeit!
- Nach wie vor sehr weit verbreitet
  - Office-Anwendungen (Word, Excel, ...)
  - Buchhaltung, etc.
- **Deshalb:** Einsatz von **Datenbanksystemen**

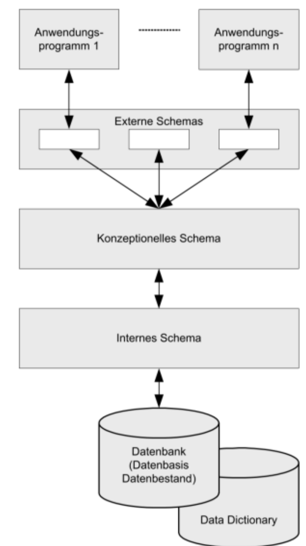
Datenbanksystem

- **Datenbanksystem** (DBS)
  - DBMS (Datenbankverwaltungssystem)
  - Datenbank (Datenbasis, Datenbestand)
- **Anwendungsprogramme** (AP)
- DBS + AP = **Informationssystem** (IS)

Relationale Datenbanken

- Relationale Datenbanksysteme sind hervorragend geeignet, um **strukturierte Daten** zu verwalten.
- Technologie/Produkte haben sich über Jahre entwickelt, sind entsprechend leistungsfähig und ausgereift.
- **Relationale Datenbankverwaltungssysteme** sind – trotz Schwächen bei der Verwaltung von semi-strukturierten und unstrukturierten Daten – heute immer noch die weitaus **wichtigste Datenbanktechnologie**.
- Die Entwicklung der letzten paar Jahre ging v.a. in Richtung «**Ausbau**» der relationalen Technologie.

3-Ebenen-Architektur eines RDBS



- 1975 von ANSI/SPARC (American National Standards Institute, Standards Planning And Requirements Committee) entwickeltes Modell. Gilt im Grundsatz heute noch.
- Beschreibung relationalen Datenbanksystems auf verschiedenen Ebenen (unterschiedliche Sichtweise):
  - **Konzeptionell** Logische Gesamtsicht
  - **Extern** Sicht einer einzelnen Anwendung / Benutzergruppe
  - **Intern** Speicherung, Datenorganisation, Zugriffsstrukturen
- Realisiert durch ein relationales **Datenbankverwaltungssystem** (RDBMS).
- Datenbeschreibungen («Meta-Daten») ebenfalls in der Datenbank gespeichert («Data Dictionary»).
- Zwischen den verschiedenen Ebenen (Schemas) erfolgen **Transformationen** zur Erreichung folgender Ziele:
  - **Logische Datenunabhängigkeit:** Externes Schema ändert → modifizieren der Transformationsregeln zum konzeptionellen Schema.
  - **Physische Datenunabhängigkeit:** Internes Schema ändert → modifizieren der Transformationsregeln zum konzeptionellen Schema.

Aufgaben/Eigenschaften eines DBMS

1. Zentrale Kontrolle über die operationalen Daten
2. Hoher Grad an Datenunabhängigkeit
3. Hohe Leistung und Skalierbarkeit
4. Mächtige Datenmodelle und Abfragesprachen / leichte Handhabbarkeit
5. Transaktionskonzept (ACID), Datenkontrolle
6. Ständige Betriebsbereitschaft (hohe Verfügbarkeit und Fehlertoleranz)
   - 7x24h-Betrieb
   - keine Offline-Zeiten für DB-Reorganisation u.ä.

➔ Effiziente und flexible Verwaltung grosser Mengen persistenter Daten.

Vorteile von Datenbanksystemen

- **Datenkonsistenz** (= Datenintegrität) einfacher sicherzustellen
  - Korrektheit (nur der Abbildung der Realität, nicht der Daten selbst)
  - Vollständigkeit
  - Arten der Datenkonsistenz:
    - Semantische Integrität: Inhaltliche Korrektheit (NICHT Richtigkeit). Z.B. erlaubter Übergang: ledig → verheiratet
    - Operationale Integrität: Korrekter Datenbestand während dem DB-Betrieb.
- Mehrere Anwendungen können **gleichzeitig** auf **dieselben** Daten zugreifen.
- Die Anwendungen sind (weitgehend) unabhängig von:
  - Physischer Datenstruktur («physische Datenunabhängigkeit»)
  - Erweiterungen der Daten («logische Datenunabhängigkeit»)
- Verwaltung & Nutzung **sehr grosser Datenmengen**.
- **Deklarativer** und **mengenorientierter Zugriff**.
- Einmalige, **zentrale Datendefinition**.
- **Automatisierung** wichtiger **Aufgaben**:
  - Integritätskontrolle
  - Redundanzverwaltung
  - Zugriffskontrolle
  - Zugriffsoptimierung
  - Gleichzeitiger Zugriff
  - Datensicherung/-wiederherstellung
- **Aber: Aufbau und Betrieb sind anspruchsvoll und teuer.**

Datenbankentwurf/-betrieb

- Aufbau, Ausbau:
  - Erstellen von verschiedenen Schemas
  - Hilfsmittel: ERM (Entity-Relationship-Model) und DDL (Data Definition Language) des DBMS
  - Achtung: stark iterativer Prozess!
  - Ausbau und Umbau im laufenden Betrieb möglich und typisch!
- Betrieb, Benutzung, Verwaltung:
  - Abfragen, Einfügen, Ändern, Löschen von Daten
  - Sicherung & Wiederherstellung
  - Überwachung & Tuning
  - Benutzerverwaltung & Rechtevergabe
  - Hilfsmittel:
    - DML (Data Manipulation Language) des entsprechenden DBMS
    - DQL (Data Query Language) des entsprechenden DBMS
    - DCL (Data Control Language) des entsprechenden DBMS

Gründe für den Erfolg des relationalen Modelles

- Einfache Datenstruktur: Relation («mathematisches Objekt»).
- **Mengenorientierte** Verarbeitung der Daten, alle Operationen führen wieder zu Relationen.
- **Wenige Grundoperationen** zur Verarbeitung und dadurch eine klare Semantik (→ relationale Algebra).
- **Formale Theorie** zur Modellierung und Anfrageverarbeitung.
- Implementationen sind relativ einfach zu benutzen und erfolgen alle über SQL:
  - DDL (data definition language)
  - DML (data manipulation language)
  - DQL (data query language)
  - DCL (data control language)

Grundbegriffe des relationalen Modelles

- Um Daten in einer Datenbank speichern zu können, muss man sie vorab entsprechend strukturell beschreiben («**definieren**»). Dazu verwendet man sogenannte Datenmodelle (es gibt viele verschiedene).
- Das relationale Datenmodell zeichnet sich dadurch aus, dass zur Beschreibung von Daten nur wenige Konzepte und Begriffe nötig sind:
  - Domänen
  - Attribute
  - Tupel
  - Relation
  - Schlüssel
  - Format, Schema, Heading, Relationsvariable
- Relationen haben grossen Vorteil, dass sie als «Tabellen» darstellen lassen (Achtung: Relation ≠ Tabelle)!

Bezeichnungen einer Relation

- **Relation:** «Tabelle», hat einen Namen: Studierende
- **Attribut: eine** Eigenschaft, Mat.-Nr oder Name…
- **Format/Schema:** die Gesamtheit aller Attribute (Header)
- **Tupel:** Eine ganze Zeile, hier bspw. (<1, Meier, 19.4.1992>)

Studierende

Matrikelnummer	Name	Geburtstag
1	Meier	19.4.1992
2	Müller	23.8.1998
3	Huber	11.9.2001

Attribut · Format, Schema · Attributwert · Relation · Tupel

Wertebereich / Domäne

- Menge einfacher bzw. «atomarer» Werte (entspricht einem **Datentyp** einer höheren Programmiersprache).
- Im Datenbankumfeld bezeichnen Domänen **endliche Mengen** von Werten **desselben Typs**.
- **Attributwerte** müssen **immer** genau **einer Domäne** entstammen.
- **Beispiele:**
  - Ganze Zahlen, Fixpunktzahlen (Dezimalbrüche), Gleitkommazahlen, …
  - Menge aller Zeichenketten, evt. einer festen Länge
  - Aufzählungstypen, z.B. {red, green, blue}, {CHF, EUR, GBP, USD}
  - Unterbereichstypen, z.B. ganze Zahlen im Intervall [100 ... 999], etc.
- Zweck von Domänen:
  - **Theorie:** Nur Attribute mit gleichen Domänen sind «kompatibel», z.B. bei Vergleichen, arithmetischen Operationen etc., Attribute mit verschiedenen Domänen sind nicht kompatibel.
  - **Praxis:** Nicht relevant, die meisten RDBMS unterstützen nur vordefinierte Domänen wie INTEGER, VARCHAR(n) etc. – und das ist auch gut so!
- Domänen dürfen **nur** sogenannte «**atomare**» Werte enthalten, d.h. nur ein **einzelner Wert** (der aber durchaus «komplex» sein darf, z.B. auch wieder eine Relation). Nicht zulässig mehrere Werte im gleichen Feld!

Attribut

- «Eigenschaften, die uns interessieren»
- Ein Attribut besteht aus zwei Teilen:
  - Bezeichnung/Name
  - Domäne/Wertebereich, aus der die zugehörigen Werte stammen können.
- Menge von Namen von Attributen $\{A_1, \ldots, A_n\}$
- Ein Attribut nimmt konkrete Werte an (können sich im Laufe der Zeit ändern, stammen immer aus der Domäne des Attributes) → **Attributwerte** (Bsp. Umsatz / 200)
- **Beispiel (Attribut / Attributwert):** Anrede / {„Herr“, „Frau“} oder Ort / string[30]

Tupel (n-Tupel) – <x, y, z, …>

- Sammlung von als zusammengehörig betrachteter Attributwerte:
  - Feste Zahl von Komponenten
  - Beliebige Anordnung (d.h. Reihenfolge ist erst wichtig, wenn einmal festgelegt)
  - Der Attributwert entstammt der für jedes Attribut festgelegten Domäne.
- **Wichtig:** Die **Attributwerte verändern** sich über die Zeit, die **Attribute** selbst, also Name, Bedeutung und Wertebereich, bleiben **konstant.**

Schema, Format, Heading

- Eine **Menge von** als **zusammengehörig** betrachteter **Attribute**:
  - Feste Anzahl
  - Beliebige Anordnung (d.h. Reihenfolge ist erst wichtig, wenn einmal festgelegt)
- Kann ein **Schema** auch als **Definition** eines **Datentyps** vorstellen. Im Beispiel wurde ein Datentyp «Studierende» definiert, mit den drei Bestandteilen Matrikelnummer, Name und Geburtstag.
- Formate sind wie Variablen, die zu unterschiedlichen Zeiten unterschiedliche Werte annehmen können.

Relation – $R = \{<\ldots, \ldots, \ldots>, <\ldots, \ldots, \ldots>, <\ldots, \ldots, \ldots>\}$

- Eine Relation R besteht aus zwei Elementen:
  - Einem sogenannten **Relationenformat**: Menge von Namen von Attributen $\{A_1, \ldots, A_n\}$, wobei jedem Attributnamen A_i eine Domäne (Wertebereich) $dom(A_i) \in \{D_1, \ldots, D_m\}$ zugeordnet wird.
  - Einer sogenannten **Ausprägung/Extension** (d.h. einer Menge von Tupeln, die dem Relationenformat entsprechen): $ext(R) \subseteq D_1 \text{ x} \ldots \text{x } D_n$
- «Eine Relation ist eine **Teilmenge** des **kartesischen Produktes** von **n endlichen Wertebereichen**».
- Wichtig: Relationen sind **Mengen**, enthalten also **keine doppelten Elemente** und sind **ungeordnet**!

Notationen

- Für Darstellung von Relationen gibt es verschiedene Notationen. Domänen werden oft weggelassen
  - **Tabellarisch** (siehe Beispiel oben «Studenten»)
  - Als **Menge** von **Tupeln** mit zughörigem Format:
  - **Beispiel:** {<1, Meier, 19.4.1992>, <2, Müller, 23.8.1998>, <3, Huber, 11.9.2001>} zum Format **Studierende** (Matrikelnummer, Name, Geburtstag) → **Angabe Format ist zwingend!**
  - Nur als **Format** (**Kurznotation**): – Studierende (Matrikelnummer, Name, Geburtstag)

Schlüsselkandidat

- Gegeben: Eine Relation R mit dem Schema $R\{A_1, \ldots, A_n\}$
- Eine nicht-leere Teilmenge von Attributen $K \subseteq \{A_1, \ldots, A_n\}$ heisst **Schlüsselkandidat** wenn gilt:
  - Für je zwei Tupel gilt zu **jedem Zeitpunkt**: Falls sie in den Attributwerten von K übereinstimmen, müssen sie gleich sein (d.h. es gibt nicht zwei Tupel mit denselben Schlüsselattribut**werten**) **und**
  - Man kann in K (= Menge der Schlüsselattribute) **nichts weglassen**, ohne diese Eigenschaft zu verlieren.
- Wenn mehrere Schlüsselkandidaten zur Verfügung stehen, kann eine Auswahl getroffen werden.
- **Beispiel Schlüsselkandidaten:** Kd.-Nr., Bestell-Nr., Kombination zweier oder mehrerer Attribute, etc.

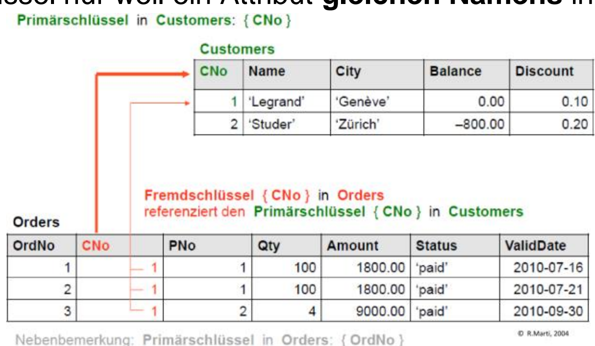
Primärschlüssel (primary key, PK)

- Ein **ausgewählter** Schlüsselkandidat, der **explizit** als Primärschlüssel bezeichnet wird.
- Kriterien für Auswahl des Primärschlüssels (allg. von Schlüsseln):
  - **Attributwert(e)** sollten sich möglichst wenig ändern (idealerweise nie).
  - **Eindeutigkeit** der Werte eines Primärschlüssels sollte über die Zeit gelten, d.h. ein einmal verwendeter Wert sollte später nicht wiederverwendet werden.
  - **Attribut(e)** sollten möglichst wenig Speicherplatz benötigen («**kurze Schlüssel**»)

  ➔ Entwurfsentscheid
- Wenn nichts passt (aber nur dann): **Surrogatschlüssel** («künstlicher Schlüssel») definieren.

Fremdschlüssel (foreign key, FK)

- Eine Menge von Attributen in einer Relation S zu der es eine Relation R gibt, deren Primärschlüssel von diesen Attributen in S referenziert werden. Ein Fremdschlüssel (in S) kann, muss aber nicht Schlüssel in S sein. → Konzept der **referentiellen Integrität**.
- Die Attributnamen des Primärschlüssels von R und des Fremdschlüssels von S **müssen nicht gleich** sein.
- Ein Attribut in S ist **nicht notwendigerweise** ein Fremdschlüssel nur weil ein Attribut **gleichen Namens** in einer anderen Relation R als Primärschlüssel auftritt.
- Primärschlüssel/Fremdschlüssel-**Beziehungen** müssen **explizit deklariert werden**.
- Es ist empfehlenswert, für sich entsprechende Fremdschlüssel- und Primärschlüsselattribute die **gleichen Namen** zu verwenden (geht aber nicht immer).
- **Beispiel: Primär-/Fremdschlüssel →**



CNo	Name	City	Balance	Discount
1	'Legrand'	'Genève'	0.00	0.10
2	'Studer'	'Zürich'	–800.00	0.20

OrdNo	CNo	PNo	Qty	Amount	Status	ValidDate
1	1	1	100	1800.00	'paid'	2010-07-16
2	1	1	100	1800.00	'paid'	2010-07-21
3	1	2	4	9000.00	'paid'	2010-09-30

Wichtig:

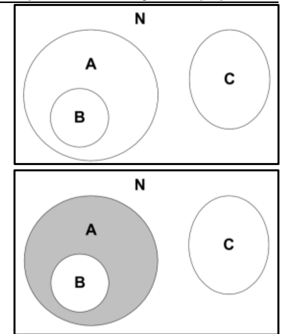
- Im relationalen Modell gibt es **NIE zwei gleiche «Zeilen»** (solche, die in allen Attributwerten identisch sind).
- In einem Format gibt es **NIE zwei gleiche Attribute** («**Spalten**»).
- Die **Reihenfolge** der «Zeilen» (Tupel) und «Spalten» (Attribute) ist **irrelevant**.

Naive Mengenlehre

- Jede betrachtete Menge ist genau definiert.
- Die in einer Menge enthaltenen **Objekte** nennt man **Elemente**.
- Die Elemente einer Menge werden, durch Kommas getrennt, zwischen geschweifte Klammern geschrieben.
- Das Konzept der Menge kennt **keine Reihenfolge** ihrer Elemente, weshalb $\{1,2,3\}$ identisch mit $\{2,3,1\}$ ist.
- Geht immer um Frage: **ob** ein Element **enthalten** ist in einer Menge, **oder** ob **nicht** enthalten ist.
- **Ein Element** ist in einer **Menge** also **höchstens einmal enthalten**.

- Ist ein Element in einer Menge enthalten schreibt man: $3 \in \{1,2,3\}$ oder wenn nicht enthalten: $4 \notin \{1,2,3\}$
- Die Betrachtungen von Mengen und deren Elemente bezieht sich stets auf eine **Grundmenge**.
- **Beispiel:** Menge aller geraden Primzahlen $Q = \{n \in \mathbb{N} |\ n\ ist\ eine\ Primzahl\ und\ gerade\}$, somit $Q = \{2\}$

Venn Diagramme



- Ist eine Variante der Darstellung von Mengen und deren Beziehungen.
- Das umschliessende Rechteck kennzeichnet jeweils die Grundmenge.
- **Beispiel:** $A = \{x \in \mathbb{N} |\ x\ ist\ gerade\ und\ x < 100\}$,
  $B = \{x \in \mathbb{N} |\ x\ ist\ durch\ 6\ teilbar\ und\ x < 100\}$, → → →
  $C = \{x \in \mathbb{N} |\ x\ ist\ gerade\ und\ x > 100\}$
  Hier ist die Menge B in der Menge A enthalten, das heisst jedes Element von B ist auch eines von A. Die Mengen C und A haben keine Elemente gemeinsam.
- Im folgenden Diagramm ist ein bestimmter Bereich schraffiert, nämlich derjenige welcher zur Menge $X = \{x \in \mathbb{N} |\ x \in A\ und\ x \notin B\}$ gehört. → → →
- Man kann Menge X auch bilden, indem man A als Grundmenge bezieht: $X = \{x \in A \mid x \notin B\}$
- **Durchschnitt ∩:** Die Menge, die die Kriterien für A **und** B gemeinsam erfüllen! → $A \cap B$
- **Vereinigung ∪:** Die Menge, die die Kriterien für A **oder** B erfüllen. → $A \cup B$
- **Differenz \:** Die Menge, die die Kriterien für A, jedoch ohne die Schnittmenge $A \cap B$ erfüllen. → $A \setminus B$
- Es gilt: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C),\ A \cap (B \cap C) = (A \cap B) \cap C,\ A \cup (B \cup C) = (A \cup B) \cup C$

Grundoperationen auf Relationen

Notationen für Relationen

- Wenn nur das Format (Schema) einer Relationen interessiert, wird oft folgende Kurzschreibweise verwendet (Domänen werden in der Regel weggelassen): $R\{A_1, \dots, A_n\}$
- Wenn vorwiegend die Menge der Tupel interessiert, wird oft folgende Mengenschreibweise verwendet: Beispiel: {<Tischtennis, 1.10.1990, 15.00>, <Schwimmclub, 31.7.1988, 10.00>, <PC, 1.10.1990, 15.00>} zum Format (die **Formatangabe** ist zwingend): Sportclub(Clubname, Gründungsdatum, Jahresbeitrag)

Äquivalenz

- Zwei Relationen sind äquivalent wenn sich die eine durch Umordnen der Attribute der anderen herstellen lässt. → Darstellung: R1 ~ R2
- **Beispiel:** BUCH1(ISBN, Titel, Jahr, Auflage, Fachgebiet), BUCH2(Titel, ISBN, Auflage, Jahr, Fachgebiet) BUCH1 ~ BUCH2
- **Annahme:** Die Domänen stimmen in den beiden Formaten überein und die Attribute bedeuten dasselbe.

Relationale Algebra (RA)

- Die **Rechenvorschriften** definieren eine «Abfragesprache», d.h. eine Sprache, mit der beliebige Abfragen an eine Datenbank gestellt werden können.
- **Inhalt** der **Datenbank** (Relationen) sind **Operanden**.
- **Operatoren** definieren Funktionen zum Berechnen von Anfrageergebnissen: «Grundlegenden Dinge, die wir mit Relationen tun wollen». Symbole, die Prozeduren repräsentieren.

Kriterien für Abfragesprachen

- **Ad-Hoc-Formulierung:** Benutzer sollen Anfrage formulieren, ohne ein Programm schreiben zu müssen.
- **Deskriptivität / Deklarativität**: Benutzer sollen formulieren «Was will ich haben?» und nicht «Wo finde ich das, was ich haben will?»
- **Optimierbarkeit:** Sprache besteht aus wenigen Operationen. Optimierungsregeln für Operatorenmenge.
- **Abgeschlossenheit:** Anfragen erfolgen auf Relationen. Anfrageergebnis ist wiederum eine Relation und kann als Eingabe für die nächste Anfrage verwendet werden → Schachtelung möglich.
- **Mengenorientiertheit**: Operationen auf Mengen von Daten. Nicht navigierend auf einzelnen Elementen.
- **Sicherheit:** Keine Anfrage, die syntaktisch korrekt ist, darf in eine Endlosschleife geraten oder ein unendliches Ergebnis liefern.

Klassifikation der relationalen Operatoren

- **Entfernende Operatoren:** Selektion, Projektion
- **Umbenennung:** Verändert nicht Tupel, sondern Schema
- **Kombinierende Operatoren:** Kartesisches Produkt, Join
- **Mengenoperatoren:** Vereinigung, Schnittmenge, Differenz
- **Ausdrücke der relationalen Algebra** = „Anfragen“ (queries)

Prädikate (Selektionsprädikate, Join-Prädikate)

- Das Relationenmodell basiert auf mathematischen Grundlagen. Ein wichtiger Begriff, der bisher verwendet wurde, war der Begriff der Menge. **Mengen haben keine zwei gleichen Elemente**!
- Prädikat: Berechenbarer Ausdruck, der als Ergebnis entweder wahr oder falsch liefert. (boolisch)

Entfernend: Selektion, σ (kleines Sigma)

- Unärer Operator (d.h. nur ein Operand), klein Sigma ist hier der Operator.
- Erzeugt **neue Relation** mit **gleichem Schema** aber einer **Teilmenge** der **Tupel** des Operanden (Relation).
- Nur Tupel, die der sogenannten **Selektionsbedingung (= Selektionsprädikat)** entsprechen, werden ins Ergebnis übernommen. **Das Ergebnis kann eine «leere» Relation sein.**
- **Selektionsbedingung:** Kombination von logischen Ausdrücken bestehend aus Attributen u./o. Konstanten.
- **Prüft** Selektionsbedingung für **jedes Tupel** der Relation.
- **Schreibweise:** $\sigma_P(R) = \sigma_{Selektionsprädikat}(Name\ der\ Relation)$
- **Beispiel:** Finde alle Zürcher Kunden, die einen Rabatt von 15% oder mehr erhalten. → $\sigma_{City\ =\ \text{Zürich}\ \wedge\ Discount\ \geq\ 0.15}\ Customers$
  Die neue Relation sieht damit so aus: →

Customers

CNo	Name	City	Balance	Discount
1	'Legrand'	'Genève'	0.00	0.10
2	'Studer'	'Zürich'	–800.00	0.20
3	'Huber'	'Zürich'	–20.00	0.05

$\sigma_{City\ =\ \text{Zürich}\ \wedge\ Discount\ \geq\ 0.15}\ Customers$

CNo	Name	City	Balance	Discount
2	'Studer'	'Zürich'	–800.00	0.20

- $R' = \sigma_{Selektionsbedingung}(R)$, Name der Relation → Name' neuer Name
- Selektionsbedingung: auswertbarer logischer Ausdruck (Prädikat)
- Resultatrelation R' hat gleiches Schema wie R (ist aber eine neue Relation).
- Für die Selektion gilt: $\sigma_{\Phi_1}\left(\sigma_{\Phi_2}(r)\right) = \sigma_{\Phi_2}\left(\sigma_{\Phi_1}(r)\right)$, Φ_n = logischer Ausdruck. Reihenfolge spielt keine Rolle!
- Wenn **Selektionsbedingung** für **alle** Tupel **falsch** ist, führt zu **leeren Relation** (d.h. **nur einem Schema**).

Entfernend: Projektion π (kleines Pi)

- Unärer Operator (d.h. nur ein Operand). Erzeugt neue Relation mit einer **Teilmenge** der ursprünglichen **Attribute** $\pi_{A_1,A_2,\dots,A_k}(R)$ ist eine Relation mit den Attributen $A_1, A_2, \dots, A_k$
- **Achtung:** Es können **Duplikate** entstehen, die **entfernt** werden **müssen**. **Überprüfung der Eindeutigkeit.**
- Die Projektion wählt eine Menge von Spalten aus (und entfernt die Übrigen). Die Projektionsliste darf auch Konstanten, Berechnungen oder Funktionsaufrufe umfassen.
- Die Projektionsliste muss auswertbar sein, sonst **Fehler**.
- **Schreibweise:** $\pi_L(R) = \pi_{Attributliste}(Name\ der\ Relation)$
- **Beispiel:** Neue Relation mit Attributen «Titel», «Jahr» und «Länge» sowie eine zweite mit Attribut «inFarbe».
  Achtung: Bei der Selektion von Attribut «inFarbe» kommt der Attributwert «True» doppelt vor, weshalb einer entfernt werden muss!

Filme

Titel	Jahr	Länge	inFarbe	Studio	ProduzentID
Total Recall	1990	113	True	Fox	12345
Basic Instinct	1992	127	True	Disney	67890
Dead Man	1995	121	False	Paramount	99999

$\pi_{Titel,Jahr,Länge}$(Filme)

Titel	Jahr	Länge
Total Recall	1990	113
Basic Instinct	1992	127
Dead Man	1995	121

$\pi_{inFarbe}$(Filme)

inFarbe
True
False

© F.Naumann, 2011

- Resultatrelation R' hat Schema gleich **Attributliste**
- Resultattupel sind alle Tupel von R, aber nur die Attribute aus Attributliste und **ohne Duplikate**, ev. ergänzt um berechnete Attribute.
- **Achtung:** Relationale Datenbankverwaltungsysteme eliminieren mehrfach vorkommende Tupel bei der Projektion **nicht automatisch**. Man spricht dann (präziser) nicht von Relationen sondern von (relationalen) **Bags (= Multimengen).**
- **Achtung:** Bei Projektion gilt jedoch $\pi_A\left(\pi_B(r)\right) \neq \pi_B\left(\pi_A(r)\right)$
- **Spezialfall:** Eine **Projektion** auf **nicht vorhandene Attribute**, führt zu einem **Fehler**.

