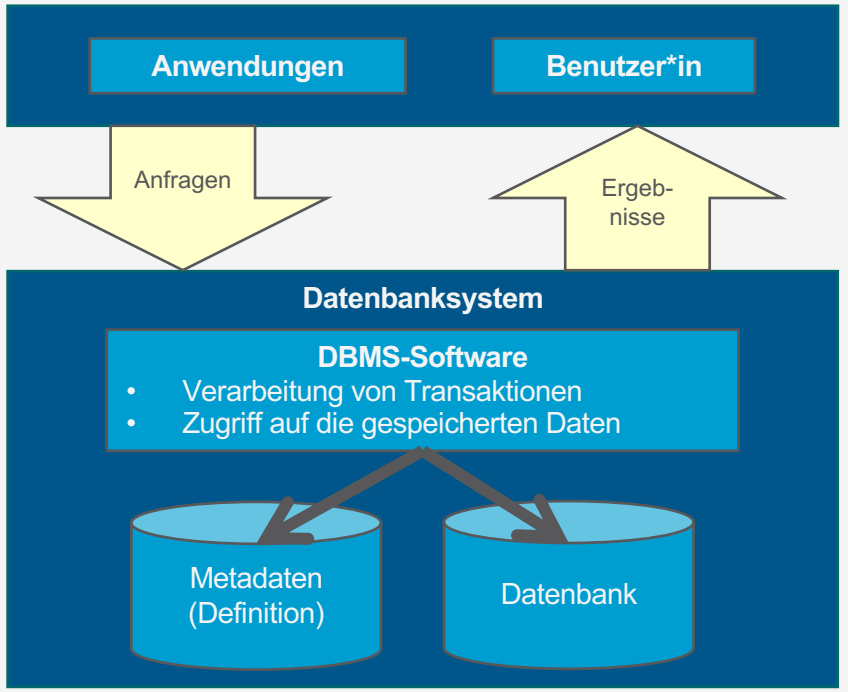
Inhaltsverzeichnis

WWU MÜNSTER

Datenbanken

Überblick



Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Organisatorisches: Vorlesung

- Modul: Datenbanken
  - Modulnummer: INF-B-107
  - 3+2 SWS, 7 ECTS (= 210 Stunden)
- Vorlesung:

Datenbanken

- Raum
  - Hörsaal **M 2**
  - Aufzeichnung der Vorlesung (nicht der Übung!) per electures
    - Erfahrung hat gezeigt, dass es ein bis zwei Tage dauern kann, bis eine Aufzeichnung zur Verfügung steht

Organisatorisches: Termine & Zeitplan

- Termine
  - **Mittwochs, 10.15 – 11.45 Uhr**
  - **Freitags, 08.15 – 09.45 Uhr**

#	Mi		Fr	
1	5.4.	Vorlesung 1	*7.4.*	*-- (Karfreitag)*
2	12.4.	Vorlesung 2	14.4.	Vorlesung 3
3	19.4.	Vorlesung 4	21.4.	Vorlesung 5
4	26.4.	Vorlesung 6	28.4.	Vorlesung 7
5	3.5.	Vorlesung 8	5.5.	Vorlesung 9
6	10.5.	Vorlesung 10	12.5.	Vorlesung 11
7	17.5.	Vorlesung 12	19.5.	*-- (Brückentag)*

- 3 SWS Vorlesung über 13 Wochen = 39 SWS → 19 Vorlesungen + Q&A
  - Vorläufiger Zeitplan siehe unten

#	Mi		Fr	
8	24.5.	Vorlesung 13	26.5.	Vorlesung 14
9	*31.5.*	*-- (Ferien)*	*2.6.*	*-- (Ferien)*
10	7.6.	Vorlesung 15	9.6.	*-- (Brückentag)*
11	14.6.	Vorlesung 16	16.6.	Vorlesung 17
12	21.6.	Vorlesung 18	23.6.	Vorlesung 19
13	28.6.	--	30.6.	--
14	5.7.	*Q&A*	7.7.	--

Organisatorisches: Übung

- Übungsbetreuer: *Sagad Hamid*
- Tutoren: *Karim Hourani*, *Julian Seljami*
- 5 Übungsgruppen
  - Di., 10.15-11.45 Uhr
  - Di., 12.15-13.45 Uhr
  - Mi., 08.15-09.45 Uhr
  - Mi., 12.15-13.45 Uhr
  - Mi., 14.15-15.45 Uhr
  - Erste Übungstermine: 18/19.4.
- Übungsblätter
  - Veröffentlichung in der Regel am Freitag Abend, 18.00 Uhr, einer Woche
- Abgabe im Learnweb in Dreier-Gruppen
  - Bis spätestens 18.00 Uhr am Freitag drauf
    - Übung dazwischen für Fragen nutzen
- Lösungen werden in der Übung nach Abgabe besprochen
  - Keine Veröffentlichung von Musterlösungen oder Lösungsvorschlägen
  - Keine Video-Aufzeichnung der Übungen

Organisatorisches: Übung

- Übungsbetreuer: *Sagad Hamid*
- Tutoren: *Karim Hourani*, *Julian Seljami*
- 5 Übungsgruppen
  - Di., 10.15-11.45 Uhr
  - Di., 12.15-13.45 Uhr
  - Mi., 08.15-09.45 Uhr
  - Mi., 12.15-13.45 Uhr
  - Mi., 14.15-15.45 Uhr
  - Erste Übungstermine: 18/19.4.
- Verteilung auf Übungsgruppen über Learnweb
  - Einteilung über gerechte Verteilung
  - Verfügbar bis 12.4.2023, 12.00 Uhr
  - Aufteilung in Dreier-Gruppen eigenständig
    - Innerhalb einer Übungsgruppe
    - Forum pro Gruppe verfügbar für Absprache
    - Sobald eine Dreier-Gruppe feststeht, Email an den Tutor mit Namen, Matrikelnummern (eine*r schreibt mit den anderen beiden in cc)
    - Wer keine*n Partner*in findet, meldet sich auch beim Tutor per Email

Organisatorisches: Übung

- Es wird SQL Übungen geben
- Da wir einen Server aufsetzen und Sie als Nutzer*innen eintragen, werden Sie im Laufe des Semesters eine Email aus dem System dazu erhalten
  - Nicht erschrecken ;)
  - Sie müssen mit der Email nichts weiter tun

Organisatorisches: Gesamtübersicht Zeitplan

- 9 Übungsblätter à 16 Punkte → 144 P.
  - B0 zum Einstieg ohne Abgabe
  - B10* kurzer Bonuszettel (8 P.)

- Vorlesung x → Vx
- Veröffentlichung Blatt x → Bx
- Abgabe Blatt x → Ax

#	Di/Mi		Fr	
1	5.4.	V1, B0	*7.4.*	*-- (Karfreitag)*
2	11/12.4.	V2	14.4.	V3, B1
3	18/19.4.	V4, Ü0	21.4.	V5, A1, B2
4	25/26.4.	V6, Ü1	28.4.	V7, A2, B3
5	2/3.5.	V8, Ü2	5.5.	V9, A3, B4
6	9/10.5.	V10, Ü3	12.5.	V11, A4, B5
7	16/17.5.	V12, Ü4	19.5.	--, A5, B6

#	Di/Mi		Fr	
8	23/24.5.	V13, Ü5	26.5.	V14, A6, B7
9	*30/31.5.*	*-- (Ferien)*	*2.6.*	*-- (Ferien)*
10	6/7.6.	V15, Ü6	9.6.	--, A7, B8
11	13/14.6.	V16, Ü7	16.6.	V17, A8, B9
12	20/21.6.	V18, Ü8	23.6.	V19, A9, B10*
13	27/28.6.	--, Ü9	30.6.	--, A10
14	4/5.7.	*Q&A*, Ü10	7.7.	--

Organisatorisches: Studienleistung

- 50% der zu erreichenden Punkte in den Übungsblättern, i.e., 72 Punkte
  - Unabhängig von Kapiteln oder Blättern
    - Änderung zu vorherigen Jahren!
  - Punkte für ernsthafte Bearbeitung, nicht nur für korrekte Ergebnisse
    - Damit: Volle Punkte in einer Übungsaufgabe heißt nicht zwangsweise volle Punkte in der Klausur!
  - Beachten Sie mögliche Anmeldefristen für die Studienleistung

- Studienleistungen aus den vorherigen Jahren haben Bestand
  - ACHTUNG: Klausurrelevant ist die Vorlesung von diesem Jahr!
    - Es würde sich daher wahrscheinlich lohnen, trotzdem an den Übungen teilzunehmen

Organisatorisches: Prüfungsleistung

- Erfolgreiche Teilnahme an der Klausur
  - 120 Minuten
  - 1. Termin
    - Datum: tba (wahrscheinlich Donnerstag, 20.7.2023)
  - 2. Termin
    - Datum: tba (September)

- Teilnahmebedingungen
  - Erfolgreiches Bestehen der Studienleistung
  - Prüfungsanmeldung
    - Bitte stellen Sie sicher, dass Sie die für Sie geltenden Regularien zur Anmeldung erfüllen!

Organisatorisches: Learnweb-Kurs

- Kurs

  https://sso.uni-muenster.de/LearnWeb/learnweb2/course/view.php?id=67442

  - Bis 30.4.2023 ohne Einschreibeschlüssel, danach per Email an Sagad Hamid oder mich
  - Der Ort für
    - Folien
    - Aktuelle Hinweise
    - Alles, was die Übung betrifft
      - Einteilung Übungstermine + Kleingruppenorganisation, Veröffentlichung Übungsblätter, Abgabe Übungsblätter
  - Eventuell mit zeitlich leichter Verzögerung finden Sie zumindest die Folien auch frei verfügbar auf unserer Webseite:

    https://www.uni-muenster.de/Informatik.AGBraun/teaching/sose23/db23.html

Lernziele

Inhalte

- Von der Anwendung her
  - Phasen des Datenbankentwurfs
    - Modellierung von Anforderungen über ER und RM zu Umsetzung mittels SQL
- Von der Umsetzung her
  - Blick hinter die Kulissen
  - Effiziente Umsetzung mittels Anfragepläne und Hilfsdatenstrukturen (Indices)
  - Sichere Umsetzung mittels Transaktionen und Wiederherstellung

Kompetenzen

- Von der Anwendung her
  - Eine Datenbank entwerfen und mittels SQL definieren und befragen können
- Von der Umsetzung her
  - Verstehen und erklären können, warum manche Anfragen schnell und andere wiederum sehr lange brauchen, und was man tun kann, um möglichst schnell zu sein
  - Probleme und Lösungsstrategien der Mehrbenutzerverwaltung und Sicherung verstehen und erklären können

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - *Noch offen*: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Skript, Literatur, Quellen

- Skript: Folien und Vorlesung vor Ort

- Literatur
  - **A. Kemper, A. Eickler:**
    ***Datenbanksysteme: Eine Einführung*.**
  - R. Elmasri, S.B. Navathe:
    Grundlagen von Datenbanksystemen.
  - A. Silberschatz, H. F. Korth, S. Sudarshan:
    Database System Concepts.

- Folien (wenn nicht anders angegeben; fast immer angepasst)
  - Vorlesungsfolien der Vorlesung „Informationssysteme I“
    Prof. Dr. Daniela Nicklas
  - Vorlesungsfolien der Vorlesung „Datenbanken“
    Prof. Dr. Ralf Möller, Dr. Özgür Öczep

Hinweis

- *Die Vorlesung wird gerade erst aufgebaut!*

- Was heißt das?
  - Es können sich kurzfristig noch einmal Dinge im inhaltlichen Aufbau ändern
  - Die Folien und Übungsblätter werden trotz größtmöglicher Sorgfalt Tippfehler beinhalten
  - Darstellungen, die mir zum Zeitpunkt des Bauens der Folien als sinnvoll erschienen, können sich als nicht hilfreich in der Vorlesung herausstellen

Wenn Sie Fehler finden oder Vorschläge zur besseren Darstellung haben, geben Sie gern Bescheid!

WWU MÜNSTER

Einführung

Datenbanken

Gestell	Sorte	Jahrgang	Anzahl_Flaschen
2	Franken	2009	5
1	Baden	2006	3
4	Rheinhessen	2007	10
1	Mosel	2013	2
2	Franken	2010	10

```
SELECT Gestell, Sorte, Jahrgang
FROM Weinkeller
WHERE Anzahl_Flaschen >= 4
```


Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Überblick: 1. Einführung

A. Datenbanken

- Datenbank (DB)
- Datenabstraktion, Datenmodelle, Datenunabhängigkeit
- Mehrbenutzersysteme

B. DB-Umgebungen

- DB-Sprachen
- Datenbanksystem (DBS)
- Datenbankmanagementsystem (DBMS)

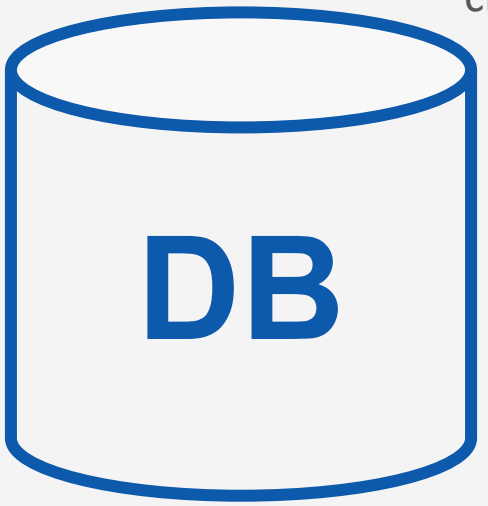
C. Phasen des DB-Entwurfs

- Anforderung, Modellierung

Datenbank (DB)

Sammlung von Daten
- in der Regel logisch zusammenhängend

“closed world assumption”



Miniwelt oder
Universe of Discourse

Daten → Information
(Interpretation)

Kennzeichen von Daten in DBs
- Lange Lebensdauer (Jahre, Jahrzehnte)
- Reguläre Strukturen
- Große Datenobjekte, große Datenmengen
- Stetig anwachsende integrierte Bestände (Giga-, Tera-, Petabyte)

Erstes Beispiel einer (relationalen) DB und einer Anfrage

- DB für Inventar eines Weinkellers
  - Tabelle Weinkeller

Gestell	Sorte	Jahrgang	Anzahl_Flaschen
2	Franken	2009	5
1	Baden	2006	3
4	Rheinhessen	2007	10
1	Mosel	2013	2
2	Franken	2010	10

- Anfrage: Alle Informationen zu Weinen, von denen es mindestens 4 Flaschen gibt

```
SELECT Gestell, Sorte, Jahrgang
FROM Weinkeller
WHERE Anzahl_Flaschen >= 4
```

- Ergebnis:

Gestell	Sorte	Jahrgang
2	Franken	2009
4	Rheinhessen	2007
2	Franken	2010

E. F. Codd: A Relational Model of Data for Large Shared Data Bank. In: *Communications of the ACM*, Vol. 13, No. 6, pages 377-387, 1970.

Datenabstraktion

- abs-trahere = "abziehen, entfernen"
  → Weglassen von Einzelheiten, Überführen in etwas Einfacheres
- Abstraktion: ein Grundprinzip der Informatik!
  - Abstraktionsschicht verbirgt Einzelheiten/Aufgaben
    → Unabhängigkeit für darüber liegende Schichten
- Unterschiedliche Dimensionen:
  - Abstraktion von der Speicherung
  - Abstraktion von Anwendungen (= mehrere Anwendungen möglich)
  - Konzeptuelle Sicht auf die Datenbanken

- Basis für Abstraktion: Datenmodell

Datenmodell

- DB: Sammlung von Daten
- DB-Struktur/Schema: Elemente zur Definition, welche Daten möglich sind
  - Datentypen (z.B. String, Integer, ...)
  - Beziehungen (z.B. „Jeder Mitarbeiter hat einen Vorgesetzten.“)
  - Einschränkungen (z.B. Das Geburtsdatum muss in der Vergangenheit liegen)

- Datenmodell
  - Elemente zur Definition einer Datenbankstruktur
  - Basis-Operationen zur Abfrage und Änderung von Daten
  - Erweiterungen durch benutzerdefinierte Operationen

- Populäres Beispiel: das **relationale Datenmodell**

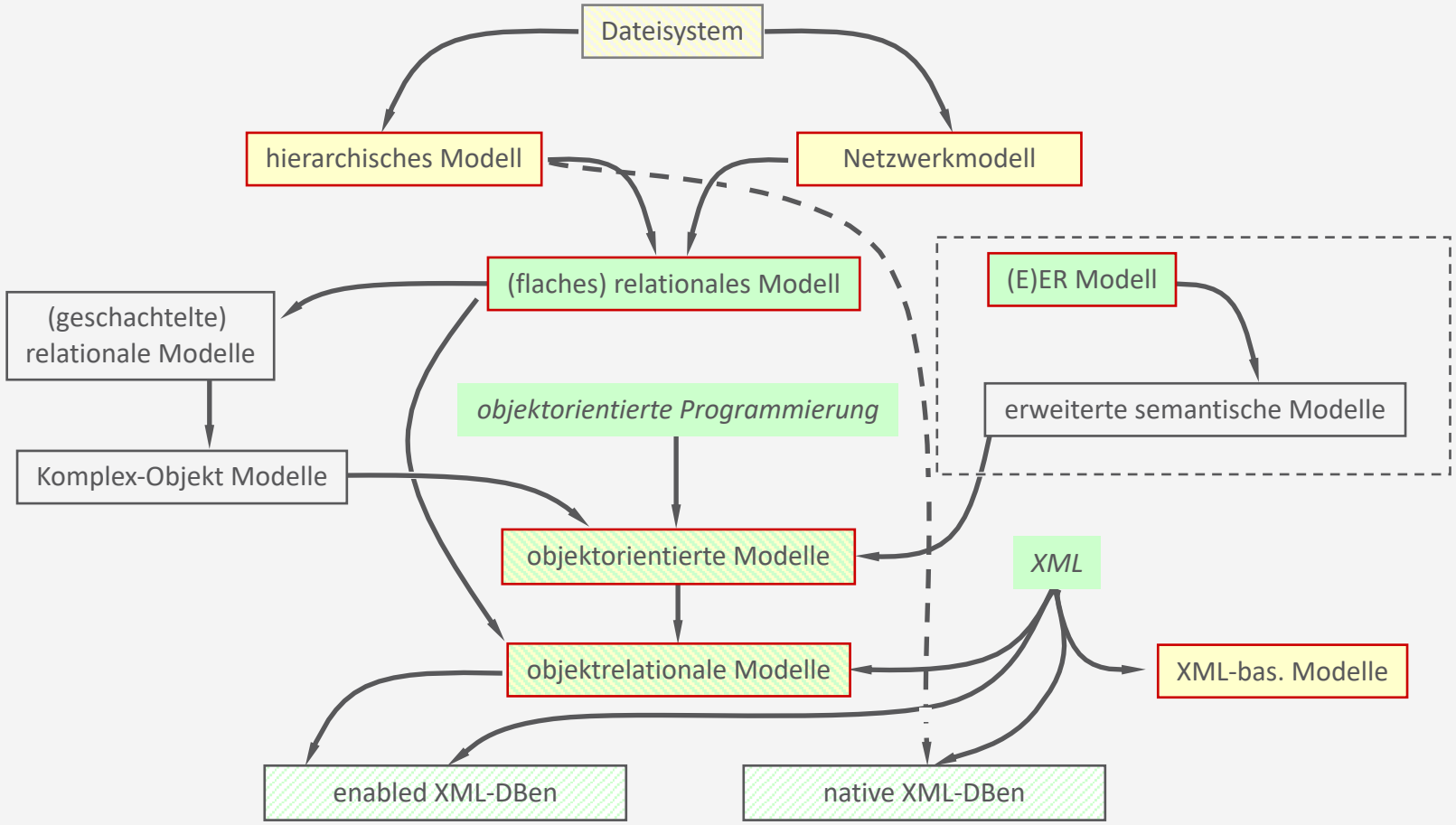
Modellierungsebenen von Datenmodellen

- Konzeptuelle Datenmodelle
  - Zur Definition der Miniwelt (→ Kundenanforderungen)
  - Beispiel: Entity-Relationship-Modell (ER-Modell)

- Logische Datenmodelle
  - Spezifikation, die leicht implementiert werden kann (→ für Entwickler, unabhängig vom DBMS)
  - Beispiel: Relationales Datenmodell

- Physische Datenmodelle
  - Spezifikation der konkreten Datenspeicherung (für ein konkretes DBMS)

Eine kurze Geschichte der Datenmodelle



Hierarchisches Modell

- Ältestes klassisches Datenmodell
- Ein Datensatz und alle von ihm abhängigen Datensätze → hierarchische Einheit

- Natürliche Hierarchie:
  - Unistrukturdatei (Universität -> Fakultät -> Institut -> Department)

- Künstliche Hierarchie:
  - Artikeldatei (Artikel -> Lieferant)

  könnte auch sein:
  - Artikeldatei (Lieferant -> Artikel)

Eigenschaften des Hierarchischen Modells

- Modellierung von Beziehungen:
  - 1 : 1 - Elternelement wird Kindelement zuordnet
  - 1 : n - Elternelement wird mehreren Kindelementen zugeordnet
  - m : n - Nicht direkt darstellbar
  - Keine Zyklen

- Zugriff
  - Entlang der Hierarchie sehr effizient
  - „Quer dazu“ sehr ineffizient
    - Beispiel: Teilnehmerdatei (Vorlesung -> Student)
      - Alle Studierenden der Vorlesung Datenbanken → prima
      - Alle Vorlesungen von Martha Mustermann → eeek …

Fazit:
- Gut bei klar definiertem Einsatz (z.B. directories, index-Verwaltung, …)
- Ineffizient bei allgemeinem Einsatz (wg. mangelnder Flexibilität)

Netzwerkmodell

- Schreibt das hierarchische Modell fort:
  - Ein Element kann mehreren Gruppen zugeordnet werden
    - Student in Vorlesung, in Studiengang, in Semester, …
  - Mehrere Wurzelelemente möglich
    - Jetzt auch n:m Beziehungen; allerdings nicht direkt:
      Kursbelegung (Student → Belegung → Kurs)

- Nachteile:
  - Wird leicht unübersichtlich
  - Für Abfragen ist genaue Kenntnis der Struktur nötig
  - Sequentielles Lesen („alle Studierenden einer Vorlesung“) wird ineffizienter

Das (flache) relationale Modell

- Für die Praxis das wichtigste Datenmodell!
- Tabellen (mit Attributen = Spaltenüberschriften) sind Container für komplexe Datenobjekte

Teilnehmer

MatrikelNr	Vorname	Nachname	Semester
4711	Martha	Mustermann	3
42	Arthur	Dent	21

- Beziehungen:
  - Durch Gruppierung in Tabellen
  - Durch wertebasierte Zusammenhänge zwischen Tabellen:

Vorlesung

VorlesungNr	Titel
42	Datenbanken
48151623	Altrömisches Abwasserecht

Belegung

VorlesungNr	MatrikelNr
42	4711
48151623	42

Schema / Instanz / DB-Zustand

- Schema (Intension)
  - Beschreibung der kompletten Struktur einer DB
  - Ändert sich (hoffentlich) selten
- Instanz (elementare Extension)
  - *Einzelne*, der vorgegebenen DB-Struktur entsprechende, aus konkreten Datenelementen bestehende Datensätze
- DB-Zustand
  (Gesamt-Extension oder Snapshot)
  - Die Gesamtheit der aktuell in einer DB gespeicherten Daten

Teilnehmer

MatrikelNr	Vorname	Nachname	Semester
4711	Martha	Mustermann	3

Vorlesung

VorlesungNr	Titel

Belegung

VorlesungNr	MatrikelNr

Physische Datenmodelle

- Physische Datenmodelle beschreiben konkret, wie die Daten anhand von Angaben zu
  - Datensatzformaten,
  - Datensatzanordnungen und
  - Zugriffspfaden

  physisch gespeichert werden (sollen)

- Ein Zugriffspfad ist eine Datenstruktur, welche die Suche nach Datensätzen in einer DB unterstützt/beschleunigt

- Beispiele: B-Bäume, B*-Bäume, R-Bäume, …

- Davon soll der*die Nutzer*in nichts mitbekommen!

Datenunabhängigkeit

- Ein Schema kann geändert werden, ohne zwangsläufig auch auf der nächsthöheren Ebene Änderungen vornehmen zu müssen.

- Logische Datenunabhängigkeit
  - Ein konzeptuelles/logisches Schema ändern, ohne immer auch externe Schemata oder Applikationen ändern zu müssen.

Teilnehmer

MatrikelNr	Vorname	Nachname	Semester

- Physische Datenunabhängigkeit
  - Ein internes Schema ändern, ohne konzeptuelle/ logische und externe Schemata sowie Applikationen ändern zu müssen.

Beispiele für Änderungen

- Logische Datenunabhängigkeit
  - Ein konzeptuelles/logisches Schema ändern
  - Hinzufügen von Attributen und Tabellen zum konzeptionellen Schema
  - Verändern der Tabellenstruktur
  - DB-Erweiterung durch neue Datensatztypen/Datenfelder
  - DB-Reduktion/Streichung bestehender Datensatztypen oder Datenfelder
  - Erweiterung („Verschärfung“) oder Reduktion („Entschärfung“) von Einschränkungen der Schemata

  →Wirkt sich nur auf externe Schemata aus, die sie nutzen

Beispiele für Änderungen

- Physische Datenunabhängigkeit
  - Ein internes Schema ändern
  - Veränderung des Speicherortes
  - Änderung des Speicherformates
  - Anlegen/Löschen von Indizes (für Anfrageoptimierung)

  →Verbleiben die gleichen Daten in der DB, so muss das konzeptuelle/logische DB-Schema i.d.R. nicht angepasst werden.

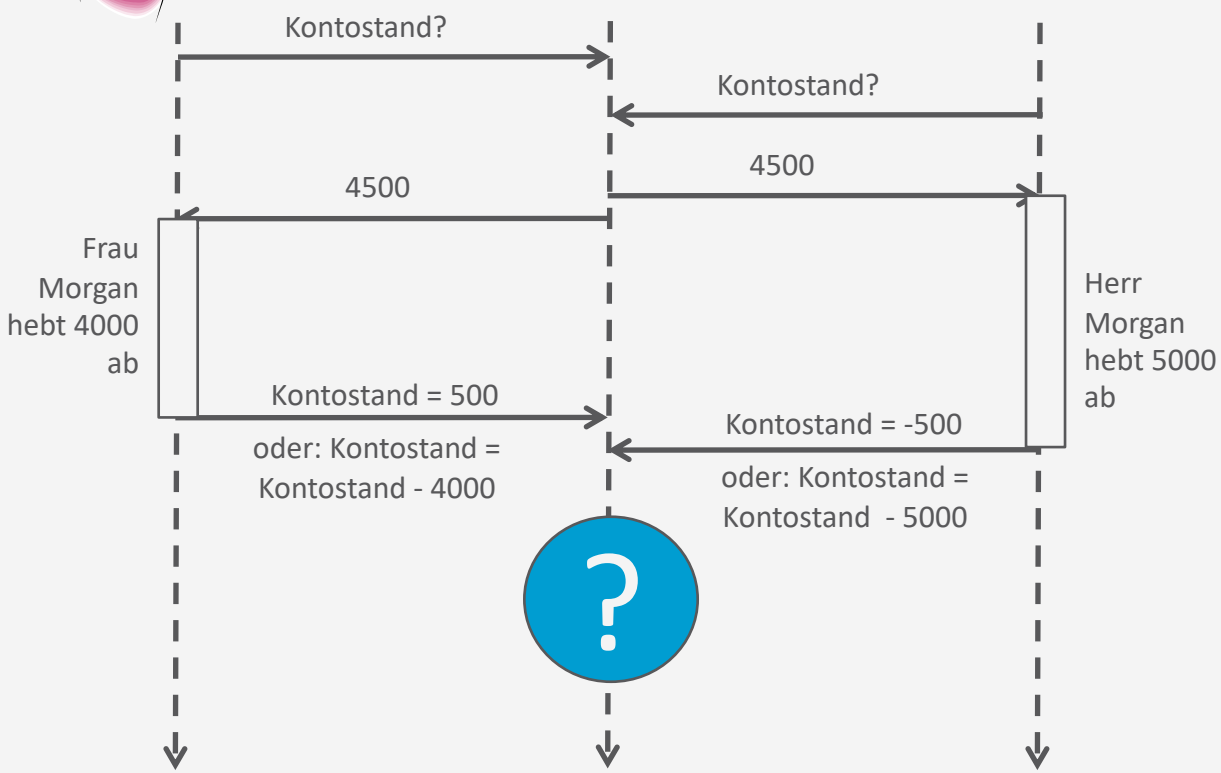
Viele gleichzeitige Benutzer...

- Jede*r Nutzer*in
  - Stellt Anfrage an den Kontostand
  - Verändert den Kontostand

  → Abfolge von DB-Befehlen = Transaktion

Welche Anforderungen ergeben sich?

Was sollte bei einem Systemabsturz passieren?

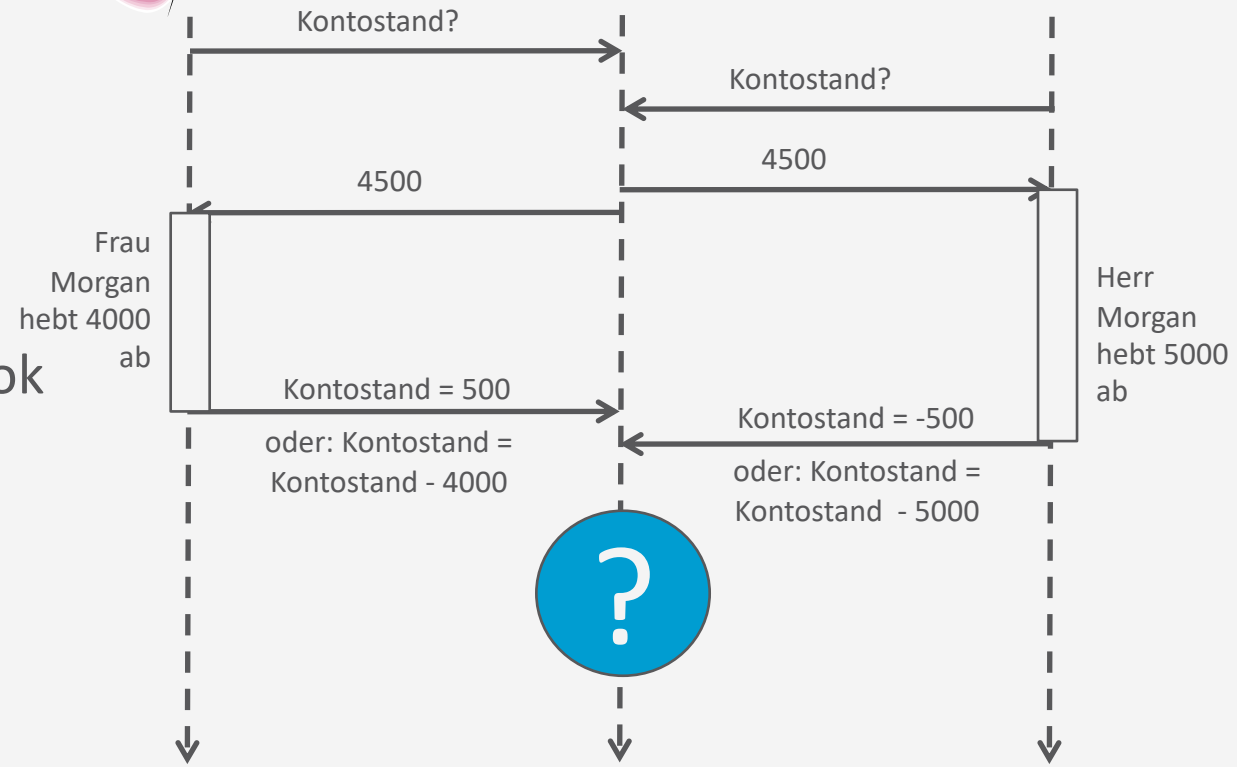


Viele gleichzeitige Benutzer...

- Jede*r Nutzer*in
  - Stellt Anfrage an den Kontostand
  - Verändert den Kontostand

  → Abfolge von DB-Befehlen = Transaktion
- Anforderungen:
  - **A**tomicity (Atomarität): Alles oder nichts
  - **C**onsistency (Konsistenz): Vorher ok, hinterher ok
  - **I**solation (Isolation): Jede*r denkt, er sei alleine auf der DB
  - **D**urability (Dauerhaftigkeit): Transaktionen bestätigt? Dann sind die Daten jetzt sicher

ACID-Eigenschaften (später mehr)



Zwischenzusammenfassung

- Datenbank
  - Persistente Speicherung großer Datenmengen
  - Logisch zusammenhängende Sammlung von Daten mit inhärenter Bedeutung
- Datenabstraktion
  - Datenmodelle: konzeptuell, logisch und physisch
  - Schema (Intension), Instanz und DB-Zustand (Extension)
  - Datenunabhängigkeit
    - Möglichkeit, ein Schema auf einer Ebene zu ändern unabhängig von anderen Ebenen
    - Genauere Betrachtung: logische und physische Datenunabhängigkeit
- Mehrbenutzung
  - Transaktionen als eine Abfolge von Datenbankbefehlen
  - Anforderungen: ACID (atomicity, consistency, isolation, durability)

Überblick: 1. Einführung

A. Datenbanken
- Datenbank (DB)
- Datenabstraktion, Datenmodelle, Datenunabhängigkeit
- Mehrbenutzersysteme

B. DB-Umgebungen
- DB-Sprachen
- Datenbanksystem (DBS)
- Datenbankmanagementsystem (DBMS)

C. Phasen des DB-Entwurfs
- Anforderung, Modellierung

DB-Sprachen

- Definition von DBs:
  - View Definition Languages (VDLs): extern
  - Data Definition Languages (DDLs): logisch
  - Storage Definition Languages (SDLs): intern

- Zugriff auf DBs
  (Einfügen, Ändern, Löschen und Anfragen von Datensätzen):
  - Data Manipulation Languages (DMLs)
    - Einfüge-, Änderungs- und Löschoperationen: Updates
    - Reine Anfragen: „Queries“
    - Alle Zugriffsarten: „Manipulation“

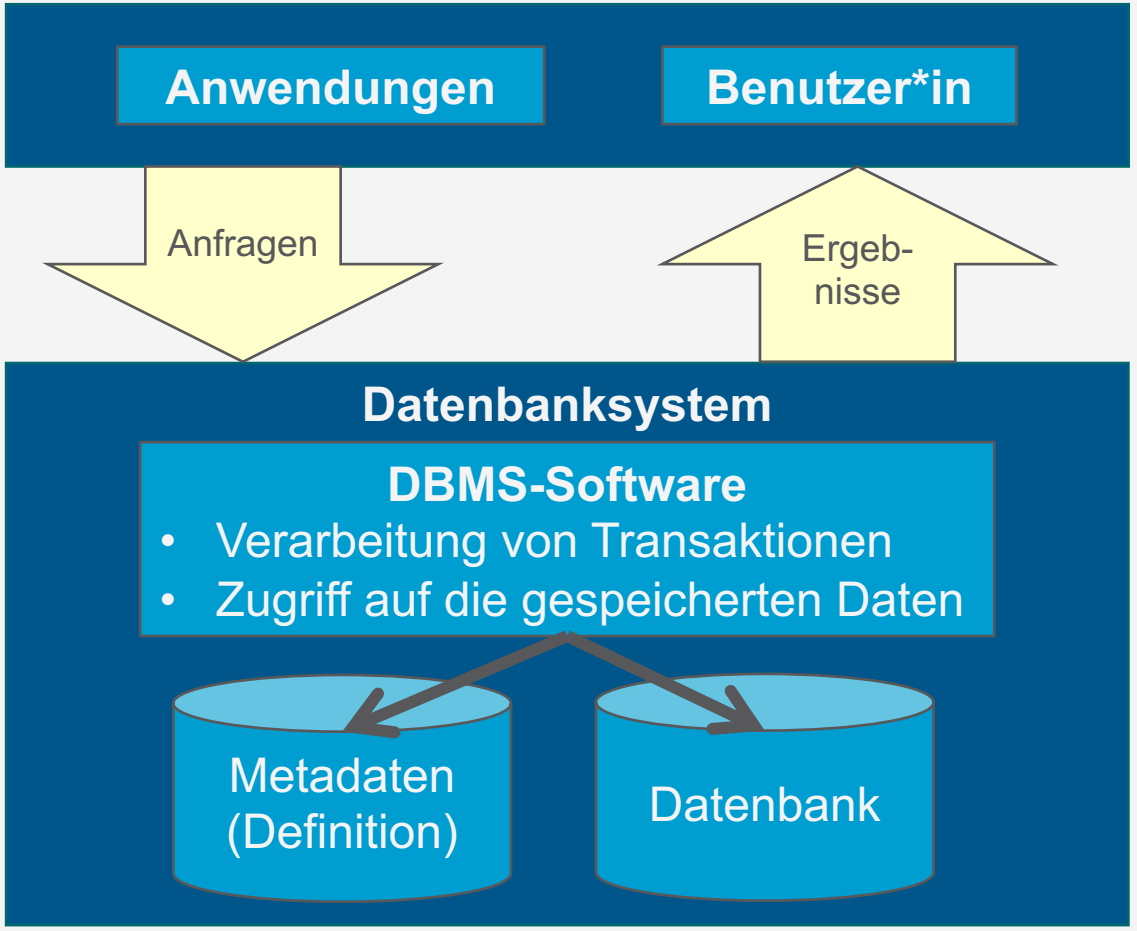
SQL : Structured Query Language

- Universal-Sprache für Datenbanken
  - VDL, DDL, SDL und DML in einem
- Haupteigenschaften:
  - Mengenorientiertes Arbeiten
    - Adressierung einer Menge von Datensätzen (zurückgegeben, geändert, gelöscht)
    - Menge kann auch nur ein Element enthalten
  - Deklarativ
    - Angabe darüber, welche Daten man möchte
    - Keine Angabe darüber wie der Zugriff erfolgen soll (Abstraktion; → Optimierungsmöglichkeit für das DBMS!)

```
CREATE TABLE Student (
  Name          VARCHAR2(100) NOT NULL,
  StudentNumber NUMBER(10)    PRIMARY KEY,
  Class         NUMBER(2)     DEFAULT 1 NOT NULL,
  Major         VARCHAR2(10)
);

INSERT INTO Student (Name, StudentNumber, Class, Major)
VALUES ('Smith', 17, 1, 'CS');

INSERT INTO Student (Name, StudentNumber, Class, Major)
VALUES ('Brown', 8, 2, 'CS');

SELECT Name
FROM Student
WHERE Major = 'CS';
```

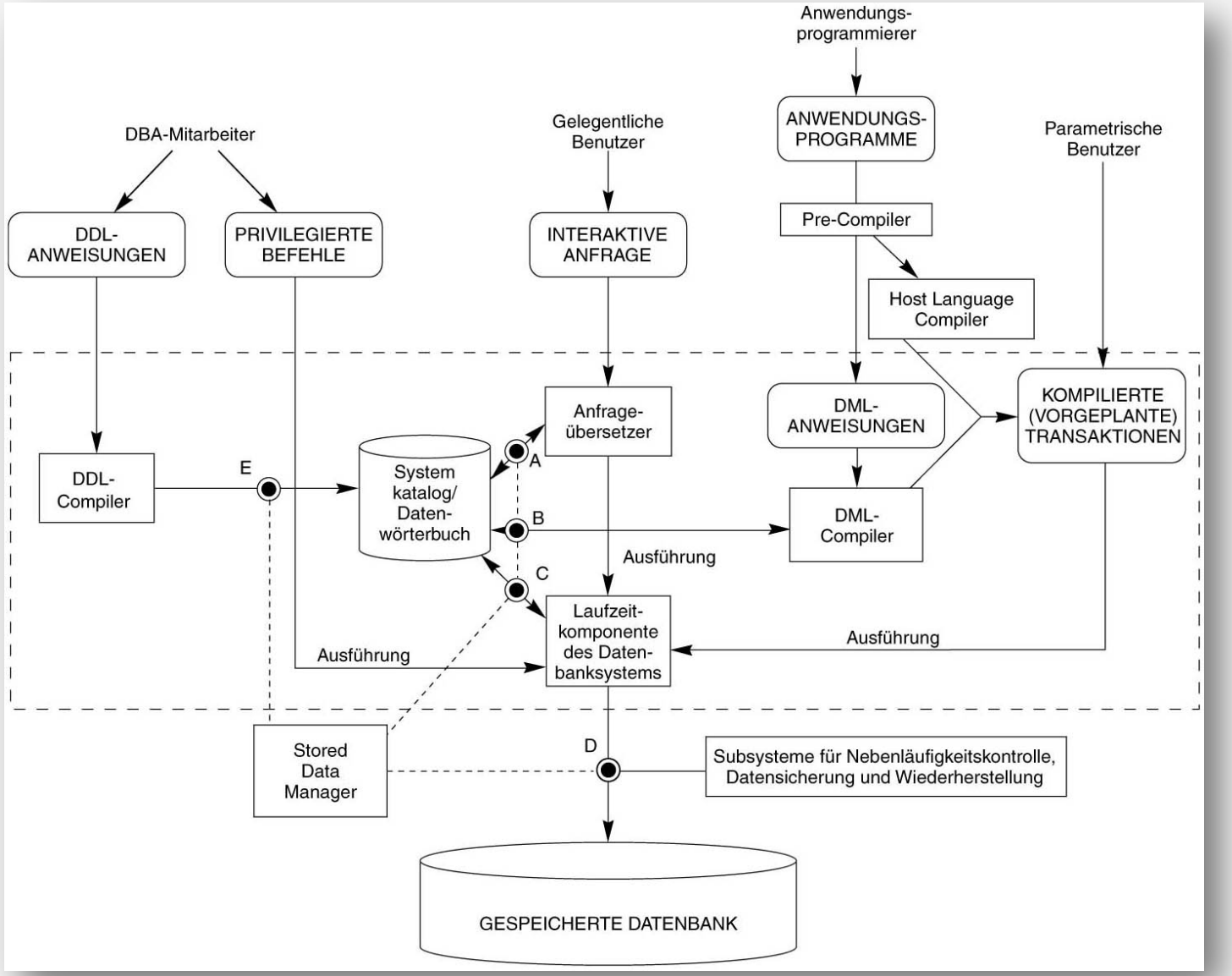
Datenbankmanagementsystem (DBMS) & Datenbanksystem (DBS)

- DBMS: Software-System, um DBs zu verwalten
  - Was muss es leisten?
    - Interfaces für Nutzung der DB
      - DB definieren, Daten ablegen + verändern, Anfragen
    - Anfrageverarbeitung / Datenmanipulationen
    - Unter Anforderungen an Korrektheit (ACID), Effizienz (Nutzbarkeit)
      - Auch bei Mehrbenutzersystemen
    - Weiteres Funktionen: Sicherheitsmechanismen, Fehlerbehandlung, Integritätskonzepte, verschiedene Sichten auf die DB, Optimierung von Anfragen
- DBS = DB + DBMS



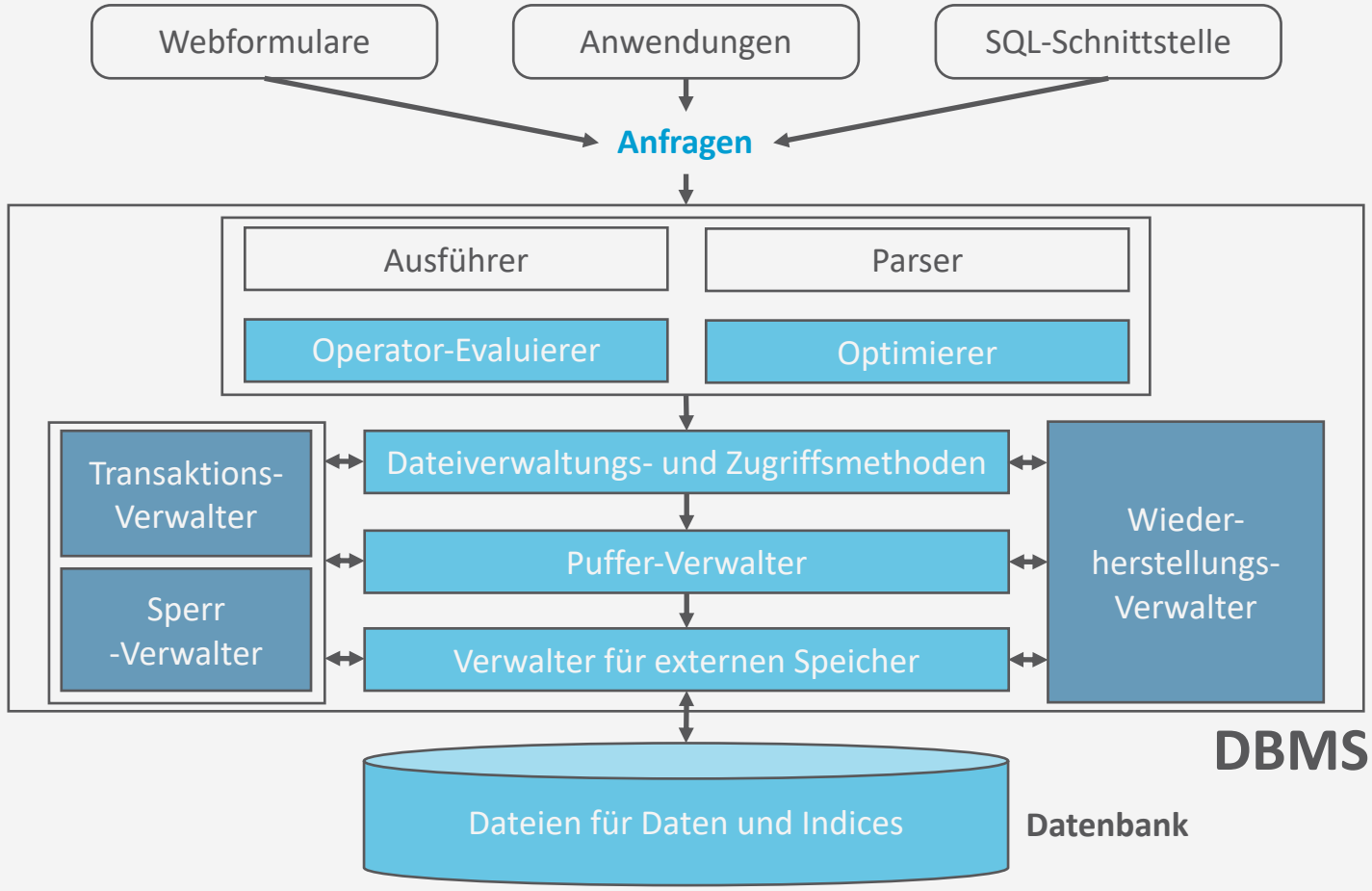
Erweiterte Systemumgebung

- Zugriffe von
  - DB-Administratoren
  - Anwendungen
  - Nutzer*innen
    - Direkt auf DB
      - Gelegentlich, interaktiv
    - Parametrisch
      - Vorgefertigte Anwendungsprogramme mit beschränktem Kommandovorrat (routinierte, wohldefinierte, formalisierte Befehle)



Architektur eines DBMS

- Ausblick: Hinter den Kulissen
  - Teil von 4. Relational Datenmodell – Relationale Algebra und 5. SQL
    - Anfragen stellen
    - Daten ändern
  - Teil von 6. Anfrageverarbeitung
    - Speicherung und Verwaltung der DB
    - Effiziente Umsetzung der SQL Befehle im DBMS
  - Teil von 7. Transaktionen
    - ACID-Umsetzung



Zwischenzusammenfassung

- DB-Sprachen
  - VDL, DDL, SDL und DML
  - SQL als Universalsprache
- DB-Systemumgebung
  - Manipulationen von Administrator*innen, Programmen, Nutzer*innen (gelegentlich, parametrisch)
  - DBMS
    - Anfrage: Ausführer, Parser, Operator-Evaluierer, Optimierer
    - Daten: Dateien, Verwalter für externen Speicher, Puffer, Dateiverwaltung, Zugriffsmethoden
    - (Mehr)benutzung: Transaktionen-, Sperr-, Wiederherstellungs-Verwalter
  - DBS = DBMS + DB

Überblick: 1. Einführung

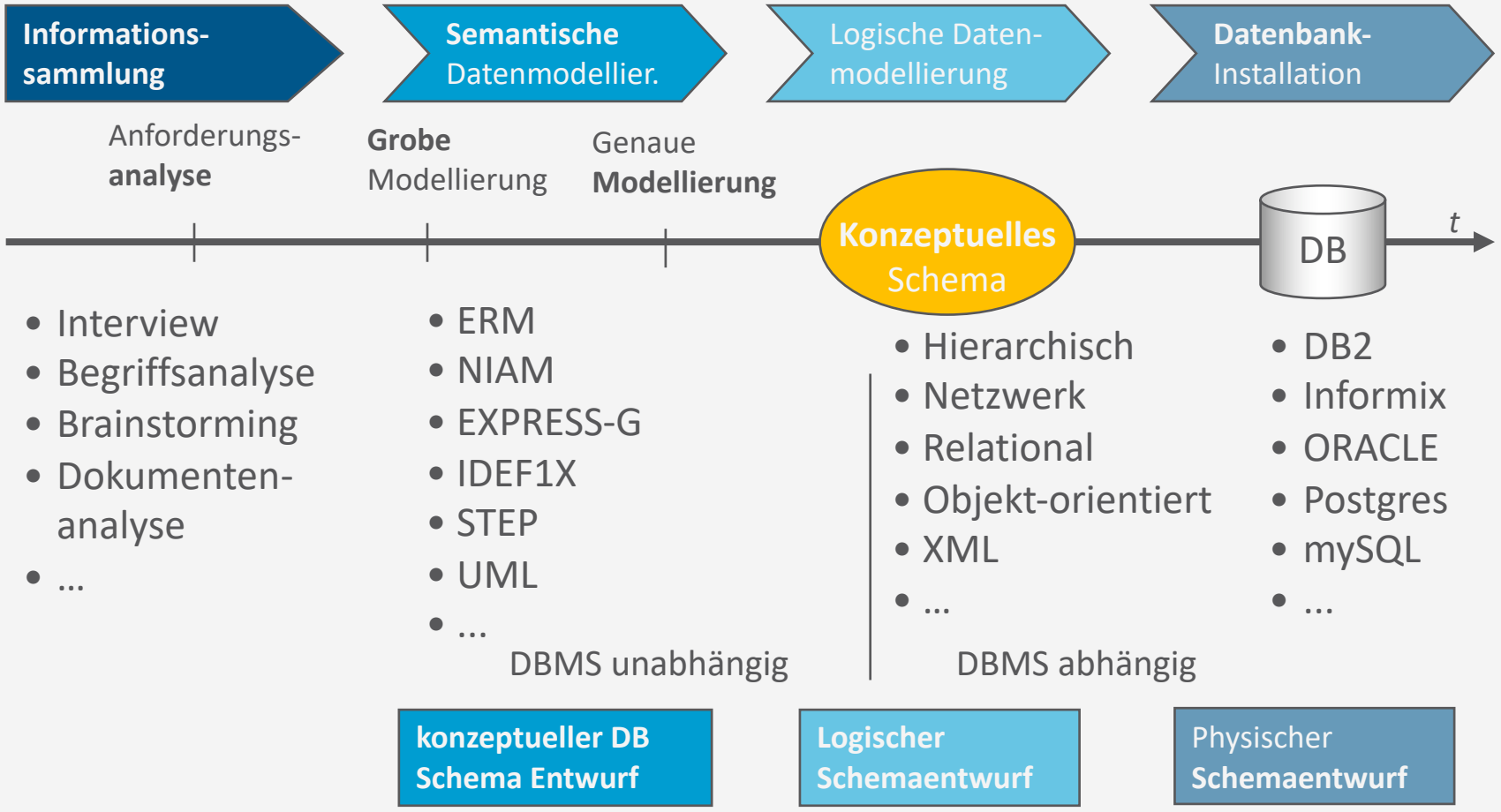
A. Datenbanken
- Datenbank (DB)
- Datenabstraktion, Datenmodelle, Datenunabhängigkeit
- Mehrbenutzersysteme

B. DB-Umgebungen
- DB-Sprachen
- Datenbanksystem (DBS)
- Datenbankmanagementsystem (DBMS)

C. Phasen des DB-Entwurfs
- Anforderung, Modellierung

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Zwischenzusammenfassung

- Phasen des DB-Entwurfs: Der Weg von den Anforderungen zur DB-Anwendung
- Von der Anwendung her
  - Informationssammlung
  - Semantische Modellierung
  - Logische Modellierung
  - Datenbankinstallation
    - Übergang zu hinter den Kulissen

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - *Noch offen*: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Überblick: 1. Einführung

A. Datenbanken
- Datenbank (DB)
- Datenabstraktion, Datenmodelle, Datenunabhängigkeit
- Mehrbenutzersysteme

B. DB-Umgebungen
- DB-Sprachen
- Datenbanksystem (DBS)
- Datenbankmanagementsystem (DBMS)

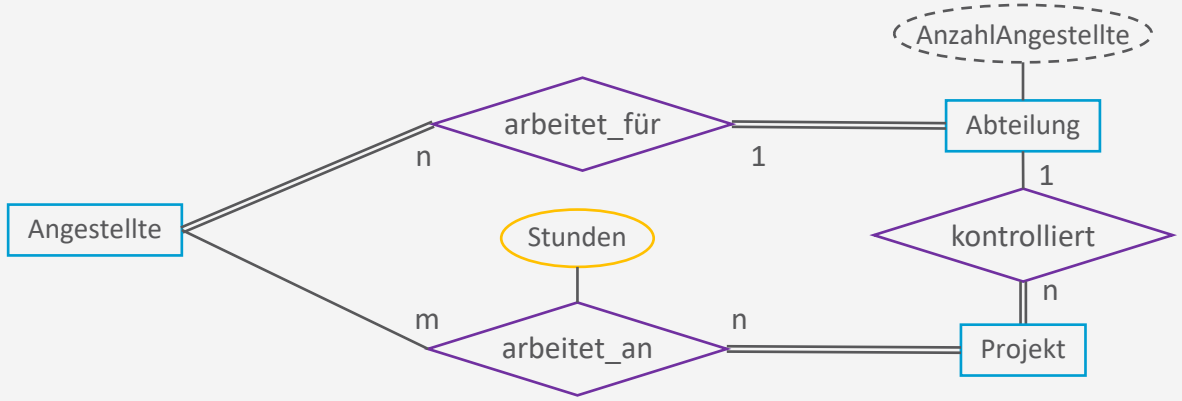
C. Phasen des DB-Entwurfs
- Anforderung, Modellierung

→ Datenbank-Modellierung

WWU
MÜNSTER

Datenbank-Modellierung

Datenbanken



wissen.leben

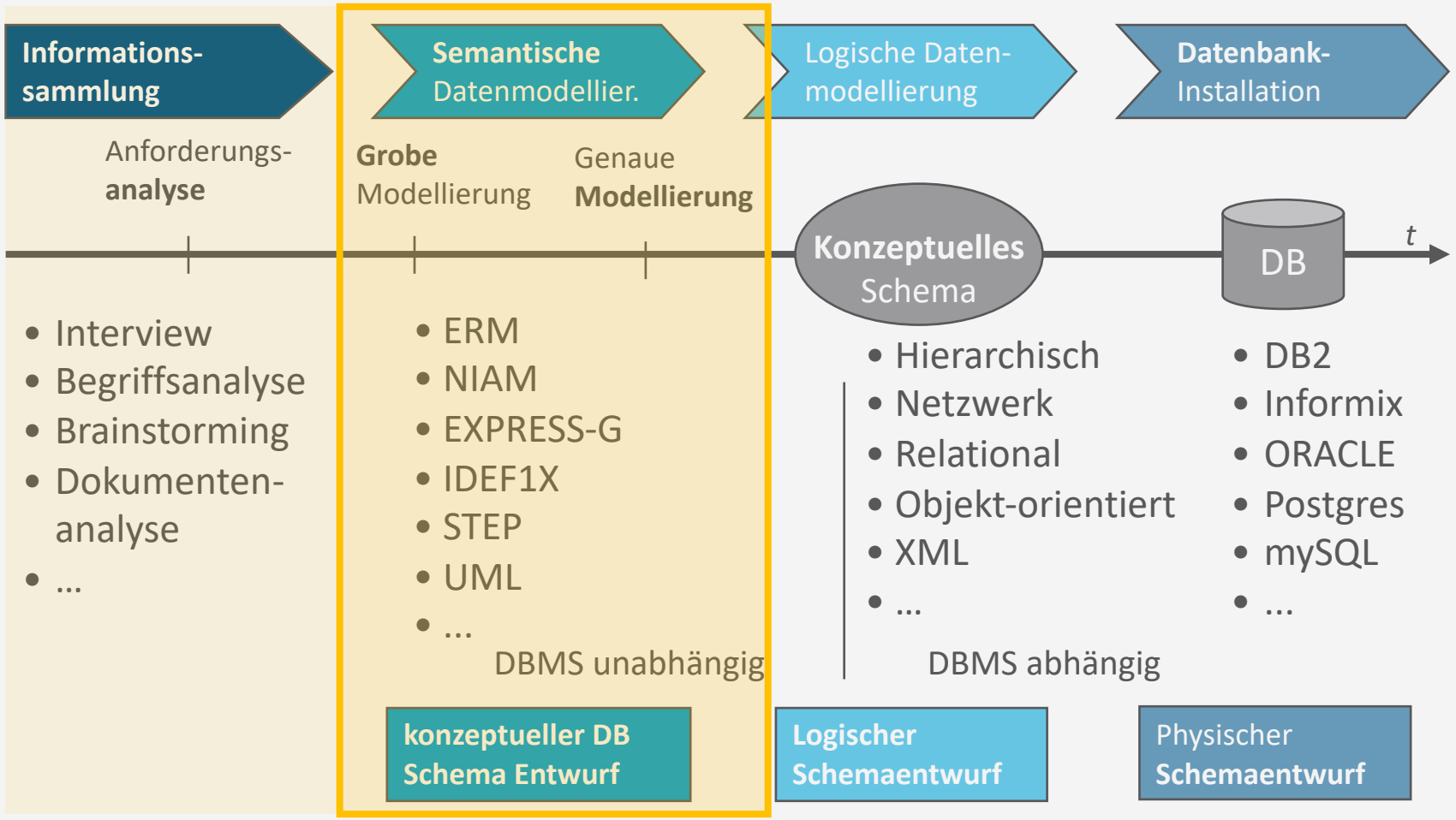
Tanya Braun
Arbeitsgruppe Date Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Überblick: 2. Datenbank-Modellierung

A. Motivation
- Modelle und Modellierung

B. Entity-Relationship-Modell (ER)
- Komponenten
- Von Anforderungen zum ER-Modell
- Interpretation von ER-Modellen

C. Enhanced ER-Modell (EER)
- Spezialisierung, Generalisierung
- Modellierungsvorgehen
- Beziehung zu UML

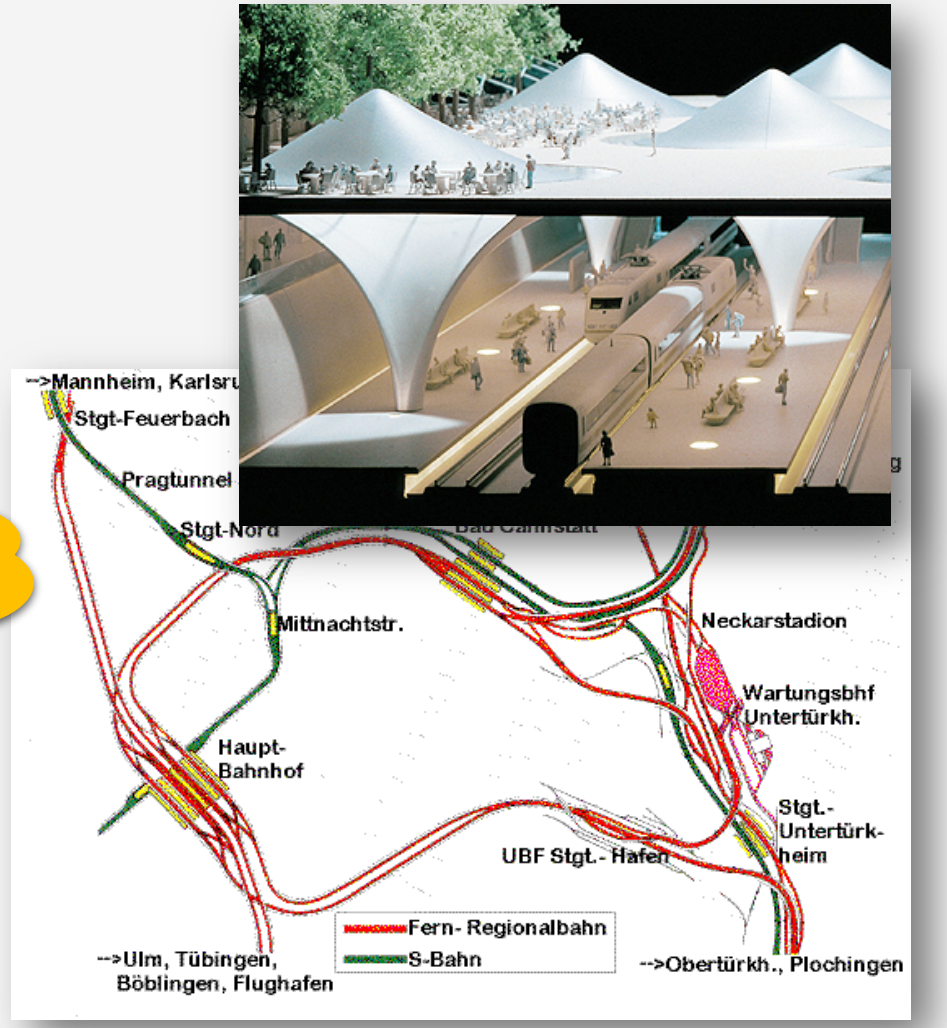
D. Dokumentation & Bewertung
- Dokumentation
- Qualitätskriterien EER

Modelle

- Modell: Abbild, Vorbild
- Typen von Modellen
  - Konkrete / abstrakte Modelle
    - Prototyp eines neuen Laptop / Erlösmodell
  - Konkrete / abstrakte Originale
    - Laptop / Personalentwicklung
- Was wird modelliert?
  - Struktur: Elemente eines Originals
  - Eigenschaften: Attribute der Elemente
  - Beziehungen zwischen Elementen
  - Verhalten: Dynamik der Elemente

Welche Modelle gibt es in der Informatik?

Was davon bildet ein DB-Modell (Schema) ab? Und was nicht?



Warum das Modell und nicht das Original?

- Motivation für Modellierungen
  - Operationen, die am Original nicht oder nur mit größerem Aufwand durchführbar sind
    - Ermöglichen Simulation
  - Untersuchen und Verstehen komplexer Zusammenhänge
  - Kommunikationsgrundlage
  - Fixieren von Anforderungen
- Verwendung bestimmt, was modelliert wird und was nicht:
  - Gebäudemodell: optischer Eindruck
  - Grundriss: Grundstücks- und Raumeinteilung
  - Gewerkeplan: Bauabwicklung
  - Kostenplan: Finanzierung
- Verwendung sollte auch die Modellierungssprache bestimmen

Überblick: 2. Datenbank-Modellierung

A. Motivation
- Modelle und Modellierung

B. Entity-Relationship-Modell (ER)
- Komponenten
- Von Anforderungen zum ER-Modell
- Interpretation von ER-Modellen

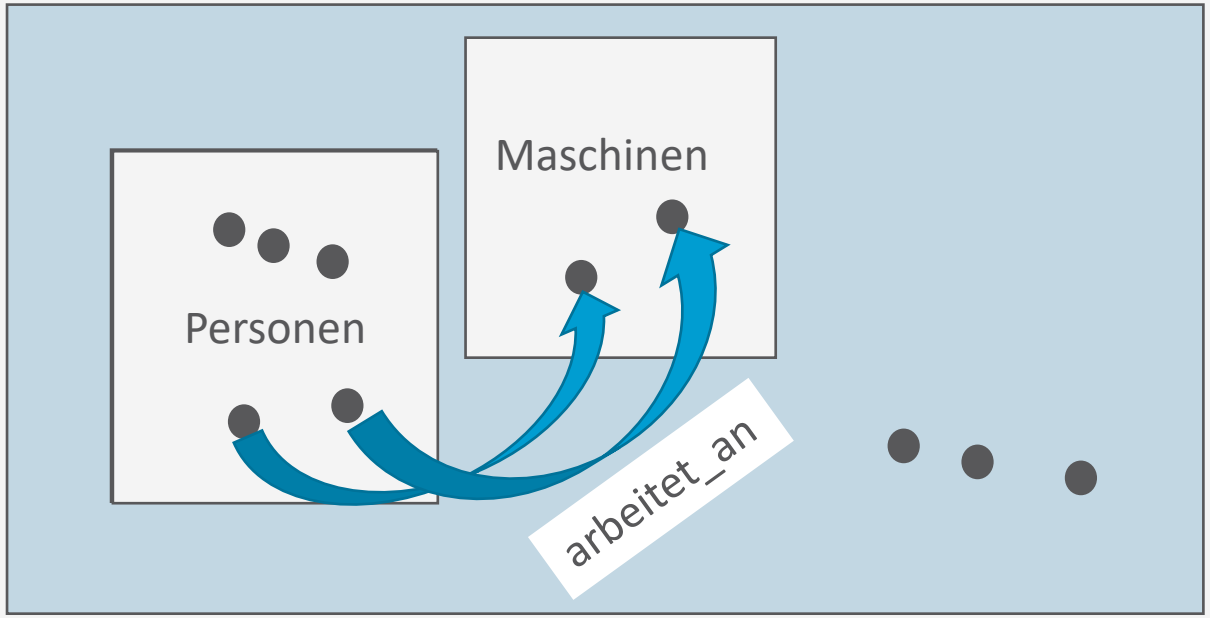
C. Enhanced ER-Modell (EER)
- Spezialisierung, Generalisierung
- Modellierungsvorgehen
- Beziehung zu UML

D. Dokumentation & Bewertung
- Dokumentation
- Qualitätskriterien EER

ER-Modellierung

- Objekte (Entitäten) mit ähnlichen Eigenschaften zu Mengen (Entitätstypen, Klassen) zusammenfassen
  - Jedes Objekt ist „Instanz“ einer oder mehrerer Klassen
- Extension (Menge aller Instanzen einer Klasse)
- Objekte können in Beziehung gesetzt werden (Beziehungstyp, Relationship)
- ER-Modell wird durch einen Graphen dargestellt

Was für Komponenten brauchen wir?



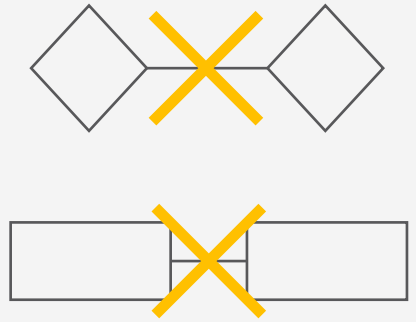
Alle Objekte (Universum)

● steht für Objekt oder Entität

Grundlegende Elemente von ER-Diagrammen

- Objekttyp 
  - Graphische Darstellung: Rechteck
  - Auch Entitätstyp oder Klasse genannt
  - Menge von Objekten
- Werttyp 
  - Graphische Darstellung: Ellipse
  - Für Basisdatentypen, Attribute
  - Menge von Werten bzw. Literalen
- Beziehungstyp 
  - Graphische Darstellung: Raute
  - Relationen zwischen Objekttypen
  - Menge von Tupeln von Objekten

- Elemente von ER-Diagrammen bilden einen bipartiten Graphen:
  - Verbindungen zwischen Symbolen der gleichen Typen sind nicht erlaubt

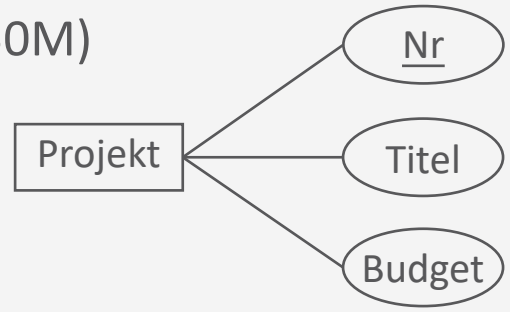


P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. In *ACM Transactions on Database Systems*, vol. 1(1), pp. 9–36, 1976.

Objekte/Entitäten und Attribute

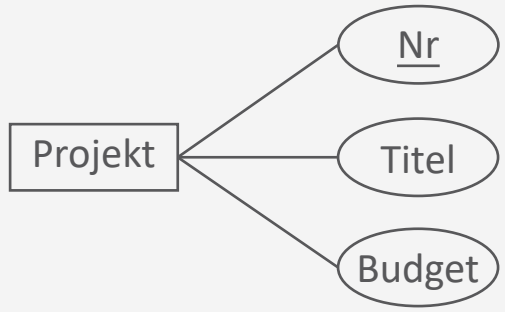
- Entitäten
  - Wesentliche Konzepte („abgrenzbare Dinge“) realer oder gedachter Miniwelten
  - Haben eine eigenständige Existenz
- Attribute beschreiben Eigenschaften von Entitäten
- Mathematische Bedeutung:
  Menge von Tupeln von Werten
  - Tupel = „Aggregat“ von Basiswerten
- Schlüssel: Attribute, die Tupel eindeutig kennzeichnen
  - In der Graphik sind diese Attribute unterstrichen
  - Entitäten mit eigenem Schlüssel = starke Entität

- Beispiel:
  - Entität: Projekt
  - Attribute: Projekt beschrieben durch
    - Eine Nummer
    - Einen Titel
    - Das Budget
  - Als Menge von Tupeln:
    - (<u>42</u>, Flughafen, 100M)
    - (<u>21</u>, Bahnhof, 50M)



Identifikation und Schlüssel

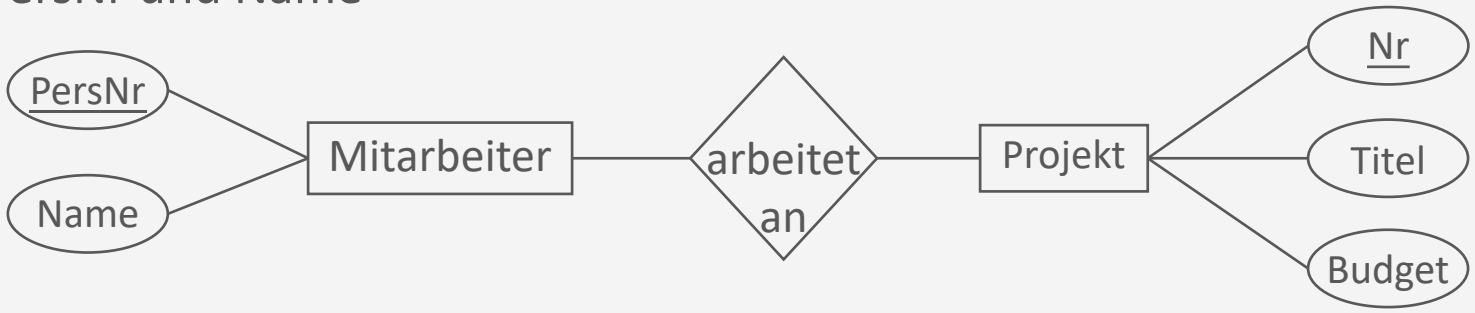
- Identifikation: Zwei grundlegende Ansätze in DB-Modellen
  - Referentielle Identifikation
    - Direkte Verweise auf Objekte (Zeiger in Programmiersprachen)
  - Assoziative Identifikation
    - Werte von Attributen oder Attributkombinationen, um sich eindeutig auf Objekte zu beziehen (Schlüssel: Ausweisnummer, Fahrgestellnummer, ...)
    - Schlüssel
      - Attribute oder Attributkombinationen mit innerhalb einer Klasse eindeutigen Werten
      - Mehrere Schlüsselkandidaten möglich (dazu mehr später)



Assoziation/Relationship

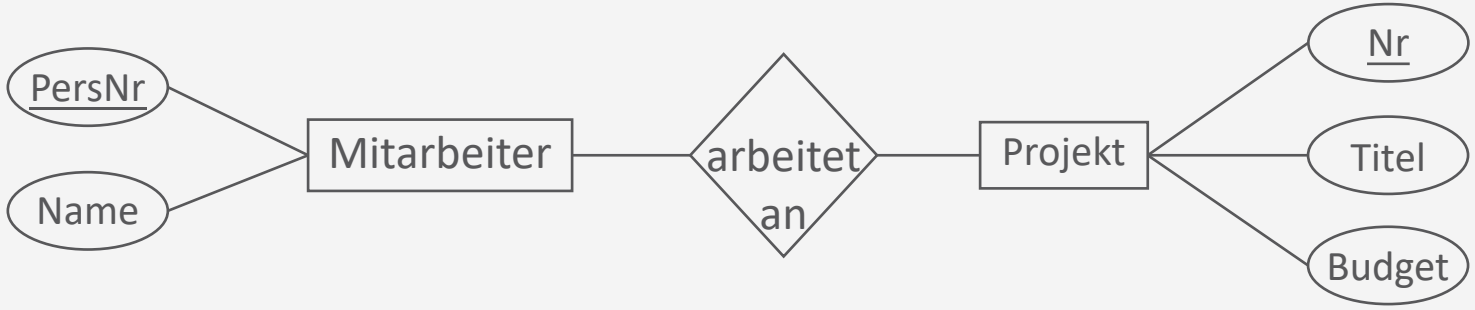
- Objekte können miteinander in Beziehung gesetzt (assoziiert) werden:
  - Binäre (ternäre, ...) Beziehungen assoziieren zwei (drei, ...) Klassen oder Objekte
  - Allgemein: n-äre Beziehungen zwischen n Klassen oder Objekten
    - n Grad der Beziehung
  - Mathematische Bedeutung: Menge an verknüpften Tupeln

- Beispiel
  - An Projekten arbeiten Mitarbeiter
    - Mitarbeiter: Entität mit Attributen PersNr und Name
    - Verknüpftes Tupel:
      ((4711, Mustermann),
      (42, Flughafen, 100M))



Assoziation/Relationship

- Schlüssel um Objekte in Assoziationen zu identifizieren
  - Schlüssel stellen als Attributwerte Beziehungen zu anderen Objekten her
  - Referenzierte Objekte müssen existieren (→ referentielle Integrität)
  - Erlauben vereinfachter Referenzierung in Tupeln
- Beispiel
  - Attribut „Nr“ Schlüssel für Projekte
  - Attribut „PersNr“ Schlüssel für Mitarbeiter
  - Ursprüngliches Tupel
    - ((4711, Mustermann),
      (42, Flughafen, 100M))
  - Unter Zuhilfenahme der Schlüssel:
    - (4711, 42)



Identifikation und referentielle Integrität

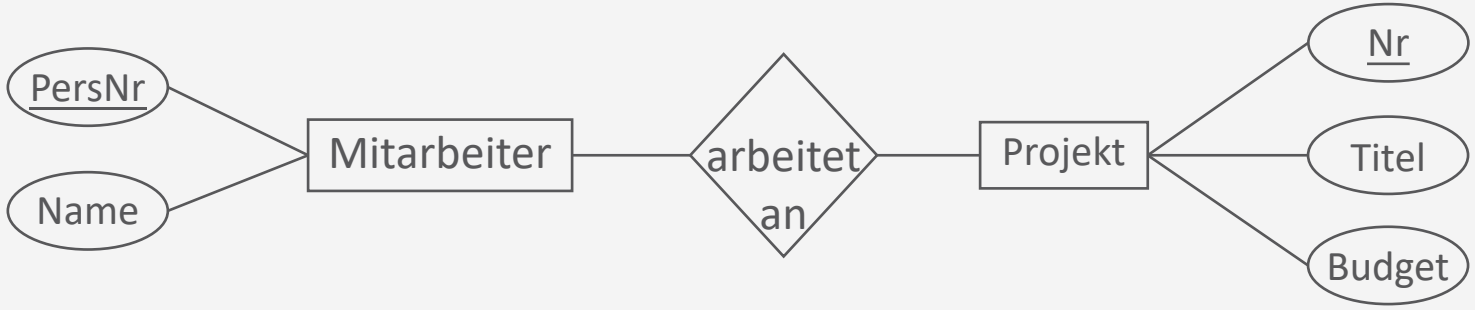
- Beispiel: Projekt
  - Projekte werden durch eine Nummer eindeutig identifiziert
  - Zwei Möglichkeiten zur Identifikation von Projekten innerhalb der Assoziation „arbeitet an“
  - Annahme: Projekt 54 existiert nicht → referentielle Integrität verletzt!