Umbenennung: ρ (kleines Rho)

- **Attributnamen** müssen innerhalb eines Relationenformates **eindeutig** sein. Bei mehreren gleich benannten Attributen in verschiedenen Relationenformaten wird der Name zusätzlich durch die Bezeichnung des Relationenformates qualifiziert: Gegeben: R(A,B,C) und S(A,B,C) dann wird z.B. A aus R und A aus S wie folgt qualifiziert: R.A bzw. S.A
- Bei mehreren gleichlautenden Relationenformaten (siehe z.B. Auto-Join) wird der sogenannte Umbenennungsoperator ρ eingesetzt:
- **Schreibweise:** $\rho_L(R) = \rho_{neuer\ Name}(Name\ der\ Relation)$, auch gebräuchlich: R AS neuerName
- **Beispiel:** $\rho_{S(D,E)}\left(R(A,B)\right)$ → Umbenennung R.A → S.D, R.B → S.E), Alternative $\pi_{A\rightarrow D,B\rightarrow E}$

Kombinierend: (kartesisches) Produkt, x (kleines x)

- Kreuzprodukt zweier Relationen R und S ist die **Menge aller Tupel**, die man erhält, wenn man **jedes Tupel aus** R **mit jedem Tupel** aus S «**kombiniert**». R und S sind die Operanden (Binär) und x der Operator.
- Schema hat ein Attribut für jedes Attribut aus R und S.
- Bei **Namensgleichheit** wird **kein Attribut weggelassen**, stattdessen: **Umbenennen** oder **qualifizieren**.
- **Schreibweise:** $R' = R\ x\ S$
- **Beispiel:**

R

A	B
1	2
3	4

S

B	C	D
2	5	6
4	7	8
9	10	11

R × S

A	R.B	S.B	C	D
1	2	2	5	6
1	2	4	7	8
1	2	9	10	11
3	4	2	5	6
3	4	4	7	8
3	4	9	10	11

- Resultatrelation R' hat Schema von R **konkateniert** mit dem Schema von S (bei doppelten Attributnamen Umbenennung oder Qualifizierung nötig).
- **Resultattupel** sind alle Tupel von R **kombiniert** mit allen Tupeln von S.
- Kreuzprodukt ist (in SQL) **nicht kommutativ**: $R\ x\ S\ \neq S\ x\ R$ (Schemas sind unterschiedlich, das eine kann aber durch Projektion in das andere überführt werden).

Kombinierend: Verbund, natural Join, ⋈ (x mit je einem Strich an der Seite aka bowtie)

- Motivation: Statt im Kreuzprodukt alle Paare zu bilden, sollen nur die Tupelpaare gebildet werden, deren Tupel irgendwie zusammenpassen (Joinbedingung). R, S sind die Operanden (Binär) und ⋈ der Operator
- Idee, wie Tupel kombiniert werden: «natural» join: **Übereinstimmung** der **Attributwerte** in allen **gemeinsamen Attributen** (d.h. **gleicher Name**, **gleiche Domäne** und **gleiche Bedeutung**)
- Gegebenenfalls Umbenennung nötig.
- Schema: Vereinigungsmenge der beiden Attributmengen:
- **Schreibweise:** $R' = R \bowtie S$
- **Beispiele:**

R

A	B
1	2
3	4

S

B	C	D
2	5	6
4	7	8
9	10	11

R ⋈ S

A	B	C	D
1	2	5	6
3	4	7	8

R

A	B	C
1	2	3
6	7	8
9	7	8

S

B	C	D
2	5	6
2	3	5
7	8	10

R ⋈ S

A	B	C	D
1	2	3	5
6	7	8	10
9	7	8	10

- Resultatrelation R' hat Attribute, die nur in R vorkommen, gefolgt von gemeinsamen Attributen von R und S gefolgt von Attributen, die nur in S vorkommen.
- Join ist (in SQL) **nicht kommutativ**: $R \bowtie S \neq S \bowtie R$)
- **Es gelten folgende Spezialfälle:**
  - $R \bowtie R = R$
  - Ein natural join von zwei Relationen, die **keine gemeinsamen Attribute** haben führt zu einem **Kreuzprodukt. Bsp.:** $\pi_A(r) \bowtie \pi_B(r) \bowtie \pi_C(r)$, werden alle kombinatorischen Möglichkeiten produziert! →

r

A	B	C
0	0	0
0	0	1
1	0	0
1	0	1
1	1	0

A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Drei schwerwiegende Nachteile des Natural join

- **Voraussetzung:** Gemeinsame Attribute (d.h. gleiche Bezeichnung, gleiche Domäne, gleiche Bedeutung). Das ist bei realen Datenbanken oft nicht gegeben.
- **Join-Kriterium:** Übereinstimmung (Gleichheit) **in allen** Attribut**werten** der gemeinsamen Attribute.
- Nur ein Test auf **Gleichheit** möglich.

→ Natural Join spielt in Praxis deshalb **untergeordnete Rolle**. Stattdessen verwendet man den «**Theta-Join**».

Beispielaufgabe Relationales Modell, relationale Algebra

Wandeln Sie untenstehende Tabelle in eine Relation um mit demselben «Informationsgehalt»:

Pers.-Nr.	Name	Hobbies
1	Meier	Tanzen, Joggen, Velofahren
2	Müller	Fischen
3	Huber	Motorradfahren, Wandern

Pers.-Nr.	Name	Hobbies
1	Meier	Tanzen,
1	Meier	Joggen
1	Meier	Velofahren
2	Müller	Fischen
3	Huber	Motorradfahren
3	Huber	Wandern

Vorgehen beim Lösen von Aufgaben mit relationaler Algebra:

1. Ist der Ausdruck valid?
2. Schema von Lösungsrelation hinschreiben
3. Ausdruck auswerten
4. Deduplikation / Dubletten entfernen

Kombinierend: Theta-join, $\bowtie_P$

- **Verknüpfungsbedingung** kann **frei** gestaltet werden (Join-Prädikat).
- Konstruktion des Ergebnisses:
  - Bilde Kreuzprodukt
  - Selektiere mittels der Joinbedingung
  - Also: $R \bowtie_P S = \sigma_P(R \times S)$
  - P: Join-Prädikat (Joinbedingung): Beliebiger logischer Ausdruck, der auch sog. θ-Vergleiche enthalten darf, $\theta \in \{=,<,>,\leq,\geq,\neq\}$
- **Schema:** Wie beim Kreuzprodukt (also ggf. Umbenennung nötig).
- Natural Join ist **Spezialfall** von Theta-Joins.
- **Beispiel:**
- **Allgemeiner Verbund** (Theta join):
  - Gegeben: $R(A_1, \dots, A_n)$ und $S(B_1, \dots, B_m)$
  - $R \bowtie_P S = \sigma_P(R \times S)$, (kartesisches Produkt mit Selektion)
  - Das Format der neuen Relation hat nun $n + m$ Attribute

R

A	B	C
1	2	3
6	7	8
9	7	8

S

B	C	D
2	5	6
2	3	5
7	8	10

$R \bowtie_{A<D} S$

A	R.B	R.C	S.B	S.C	D
1	2	3	2	5	6
1	2	3	2	3	5
1	2	3	7	8	10
6	7	8	7	8	10
9	7	8	7	8	10

$R \bowtie_{A<D \text{ AND } R.B \neq S.B} S$

A	R.B	R.C	S.B	S.C	D
1	2	3	7	8	10

Verbundvarianten: Sonstige

- **Equi-Join:** Prädikat P prüft **nur** auf **Gleichheit** und die Attribute sind mit logischem „und" verknüpft.
- Jeder Natural Join ist ein Equi-Join, aber nicht jeder Equi-Join ist ein Natural Join. Warum?
- **Auto-Join:** $R_1 \bowtie R_1$ (benötigt Umbenennung)
- **Cross-Join:** $R_1 \times R_1$ (ist kartesisches Produkt)
- **Semi-Join:** $\pi_{r_1,\dots,r_n}(R_1 \bowtie R_2)$– Natural Join mit Projektion auf die Attribute von R_1 oder R_2
- **Outer-Joins:** Siehe später

Mengenoperator: Vereinigung, ∪

- Sammelt Elemente (Tupel) zweier Relationen unter einem gemeinsamen Schema auf.
- **Attributmengen/Schema** beider Relationen müssen **gleich** sein: Namen, Typen, (Zur Not: Umbenennung)
- Ein Element ist nur einmal in $R \cup S$ vertreten, auch wenn es je in R und S auftaucht: **Duplikatentfernung.**
- **Schreibweise:** $R' = R \cup S$ → R, S sind die Operanden (Binär) und ∪ der Operator.
- Resultatrelation R' hat gleiches Schema wie R bzw. S
- Resultattupel sind **diejenigen Tupel** die in R **ODER** in S vorkommen (ohne Duplikate)
- Vereinigung ist **kommutativ**: $R \cup S = S \cup R$

Mengenoperator: Durchschnitt ∩

- Durchschnitt $R \cap S$ ergibt die **Tupel**, die in **beiden Relationen gemeinsam vorkommen**.
- **Attributmengen/Schema** beider Relationen müssen **gleich** sein: Namen, Typen, (Zur Not: Umbenennung)
- Ein Element ist nur einmal in $R \cap S$ vertreten, auch wenn es je in R und S auftaucht: **Duplikatentfernung.**
- **Schreibweise:** $R' = R \cap S$ → R, S sind die Operanden (Binär) und ∩ der Operator.
- Resultatrelation R' hat gleiches Schema wie R bzw. S
- Resultattupel sind **diejenigen Tupel** die in R **UND** in S gemeinsam vorkommen.
- Durchschnitt ist **kommutativ**: $R \cap S = S \cap R$

Mengenoperator: Differenz, \ oder -

- Differenz $R \setminus S$ oder $R - S$ eliminiert die Tupel aus ersten Relation, die auch in zweiten Relation vorkommen.
- **Attributmengen/Schema** beider Relationen müssen **gleich** sein: Namen, Typen, (Zur Not: Umbenennung)
- **Schreibweise:** $R' = R \setminus S$ → R, S sind die Operanden (Binär) und \ der Operator.
- Resultatrelation R' hat gleiches Schema wie R bzw. S
- Resultattupel sind **diejenigen Tupel** die in R **ABER NICHT** in S vorkommen.
- Vereinigung ist **nicht kommutativ, Achtung:** $R \setminus S \neq S \setminus R$, R ohne S oder S ohne R ist nicht das gleiche!

Beispielaufgabe für Prüfung

Gegeben sei die Relation r mit den Attributen Personenname und der entsprechenden Sockenfarbe. Aufgabe: Filtern, nach Personen, die blaue UND grüne Socken haben:

$r' = \pi_{Name}(\sigma_{Farbe="Blau"}(r)) \cap \pi_{Name}(\sigma_{Farbe="Grün"}(r))$ →

r'	Name
	Jo

r	Name	Farbe
	Jo	Blau
	Jo	Grün
	Franz	Blau
	Hans	Gelb
	Hans	Grün

«bag»-Algebra

- Relationale Algebra baut auf Mengen auf. SQL ist aber «bag»-Sprache (d.h. arbeitet mit sog. **Multimengen**)
- Relationale Algebra lässt sich auf Multimengen erweitern. Man kann dann **nicht unterscheiden** zwischen **identischen Elementen**. Es ist **aber unterscheidbar**, **wie oft** ein **Bag** ein **Element enthält**.
- Die **Anzahl Vorkommen** eines Tupels in einem bag wird «**Multiplizität**» genannt.
- Für die Berechnungen der bag-Algebra gelten folgende Regeln:

Operation, Symbol	
Selektion, $\boldsymbol{\sigma}$	Gleich wie relationale Algebra, Duplikate behandeln wie «normale» Tupel.
Projektion, $\boldsymbol{\pi}$	Gleich wie relationale Algebra, aber Duplikate **nicht entfernen**.
Kreuzprodukt, **x**	Gleich wie relationale Algebra, Duplikate behandeln wie «normale» Tupel.
Joins, ⋈	Gleich wie relationale Algebra, Duplikate behandeln wie «normale» Tupel.
Vereinigung, ∪ «bag union»	Gleich wie relationale Algebra. Duplikate: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt davon die **grössere Anzahl**.
Vereinigung, ⊔ «bag concatenation»	Auch hier werden Tupel aus linken und rechten Operanden zusammengeführt. Duplikaten: Zählt im linken und rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt die **Summe** davon.
Durchschnitt, ∩	Gleich wie relationale Algebra. Duplikate: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten») und nimmt davon die **kleinere Anzahl**.
Differenz, \	Gleich wie relationale Algebra. Duplikate: Man zählt im linken und im rechten Operanden die Duplikate (= «Multiplizitäten»). Dann rechnet man die **Differenz** aus zwischen linker Multiplizität und rechter **Multiplizität**. Wenn diese positiv ist (links hat es mehr als rechts) dann nimmt man diese **Differenz**, sonst **0** als Multiplizität.
Duplikatelimination, $\boldsymbol{\delta}$	**Entfernt Duplikate:** Mit δ (Delta) kann man aus einem bag wieder eine Relation machen.

Outer Joins – Left outer join ⟕, right outer join ⟖ und full outer join ⟗

- Bei outer joins werden jeweils alle Tupel des linken (oder rechten, oder von beiden) Operatoren ins Resultat übernommen. Wenn es einen passenden «Join-Partner» gibt, so wird normal gejoint, ansonsten werden die fehlenden Werte durch den Platzhalter «NULL» ersetzt.
- NULL ist selbst kein Wert sondern ist ein Indikator für fehlende Werte!
- **Achtung:** NULLs verursachen oft Probleme und sind nach Möglichkeit zu vermeiden.

L ⟕ R = Resultat

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟕

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	NULL	NULL

L ⟖ R = Resultat

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟖

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
NULL	NULL	c_3	d_2	e_2

L ⟗ R = Resultat

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

⟗

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

=

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	NULL	NULL
NULL	NULL	c_3	d_2	e_2

Erweiterungen der Projektion

- Bisher eingeführt: $\pi_L(R)$
- In der klassischen relationalen Algebra gilt, dass L eine **Teilmenge** der Attribute von R sein muss.
- Für praktische Belange kann man die Projektion erweitern: $\pi_L(R)$ L kann dann folgendes sein:
  - Eine Teilmenge der Attribute von R (wie bisher).
  - Eine **Menge** von **Ausdrücken** der Art $E \rightarrow \mathrm{x}$, wobei E ein Ausdruck bestehend aus **Attributen** von R, **Konstanten**, **arithmetischen Operatoren**, **Zeichenkettenoperatoren**, … oder auch **Funktionsaufrufen** sein kann. Das Ergebnis von E wird dann im **Resultatschema** x genannt.
- **Beispiel:**

R	A	B	C
	1	2	3
	4	5	6

$\pi_{3 \cdot A \rightarrow \mathrm{X}, B+C \rightarrow D, C}(R)$ →

R'	X	D	C
	3	5	3
	12	11	6

$\pi_{SQR(A) \rightarrow \mathrm{X}}(R)$ →

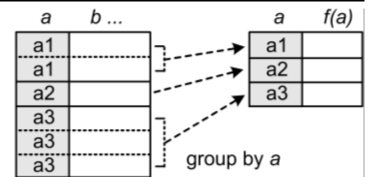
R'	X
	1
	16

Aggregieren, Aggregat-Funktionen

- Alle bisherigen Operationen behandelten nur **einzelne** Tupel. Diese sind ungeordnet in der Menge (Relation) enthalten und «wissen» nichts voneinander. Viele Aufgaben in DB müssen **Daten zusammentragen**.
- **Beispiele:**
  - Wie gross ist die Summe aller Verkäufe? → Aggregation über alle Tupel
  - Wieviel hat welcher Verkäufer verkauft? → Aggregation pro Verkäufer (Gruppierung).
- **Typische Aggregatfunktionen:** Summe, Durchschnitt, Minimum, Maximum, Anzahl Tupel.

Gruppierung

- Jede Gruppe ist eine Multimenge von Tupeln mit denselben Werten für das Gruppierungskriterium (Attributkombination)
- Zusammenfassung nach gleichen Werten für Attribut a:

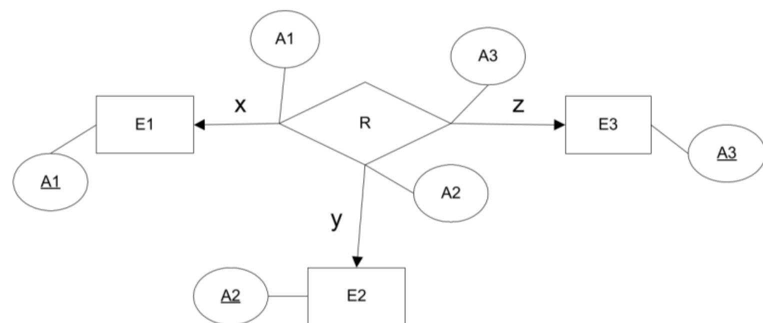


ODER/OR: ∨
UND/AND: ∧

Für jeden Pfeil mit einer Kardinalität 1 braucht es einen Schlüssel. Dieser setzt sich zusammen aus allen Primärschlüsselattributen aller an den anderen Pfeilen hängenden Entitätstypen (unabhängig was dort für eine Kardinalität steht).

Sonderfall „alles M-Kardinalität": Es braucht einen Schlüssel, um Duplikate zu vermeiden. Dieser Schlüssel setzt sich aus allen Primärschlüsselattributen der an den Pfeilen hängenden Entitätstypen zusammen.

Gegeben ist das folgende Diagramm mit unbekannten Markierungen x, y und z an den Pfeilen.



Die Markierungen müssen natürlich $x, y, z \in \{1, m\}$ sein. Man gebe für jede der acht Möglichkeiten der Wahl dieser Markierungen an, was für Schlüsselbedingungen in R dadurch impliziert werden.

Lösung:

x	y	z	Schlüssel
m	m	m	{A1, A2, A3}
m	m	1	{A1, A2}
m	1	m	{A1, A3}
m	1	1	{A1, A2} und {A1, A3}
1	m	m	{A2, A3}
1	m	1	{A2, A3} und {A1, A2}
1	1	m	{A2, A3} und {A1, A3}
1	1	1	{A2, A3} und {A1, A3} und {A1, A2}

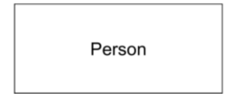
Entity-Relationship-Design

Entity-Relationship-Diagramme

- Entity Relationship (ER) → Diagramm-Sprache für das Design von Datenstrukturen
- Darstellung eines ER-Dialektes, der sich für das Design von relationalen Datenstrukturen eignet.
- Liefert Relationen in IDNF (Inclusion Dependency Normal Form), vermeidet NULLs, lässt nur Semantik zu, die von DBMS überwacht werden kann

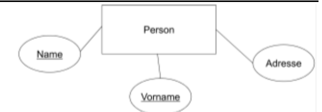
Entität, Entitätstyp

- Entitätstyp als „undefined notion" (siehe auch Punkt in Geometrie, Punkt ist dort als unendlich klein definiert)
- **Darstellung** durch **Rechteck** mit **eindeutigem Namen** →



- Konkrete Entitätstypen sollen **möglichst genau definiert** werden.
- Ein Entitätstyp steht für Mengen von Entitäten. Analogie in Java: Klassen/Objekte
- Ein **Entitätstyp** wird in eine **Relation abgebildet** werden, also eine **Tabelle** mit Schlüssel(n), und die **Entitäten** werden die **Zeilen** der **Tabelle** sein.

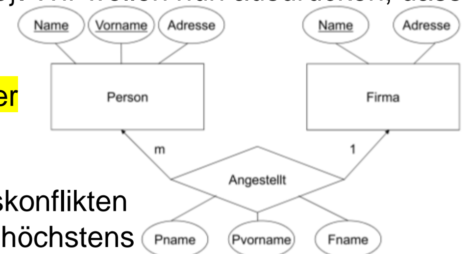
Attribut, Attributwert (attribute, attribute value)



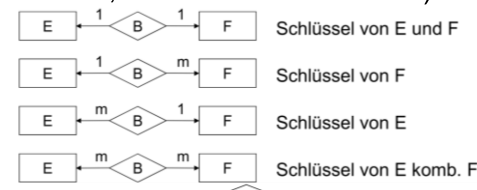
- **Entitätstypen** haben **Attribute**
- **Entitäten** haben **Attributwerte**
- Bsp. "Meier" könnte z.B. Attributwert des Attributs "Name" einer Entität des Typs "Person" sein
- Wir stellen **Attribute** als **Ovale** dar
- Die **Attribute** sind mit **Linien** mit dem zugehörigen **Entitätstyp verbunden**
- Lassen grundsätzlich nur "**einfache**" Attribute zu. D.h., hinter "Name" steckt ein Wert, nicht Liste von Werten
- Die **Unterstreichung** zeigt, dass die entsprechenden **Attribute** als **Primärschlüssel** gewählt wurden
- Wählen i.A. erst dann einen Primärschlüssel, wenn Entitätstyp referenziert wird (ansonsten nur Schlüssel)

Beziehung, Beziehungstyp

- Gegeben sei ein zweiter Entitätstyp "Firma", mit Primärschlüssel {Name}. Wir wollen nun ausdrücken, dass Personen in Firmen angestellt sein können.



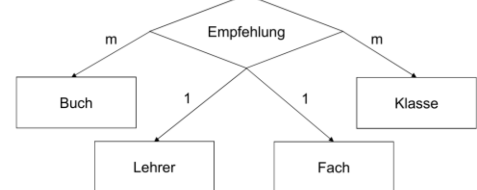
- Stellen **Beziehungstyp** (hier: "Angestellt") durch einen **Rhombus** dar.
- Der **Beziehungstyp** "Angestellt" **erbt** die **Primärschlüsselattribute** der **Entitätstypen** "Person" und "Firma", von denen er **abhängig ist.**
- Er kann auch noch weitere, "**eigene**" **Attribute haben**.
- **Fremdschlüssel** wählt i.d.R. mit gleichem Namen, ausser bei Namenskonflikten
- Die Pfeilmarkierungen (**Kardinalitäten**) drücken aus, dass pro Person höchstens eine Firma als Arbeitgeber existiert, während eine Firma beliebig viele (auch 0) Angestellte haben kann.
- "m" bedeutet also beliebig viele, «unbestimmt» o.ä.
- Will man "Angestellt" als Relation abbilden, muss {Pname, Pvorname} als Schlüssel gewählt werden.
- **Beziehungstypen** haben **Schlüssel**, **keine Primärschlüssel** (Es sei denn, sie werden referenziert)
- Mögliche Kombinationen: (inkl. passendem Schlüssel in B)



- Der **Beziehungstyp** ist **existentiell** abhängig von den **Entitätstypen**, welche er **referenziert**

Beziehungen mit >2 ET

- Dieser Beziehungstyp ist **existentiell abhängig** von vier Entitätstypen (siehe Pfeile)
- **Existentiell unabhängige** Entitätstypen sind **jene**, die **keine ausgehenden Pfeile** haben

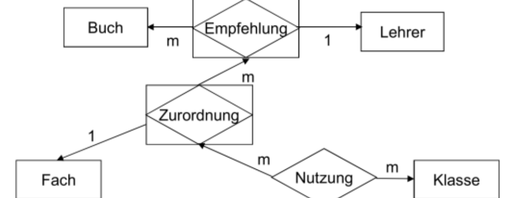


- Zwei Schlüsselbedingungen (aus Lehrer und Fach)
- Eine Entität (Viereck) muss jeweils «fixiert» werden.
- Eine Klasse kann von einem Lehrer für ein Fach mehrere Bücher empfohlen bekommen. Oder: Ein Lehrer kann mehrere Bücher für ein Fach für mehrere Klassen empfehlen.
- Die Markierung "1" beim Pfeil zu "Lehrer" bedeutet: ein Buch wird einer Klasse in einem Fach von höchstens einem Lehrer empfohlen
- Die Markierung "1" beim Pfeil zu "Fach" bedeutet: ein Buch wird einer Klasse von einem Lehrer in höchstens einem Fach empfohlen.
- Zwei Schlüssel (aus Lehrer und Fach): Für Lehrer: {B#, F#, K#}, sowie für das Fach: {B#, L#, K#}. Nehmen somit jeweils alle Elemente von Entität mit 1 bis auf das ausgewählte mit der 1.
- Mehr ist aus den Kardinalitätsbedingungen NICHT herauszulesen
- Man läuft leicht Gefahr, weitere erwünschte Bedingungen in solche Beziehungen zwischen mehr als 2 Entitätstypen "herein zu interpretieren" (z.B. " pro Buch höchstens ein Fach" – dies ist NICHT der Fall!)
- **Vorsicht bei solchen Beziehungstypen**

Schlüsselbedingungen für Beziehungstypen (allgemeiner Fall)

- Es sei $R(E_1, L_1, E_2, L_2, E_3, L_3, \dots, E_n, L_n)$ ein Beziehungstyp
- Dieser hängt ab von den Entitätstypen $E_1, \dots, E_n$, wobei die Pfeile je mit $L_1, \dots, L_n$ markiert seien.
- Mit M_j bezeichnen wir die Menge der Fremdschlüsselattribute von R, welche dem Primärschlüssel des Entitätstypen E_j entspricht. (M_j nicht leer, sowie M_i, M_j paarweise elementfremd)
- $M = M_1 \cup M_2 \cup M_3 \cup \dots \cup M_n$
- Ist für mind. ein j $L_j = 1$, so gilt für jedes dieser j mit $L_j = 1$: Die Menge $M \setminus M_j$ ist ein Schlüssel von R
- Sind alle $L_j = m$, so ist M ein Schlüssel von R (wir wollen Relationen, nicht Bags)
- → Das obige einfach ausgedrückt: Bei allen Pfeilen/Entitäten mit einer 1, wird der Schlüssel so gebildet, dass man pro solcher Entität alle anderen Entitäten/Objekte (ohne den gewählten) zusammen nimmt. Diese bilden den Schlüssel. Für die Entitäten mit Pfeilen ungleich 1, braucht es keine Schlüssel.

Einschätzung Beziehungstypen >2 ET

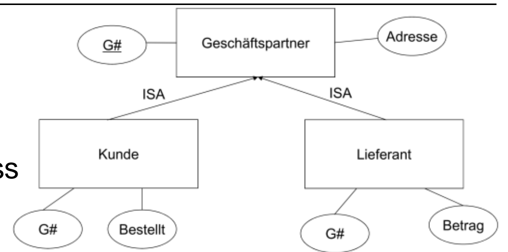


- Die Semantik solcher Beziehungstypen mit >2 ET ist «knifflig».
- Wie Sie gesehen haben, will man oft weitere Beziehungen, die nur einen Teil der ET betreffen.
- Ziehen Sie in Betracht, mehrere binäre Beziehungen zu verwenden!
- Achtung: die Semantik ist NICHT identisch.

Unabhängiger Entitätstyp

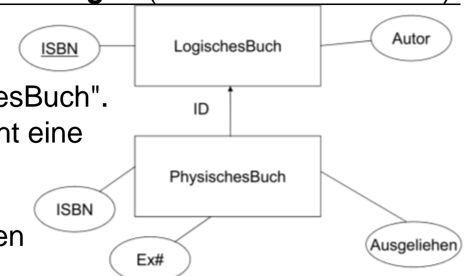
- Wir sprechen bei den Entitätstypen wie bisher kennengelernt von "unabhängigen" Entitätstypen.
- Diese Entitätstypen repräsentieren Entitäten, die für sich selbst "leben" können
- Einträge im Beziehungstyp "Angestellt" können nicht für sich selbst existieren, sie beziehen sich immer auf Personen und Firmen
- Haben **nur eingehende Pfeile**, aber **keine ausgehenden Pfeile**

ISA-abhängiger Entitätstyp (ISA-dependent entity type)



- Jeder Kunde resp. jeder Lieferant ist auch Geschäftspartner
- Die Pfeile zeigen eine existentielle Abhängigkeit ("is a" = "ist ein")
- Die Pfeile führen zu Schlüsselbedingungen
- {G#} ist Schlüssel in Kunde und in Lieferant. Es wird erzwungen, dass jedem Kunden ein Geschäftspartner entspricht – der Kunde selbst!)
- Ein Geschäftspartner kann Kunde **UND** Lieferant sein.
- Eliminiere ich einen Eintrag beim Geschäftspartner, muss ich auch den entsprechenden Eintrag beim Kunden respektive Lieferanten eliminieren.
- "ISA" ist ein **Generalisierungs**-/**Spezialisierungsmuster**.
- Kunde und Lieferant werden zu Geschäftspartnern **verallgemeinert**
- Kunde und Lieferant sind **Spezialisierungen** von Geschäftspartner
- Eine **Generalisierung** ist **interessant**, wenn:
  - Die **einzelnen Spezialisierungen** sich **deutlich** voneinander **unterscheiden** (Kunde hat andere Attribute als Lieferant, oder hängt mit anderen Beziehungstypen zusammen)
  - Generalisierung "Geschäftspartner" kann **gemeinsame Strukturen auffangen** (hier Attribut "Adresse")

ID-abhängiger Entitätstyp (ID-dependent entity type)



- Eine Bibliothek hat für beliebte Titel mehrere Exemplare
- Entitätstyp "PhysischesBuch" ist ID-abhängig vom Entitätstyp "LogischesBuch".
- Einem logischen Begriff eines Buchs (Primärschlüssel {ISBN}) entspricht eine Menge von physikalischen Büchern
- "Innerhalb" eines logischen Buchs brauchen wir ein weiteres Attribut, z.B. "Ex#", um die einzelnen physikalischen Exemplare zu unterscheiden
- Es bilden also {ISBN, Ex#} einen Schlüssel
- Eine Entität des ID-abhängigen Typs kann "auf natürliche Weise" nur innerhalb der Hierarchie identifiziert werden (Bsp. Kind durch Vornamen innerhalb der Familie)

ID vs. ISA-abhängiger Entitätstyp

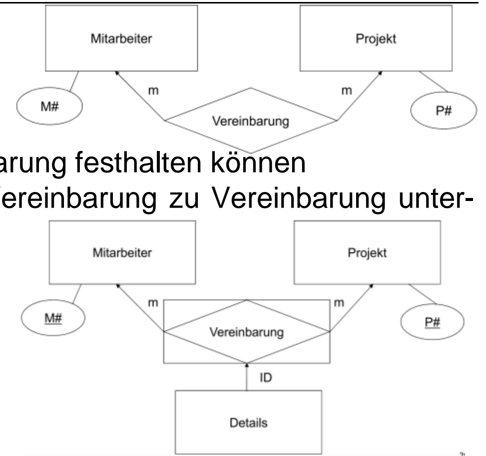
ISA-abhängig:

- Genauere Spezifikation (**Spezialisierung**)
- Ist Entitätstyp F von Entitätstyp E ISA-abhängig, so gilt: Ist M die Menge der Primärschlüsselattribute in E, so muss M in F ein Schlüssel sein.
  → Schlüssel der Superklasse werden von Subklassen geerbt!
- 1:1 Abhängigkeit!

ID-abhängig 1:m Abhängigkeit

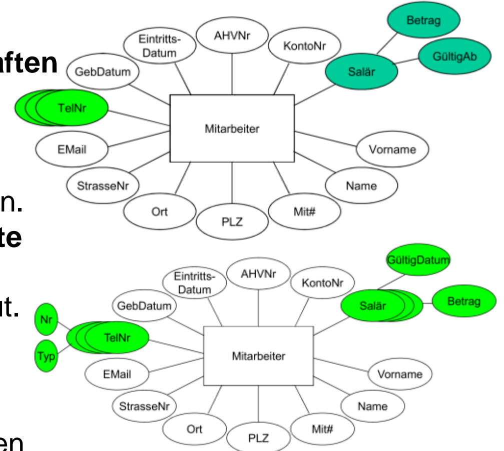
- Hierarchie, hängt an "**Oberklasse**". Ist nur innerhalb der Hierarchie definiert
- Ist Entitätstyp F von Entitätstyp E ID-abhängig, so gilt: Ist M die Menge der Primärschlüsselattribute in E, so muss $M \cup N$ ein Schlüssel in F sein, wobei N eine zu M elementfremde Menge von Attributen von F ist (min. 1 Element)

Zusammengesetzter Entitätstyp (composite entity type)



- Wir behandeln einen Sonderfall, der auftritt, wenn wir an Beziehungstypen Entitätstypen anhängen wollen.
- Ausgangslage:
- Wir wollen unterschiedlich viele (auch keine) Details zu jeder Vereinbarung festhalten können
- Wir hängen einen ID-abhängigen Entitätstyp "Details" an, der von Vereinbarung zu Vereinbarung unterschiedlich viele Details aufnehmen soll.
- Der Beziehungstyp wird "**umgewandelt**" in einen **zusammengesetzten Entitätstyp** (zeichnen ein Rechteck um den Beziehungstyp)
- Es sei {M#,P#} **Primärschlüssel** in Vereinbarung.
- Es ist mindestens ein weiteres Attribut nötig. So kann z.B. {M#, P#, D#} ein Schlüssel sein (wegen ID-Abhängigkeit)

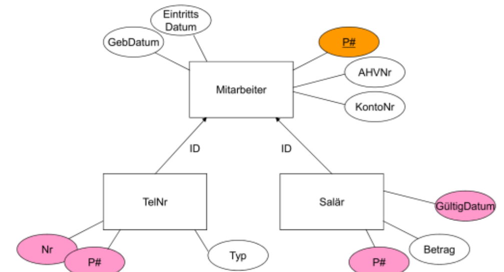
Erweiterungsmöglichkeiten und Spezialfälle



- Wollen bei einem **Attribut** von einem Entitätstyp **weitere Eigenschaften vermerken** oder mehrere «gleiche» Eigenschaften hinterlegen.
- Haben den Entitätstyp «Mitarbeitende» und wollen beim Attribut Salär vermerken, ab wann es gültig ist.
- Zudem wollen wir mehrere Telefon-Nr. für einen Mitarbeiter verwalten.
- **Lösung: zusammengesetzte Attribute** resp. **mehrwertige Attribute**
- Wir möchten nun noch dazu die Salärgeschichte festhalten.
  → Wir benötigen ein «**mehrwertiges, zusammengesetztes**» Attribut.
- Wir wollen den jeweiligen Typ der Telefonnummer vermerken

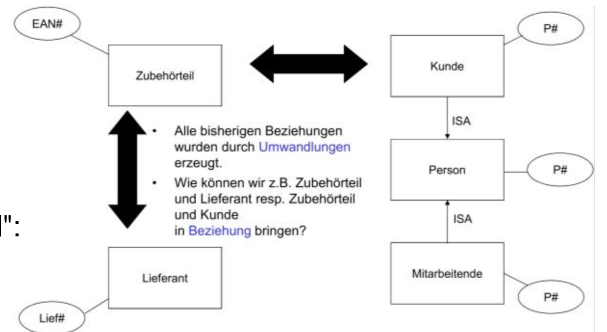
Umwandeln komplexer Attribute

Komplexe Attribute führen zu ID-abhängigen Entitätstypen. Konsequenzen



- Der **Entitätstyp «Mitarbeitende»** bekommt einen **Primärschlüssel**
- Es sind jeweils mehrere Entitäten "TelNr" und "Salär" pro Mitarbeiter/in möglich. Daher reicht {P#} nicht als Schlüssel. Wir **brauchen** jeweils ein/**mehrere differenzierende**(s) **Attribut**(e)
- **Schlüssel** in TelNr: **{P#, TelNr}**
- **Schlüssel** in Salär: **{P#, GültigDatum}**
- Mitarbeitende **können** auch **kein Telefon** haben.
- Eine **Telefonnummer** kann aber **nicht ohne Mitarbeiter**/in "leben", d.h., TelNr ist **existenzabhängig** von «**Mitarbeitende**»

Beziehungen



- Wir modellieren folgende Beziehungen
  - Kunde kauft Zubehörteil
  - Lieferant hat Zubehörteil geliefert
  - Lieferant hat Zubehörteil im Sortiment (aber ev. (noch) nicht geliefert)
- Weitere Überlegungen zum Thema "Kunde kauft Zubehörteil":
  - wann
  - welche
  - wieviele?
  - (Achtung: Zubehörteil ist Produkt, nicht Exemplar!)
- Ein Kauf wird zur "Bestellung" (Warenkorb) mit mehreren Bestellpositionen (mehrere Zubehörteilen im Warenkorb)

Konsequenzen daraus

- Führen zwei neue Entitätstypen "Bestellung" und "Bestellposition" ein
- Führen zwei Beziehungstypen "BestPosEnthält" und "KaufHistorie" ein

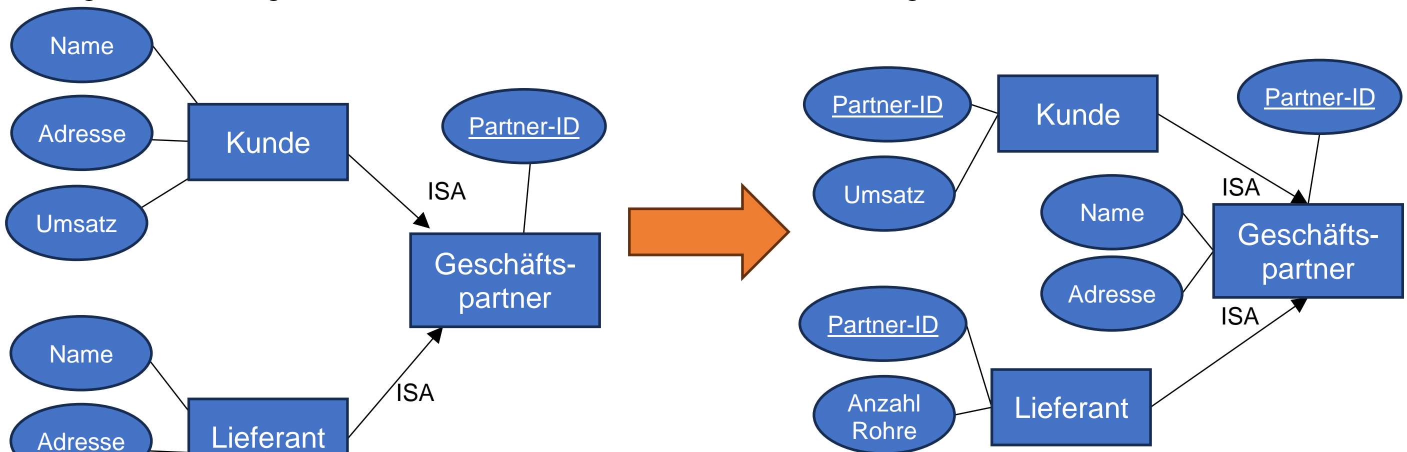
- **Beziehungstypen** werden **dann** verwendet, wenn eine Beziehung dargestellt werden soll, aber **keine existentielle Abhängigkeit** zwischen den Entitätstypen **besteht**.
- Pro Bestellposition gibt es höchstens eine Zubehörteil. Ein Zubehörteil kann in mehreren Bestellpositionen vorkommen
- Mehrere Bestellpositionen pro Bestellung = möglich (ID-Beziehung: Position ohne Bestellung = unbestimmt)
- "KaufHistorie" könnte auch als ID-Beziehungstyp dargestellt werden (Bestellung muss Kunden gehören)

Beispielprüfungsaufgabe

- Entwerfen Sie gemäss untenstehenden Angaben ein vollständiges (und korrektes!) ER-Diagramm für einen Handelsbetrieb. Der Betrieb ist als Zwischenhändler im Baugewerbe tätig für den Handel mit Stahlrohren, d.h. er bezieht Rohre von Lieferanten und verkauft diese an Kunden weiter. Im konzeptionellen Entwurf sollen folgende Rahmenbedingungen abgebildet werden:
- Eingekauft und verkauft werden ausschliesslich Stahlrohre. Über alle diese Rohre werden folgende Angaben festgehalten:
  - 7-stellige RohrID (entspricht einer eindeutigen Artikel-Nr.)
  - Lagermenge
  - Bezeichnung
- Einzelne (physische) Rohre desselben Typs werden nicht unterschieden.
- Rohre werden auf grossen Regalen gelagert. Eine einzelne Lagerposition wird dabei eindeutig mit einer Regalnummer sowie einer Fachnummer identifiziert. In einem Fach werden nur Rohre eines Typs gelagert.
- Die Rohre werden von verschiedenen Lieferanten geliefert (aus Qualitätssicherungsgründen darf aber ein einzelner Rohrtyp nur bei genau einem Lieferanten bezogen werden). Bei den Lieferanten interessiert der Name, die Adresse und wieviel bisher (seit Bestehen der Lieferantenbeziehung) beim entsprechenden Lieferanten eingekauft wurde (Anzahl Rohre total). Bei Kunden interessiert ebenfalls der Name und die Adresse sowie der bisher (seit Bestehen der Kundenbeziehung) getätigte Umsatz. Adressen werden der Einfachheit halber hier als ein einziges Attribut modelliert.
- Kunden und Lieferanten werden betriebsintern „Geschäftspartner" genannt und haben eine eindeutige PartnerID. Kunden sind nie Lieferanten und umgekehrt (da aus völlig verschiedenen Branchen stammend).
- Bei Einkäufen (Bestellungen beim Lieferanten) bzw. Verkäufen (Bestellungen von Kunden) muss das Liefer- bzw. Bestelldatum sowie die Menge in Stück festgehalten werden.
- Nebst ER-Diagramm sind die Schlüssel der Beziehungstypen explizit auf Diagramm textuell festzuhalten.

Vorgehen/Ablauf

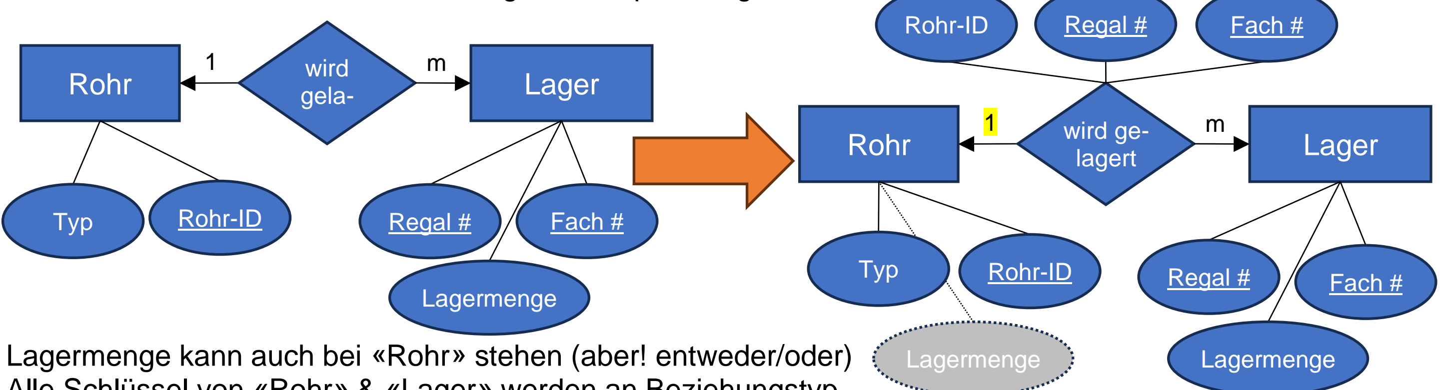
1. Benötigte **Entitäten** aufschreiben: **Geschäftspartner, Kunde, Lieferant, Rohr, Bestellungen und Lager.**
2. Irgendwo anfangen + Modell «einfach/rudimentär» aufstellen. Beginn mit Entität Kunde/Lieferant + Attribut



Im zweiten Schritt werden die identischen Attribute von «Kunde» und «Lieferant» unter «Geschäftspartner» zusammengefasst → Der/die Schlüssel von «Geschäftspartner» («Partner-ID») wird/werden an die Entitäten K/L vererbt!

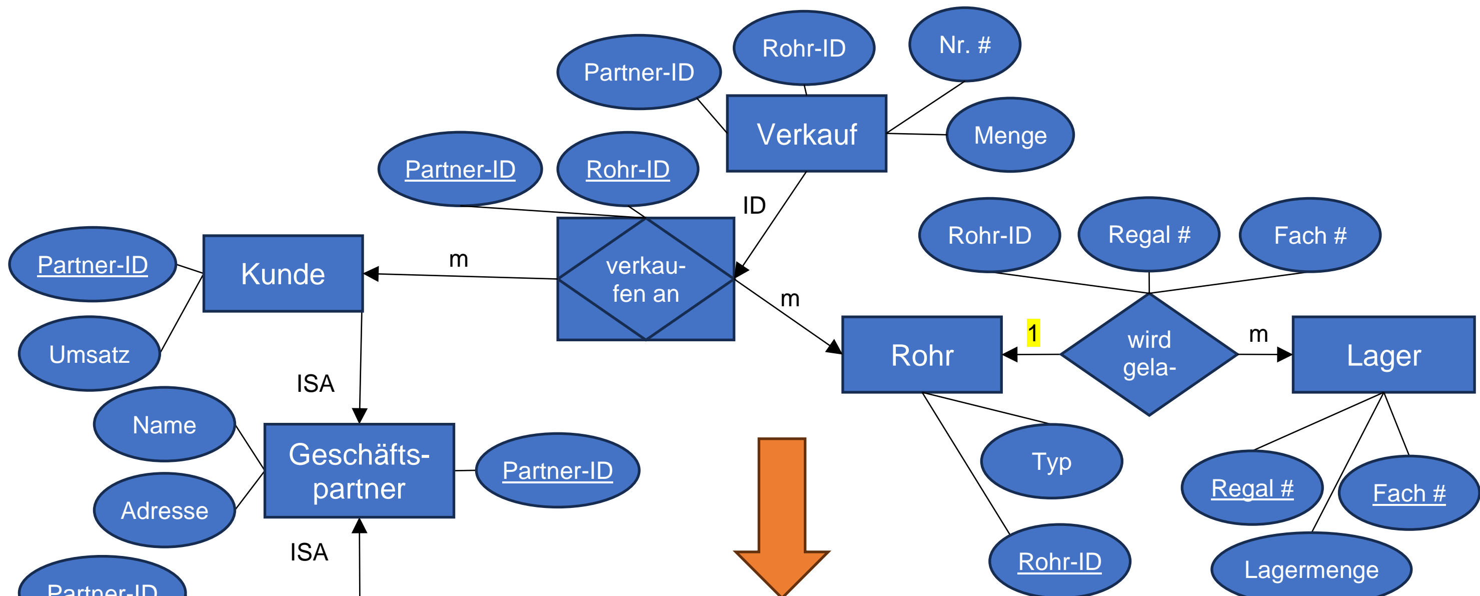
Hinweis: gemäss Aufgabenstellung darf «Kunde» nicht gleichzeitig Lieferant sein & vice versa. Dies muss explizit geschrieben werden, es wird/kann durch dieses Modell nicht abgedeckt werden (graphisch/zeichnerisch).

3. Nun möchten wir «Rohr» und «Lager» ins Spiel bringen:



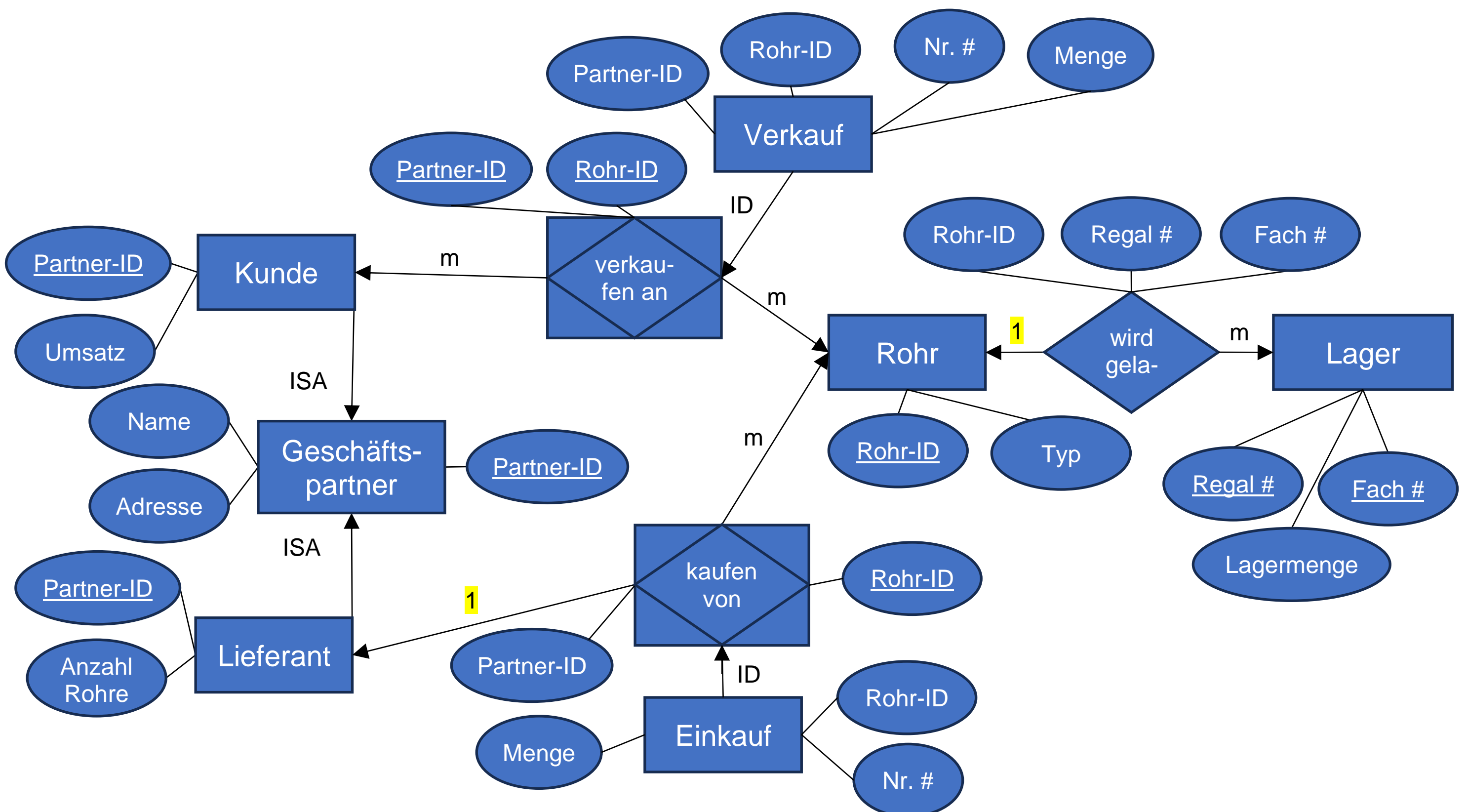
Lagermenge kann auch bei «Rohr» stehen (aber! entweder/oder)
Alle Schlüssel von «Rohr» & «Lager» werden an Beziehungstyp «wird gelagert» vererbt, dabei gilt andere Seite als wo «1» steht, diese Schlüssel werden dann unterstrichen.

4. Nun beide «Modelle» mit «Bestellung» verknüpfen. Zuerst Vorgang Produkte an Kunde Verkaufen



Da wir für die Beziehung zwischen Rohr und Kunde (als der Verkaufsvorgang) noch weitere Attribut benötigen, machen wir dies via zusammengesetztem Entitätstyp (Viereck + Rhombus). Dieses Objekt übernimmt die Schlüssel von Rohr und Kunde (Partner-ID und Rohr-ID). Daran wird eine weitere Entität «Verkauf» angefügt. Dort wird mit Nr # die Anzahl hinterlegt. Alle Attribute werden zum Schlüssel.

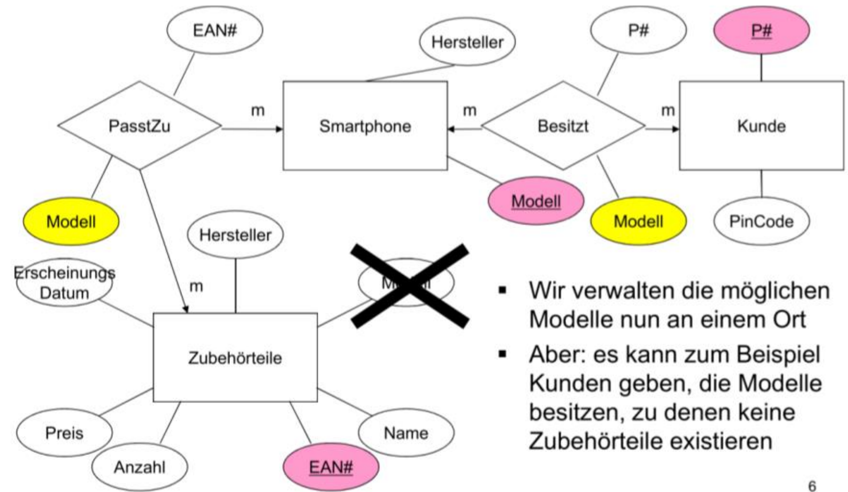
5. Nun gilt es noch das Gleiche für den Einkauf zu tätigen.



➔ Überall wo ein Pfeil reingeht, braucht es einen Identifikator (dieser muss unterstrichen werden und wird vom Objekt vererbt, wo der Pfeil rausgeht)

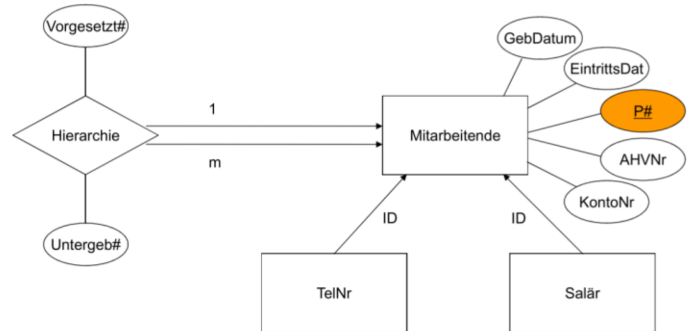
Weiteres Modellbeispiel

Wir wollen festhalten können, dass ein Zubehörteil zu verschiedenen Smartphones passt, so dass wir Kunden mit Vorschlägen zu ihren Smartphones beglücken können



Beziehungstyp-Hierarchie

- Wir wollen die Geschäftshierarchie modellieren
- Wir führen einen Beziehungstypen "Hierarchie" ein
- Beide Fremdschlüssel referenzieren denselben Primärschlüssel P# in Mitarbeiter
- Die Bezeichnungen sollten so gewählt werden, dass die Semantik verständlich ist
- Mitarbeitenden ist höchstens ein/e Vorgesetzte/r zugeordnet
- Vorgesetzte haben beliebig viel Untergebene



Korrektes ER-Diagramm

- Begriff von Markowitz (1987)
- Welche formal syntaktischen Eigenschaften benötigt ein ERDiagramm, um gewisse erwünschte Eigenschaften bei Abbildung in Relationen zu garantieren?
- Wir betrachten eine vereinfachte, rekursive Fassung (Buff 1989)
- Erfahrung zeigt, dass alle in der Praxis vorkommenden Anforderungen an Datenstrukturen durch ein korrektes Diagramm abgedeckt werden können.
- Ein korrektes ER Diagramm ist ein ER Diagramm, welches aus dem leeren Diagramm hergestellt werden kann durch eine Folge von Metaoperationen (oder Regeln)

Regel 1

- Definiere unabhängigen Entitätstyp
- Voraussetzungen: keine
- Ergebnis: ein neues Rechteck (mit eindeutigem Namen)

Regel 2

- Definiere Beziehungstyp
- Voraussetzungen: Ej für 1 ≤ j ≤ n (und 1 ≤ n) gegebene Rechtecke oder rechteckumschlossene Rhomben (Achtung: 1 genügt, siehe Beziehungstyp «Hierarchie»)
- Ergebnis: ein neuer Rhombus R mit '1' oder 'm' markierten Pfeilen zu den E_j

Regel 3

- Definiere Attribut
- Voraussetzung: F ein Rechteck oder Rhombus oder rechteckumschlossener Rhombus
- Ergebnis: neues Oval strichverbunden mit F und mit innerhalb F eindeutigem Namen

Regel 4

- Umwandlung Beziehungstyp in zusammengesetzten Entitätstyp
- Voraussetzungen: D ein Rhombus
- Ergebnis: Rhombus D durch ein Rechteck umschlossen

Regel 5

- Definiere ID-abhängigen Entitätstyp
- Voraussetzungen: F ein Rechteck oder rechteckumschlossener Rhombus
- Ergebnis: neues Rechteck mit 'ID' markiertem Pfeil zu F

Regel 6

- Definiere ISA-abhängigen Entitätstyp
- Voraussetzungen: F ein Rechteck oder rechteckumschlossener Rhombus
- Ergebnis: neues Rechteck mit 'ISA' markiertem Pfeil zu F

Schlüssel

- Bevor ein Diagramm in Relationen abgebildet werden kann, müssen die Schlüssel definiert sein.
- Wir gehen aus von einem korrekten Diagramm
- **Jeder unabhängige Entitätstyp erhält einen oder mehrere Schlüssel**
- **Falls der Entitätstyp eingehende Pfeile hat, wählen wir einen Primärschlüssel**
- Für Beziehungstypen: wir wählen Fremdschlüssel und Schlüssel (gemäss Kardinalitäten)
- Für Umwandlung in zusammengesetzte Entitätstypen: Primärschlüssel wählen
- Entitätstyp E ist ID- oder ISA-abhängig von F: Primärschlüssel in F wählen, Fremdschlüssel und Schlüssel in E wählen

Umwandlung in relationales Modell

- "Rekursiv dem Aufbau des Diagramms entlang"
- Jedes Kästchen geht in eine Relation über, mit entsprechenden Attributen (Domänen sind zu wählen)
- Schlüssel, Fremdschlüssel, Primärschlüssel werden als solche übernommen
- Dies nennt man die "kanonische" Abbildung eines (angereicherten) korrekten ER-Diagramms

Exkurs: Normalisierung

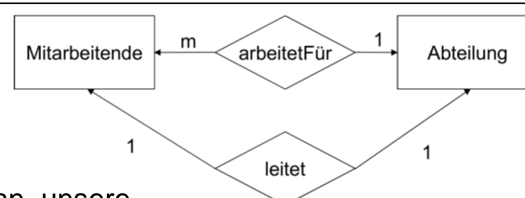
- Wir wollen Redundanzen und Anomalien in unserer Datenbank vermeiden
- Wenn wir einfach "drauflos" Tabellen definieren, treten einige Probleme auf
- Wir müssen so entstehende relationale Modelle noch normalisieren.
- Betrachten Sie folgende Tabelle. Hier gibt es etliche Probleme:
- Redundanzen: die selbe Information wird wiederholt
- Inkonsistenzen: Widersprüche (Berger vs. Burger)
- Was ist, wenn wir MA ohne Abteilung haben?
- Abteilungen ohne MA?
- Was passiert mit der Info zu Abteilung Produktion, wenn uns Huber verlässt?