Referentielle Identifikation

	Projekt	Mitarbeiter
		...
		...
NULL		...
	...	...

Assoziative Identifikation

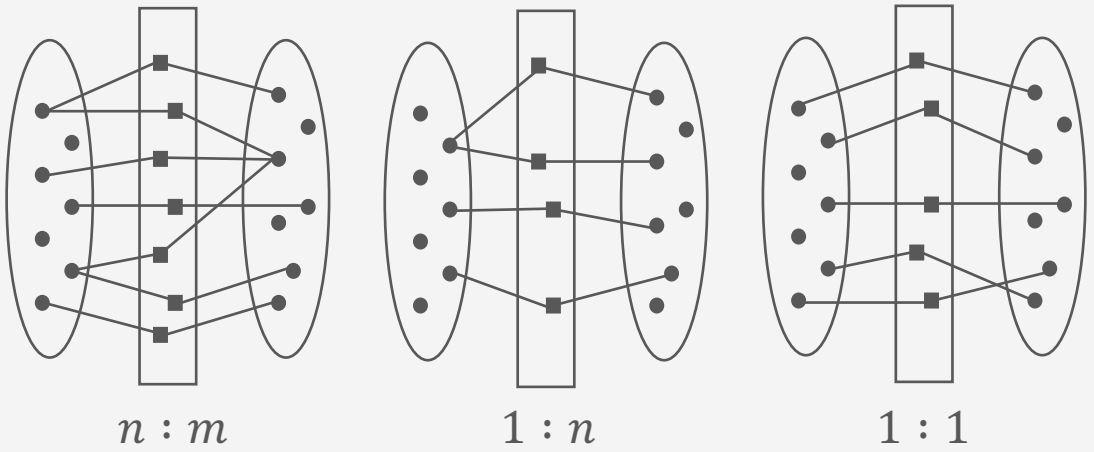
Projekt	Mitarbeiter
42	...
21	...
54	...
....	...

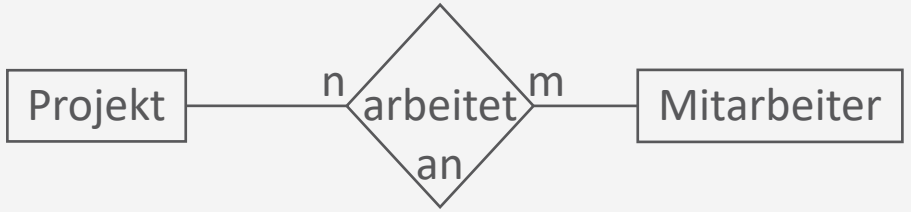


Funktionalitäten

Beispiele für $1:n, 1:1$?

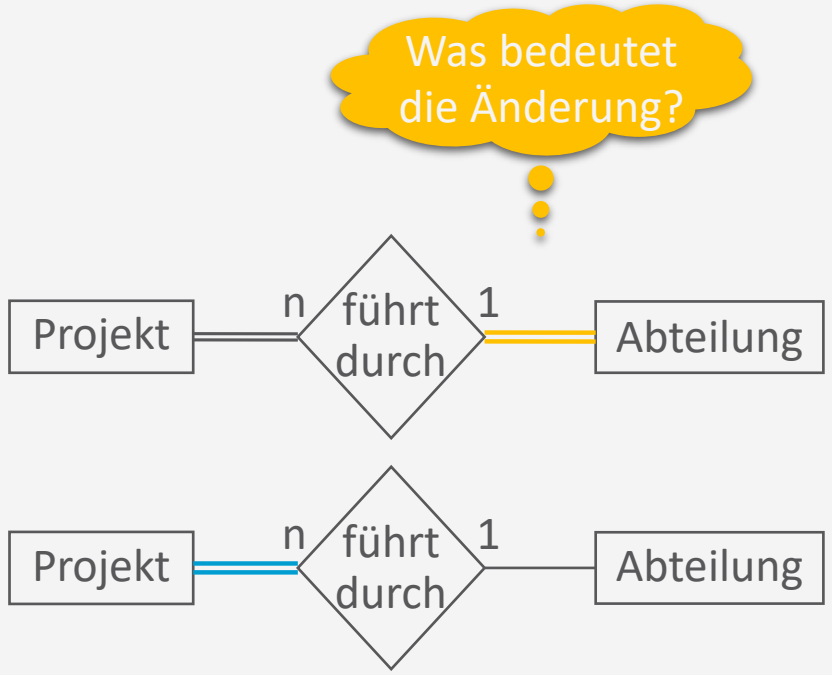
- Funktionalitätsangaben definieren Einschränkungen:
  - Varianten sind $n:m, 1:n, 1:1$
  - Gegeben $x:y$,
    - Jede Instanz der ersten Klasse kann mit bis zu y Instanzen der zweiten Klasse assoziiert werden
    - Jede Instanz der zweiten Klasse mit bis zu x Instanzen der ersten Klasse assoziiert werden
- Beispiel
  - An Projekten arbeiten Mitarbeiter
    - Ein Mitarbeiter kann an mehreren (n) Projekten arbeiten
    - Jedes Projekt wird von beliebig vielen (m) Mitarbeitern bearbeitet





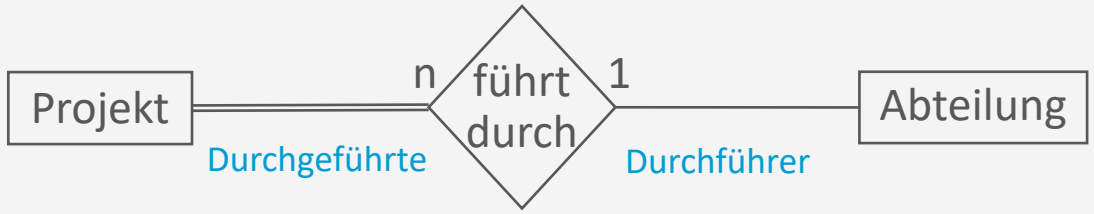
Partizipation

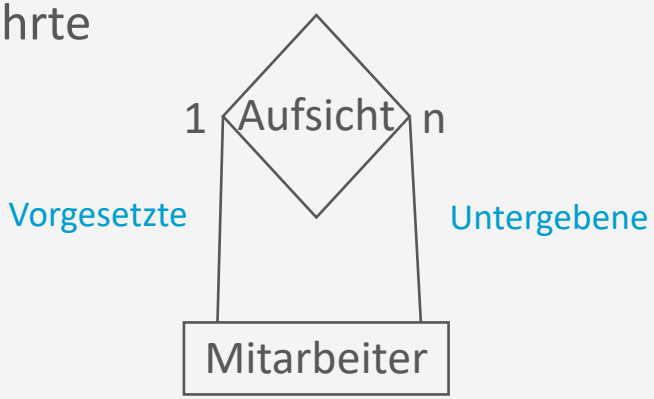
- Beteiligung an einer Beziehung
- Formen:
  - Totale Partizipation: Jede Instanz einer Klasse muss mit einer Instanz der zweiten Klasse in Beziehung stehen (═══)
  - Partielle Partizipation: Eine Instanz einer Klasse kann in Beziehung zu einer Instanz der zweiten Klasse stehen (———)
- Beispiel:
  - Projekte werden von Abteilungen durchgeführt
  - Jedes Projekt muss einer Abteilung zugeordnet sein
  - Eine Abteilung kann mehrere Projekte ausführen.



Partizipation: Rollennamen

- Rollennamen (Namen für die Argumente der Relation) identifizieren die Menge der Instanzen, die mit einer anderen Instanz in Beziehung stehen.
  - Rollen können als abgeleitete Attribute verstanden werden, die die Menge der Instanzen als Attributwerte besitzen
  - Besonders bei rekursiven Assoziationen eingesetzt
  - Beispiele:
    - Abteilungen als Durchführer von Projekten als Durchgeführte
    - Rekursiv: Vorgesetzte und Untergebene

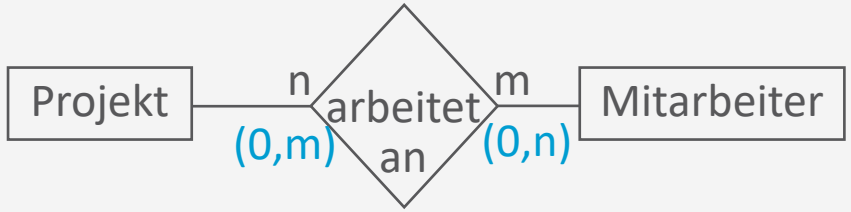


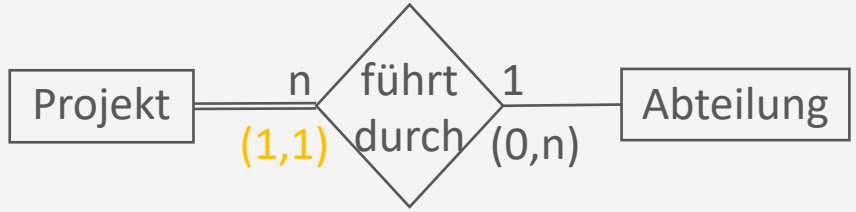


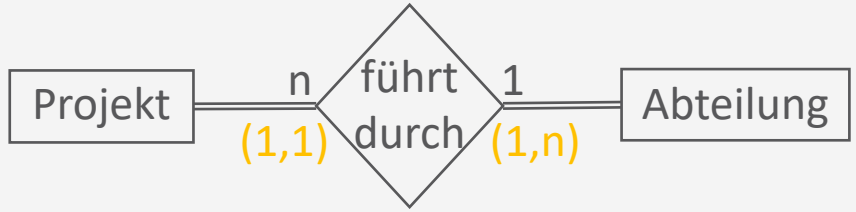
Funktionalitäten (Reprise)

- Explizite Angabe der Kardinalitäten über (min, max)-Kardinalitäten
  - Instanz ist mit min bis max Instanzen assoziiert
  - Leider invers zu $n : m$ Funktionalitäten an den Kanten
  - Erlaubt totale Partizipation direkt anzugeben
    - Durch min=1
    - Doppelte Striche dann nicht mehr nötig
    - Beispiele
      - Totale Partizipation von Projekt
      - Totale Partizipation beider Entitäten

Bemerkung: In der Literatur findet man weitere Beschriftungsregeln.

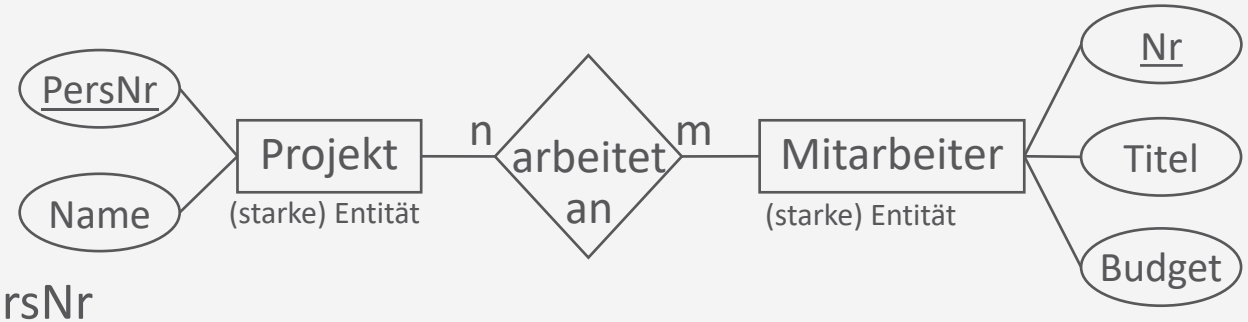






Schwache, existenzabhängige Entitäten

- Starke Entitäten
  - Besitzen ausgezeichnete Attribute zur eindeutigen Identifikation (Schlüssel)
  - Beispiel:
    - Projekt Schlüssel: Nr, Mitarbeiter Schlüssel: PersNr

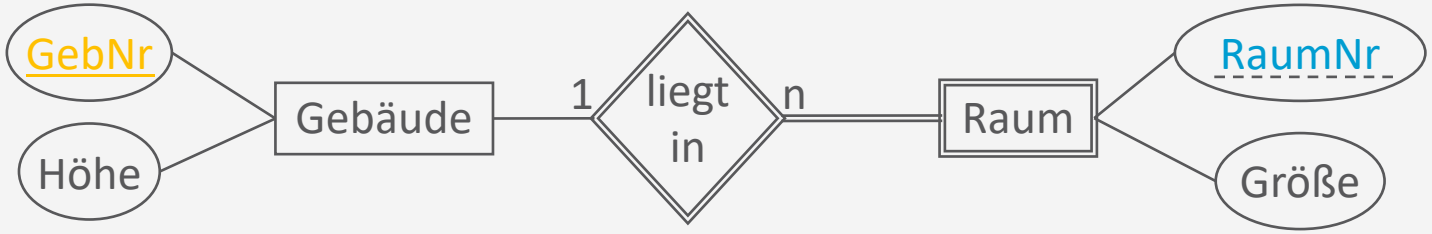


- Schwache Entitäten
  - Benötigen identifizierende Einheit zur Identifikation
  - Starke Entität mit schwacher Entität in Beziehung
    - Totale Partizipation der schwachen Entität nötig
  - Beispiel:
    - Beziehung „Angehörige von“ stellt die Verbindung zur identifizierenden Einheit (Angestellter) her

Schwache Entitäten und Schlüssel

- Beziehung zwischen starken und schwachem Typ ist immer 1:n (oder 1:1 in seltenen Fällen)
- Schwache Entitäten haben partiellen Schlüssel
  - Wird eindeutig mit Schlüssel der identifizierenden Entität
  - Beispiel:
    - RaumNr ist nur innerhalb eines Gebäudes eindeutig
    - Eindeutiger Schlüssel ist: GebNr und RaumNr

Warum kann das keine n:m-Beziehung sein?



Von der Anforderungsdefinition

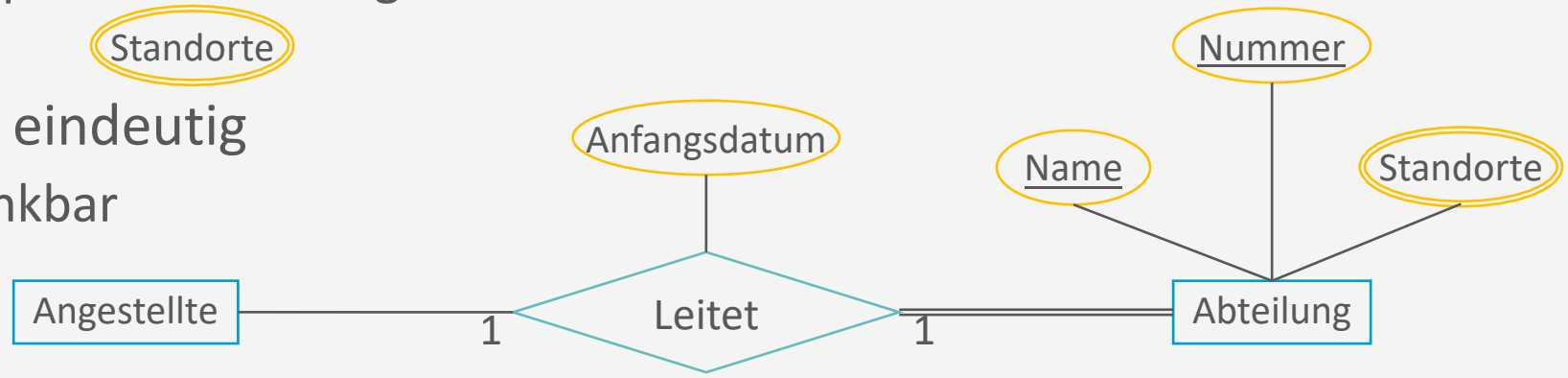
Zum ER-Diagramm

Anwendungsdefinition „Firma“ …

1. Die Firma ist in Abteilungen organisiert. Jede Abteilung hat eine eindeutige Bezeichnung, eine eindeutige Nummer und einen bestimmten Angestellten, der die Abteilung leitet. Es wird das Anfangsdatum verfolgt, ab dem dieser Angestellte die Leitung der Abteilung übernommen hat. Eine Abteilung verfügt über mehrere Standorte.
2. Eine Abteilung kontrolliert eine Reihe von Projekten, die jeweils einen Namen, eine eindeutige Nummer und einen einzigen Standort haben.
3. Zu jedem Angestellten sollen Name, Sozialversicherungsnummer, Adresse, Gehalt, Geschlecht und Geburtsdatum gespeichert werden. Ein Angestellter wird einer Abteilung zugewiesen, kann aber an mehreren Projekten arbeiten, die nicht unbedingt alle von der gleichen Abteilung kontrolliert werden. Dabei wird die Stundenzahl pro Woche verfolgt, die ein Angestellter an jedem Projekt arbeitet. Es wird auch der unmittelbare Vorgesetzte eines Angestellten gespeichert.
4. Zu Versicherungszwecken sollen die Familienangehörigen jedes Mitarbeiters gespeichert werden. Hierzu werden für jeden Angehörigen Vorname, Geschlecht, Geburtsdatum und Verwandtschaftsgrad zum jeweiligen Angestellten erfasst.

Anwendungsdefinition „Firma“ …

1. Die Firma ist in Abteilungen organisiert. Jede Abteilung hat eine eindeutige Bezeichnung, eine eindeutige Nummer und einen bestimmten Angestellten, der die Abteilung leitet. Es wird das Anfangsdatum verfolgt, ab dem dieser Angestellte die Leitung der Abteilung übernommen hat. Eine Abteilung verfügt über mehrere Standorte.
   - Interpretation: totale Partizipation der Abteilung
- Mehrwertige Attribute:
  - Graphische Darstellung: doppelte Umrandung
  - Mehrere Orte pro Abteilung
- Anforderungen nicht immer eindeutig
  - Standort auch als Entität denkbar

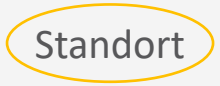


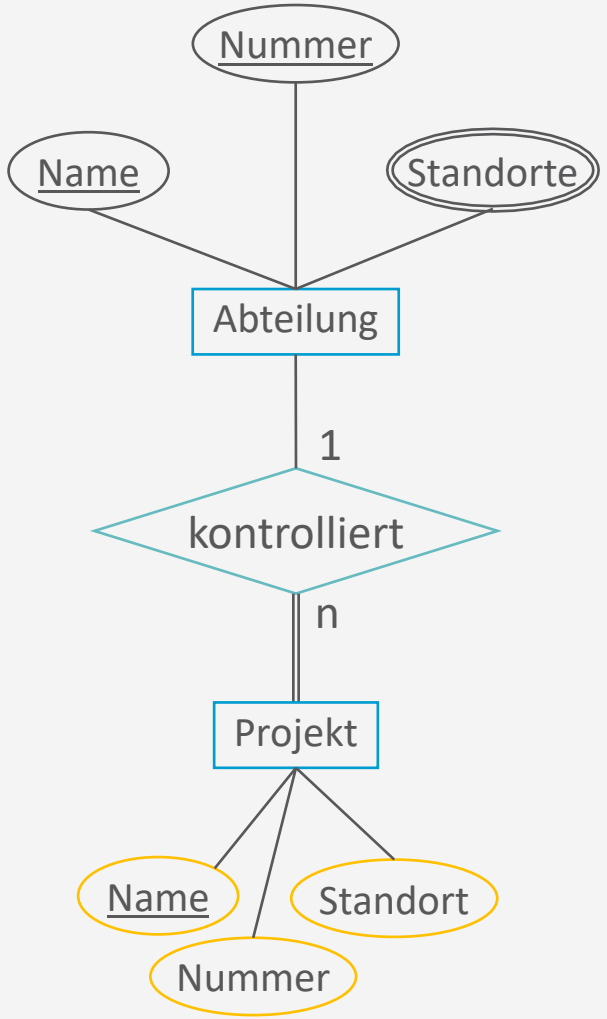
Anwendungsdefinition „Firma“ …

2. Eine Abteilung kontrolliert eine Reihe von Projekten, die jeweils einen Namen, eine eindeutige Nummer und einen einzigen Standort haben.
   - Interpretation: totale Partizipation der Projekte

- Mehrwertige vs. einwertige Attribute
  - Standorte: mehrere Orte pro Abteilung

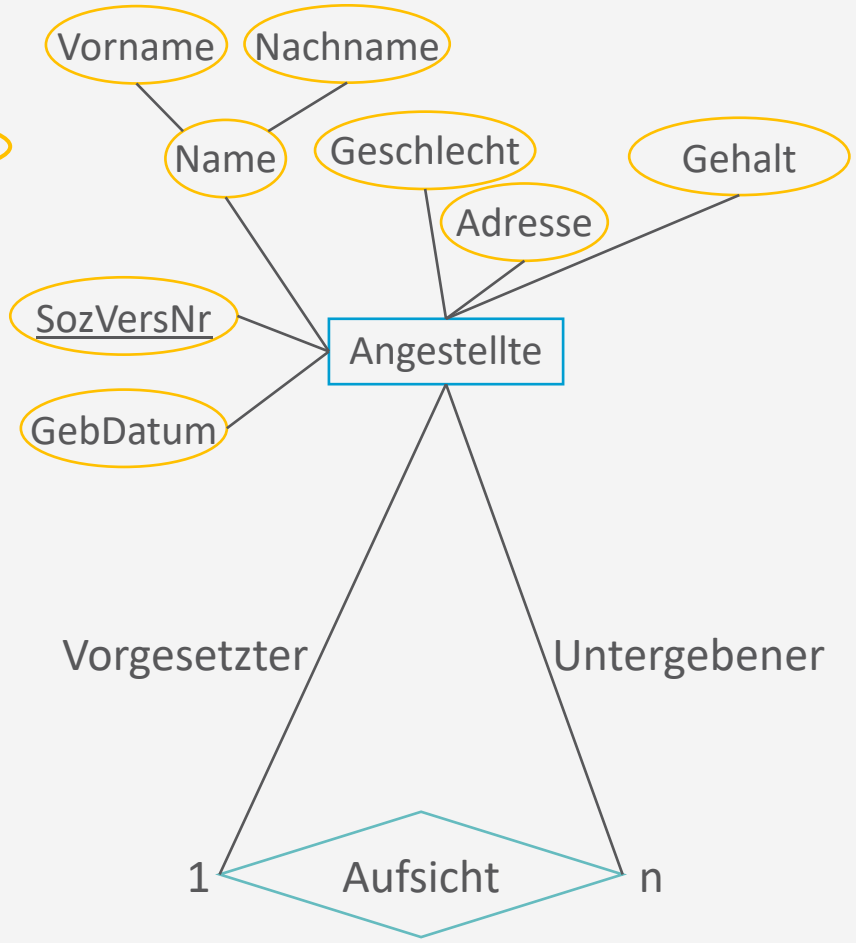
  - Standort: ein Ort pro Projekt





Anwendungsdefinition „Firma“ …

3. Zu jedem Angestellten sollen Name, Sozialversicherungsnummer, Adresse, Gehalt, Geschlecht und Geburtsdatum gespeichert werden. … Es wird auch der unmittelbare Vorgesetzte eines Angestellten gespeichert.
   - Interpretation: Name als mehrwertiges Attribut, auch bei anderen Attributen denkbar

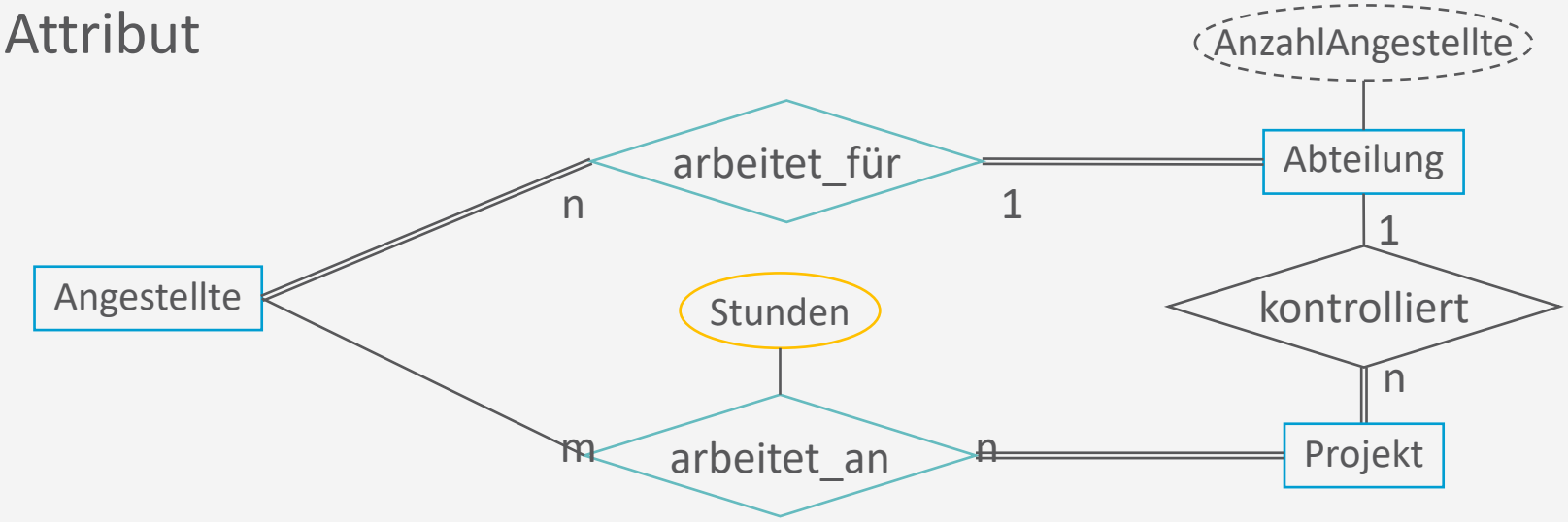


- *Neues Konzept*: Atomare vs. zusammengesetzte Attribute
  - Atomar: unteilbar (nicht sinnvoll zerlegbar); z.B. SozVersNr
  - Zusammengesetzt: aus mehreren Attributen bestehend; z.B. Name besteht aus Vorname und Nachname

Anwendungsdefinition „Firma“ …

3. … Ein Angestellter wird einer Abteilung zugewiesen, kann aber an mehreren Projekten arbeiten, die nicht unbedingt alle von der gleichen Abteilung kontrolliert werden. Dabei wird die Stundenzahl pro Woche verfolgt, die ein Angestellter an jedem Projekt arbeitet…
   - Interpretation: totale Partizipation der Abteilung und der Projekte in den Beziehungen zu Angestellten
- Neues Konzept: abgeleitetes Attribut
  - Kann berechnet werden
  - Beispiel:

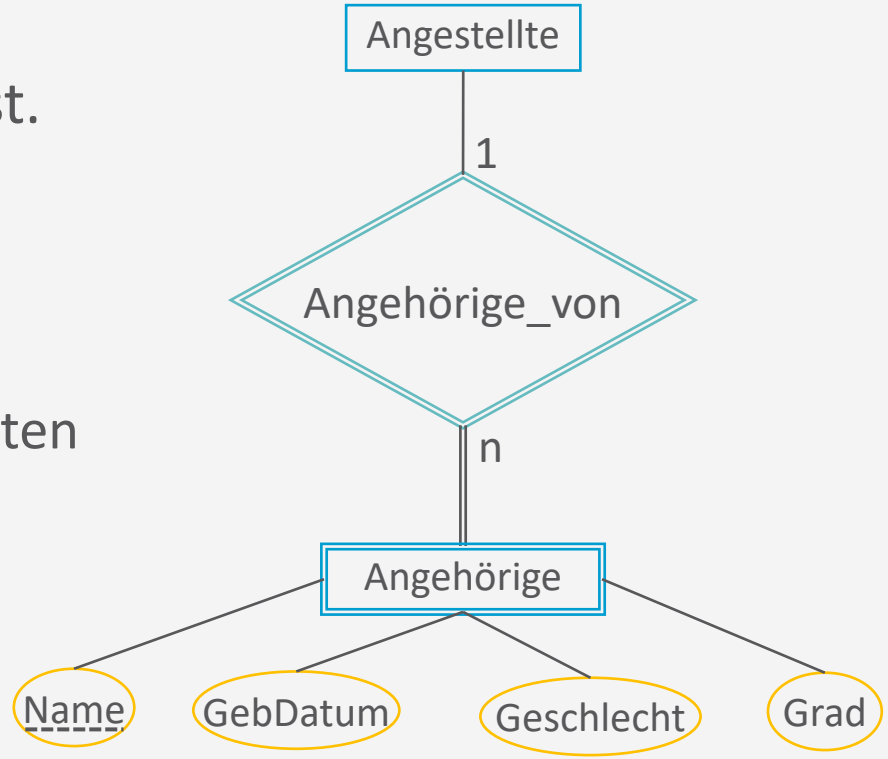
AnzahlAngestellte

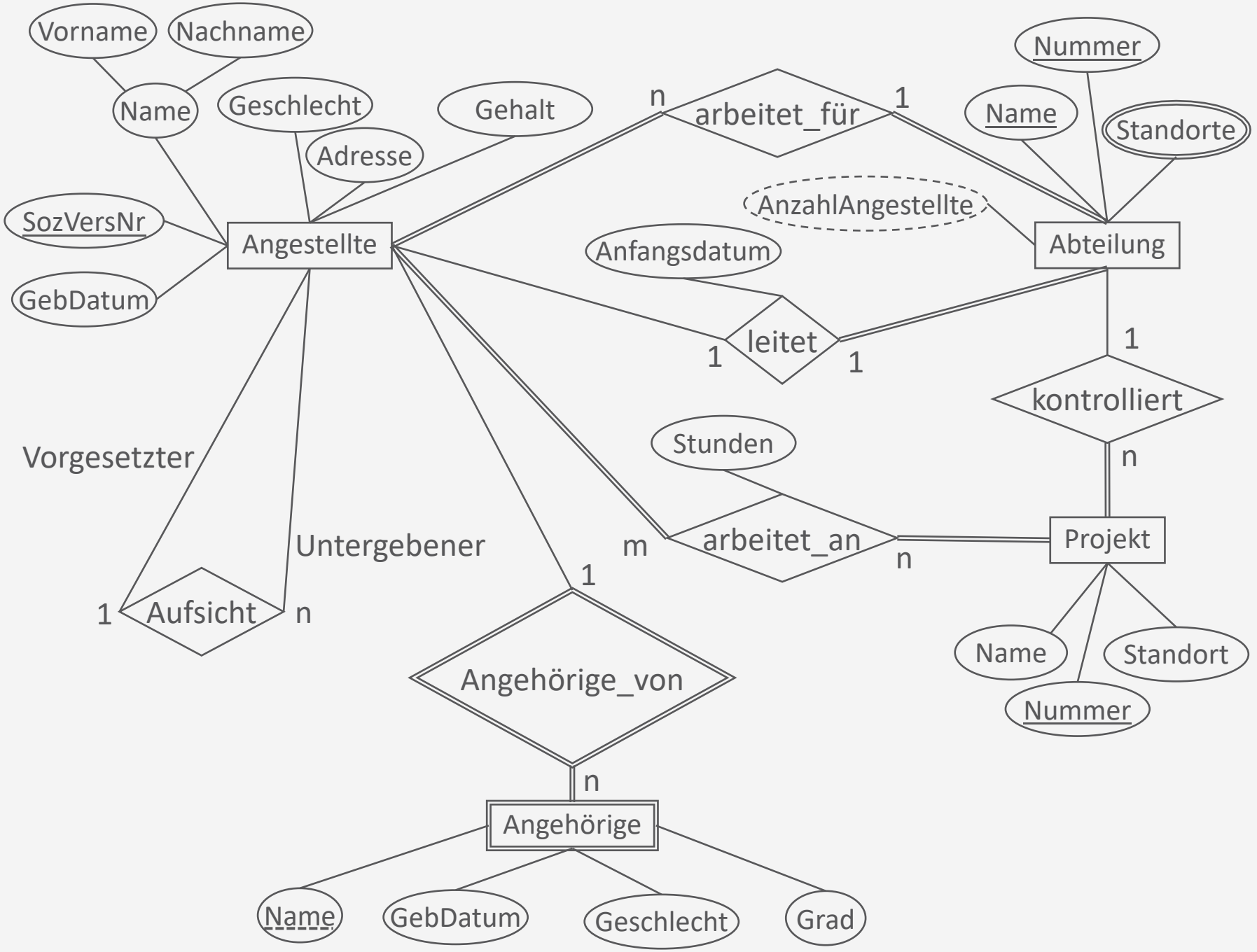


Anwendungsdefinition „Firma“ …

4. Zu Versicherungszwecken sollen die Familienangehörigen jedes Mitarbeiters gespeichert werden. Hierzu werden für jeden Angehörigen Vorname, Geschlecht, Geburtsdatum und Verwandtschaftsgrad zum jeweiligen Angestellten erfasst.