Name	M#	GebDat	Abt#	AName	AMgr
Meier	17	1985-03-03	3	Verkauf	Berger
Müller	18	1982-07-13	4	Personal	Gerber
Huber	25	1961-08-01	1	Produktion	Baumann
Bühler	28	1982-02-09	3	Verkauf	Burger
Fischer	37	1977-01-16	-	-	-
-	-	-	5	Marketing	Tanner
Schneider	33	1971-06-22	4	Personal	Gerber

Normalisierungstheorie

- Wir wollen all diese Probleme vermeiden
- Im Wesentlichen stehen uns zwei Vorgehensweisen zur Wahl:
    - 1. Vermeidung durch Designmethodik
    - 2. Normalisierungstheorie
- Die Normalisierungstheorie liefert Hinweise, wie man Tabellen zerlegen muss, damit die benannten Probleme nicht auftreten
- Wünschenswert: BCNF – Boyce-Codd Normalform

"Korrekte" Lösung



- Unabhängiges wird unabhängig modelliert
- «Versteckte» Bedingungen werden sichtbar: jeder Mitarbeitende darf höchstens eine Abteilung leiten.

Korrekte ER-Diagramme vs. Normalisierung

- Die Verwendung von korrekten ER-Diagrammen leitet uns dazu an, unsere Tabellen so zu designen, dass die benannten Probleme nicht auftreten
- Die Probleme, die durch Normalisierung "geflickt" werden, werden von vorneherein gemieden
- Dies ist aber natürlich keine Garantie: die Datenbank schützt Sie nicht davon, Unsinn zu designen! (vermischte Entitätstypen etc.) (KEINE «Intelligenz»)
- Für interessierte Studierende: bei Einhaltung der vorgestellten Regeln ist es einfach, normalisierte Datenbanken nach BCNF zu erstellen

SQL

- SQL: Programmsprache für die Bearbeitung von Datenbanken
- **Nicht Turing-complete**: keine vollumfängliche Programmsprache wie z.B. Java; man kann nicht alle denkbaren Aufgaben lösen → meistens von anderen Programmsprachen aufgerufen.
- Dafür **einfach** und sehr **mächtig** für die **Behandlung** von **Mengen**
- Die meisten anderen Programmsprachen behandeln Mengen "one record at the time", nicht so SQL
- Structured English Query Language" ("SEQUEL"). Aus SEQUEL wurde nach einem Trademark-Streit SQL
- In SEQUEL werden per Default Duplikate eliminiert
- Im Nachfolger SEQUEL/2 werden per Default keine Duplikate mehr eliminiert. → Übergang in Welt der Bags
- Hier besprochen: Standard SQL 92
- Syntax: SQL ist **nicht case-sensitive**
- Wir sprechen **in SQL** von **Tabellen**, nicht **Relationen**

- Tabellen implizieren eine bestimmte Reihenfolge der Spalten & Zeilen, im Gegensatz zu "idealen" Relation
- Streng betrachtet ist SQL keine Mengensprache, sondern eine Sprache für den Umgang mit relationalen Bags, resp. Tabellen (**da Duplikate erlaubt**)

Einschub: ER-Schema → Relationenformat

- Jedes „Kästchen“ (jeder Entitäts- und jeder Beziehungstyp) ergibt ein Relationenformat
- Reihenfolge der Attribute beliebig
- Alle Fremdschlüssel-Attribute müssen im Relationenformat aufgeführt werden
- Primärschlüssel-Attribute werden auch bei der Dokumentation des Relationenformats unterstrichen
- Reihenfolge ist gleich wie beim Erstellen eines korrekten ER-Diagramms
- Jedes Relationenformat wird zu einer Datenbank-Tabelle
- Die Tabellen bestehen aus Spalten = Attributen (Struktur-Information) und Zeilen = Tupel (Daten)
- Die Tupel in einer Tabelle sind NICHT geordnet (obschon es optisch natürlich eine „1. Zeile“ gibt), SQL folgt hier der Mengentheorie
- Demzufolge gibt es in SQL auch keine Befehle wie z.B. „gib das zehnte Tupel zurück“
- Die Reihenfolge der Attribute ist beliebig
- Ist die Attributs-Reihenfolge einmal festgelegt, spielt sie jedoch eine Rolle:
  z.B. sind <A, B> und <B, A> verschiedene Schlüssel!
- Jedes Tupel muss eindeutig identifizierbar sein → Unique-Schlüssel ist nötig
- Abfragen können Tupel mit Duplikaten liefern = relationaler Bag

Erweiterte Backus-Naur Form (EBNF)

- EBNF ist eine formale Syntaxbeschreibungssprache
- Wir werden EBNF benutzen, um die Sprache SQL zu beschreiben

Symbol	Bedeutung
" "	Bezeichnen Symbole der Sprache, die **wörtlich** zu **übernehmen** sind
|	**Alternativen** werden mit einem senkrechten Strich getrennt
()	Runde Klammern **dienen** lediglich der **Gruppierung**.
[]	Eckige Klammern stehen für einen **optionalen Inhalt**, der **Null oder einmal vorkommt**.
{}	Geschweifte Klammern stehen für **beliebige Wiederholung** des Inhalts: 0-mal, 1-mal, 2-mal, ...
<>	Spitze Klammern stehen für **Nichtterminale**/**Variablen**.
::=	**Definition** / **Produktionsregel** (z.b. a ::= b)

SQL DDL

- DDL = Data Definition Language
- Ermöglicht Erzeugen und Löschen von Datenbanken, Tabellen und Beziehungen
- Ein DBMS kann mehrere Datenbanken unterhalten und pflegen
- Die einzelne DB (ein „Schema“) ist ein relationales Modell einer realen Problemstellung
- **SQL DDL-Befehle behandeln die Struktur einer Datenbank, nicht den Inhalt**
- Erzeugen einer Datenbank (in SQL Schema genannt):
  "CREATE SCHEMA" <dbName> **["AUTHORIZATION"** <userName>] **";"**
  dbName → kann irgendein freier Name sein, muss jedoch eindeutig sein. (auch **"CREATE DATABASE"**)
- <userName> Administrator oder ‚Besitzer‘ der DB. Nur er kann das Schema wieder löschen
- **Löschen** der DB: **"DROP SCHEMA"** <dbName> [**"CASCADE"**] **";"**
- **Löscht** das **Schema** per Default **nur**, **wenn** es **keine Elemente** mehr **enthält**
- CASCADE: das **Schema** und **alle** seine **Elemente** werden **ohne Rückfrage gelöscht** („und tschüss…..“)
- **Elemente** einer **Datenbank**:
  - Tabellen
  - Constraints = Einschränkungen. Es gibt Einschränkungen auf DB-Ebene, Tabellen-Ebene, Attributs-Ebene. Eine saubere Unterscheidung verhütet Fehler!
  - Indizes (Plural von Index: Indizes/Indices) *
  - Code-Fragmente: Stored Procedures *, Triggers *
    * Hier nicht behandelt. Obschon in den meisten DB-Produkten verfügbar, sind diese Elemente nicht standardisiert, d.h. proprietär und nicht zwischen verschiedenen Produkten transportierbar. SQL kennt Befehle wie z.B. ‚Create Index…“, diese Befehle sind jedoch nicht Bestandteil des SQL-Standards!

Merke für SQL DDL:

- **„CREATE“ erzeugt ein Element** • **„ALTER“ ändert ein Element** • **„DROP“ löscht ein Element**

SQL Datentypen

Einige **grundlegende Datentypen** aus dem SQL-Standard:

- CHAR(n)/CHARACTER(n): Zeichenkette, fixe Länge
- CHAR VARYING(n)/VARCHAR(n): Zeichenkette, variable Länge
- INT/INTEGER: Ganzzahl
- REAL: Fliesskommazahl
- NUMERIC(p,s)/DECIMAL(p,s): Festkommazahl

SQL-DDL, DB-Element „Domain“

- Ein Domain ist ein Constraint auf Datenbank-Ebene
- Der Wertebereich (domain) kann festgelegt werden und gilt für das ganze Schema, d.h. kann in jeder Tabelle referenziert werden
- **"CREATE DOMAIN"** <domainName> **"AS"** dataType["**(**"domain"**)**"] [attributeConstraintDef]"**;**"
  "DROP DOMAIN" <domainName>;
- dataType: von DB-Produkt zu DB-Produkt verschieden → Handbuch
- **Beispiel:** CREATE DOMAIN ssn_type AS CHAR(9);
- **Vorteile von Domains:**
  - **Einmal** zentral definiert und gepflegt
  - Gilt für das ganze DB-Schema
  - v.a. für Schlüssel-Attribute und ausführliche Attributs-Formate geeignet
- Anwendung von Domains: siehe Attributs-Definition

SQL-DDL, DB-Element Table

- **"CREATE TABLE"** <tableName> **"("** tableElementDef {"**,**" tableElementDef} **");"**
- tableElementDef **::=** columnDef | tableConstraintDef
- columnDef **::=** <attributeName> dataType["**(**" domain "**)**"] [attributeConstraintDef]
- attributeConstraintDef **::=** ["**CONSTRAINT** " <constraintName>]
  {"**DEFAULT**" defaultClause | "**NOT NULL**" | "**CHECK**" "**(**"checkCondition"**)**"}
- defaultClause **::=** "**NULL**" | <constant> | <systemVariable>
- **Beispiel:** CREATE TABLE person (name varchar(20) NOT NULL, vorname varchar(20) DEFAULT ('jim'), alter int CHECK(alter > 20));

SQL-DDL, DB-Element Table, Constraints

- ["**CONSTRAINT**" <constraintName>] ist optional
- Es wird **empfohlen**, wenn möglich jeden **Constraint** zu **benennen**
- Nachträglich können (einfach) nur benannte Constraints geändert / gelöscht werden
- Sollte nachträglich ein Constraint geändert werden, welcher nicht benannt wurde → im worst-case die ganze Tabelle neu erzeugen
- In der Praxis haben Tabellen oft mehrere 10 bis hundert Attribute → viel Vergnügen!
- Leider können in PostgreSQL "not null"-Constraints nicht direkt benannt werden (aber: proprietäre Erweiterungen "set not null", "drop not null")
- Beispiel: geg. Tabelle A, Attribut B (Zahl, darf nicht null sein)
- CREATE TABLE A
  (B int **CONSTRAINT cABNN** NOT NULL, ….); ← geht nicht
  (B int **CONSTRAINT cABNN** CHECK(B IS NOT NULL); ← geht theoretisch, wird in Praxis nicht so gelöst
- Später wieder löschen: ALTER TABLE A ALTER B DROP cABNN;
- Beispiel ColumnDef accept CHAR(1) CONSTRAINT aval CHECK(accept='Y' OR accept='N'),
- Beispiel Primärschlüssel PRIMARY KEY (Name,Vorname)
- Beispiel Fremdschlüssel FOREIGN KEY (Name, Vorname) REFERENCES Person
- Verwendung eines bereits definierten Domains: ssn ssn_type,

SQL-DDL, DB-Element Table, CheckKlausel

- Check-Klausel:
  - Ausdruck muss ein logisches true oder false ergeben
  - Test wird bei Eingabe von Daten automatisch ausgeführt
  - Kann Attributswert mit Konstanten oder mit Werten anderer Attribute **derselben** Tabelle vergleichen
- Enthält die Check-Klausel mehrere Attribute, so ist sie eine **TableConstraint** und wird erst nach den Attributs-Definitionen aufgeführt
- Eine Check-Klausel erlaubt eine zentrale Einschränkung des möglichen Wertebereichs → ausnützen
- Bei Verzicht auf Check-Klauseln müssen unzulässige Eingaben in JEDEM Anwendungsprogramm abgefangen werden!

SQL DDL, DB-Element Table

- tableConstraintDef **::=** ["**CONSTRAINT** " <constraintName>] tableConstraintType
- tableConstraintType **::=** Check-Klausel mit mehreren Attributen |
  "**UNIQUE (**"<attributeName> {"**,**" <attributeName>}"**)**" |
  "**PRIMARY KEY (**"<attributeName>{"**,**" <attributeName>}"**)**" |
  "**FOREIGN KEY (**"<attributeName>{"**,**" <attributeName>}"**)**"
  "**REFERENCES** " <tableName> ["**(**"<attributeName>{, <attributeName>}"**)**"]
  [impliziter Foreign Key-Trigger {impliziter Foreign Key-Trigger}]
- Unique: mehrere Unique-Klauseln pro Tabelle möglich (→ Schlüssel!)
- Die **Werte** der **Attributsmenge** jeder Unique-Klausel müssen in **jedem Tupel verschieden** sein, wird bei der Dateneingabe vom DBMS überprüft
- Beispiel Online-Shop, Tabelle Besitzt CONSTRAINT BesitztUni UNIQUE(pNr, Modell)
- Primary Key (Primärschlüssel): es darf höchstens einen PK pro Tabelle geben
- Ein PK ist dann nötig, wenn die Tabelle von einer anderen (mit Foreign Key) referenziert wird. Ansonsten genügt eine UniqueKlausel.
- Beispiel Online-Shop, Tabelle BestellPosition CONSTRAINT BestPosPK PRIMARY KEY(bestNr, posNr)
- "**FOREIGN KEY (**" <attributeName>{, <attributeName>} "**) REFERENCES**" <tableName> ["**(**" <attributeName>{, <attributeName>} "**)**"] [impliziter Foreign KeyTrigger {, impliziter Foreign Key-Trigger}]
- "**FOREIGN KEY (**" <attributeName>{, <attributeName>} "**) REFERENCES**" <tableName> → ist minimale Angabe

SQL DDL, DB-Element Table, Foreign Key

- Falls in Table <tableName> keine Attribute genannt werden, wird automatisch der Primary Key von <tableName> referenziert
- d.h.: durch explizite Angabe können auch andere Attribute als die des PK referenziert werden. Davon ist Anfängern abzuraten!
- Anhand dieser Referenz sichert das DBMS bei Dateneingabe, aber auch bei Löschen von Tabellen, die referentielle Integrität
- Beispiel Online-Shop, Tabelle ‚PasstZu' (2 Fremdschlüssel!) :
  CONSTRAINT PasstZuFK1 FOREIGN KEY (Modell) REFERENCES Smartphone,
  Constraint PasstZuFK2 FOREIGN KEY (eanNr) REFERENCES Zubehoerteil
- Implizite Foreign Key-Trigger sind Aktionen, die vom DBMS automatisch bei Dateneingabe ausgeführt werden FK-Trigger ::= **("ON DELETE" | "ON UPDATE"**), (**"NO ACTION" | "SET NULL" | "SET DEFAULT" | "CASCADE"**)

SQL DDL, DB-Element Table

- ON UPDATE: wenn ein Tupel **in der referenzierten Tabelle** geändert oder eingefügt wird
- ON DELETE: wenn ein Tupel **in der referenzierten Tabelle** gelöscht wird
- NO ACTION: falls der Wert des Fremdschlüssels nach der Aktion keinem gültigen Primärschlüsselwert der referenzierten Tabelle mehr entsprechen würde, wird die Aktion verboten
- SET NULL, SET DEFAULT: selbsterklärend (SET NULL = pfui)
- CASCADE: Werte des Fremdschlüssels werden bei Ändern des PKWerts der referenzierten Tabelle automatisch angepasst
- **ACHTUNG:** wird das referenzierte Tupel gelöscht, werden alle Tupel dieser Tabelle mit diesem FK-Wert ohne Warnung gelöscht (ON DELETE CASCADE)
- Beispiel Online-Shop: Bestellung löschen → alle zugehörigen BestellPositionen auch gelöscht
- Beispiel Online-Shop, Tabelle ‚PasstZu' (betrachten nur Fremdschlüssel zu ‚Smartphone') :
- CONSTRAINT PasstZuFK1 FOREIGN KEY (Modell) REFERENCES Smartphone ON UPDATE CASCADE
- FK-Element erst hier fertig ON DELETE CASCADE,
- Hat folgende Auswirkungen:
  - wird in ‚Smartphone' ein Modellname geändert (z.B. von ‚S22' auf ‚Galaxy S22'), so werden alle Tupel von ‚PasstZu', deren Attributswerte von ‚Modell' ‚S22' enthalten, auf ‚Galaxy S22' geändert
  - wird in ‚Smartphone' das Modell ‚S3' gelöscht, so werden in ‚PasstZu' alle **Tupel** mit ‚S3' ebenfalls **gelöscht**! Die entsprechenden Zubehörteile sind danach keinem Modell mehr zugeordnet.

SQL DDL, Tabellen ändern

- "ALTER TABLE" <tableName> action ";" action ::= "ADD" <column> | "DROP" <column> | "ALTER" <column>| "ADD" constraint | "DROP" constraint
- **Hierarchie:** Table → TableElement → ElementContraintDef

Jede Hierarchie-Ebene kann geändert werden

- ADD: neues Attribut, neuer Table-Constraint, neuer AttributConstraint (teilweise)
- Beispiel Online-Shop, Tabelle ‚HatGeliefert', neues Attribut ‚menge':

- ALTER TABLE HatGeliefert ADD menge INT NOT NULL CONSTRAINT hatGelMengePos CHECK(menge>0);
- Änderung "not null" in PostgreSQL:
  - ALTER TABLE Zubehoerteil ALTER name SET NOT NULL;
  - ALTER TABLE Zubehoerteil ALTER name DROP NOT NULL;
- Beispiel Online-Shop, Tabelle ‚HatGeliefert', Constraint wieder löschen:
  ALTER TABLE HatGeliefert DROP CONSTRAINT hatGelMengePos;
- Beispiel Online-Shop, Tabelle ‚HatGeliefert', Attribut ‚menge', defaultWert setzen:
- ALTER TABLE HatGeliefert ALTER menge SET DEFAULT 1;
- Bei Löschen eines Attributs / eines Constraints sind Folgeaktionen mit Restrict | Cascade steuerbar.
- Beispiel Online-Shop, Tabelle ‚HatGeliefert', Attribut eanNr löschen mit Cascade:
  ALTER TABLE HatGeliefert DROP eanNr CASCADE;
- eanNr ist im Unique enthalten und ausserdem Fremdschlüssel (eanNr wird von diesen Constraints referenziert!) → Attribut wird gelöscht, Unique-Klausel und entsprechender FK wird "gebrochen"!
- Beispiel Online-Shop, Tabelle ‚HatGeliefert', Attribut eanNr löschen mit Restrict:
  ALTER TABLE HatGeliefert DROP eanNr RESTRICT;
- eanNr ist im Unique enthalten und ausserdem Fremdschlüssel → Attribut kann nicht gelöscht werden, bevor Unique- und FK-Klausel geändert worden sind. Das gilt auch, falls Views das Attribut eanNr referenzieren.

Zum Thema "Standards":

- Also, the ability to specify more than one manipulation in a single ALTER TABLE command is an extension.
- D.h. "ALTER TABLE bar ADD a int, ADD b char(20);" funktioniert in PostgreSQL, aber nicht notwendigerweise in anderen DBs → In Oracle: "ALTER TABLE bar ADD (a int, b char(20));"

SQL DDL, Tabellen löschen

- Tabellen werden gelöscht mit "Drop" (analog zum Löschen von Schemata und Domains)
- **"DROP TABLE"** <tableName> {**","** <tableName>} [**"CASCADE"** | **"RESTRICT"**] **";"**
- Löscht ganze Tabellen, inklusive der Struktur (nicht nur die Einträge)
- Wenn Objekte von Tabelle abhängen (z.B. foreign keys) muss Löschen mit CASCADE erzwungen werden

SQL DML

- DML = Data Manipulation Language
- SQL-Teil, der **Inhalt, nicht Struktur**, von Datenbanken pflegt
- Daten pflegen = Einfügen, Ändern, Löschen
- SQL-Befehle Insert, Update, Delete → → → → →

	DDL	DML
Einfügen / erzeugen	CREATE	INSERT
Ändern	ALTER	UPDATE
Löschen	DROP	DELETE

SQL DML: Insert

- "INSERT INTO" <tableName> ["(" attrList ")"] ("VALUES (" <valueList> ")") | query ";"
- attrList ::= <attributeName> {"," <attributName>} → Reihenfolgen müssen übereinstimmen
- valueList ::= <attributeValue> {"," <attributeValue>} → Reihenfolgen müssen übereinstimmen
- Insert fügt immer ganze Tupel in eine Tabelle ein
- Beispiel Online-Shop, Tabelle Salaer INSERT INTO Salaer VALUES(12, 35000, 24, 5, 2003);
- Falls nicht für alle Attribute Werte einfügen soll INSERT INTO Salaer(pNr, betrag) VALUES(12, 35000);

SQL DML: Update

"UPDATE" <tableName>
"SET" <attributeName> "=" <attributWert> {"," <attributeName> "=" <attributWert>}
["WHERE" searchCondition] ";"

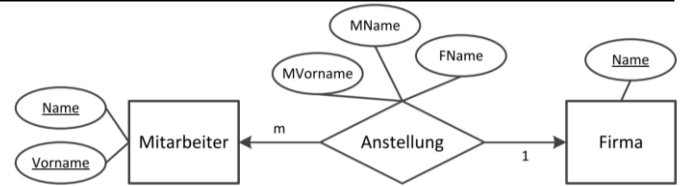
- Beispiele Online-Shop, Tabelle Salaer
  UPDATE Salaer SET betrag = 36000; UPDATE Salaer SET betrag = 1.05 * betrag;
- **Merke: 4 Grund-Rechenoperationen erlaubt**

SQL DML: Delete

- "DELETE FROM" <tableName> [searchCondition] ";"
- Delete löscht immer ganze Tupel! Man kann nicht ein einzelnes Attribut „löschen".
- Beispiel Online-Shop, Tabelle Salaer: DELETE FROM Salaer WHERE betrag > 100000;
- Ohne SearchCondition wird die ganze Tabelle geleert!
- ABER: ein Attribut „löschen" → ändern.

UPDATE Salaer SET betrag = null WHERE betrag > 100000;

Beispielaufgabe – SQL DDL und DML



Erstellung der Tabellen inkl. PK und FK

- DROP TABLE IF EXISTS Mitarbeiter;
- CREATE TABLE Mitarbeiter(
  Vorname varchar(100) NOT NULL,
  Name varchar(100) NOT NULL,
  CONSTRAINT PK_Mitarbeiter PRIMARY KEY(Vorname, Name));

- DROP TABLE IF EXISTS Firma;
- CREATE TABLE Firma(Name varchar(100) NOT NULL,
  CONSTRAINT PK_Firma PRIMARY KEY(Name));

- DROP TABLE IF EXISTS Anstellung;
- CREATE TABLE Anstellung(MVorname varchar(100) NOT NULL,
  MName varchar(100) NOT NULL, FName varchar(100) NOT NULL,
  CONSTRAINT PK_Anstellung PRIMARY KEY (MVorname,MName),
  CONSTRAINT FK_Mitarbeiter FOREIGN KEY (MVorname,MName) REFERENCES
  Mitarbeiter(Vorname,Name),
  CONSTRAINT FK_Firma FOREIGN KEY (FName) REFERENCES Firma(Name));

Neues Attribut bei bestehender Tabelle hinzufügen

- ALTER TABLE Firma ADD Gründungsjahr integer NOT NULL; -- NOT NULL erzwingt beim Hinzufügen
- ALTER TABLE Anstellung ADD Jahreslohn decimal(10,2) NOT NULL; -- eines neuen Tupels auch einen
- ALTER TABLE Mitarbeiter ADD PLZ integer NOT NULL; -- Wert zu hinterlegen.
- ALTER TABLE Mitarbeiter ADD Ort varchar(100) NOT NULL;
- ALTER TABLE Mitarbeiter ADD Strasse varchar(100) NOT NULL;
- ALTER TABLE Mitarbeiter ADD HausNr varchar(8) NULL; -- bei NULL wird Defaut Element zu NULL

Tupels hinzufügen

- INSERT INTO Firma (Name, Gründungsjahr) VALUES ('Migros', 1925);
- INSERT INTO Mitarbeiter (Name, Vorname, PLZ, Ort, Strasse, HausNr)
  VALUES ('Müller', 'Heinz', 8400, 'Winterthur', 'Badgasse', '12');
- INSERT INTO Anstellung (MName, MVorname, FName, Jahreslohn)
  VALUES ('Müller', 'Heinz', 'Migros', 70000);

Neues Attribut bei bestehender Tabelle hinzufügen, welches nicht leer sein darf!

- ALTER TABLE Mitarbeiter ADD TelNr varchar(12) NOT NULL DEFAULT 'n/a'; -- Da bereits Datensätze für Mitarbeiter vorhanden sind, werden auch diese um eine neue Spalte erweitert. Wenn nun „NOT NULL“ definiert ist, aber kein „DEFAULT“, wird automatisch NULL als Wert in den neuen Spalten der bestehenden Datensätze eingefügt. Dies verletzt in diesem Moment jedoch sofort die eben definierte Einschränkung.

Änderung von bereits bestehenden Tupels

- UPDATE Mitarbeiter SET PLZ = '8401', Ort = 'Winterthur', Strasse = 'Im Lee', HausNr = NULL
  WHERE Nachname = 'Müller' AND Vorname = 'Heinz';

Änderung eines Wertes bei einem Attribut, welches Referenz in einer anderen Tabelle ist

- UPDATE Firma SET Name='Migrolino' WHERE Name='Migros'; -- **Führt so zu Fehler,**
- ALTER TABLE Anstellung DROP CONSTRAINT fk_firma; -- müssen zuerst Verbindung entfernen
- UPDATE Firma SET Name = 'Migrolino' WHERE Name = 'Migros'; -- danach wird Wert angepasst
- UPDATE Anstellung SET FName = 'Migrolino' WHERE FName = 'Migros'; -- und zwar in beiden Tabellen
- ALTER TABLE Anstellung ADD CONSTRAINT FK_Firma FOREIGN KEY (FName)
  REFERENCES Firma(Name); -- danach muss der Fremdschlüssel wieder definiert werden.