- Partielle Schlüsselattribute Name
  - Name ist nicht eindeutig ohne den dazugehörigen Angestellten

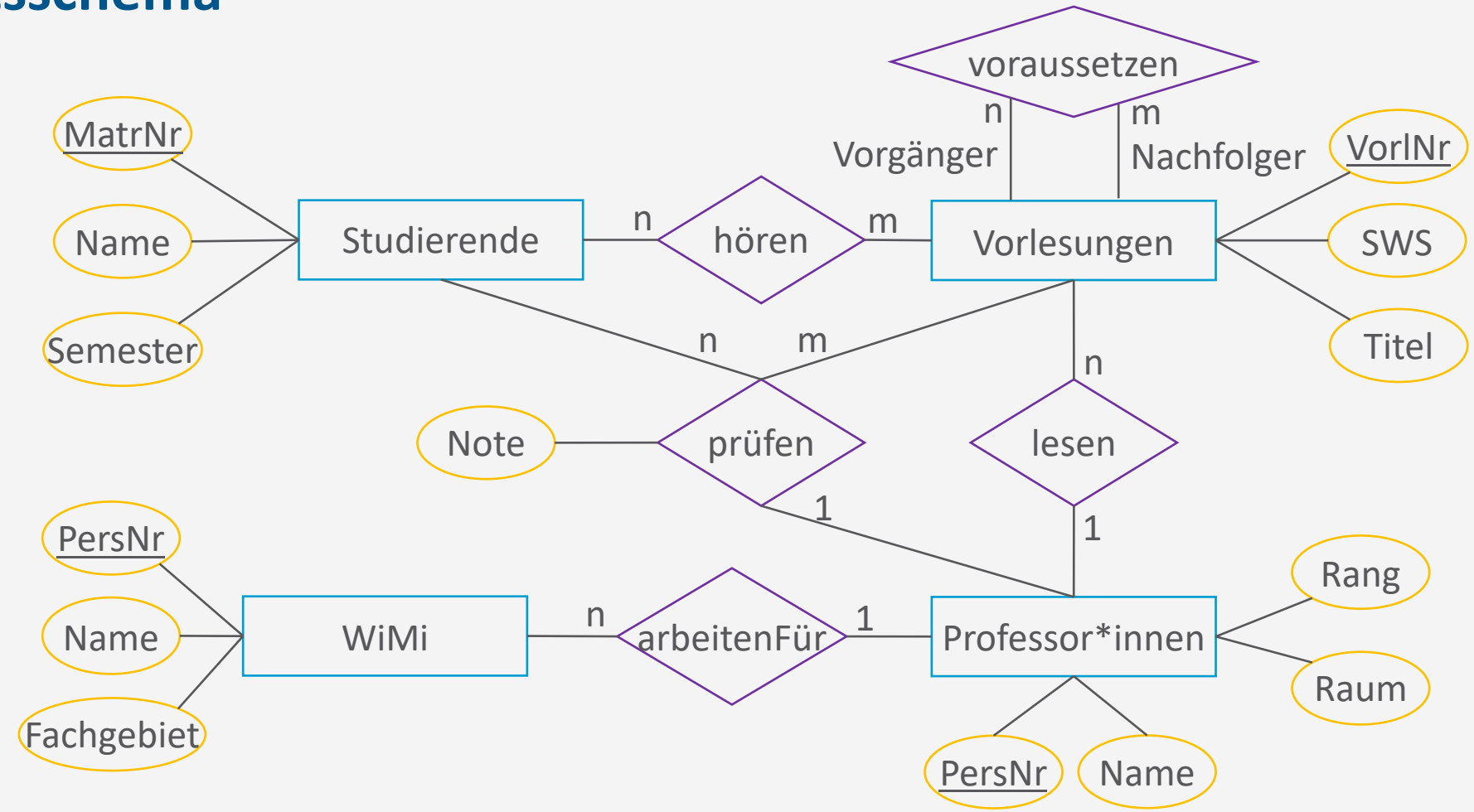



Vorname
Nachname
Name
Geschlecht
Adresse
Gehalt
SozVersNr
GebDatum
Angestellte
n
arbeitet_für
1
Nummer
Name
Standorte
AnzahlAngestellte
Abteilung
Anfangsdatum
leitet
1
1
1
kontrolliert
n
Stunden
Vorgesetzter
Untergebener
1
Aufsicht
n
m
arbeitet_an
n
Projekt
1
Angehörige_von
n
Name
Standort
Nummer
Angehörige
Name
GebDatum
Geschlecht
Grad

Lesen von ER-Diagrammen

Interpretation eines gegebenen ER-Diagramms

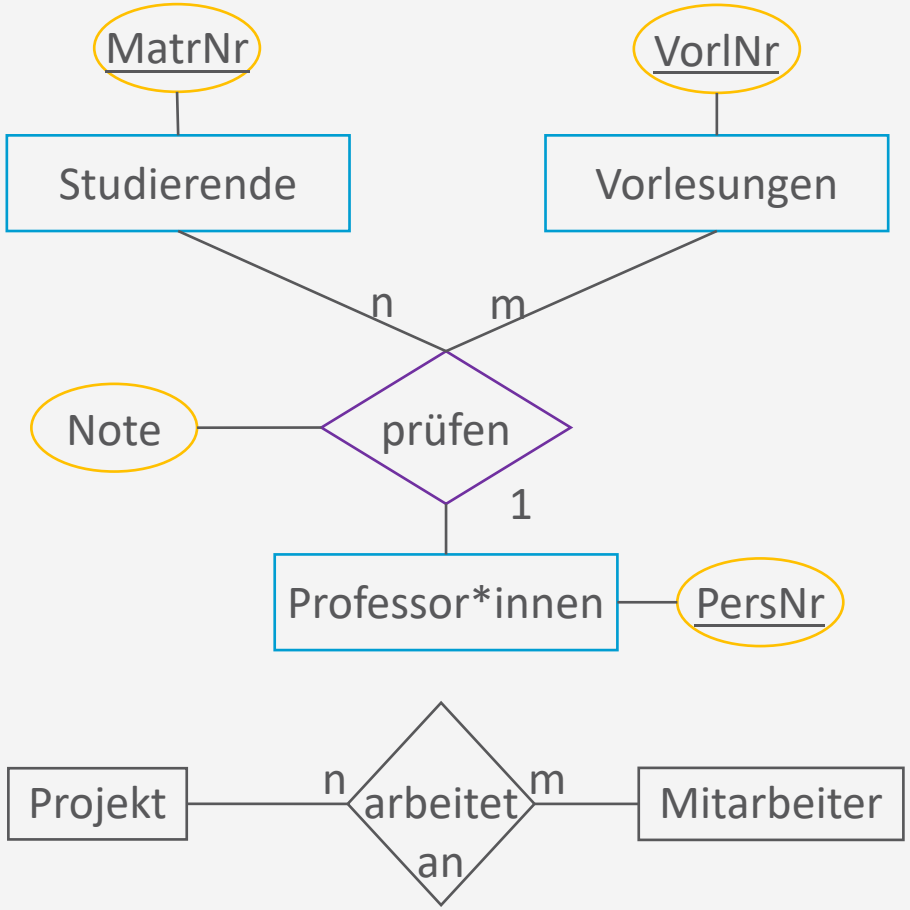
Universitätsschema



Beziehungen mit Grad > 2 werden auch n-äre Beziehungen genannt (wie auch hier auf Folie 12). Um Verwechslung mit dem n aus n:m zu vermeiden, heißt es hier k-äre Beziehung.

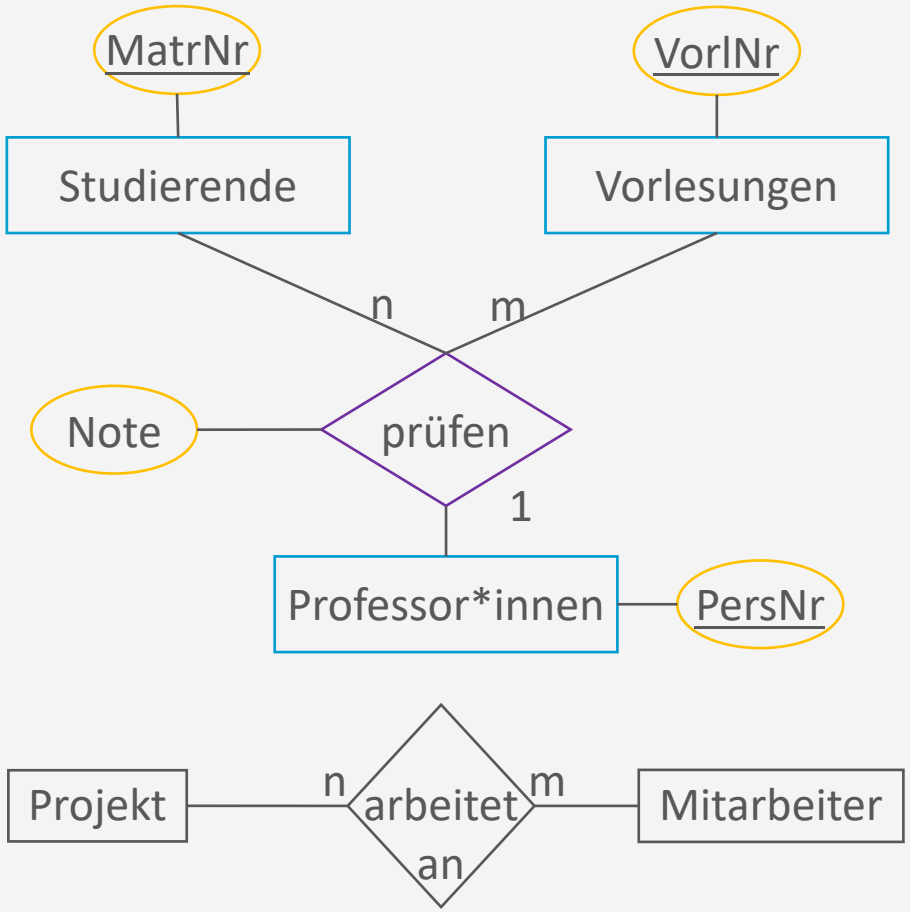
Funktionalitäten (Reprise)

- Funktionalitäten bei binären Beziehungen: n:m, 1:n, 1:1
- Generalisierung:
  Funktionalitätsangaben bei k-ären Beziehungen
  - Funktionalität zwischen einem Tupel von $k-1$ Entitäten und des verbliebenen Entitätstyps
  - Beispiel
    - Ternäre Beziehung „prüfen“
      - Ein*e Studierende*r in einer Vorlesung kann von einem*r Professor*in geprüft werden → 1
      - Ein*e Professor*in in einer Vorlesung kann mehrere Studierende prüfen → n
      - Ein*e Professor*in kann eine*n Studierende*n in mehreren Vorlesungen prüfen → m (≠ n)

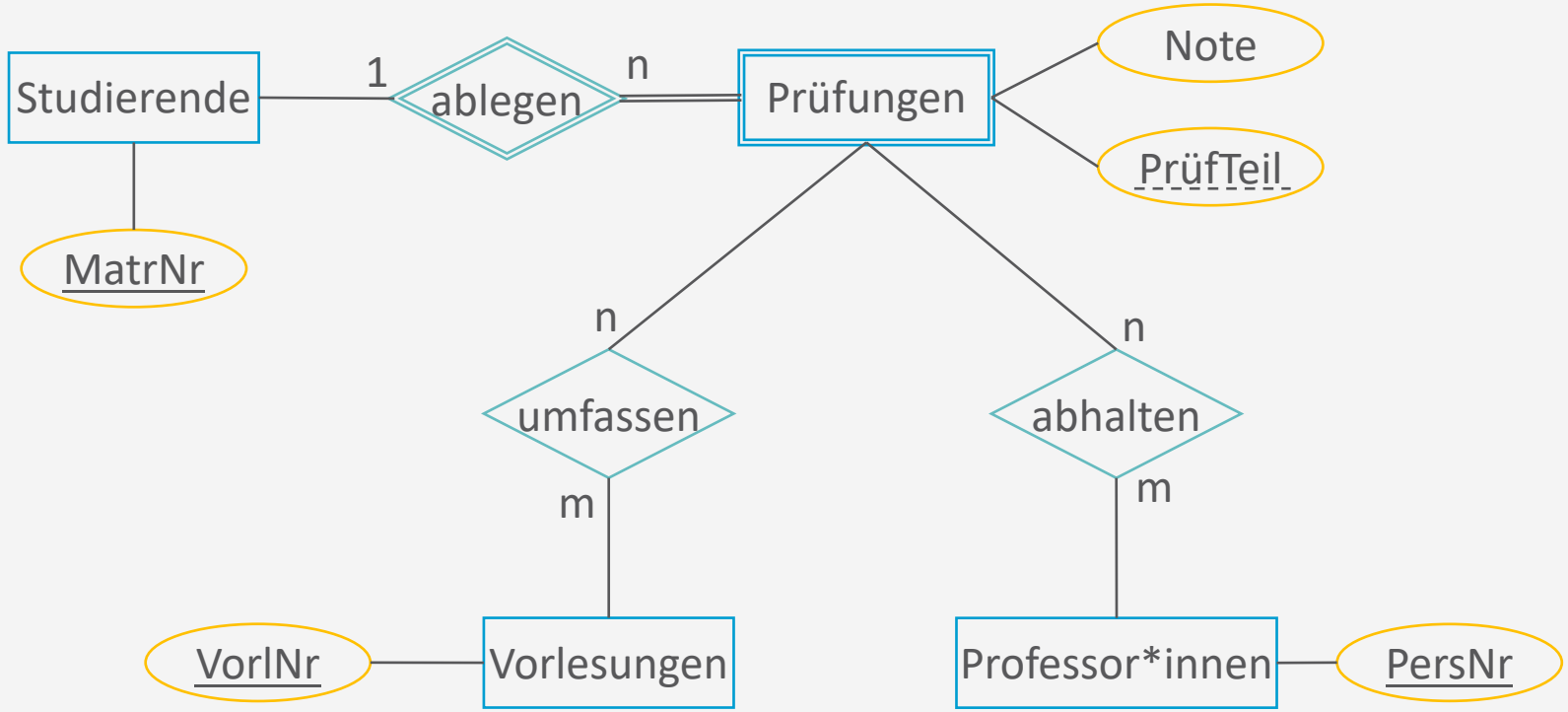


Funktionalitäten (Reprise)

- Bestimmung von Funktionalitäten / Partizipation
  - *Kann* ein Tupel von $k-1$ Entitäten mit *mehreren* Instanzen des verbliebenen Entitätstyps in Beziehung stehen? $\rightarrow n$
    - Mit genau einer Instanz? $\rightarrow 1$
  - *Muss* eine Tupel von $k-1$ Entitäten mit *mindestens* einer Instanz des verbliebenen Entitätstyps in Beziehung stehen? $\rightarrow$ total
    - Sonst $\rightarrow$ partiell



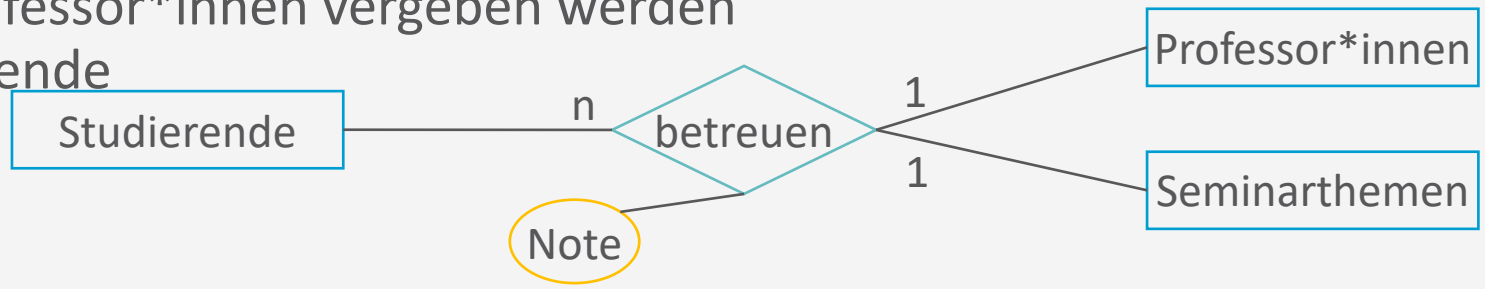
Prüfungen



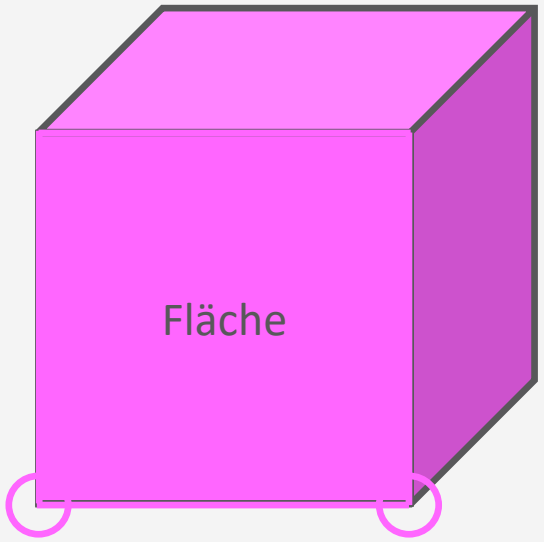
- Mehrere Prüfer in einer Prüfung
- Mehrere Vorlesungen werden in einer Prüfung abgefragt

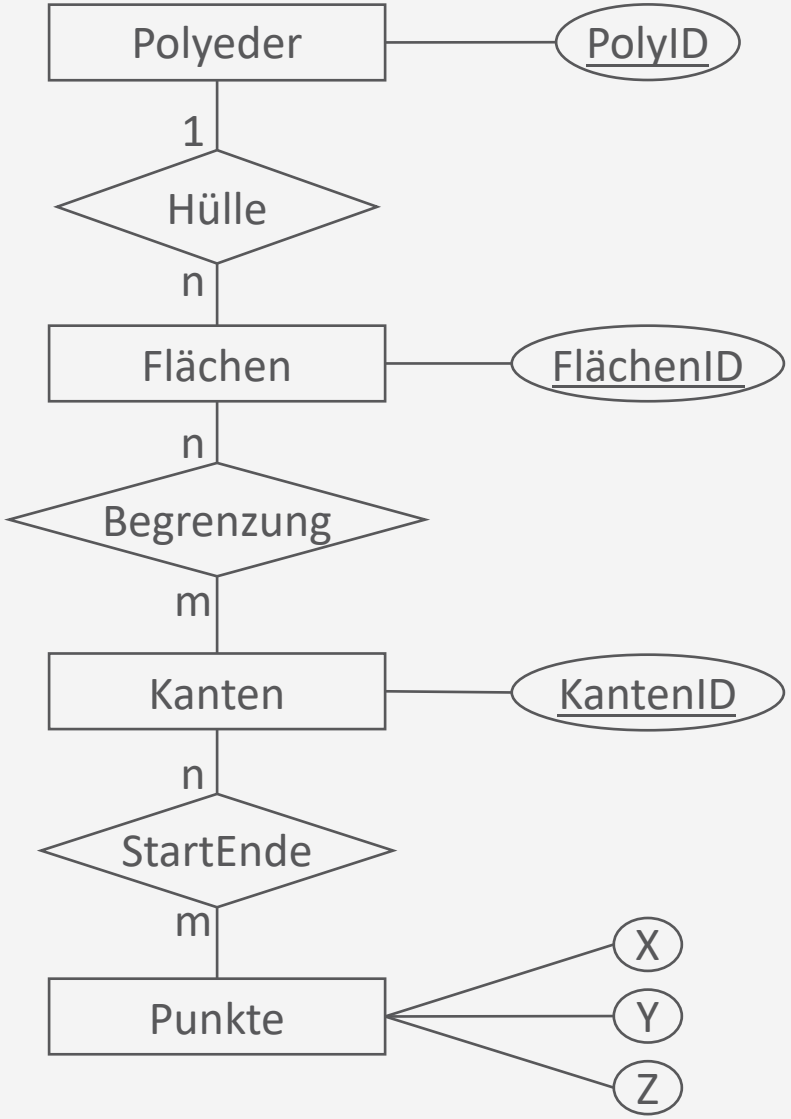
N-stellige Beziehung: *betreuen*

- Erzwungene Konsistenzbedingungen
  - Studierende dürfen bei demselben Professor bzw. derselben Professorin nur ein Seminarthema „ableisten“ (damit ein breites Spektrum abgedeckt wird)
  - Studierende dürfen dasselbe Seminarthema nur einmal bearbeiten – sie dürfen also nicht bei anderen Professor*innen ein schon einmal erteiltes Seminarthema nochmals bearbeiten
- Mögliche Zustände
  - Professor*innen können dasselbe Seminarthema „wiederverwenden“ – also dasselbe Thema auch mehreren Studierende erteilen
  - Ein Thema kann von mehreren Professor*innen vergeben werden – aber an unterschiedliche Studierende

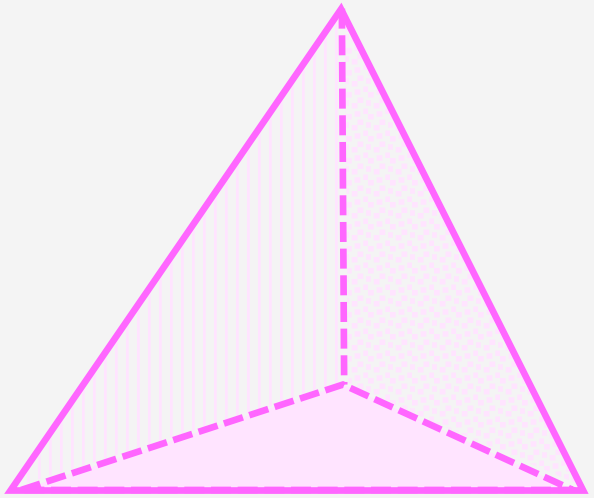


Komplex-strukturierte Entitäten

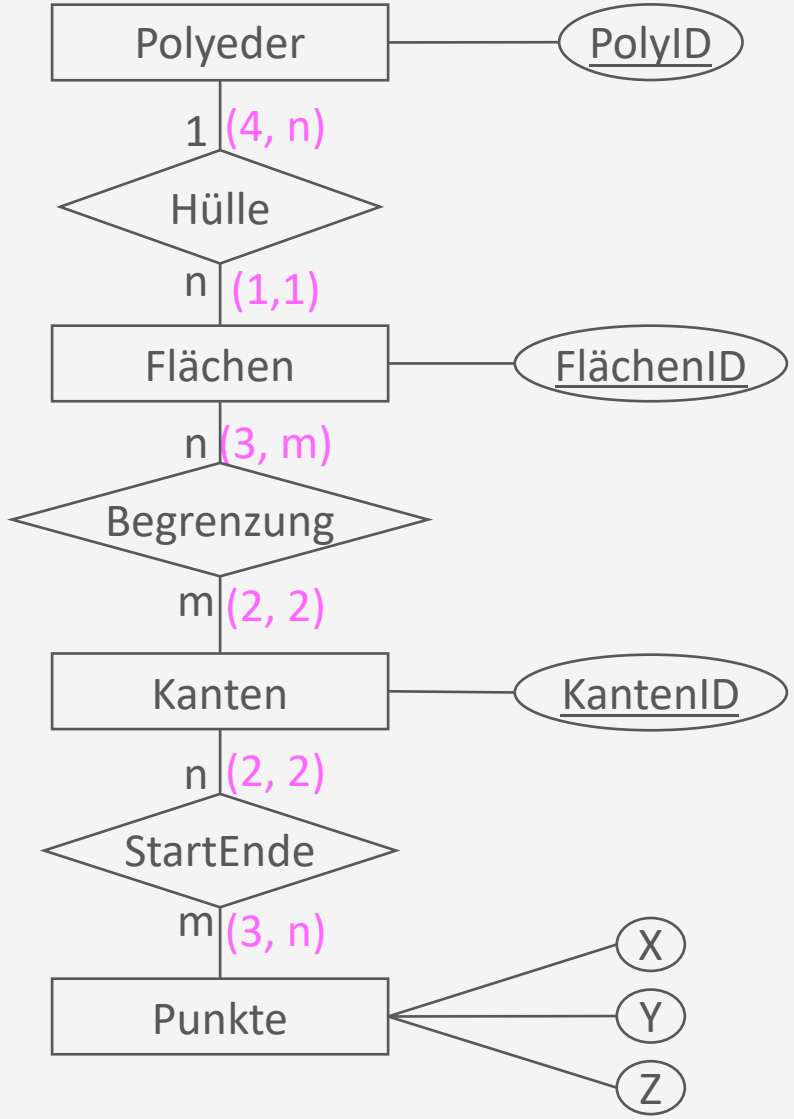




Komplex-strukturierte Entitäten

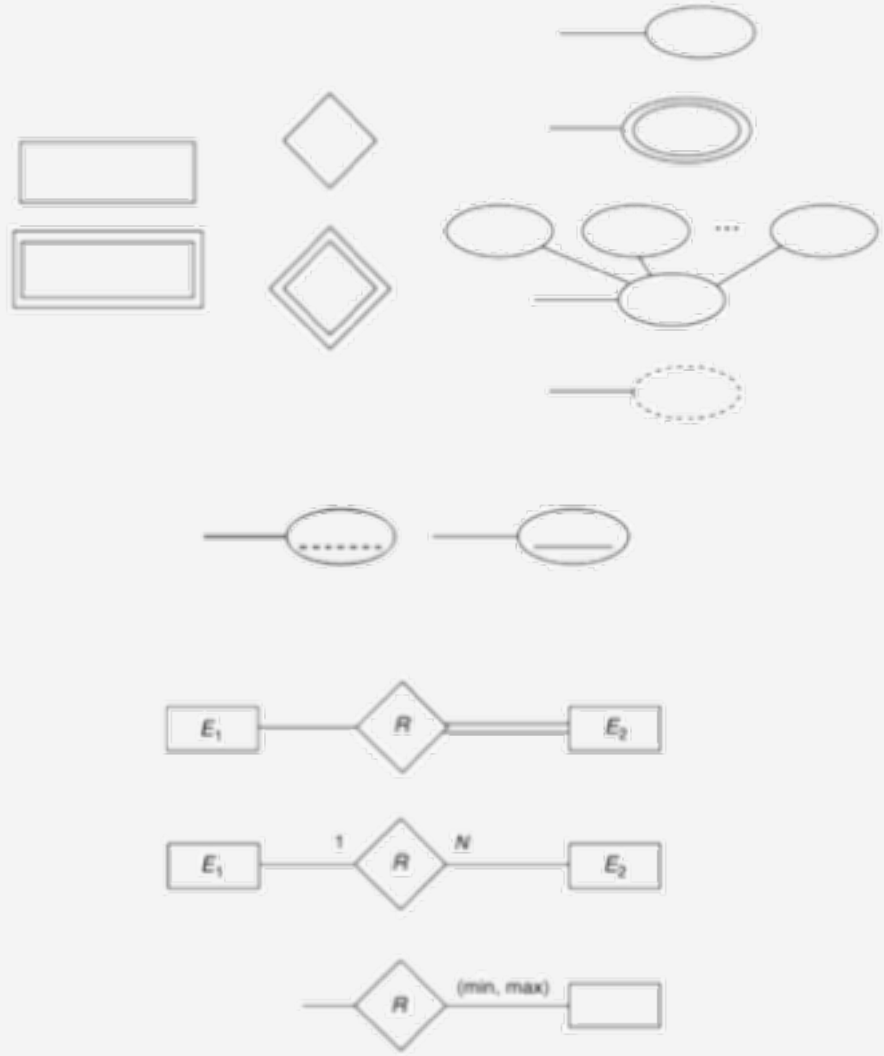


Kleinster Polyeder



Zwischenzusammenfassung

- Entitäten
  - Starke / schwache Entitäten
- Assoziationen / Beziehungen
- Attribute
  - Mehrwertig / zusammengesetzt / abgeleitet
- Schlüssel
  - Referentielle Integrität
  - Partielle Schlüssel
- Funktionalitäten / Kardinalitäten
- Partizipation: total, partiell



Überblick: 2. Datenbank-Modellierung

A. Motivation
- Modelle und Modellierung

B. Entity-Relationship-Modell (ER)
- Komponenten
- Von Anforderungen zum ER-Modell
- Interpretation von ER-Modellen

C. Enhanced ER-Modell (EER)
- Spezialisierung, Generalisierung
- Modellierungsvorgehen
- Beziehung zu UML

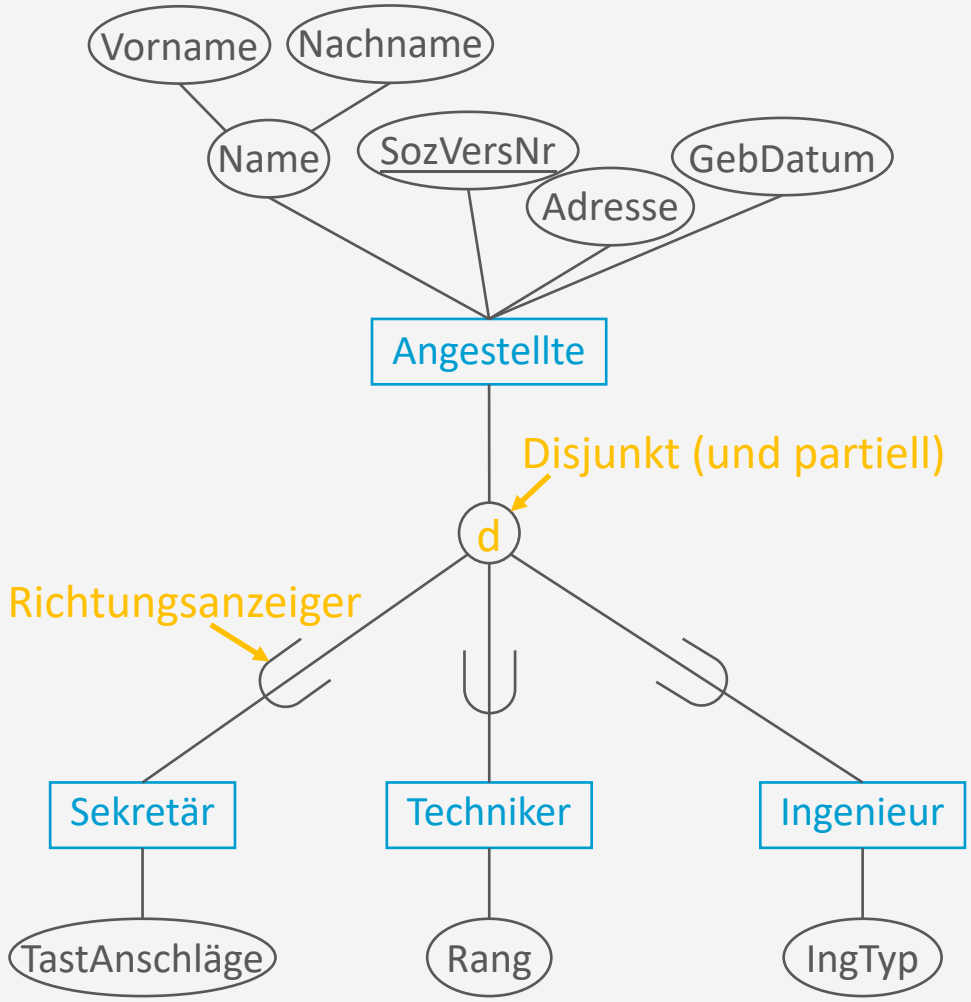
D. Dokumentation & Bewertung
- Dokumentation
- Qualitätskriterien EER

Enhanced ER-Modell (EER-Modell)

- Erweitert das ER-Modell um Spezialisierung und Generalisierung
  - Spezialisierung: "top-down"
    - Definition einer Menge von Subentitäten einer (Super-) Entität
  - Generalisierung: "bottom-up"
    - Bildung einer generalisierenden (Super-) Entität
  - Entitäten, Superentitäten und Subentitäten werden häufig auch als Klassen, Superklassen und Subklassen bezeichnet
    - Bilden eine Klassenhierarchie/Typhierarchie
    - Objekte einer Entität sind auch Objekte der Superklasse
    - Subklassen erben Eigenschaften der Superklasse
- Ausprägungen: Disjunkt oder überlappend (overlapping) sowie partiell oder total
- Definition eines Aufzählungstypen: definierendes Attribut

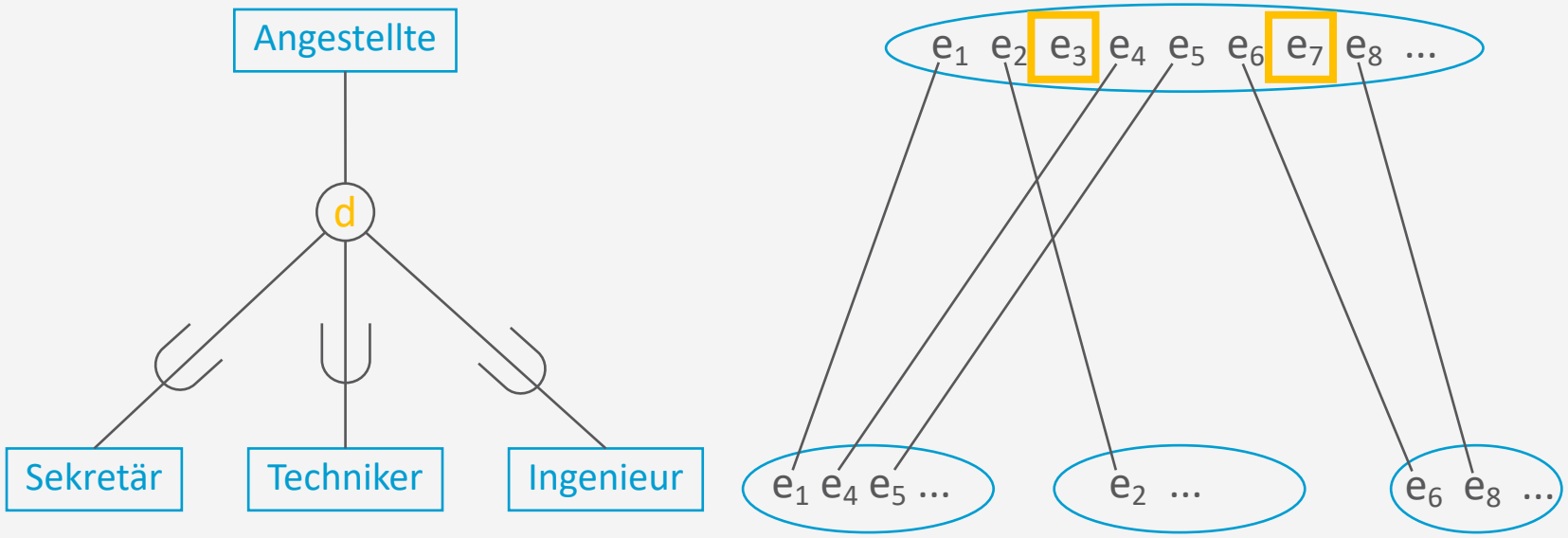
Spezialisierung

- Ermöglicht
  - Definition einer Menge von Subentitäten zu einer Superentität
  - Erzeugung zusätzlicher spezifischer Attribute für Subentitäten
  - Erzeugung zusätzlicher spezifischer Beziehungen zwischen jeder Subentität und anderen Entitäten oder Subentitäten
- Beispiel:
  - Disjunkt
  - Partiell



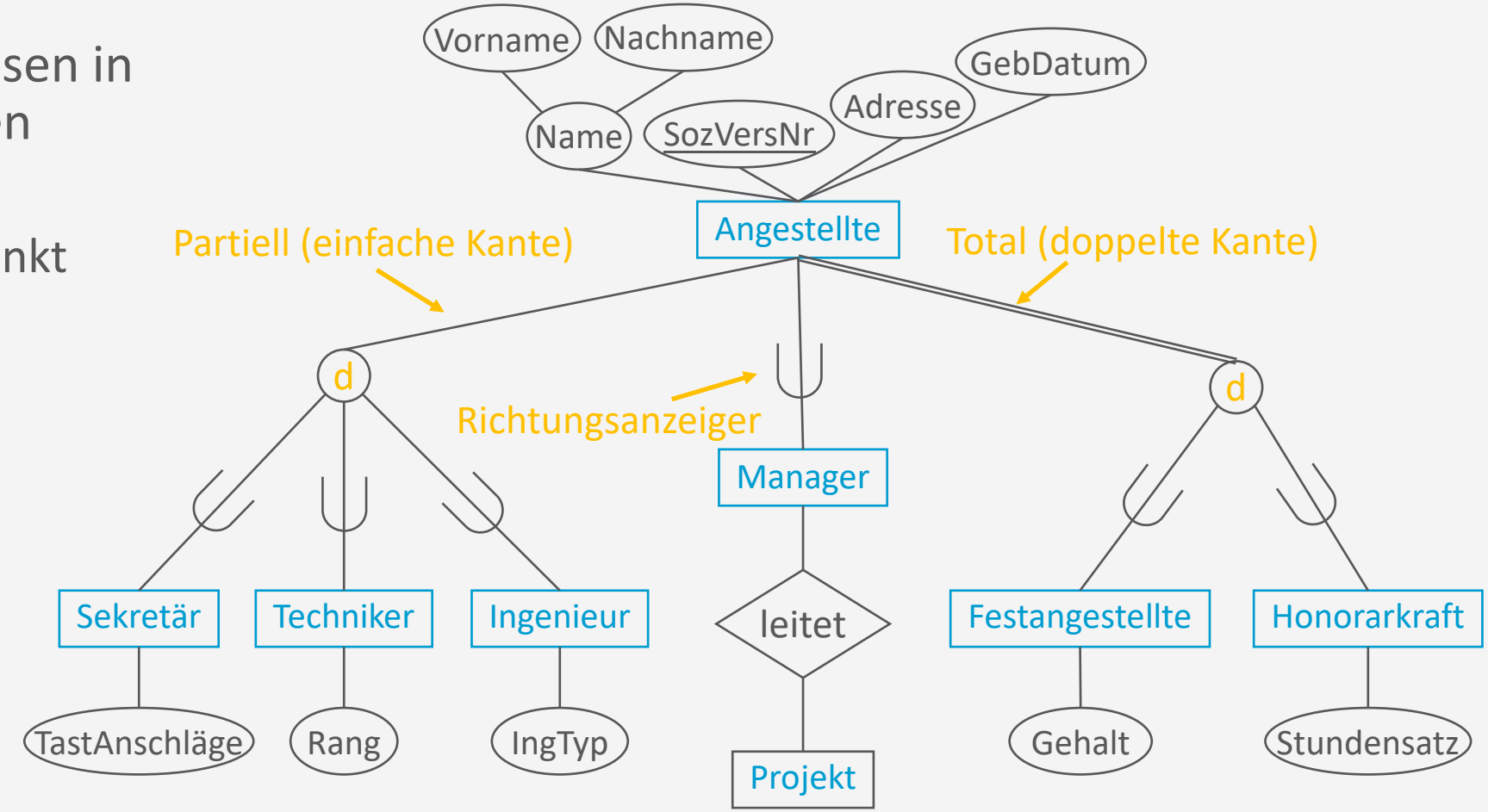
Instanzen in Spezialisierungen

- Disjunkt: Keine Instanz fällt in mehrere Spezial-Klassen
- Partiell: Es gibt Instanzen, die in keine Spezialklasse fallen (z.B. e_3, e_7)