Löschen der ganzen Tabellen

- DROP TABLE Mitarbeiter CASCADE; -- mit CASCADE, da Verbindungen zu anderen Tabellen aktiv

SQL Query Übersicht

- Häufigste Anwendung von SQL: Daten abfragen
- Wenige Befehle, aber komplexe Verschachtelungen möglich
- Ist an relationale Algebra angelehnt, setzt diese aber nicht exakt um: gewisse Befehle eliminieren Duplikate nicht automatisch
  ➔ Widerspruch zur Definition von Mengen
- Bei **Duplikaten** erhält man einen **relationalen Bag** (kompliziertere Algebra)
- Nullwerte folgen einer dreiwertigen Logik (true, false, unknown) → unknown = „so falsch, dass auch das Gegenteil nicht stimmt"
- Schlussfolgerung: Nullwerte schon im DB-Design vermeiden (was wir mit unserer Design-Methode tun)

Flüchtigkeit der Abfragen

- SQL-Abfragen in unserem Sinne sind grundsätzlich flüchtig.
- Grund: zu viel Speicher würde benötigt werden

SQL Query: allgemeiner Ausdruck für Abfragen

- Query ::= <subquery> {("UNION" | "INTERSECT"[2] | "EXCEPT"[2]) ["ALL" | "<u>DISTINCT</u>"[1]] <subquery>} ";[3]"
- subquery ::= vollständige SQL-Abfrage
- Abfragen können verschachtelt sein. Egal, wie kompliziert Abfrage, liefert nur EIN Resultat als neue Tabelle
  1. Defaultwert ist unterstrichen. Wird Angabe weggelassen, so wird automatisch Defaultwert eingesetzt
  2. **INTERSECT** ist der SQL-Ausdruck für **Durchschnitt** (∩), **EXCEPT** der für **Differenz** (\)
  3. SQL-Statements werden in den meisten DBMS mit „;" abgeschlossen, gehört jedoch nicht zum Standard. Manche Systeme verlangen dies sogar explizit.

Einfache Subquery: SELECT-Klausel

- subquery ::= "SELECT" ["<u>ALL</u>" | "DISTINCT"] <attributeList> "FROM" <tableName> ";"
- Das ist die einfachste Form einer Subquery.
- Beispiel Webshop: SELECT[1] name, hersteller FROM Zubehoerteile;
- SQL ist **nicht case-sensitive**, „SELECT" bedeutet dasselbe wie „Select" oder „sElEct"
- "Select" ist historisch "gewachsen". Im Folgenden wird nicht immer die gebräuchlichste Form einer Abfrage zuerst gezeigt, da sich das Vorgehen an den Erkenntnissen aus den früheren Kapiteln orientiert.

SELECT und Projektion

- Diese Form des SELECT entspricht der Projektion in der Welt der Bags:
- SELECT Name FROM Besucher; entspricht $\pi_{Name}(Besucher)$,
- während SELECT DISTINCT Name FROM Besucher; äquivalent zu $\delta(\pi_{Name}(Besucher))$ ist.
- SELECT * FROM Besucher; wählt alle Attribute der Datenquelle aus
- SELECT Name n FROM Besucher; oder identisch SELECT Name AS n FROM Besucher → Umbenennen von Attributen im **Resultat/Ausgabe**
- SELECT Besucher.Name n FROM Besucher; oder identisch SELECT b.Name n FROM Besucher b;
- attributeList::= (<columnSpec> {"," <columnSpec>} | "*")
- columnSpec ::= <scalarExpr> ["AS" <neuer spaltenName>]
- scalarExpr ::= (<columnRef> | <literal>) {<operator> <scalarExpr>}
- operator ::= ("+" | "-" | "*" | "/")
- columnRef ::= [[<schemaName> "."] <tableName> "."][3] <attributeName>
- <attributeName> muss in Datenquelle **eindeutig** sein → soviele Angaben, bis Eindeutigkeit gewährt ist
- Default ist **KEINE Duplikatelimination** (SELECT ALL)
- SELECT 1 FROM Hierarchie; listet so oft ‚1' auf, wie es Hierarchie-Tupel gibt
- SELECT frequenz * 1.5 FROM gast; einfache Operatoren können direkt auf Attribute angewendet werden

Einfache Subquery: Datenquelle

- Datenquelle ::= <tableExpr> | <joinExpr>
- tableExpr ::= [<schemaName>". "]<tableName> [["AS"] <aliasName>]
- joinExpr ::= (<tableExpr> | "(" <joinExpr> ")") ["NATURAL" | ("LEFT" | "RIGHT" | "FULL") "OUTER" | "CROSS"] "JOIN" <tableExpr> ["ON" <joinCondition>][1]
- joinCondition ::= <attributeComparison> {("AND" | "OR") <attributeComparison>
- attributeComparison ::= ([<tableName>". "]<attributeName> | <literal>) <theta-Operator> ([<tableName>". "]<attributeName> | <literal>)
- theta-Operator ::= "<" | "<=" | "=" | ">=" | ">" | "<>"[2]
  1. Alle Join-Typen mit Ausnahme Natural Joins und Kreuzprodukts (CROSS JOIN) erfordern JoinCondition
  2. in der Praxis fast ausschliesslich Equi-Join üblich ("=")

SELECT und Join

- Diese Form des SELECT entspricht in der Welt der Bags dem Join.
- ACHTUNG: In SQL sind Attributnamen grundsätzlich nur innerhalb einer Tabelle eindeutig.
- SELECT * FROM Zubehörteile CROSS JOIN Smartphone
- Entspricht Zubehörteile ⋈ Smartphone, falls beiden keine gemeinsamen Attribute haben (=Kreuzprodukt)
- Die Variante SELECT * FROM Zubehörteile NATURAL JOIN PasstZu entspricht aber dem Ausdruck Zubehörteile ⋈ PasstZu in der Bag-Welt (JOIN auf gleich benannten Attributen)
- SELECT * FROM restaurant AS r JOIN gast AS g ON r.name = g.rname
  → mit ON r.name = g.rname können zwei Attribute aus zwei verschiedenen Relationen mit gleicher Bedeutung, aber unterschiedlichen Attributnamen gejoint werden (hier funktioniert Natural join nicht….
  → Attribute r.name, g.rname haben anschliessend die identischen Einträge, eines ist damit überflüssig
- SELECT * FROM (besucher b JOIN gast g ON b.name = g.bname AND b.vorname = g.bvorname)
  → JOIN mit AND können zwei Bedingungen oder mehr für das joinen definiert werden
- SELECT * FROM (besucher NATURAL JOIN gast) NATURAL JOIN Restaurant
  → mit () und erneutem JOIN können zwei oder mehr Relationen gejoint werden
  → Natural Join eliminiert jeweils eines der übereinstimmenden Attribute

SearchCondition:

- Entspricht der Selektion in der Welt der Bags
- searchCondition ::= "WHERE" ["**NOT**"] <logicalComparison> {("**AND**" | "**OR**") ["**NOT**"] <logicalComparison>}
- compareOperator ::= ("<" | "<=" | "=" | ">=" | ">" | "<>" | "LIKE") -- LIKE ausschliesslich für Text (String)
- SELECT * FROM Person WHERE Name = 'Müller' -- entspricht $\sigma_{Name=Müller}(Person)$
- SELECT * FROM restaurant WHERE suppenpreis >= 2.2 -- gibt alle Restaurants mit Suppenpreis >= 2.2
- WHERE ist Tupelfilter. Alle Tupel, für die logische Ausdruck wahr ist, werden im Resultat übernommen.
- Logischer Ausdruck (suppenpreis >= 2.2) kann beliebig komplex werden.
- Suchprädikate werden mit dreiwertigen Logik ausgewertet (false, true, unknown). Unknown steht für NULL.
  - ¬ unknown = unknown
  - false ^ unknown = false
  - unknown ^ unknown = unknown
  - true ^ unknown = unknown
  - false V unknown = unknown
  - unknown V unknown = unknown
  - true V unknown = true
- SELECT * FROM gast WHERE bname LIKE 'M__er' -- Suchen Mitarbeiter, die Meier, Maier, Mayer heissen
- '_' steht für ein **einzelnes**, **beliebiges Zeichen**
- '%' steht für eine **beliebige Anzahl Zeichen**
- LIKE kann **nur** für **Strings** verwendet werden.
- Text und Datum werden in SQL in einfache Anführungszeichen gebettet (')
- SELECT * FROM gast WHERE bname LIKE 'M%' AND bname LIKE '%er' -- Starten mit M und endet mit er
- SELECT * FROM gast WHERE bname LIKE 'M%' OR bname LIKE 'A%' -- Namen, starten mit M ODER A
- SELECT * FROM gast WHERE bname BETWEEN 'A%' AND 'N%' -- Namen, starten mit A, or B, …, or M
- SELECT * FROM gast WHERE frequenz IS NULL -- Nicht etwa den Wert Null, sondern Preis fehlt / ist leer
- SELECT * FROM gast WHERE bname IN ('Meier', 'Müller') -- Vergleich mit Werte in einer Liste

Ordnung

- Resultate von SQL-Abfragen sind grundsätzlich **in** ihrer **Ordnung nicht vorgegeben**. D.h., dieselbe Abfrage kann (muss aber nicht) ein paar Minuten später ein umgeordnetes Resultat ergeben.
- SQL ist eben eine Mengensprache, Tupel in Relationen sind auch nicht geordnet.
- Man kann SQL-Resultate aber nach beliebigen Attributen ordnen (sortieren) lassen:
- SELECT * FROM gast ORDER BY bname, bvorname -- sortiert **auf**steigend (a, b, …) zuerst Attribut bname, dann Attribut bvorname (**Ab**steigend sortieren: "ORDER BY bname DESC, bvorname DESC")
- Ist mit "ORDER BY" gewährleistet, dass Resultat bei mehreren gleichen Abfragen immer gleich geordnet ist? Wenn verschiedene Tupel gemäss Sortierkriterium gleichwertig sind, kann es **weiterhin** zu **Umsortierungen** kommen.
- Dieser Effekt kann ausgeschlossen werden, wenn das **Sortierkriterium einen Schlüssel enthält**.
- Merke: Spalten werden immer in der gleichen Reihenfolge ausgegeben (=Reihenfolge der Definition)

Lexikographische Ordnung

- Für manche Abfragen wollen wir eine lexikographische Ordnung à la Telefonbuch verwenden.
- **Beispiel:** Alle, die in lexikographischen Reihenfolge zwischen Meier, Hans und Schmid, Joseph liegen.
- SELECT * FROM gast WHERE ((bname = 'Meier' AND bvorname >= 'Hans') OR
  (bname > 'Meier' AND bname < 'Schmid') OR (bname = 'Schmid' AND bvorname <= 'Joseph'))

EXISTS

- Gesucht sind alle Besucher, deren Vorname bei einem weiteren Besucher auch vorkommt.
- Jeder Besucher erfüllt die Bedingung, dass in der gleichen Tabelle eine anderer (=verschiedener) Besucher existiert mit gleichem Vornamen. → Wir brauchen eine Art von innerer Schleife: für jedes Tupel wird die Tabelle nach passenden Gegenstücken "abgegrast"
- SELECT * FROM besucher AS x
  WHERE EXISTS (SELECT 1 FROM besucher AS y -- → Bereichsvariablen
  WHERE x.vorname = y.vorname AND NOT (x.name = y.name))

UNION/INTERSECT/EXCEPT

- SQL erlaubt Mengenoperationen (Bag Operationen) auf **kompatiblen** Tabellenformaten (gleiche Domänen)
- Die Bag **Concatenation** ⊔ entspricht UNION ALL
- Der **Durchschnitt** ∩ entspricht INTERSECT ALL
- Die **Differenz** \ entspricht EXCEPT ALL
- Mit Duplikatelimination kombinierbar: UNION DISTINCT / INTERSECT DISTINCT / EXCEPT DISTINCT
- Default ist "DISTINCT"!
- **Beispiel:** Wir haben 2 Tabellen mit Personendaten (**gleiches Tabellenformat**) und wollen eine Liste nur derjenigen Einträge, welche in der ersten, nicht aber zweiten Liste, vorkommen:
  SELECT * FROM Besucher1 EXCEPT ALL SELECT * FROM Besucher2;

Aggregatfunktionen ohne Gruppierung

- SQL kennt die 5 Aggregatfunktionen:
  - COUNT
  - MAX
  - MIN
  - SUM[1]
  - AVG[1]
- [1]Attribute müssen einen zählbaren Domain haben
- **Beispiel Webshop:** zähle die Anzahl verschiedener Namen der Zubehörteile
  SELECT COUNT(name) AS anzahlNamen FROM Zubehoerteile; -- Umbenennen Attribut bietet sich an,
  SELECT COUNT(*) FROM Zubehoerteile; -- weil einige DB-Systeme Phantasie-Namen vergeben.
  → im Resultat ist sonst nicht ersichtlich, was die Zahl bedeutet
- Unterschied zwischen COUNT(*) und COUNT(<attributName>):
  - COUNT(<attributName>) zählt nur diejenigen Tupel, bei denen der Wert des Attributs **nicht NULL** ist
  - COUNT(*) zählt alle Tupel (es gibt kein Tupel, bei dem alle Attribute gleichzeitig NULL sein können)
    → deshalb immer mit COUNT(*) arbeiten!
- Weitere Option: SELECT COUNT (DISTINCT NAME) FROM Zubehoerteile;
- Zählt Anzahl verschiedener Namen (falls es verschiedene Zubehörteile mit demselben Namen geben sollte)
- Aggregatfunktionen ohne Gruppierung liefern eine Tabelle mit EINEM Tupel und EINEM Attribut (also eine Zahl. Ist aber eine richtige Tabelle, die weiterverarbeitet werden kann).

Aggregatfunktionen mit Gruppierung

- SELECT bname, COUNT(rname) AS Anzahl_restaurants -- Zählt pro Gastname die Anzahl Restaurants
  FROM gast GROUP BY bname ORDER BY bname -- Problematik Gäste mit gleichen Nachnamen ↓
- SELECT bname, bvorname, COUNT(rname) AS Anzahl_restaurants -- Zählt pro Gastname und Vorname,
  FROM gast GROUP BY bname, bvorname ORDER BY bname, bvorname -- die Anzahl Restaurants
- Liefert Tabelle mit einem Tupel pro bname respektive pro bname und bvorname, d.h. ein Tupel pro Gast
- **Komplizierteres Beispiel:** Summe aller Biere pro Person, unabhängig von Sorte, welche > 5 Liter sind
- SELECT bname, bvorname, SUM(literprowoche) AS totalliterbierprowoche
  FROM lieblingsbier GROUP BY bname, bvorname HAVING SUM(literprowoche) > 5
- SUM(literprowoche) wird zu totalliterbierprowoche umbenannt. Das ist letzte Aktion, die DBMS in Abfrage vornimmt. Innerhalb der Abfrage (bei HAVING…) muss das Attribut deshalb noch mit ursprünglichen Namen SUM(literprowoche) angesprochen werden.

Reihenfolge, in der SQL eine Abfrage bearbeitet:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

- Diese Reihenfolge ist wichtig, um den Unterschied zwischen WHERE und HAVING zu verstehen: WHERE wirkt auf **jedes Tupel der Datenquelle**, **unabhängig** von einer allfälligen **Gruppierung**
- **Gruppiert wird die durch WHERE bereinigte Datenquelle**
- HAVING schliesst anschliessend an die Gruppierung ganze Gruppen aus
- **Beispiel**: Liste aller Abteilungen (dno), welche min. 3 Angestellte beschäftigen, die > 35‘000.- verdienen.
- SELECT dno, COUNT(*) AS ANZAHL FROM Employee JOIN WorksFor ON ssn = wssn
  WHERE salary > 35000 GROUP BY dno HAVING COUNT(*) > 2;

IN-Prädikat

- Das «IN» Prädikat hat zwei Grundformen: «IN <**Werteliste**>» und «IN <**Subquery**>».
- **Erste Form:**
- SELECT Name, Vorname, Strasse, Gebtag FROM Besucher
  WHERE Name IN ('Meier', 'Müller', 'Maurer')
- **Zweite Form:**
- SELECT Name, Strasse, Wirtsname FROM Restaurant
  WHERE Name IN (SELECT Rname FROM Sortiment WHERE Bsorte = 'Sorte1' AND AnLager > 0)
- In manchen Systemen (Teil des SQL92-Standards!) ist auch eine allgemeinere Form möglich:
- SELECT Name, Vorname, Strasse, Gebtag FROM Besucher
  WHERE (Name, Vorname) IN (SELECT Bname, Bvorname FROM Gast WHERE Rname = 'Ochsen');
- **Achtung bei negiertem IN-Operator!**
- Beispiel Vorname, Name aller Angestellten, welche keine Angehörigen desselben Geschlechts haben:
- SELECT fName, lName FROM Employee
  WHERE (ssn, sex) NOT IN (SELECT Dependent.ssn, Dependent.sex FROM Dependent
  WHERE Dependent.ssn = Employee.ssn);
- Liefert nicht nur die Angestellten, welche eben keine Angehörigen desselben Geschlechts haben, sondern auch jene, welche überhaupt keine Angehörigen haben. War das gemeint??

ALL, SOME/ANY-Operator

- Prädikat «ALL» wird zusammen mit Vergleichsoperator (=, <>, <, <=, >, >=) bei Subqueries eingesetzt.
- SELECT * FROM Gast
  WHERE Frequenz > ALL(SELECT Frequenz FROM Gast WHERE Rname = 'Ochsen');
- SELECT * FROM Gast g1 WHERE NOT EXISTS
  (SELECT 1 FROM Gast g2 WHERE Rname = 'Ochsen' AND g2.Frequenz >= g1.Frequenz)
- Die beiden Formen sind bei Vorkommen von NULL nicht identisch. Wenn mindestens eine Frequenz eines Gasts des Restaurant Ochsen "NULL" ist, liefert die Form mit ALL nichts.
- Die Form mit EXISTS liefert hingegen für NULL-Werte "false", d.h. bei NOT EXISTS "true" → Alle Gäste mit Frequenzen oberhalb der bekannten Frequenzen, sowie Gäste mit Frequenzen "NULL" werden gelistet!
- $\pi_{name,vorname}(Besucher) \setminus (\pi_{bname,bvorname}(Gast) \sqcup \pi_{bname,bvorname}(Lieblingsbier))$ ohne EXCEPT
- SELECT Name, Vorname FROM Besucher AS x
  WHERE NOT EXISTS (SELECT 1 FROM Gast WHERE Bname = x.Name AND Bvorname = x.Vorname)
  AND NOT EXISTS (SELECT 1 FROM Liebgsbier WHERE Bname = x.Name AND Bvorname = x.Vorname)
- Auch das Prädikat "ANY" kann zusammen mit einem Vergleichsoperator (=, <>, <, <=, >, >=) bei Subqueries eingesetzt werden. SOME ist ein Synonym für ANY.
- SELECT * FROM Gast -- "= ANY" entspricht "IN"
  WHERE Frequenz > ANY(SELECT Frequenz FROM Gast WHERE Rname = 'Ochsen');

Weitere JOINS

- Es stehen uns bekanntlich weitere Formen des JOINS zur Verfügung: "CROSS JOIN":
- SELECT * FROM a CROSS JOIN b;
- Oder auch: SELECT * FROM a, b;

OUTER JOINS

- In NATURAL JOINs und Theta-JOINS zweier Bags r und s sind Tupel aus r, welche keine Entsprechung in s haben gemäss der JoinKriterien, unsichtbar (→ "fallen heraus")
- OUTER JOINs sollen "nonmatching"-Tupel berücksichtigen
- [NATURAL] **LEFT** OUTER JOIN: $r ⟕ s$
- Alle Tupel, die bei einem normalen Join "erzeugt" werden, plus die weiteren Tupel aus r, mit NULLs ergänzt.
- [NATURAL] **RIGHT** OUTER JOIN: $r ⟖ s$
- Alle Tupel, die bei einem normalen Join "erzeugt" werden, plus die weiteren Tupel aus s, mit NULLs ergänzt.
- [NATURAL] **FULL** OUTER JOIN: $r ⟗ s$
- Alle Tupel, die beim normalen Join "erzeugt" werden, plus die weiteren Tupel aus r, s, mit NULLs ergänzt.
- **Bei allen drei JOINs**, ohne NATURAL müssen die entsprechenden **JOIN-Bedingungen gelistet werden**!
- Outer Joins können gebraucht werden, um «nicht-vorhandene» Information zu suchen:
- SELECT Bname, Bvorname
  FROM Gast NATURAL LEFT OUTER JOIN Lieblingsbier WHERE Bsorte IS NULL;
- **Vorsicht:** OUTER JOINs sind nicht immer effizient
- Es existieren verschiedene (wilde) OUTER JOIN-Syntaxen
- Es werden NULLs eingeführt, eigentlich nur der Darstellung willen → OUTER JOINs mit **Vorsicht** geniessen

Komplexere Abfragen / Beispiele

- In einer Tabelle hat es mehrere Einträge, wir suchen pro Person jeweils den einen Eintrag mit der tiefsten Bewertung. Dies können wir mit zwei identischen Tabellen machen. Diese identischen Bewertungen pro Person werden dann voneinander unterschieden. Mit dem kleiner als Operator werden alle bis auf das kleinste tiefste Element in der Liste aufgenommen und anschliessend mit dem NOT EXISTS wieder abgezogen, sodass nur der tiefste Eintrag pro Person übrigbleibt.
- SELECT Name, Vorname, Gebtag, Bsorte, Literprowoche
  FROM Besucher b, Lieblingsbier l WHERE b.Name = l.Bname AND b.Vorname = l.BVorname
  AND NOT EXISTS (SELECT 1 FROM Lieblingsbier l2 WHERE l.Bname = l2.Bname
  AND l.BVorname = l2.BVorname AND l2.Bewertung < l.Bewertung)
- Gesucht sind alle Restaurants mit Name, Strasse und Suppenpreis, welche mindestens die Biersorten 'Sorte1' und 'Sorte2' im Sortiment haben (hier Malzdrink und Hopfdrink). Wir können nicht **direkt** zwei Mal das gleiche Element auf zwei Unterschiedliche eindeutige Wörter überprüfen (Malzdrink und Hopfdrink). Dabei kann niemals nie nicht TRUE herauskommen. Deshalb machen wir wieder einen Trick:
- SELECT r.name, r.strasse, r.suppenpreis
  FROM restaurant r, sortiment s WHERE r.name = s.rname AND bsorte = 'Malzdrink'
  AND EXISTS (SELECT 1 FROM restaurant r2, sortiment s2
  WHERE r2.name = s2.rname AND s2.bsorte = 'Hopfdrink')
- SELECT r.Name, r.Strasse, r.Suppenpreis
  FROM Restaurant AS r, Sortiment AS x, Sortiment AS y WHERE x.Rname = r.Name
  AND x.Bsorte = 'Malzdrink' AND y.Rname = r.Name AND y.Bsorte = 'Hopfdrink'
- Welche Besucher wohnen an einer Strasse, deren Bezeichnung das Wort ‚bach' enthält?
- SELECT name, vorname, strasse FROM besucher WHERE LOWER(strasse) LIKE '%bach%'

CASE

- Ermöglicht Fallunterscheidungen
- SELECT bname, bvorname, CASE
  WHEN SUM(frequenz) > 10 THEN ' ist ein Säufer' ELSE ' trinkt nicht so viel' END
  FROM gast GROUP BY bname, bvorname
- Allgemein:
  "CASE"
  "WHEN" <Bedingung1> "THEN" <Wert1>
  {"WHEN" <Bedingung2> "THEN" <Wert2>}
  ["ELSE" <Wert3>] "END"

NULL Revisited: einige nicht wünschenswerte Probleme

- Satz vom ausgeschlossenen Dritten:
- SELECT * FROM Besucher WHERE (Geburtstag = '1.4.1895') OR NOT (Geburtstag = '1.4.1895');
- SELECT * FROM Besucher;
  → Ergeben nicht zwingend das Gleiche
- Mengen Vergleiche SELECT A FROM S WHERE A NOT IN (SELECT A FROM T)
  Wenn $S \setminus T$ eine **leere Menge** gibt, können wir nicht sagen, ob $S \subseteq T$, da die Query mit NULL Werten in T immer die leere Menge zurück gibt.
- Leere Mengen SELECT SUM(suppenpreis) FROM Restaurant WHERE 0 = 1; → Ergibt NULL nicht 0
- Matrix Checksumme
- SELECT SUM(A) + SUM(B)
- SELECT SUM(A + B)

A	B
1	NULL
3	4

← 1 + NULL = NULL

↑ NULL + 4 = 4

- Horizontal wird aus einer Addition mit NULL → NULL
- Vertikal wird NULL ignoriert → Ergibt nicht das Gleiche
- **EXISTS und IN**
- WHERE S.A NOT IN (SELECT T.A FROM T)
- WHERE NOT EXISTS (SELECT 1 FROM T WHERE T.A = S.A)
  → Nicht das Gleiche, da Wahrheitswerte bei **IN UNKNOWN** sein **können**, jedoch **nicht bei EXISTS**
- **Durchschnittsberechung**
- SUM und AVG ignorieren NULL, COUNT nicht
  → Durchschnitt mit AVG <> Durchschnitt mit SUM und COUNT
- **Constraints** sind erfüllt, wenn immer diese TRUE oder UNKNOWN ergeben
- **Wird NULL immer gleichbehandelt?**
  - Ja, bei GROUP BY → Es entsteht eine Gruppe mit allen NULL-Werten
  - Nein, bei Vergleichen
- Viele weitere Unterschiede je nach Datenbanksystem

Tabellenausdrücke

- Tabellenausdrücke sind **benannte Abfragen**, die benutzt werden können wie Basistabellen (Tabellen, die in Datenbank gespeichert sind), die aber **nur** zur Ausführungszeit Daten generieren (nichtpersistent Daten).
- In SQL gibt es verschiedene Varianten von Tabellenausdrücken:
  - Abgeleitete Tabellen («derived tables»)
  - Sichten («views»)
  - Tabellenausdrücke (common table expressions, CTE's) (z.B. Funktionen, die Tabelle zurückgeben)
- Bei der Verwendung von **Tabellenausdrücken** muss sichergestellt werden, dass **alle Attribute** (auch **berechnete**) einen **Namen erhalten** (ggf. einen Alias verwenden) und diese **Namen** müssen **eindeutig** sein.