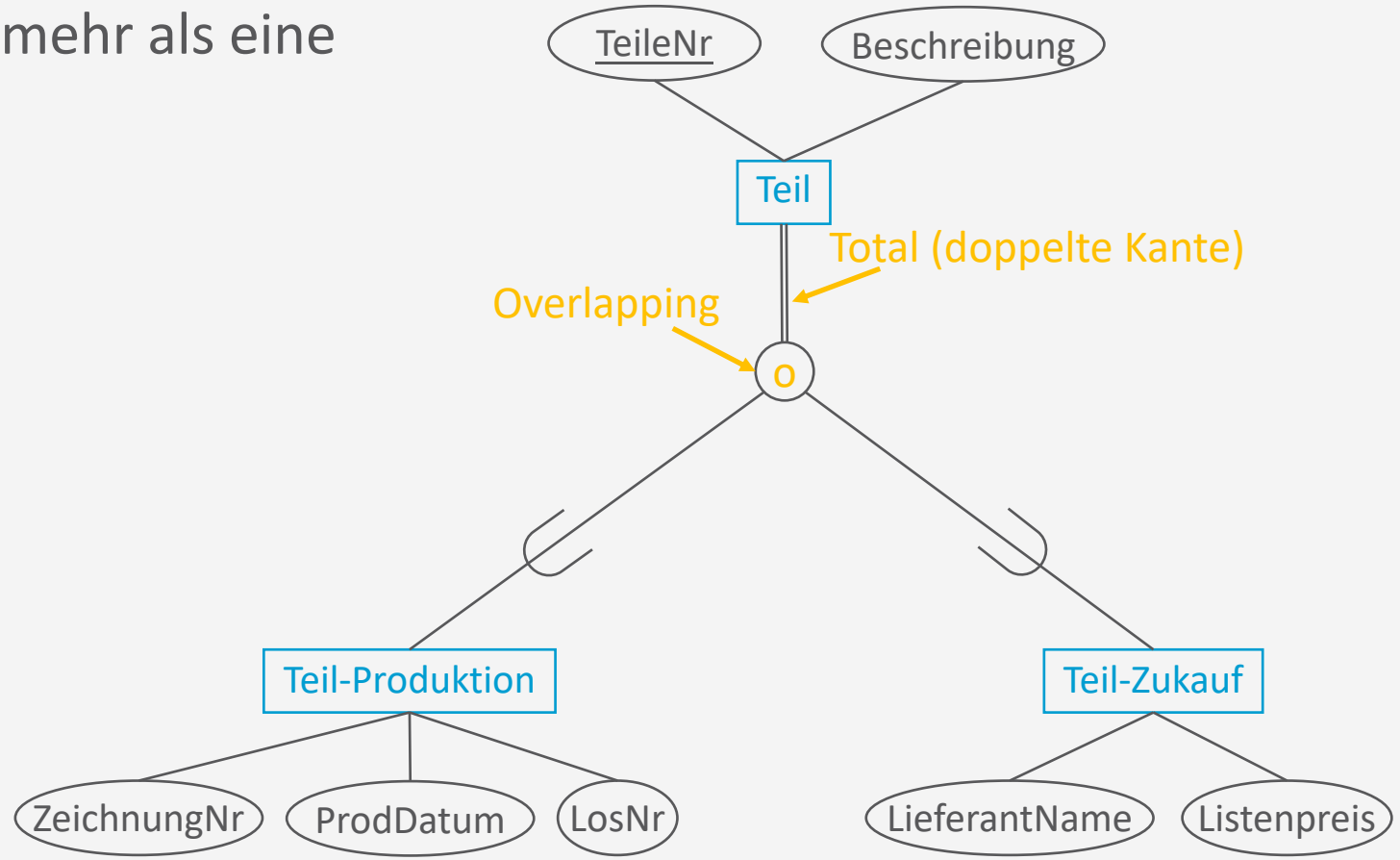
Ausprägungen der Spezialisierung

- Total: alle Instanzen müssen in eine Spezialisierung fallen
  - Gegenstück zu partiell
  - Genau eine: total + disjunkt



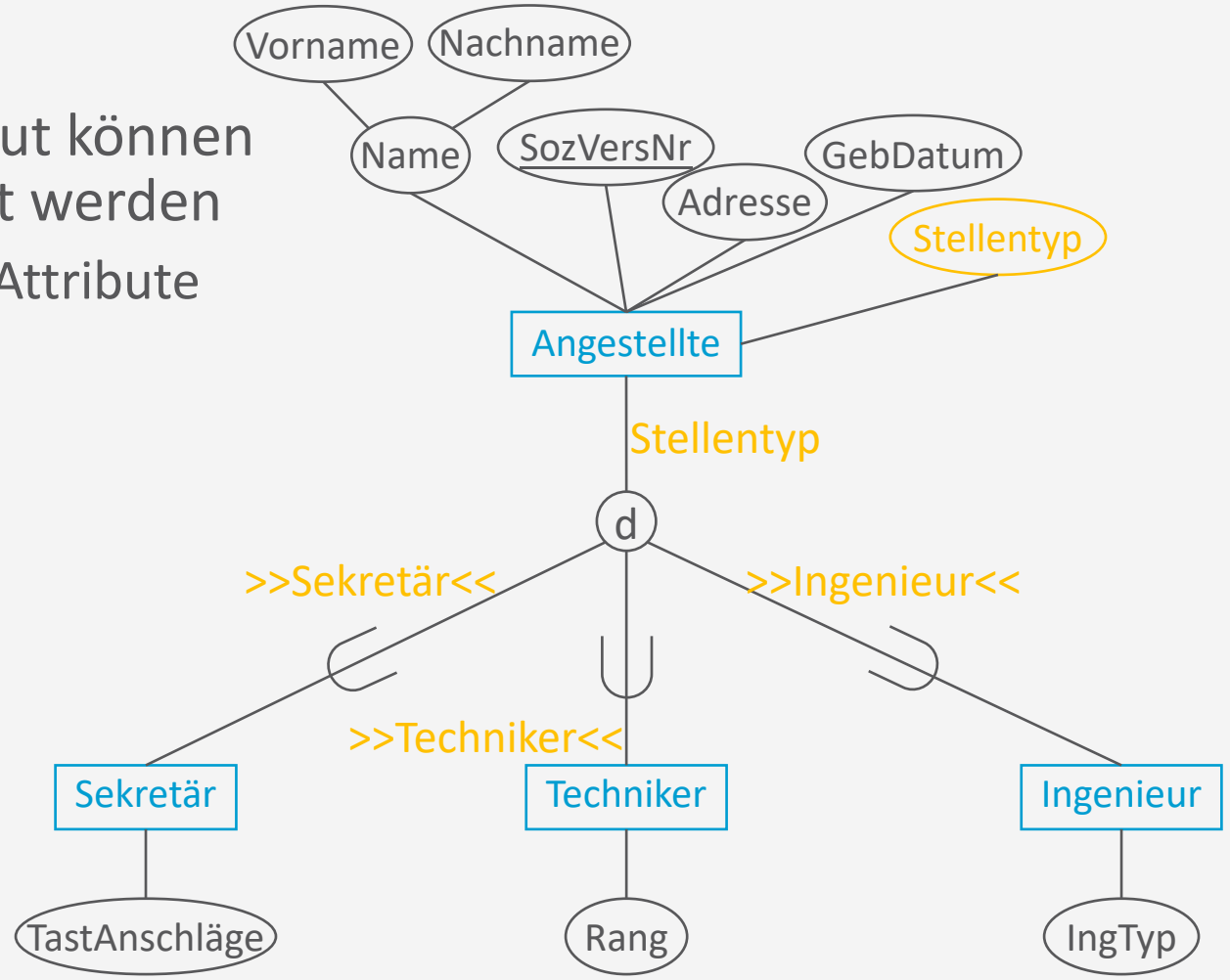
Ausprägungen der Spezialisierung

- Überlappend: Instanzen können in mehr als eine Spezialisierung fallen

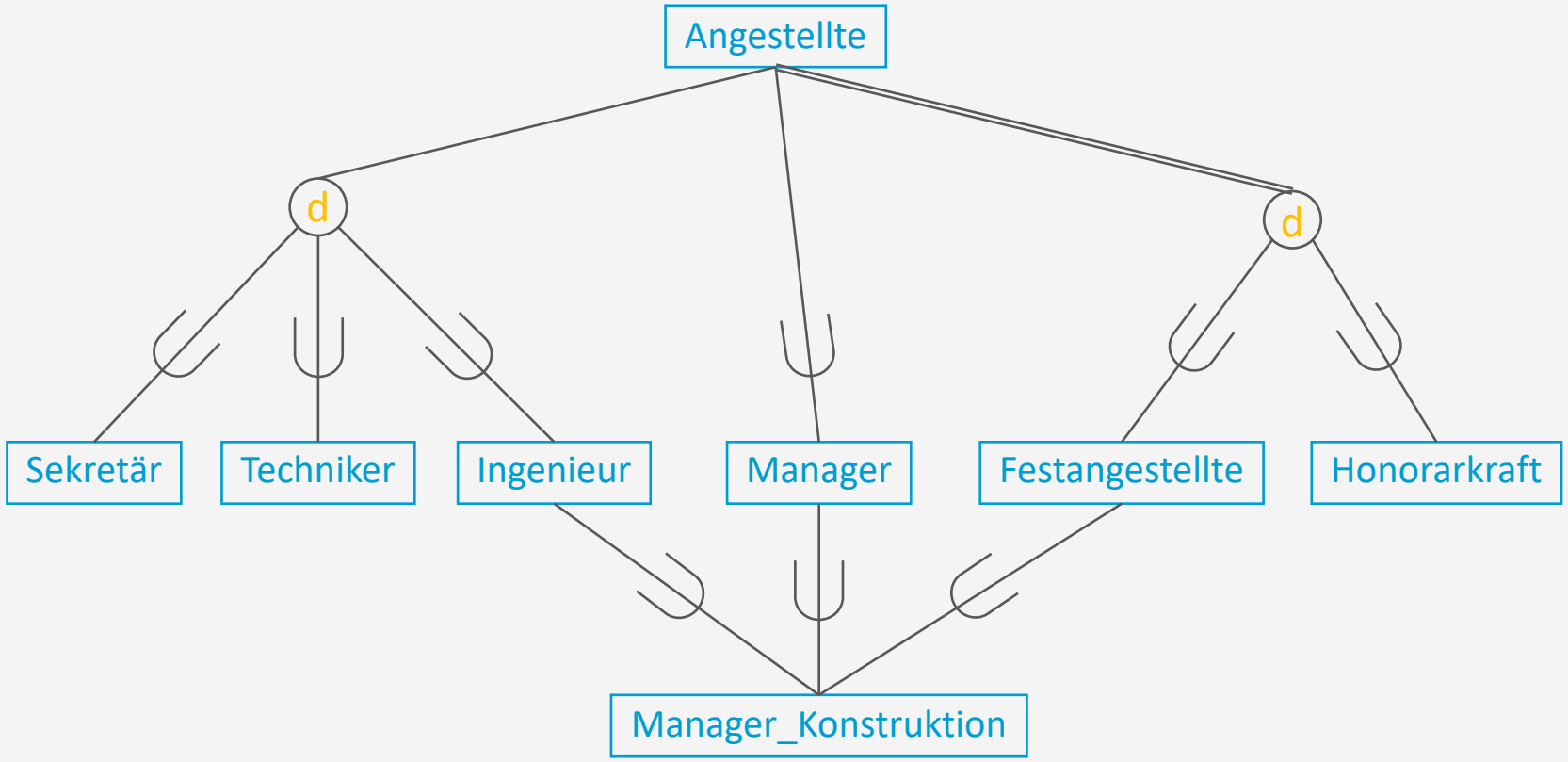


Aufzählungstypen

- Zu einem bestimmten definierendes Attribut können eine Menge von Spezialisierungen definiert werden
  - Erlaubt Attributwert-spezifische zusätzliche Attribute zu definieren
  - Beispiel: Stellentyp mit Spezialisierungen
    - Sekretärin
      - Attribut: TastAnschläge
    - Techniker
      - Attribut: Rang
    - Ingenieur
      - Attribut: IngTyp

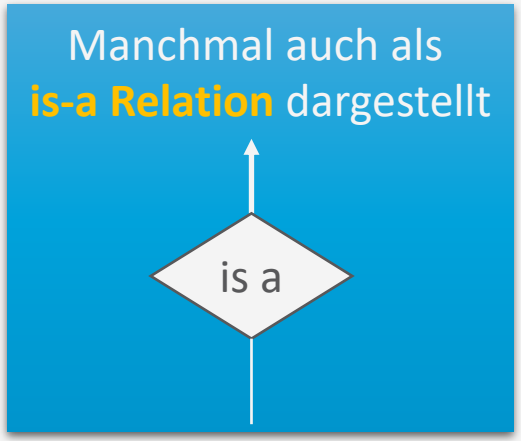


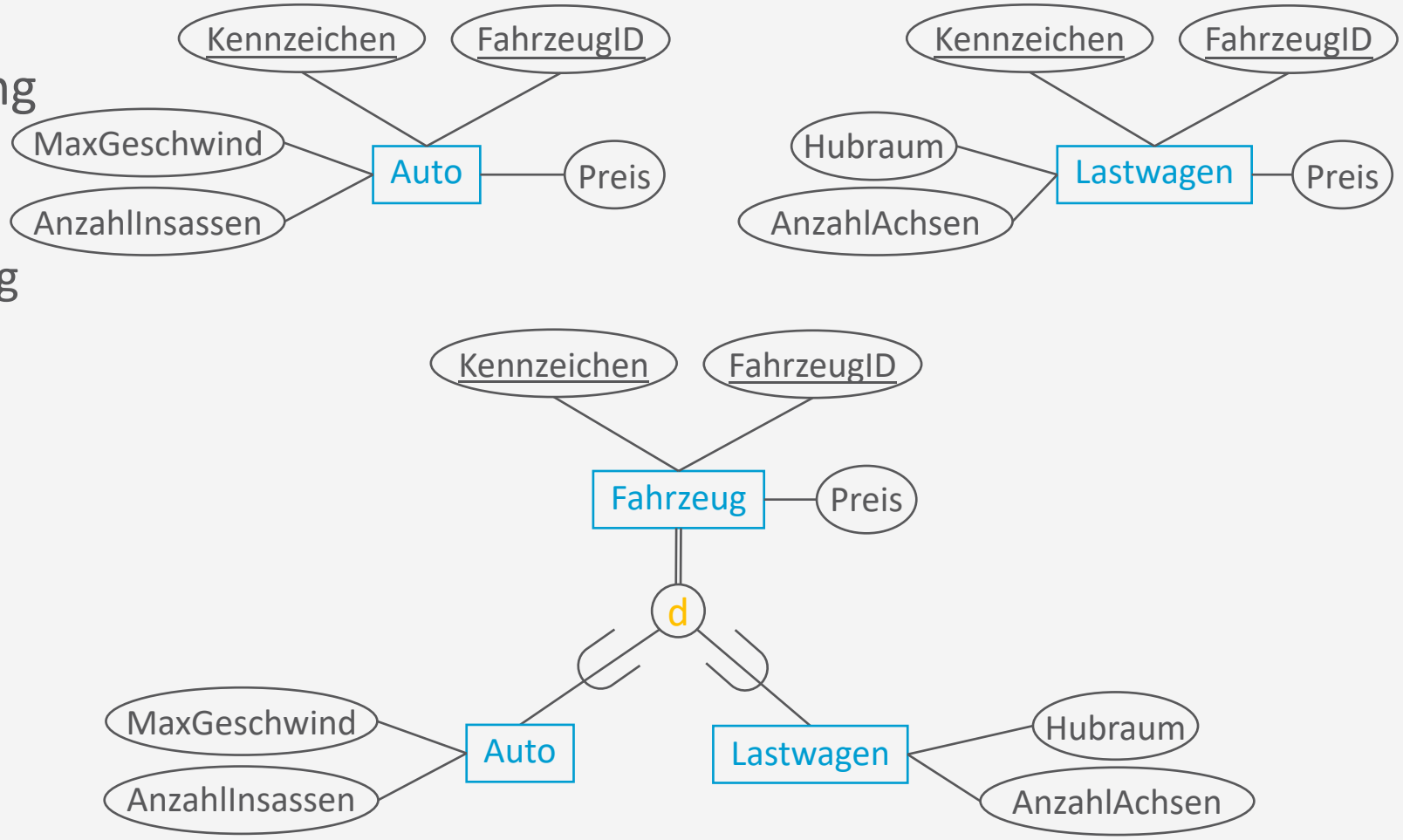
Spezialisierungsnetzwerk



Generalisierung

- Umkehrung der Spezialisierung
  - Zusammenfassung von spezielleren Entitäten
  - Ausprägungen und Darstellung gleich zu Spezialisierung





EER-Modellierung

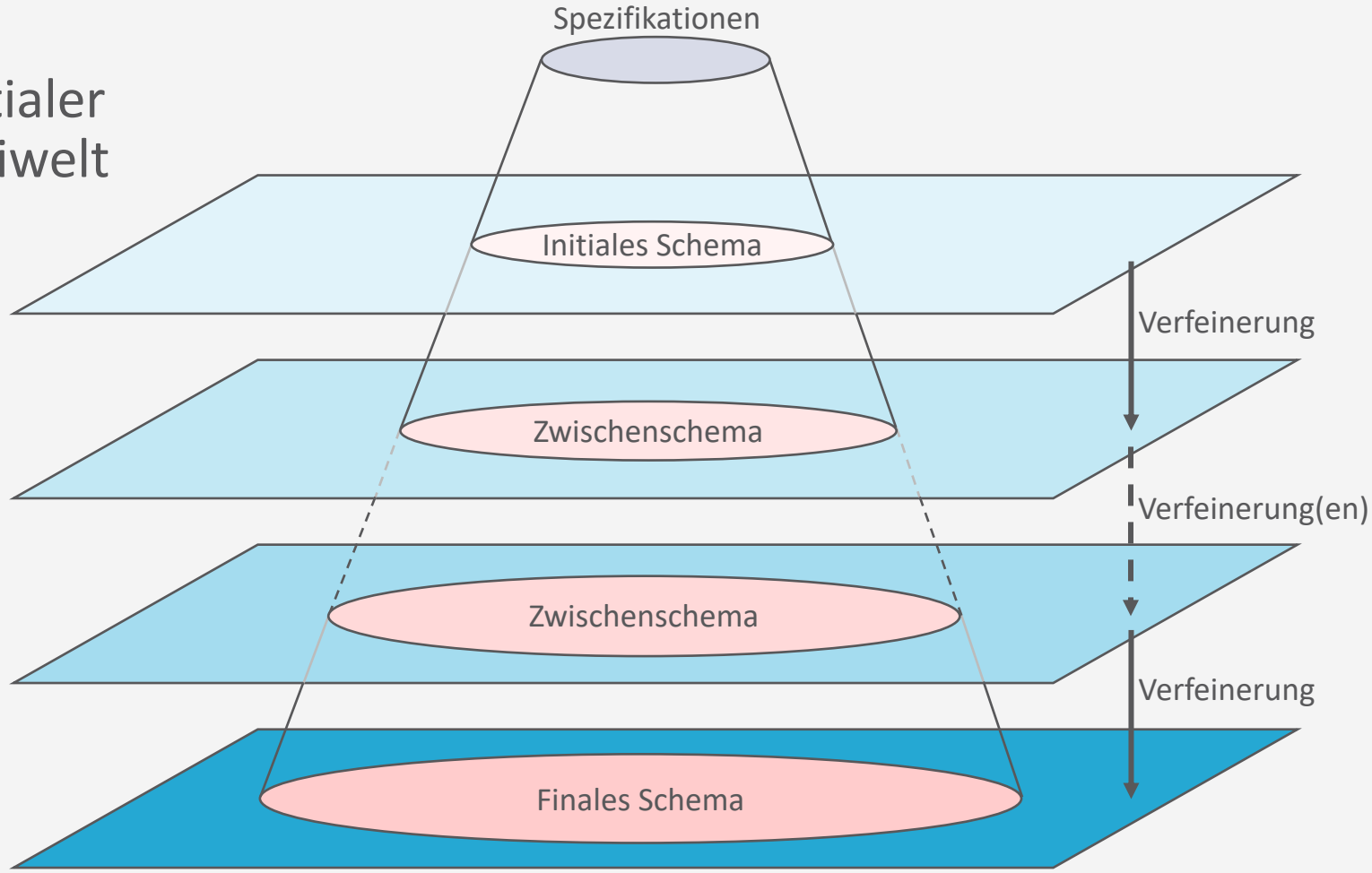
Vorgehen

Vorgehen

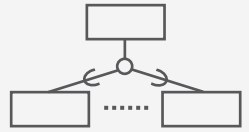
- Top-down
  - Sukzessiver Verfeinerungen initialer (abstrakter) Konzepte der Miniwelt
- Bottom-up
  - Zerlegung der Miniwelt-Darstellung in kleinste Konzepte
  - Abschließend Integration in ein Gesamtschema
- Inside-out
  - wie bottom-up, aber ausgehend von einem zentralen Konzept
    - Vom „Wichtigen“ zum „Un-wichtigen“
- Mixed
  - Mischform des Top-down- und des Bottom-up-Ansatzes
- Für ein gegebenes ER-Modell: verschiedene Transformationen

Top-down

- Sukzessive Verfeinerungen initialer (abstrakter) Konzepte der Miniwelt

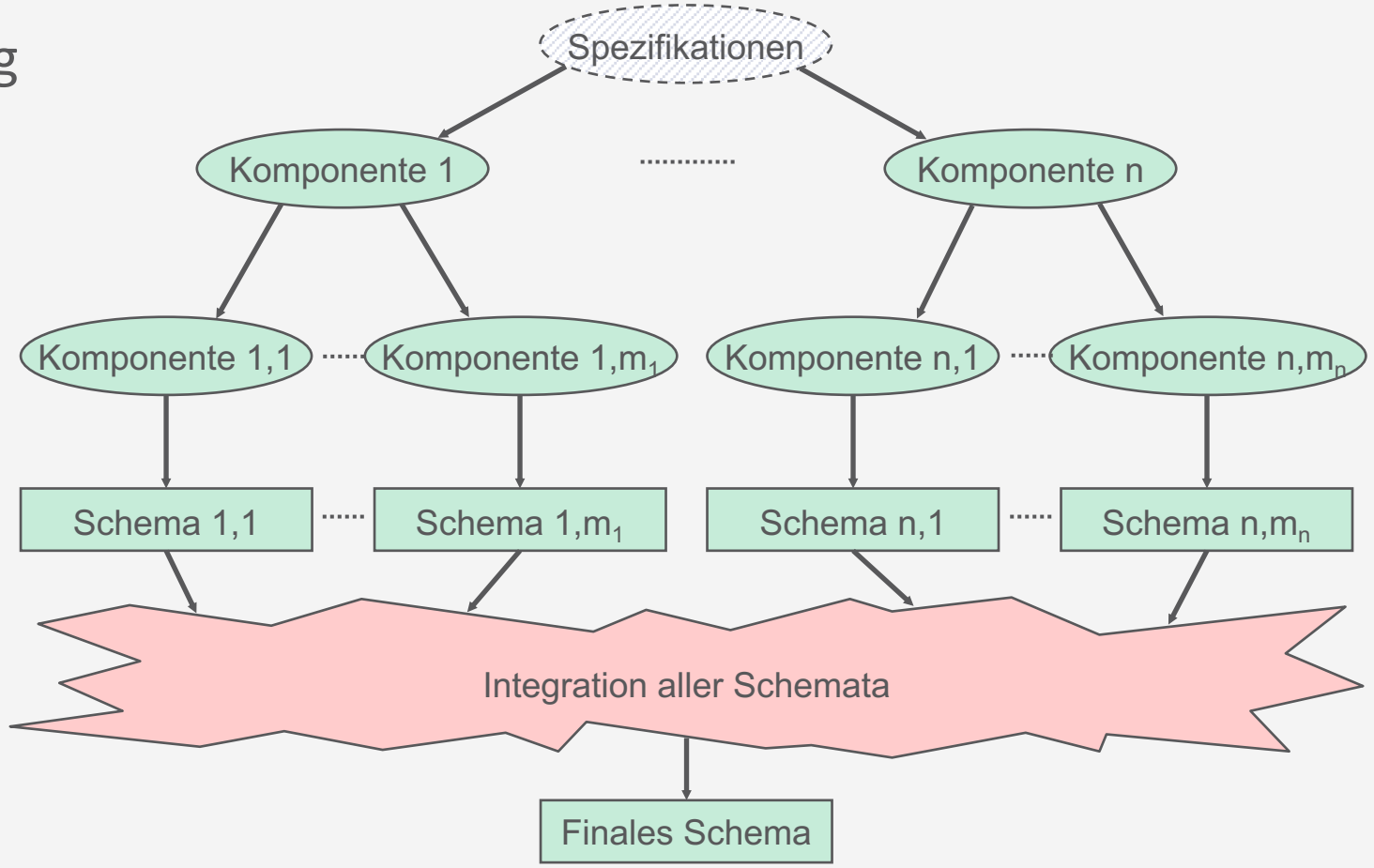


Top-down EER-Transformationen

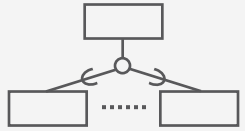
Transformation	Initiales Konzept	Ergebnis
T-TD$_1$:Von einer Entität zu zwei Entitäten mit einer Beziehung		
T-TD$_2$: Von einer Entität zu Entität und Spezialisierungen		
T-TD$_3$: Von einer Beziehung zu mehreren Beziehungen		
T-TD$_4$: Von einer Beziehung zu einer Entität mit Beziehungen		
T-TD$_5$: Hinzufügen von Attributen zu einer Entität		
T-TD$_6$: Hinzufügen von Attributen zu einer Beziehung		

Bottom-up

- Zerlegung der Miniwelt-Darstellung in kleinste Konzepte; abschließend Integration in ein Gesamtschema

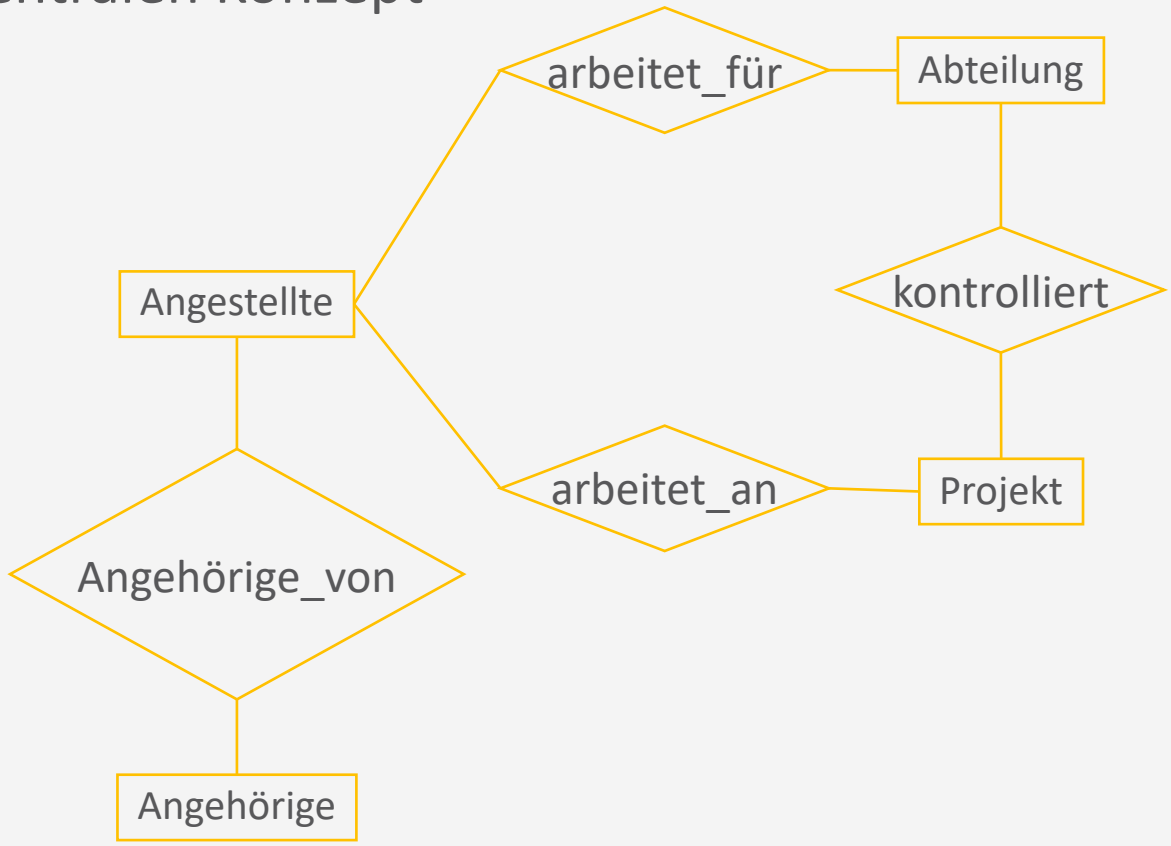


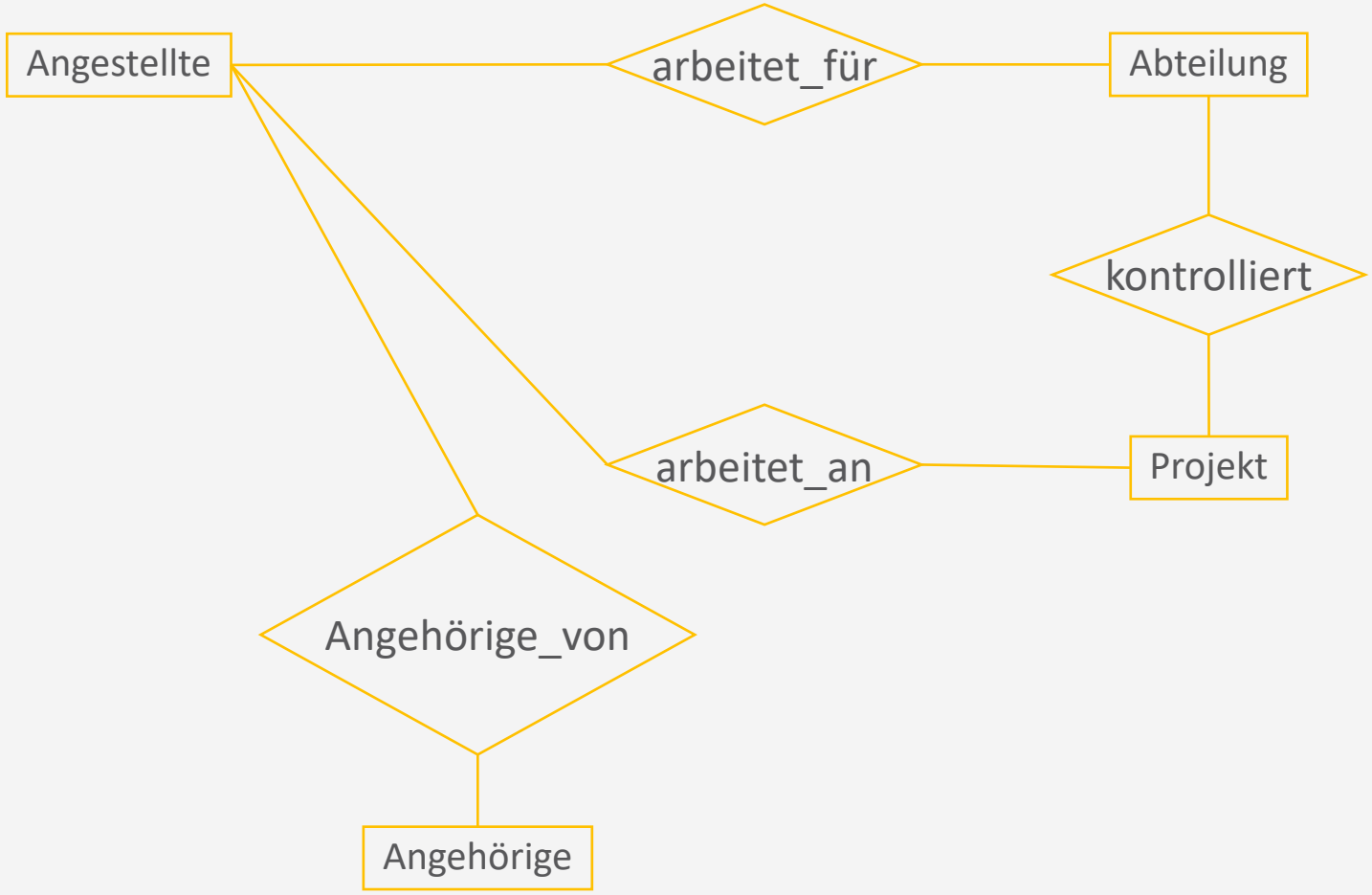
Bottom-up EER-Transformationen

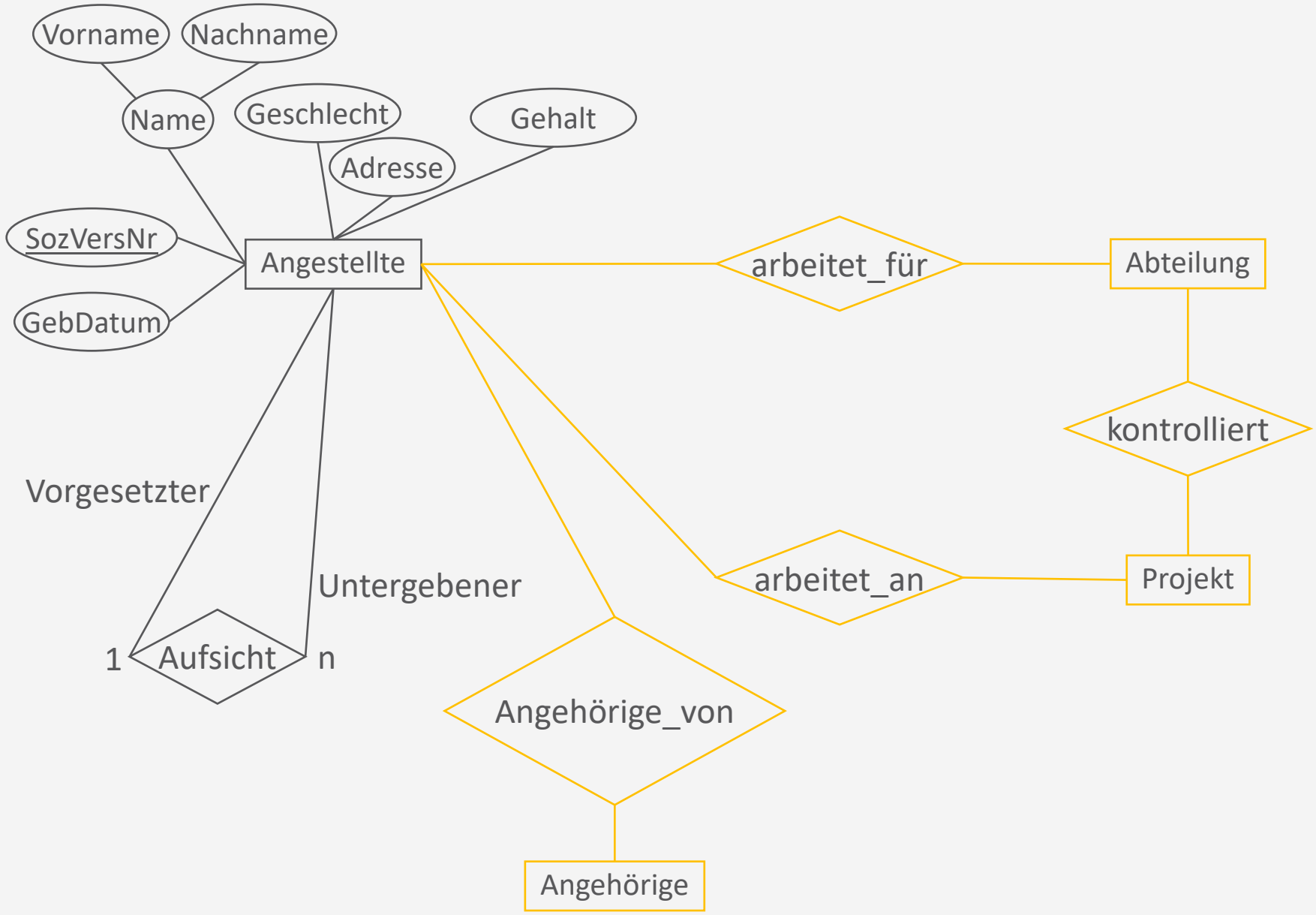
Transformation	Initiales Konzept	Ergebnis
T-BU$_1$: Erzeugung einer Entität		
T-BU$_2$: Erzeugung einer Beziehung		
T-BU$_3$: Erzeugung einer Generalisierung		
T-BU$_4$: Zusammenfassung und Zuordnung von Attributen zu einer Entität		
T-BU$_5$: Zusammenfassung und Zuordnung von Attributen zu einer Beziehung		

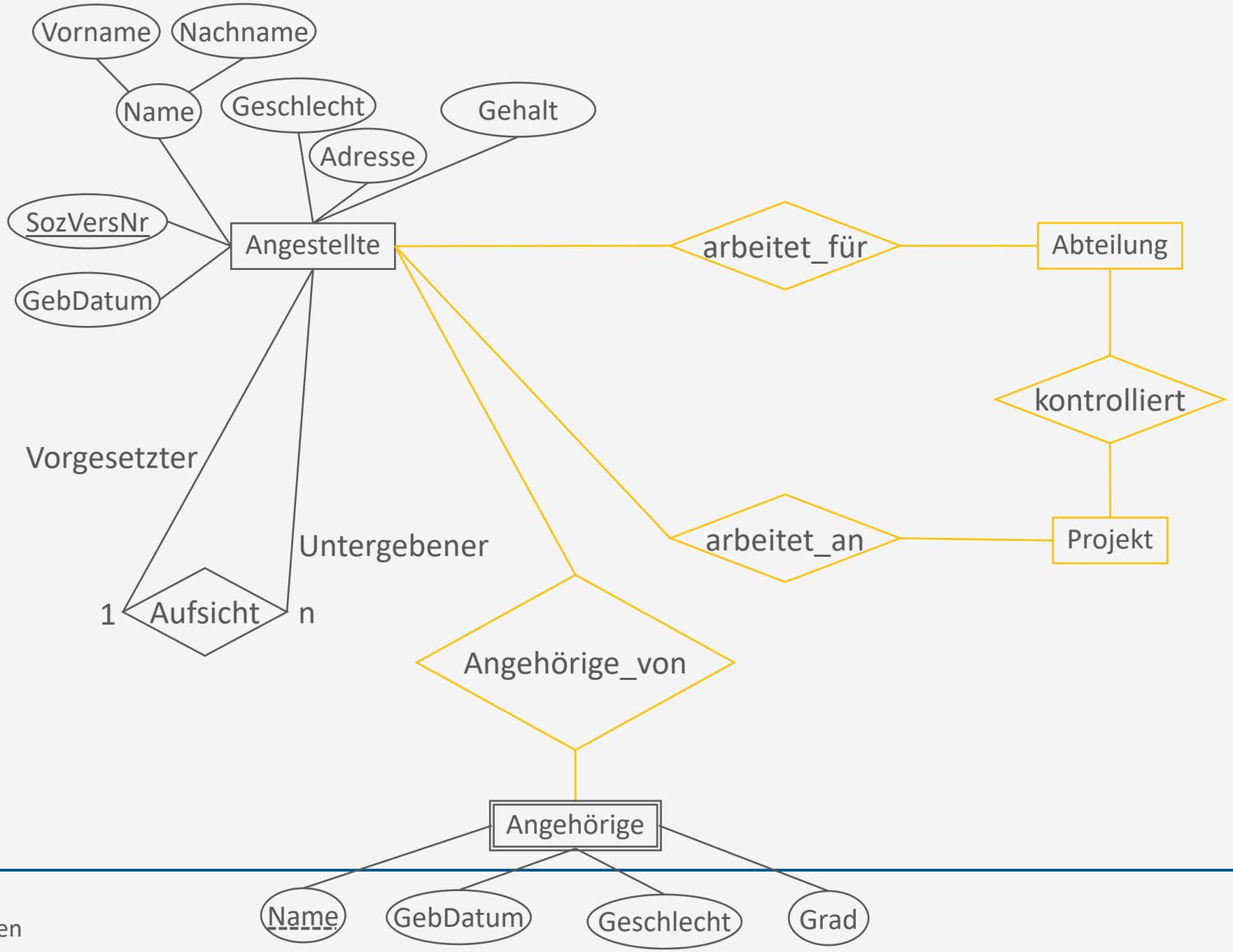
Inside-out

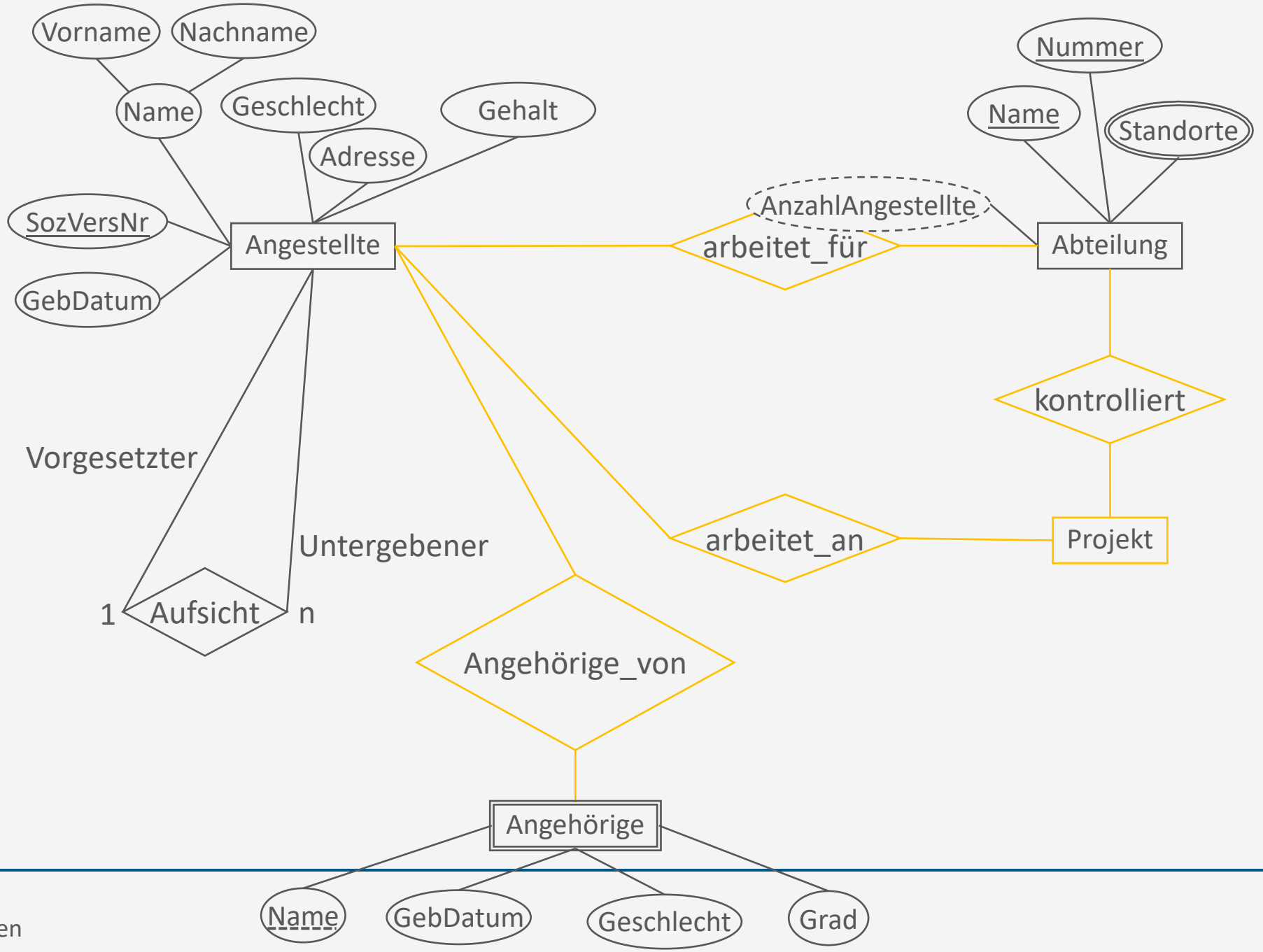
- Wie bottom-up, aber ausgehend von einem zentralen Konzept
  - Vom „Wichtigen“ zum „Un-wichtigen“
  - Beispiel
    - Entitäten
      - Angestellte
      - Abteilung
      - Projekt
      - Angehörige
    - Beziehungen
      - Angestellte ist Abteilung zugeordnet
      - Angestellte arbeitet für Projekt
      - Abteilung kontrolliert Projekt
      - Angestellte haben Angehörige

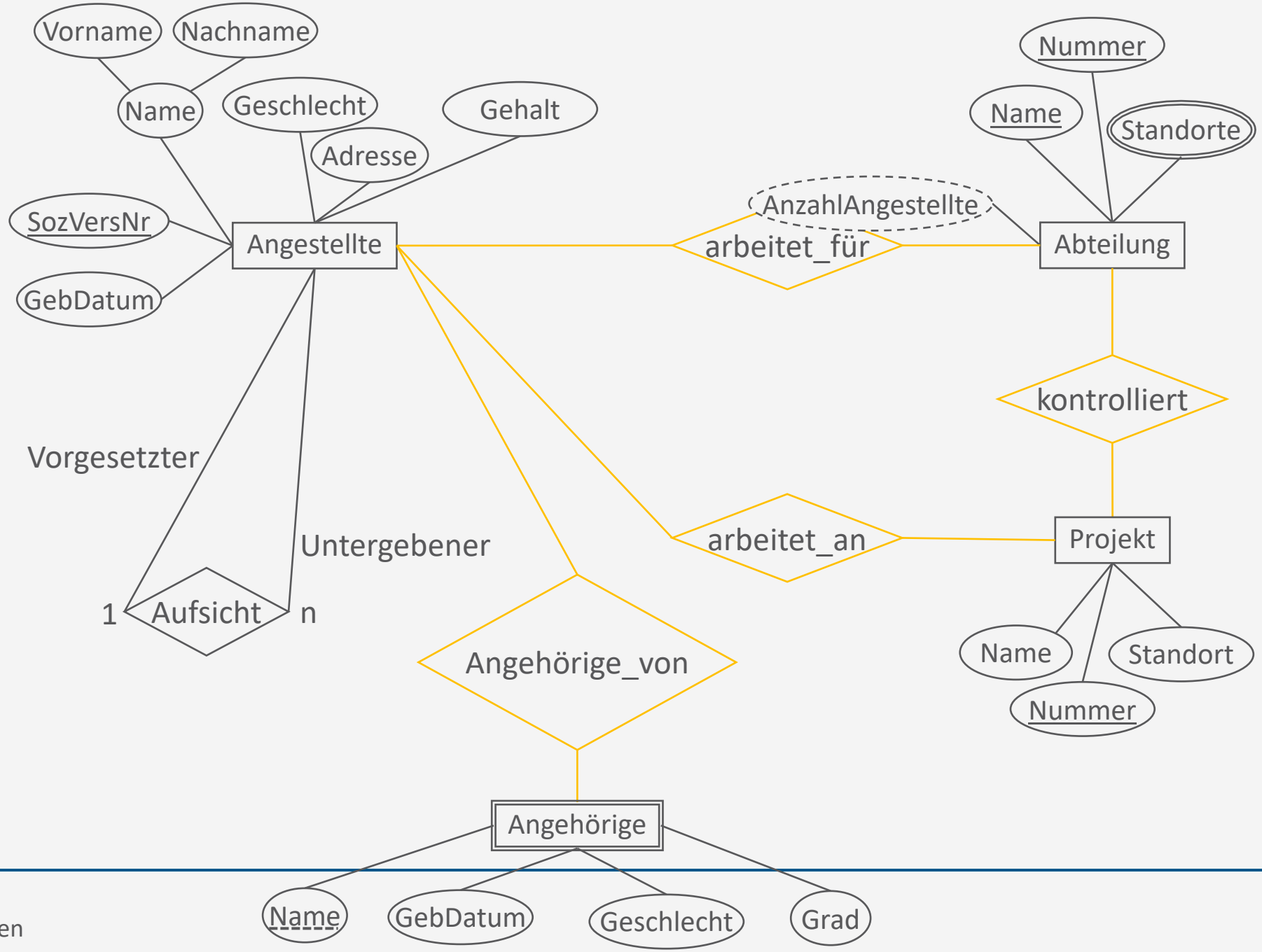


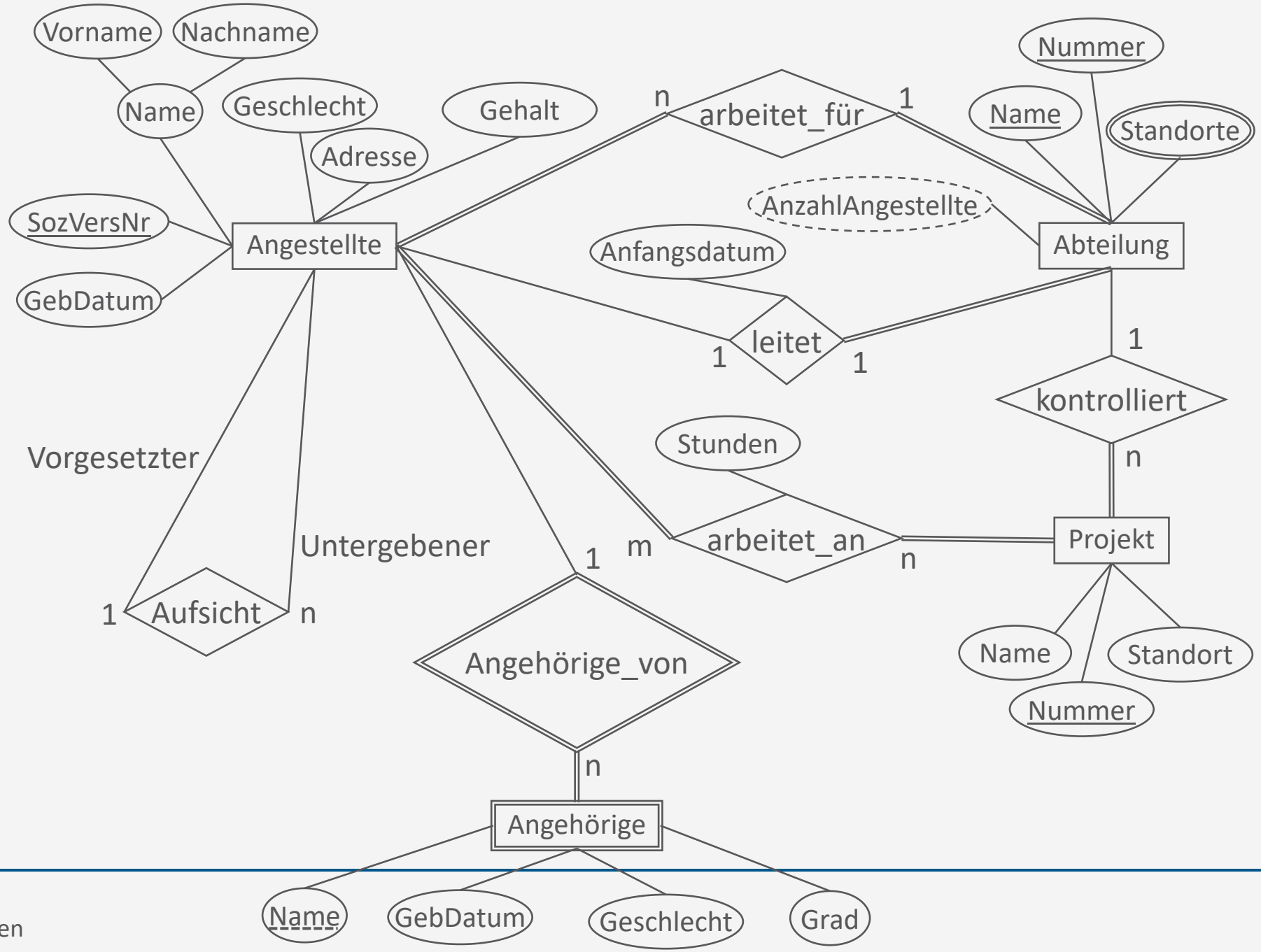

Angestellte
arbeitet_für
Abteilung
kontrolliert
arbeitet_an
Projekt
Angehörige_von
Angehörige


Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
Angestellte
GebDatum
arbeitet_für
Abteilung
kontrolliert
Vorgesetzter
Untergebener
arbeitet_an
Projekt
1
Aufsicht
n
Angehörige_von
Angehörige


Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
Angestellte
GebDatum
arbeitet_für
Abteilung
kontrolliert
Vorgesetzter
Untergebener
arbeitet_an
Projekt
1
Aufsicht
n
Angehörige_von
Angehörige
Name
GebDatum
Geschlecht
Grad


Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
GebDatum
Angestellte
Nummer
Name
Standorte
AnzahlAngestellte
arbeitet_für
Abteilung
kontrolliert
Vorgesetzter
Untergebener
1
Aufsicht
n
arbeitet_an
Projekt
Angehörige_von
Angehörige
Name
GebDatum
Geschlecht
Grad

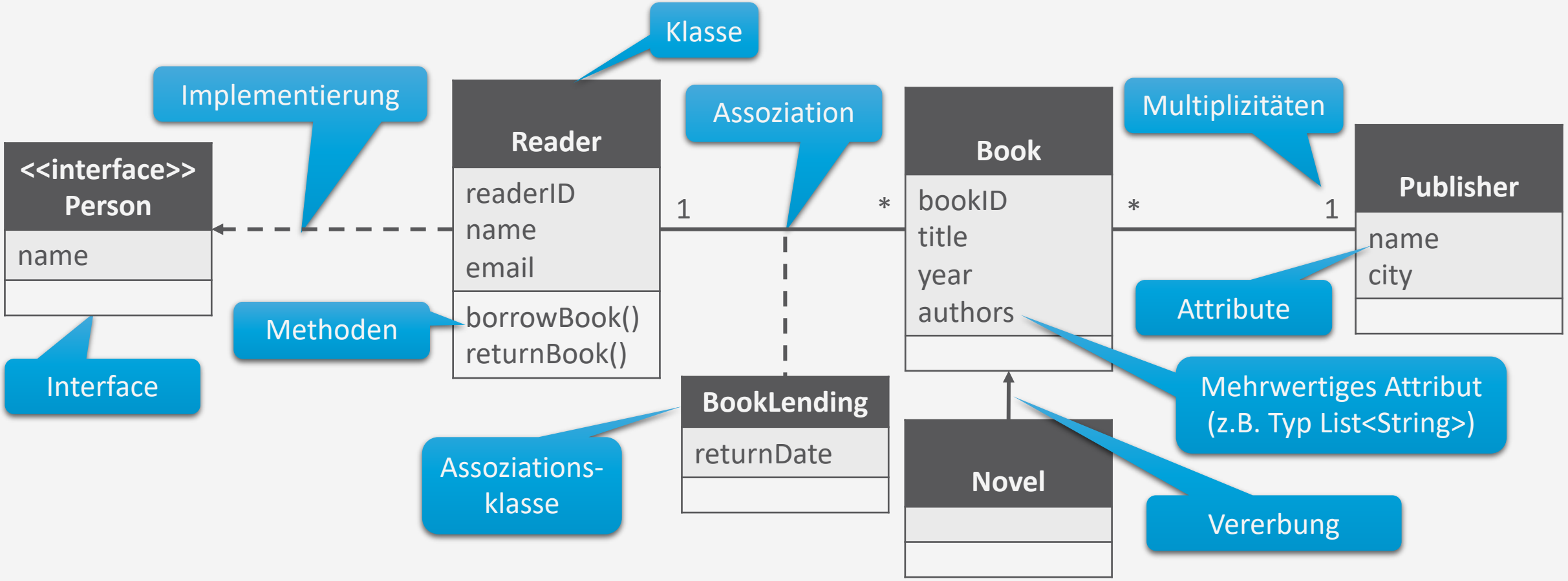

Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
GebDatum
Angestellte
Vorgesetzter
Untergebener
1
Aufsicht
n
Nummer
Name
Standorte
AnzahlAngestellte
arbeitet_für
Abteilung
kontrolliert
arbeitet_an
Projekt
Name
Standort
Nummer
Angehörige_von
Angehörige
Name
GebDatum
Geschlecht
Grad


Vorname
Nachname
Name
Geschlecht
Adresse
Gehalt
SozVersNr
GebDatum
Angestellte
n
arbeitet_für
1
Nummer
Name
Standorte
AnzahlAngestellte
Abteilung
Anfangsdatum
leitet
1
1
1
kontrolliert
n
Stunden
Vorgesetzter
Untergebener
1
Aufsicht
n
m
arbeitet_an
n
Projekt
1
Angehörige_von
n
Name
Standort
Nummer
Angehörige
Name
GebDatum
Geschlecht
Grad

UML als Datenmodell

Für UML-Bekannte

Unified Modeling Language (UML)



Lena Wiese: Advanced Data Management, DeGruyter, 2015.

(E)ER vs. UML

(E)ER	UML
Entität	Klasse
Relation	Assoziation
Kardinalität: 1:1, 1:n, n:m oder min/max	Multiplizität: 0, 1, eine Zahl, Intervalle, *
Attribute	Attribute
Mehrwertige Attribute	Attribute des Typs List<type>
Schlüsselattribute	-- (als normales Attribut)
--	Methoden (im Allgemeinen)
Abgeleitete Attribute	Methoden (die ein berechnetes Attribut zurückgeben)
Generalisierung	Vererbung
--	Interface, Implementierung

gleiches Konzept, gleicher Name

gleiches Konzept, anderer Name

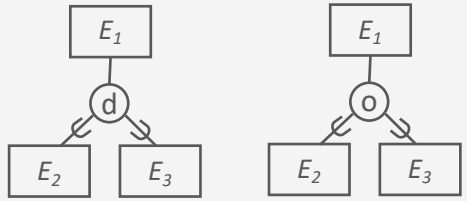
Konzept fehlt in ER

Konzept fehlt in UML

Ähnliches Konzept

Zwischenzusammenfassung

- EER-Modellierung
  - Spezialisierung
    - Disjunkt vs. überlappend
    - Partiell vs. total
    - Aufzählungstypen für definierende Attribute
  - Generalisierung als inverse Spezialisierung
    - is-a Beziehung als alternative Darstellung
- Vorgehensmodelle zur konzeptuellen Modellierung
  - Top-down
  - Bottom-up
  - Inside-out



Überblick: 2. Datenbank-Modellierung

A. Motivation
- Modelle und Modellierung

B. Entity-Relationship-Modell (ER)
- Komponenten
- Von Anforderungen zum ER-Modell
- Interpretation von ER-Modellen

C. Enhanced ER-Modell (EER)
- Spezialisierung, Generalisierung
- Modellierungsvorgehen
- Beziehung zu UML

D. Dokumentation & Bewertung
- Dokumentation
- Qualitätskriterien EER

Dokumentation

- Ein (E)ER-Schema ist i.d.R. nicht ausreichend, um alle interessierenden Details einer betrachteten Miniwelt zu beschreiben.
- Beispiele:
  - Jede Abteilung hat wenigstens fünf Mitarbeiter, die der Firma jeweils mehr als zehn Jahre angehören.
  - In keiner Abteilung gibt es Mitarbeiter, die ein höheres Gehalt bekommen als ihr jeweiliger Abteilungsleiter.
  - „JahresSonderzahlung“ ist ein abgeleitetes Attribut, welches sich stets wie folgt errechnet:
    - JahresSonderzahlung = 0,01 · Projektbudget
  - Ein Projektbudget liegt nie unter 50T€
- Lösung → Geschäftsregeln (Business Rules)

Geschäftsregeln (Business Rules)

- Beschreibend (*descriptions*):
  - strukturierte Dokumentation: z.B. Tabellen
- Einschränkungen (*constraints*):
  - <Konzept> muss / darf nicht <formale Einschränkung>
- Spezifikation abgeleiteter Größen (*derivations*):
  - <Konzept> wird gebildet über <Operation zur Gewinnung des Konzepts>

Beispiel: Beschreibende Regeln

Entität	Beschreibung	Attribute	Identifikator
ANGESTELLTE	Angestellter der Firma.	SozVersNr, GebDatum, Name, Geschlecht, Adresse, Gehalt	SozVersNr
PROJEKT	Ein Projekt der Firma, bei dem Angestellte mitwirken.	Name, Nummer, Standort	Nummer
...	...	...	...

Relation	Beschreibung	Beteiligte Entitäten	Attribute
LEITET	Ordnet einen Angestellten in der Rolle des Abteilungsleiters einer Abteilung zu.	Angestellter (1, 1) Abteilung (1, 1)	Anfangsdatum
ARBEITET_FÜR	Ordnet einen Angestellten einer Abteilung in der Rolle des Mitarbeiters zu.	Angestellter (1, N) Abteilung (1, 1)	
...	...	...	...

Beispiel: Einschränkungen & Ableitungen

Einschränkungen (Constraints)	
(BR1)	Ein Abteilungsleiter muss auch Mitarbeiter der Abteilung sein.
(BR2)	Ein Mitarbeiter darf kein höheres Gehalt beziehen als das des Abteilungsleiters der Abteilung, zu der er gehört.
(BR3)	Eine Abteilung mit Standort in Schweden muss von einem Mitarbeiter geleitet werden, der mindestens zehn Jahre durchgehend bei der Firma angestellt ist.
(BR4)	Ein Mitarbeiter, der keiner Abteilung zugeordnet ist, darf an keinem Projekt mitwirken.
...	
Ableitungen (Derivations)	
(BR7)	Die Anzahl der Angestellten einer Abteilung ergibt sich aus der Anzahl der Angestellten, die über die Relation ARBEITET_FÜR mit der jeweiligen Abteilung in Beziehung stehen.
...	

Qualitätskriterien für (E)ER-Schemata

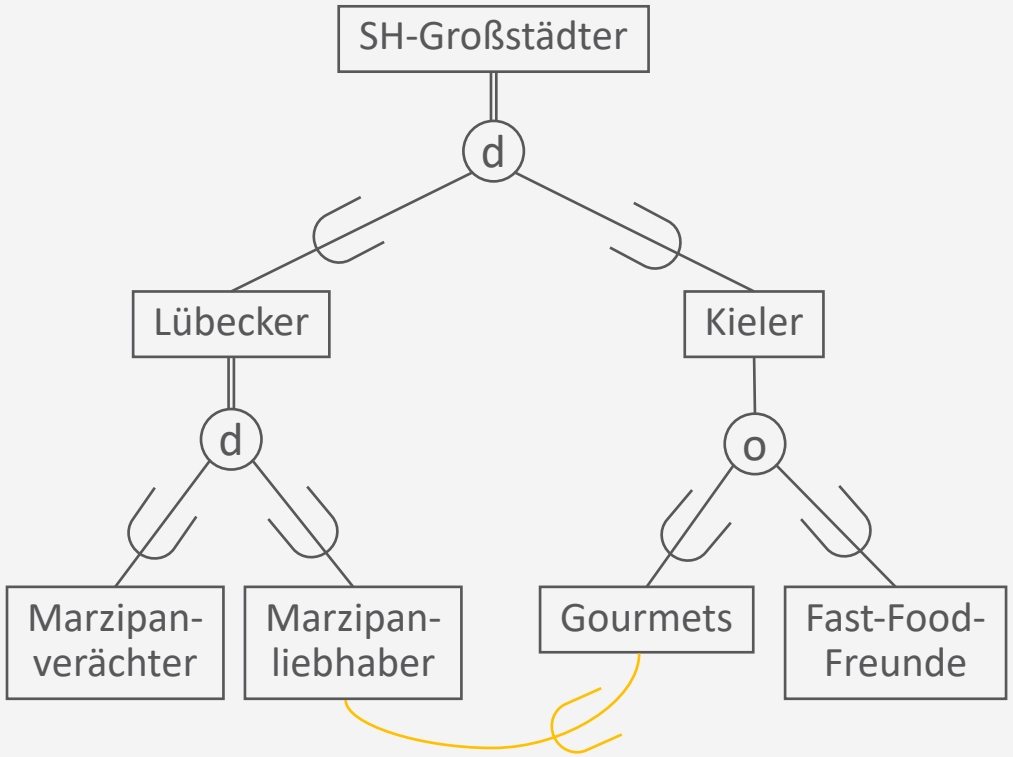
- Korrektheit
  - Modellierten Entitäten, Beziehungen, Attribute sind in der zugrunde liegenden Miniwelt enthalten, werden getreu der gegebenen Miniwelt-Darstellung in einen Zusammenhang gesetzt
- Vollständigkeit
  - Miniwelt-Darstellung wird (soweit möglich) vollständig durch das (E)ER-Schema wiedergegeben
- Minimalität
  - Im (E)ER-Schema berücksichtigten Konzepte, Beziehungen, Attribute der Miniwelt sind möglichst redundanzfrei modelliert
- Lesbarkeit
  - (E)ER-Schema ist übersichtlich organisiert, zeigt einen systematischen Aufbau
- Verständlichkeit
  - Modellierte (logische) Zusammenhänge entsprechen einer „natürlichen“ Intuition

Konsistenzprüfungen möglich

- Ein Beispiel aus dem Norden

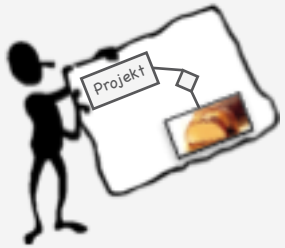
- Es gibt keine Marzipanliebhaber mehr
  → Alle Lübecker sind Marzipanverächter?!

- Konsistenzprüfungen
  - Automatische Bestimmung von leeren Begriffen
  - Prüfung auf globale Konsistenz



Zwischenzusammenfassung

- Dokumentation von (E)ER-Schemata: Business Rules
  - Beschreibungen
  - Einschränkungen
  - Ableitungen
- Qualitätskriterien für (E)ER-Schemata
  - Korrektheit, Vollständigkeit, Minimalität, Lesbarkeit, Verständlichkeit
  - EER-Schemata erlauben Konsistenzprüfungen



Überblick: 2. Datenbank-Modellierung

A. Motivation
- Modelle und Modellierung

B. Entity-Relationship-Modell (ER)
- Komponenten
- Von Anforderungen zum ER-Modell
- Interpretation von ER-Modellen

C. Enhanced ER-Modell (EER)
- Spezialisierung, Generalisierung
- Modellierungsvorgehen
- Beziehung zu UML

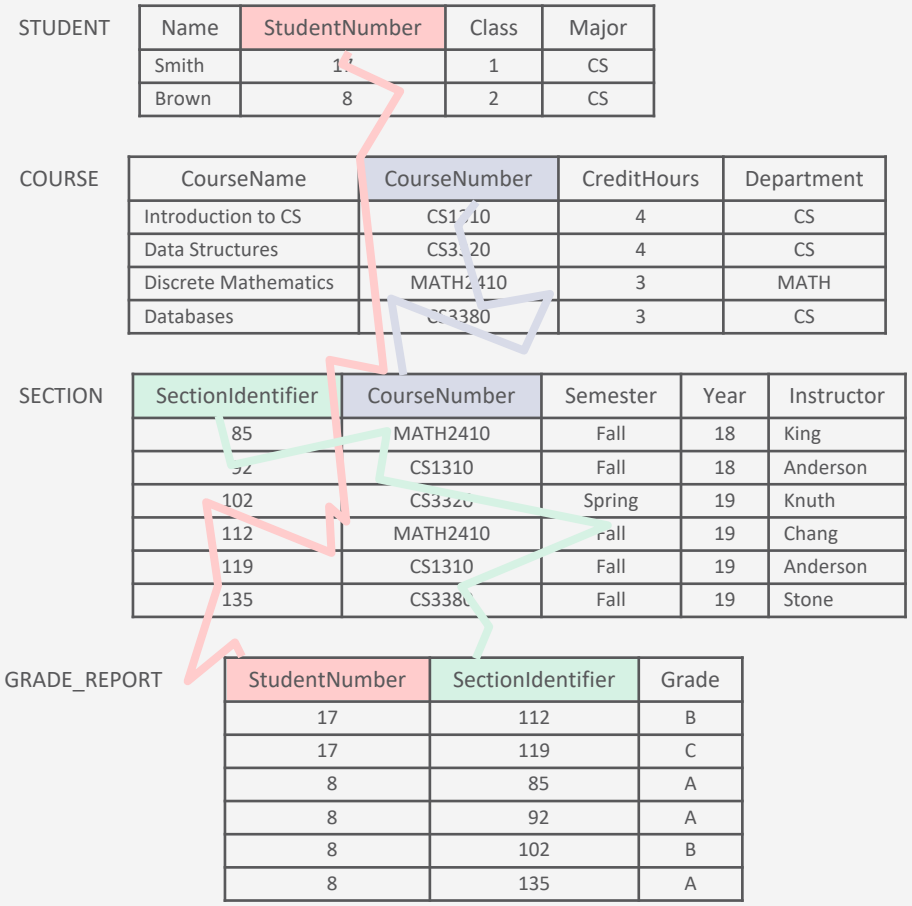
D. Dokumentation & Bewertung
- Dokumentation
- Qualitätskriterien EER

→ Das relationale Modell

WWU
MÜNSTER

Das relationale Datenmodell

Datenbanken



STUDENT

Name	StudentNumber	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

GRADE_REPORT

StudentNumber	SectionIdentifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

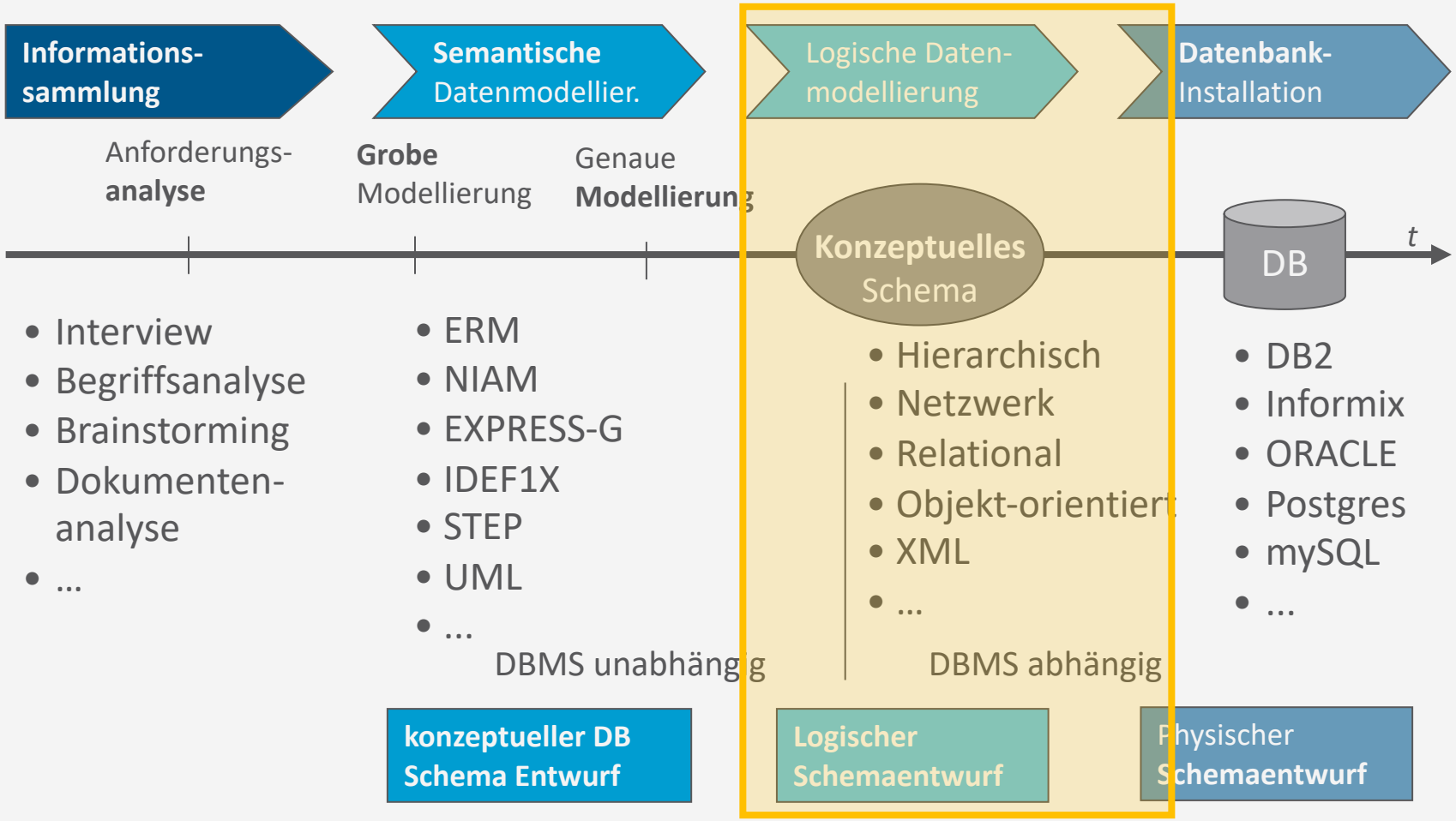
Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Übersicht: 3. Das relationale Datenmodell

A. Relationales Datenmodell

- Relationen, Attribute, relationale Datenbanken und –schemata
- Schlüssel: Primärschlüssel, Fremdschlüssel, referentielle Integrität

B. Entwurf relationaler Schemata

- Vom ER-Diagramm zum relationalen Modell

C. Relationale Algebra

- $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$
- Minimalität
- Aggregieren, gruppieren
- Einfügen, löschen, aktualisieren

Ein erstes relationales Modell (RM)

- DB einer Universität: Tabellen
  - Studierende
  - Kurse
  - Arbeitsgruppen
  - Noten
  - Voraussetzungen
- Datenelemente von unterschiedlichem Typ
  - String, Integer, etc.
  - Schlüssel
- Logische Zusammenhänge
  - Innerhalb einer Tabelle
  - Zwischen Tabellen (über Schlüssel)

STUDENT

Name	StudentNumber	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

GRADE_REPORT

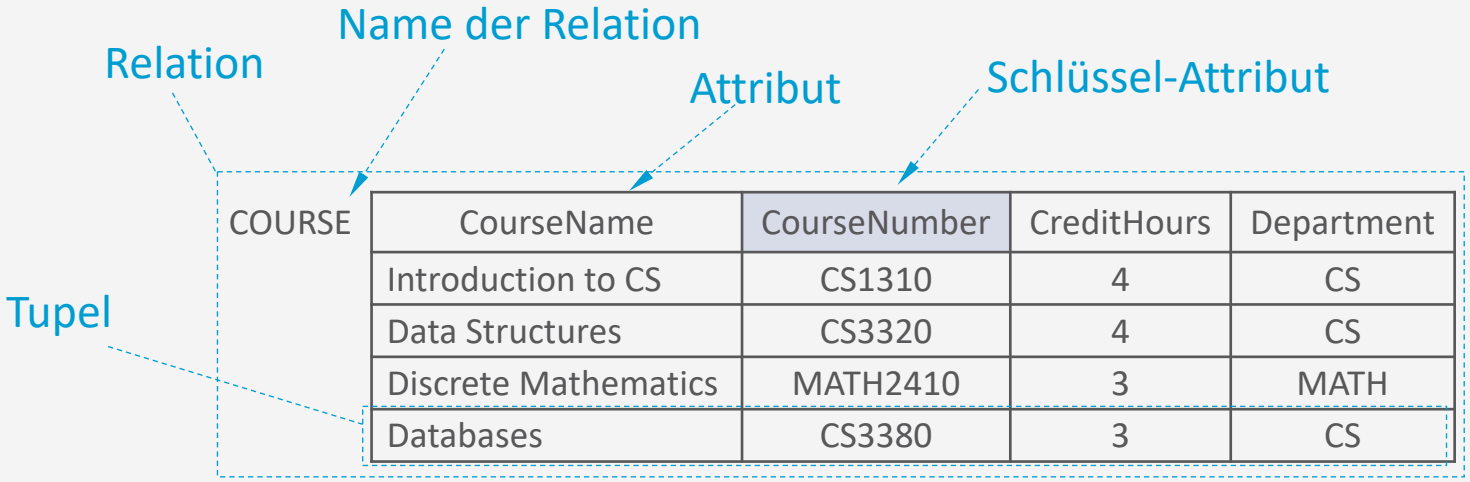
StudentNumber	SectionIdentifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

CourseNumber	PrerequisiteNumber
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Relationen

- Relation, Name der Relation, Attribut und Tupel:



COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

- Namen der Relation und Attribute:
  - Frei wählbar; sinnvoll sind sprechende Bezeichner
- Wertebereich / Domain eines Attributs
  - Datentyp, der die Wertetypen der Attribute beschreibt

Wertebereiche/Domänen von Attributen

- Wertebereich $\boldsymbol{D}$: eine Menge atomarer Werte
  - Jeder einzelne Wert ist aus Sicht des relationalen Modells unteilbar
- Festlegung: Angabe eines Datentyps, dessen Datenwerte den Wertebereich bilden.
  - Angabe eines Namens vereinfacht die Interpretation der Werte
  - Angabe eines Datenformats konkretisiert die Darstellung
  - Ggf. Angabe einer Maßeinheit
- Beispiel

USA_Phone_Numbers:

Der Datentyp ist die Menge der zehnstelligen Telefonnummern, die in den USA gültig sind.