Abgeleitete Tabellen – «virtuelle» Tabellen

- Abgeleitete Tabellen sind Resultate von Abfragen, die **Teil einer anderen Abfrage sind**.
- Abgeleitete Tabellen müssen in der Regel einen (beliebigen, aber eindeutigen) Alias-Namen erhalten, über den auf die Attribute zugegriffen werden kann. Wenn die Abfrage berechnete Attribute enthält, sollten auch diese in der Regel einen Alias-Namen erhalten. Die **Abfragen muss** in **Klammern** stehen.
- Wenn aus Kontext klar ist, welche Attribute gemeint sind, können Alias-Namen weggelassen werden (nicht empfohlen).
- Abgeleitete Tabellen stehen nur zur Ausführungszeit der Abfrage, von der sie Teil sind, zur Verfügung. Man nennt sie darum auch «virtuelle» Tabellen.
- **Beispiel: Abgeleitete Tabelle** (nicht-korrelierte Unterabfrage) in der SELECT-Klausel
- Gesucht ist Name Restaurants, ihr Suppenpreis sowie durchschnittliche Suppenpreis von allen Restaurants
- SELECT r.Name, r.Suppenpreis,
  (SELECT AVG(Suppenpreis) FROM Restaurant) AS Durchschnittssuppenpreis
  FROM Restaurant r;
- **Beispiel: Abgeleitete Tabelle** (korrelierte Unterabfrage) in der FROM-Klausel
- Gesucht Restaurant, Suppenpreis & durchschnittliche Suppenpreis von Restaurants in derselben Strasse
- SELECT x.Name, x.Suppenpreis, y.Durchschnittspreis
  FROM Restaurant x JOIN (SELECT Strasse, AVG(Suppenpreis) AS Durchschnittspreis
  FROM Restaurant GROUP BY Strasse) AS y ON x.Strasse = y.Strasse;

Sichten

- Sicht: **Gespeicherte Abfrage**. Es wird nur die **Abfrage gespeichert**, **nicht** das **Resultat**.
- Sichten können in Abfragen überall dort verwendet werden, wo ein Tabellenname stehen kann. Man kann also auch «Sichten auf Sichten» bilden (siehe Beispiel).
- Sichten sind im allgemeinen nicht aktualisierbar, d.h. es sind keine DML-Operationen auf Sichten anwendbar. Es gibt je nach System zwar einzelne Ausnahmen, diese werden hier aber nicht betrachtet.
- **Erzeugen:** Sichten sind Datenbankobjekte, werden mittels **DDL-Anweisung** erzeugt. Syntax ist einfach:
- CREATE VIEW Name_der_Sicht AS Abfrage;
- **Verwendung (Beispiel):** SELECT * FROM Name_der_Sicht;
- Löschen: DROP VIEW Name_der_Sicht;
- Ändern: ALTER VIEW Name_der_Sicht …; → in der Regel einfacher: neu erstellen
- Man möchte beispielsweise Abfragen auf der Basis von Namen, Vornamen, Strasse, Geburtsdatum und Besuchsfrequenz von Gästen des Restaurants Ochsen (und nur von solchen Gästen!) machen. Die Information ist in der Datenbank in zwei Tabellen vorhanden:
- SELECT x.Name, x.Vorname, x.Strasse, x.Gebtag, y.Frequenz
  FROM Besucher x, Gast y
  WHERE x.Name = y.Bname AND x.Vorname = y.Bvorname AND y.Rname = 'Ochsen';
- **Definition als «Sicht»:**
- CREATE VIEW Ochsengast AS SELECT x.Name, x.Vorname, x.Strasse, x.Gebtag, y.Frequenz
  FROM Besucher x, Gast y
  WHERE x.Name = y.Bname AND x.Vorname = y.Bvorname AND y.Rname = 'Ochsen';
  SELECT * FROM Ochsengast; -- um sie anschliessend zu sehen
- Man kann nun mit dieser Sicht Abfragen tätigen wie mit einer «normalen» Tabelle:
- SELECT * FROM Ochsengast WHERE Vorname = 'Hans' AND Strasse = 'Bachweg';
- **Dieselbe Abfrage ohne Sicht:**
- SELECT x.Name, x.Vorname, x.Strasse, x.Gebtag, y.Frequenz
  FROM Besucher x, Gast y WHERE x.Name = y.Bname AND x.Vorname = y.Bvorname
  AND y.Rname = 'Ochsen' <u>AND x.Vorname = 'Hans' AND x.Strasse = 'Bachweg'</u>;
- Man kann auch «Sichten auf Sichten» definieren:
- CREATE VIEW MNamen AS SELECT Name FROM Ochsengast WHERE Name LIKE 'M%';

- Gesucht ist Liste von Besuchern mit Name, Vorname und Anzahl Restaurantbesuche pro Woche (= Frequenz). Falls ein Besucher nie Gast ist, soll er auf der Liste mit einer Anzahl Besuche von 0 erscheinen. Verwenden Sie dazu einen OUTER JOIN.
- SELECT name, vorname, SUM(COALESCE(frequenz, 0)) AS AnzahlBesuche
  FROM besucher b LEFT OUTER JOIN gast g ON b.name = g.bname AND b.vorname = g.bvorname
  GROUP BY b.name, b.vorname ORDER BY AnzahlBesuche DESC
- Gesucht ist eine Liste der Hersteller von Biersorten zusammen mit der Anzahl Biersorten, die sie produzieren und der Anzahl **verschiedener** dabei verwendeter Grundstoffe.
- SELECT hersteller, COUNT(*) AS AnzahlBiere, COUNT(DISTINCT Grundstoff) AS AnzahlGrundstoffe
  FROM biersorte GROUP BY hersteller
- Welche Biersorten sind von allen mit selben Note bewertet worden (das kleinste Note = grösste Note)?
- SELECT bsorte FROM lieblingsbier
  GROUP BY bsorte HAVING MIN(bewertung) = MAX(bewertung)
  → Auf HAVING können Aggregatsfunktionen angewendet werden, nicht aber auf WHERE

DDL Revisited

- CREATE TABLE Besucher2 (LIKE Besucher)
- Erzeugt neue Tabelle Besucher2 mit den gleichen Attributen wie Besucher (also vereinigungskompatibel)
- Die Tabellen sind vollständig entkoppelt, d.h., Änderungen in einer Tabelle wirken sich nicht auf andere aus
- Per Default werden **Constraints NICHT übernommen** (INCLUDING/EXCLUDING DEFAULTS)
- Erzeugen einer Tabelle mit Daten einer Query: CREATE TABLE <tableName> AS (<query>)
- Speichert die Resultate der Query in einer neuen Tabelle
- Übernimmt keine Constraints/Schlüssel etc.
- Beispiel: CREATE TABLE Besucher3 AS (SELECT * FROM Besucher WHERE name LIKE 'M%')

DML Reloaded

- Einfügen ganzer Resultattabellen: INSERT INTO <tableName> (<query>)
- Fügt die Resultate der Abfrage in die Tabelle ein.
- Das Abfrageresultat muss die geeigneten Domänen für die Attribute haben
- Beispiel: INSERT INTO Besucher2 (SELECT * FROM Besucher WHERE name LIKE '%a%')
- Modifizieren von Tupeln: UPDATE <tableName> SET <attributeName> = <attributeValue> {, <attributeName> = <attributeValue>} [WHERE <searchCondition>]
- **Modifiziert** die **Tupel**, welche durch die Suchbedingung "selektiert" werden
- Beispiel: UPDATE besucher SET gebtag = '1945-01-01' WHERE name = 'Meier' AND vorname = 'Hans'
- Probleme bei folgender Abfrage: UPDATE Besucher SET vorname = 'Jim' WHERE name = 'Meier'
  - Primärschlüssel ist {Name, Vorname}
  - Wenn nun mehrere Einträge «ansprechen» (weiterer Meier), wird Primärschlüssel-Constraint verletzt: ERROR: duplicate key violates unique constraint "besucher_pkey"
  - Analog kann es auch zu Problemen mit Fremdschlüssel-Constraints kommen
- **Löschen von Tupeln:** DELETE FROM <tableName> [WHERE <searchCondition>]
- Beispiel: DELETE FROM Besucher WHERE name = 'Meier'

Sichten – Aktualisierung

- Kann man Sichten auch aktualisieren?
- UPDATE Ochsengast SET Frequenz = 5 WHERE Name = 'Meier' AND Vorname = 'Hans'
- UPDATE Gast SET Frequenz = 5
  WHERE Bname = 'Meier' AND Bvorname = 'Hans' AND Rname = 'Ochsen'
- Auch wenn es in einzelnen Fällen möglich wäre (und bei einigen Systemen auch ist) Aktualisierungen via Sichten vorzunehmen, **geht** das **nicht allgemein**. Deshalb **Finger weg** von dieser Möglichkeit.
- Sichten können nie Daten liefern, die nicht schon in den Basistabellen grundsätzlich vorhanden sind (bzw. daraus berechnet werden können). Die vorhandenen Daten können aber sehr wohl anders dargestellt werden. Man könnte aber ja auch schreiben:
- CREATE TABLE Ochsengast(…)
- INSERT INTO Ochsengast SELECT x.Name, x.Vorname, x.Strasse, x.Gebtag, y.Frequenz FROM Besucher x, Gast y WHERE x.Name = y.Bname AND x.Vorname = y.Bvorname AND y.Rname = 'Ochsen'
- Im ersten Fall wird Abfrage einmal ausgewertet und Daten werden dauerhaft in Tabelle gespeichert.
- Im Falle einer Sicht wird zugrunde liegende Abfrage jedes Mal neu ausgewertet, wenn Sicht benutzt wird.
- Wenn die Basistabellen sich ändern, widerspiegeln Sichten immer den aktuellen Stand der Daten.
- Moderne Systeme bieten auch sogenannte «materialisierte» Sichten an. Dabei werden die Resultate einer Sicht dauerhaft gespeichert und bei einer Änderung der Basistabelle(n) automatisch nachgeführt.

Sichten – Vorteile

- **Verbergen von Komplexität:** kann Benutzern mit weniger ausgeprägten SQL-Kenntnissen «fertige» Abfragen zur Verfügung stellen.
- **Logische Datenunabhängigkeit:** Man kann Tabellen zusätzliche Attribute hinzufügen und die vorhergehende Version noch als Sicht zur Verfügung stellen.
- **Vereinfachung:** Man kann komplexe Abfragen schrittweise aufbauen.
- **Sicherheit:** Man kann Benutzern Leserechte auf Sichten erteilen, ohne dass sie Zugriffsrechte auf die zugrunde liegenden Tabellen haben müssen. Kann z.B. auch Zugriff nur auf einzelne Attribute bereitstellen.

Sichten – Nachteile

- **Performanz:** Die Sicht wird bei jeder Verwendung neu berechnet.
- **Keine Sortierung**: es ist kein ORDER BY in Sichten erlaubt. Gibt aber Umgehungsmöglichkeiten.
- **Wildwuchs:** Praxis «Wildwuchs» an Sichten, es braucht disziplinierte Organisation und Verwaltung davon

Common table expressions (CTE's)

- CTE ist – analog wie abgeleitete Tabelle oder Sicht – ein **temporäres Resultat** einer **Abfrage** auf das in einer anderen Abfrage Bezug genommen werden kann.
- Vieles, was man mit CTE's machen kann, kann man auch mit Unterabfragen, abgeleiteten Tabellen oder Sichten machen. Mit CTE's geht es jedoch oft einfacher.
- CTE's bieten zudem **exklusiv** die **Möglichkeit rekursive Abfragen** zu formulieren.
- Syntax: WITH cte_name (column_list) AS (CTE_query_definition) Abfage
- Nach Schlüsselwort WITH folgt CTE-Namen gefolgt von (optionalen) Namensliste, die den Ergebnisspalten eindeutige Namen zuweist. Danach folgt Schlüsselwort AS, das eigentliche Definition in Klammern einleitet.
- Die WITH-Anweisung ist **keine eigenständige** Anweisung, sondern muss von Abfrage gefolgt werden.
- WITH-Anweisungen können auch geschachtelt werden (korreliert oder unkorreliert):
- WITH query_name1 AS (SELECT ...), query_name2 AS (SELECT ... FROM query_name1 ...) SELECT ...
- Beispiel: Gesucht ist eine Liste von Restaurantnamen, den zugehörigen Suppenpreisen, sowie dem durchschnittlichen Suppenpreis aller Restaurants derselben Strasse.
- ```
  WITH DSPpS AS (SELECT Strasse, AVG(Suppenpreis) AS DurchschnittsPreis
    FROM Restaurant GROUP BY Strasse)
  SELECT Name, Suppenpreis, Durchschnittspreis FROM Restaurant JOIN DSPpS
    ON Restaurant.Strasse = DSPpS.Strasse
  ```

CTE's – Vorteile

- Kann grosse Queries gut **strukturieren**. Mehrere verschachtelte Unterabfragen sind schwer verständlich.
- Machen Abfragen **besser lesbar**, da die einzelnen Teilabfragen einen eigenen Namen haben.
- Bauen Abfrage so auf, dass sie der **menschlichen Denkweise entspricht**. Man beginnt mit den Teilabfragen, auf die dann in der Hauptabfrage bezuggenommen wird. Bei geschachtelten Abfragen sind die Teilabfragen irgendwo innerhalb der Hauptabfrage zu finden.
- Unterstützen rekursive Abfragen. Das geht mit geschachtelten Abfragen nicht.
  ```

Integritätsbedingungen, Datenbankprogrammierung

Integrität, Konsistenz

- Im Kontext relationalen Datenbanken: Integrität bedeutet gewisse **Annahmen über Zustand** der **Daten.**
- Man definiert dazu **Regeln**, die die **Daten einhalten müssen**. Diese Regeln nennt man:
  - **Integritätsbedingungen** bzw.
  - **Konsistenzregeln** (Konsistenzbedingungen).
- Wenn ein **Datenbestand** alle diese **Regeln** bzw. Bedingungen **erfüllt**, so **nennt** man ihn **konsistent.**
- **Achtung:** Konsistenz bedeutet **Übereinstimmung** Daten mit **Regeln**, bedeutet aber nicht Datenkorrektheit!

Integritätsbedingungen

- Durch Integritäts-/Konsistenzbedingungen will man verhindern, dass **falsche** Daten in Datenbank gelangen.
- Man kann durch solche Regeln nicht die Korrektheit der Daten sicherstellen, aber man kann Daten, die sicher falsch sind, von der Datenbank fernhalten.
- **Beispiel:** Erfassung des Geburtsdatums einer Person
  - 33.04.1978 ist sicher **falsch.** Kann durch RDBMS eindeutig erkannt werden (z.B. durch Wahl Datentyp).
  - 28.04.1978 kann **wahr oder falsch** sein. Das kann nicht mit einer Regel festgestellt werden.

Massnahmen zur Sicherstellung der Konsistenz in relationalen Datenbanken

Von relationalen Datenbankverwaltungssystemen durchgesetzt:

- **Bereichsintegrität:** Der Wert eines Attributes muss in bestimmten Wertebereich liegen. Sichergestellt durch Datentypen/Domänen sowie NULL bzw. NOT NULL.
- **Entitätsintegrität:** Der Primärschlüssel einer Tabelle muss eindeutig und immer vorhanden (nicht NULL) sein. Sichergestellt durch das RDBMS durch Definition einer nicht-leeren Attributmenge als Primary Key.
- **Referentielle Integrität:** Der Inhalt eines Fremdschlüssels muss entweder leer sein (d.h. NULL), oder genau ein Tupel mit einem solchen Schlüsselwert muss in der referenzierten Tabelle vorhanden sein.

Einschränkungen, constraints

Mit Einschränkungen bzw. Constraints werden Regeln bezeichnet, die die Menge der möglichen Datenwerte einschränken, die in eine Datenbank eingegeben werden können. Dazu zählen:

- **UNIQUE-Constraints:** Nebst Primär- und Fremdschlüsseln können weitere Schlüssel definiert werden.
- **CHECK-Constraints:** Es können Regeln definiert werden, die Aussagen über Attribute einer Tabelle (genauer: eines Tupels) festlegen.
  Beispiel: `CONSTRAINT ck_artikel_ekpreis_vkpreis CHECK (ekpreis >= 0 AND vkpreis >= ekpreis);`
- **DEFAULT-Constraints:** Es können Regeln definiert werden, welche Werte als Vorgabewerte verwendet werden sollen, falls für ein Attribut kein Wert geliefert wird.
  Beispiel: `CONSTRAINT df_auftrag_datum DEFAULT SYSDATETIME();`

Komplexere Geschäftsregeln

- Mit den bisher erläuterten Mechanismen und Regeln können zwar schon eine ganze Reihe von Bedingungen formuliert werden. Es gibt in der Praxis jedoch oft weitere, komplexere, Bedingungen, die auch erfüllt sein müssen. Man spricht in dem Zusammenhang oft von «**Geschäftsregeln**» oder «**business rules**».
- Insbesondere wenn Regeln definiert werden sollen, die Zusammenhänge herstellen zwischen Daten verschiedener Tabellen, reichen die bisherigen Möglichkeiten nicht aus.
  Beispiel: Kleinkredit nur gewähren, wenn Kunde innerhalb des letzten Jahres kein Kontoüberzug hatte.
- Dazu werden mächtigere Verfahren benötigt → Gespeicherte Prozeduren, Funktionen und Trigger

SQL als Programmiersprache?

- SQL ist eine **deskriptive** (**deklarative**) Sprache. Man formuliert in solchen Sprachen, was man haben möchte, aber nicht wie Ergebnis zustande kommt (Gegensatz zu imperativen Sprachen → Java, Python).
- Der DQL-Teil von SQL wurde ursprünglich dazu entwickelt, um **Abfragen** zu **formulieren**.
- Um komplexe Algorithmen oder GUI's u.a. zu implementieren, **braucht** man aber eine «**Turing-komplette**» **Programmiersprache** (Java, C, C++, Python, Scala, …).
- Solche Programmiersprachen bieten oft die Möglichkeit, SQL-Anweisungen «einzubetten», so dass beispielsweise aus einem Java/Python-Programm heraus auch SQL-Abfragen formuliert werden können. Man spricht in dem Zusammenhang dann von «**embedded SQL**».
- Man hat noch anderen Weg beschritten, indem man SQL um Möglichkeiten einer prozeduralen Programmiersprache «erweitert» hat, so dass auch mit SQL komplexere Algorithmen implementiert werden können.
- Diese Erweiterungen wurden allerdings erst spät standardisiert, was dazu führte, dass Hersteller seine eigenen Vorstellungen umgesetzten → es sind diverse proprietäre Spracherweiterungen entstanden, z.B. T-SQL («Transact-SQL) von Microsoft oder PL/SQL von Oracle oder eben auch PL/pgSQL von PostgreSQL.
- Sprachen sind untereinander **nicht kompatibel.** Die Konzepte sind aber überall ähnlich umgesetzt.
- Welche Möglichkeiten muss Programmiersprache bieten? Je nach Programmierparadigma (objektorientierte, funktionale, prozedurale, …) muss Sprache unterschiedliche Möglichkeiten bereitstellen.
- Bei SQL hat man Eigenschaften für eine (einfache) **prozedurale Sprache** hinzugefügt. Dazu gehören:

- **Variablen**
- **Ablaufkontrollstrukturen:** Verzweigungen («if-then-else») und Schleifen («while … do»)
- **Strukturierungsmöglichkeiten:** Prozeduren mit Parametern und Funktionen mit Rückgabewerten

Vorteile

- **Reduktion** des **Datenverkehrs** zwischen Client und DBMS.
- **Realisierung** sehr **komplexer Abfragen** möglich (die allein mit SQL nicht möglich wären).
- Verwendbar für verschiedene Anwendungen. (Kapselung von „business rules").
- Zugriffsoptimierung durch DBMS zur Erstellungszeit.
- **Sicherheit** (nur Ausführungsrechte für stored procedure benötigt anstatt Schreib-/Leserechte auf Tabellen/Sichten).
- Ermöglicht **Parametrisierung** von Abfragen.
- Änder-/erweiterbar ohne die Anwendungen anzupassen, die die Prozeduren und Funktionen nutzen.

Nachteile

- Syntax und Semantik sind nicht standardisiert, d.h. jeder DBMS-Hersteller hat seine eigene Sprache, z.B.:
  - Transact SQL (Microsoft)
  - PL/SQL (Oracle)
  - PSM/SQL (SQL3), nicht durchgesetzt -> Keine Portabilität
- Verwaltungsaufwand (müssen analog wie Softwareobjekte, aber in Betriebsumgebung, verwaltet werden).
- Fehlerbehandlung oft etwas umständlich.
- Oft keine «höheren» Strukturierungsmöglichkeiten (z.B. keine interne Prozeduren, Module, «Klassen», ...).
- Weniger komfortable Entwicklungsumgebungen als bei anderen Sprachen.

PL/pgSQL – Procedural Language/PostgreSQL

- PL/pgSQL ist SQL-Erweiterung von PostgreSQL zur Datenbankprogrammierung.
- PL/pgSQL sollte immer dann eingesetzt werden, wenn Datenzugriffe (lesend und/oder schreibend) im Vordergrund stehen. Effiziente Formen des Datenzugriffs sind die Stärke von PL/pgSQL.
- Mit PL/pgSQL können folgende Datenbankobjekte entwickelt werden:
  - Gespeicherte Prozeduren
  - Funktionen
  - Trigger

Functions bzw. Stored Procedures

- Menge von SQL-Anweisungen, die unter gemeinsamen Namen gespeichert und vom DBMS als Einheit ausgeführt werden.
- Stored procedures werden **in Datenbank abgelegt** und dort **ausgeführt** und können von Anwendungsprogrammen und/oder Benutzern aufgerufen werden.
- Functions / Stored procedures bieten Programmierkonstrukte wie benutzerdefinierte Variablen, Rückgabewerte und Kontrollflussanweisungen. Die Programmiersprache ist der „prozedurale Kitt" zwischen den nicht-prozeduralen (mengenorientierten) SQL-Anweisungen.
- Konzept entspricht teilweise klassischen Prozeduren und Funktionen in höheren Programmiersprachen.

Stored Procedures, Triggers: DDL

- Trigger/Stored Procedures werden wie Datenbankobjekte mittels SQL-DDL-Anweisungen bearbeitet:
  - CREATE {TRIGGER|PROCEDURE|FUNCTION} Test ...
  - ALTER {TRIGGER|PROCEDURE|FUNCTION} Test ...
  - DROP {TRIGGER|PROCEDURE|FUNCTION} Test
- Trigger und Stored Procedures werden im Systemkatalog abgelegt.

PL/pgSQL – Grundaufbau

- PL/pgSQL-Code besteht aus einzelnen «Blöcken»:

```
DECLARE -- Deklarationsblock - Der DECLARE Abschnitt ist optional
BEGIN -- Ausführungsteil
EXCEPTION -- Ausnahmeverarbeitung - Der EXCEPTION Abschnitt ist optional
END;
```

- **Beispiel: «Hallo Welt»:**
- ```
  CREATE OR REPLACE FUNCTION HalloWelt() RETURNS void AS $body$
  BEGIN
    RAISE NOTICE 'Hallo Welt';
  END;
  $body$ -- Ende des Funktionskörpers
  LANGUAGE plpgsql;
  ```
- `SELECT HalloWelt();`
- `DROP FUNCTION HalloWelt();`
  ```

PL/pgSQL – Variablen

- Variablen in PL/pgSQL: Analoges Konzept wie in anderen Programmiersprachen. Variablen sind benutzer-definierte Objekte, die einen **Datentyp haben** und in denen **Werte** während der Programmausführung **zwischengespeichert** und **abgerufen** werden können.
- Für Variablen in PL/pgSQL stehen dieselben Datentypen zur Verfügung wie für Attribute sowie zusätzliche.
- **Syntax:** name [CONSTANT] type [NOT NULL] [(DEFAULT | :=) expression];
- PL/pgSQL unterscheidet verschiedene Arten von Variablen, Beispiele:
- DECLARE
  KNr INTEGER; -- Alle SQL-Datentypen sind erlaubt
  ProdNr INTEGER := 0; -- mit Initialisierung (default: NULL) UserID users.UserID%TYPE; -- «Copying type»
  Zeile users%ROWTYPE; -- «Composite type», ganze Zeile -- einer Tabelle
  name RECORD; … -- «untypisierte Zeile»

PL/pgSQL – eingebaute Funktionen

- PL/pgSQL umfasst nützliche «eingebauten» Funktionen die direkt verwendet werden können.
- Funktion: Bezeichnung eines Programmkonstrukts, mit dem der Programm-Quellcode strukturiert werden kann, so dass Teile der Funktionalität des Programms wiederverwendbar sind. Das besondere Merkmal einer Funktion (im Vergleich zum ähnlichen Konstrukt der Prozedur) ist, dass die **Funktion** ein **Resultat direkt zurückgibt** und deshalb in **Ausdrücken verwendet werden kann**.
- Funktionen können Parameter haben und geben Skalarwerte oder Tabellen als Funktionswert zurück.
- SELECT CONCAT(Vorname,' ',Name) FROM Besucher;
- PL/pgSQL-Funktionen lassen sich in Gruppen gliedern:
  - Konfigurationsfunktionen: Geben Auskunft über den Zustand des Systems
  - Datum- und Zeitfunktionen: Funktionen zur Bearbeitung von Daten und Zeiten
  - Mathematische Funktionen
  - Zeichenfolgefunktionen
  - Metadaten-Funktionen
  - Konversions- / Datentypprüffunktionen
  - Sicherheitsfunktionen
  - Rangfolgefunktionen
- Kenntnis der vorhandenen Funktionen erleichtert Programmierung und Abfrageformulierung oft erheblich.

PL/pgSQL – Kontrollstrukturen

- Jede höhere Programmiersprache braucht Kontrollstrukturen, um komplexe Programmabläufe abzubilden, bei denen nicht sequenziell ein Befehl nach dem anderen abgearbeitet werden soll, sondern der Programm-ablauf durch Bedingungen und Wiederholungen gekennzeichnet wird. Man unterscheidet:
  - Auswahl- oder Entscheidungsstrukturen
  - Wiederholungsstrukturen («Schleifen»)
- PL/pgSQL kennt verschiedene Kontrollstrukturen:
  - IF … [ELSE | ELSIF …]
  - WHILE … LOOP … END LOOP
  - CASE … WHEN … ELSE … END
  - FOR … IN … LOOP … END LOOP
  - LOOP … EXIT

PL/pgSQL – Cursor

- Wichtiges Element in SQL-Programmierung. Sie bieten Möglichkeit, mehrere Zeilen eines Abfrageergebnisses zu verarbeiten. Cursor hat definierte Menge von Zeilen im Zugriff, innerhalb derer man sich vorwärts & rückwärts bewegen kann. Den jeweils aktuellen Datensatz kann man für bestimmte Operationen verwenden. Cursor dazu da bestimmte Aktion in gleicher Art/Weise auf mehrere Datensätze anzuwenden.
- Cursor sind also eine Art «**Schleifenvariablen**», die Zugriff auf einzelne Tupel einer Abfrage ermöglichen.
- Cursor sind nützlich, wenn gespeicherte Prozedur für jede Zeile v. Ergebnismenge aufgerufen werden soll.
- Bevor ein Cursor benutzt werden kann, muss er deklariert werden (DECLARE). Auf DECLARE CURSOR folgt immer SELECT-Anweisung, welche die Daten auswählt, die im Cursor enthalten sein sollen.
- Definition Cursor (vereinfacht): DECLARE cursor_name CURSOR FOR select_anweisung [FOR UPDATE];
- Auf deklarierten Cursor kann erst zugegriffen werden, nachdem er «geöffnet» wurde (ähnlich wie Dateien in Python zuerst geöffnet, bevor lesen werden können). Dies geschieht mit der OPEN-Anweisung:
- OPEN cursor_name;
- Erst beim Öffnen eines Cursors erfolgt der eigentliche Zugriff auf die zugrunde liegenden Daten.
- Daten werden aus einem Cursor über die Anweisung FETCH abgerufen. Dies geschieht Zeile für Zeile, wobei der Inhalt der abgerufenen Zeilen in lokalen Variablen gespeichert wird. Typischerweise wird aus einem Cursor mit der ersten Zeile beginnend eine nach der anderen bis zur letzten abgerufen.
- Die allgemeine Syntax für das Abrufen von Cursor-Zeilen lautet:
- FETCH cursor_name INTO @variable1, @variable2, @variable3, …;
- Wenn Cursor nicht mehr benötigt wird, so muss er geschlossen werden: CLOSE cursor_name;

Beispiel Cursor:

```
CREATE OR REPLACE FUNCTION Show_AlleBesuchernamen()
RETURNS VOID AS $$
DECLARE
  rec_Besucher record;
  c_Namen CURSOR FOR SELECT Name,Vorname FROM Besucher;
BEGIN
  OPEN c_Namen;
  LOOP
    FETCH c_Namen INTO rec_Besucher;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'Name: % Vorname: % ',rec_Besucher.Name, rec_Besucher.Vorname;

  END LOOP; CLOSE c_Namen;
END; $$
LANGUAGE plpgsql;
```

Trigger

- Programmcode, der vom DBMS ausgeführt wird.
- Auslösung Abarbeitung des Programmcodes durch DBMS aufgrund von sich ändernden Datenbankzuständen: INSERT, UPDATE, DELETE
- Gut geeignet für:
  - Realisierung von Integritätsbedingungen
  - Protokollieren von Datenänderungen
  - Berechnungen (z. B. das Nachführen von Summen)
  - vieles andere mehr!
- Vorteile: Entlastung der Anwendungsprogramme, effiziente Ausführung möglich, einfacher «rollout».
- Prinzip: ECA
  - Event – Condition – Action
  - ON Ereignis IF Bedingung DO Aktion
- Können geschachtelt (auch kaskadierend) aufgerufen werden.
- Trigger «hängen» an einer Tabelle.
- Werden in SQL(-Dialekt) formuliert, haben einen Namen:
  - CREATE Trigger ...
  - DROP Trigger ...
  - ALTER Trigger ... {DISABLE ¦ ENABLE}
- Auslösung (je nach System sind unterschiedliche Varianten vorhanden):
  - BEFORE vor der DML-Operation
  - AFTER nach der DML-Operation
  - INSTEAD OF anstelle der DML-Operation (gibt es nicht in PostgreSQL)
- Geänderte Werte werden in virtuellen Tabellen gespeichert:
  - SELECT * FROM NEW (liefert eingefügte und geänderte-neue Datensätze)
  - SELECT * FROM OLD (liefert gelöschte und geänderte-alte Datensätze)
- **Trigger Beispiel:** Funktion für Trigger erzeugen
-

```
CREATE OR REPLACE FUNCTION log_Strasse()
RETURNS TRIGGER LANGUAGE PLPGSQL AS $$
BEGIN
  IF NEW.Strasse <> OLD.Strasse THEN INSERT INTO
    Adressänderung(Name, Vorname, StrasseAlt, StrasseNeu, Geändert_am)
    VALUES(OLD.Name, OLD.Vorname, OLD.Strasse, NEW.Strasse, now());
  END IF;

  RETURN NEW;
END; $$

CREATE TRIGGER Strassenänderung BEFORE UPDATE ON Besucher -- Trigger erzeugen
FOR EACH ROW
  EXECUTE PROCEDURE log_Strasse();
```

- Einsatz von INSTEAD OF Triggern:
  - Verhindern unerlaubter DML-Operationen.
  - Wenn SQL-Constraints nicht ausreichen.
  - Nicht änderbare Sichten ‚änderbar' machen.
- Einsatz von AFTER Triggern:
  - Weitere Operationen sollen ausgelöst werden.
  - Berechnungen ausführen und speichern.
  - Historisierung.