Das Datenformat für USA_Phone_Numbers kann als Zeichenkette im Format ddd-dd-ddddd deklariert werden, wobei jedes einzelne d für eine Ziffer (0|1|2|3|4|5|6|7|8|9) steht

Typische Datentypen für Wertebereiche

- Numerische Standardtypen, z.B.
  - short-integer
  - integer
  - long-integer
  - float
  - double-float
- Zeichenketten, z.B.
  - string
  - char(255)
- Spezielle Datentypen, z.B.
  - date
  - timestamp
- Benutzerdefinierte Datentypen, z.B.
  - Persistent identifier
  - Sozialversicherungsnummer

Relationenschemata und Relationen

- Relationenschema $R(A_1, \ldots, A_n)$:
  - (DB-weit eindeutiger) Name R
  - Liste von Attributen $A_1, \ldots, A_n$
    - Attribut A_i: Name einer Rolle, die ein Wertebereich in R spielt
    - Wertebereich $\boldsymbol{D}$ von A_i: $\mathrm{dom}(A_i) = \boldsymbol{D}$
- Relation r wird über R identifiziert
  - „Intension“ → Relationenschema R
  - „Extension“ → „aktuelle“ Relation $r(R)$ oder r
- Grad (degree) von R bzw. r:
  - die Anzahl n von Attributen, die R bzw. r bilden

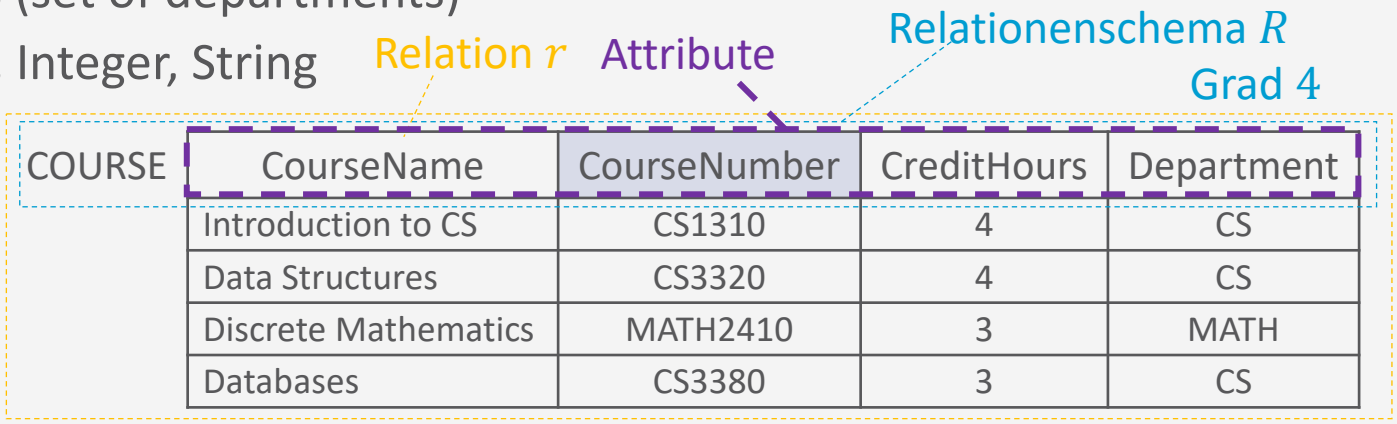
Relation r Attribute Relationenschema R Grad 4

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Relationenschemata und Relationen

- Beispiel:
  - Relationenschema: COURSE(CourseName, CourseNumber, CreditHours, Department)
  - Wertebereiche für Attribute (hier alle benutzerdefiniert)
    - dom(CourseName) = CourseNames (set of course names)
    - dom(CourseNumber) = CourseNumbers (set of course numbers)
    - dom(CreditHours) = CreditHours (set of credit hours)
    - dom(Department) = Departments (set of departments)
    - Allgemeinere Alternativen: String, String, Integer, String



COURSE	CourseName	CourseNumber	CreditHours	Department
	Introduction to CS	CS1310	4	CS
	Data Structures	CS3320	4	CS
	Discrete Mathematics	MATH2410	3	MATH
	Databases	CS3380	3	CS

Relationen/Relationenzustände

- Eine Relation (oder ein Relationenzustand) r bzw. $r(R)$ eines Relationenschemas $R(A_1, \dots, A_n)$ ist eine Menge von n-Tupeln t_j mit $j = 1, \dots, m$:
  - $r = \{t_1, \dots, t_m\}$
  - $m =$ Kardinalität von r (Anzahl an Tupeln/Zeilen in r)
- Jedes n-Tupel t_j ist eine geordnete Liste von n Werten:
  - $t_j = \langle v_1, \dots, v_n \rangle$
- Jeder Wert v_i:
  - entweder ein Element aus $\mathrm{dom}(A_i)$
  - oder NULL (spezieller Wert, bedeutet „undefiniert“ oder „unbekannt“)

$m = 4$

COURSE

	CourseName	CourseNumber	CreditHours	Department
t_1	Introduction to CS	CS1310	4	CS
t_2	Data Structures	CS3320	4	CS
t_3	Discrete Mathematics	MATH2410	3	MATH
t_4	Databases	CS3380	3	CS

Relationen/Qualifizierung

- Gegeben
  - $R(A_1, \ldots, A_n)$ ein Relationenschema n-ten Grades
  - $t = \langle v_1, \ldots, v_n \rangle$ ein Tupel der Relation $r(R)$
    - v_i ist der Wert, der im Tupel t dem Attribut A_i entspricht
- $R.A$: qualifiziert ein Attribut A mit dem Relationennamen R
- Für einzelne Komponentenwerte von einem Tupel t gilt:
  - $t[A_i]$ bzw. $t.A_i$ beziehen sich auf den Wert v_i in t für Attribut A_i
  - $t[A_u, \ldots, A_z]$ und $t.(A_u, \ldots, A_z)$ beziehen sich auf Werte $\langle v_u, \ldots, v_z \rangle$ von Subtupeln von t, die den Attributen $A_u, \ldots, A_z$ von R entsprechen

Course.CourseName

$t_1[CreditHours] = t_1.CreditHours$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Mathematische Sichtweise von Relationen

- Eine Relation $r(R)$ ist eine mathematische Relation n-ten Grades auf die Wertebereiche $\mathrm{dom}(A_1), \ldots, \mathrm{dom}(A_n)$.
- $r(R)$: Teilmenge des kartesischen Produkts der Wertebereiche, die R definieren:
  - $r(R) \subseteq \left(\mathrm{dom}(A_1) \times \cdots \times \mathrm{dom}(A_n)\right)$
  - Das kartesische Produkt spezifiziert alle Wertekombinationen der zugrundeliegenden Wertebereiche, d.h. alle möglichen Tupel
  - $r(R)$ alle tatsächlichen Tupel
- Beispiel
  - course(COURSE) hat vier tatsächliche Tupel

Wie sehen alle möglichen Tupel aus?

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Umfang

- Anzahl der möglichen Werte / Kardinalität eines Wertebereichs:
  - $|\mathrm{dom}(A_i)|$
- Damit die Anzahl möglicher unterschiedlicher Tupel:
  - $|\mathrm{dom}(A_1)| \cdot \cdots \cdot |\mathrm{dom}(A_n)|$
- Davon enthält der aktuelle Relationszustand i.d.R. nur einen Ausschnitt (zum Glück)
  - Anzahl tatsächlicher Tupel = Kardinalität m der Relation

$$m \ll |\mathrm{dom}(A_1)| \cdot \cdots \cdot |\mathrm{dom}(A_n)|$$

- Wertebereich kann einschränkend wirken (String vs. Menge von Kursnamen)
  - Logische Zusammenhänge zwischen Attributen schränken die möglichen Tupel weiter ein

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Keine Ordnung im relationalen Modell!

- Erinnerung: r bzw. $r(R)$ eines Relationenschemas $R(A_1, \dots, A_n)$ ist eine Menge von n-Tupeln t_j mit $j = 1, \dots, m$:
  - $r = \{t_1, \dots, t_m\}$
- Eine Menge hat keine Ordnung
  → keine Reihenfolge im relationalen Modell

COURSE

CourseName	CourseNumber	CreditHours	Department
Discrete Mathematics	MATH2410	3	MATH
Introduction to CS	CS1310	4	CS
Databases	CS3380	3	CS
Data Structures	CS3320	4	CS

=

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

„Dynamik“

- Relation $r(R)$ variiert über die Zeit
  - Änderungen an Tupelwerten
  - Löschen und Hinzufügen von Tupeln
  - Beispiele
    - Die Stunden des Kurses „Database“ steigen auf 4
    - Neuer Kurs „Algorithms“
- Relationenschema R ändert sich hingegen selten
  - Schema-Evolution: Re-Design des DB-Schemas und damit der darin enthaltenen Relationenschemata
    - Bestehende Daten müssen entsprechend angepasst werden

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	~~3~~ 4	CS
Algorithms	CS3390	4	CS

Spezielle Werte in Tupeln: NULL-Werte

- NULL-Wert kann eingetragen werden, wenn richtiger Wert unbekannt oder nicht zutreffend ist
- Problem: unterschiedliche Interpretation möglich
  - Wert ist nicht vorhanden
  - Wert ist im Kontext des Tupels nicht zutreffend
  - Es ist unbekannt, ob der Wert nicht vorhanden oder nicht zutreffend ist
    - Kann mit der *Closed World Assumption* brechen
  - NULL ≠ NULL (in einer Spalte)
    - Unbekannte, aber möglicherweise unterschiedliche Werte

Noch nicht bekannt

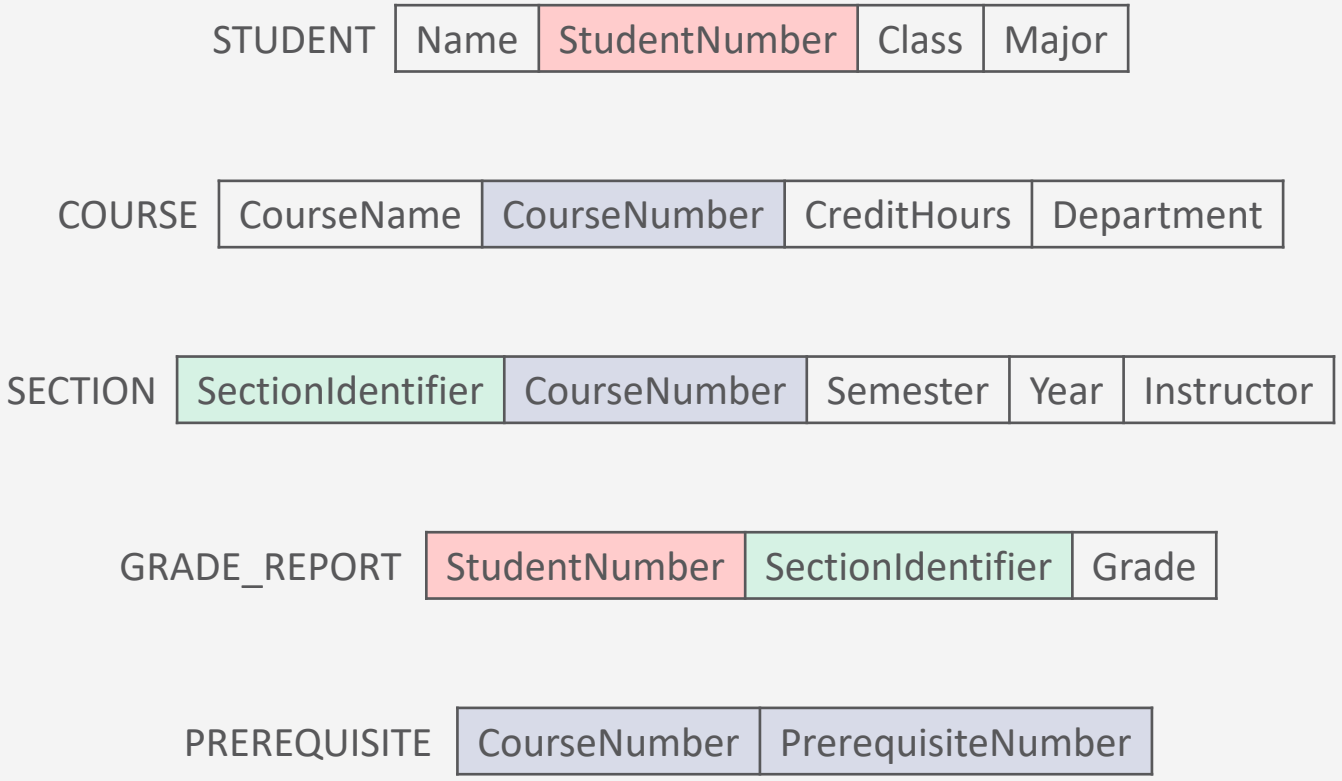
COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS
Algorithms	CS3390	4	NULL

Das relationale Datenmodell und relationale DBs

- Die meisten Datenbanken sind relationale Datenbanken
  - Ihnen unterliegt ein relationales Datenbankschema
- Ein relationales Datenbankschema DS umfasst:
  - Menge von Relationenschemata:
    $$DS = \{R_1, \dots, R_m\}$$
  - Menge von Integritätsbedingungen IC (*integrity constraints*)
  - „Intension“
- Ein Datenbankzustand DB (zum Zeitpunkt x) von DS umfasst die Menge der zum Zeitpunkt x aktuellen Relationszustände, d.h.
  $$DB = \{r_1, \dots, r_m\}$$
  - „Extension“
- Für $DB = \{r_1, \dots, r_m\}$ gilt:
  - Jedes r_i ist ein zulässiger Zustand für R_i
  - Die Gesamtheit der Relationszustände r_i erfüllen die in IC spezifizierten Integritätsbedingungen

Beispiel eines DB-Schemas



Beispiel eines DB-Zustands

STUDENT

Name	StudentNumber	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

GRADE_REPORT

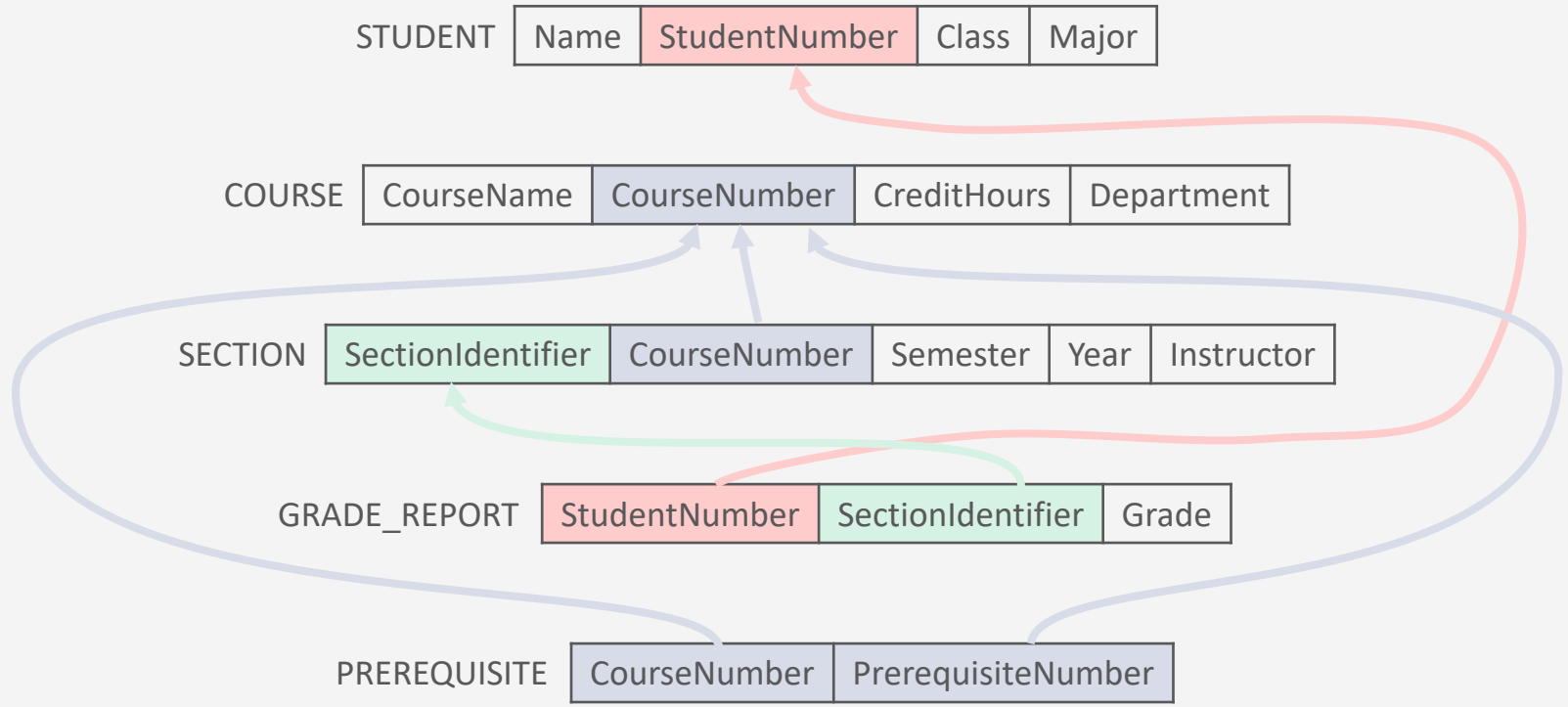
StudentNumber	SectionIdentifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

CourseNumber	PrerequisiteNumber
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

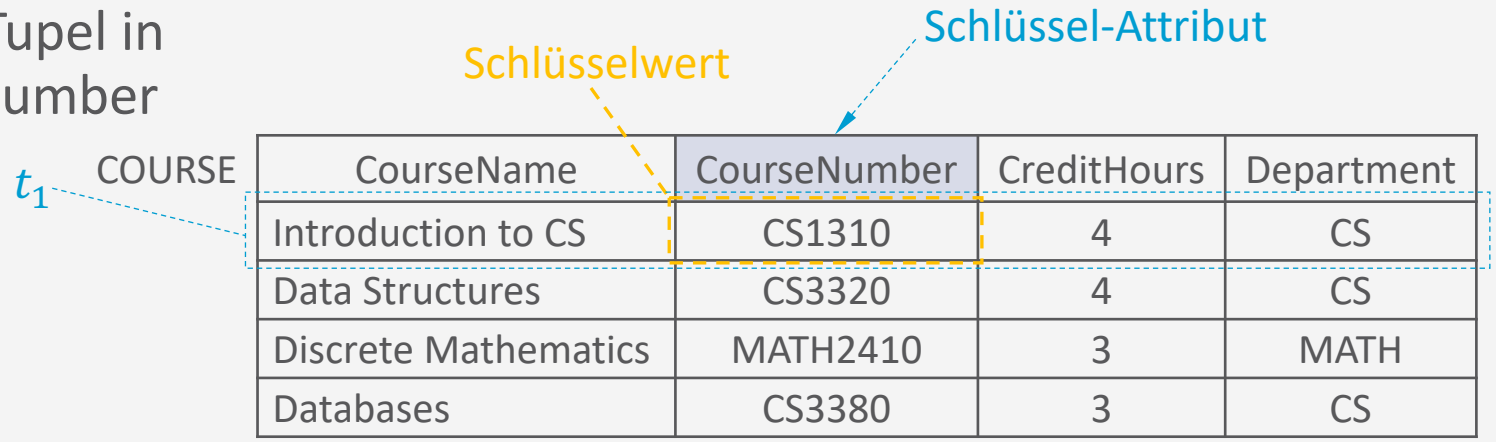
Schlüssel

Spezielle Attribute



Schlüssel

- Jede Relation besitzt einen Primärschlüssel, der ein einzelnes Attribut oder eine Kombination von Attributen ist, so dass eine eindeutige Identifikation jedes Tupels innerhalb der Relation ermöglicht wird
  - Ein Schlüsselwert v identifiziert ein Tupel t
- Beispiel
  - Das Attribut CourseNumber ist Schlüssel für die Relation COURSE
  - CourseNumber identifiziert jedes Tupel in COURSE eindeutig, da die CourseNumber einzigartig für jeden Kurs ist
  - Beispiel: Schlüsselwert CS1310 identifiziert Tupel t_1



CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Schlüssel: Definitionen

- Für Relationenschemata gilt i.d.R.:
  - Es gibt Teilmengen SK der Attribute von R, für die zwei Tupel eines Relationenzustands $r(R)$ nie die gleiche Wertkombination besitzen dürfen; d.h.:
    - Für alle Tupel t_1 und t_2 gilt: $t_1[SK] \neq t_2[SK]$
      - Alle Tupel sind bzgl. der Attributmenge SK eindeutig
- Alle solchen Teilmengen SK heißen Superschlüssel
  - Trivialer Superschüssel: alle Attribute von R
- Minimaler Superschlüssel = Schlüssel
  - Minimal: kein Attribut kann weggelassen werden, ohne dass die Schlüsseleigenschaft verloren geht

trivialer Superschlüssel

minimaler Schlüssel

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Schlüssel und Primärschlüssel

- Relationen können mehrere Schlüssel (minimale Superschlüssel) enthalten
  - Genannt Schlüsselkandidaten
  - Attribute eines Schlüsselkandidaten: Prime-Attribute
- Einer der Schlüsselkandidaten wird gewählt als Primärschlüssel
  - Attribute des Primärschlüssels **(dürfen nicht NULL sein)**: Primärschlüssel-Attribute
- Oft ist die Einführung eines künstlichen Schlüssels sinnvoll
  - z.B. eine eindeutige Nummer (ID)

gewählter Primärschlüssel

Schlüsselkandidaten

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Schlüssel

- Eine Relation mit Schlüssel repräsentiert eine Funktion von den Primärschlüsselattributen zu den Nicht-Schlüsselattributen
  - Ein Schlüsselwert v identifiziert ein Tupel t
  - Damit erhält man die Attributwerte von t
- Beispiel
  - Schlüsselwert CS1310 identifiziert Tupel t_1
  - CS1310 → (Introduction to CS, CS1310, 4, CS) bzw.
    - CS1310 → Introduction to CS
    - CS1310 → CS1310
    - CS1310 → 4
    - CS1310 → CS

Schlüsselwert

t_1

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Beziehungen zwischen Tupeln

- Identifikation des referenzierten Objektes über seinen Primärschlüssel (→ assoziative Identifikation)
  - Einen Schlüssel, der in Relation R_1 zur Identifikation eines Tupels in Relation R_2 benutzt wird, bezeichnet man als Fremdschlüssel
  - Rekursive Beziehungen führen zu reflexiven Fremdschlüsseldeklarationen ($R_1 = R_2$)
    - Beispiel: Angestellte : Vorgesetze
    - Beispiel: CourseNumber : PrerequisiteNumber

Fremdschlüssel

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	23	King
92	CS1310	Fall	23	Anderson
102	CS3320	Spring	24	Knuth
112	MATH2410	Fall	24	Chang
119	CS1310	Fall	24	Anderson
135	CS3380	Fall	24	Stone

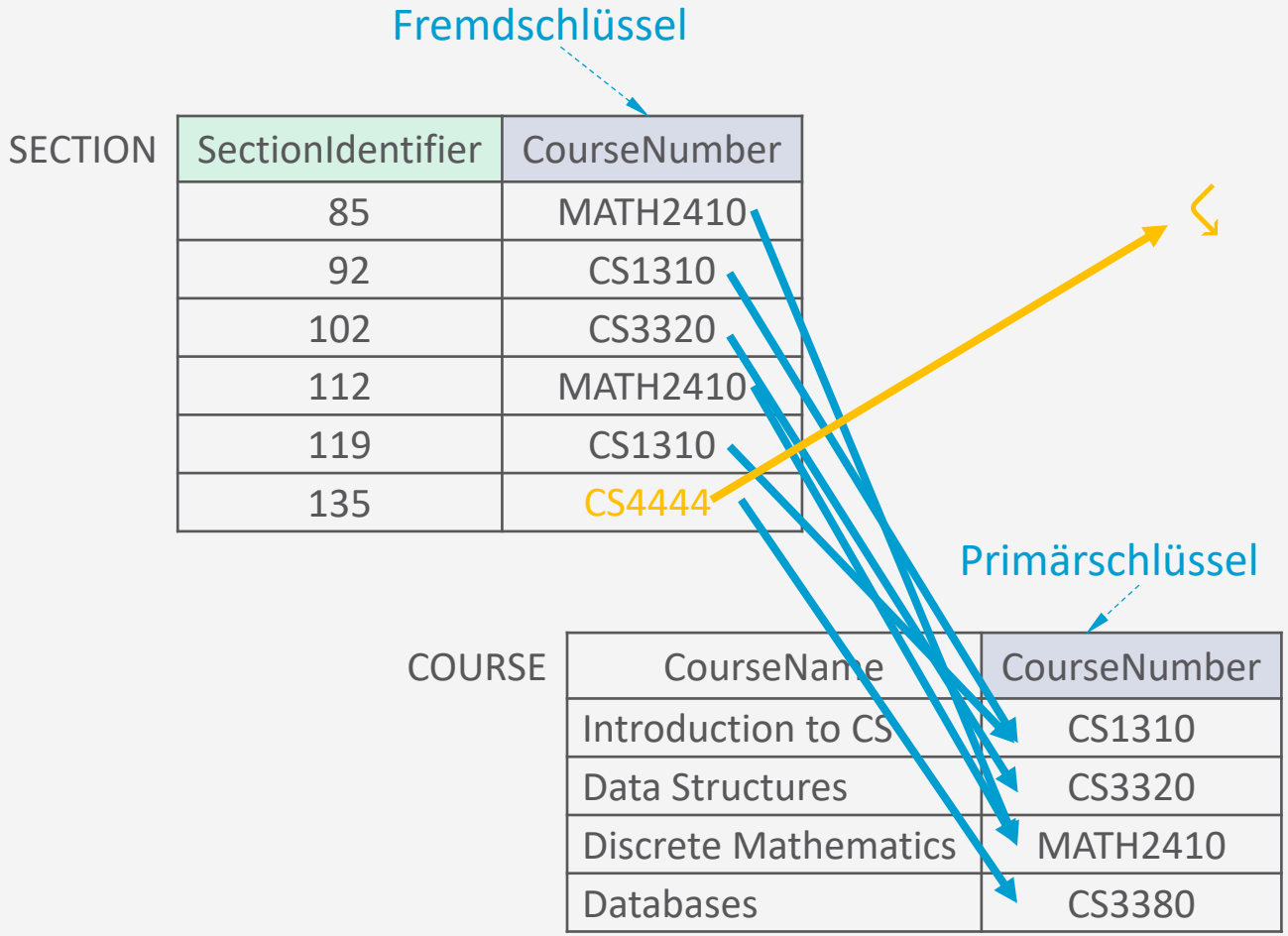
Primärschlüssel

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

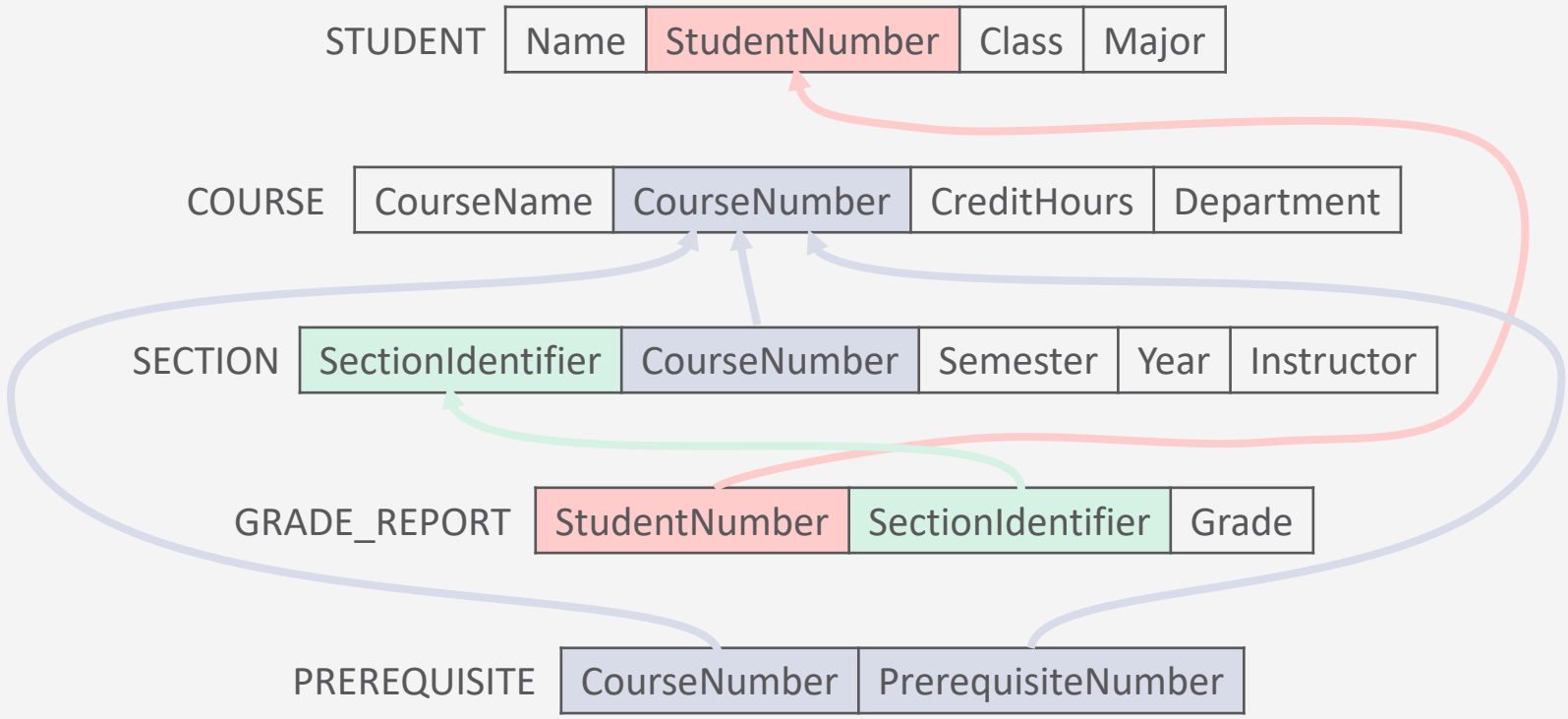
Fremdschlüssel: Definition

- Eine Attributmenge FK im Schema R_1 ist ein Fremdschlüssel von R_1, der auf Schema R_2 referenziert, falls gilt:
  1. Die Attribute in FK haben die gleichen Wertebereiche wie die Primärschlüssel-Attribute PK von R_2
     - Die Attribute FK gelten als Referenz auf R_2
  2. Ein Wert von FK in einem Tupel t_1 des aktuellen Zustands $r_1(R_1)$ ist
     - NULL (keine Beziehung)
     - Kommt als Wert von PK in einem Tupel t_2 im Zustand $r_2(R_2)$ vor: $t_1[FK] = t_2[PK]$
       - Heißt: Tupel t_1 referenziert t_2



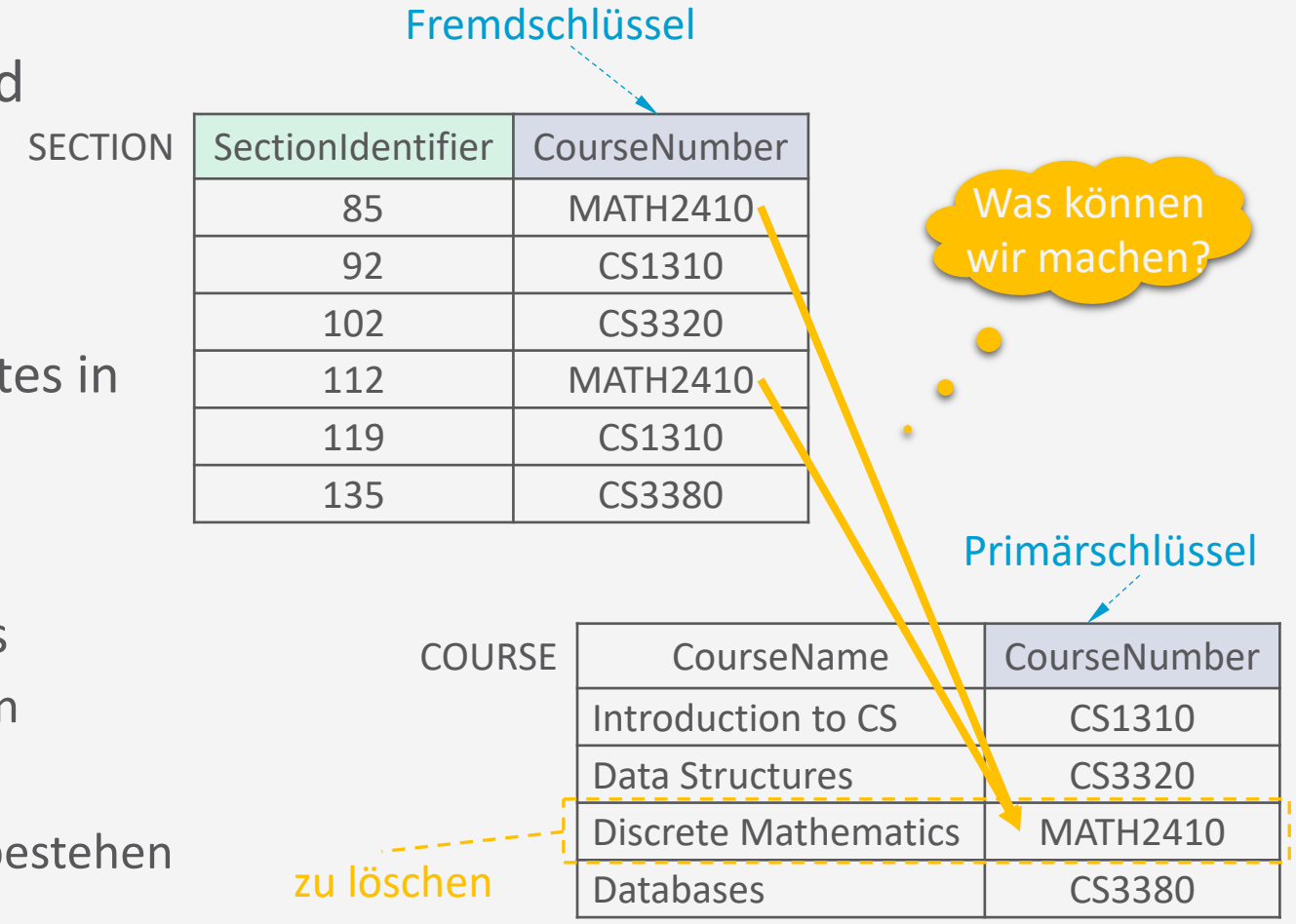
Referentielle Integrität - Fremdschlüssel

- Problem: Inkonsistenz
- Lösung: referentielle Integritätsbedingungen auf Fremdschlüsseln
  - Referentielle Integrität: zu jedem benutzten Fremdschlüssel existiert ein Tupel mit einem entsprechenden Primärschlüsselwert in der referenzierten Tabelle