- Einsatz sinnvoll, wenn:
  - Überprüfung/Funktion oft ausgeführt wird.
  - SQL-Constraints nicht ausreichen.
  - Logik von der DB und nicht von den Anwendungsprogrammen durchgeführt werden soll.
  - Lösung dadurch stark vereinfacht wird.
- Probleme:
  - Fehlerbehandlung
  - Testen, Debuggen
  - Gefahr der Unübersichtlichkeit
  - Ergebnis von Triggeroperationen evtl. abhängig von der Aufrufreihenfolge.
  - Terminierung von geschachtelten Triggeraufrufen.

Stored Procedures vs. Triggers

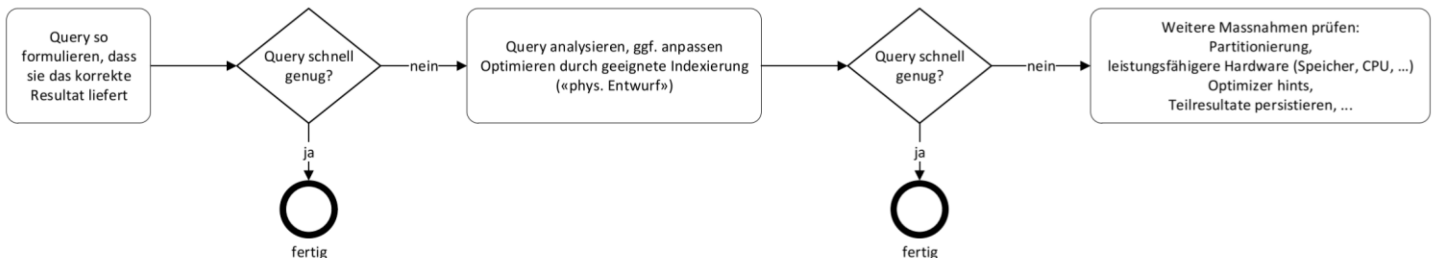
Stored procedures

- Aufruf (explizit):
  - Durch Benutzer oder Anwendungsprogramm
  - Transaktionskontrolle
- Einsatzbereiche:
  - Kapselung von «business rules»
  - Optimierung von Abfragen, Reduktion des Netzwerkverkehrs → Batch Processing
  - Erhöhte Sicherheit benötigt
- Probleme:
  - Fehlerbehandlung

Trigger

- Aufruf (implizit):
  - Durch DBMS, in Abhängigkeit Datenänderungen
  - Keine Transaktionskontrolle
- Einsatzbereiche:
  - Konsistenzsicherung (referentielle Integrität)
  - Logging
  - Nachführen von Tabellen
- Probleme:
  - Kompliziert zu testen
  - Aufrufreihenfolge nicht determiniert

Vorgehen beim Coden



Speicherhierarchie

- Tiefste Schicht (Betriebssystem) RDBMS geht um Sicherstellen **Persistenz** (dauerhafte Datenspeicherung)
- Es gab/gibt unüberschaubare Anzahl Speichermedien mit ganz unterschiedlichen Eigenschaften.
- Beispiele von Speichermedien: Harddisk, SSD, RAM, Tape, DVD, …
- Eine Klassifizierung hilft, Übersicht zu gewinnen: → Speicherhierarchie

Speicherhierarchie (grob)

- Primärspeicher (Bsp.: RAM, Cache)
  - Sehr schnell, aber teuer
  - Volatil (Inhalt nur zur Laufzeit des Systems vorhanden) → Für dauerhafte Speicherung ungeeignet!
- Sekundärspeicher (Bsp.: Harddisk, SSD, …)
  - Faktor $10^3 - 10^6$ langsamer als Primärspeicher!
  - Günstig
  - Persistent (Inhalt über die Laufzeit Systems hinaus vorhanden) → Für dauerhafte Speicherung geeignet!

Speicherhierarchie – Speichermedien

Grössenordnungen: → → → →

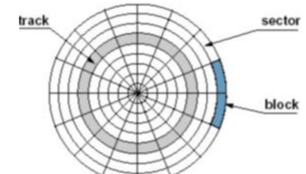
- Für persistente Datenspeicherung wichtigstes Medium: Harddisk (zunehmend SSD, sind aber teurer).
- Da auch RAM immer schneller wurde besteht diese Lücke auch noch beim Einsatz von SSD!
- Zugriffslücke SSD ~10^3

Speicherart	Typische Zugriffszeit	Typische Kapazität
Cache (L1, L2, …)	6 ns	256 kByte (L2), 8 MByte (L3)
Hauptspeicher	60 ns	Bis 512 GByte
	Zugriffslücke ~10^5	
Magnetplatten	8-12 ms	Bis 22 TByte
Platten-Array	12 ms	TByte- bis PByte-Bereich

Speicherhierarchie

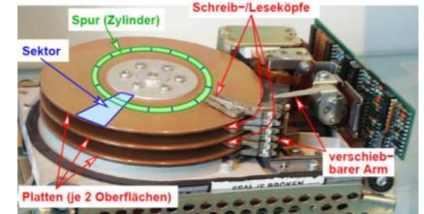
- Tertiärspeicher (Bsp.: Tape, optische Medien, …)
  - Langsam, billig, hohe Kapazität
  - Nicht permanent direkt zugreifbar
- Persistent (Inhalt über die Laufzeit des Systems hinaus vorhanden)
  → Für dauerhafte Speicherung geeignet! → Für Datenbanken aber ungeeignet (viel zu langsam)!
- Ein RDBMS verwendet eine Kombination von Primär- und Sekundärspeicher (RAM und meist Harddisk). Was immer möglich, soll im RAM erledigt werden. Zugriffe auf die Sekundärspeichermedien sollten nur so selten wie nötig erfolgen, sind aber nötig für die Persistenz.
- Wie werden die Daten auf einem Sekundärspeichermedium gespeichert?

Speicherhierarchie – Magnetplatten



Dominantes Speichermedium für Datenbanken. Begriffe:

- Platte, Plattenstapel (Disk)
- Spur (Track)
- Zylinder (übereinander liegende Spuren)
- Sektor (kleinster Teil einer Spur = Spur-Sektor, oder Platten-Sektor)
- Block (Cluster) (mehrere Sektoren einer Spur hintereinander)
- Aufgrund mechanischer Komponenten sind Magnetplatten «langsam».



Zugriff auf Magnetplattenspeicher

- Daten werden immer «seitenweise» gelesen und geschrieben. Das Suchen eines Tupels innerhalb einer Seite erfolgt sehr rasch im RAM.
- Zwischen Hauptspeicher und Magnetspeicher werden immer ganze Seiten übertragen. Eine Seite in PostgreSQL entspricht 8192 Bytes. Man spricht auch von «Blöcken» statt Seiten.
- Magnetscheiben müssen vorab entsprechend «positioniert» werden.

Abbildungen SQL ↔ Plattenspeicher

- Auf logischen Ebene des Relationenmodelles (auch SQL-Schnittstelle) sind Daten in Tabellen organisiert.
- Diese logische Datenorganisation muss durch das RDBMS (unter Mitwirkung des Betriebssystems) auf eine Folge von Seiten (= «verkettete Liste») abgebildet werden, und zwar so, dass möglichst wenige Lese- bzw. Schreibzugriffe auf die Sekundärspeichermedien nötig sind.
- Wie kann man nun die Datenspeicherung möglichst optimal umsetzen?
- Beispiel soll Grundideen für Datenbanksysteme wesentlichsten Datenorganisationsformen erläuteren.

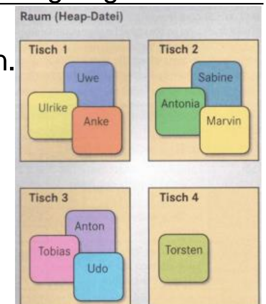
Anforderungen an Speichertechniken

- Effizienz beim Einzelzugriff (Schlüsselsuche beim Primärindex).
- Effizienz beim Mehrfachzugriff (Schlüsselsuche bei Sekundärindexen).
- Effizienz beim sequenziellen Durchlauf (Sortierung, geclusterter Index).
- Unterstützung für verschiedene Anfragetypen: –
  - exact-match
  - partial-match
  - range queries (Bereichsanfragen)
- Dynamisches Verhalten, d.h. Anpassung an sich ändernde Datenmengen.

Datenorganisation bei Datenbanken

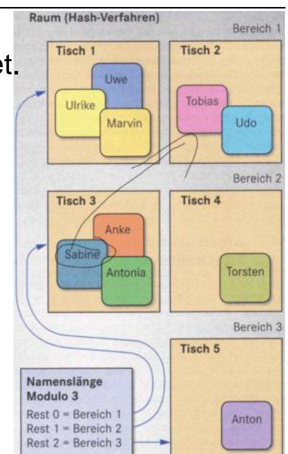
- Ausgangslage: Ein Kinder-Malwettbewerb.
- Folgende zehn Kinder haben am Wettbewerb teilgenommen:
  Uwe, Ulrike, Anke, Sabine, Antonia, Marvin, Anton, Tobias, Udo, Torsten
- Wie kann nach dem Malwettbewerb die Rückgabe der Bilder am geschicktesten organisiert werden?
- Für Ausgabe der Bilder wurde ein Raum zur Verfügung gestellt. Im Raum befinden sich viele Tische («Seiten»), auf denen Bilder ausgebreitet werden. Auf jedem Tisch können bis zu drei Bilder abgelegt werden.

Datenorganisation: HEAP-Datei



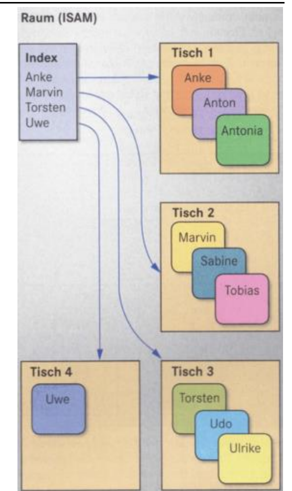
- Bei Heap-Datei werden Datensätze unsortiert hintereinander (sequenziell) geschrieben.
- Im Beispiel wurden Tische einfach hintereinander aufgestellt und Zeichnungen darauf unsortiert ausgelegt.
- **Vorteil:** Das Auslegen der Zeichnungen ist sehr einfach.
- **Nachteil:** Wenn ein Kind seine Zeichnung abholen möchte, muss es alle Tische nacheinander absuchen bis es Zeichnung gefunden hat. Dies heisst «Tablescan».
- **Fazit:** Die einfache Heap-Datei ist für viele Abfrageoperationen schlecht geeignet, da alles durchsucht werden muss. Das Auslegen («Anfügen von Daten») geht schnell.

Datenorganisation: HASH-Verfahren



- Bei HASH-Verfahren werden die Datensätze unterschiedlichen Bereichen zugeordnet. Innerhalb Bereiche sind Datensätze wieder ungeordnet.
- Zuordnung Datensätze zu Bereichen erfolgt über «Hashfunktion».
- Im Beispiel werden Bilder anhand der Länge der Kindernamen mit Hilfe der Modulo 3 Hash-Funktion auf drei Bereiche aufgeteilt.
- Bem: Die Darstellung enthält noch einen Fehler, welchen?
- **Vorteil:** Das Auslegen der Zeichnungen ist immer noch einfach. Das Durchsuchen nach dem Namen ist zudem extrem schnell.
- **Nachteil:** Die einzelnen Tische eines Bereiches müssen sequentiell durchsucht werden. Wenn Tisch voll, müssen weitere Tische hinzugefügt werden. Für Suchen nach Bereichen («von – bis»), ist diese Datenorganisation nicht geeignet.
- **Fazit:** Verfahren ermöglicht einfaches Einfügen von Daten. Löschen und Lesen von Daten ist im Vergleich zu Heap-Datei schneller. Nachteil: sortierte Ausgabe Daten nicht unterstützt wird.

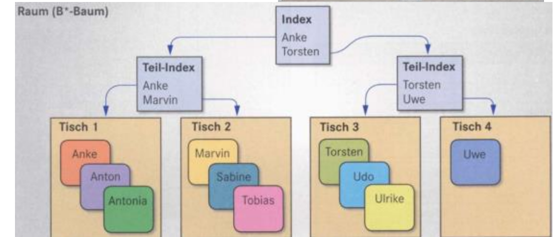
Datenorganisation: ISAM-Verfahren (Index Sequential Access Method)



- Es wird neben eigentlichen Datendatei, die Datensätze in sortierter Reihenfolge enthält, eine weitere Datei, die sog. Index-Datei gepflegt. Diese Index-Datei ermöglicht beschleunigten Zugriff auf Daten.
- Im Beispiel werden Zeichnungen alphabetisch sortiert ausgelegt. Über alphabetisch ersten Name wird «Index» erstellt.
- **Vorteil:** Der richtige Tisch wird über Index bestimmt. Die alphabetische Sortierung erleichtert die Suche auf dem Tisch.
- **Nachteil:** Auslegen der Bilder ist wegen Sortierung sehr aufwändig. Die Indexdatei muss immer wieder angepasst werden.
- **Fazit:** Durch Sortierung der Datensätze und der Indexdatei wird Lesen von Daten deutlich effizienter. Das Einfügen und Löschen von Daten werden aber aufwändiger.

Datenorganisation: B*-Baum-Verfahren

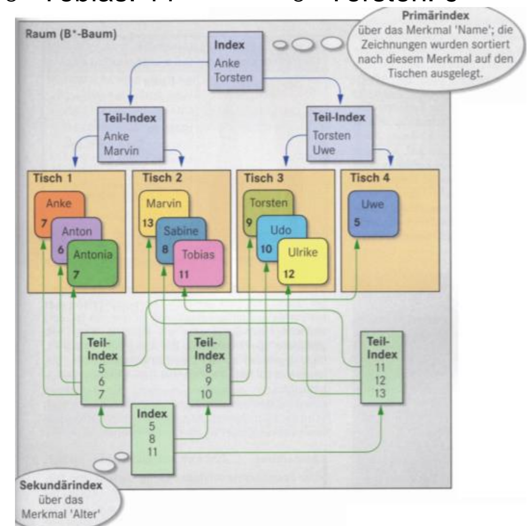
- Betrachtet man Indexdateien des ISAM-Verfahrens wiederum als Datensätze, so kann zu Datensätzen wiederum Indexdatei generiert werden. Durch «Verschachtelung» von Indexdateien kommt man zu «baumartigen» Struktur.



- Merkmale eines B*-Baumes:
  - Jeder Knoten kann Kindknoten haben (Mehrwegbaum).
  - Der Baum ist ausgeglichen (balancierter Baum).
  - Knoten enthalten nur Verweise und nicht eigentlichen Daten.
- Zeichnungen werden wieder sortiert auf Tischen ausgelegt. Dann werden mehrere Teilindizes erzeugt, die mehrere benachbarte Tische umfassen. Über Teilindizes wird übergeordnete Indexdatei angelegt.
- Gibt zahlreiche Varianten (B-Baum, B+-Baum, B*-Baum, …).
- **Vorteil:** Es entstehen keine «übergrossen» Indexdateien.
- **Nachteil:** Relativ aufwändige Einfüge- und Löschoperationen.
- **Fazit:** Optimale Lösung für Anwendungen mit überwiegend lesendem Zugriff.

Datenorganisation: Sekundärindex

- Ein Sekundärindex ist ein Index (beispielsweise ein B*-Baum) für ein zusätzliches Merkmal.
- Im Beispiel wird Sekundärindex für das Alter der Kinder erzeugt. Dabei haben die Kinder folgende Alter:
  - Uwe: 5
  - Anke: 7
  - Antonia: 7
  - Anton: 6
  - Udo: 10
  - Ulrike: 12
  - Sabine: 8
  - Marvin: 13
  - Tobias: 11
  - Torsten: 9



- **Vorteil:** Effizienz von lesenden Zugriffen bezüglich des zusätzlichen Merkmals können erheblich gesteigert werden. Der Primärindex bleibt unverändert.
- **Nachteil:** Einfüge- und Löschoperationen werden nochmals komplizierter, da mehrere Indizes angepasst werden müssen.
- **Fazit:** Optimale Lösung zur Beschleunigung von lesenden Zugriffen auf weitere Merkmale. Primärindex bleibt unbeeinflusst.
- **Beobachtung:** Gibt keine für alle Zwecke optimale Organisation!

Indexarten

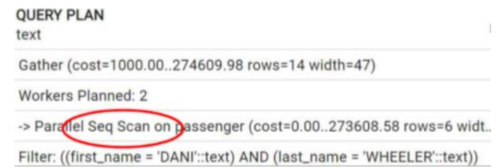
- Im Kontext von RDBS bezeichnen Indexe vollständig redundante Datenstrukturen, die dazu dienen, DQL-Anweisungen zu beschleunigen (DML-Anweisungen können u. U. auch von Indexen profitieren, steht aber meist nicht im Vordergrund). Indexarten:
- **Geclusterte, nicht geclusterte Indexe:**
  - Geclustert: Daten sind nach dem Indexkriterium physisch sortiert gespeichert.
  - Ungeclustert: Daten sind nicht nach dem Indexkriterium physisch sortiert gespeichert.
- **Primär- / Sekundärindexe:**
  - Primärindex (es gibt höchstens einen, warum?): Geclusterter Index nach den Primärschlüsselattributen.
  - Sekundärindexe: Weitere (ungeclusterte) Indexe.
- **Dicht-besetzte/dünn-besetzte Indexe:**
  - Dicht-besetzt: Für jedes Tupel ist ein Eintrag in der Indexdatei vorhanden.
  - Dünn-besetzt: Nicht für jedes Tupel ist Eintrag in Indexdatei vorhanden. Nur geclusterte Index möglich.
- Abdeckende Indexe: Index enthält nebst den Suchattributwerten noch weitere Attributwerte.
- Partielle Indexe: Index «deckt» nicht alle Tupel ab.
- Ein-Attribut, Mehr-Attribut-Indexe.
- In RDBMS haben sich zur Implementation hauptsächlich verschiedene Arten von B-Bäumen bewährt.
- Die Möglichkeiten zur Indizierung sind stark vom verwendeten RDBMS abhängig.

Indexe – Erstellung

- Indexe können auf zwei Wegen entstehen:
  - Durch das **RDBMS selbst**
  - Manuell **durch DB-Fachleute**
- RDBMS (gilt für PostgreSQL, andere verhalten sich ggf. anders):
  - RDBMS erstellt **automatisch geclusterten Index**, wenn **Primärschlüssel** definiert wird (DDL).
  - Für sonstige Schlüssel (**UNIQUE**) werden **nicht geclusterte Indexe** erstellt (DDL).
  - Zur Betriebszeit einer DB erstellt das RDBMS aber **selbständig keine weiteren Indexe**.
- DB-Fachleute: Manuelles Erstellen von weiteren Indexen **nach Bedarf**.

Ausführungspläne

- «Optimierer» (Teil eines RDBMS) entscheidet, wie eine Query berechnet wird (Reihenfolge der Operatoren, Auswahl der Verfahren, …). Ein solcher Entscheid nennt man einen Ausführungsplan. Diesen kann man sich in PostgreSQL mit dem Befehl EXPLAIN anzeigen lassen.
- **Beispiel** (postgres_air): gesucht sind Passagierangaben für Passagiere mit dem Namen 'Dani Wheeler':
- SELECT * FROM passenger WHERE first_name = 'DANI' AND last_name = 'WHEELER';
- **Ausführungsplan:** EXPLAIN SELECT * FROM passenger WHERE first_name = 'DANI' AND …
- Ausführungspläne sind anspruchsvoll zu lesen (und die Darstellung ist leider abhängig vom gewählten RDBMS). Im vorherigen Beispiel sieht man auch ohne Detailkenntnisse, dass Index auf Attributen first_name + last_name vermutlich Abfrage beschleunigen würde.


QUERY PLAN
text
Gather (cost=1000.00..274609.98 rows=14 width=47)
Workers Planned: 2
-> Parallel Seq Scan on passenger (cost=0.00..273608.58 rows=6 widt..
Filter: ((first_name = 'DANI'::text) AND (last_name = 'WHEELER'::text))

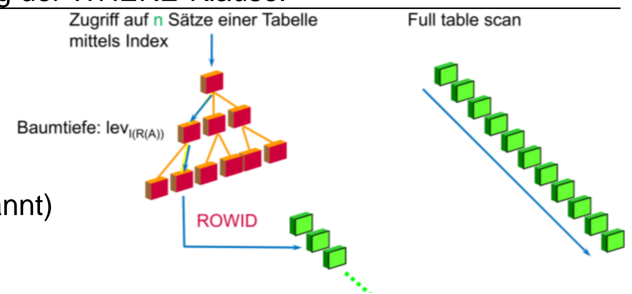
- Um Indexe zu erstellen gibt es die DDL-Anweisung CREATE INDEX:
- CREATE INDEX Name_des_Index ON Tabelle(Attribut(e));
- CREATE INDEX IX_Passenger ON Passenger(first_name,last_name);
- Ausführungsplan neu: → → →
- Ausführungszeiten (geht hier nur um Grössenordnung):
  - Query ohne Index: 6 Sekunden
  - Index erstellen: 1 Minute 9 Sekunden
  - Query mit Index: 150 Millisekunden

QUERY PLAN
text
Index Scan using ix_passenger on passenger (cost=0.43..60.71 rows=14 width=4...
Index Cond: ((first_name = 'DANI'::text) AND (last_name = 'WHEELER'::text))

Indexierung

- Index **wichtig Hilfsmittel**, um Abfrageausführung zu beschleunigen, allerdings müssen sie richtig einsetzen
- Da Indexe Abfragen signifikant beschleunigen können, ist Versuchung gross, dieses **Mittel oft** einzusetzen.
- Im Rahmen von DML-Anweisungen muss Datenbank diese **Redundanzen** aber alle **konsistent** halten. Man spricht vom **Update ↔ Query-Tradeoff**, d.h. Nutzung **abwägen** durch Entwickler/DBA!
- Bemerkung: DML-Anweisungen selbst profitieren nur in eingeschränktem Masse selbst von Indexen:
  - Insert: Gar nicht
  - Update, Delete: Eventuell, wenn nutzbar zur Auswertung der WHERE-Klausel

Wann «lohnt» sich ein Index?



Beispielannahmen zu den «Kosten»:

- Baumtiefe sei 3 (jeder Level erfordere einen Seitenzugriff)
- Anz. Sätze pro Seite: 6
- Abschätzung:
- s = Anzahl Seiten (unbekannt), r = Anzahl Tupel (unbekannt)
- n = Anzahl Tupel, die gelesen werden
- Beispielannahmen:
  - $\frac{r}{s} = 6$ Sätze pro Seite,
  - Baumtiefe = 3, d.h. für jedes Tupel müssen 4 Seiten gelesen werden (3 Indexseiten + 1 Datenseite)
- Damit der Index «besser» ist (unter den Beispielannahmen):
  - $lev_{I(R(A))+1} \cdot n < s$ → bei einem Scann müssen ja s Seiten gelesen werden
  - $n < \frac{s}{lev_{I(R(A))+1}}$
  - $\frac{n}{r} < \frac{s}{(lev_{I(R(A))}+1) \cdot r}$ gesucht $\frac{n}{r}$
  - $\frac{n}{r} < \frac{1}{(lev_{I(R(A))}+1) \cdot \frac{r}{s}}$ im Bsp. $\frac{n}{r} < \frac{1}{4 \cdot 6}$
  - Damit sich Abfrage unter diesen Umständen lohnt, muss sie weniger als $\approx 4\,\% = \left(\frac{1}{24}\right)$ der Sätze liefern, (sonst wäre Tablescan schneller).

1. Attribute, die oft abgefragt werden, sollten indiziert werden.
2. Fremdschlüssel indexieren, insbesondere, wenn über «Primär-Fremdschlüssel» gejoint wird (häufig so).
3. Attribute über die oft gejoint wird; wenn mehrere Attribute gejoint wird → dann zusammengesetzter Index.
4. Attribute mit niedriger Kardinalität (Extrembeispiele: Geschlecht, Ja/Nein-Flags u.ä.) sollten nicht indexiert werden (es gibt dafür spezielle Indexstrukturen, hier aber nicht behandelt).

➔ Achtung: **Überindexierung kostet Ausführungszeit** und Speicherplatz und **kann** den **Optimizer in** die **Irre führen** (d.h. Abfragen können sogar langsamer werden, siehe vorheriges Beispiel).

Transaktionsverarbeitung

Wichtigste Anforderungen an Transaktionsverarbeitung:

1. **Atomarität (Atomicity) / Unit of Work:** Zusammengehörige Folge von Lese- und Schreibzugriffen (in sich geschlossene „Arbeitseinheit"), muss als Ganzes entweder erfolgreich abgeschlossen (Commit) oder rückgängig gemacht werden können (Rollback).
2. **Consistency / Konsistenz:** Alle Operationen hinterlassen die Datenbank in einem konsistenten Zustand.
3. **Isolation / Nebenläufigkeit (Concurrency):** «Gleichzeitiger» Zugriff mehrerer Benutzer ermöglichen, so dass diese Transaktionen keinen unerwünschten Einfluss aufeinander haben (bei nur einer CPU auch möglich → Gleichzeitigkeit wird durch Zeitscheibenverfahren «simuliert»).
4. **Dauerhaftigkeit (Durability) / Recovery:** Automatische Behandlung von Ausnahmesituationen (Fehlern) und schneller (möglichst automatischer) Wiederanlauf nach schwerwiegenden Fehlern. Wiederherstellung (Rollforward) verlorener Daten und Rücksetzten (Rollback) fehlerhafter Daten.

Nebenläufigkeit: Lost-Update-Problem

Lost-Update: Überschreiben bereits getätigter Updates

- Keine Isolation der Transaktionen beider Benutzer → Update-Operation von Benutzer 1 geht verloren!
- Lösung: Durch andere Transaktion gelesene Daten dürfen bis zur deren Beendigung nicht verändert werden.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2		SELECT Wert INTO W FROM Tbl
3	UPDATE Tbl SET Wert = 100	
4		UPDATE Tbl SET Wert = 200
5	SELECT Wert INTO W FROM Tbl	
6		SELECT Wert INTO W FROM Tbl

Nebenläufigkeit: Dirty-Read-Problem

Dirty-Read: Lesen von Veränderungen noch nicht abgeschlossener Transaktionen

- Keine Isolation der Transaktionen beider Benutzer → Benutzer 2 arbeitet mit einem nicht bestätigten Wert, der später sogar zurückgenommen wird!
- Lösung: Nur Updates bestätigter Transaktionen lesen.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2	UPDATE Tbl SET Wert = NeuerWert	
3		SELECT Wert INTO W FROM Tbl
4	ROLLBACK	
5		UPDATE Tbl SET Wert = W + 1
6		SELECT Wert INTO W FROM Tbl
7	SELECT Wert INTO W FROM Tbl	

Nebenläufigkeit: Non-Repeatable-Read-Problem

Non-Repeatable-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 erhält nicht immer denselben Wert!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt.

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2		UPDATE Tbl SET Wert = Wert + 5
3		COMMIT
4	SELECT Wert INTO W FROM Tbl	

Nebenläufigkeit: Phantom-Read-Problem

Phantom-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 sieht Zwischenresultate, die von Benutzer 2 eingefügt wurden!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT count(*) INTO cnt FROM Tbl	
2	N = cnt	
3		INSERT INTO Tbl VALUES (…)
4		COMMIT
5	SELECT count(*) INTO cnt FROM Tbl	
6		

Nebenläufigkeit in der Praxis

Zusammenfassend:

- Lost Update: kann in einem Transaktionssystem **fast nie toleriert werden**.
- Dirty-, Non-Repeatable-, Phantom-Read: bei konkurrentem Lesen oft zu gewissen Ausmass **tolerierbar**

Isolationsebenen im SQL-Standard

- Isolationsebenen im SQL-92 Standard SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | …}
- Isolationsebenen vs. Phänomene:
- Postgres: READ UNCOMMITTED = READ COMMITTED

Isolationsebene	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update
READ UNCOMMITTED	~~möglich~~ (nicht in Postgres)	möglich	möglich	~~möglich~~ (nicht in Postgres)
READ COMMITTED (Häufig in der Praxis)	verhindert	möglich	möglich	verhindert
REPEATABLE READ	verhindert	verhindert	~~möglich~~ (nicht in Postgres)	verhindert
SERIALIZABLE	verhindert	verhindert	verhindert	verhindert

Transaktionen in der Praxis

- Anwendungsentwickler muss lediglich Transaktions**grenzen** festlegen, wird aber von allen anderen Aspekten zur Realisierung der Garantien befreit:
  - **COMMIT:** Transaktionsresultat wird **permanent.**
  - **ROLLBACK, ABORT: Annullierung** aller Veränderungen, Unsichtbarkeit für andere Transaktionen.
- Transaktionsanweisungen in SQL:
  - BEGIN TRANSACTION (implizit und/oder explizit)
  - COMMIT TRANSACTION
  - ROLLBACK TRANSACTION
- Wenn Transaktionen durch Fehler beendet, Sitzung abbrechen, durch Systemabsturz oder durch Verlassen der Sitzung ohne Commit, so entsteht inkonsistenter Zustand, der durch Recovery behoben werden **MUSS**.
- DDL-Anweisungen / je DBMS («Transactional DDL»):
  - Einbettung in laufende Transaktion (z.B. Postgres, SQL-Server)
  - Kapselung in eigener Transaktion (z.B. Oracle; die laufende Transaktion wird committet!)
- Constraint-Verletzung / je DBMS:
  - Laufende Überprüfung und sofortiger Rollback (Postgres, Oracle, SQL-Server).
  - «deferred constraint checking»: Constraints werden am Ende der Transaktion überprüft (Postgres, Oracle). Postgres erlaubt kein «deferred» für NOT NULL und CHECK-Constraints. SQL Server kennt keine deferred constraints.

Zustand DB bei offener Transaktion:

- vorhergehender Zustand wird in **Before-Images** festgehalten und kann damit wieder hergestellt werden.
- betroffenen Sätze sind mit Schreib-/Lese-Sperren blockiert; können in anderen Transaktionen nicht gelesen/geändert werden.
- andere Transaktionen sehen geänderte Daten nicht (ausser bei Uncommitted Read – nicht Postgres!).

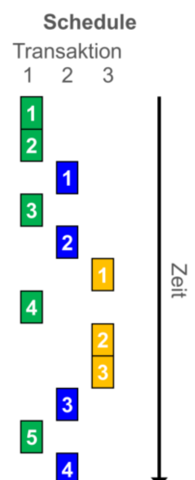
Zustand DB nach COMMIT:

- Die geänderten Daten sind in der Datenbank festgeschrieben.
- Alle Datenänderungen - alter Zustand (Before-Images) und neuer Zustand (After-Images) - sind in den Transaktionslogs protokolliert.
- Vorhergehender Zustand kann nicht mehr mittels ROLLBACK wieder hergestellt werden. Aber wie sonst?
- Alle Sperren der Transaktion sind frei gegeben.
- Geänderten Daten können in Transaktionen mit «Read Committed Isolation» oder in nachfolgenden Transaktionen gelesen werden.
- Geänderte Daten können in anderen Transaktionen geändert werden (falls nicht weitere Transaktionen Sperren gesetzt haben).

Zustand DB nach ROLLBACK:

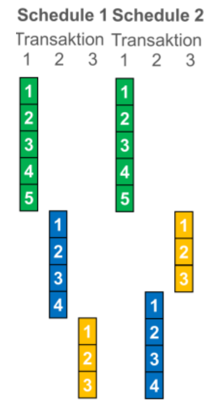
- Geänderten Daten sind vollständig zurückgesetzt.
- Herstellen des alten Zustandes durch Einsetzen der Before-Images.
- Rollback-Vorgang selbst wird auch in den Transaktionslogs aufgezeichnet.
- Sperren der betroffenen Daten sind frei gegeben.
- Daten können in anderen Transaktionen gelesen und geändert werden.

Schedules



- **Schedule (Ablaufplan):**
  - Folge von Lese-/Schreiboperationen für die (parallele) Ausführung einer oder mehrerer Transaktionen.
  - Schedules können ACID-Eigenschaften verletzen.
- **Vollständiger Schedule = History**
  - Sämtliche Schritte aller anstehenden Transaktionen inkl. Terminierung (COMMIT, ABORT)
  - Für jede Transaktion ist festgehalten, ob sie erfolgreich endet oder abbricht (dies kann in einem Schedule noch offen sein).
- Darstellung Lesen und Schreiben: $r_n(x)$ bzw. $w_n(x)$ (r = read, w = write, x = DB-Objekt, n = Transaktionsnummer)
- Darstellung Terminierungsoperationen: c_n = commit, a_n = abort
- Beispiel mit drei Transaktionen $T = \{t_1, t_2, t_3\}$:
  - $t_1 = r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)c_1$
  - $t_2 = r_2(x)w_2(x)w_2(y)w_2(z)c_2$
  - $t_3 = w_3(x)r_3(y)w_3(z)c_3$
  - Lost-Update Schedule; $w_1(x)$ geht verloren: $s_{lu} = r_1(x)r_2(x)w_1(x)w_2(x)$ …
  - Vollständiger Schedule (History):
    $s_{vs} = r_1(x)w_1(x)r_2(x)r_1(y)w_2(x)w_3(x)w_1(y)r_3(y)w_3(z)w_2(y)r_1(z)c_1c_3w_2(z)c_2$

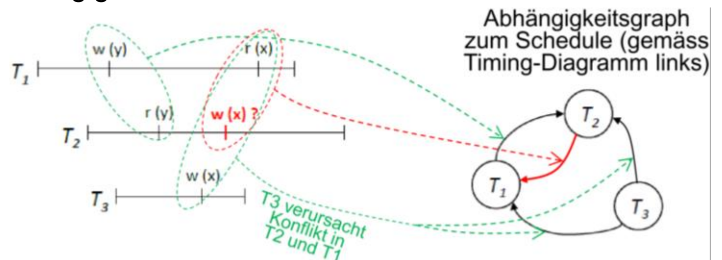
Serieller Schedule:



- Alle Transaktionen nacheinander.
- Für n Transaktionen existieren $n!$ verschiedene serielle Schedules.
- Serielle Schedules werden als konsistenzerhaltend betrachtet.
- Sicher, aber **schlechte Performance** → verschränkte Ausführung ist erwünscht.
- Serialisierbarer Schedule ist auch konsistenzerhaltend.
- Konfliktserialisierbar: Hat denselben Effekt auf die Datenbank, wie einer der seriellen Schedules. Hierzu existieren effiziente Algorithmen.

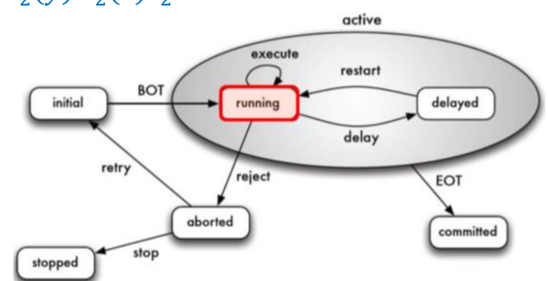
Serialisierbarer Schedule

- Wann ist Schedule **konfliktserialisierbar**? (nachfolgend nur noch von serialisierbar)
- Abhängigkeit (Konflikt) besteht, wenn zwei Transaktionen auf dasselbe Objekt mit reihenfolgeabhängigen Operationen zugreifen. Konfliktarten:
  - Schreib-/Lese-Konflikt $w_1(x)r_2(x)$
  - Lese-/Schreib-Konflikt $r_1(x)\ w_2(x)$
  - Schreib-/Schreib-Konflikt $w_1(x)\ w_2(x)$
- Ein Schedule s ist nun konfliktserialisierbar, falls serieller Schedule existiert, dessen Konflikte identisch sind.
- Nachweis Serialisierbarkeit: Führen von zeitlichen Abhängigkeiten zwischen Transaktionen in einem Abhängigkeitsgraphen (Konfliktgraphen).
- Man kann zeigen, dass ein Schedule serialisierbar ist (d.h. er führt zum selben Resultat wie ein beliebiger serieller Schedule), wenn der Abhängigkeitsgraph **keine Zyklen** enthält (in $O(n^2)$ entscheidbar):
- Dieser Schedule ist nicht serialisierbar, da der Graph einen Zyklus aufweist.



Scheduler / Transaktionsmanager Scheduler:

- Erzeugt serialisierbaren Ablaufplan s für parallel auszuführende Transaktionen T → welche Transaktion macht den nächsten Schritt?
- Verwaltet notwendige Synchronisationsinformationen, z.B. die Lese- und Schreibsperren
- Beispiel mit drei Transaktionen $T = \{t_1, t_2, t_3\}$:
  - $t_1 = r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)c_1$
  - $t_2 = r_2(x)w_2(x)w_2(y)w_2(z)c_2$
  - $t_3 = w_3(x)r_3(y)w_3(z)a_3$
  - $s = r_1(x)w_1(x)w_3(x)r_1(y)r_2(x)w_2(x)w_1(y)r_3(y)r_1(z)c_1w_3(z)a_3w_2(y)w_2(z)c_2$
- Eine Transaktion kann folgende **Zustände** einnehmen: → → →
- Für den Scheduler bestehen folgende drei Möglichkeiten zur Behandlung eines Schrittes einer laufenden (running) Transaktion:
  - **Ausführen** (execute): Transaktion → Zustand running
  - **Verzögern** (delay): Transaktion → Zustand delayed
  - **Zurückweisen** (reject): Transaktion → Zustand aborted (z.B. wenn Serialisierbarkeit gefährdet würde)



Scheduling-Verfahren:

1. Ein Scheduler **arbeitet aggressiv**, wenn er **Konflikte zulässt** und dann versucht, aufgetretene Konflikte zu erkennen und aufzulösen:
   - Ziel: Maximiert die Parallelität von Transaktionen
   - Risiko: Transaktionen werden erst am Ende ihrer Ausführung zurückgesetzt
   - Grenzfall: Im Extremfall ist keine Transaktion mehr erfolgreich
   - Beispiel: Postgres, Oracle («Multiversion Concurrency Control»)
2. Scheduler **konservativ**, wenn Konflikte möglichst vermeidet, Transaktionsverzögerungen in Kauf nimmt:
   - Ziel: Minimiert den Rücksetzungsaufwand für abgebrochene Transaktionen
   - Risiko: Erlauben eine geringere Parallelität von Transaktionen
   - Grenzfall: Im Extremfall findet keine Parallelisierung von Transaktionen mehr statt, d.h. die Transaktionen werden sequentiell ausgeführt
   - Beispiel: SQLServer (Scheduling-Verfahren: Sperrverfahren; 7 Haupt-Lock-Modi und diverse Untermodi)

Transaktionsverwaltung: Sperrverfahren

- Sperren (Locks) auf Datenbankobjekten: Abfrage der Sperre vor jedem Zugriff auf ein Datenbankobjekt und setzen einer Sperre wenn verfügbar.
- Erweiterung jeder Transaktion durch spezifische Operationen:
  - **Lese-Sperre** (Share Lock): read lock (rl) – read unlock (ru) andere Transaktionen weiterhin lesend auf Datenbank zugreifen und Lesesperren auf Datenbankobjekt setzen. Keine Schreibsperre möglich)

- **Schreib-Sperre** (Exclusive Lock): write lock (wl) – write unlock (wu) das betreffende Datenbankobjekt nur dieser Transaktion zur Verfügung und kann nur durch diese Transaktion geändert werden
- Unlock: read unlock (ru) und write unlock (wu) werden üblicherweise zu unlock (u) zusammengefasst.

Regeln zur Sperrdisziplin:

- Schreibzugriff w(x) nur nach Setzen einer Schreibsperre wl(x) möglich.
- Lesezugriffe r(x) nur nach Setzen einer Lesesperre rl(x) oder wl(x) erlaubt.
- Eine Schreibsperre w(x) kann nur gesetzt werden, wenn auf x keine Sperren existiert.
- Eine Lesesperre r(x) kann nur gesetzt werden, wenn auf x keine Schreibsperre existiert.
- Nach u(x) darf die Transaktion kein erneutes rl(x) oder wl(x) ausführen.
- Eine Transaktion darf eine Sperre der selben Art auf demselben Objekt nicht nochmals anfordern.
- Beim Commit/Rollback müssen alle Sperren aufgehoben werden.

Sperren führen zu folgenden (vier) Problemen:

- **Blocking**: Eine gesperrte Ressource zwingt andere Prozesse zu warten, bis diese wieder freigegeben wird → Reduktion des Durchsatzes an Transaktionen.
  Lösung: Keine, ohne Sperren funktioniert die Transaktionsverwaltung nicht.
- **Verhungern,** Livelock: Transaktion kommt nie dran, weil immer andere vorher berücksichtigt werden.
  - T1 sperrt x
  - T2 will x sperren, muss aber warten
  - T3 will danach x sperren, muss auch warten
  - T1 gibt x frei
  - T3 kommt vor T2 an eine Zeitscheibe, sperrt x
  - T2 will weiterhin x sperren, muss aber warten
  - T4 will danach x sperren, muss auch warten
  - T3 gibt x frei
  - T4 kommt vor T2 an die nächste Zeitscheibe ...

  Lösung: RDBMS muss geeignet «fair» auswählen.
- **Deadlock** (Verklemmung): Eine Menge von Transaktionen sperren sich gegenseitig, wenn jede Transaktion der Menge auf ein Ereignis wartet, das nur durch eine andere Transaktion der Menge ausgelöst werden kann. Da alle am Deadlock beteiligten Transaktionen warten, kann keine ein Ereignis auslösen, so dass eine andere geweckt wird. Also warten alle beteiligten Transaktionen ewig. Beispiel:

t_1	t_2
$wl(x)$	
	$wl(y)$
$wl(y)$	
	$wl(x)$
Verklemmung!	

  Lösung: RDBMS erkennt Deadlocks durch Zyklensuche im sogenannten «wer wartet auf wen» Graphen:
  - «Opfer»-Transaktion auswählen und zurücksetzen
  - Zu einem späteren Zeitpunkt erneut starten
- **Phantom-Read:** Sperren als Lösung? Beispiel: Ein Bonus von 100.000 Euro soll gleichmässig auf alle Mitarbeiter der Firma verteilt werden parallel dazu wird ein neuer Mitarbeiter eingefügt.

Zeit	T_1	T_2
1	select count(*) into X from Mitarbeiter;	
2		insert into Mitarbeiter values (Meier, 50.000, …);
3		**commit;**
4	update Mitarbeiter set Gehalt = Gehalt + 100.000/X;	
5	**commit;**	

  Lösung 1: Zusätzlich zu Tupeln muss der **Zugriffsweg**, auf dem man zu Objekten gelangt ist, gesperrt werden.
  Beispiel: SELECT COUNT(*) INTO X FROM Mitarbeiter
  - Alle Mitarbeiter (bzw. deren Primärschlüssel-Index) müssen mit einer RL-Sperre belegt werden.
  - Beim Einfügen eines neuen Mitarbeiter wird dies erkannt und T2 muss warten.

  Sperre kann ggf. auch selektiver sein z.B.: SELECT COUNT(*) INTO X FROM Mitarbeiter WHERE PNr BETWEEN 1000 AND 2000 Nur die Mitarbeiter mit der entsprechenden PNr müssen gesperrt werden
  Lösung 2: Von der Tabelle vorgängig einen **Snapshot** erstellen (siehe Mehrversionensynchronisation).
  Lösung 3 (unabhängig vom Isolationslevel): Die **Tabelle** ganz **sperren** Postgres: LOCK TABLE

Transaktionsverwaltung: Hinweise für die Praxis

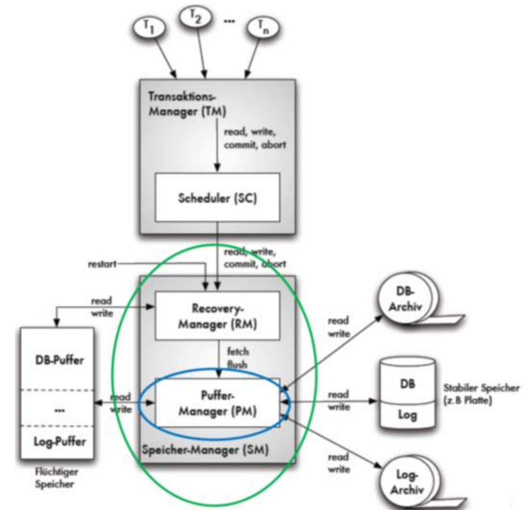
- **Blockierung:**
  - Lange laufende Abfragen vermeiden, kurze Transaktionen
  - Keine Benutzereingaben innerhalb von Transaktionen
  - Vernünftig indizieren und Indexe verwenden
  - Ineffiziente Abfragen optimieren
  - Sperr-Timeouts verwenden
- **Deadlocks:**
  - Objekte immer in derselben Reihenfolge ansprechen
  - Angepassten Isolationslevel verwenden
  - 'Teure' Objekte zuletzt ansprechen

Recovery (Wiederherstellung)

- **Recovery:** Alle Massnahmen zur Wiederherstellung verloren gegangener Datenbestände.
- **Behandlungsstrategien:** Je nach Fehlerart unterschiedliche Behandlungsstrategien ausführen
- Transaktions-Manager (TM) / Scheduler (SC) wahrt Isolation- und Konsistenzeigenschaft einer Transaktion.
- Recovery-Manager sichert die Atomaritäts- und Dauerhaftigkeitseigenschaft einer Transaktion.

Recovery-Komponenten

- Der **Speicher-Manager (SM)**, besonders wichtig für das Recovery, bildet die Schnittstelle zwischen flüchtigem und stabilem Speicher, er umfasst den Recovery-Manager und den Puffer-Manager.
- **Recovery-Manager (RM):** sorgt dafür, dass nach einem Fehler:
  - alle Änderungen „committeter" Transaktionen im stabilen Speicher abgelegt werden
  - keine Änderungen von aktiven oder abgebrochenen Transaktionen im stabilen Speicher verbleiben
- **Puffer-Manager (PM)** verwaltet den Puffer (DB- und Log-Puffer):
  - holt Daten (Seiten) vom stabilen Speicher in den Puffer
  - schreibt Daten (Seiten) vom Puffer in den stabilen Speicher
  - ersetzt Daten (Seiten) im Falle eines «Pufferüberlaufs»



Fehlerklassifikation

Klassifikation der Fehler nach dem betroffenen Bereich des DBMS:

- Transaktionsfehler
- Systemfehler (DB-/Log-Puffer)
- Mediafehler (DB-/Log-Archiv)

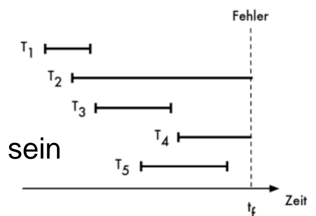
→ versch. Recovery-Massnahmen

Transaktionsfehler:

- haben den Abbruch der jeweiligen Transaktion zur Folge –
- haben keinen Einfluss auf Systemspeicher → lokaler Fehler
- Typische Transaktionsfehler:
  - Fehler im Anwendungsprogramm (z.B. Division durch 0)
  - Transaktionsabbruch durch Benutzer (z.B. unzulässige Dateneingabe, Timeout bei Inaktivität)
  - Transaktionsabbruch durch System (z.B. Deadlock, Verletzung Integritätsbedingungen/Zugriffsrechte)
- Behandlung: «Isoliertes» Zurücksetzen aller Transaktionsänderungen: UNDO/Rollback Datenänderungen

Systemfehler:

- Folge: Zerstörung Daten Hauptspeicher (flüchtige DB), jedoch nicht Hintergrundspeicher (permanente DB)
- Typische Systemfehler:
  - DBMS-Fehler (z.B. Konfigurationsfile fehlerhaft)
  - Betriebssystemfehler (Seitenfehler, ungültiger Befehl)
  - Hardware-Fehler (z.B. Stromausfall, Fehler im Memory)
- Behandlung:
  - Nachvollziehen Änderungen, die von abgeschlossenen Transaktionen nicht in DB eingebracht (REDO)
  - Zurücksetzen Änderungen, die von nicht beendeten Transaktionen in DB eingebracht wurden (UNDO).
- Beispiel Systemfehler: flüchtiger Speicher zum Zeitpunkt t_f ist verloren oder unbrauchbar.
- Transaktionszustände:
  - zum Fehlerzeitpunkt noch aktive Transaktionen (T_2 und T_4)
  - bereits vor dem Fehlerzeitpunkt beendete Transaktionen (T_1, T_3 und T_5)
- Probleme:
  - Dauerhaftigkeitseigenschaft → Effekte von T_1, T_3 und T_5 **müssen** dauerhaft in DB sein
  - Atomaritätseigenschaft → Effekte von T_2 und T_4 **dürfen nicht** in der DB sein



Fehlerklassifikation: Mediafehler

- Mediafehler: Verlust von Daten der stabilen Datenbank
- Typische Mediafehler:
  - «Head-Crashes», Controller-Fehler, …
  - Operator-Fehler / Fehler im Programm
  - Naturgewalten wie Feuer oder Erdbeben
- Behandlung:
  - Voraussetzung: Es existieren Sicherungskopien (DB-Archiv, Log-Archiv) auf separatem «Medium» (ev. auch Log-Datei auf eigenem Medium)
  - Einspielen der verlorenen DB-Dateien vom DB-Archiv (→ DB-Restore)
  - Anwenden aller Transaktionslogs aus Log-Archiv, die seit dem Anlegen des DB-Archivs erzeugt wurden
  - Datenbank muss mit REDO und UNDO in konsistenten Zustand gebracht werden (analog Systemfehler)

Logging

- Zur Wiederherstellung muss Historie aller Änderungen protokolliert (Logging) werden: Logbuch (''Transaction Log'')
- Physisches Log:
  - die alten und die neuen Werte der geänderten Daten werden gespeichert
  - die alten Werte nennt man Before Images und die neuen After Images.

Das WAL-Prinzip (Write Ahead Log-Prinzip)

- Wie muss das Schreiben des Logs (Log!) erfolgen, so dass die gepufferten Log-Seiten rechtzeitig auf dem stabilen Speicher sind?
- Das **Write Ahead Log-Prinzip (WAL-Prinzip)** fordert die Einhaltung folgender zwei Regeln:
  - **Vor COMMIT** einer **Transaktion** müssen zugehörigen (gepufferten) Log-Einträge (‘’log write on commit’’) auf stabilen Speicher ausgelagert werden (Logbuch). Regel ist notwendig, um REDO durchzuführen.
  - **Vor** dem **Auslagern** einer **modifizierten** (gepufferten) **Seite** in die persistente Datenbank (stabiler Speicher) müssen alle zugehörigen (gepufferten) LogEinträge zur Seite auf stabilen Speicher ausgelagert werden (Logbuch). Diese Regel ermöglicht das UNDO bei abgebrochenen Transaktionen.

Sicherungspunkte (SP)

- Problem Recovery: Wissen nicht, wie alt älteste Seite im Puffer war, wir müssen gesamte Log einspielen
- Lösung: Sicherungspunkte (checkpoint). Beim Aufruf eines Sicherungspunktes werden:
  - Die geänderte Seiten (dirty pages) der flüchtigen DB (Puffer) in die persistente DB geschrieben.
  - Der Puffer des Logs ebenfalls auf stabilen Speicher geschrieben.
  - Der Checkpoint wird im Logbuch registriert.

Finden Sie die Projektnummern aller Projekte, welche mindestens ein Teil geliefert bekommen, das auch von Sulzer (irgendwohin) geliefert wird.

- ```
  SELECT DISTINCT PNr
    FROM LTP
    WHERE TNr IN (SELECT TNr
                        FROM LTP, L
                        WHERE LTP.LNr = L.LNr AND L.LName ='Sulzer');
  ```

- Finden Sie alle Lieferanten/Teile-Paare, bei denen der Lieferant das Teil nicht liefert.
- ```
  SELECT lnr, tnr    FROM l, t
    EXCEPT (SELECT lnr, tnr    FROM ltp)
  ```

Datumseingaben können wie folgt aussehen
WHERE r.eroeffnungsdatum < '2010-01-01'   oder r.eroeffnungsdatum < '01.01.2010'

WHERE a = 'ABC' --> stimmt genau überein, dann =
WHERE a LIKE 'ABC%' --> nur ein Teilstring, dann mit LIKE

DISTINCT (Diese Operation kostet Millions!)

- DISTINCT erfordert ein internes Sortieren der Tabelle ($\sim n \cdot log(n)$ Operationen auf n Tupel) und beeinträchtigt die Performance, deshalb DISTINCT nur anwenden, wenn wirklich nötig!
- **Beispiel:** $n \cdot \log_2 n$ bei $1'000$ Tupeln $\simeq 10'000$ Vergleichs- und/oder Tausch-Operationen
- Mit Duplikatelimination **SPARSAM** umgehen. Nach jedem Schritt überlegen, ob Elimination notwendig ist!

SELECT DISTINCT name, vorname
FROM Mitglieder m, besuchen b, Kurse k
WHERE m.MNr = b.MNr AND k.KNr = b.KNr AND ~~UPPER(k.Bezeichnung) LIKE ‘%POWER%’ AND~~
EXISTS IN (SELECT k2.KNr FROM Kurse k2 WHERE UPPER(k2.Bezeichnung) LIKE ‘%POWER%’ AND k2.KNr = k.KNr);

SELECT DISTINCT k.KNR, k.Bezeichnung
FROM besuchen b, Kurse k
WHERE b.KNr = k.KNr
GROUP BY b.KNr
HAVING COUNT(DISTINCT MNr)/5 > (SELECT * FROM besuchen b2 GROUP BY KNr);

DROP TABLE IF EXISTS Dozenten;

CREATE TABLE Dozenten(PNr integer NOT NULL,

Lohnstufe integer NOT NULL,

CONSTRAINT PK_D PRIMARY KEY (PNr),

CONSTRAINT FK_D FOREIGN KEY (PNr) REFERENCES Person(PNr);

DROP TABLE IF EXISTS Zuordnung;

CREATE TABLE Zuordnung(DozPNr integer NOT NULL,

Bezeichnung varchar(50) NOT NULL,

TeilPNr integer NOT NULL,

CONSTRAINT PK_Z PRIMARY KEY (TeilPNr, Bezeichnung),

CONSTRAINT FK_Dozent FOREIGN KEY (DozPNr) REFERENCES Dozent(PNr),

CONSTRAINT FK_Teilnehmer FOREIGN KEY (TeilnPNr) REFERENCES Teilnehmer(PNr),

CONSTRAINT FK_Teilnehmer FOREIGN KEY (Bezeichnung) REFERENCES Module(Bezeichnung);

Stand der Arbeit

SW	Vorlesung	AB/Aufgabe	Skript	ZF
1	☒	☒	☒	☒
2	☒	☒	☒	☒
3	☒	☒	☒	☒
4	☒	☒ Rel. Algebra	☒	☒
5	☒	☒	☒	☒
6	☒	☒	☐	☒
7	☒	☒ Rel. Algebra	☐	☒
8	☒	☒	☐	☒
9	☒	☒	☐	☒
10	☒	☒	☐	☒
11	☒	☒	☐	☒
12	☒	☒	☐	☒
13	☒	☐	☐	☐
14	☒	☐	☐	☐

Modulauflagen

- Im Verlauf des Semesters werden drei Praktika eingezogen und bewertet. Die Terminierung der entsprechenden Praktika können dem Semesterprogramm entnommen werden. Zusammen fliessen diese drei bewerteten Praktika mit 20% in die Modulschlussnote ein.
- Es wird eine Semesterendprüfung durchgeführt, deren Bewertung 80% der Modulschlussnote ausmacht.
- In der Semesterendprüfung gilt: **Open Book**
- Es ist während der Prüfung keinerlei Kommunikation zugelassen.

Prüfungsaufbau

wird 1 Aufgabe zu Indizierung kommen

viel multiple choice

Rel. Algebra 16Pt → 4 Varianten & die richtige auswählen

SQL 30

ER 24 → am Schluss machen

Technik 10 — Transaktion

Index