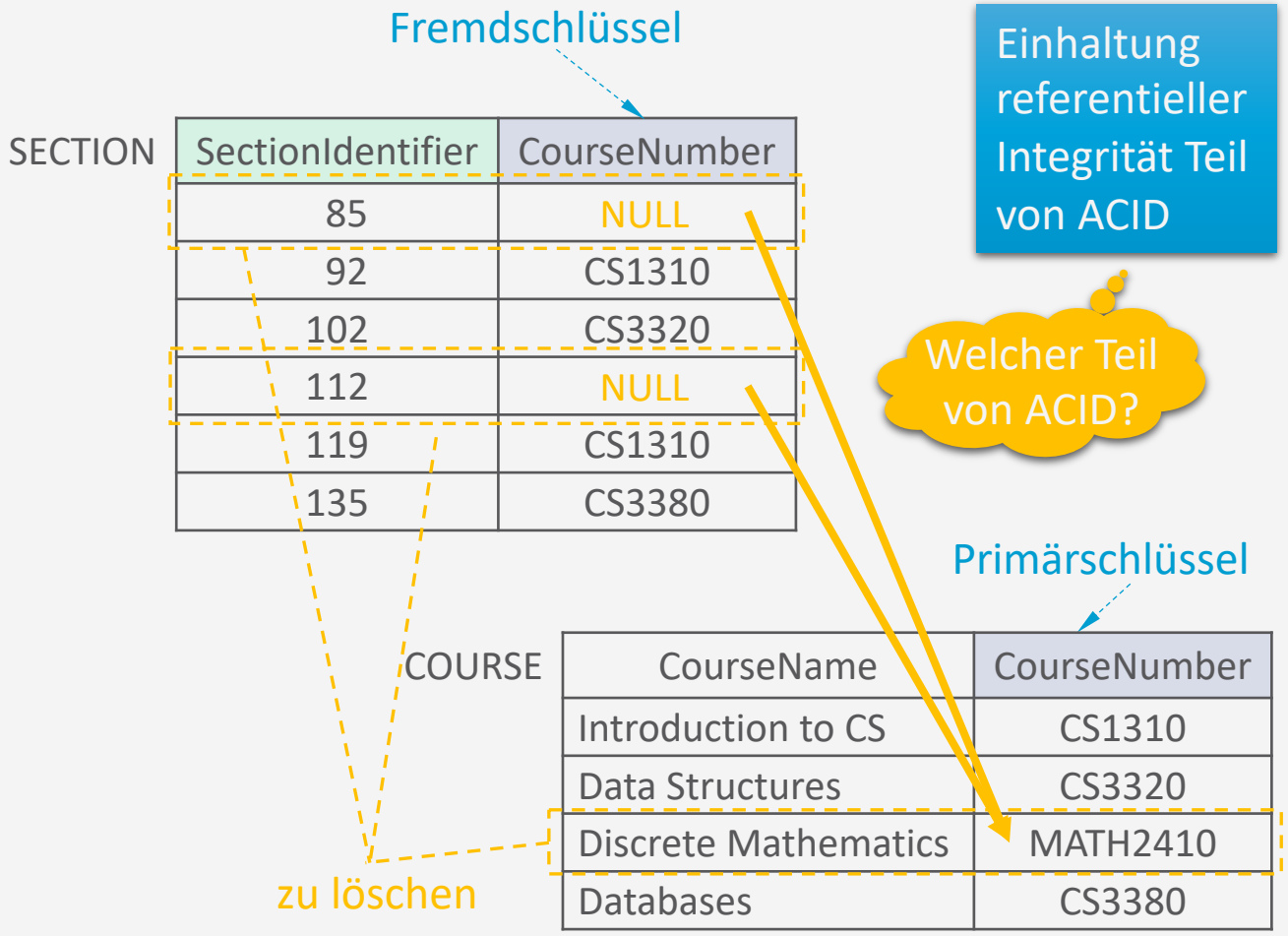
Referentielle Integrität: Konflikte

- Durch Änderungen im Relationenzustand Bedingungen referentieller Integrität möglicherweise verletzt
- Integrität überprüfen notwendig beim
  - Einfügen eines neuen Fremdschlüsselwertes in eine Beziehungstabelle
    - Referenziertes Objekt mit diesem Wert als Primärschlüssel muss existieren
  - Aktualisieren eines Primärschlüsselwertes
    - Alle Referenzen müssen aktualisiert werden
  - Löschen eines Tupels aus einer Relation
    - Auf dieses Tupel dürfen keine Referenzen bestehen



Referentielle Integrität: Konflikte – Löschen

- Gibt es noch Referenzen, bieten sich mehrere Möglichkeiten an, u.a.:
  - Fehlermeldung erzeugen
  - Löschoperation propagieren, so dass das referenzierende Tupel ebenfalls gelöscht wird (→ kaskadiertes Löschen)
  - Referenzen durch Setzen des Fremdschlüssels auf einen Nullwert ungültig machen, sofern dieser nicht Bestandteil des Schlüssels ist
- DBMS bieten Unterstützung zur Verhinderung solcher Konflikte



Zwischenzusammenfassung

- Ein relationales Datenmodell ist eine Menge benannter Relationen
- Eine Relation ist eine Menge von Elementen (Tupeln)
  - deren Struktur durch Attribute definiert,
  - deren Identität durch Schlüssel realisiert und
  - deren Werte durch Domänen kontrolliert werden
- Beziehungen zwischen Relationen werden über Fremdschlüssel realisiert
  - Referentielle Integrität muss gewahrt bleiben!
- Relationen werden meist durch Tabellen dargestellt
  - Zeilen: Elemente der Relation (ein Tupel)
    - Die Zahl der Zeilen ist variabel und wird Kardinalität der Relation genannt
  - Spalten: Attribute der Relation

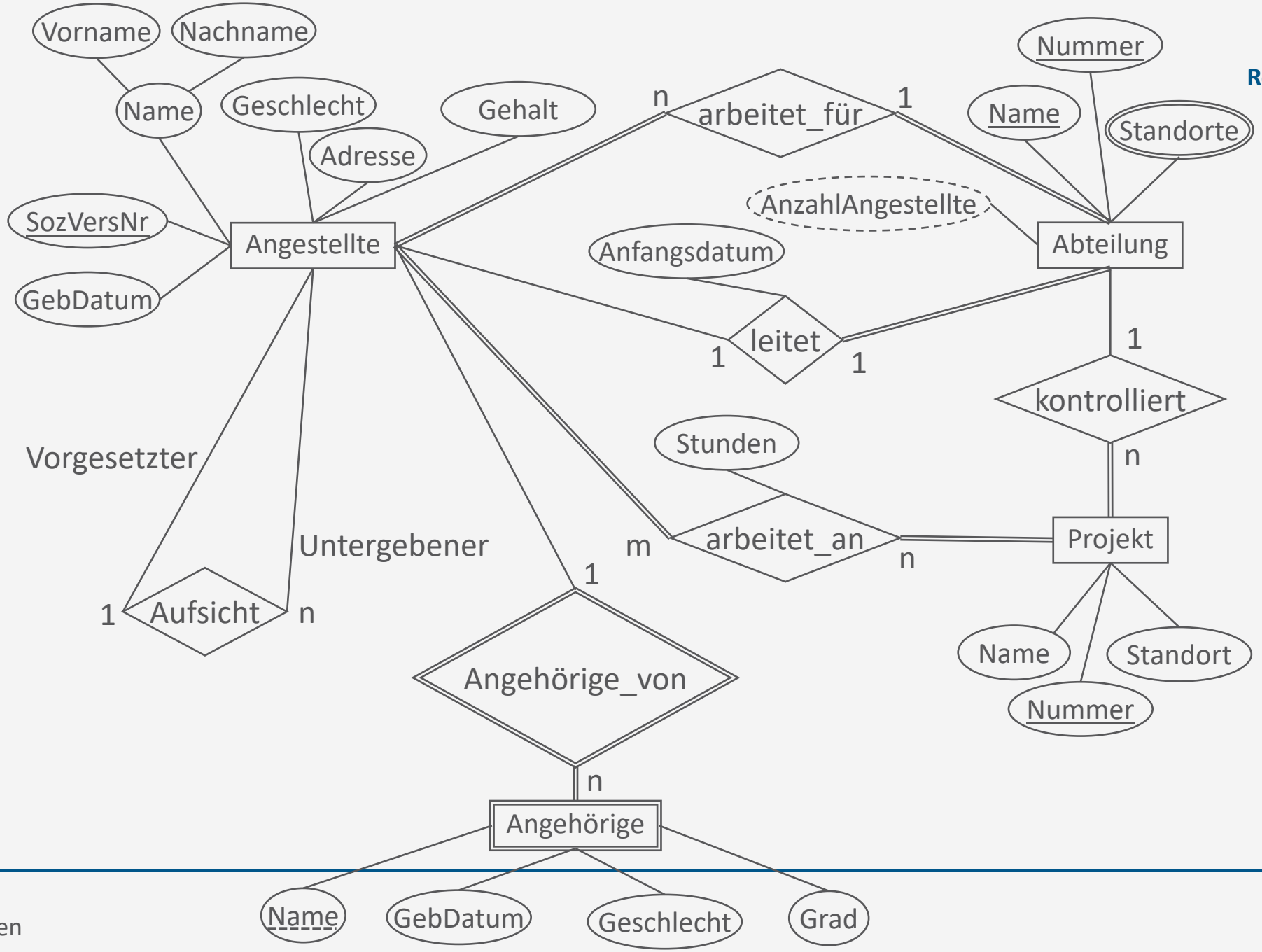
Übersicht: 3. Das relationale Datenmodell

A. *Relationales Datenmodell*
- Relationen, Attribute, relationale Datenbanken und –schemata
- Schlüssel: Primärschlüssel, Fremdschlüssel, referentielle Integrität

B. *Entwurf relationaler Schemata*
- Vom ER-Diagramm zum relationalen Modell

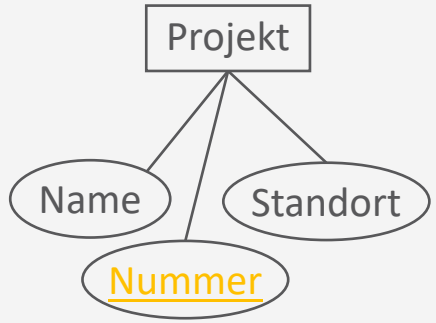
C. *Relationale Algebra*
- $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$
- Minimalität
- Aggregieren, gruppieren
- Einfügen, löschen, aktualisieren

ER-Diagramm


Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
GebDatum
Angestellte
n
arbeitet_für
1
Nummer
Name
Standorte
AnzahlAngestellte
Abteilung
Anfangsdatum
leitet
1
1
1
kontrolliert
n
Stunden
Vorgesetzter
Untergebener
1
Aufsicht
n
m
arbeitet_an
n
Projekt
1
Angehörige_von
n
Name
Standort
Nummer
Angehörige
Name
GebDatum
Geschlecht
Grad

Relationale Darstellung von Entitätstypen

- Übersetzung von Entitäten mit Attributen in Relationen
  - Entität E mit Attributen $A_1, \ldots, A_n$ wird zu Relation $E(A_1, \ldots, A_n)$
  - Wertebereiche für $A_1, \ldots, A_n$ festlegen

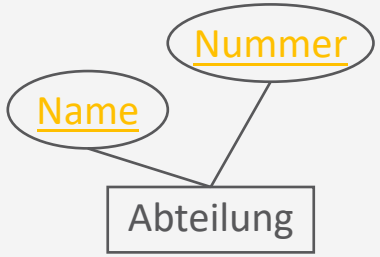


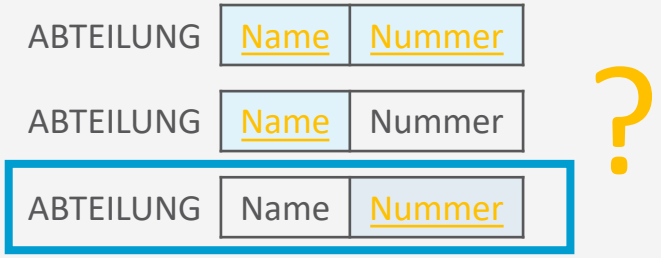
PROJEKT	Nummer	Name	Standort

Wertebereiche
- Nummer: Integer
- Name: String
- Standort: String
- Oder benutzerdefiniert

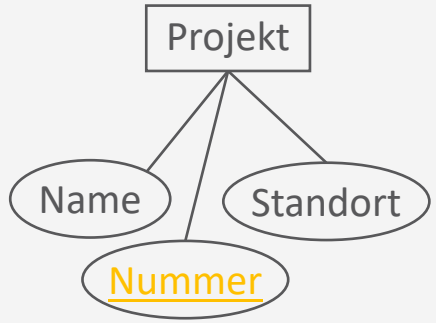
Relationale Darstellung von Entitätstypen

- *Schlüssel*-Attribute
  - In ER-Diagramm identifizierende Attribute angegeben
    - Vorsicht geboten! Nicht immer direkt nutzbar
    - Schlüssel im relationalen Modell minimale Superschlüssel
    - Interpretation der ER-Schlüssel teilweise nicht eindeutig
    - Beispiel: Name und Nummer zusammen eindeutig? Allein eindeutig?





Entscheidung (evtl. nach Rücksprache mit Kunden);
Nummer als Schlüssel von Vorteil, da
- Nummer kleiner als Name
- Vergleiche weniger aufwendig



Wertebereiche
- Nummer: Integer
- Name: String
- Standort: String
- Oder benutzerdefiniert

Relationale Darstellung von Entitätstypen

- *Mehrwertige* Attribute
  - Nicht direkt ins relationale Datenmodell übersetzbar
    - Relationenzustand für ABTEILUNG (Name, Nummer, Standort)

Warum?

ABTEILUNG

Name	Nummer	Standort
XYZ	42	A
XYZ	42	B
XYZ	42	C

Schlüssel nicht mehr eindeutig!

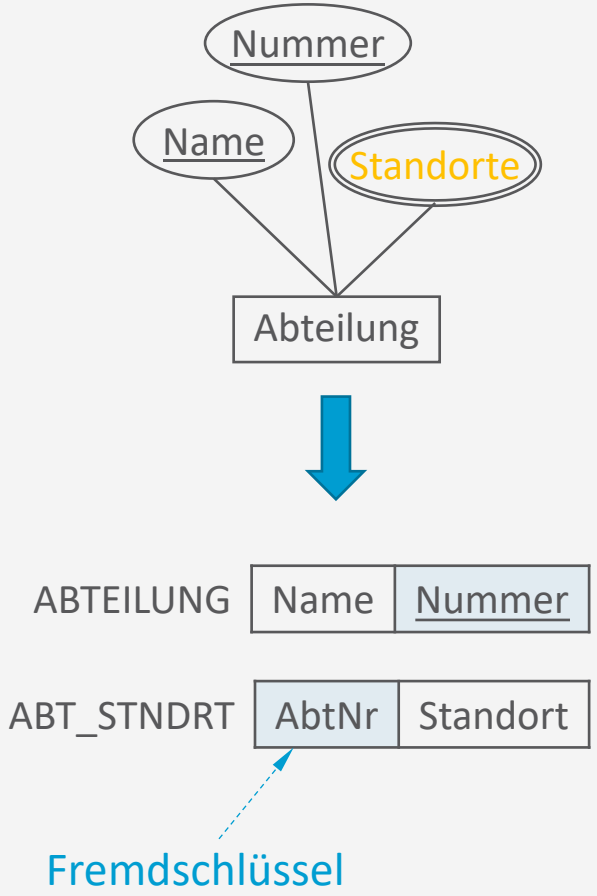
  - Lösung: eigene Relation bauen
    - Übersetzter Relationenzustand

ABTEILUNG

Name	Nummer
XYZ	42

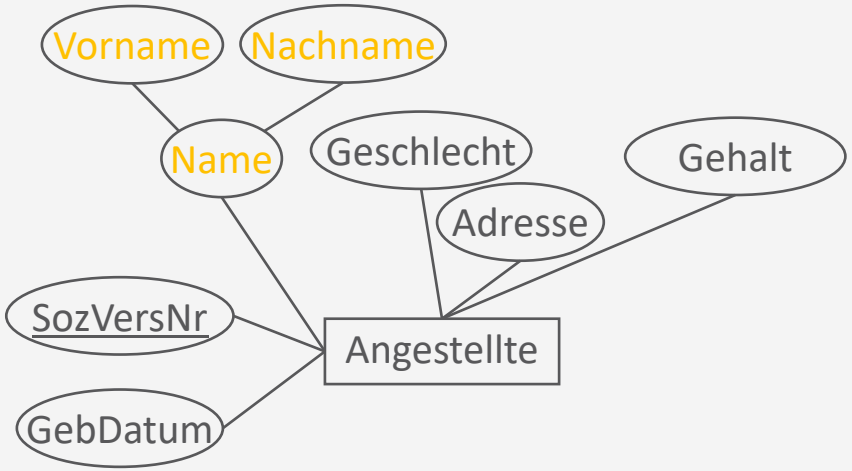
ABT_STNDRT

AbtNr	Standort
42	A
42	B
42	C



Relationale Darstellung von Entitätstypen

- *Zusammengesetzte* Attribute
  - Als ein Attribut in Relation aufnehmen
    - Gibt Teilung auf
  - Als einzelne Attribute in Relation aufnehmen
    - Gibt Zusammengehörigkeit auf
  - Beispiel:
    - Entweder



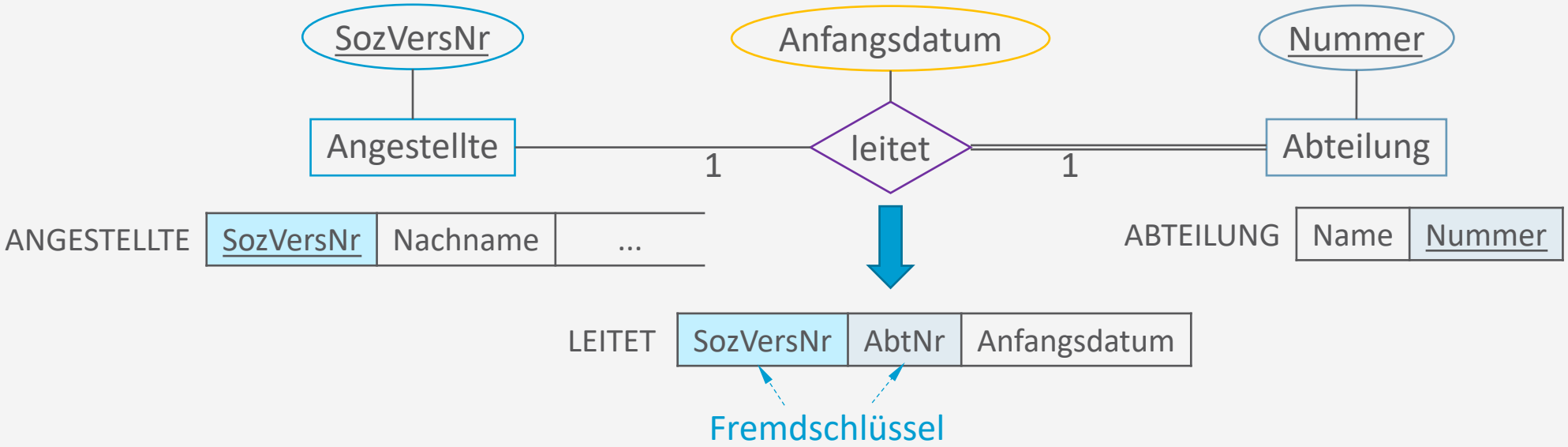
ANGESTELLTE	SozVersNr	Name	Geschlecht	Adresse	Gehalt	GebDatum

    - Oder

ANGESTELLTE	SozVersNr	Nachname	Vorname	Geschlecht	Adresse	Gehalt	GebDatum

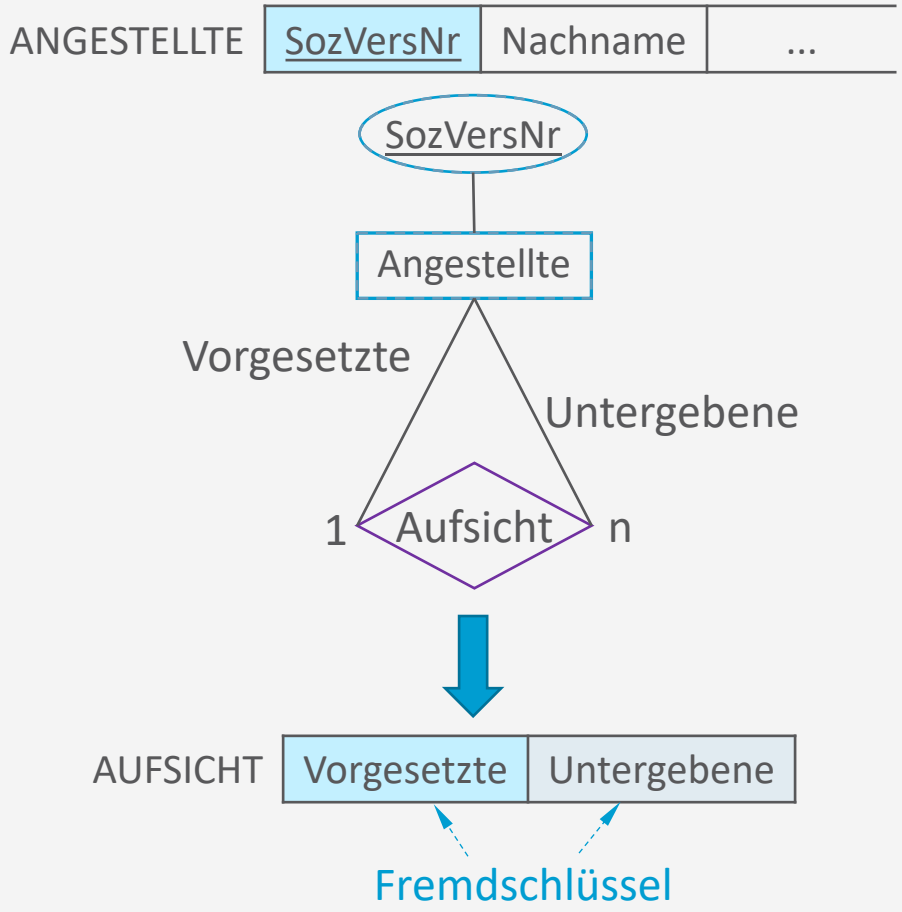
Relationale Darstellung von Assoziationen

- Übersetzung von Assoziationen/Beziehungen mit Attributen in Relationen
  - Beziehung R mit Attributen $A_1, \dots, A_n$
  - Zwischen Entitäten E_1 und E_2 mit Schlüsseln $P_{1,1}, \dots, P_{1,l}$ und $P_{2,1}, \dots, P_{2,m}$
  - Wird zu Relation $R(P_{1,1}, \dots, P_{1,l}, P_{2,1}, \dots, P_{2,m}, A_1, \dots, A_n)$



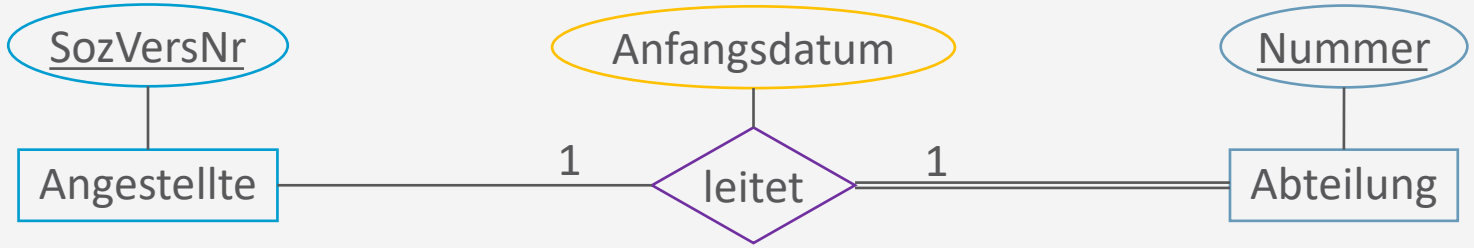
Relationale Darstellung von Assoziationen

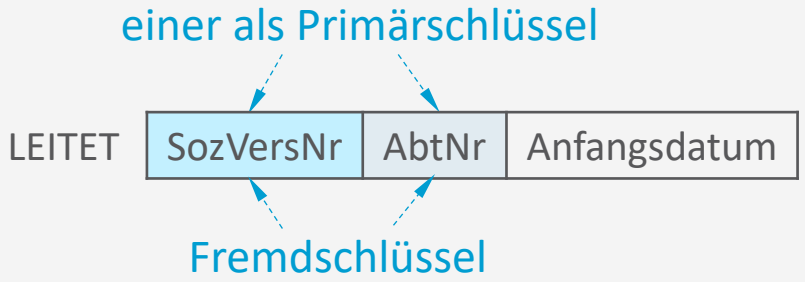
- Auch bei rekursiven Relationen, dann $E_1 = E_2$
  - Beispiel: Vorgesetzte / Untergebene
  - Beziehung R mit Attributen $A_1, \dots, A_n$
  - Zwischen Entitäten E_1 und $E_2 = E_2$ mit Schlüsseln $P_{1,1}, \dots, P_{1,l}$ und $P_{1,1}, \dots, P_{1,l}$
  - Wird zu Relation $R(P_{1,1}, \dots, P_{1,l}, P_{1,1}, \dots, P_{1,l}, A_1, \dots, A_n)$



Relationale Darstellung: Assoziation + Schlüssel

- Häufig abhängig von Fremdschlüsseln
  - Fremdschlüssel $P_{1,1}, \dots, P_{1,l}$ und $P_{2,1}, \dots, P_{2,m}$ in Relation $R(P_{1,1}, \dots, P_{1,l}, P_{2,1}, \dots, P_{2,m}, A_1, \dots, A_n)$
- Primärschlüssel bestimmen über Kardinalitäten
  - 1:1 Beziehung





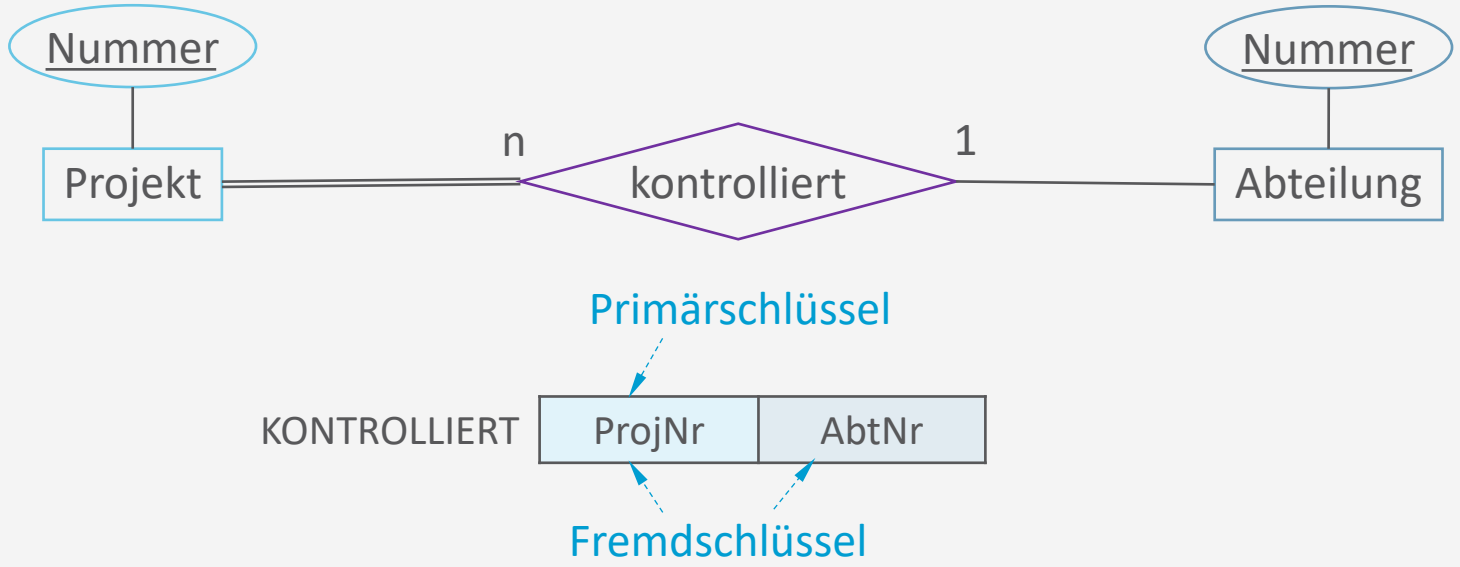
Bei 1:1 Beziehungen kommen beide Fremdschlüssel maximal einmal vor, können also als Schlüssel verwendet werden.

LEITET

SozVersNr	AbtNr	Anfangsdatum
A1	42	01.11.2020
B2	21	15.02.2010
C4	54	01.01.1999

Relationale Darstellung: Assoziation + Schlüssel

- Häufig abhängig von Fremdschlüsseln
  - Fremdschlüssel $P_{1,1}, \dots, P_{1,l}$ und $P_{2,1}, \dots, P_{2,m}$ in Relation $R(P_{1,1}, \dots, P_{1,l}, P_{2,1}, \dots, P_{2,m}, A_1, \dots, A_n)$
- Primärschlüssel bestimmen über Kardinalitäten
  - 1:n Beziehung



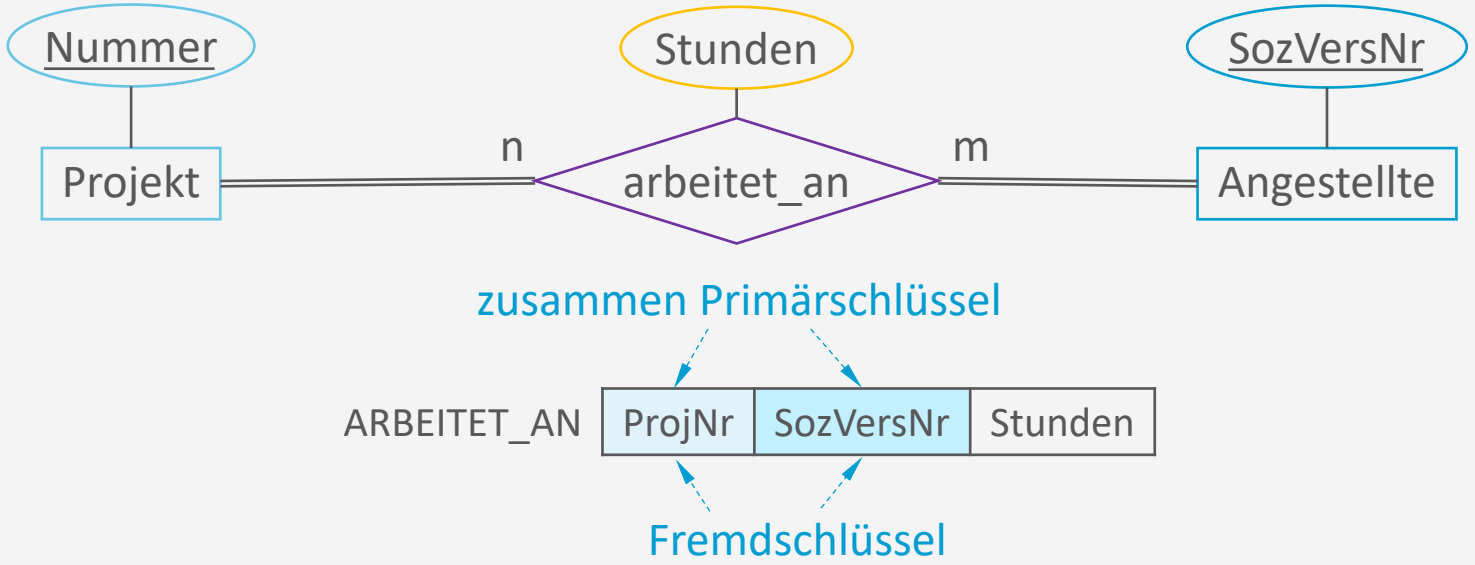
Bei 1:n Beziehungen kommt der Fremdschlüssel der Entität E_1 maximal einmal vor, kann also als Schlüssel verwendet werden.

KONTROLLIERT

ProjNr	AbtNr
1	42
2	42
3	21
4	54

Relationale Darstellung: Assoziation + Schlüssel

- Häufig abhängig von Fremdschlüsseln
  - Fremdschlüssel $P_{1,1}, \dots, P_{1,l}$ und $P_{2,1}, \dots, P_{2,m}$ in Relation $R(P_{1,1}, \dots, P_{1,l}, P_{2,1}, \dots, P_{2,m}, A_1, \dots, A_n)$
- Primärschlüssel bestimmen über Kardinalitäten
  - n:m Beziehung



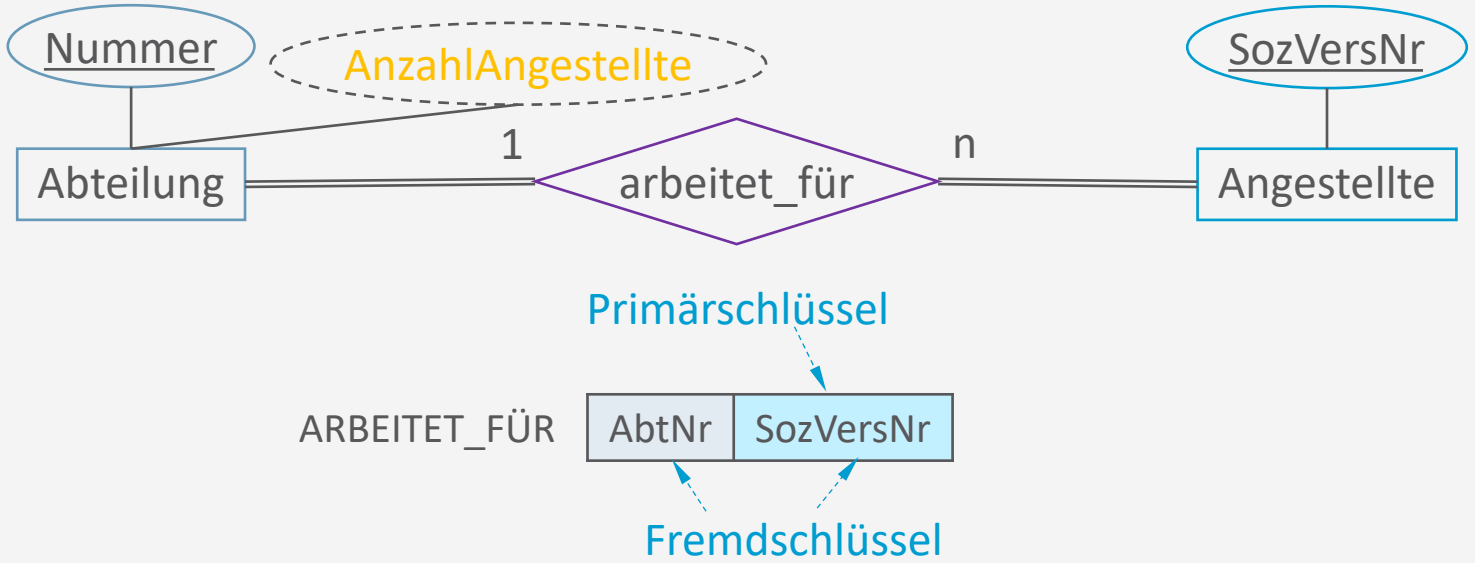
Bei n:m Beziehungen kommt die Kombination der Fremdschlüssel der Entitäten E_1, E_2 maximal einmal vor, können zusammen als Schlüssel verwendet werden.

ARBEITET_AN

ProjNr	SozVersNr	Stunden
1	A1	20
1	B2	10
2	A1	10
3	A1	10
3	C4	30

Relationale Darstellung: Abgeleitete Attribute

- Abgeleitete Attribute ergeben sich aus Zuständen anderer Teile
  - „AnzahlAngestellte“ ergibt sich durch „arbeitet_für“



Klassische Anfrage an DB (Wie? - nächster Block zu relationaler Algebra)

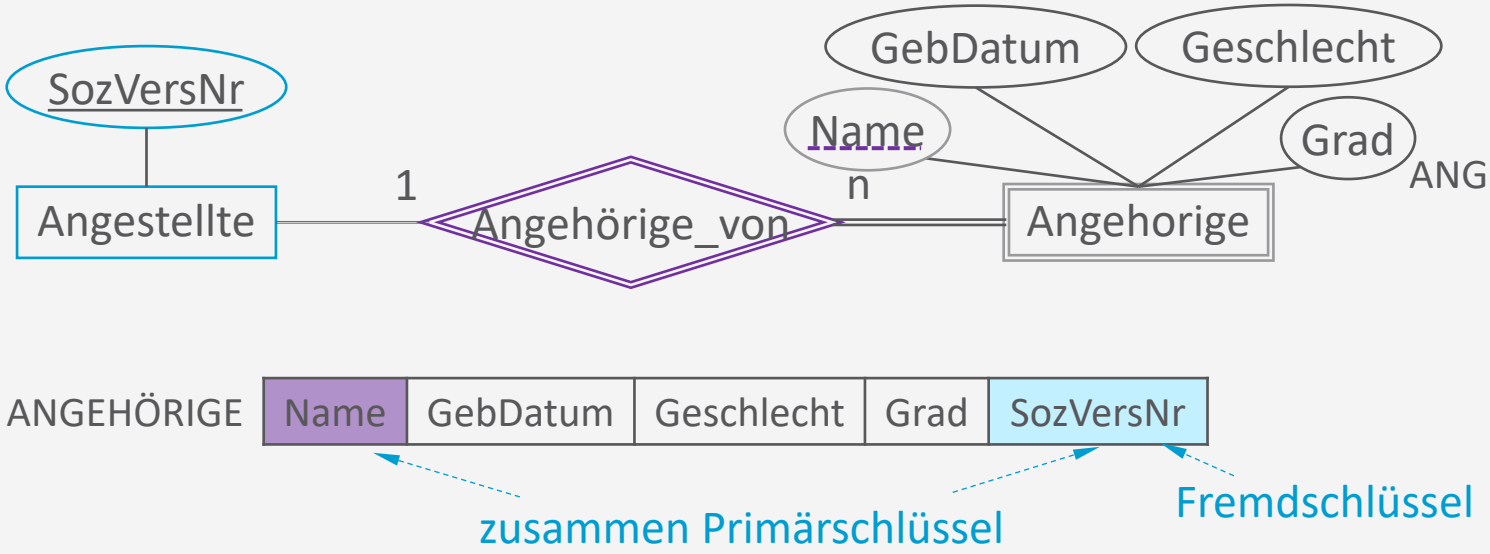
- Anzahl an Angestellten, die für eine Abteilung (z.B. 42) arbeiten (Ergebnis: 2)

ARBEITET_FÜR

AbtNr	SozVersNr
42	A1
42	B2
54	C4

Relationale Darstellung: Schwache Entitäten

- Schwache Entität E_1 mit partiellen Schlüsseln $P_{1,1}, \dots, P_{1,l}$
  - mit starker Entität E_2 mit Schlüsseln $P_{2,1}, \dots, P_{2,m}$
  - identifiziert über Beziehung R mit Attributen $A_1, \dots, A_n$
  - wird zu Relation $R(P_{1,1}, \dots, P_{1,l}, P_{2,1}, \dots, P_{2,m}, A_1, \dots, A_n)$



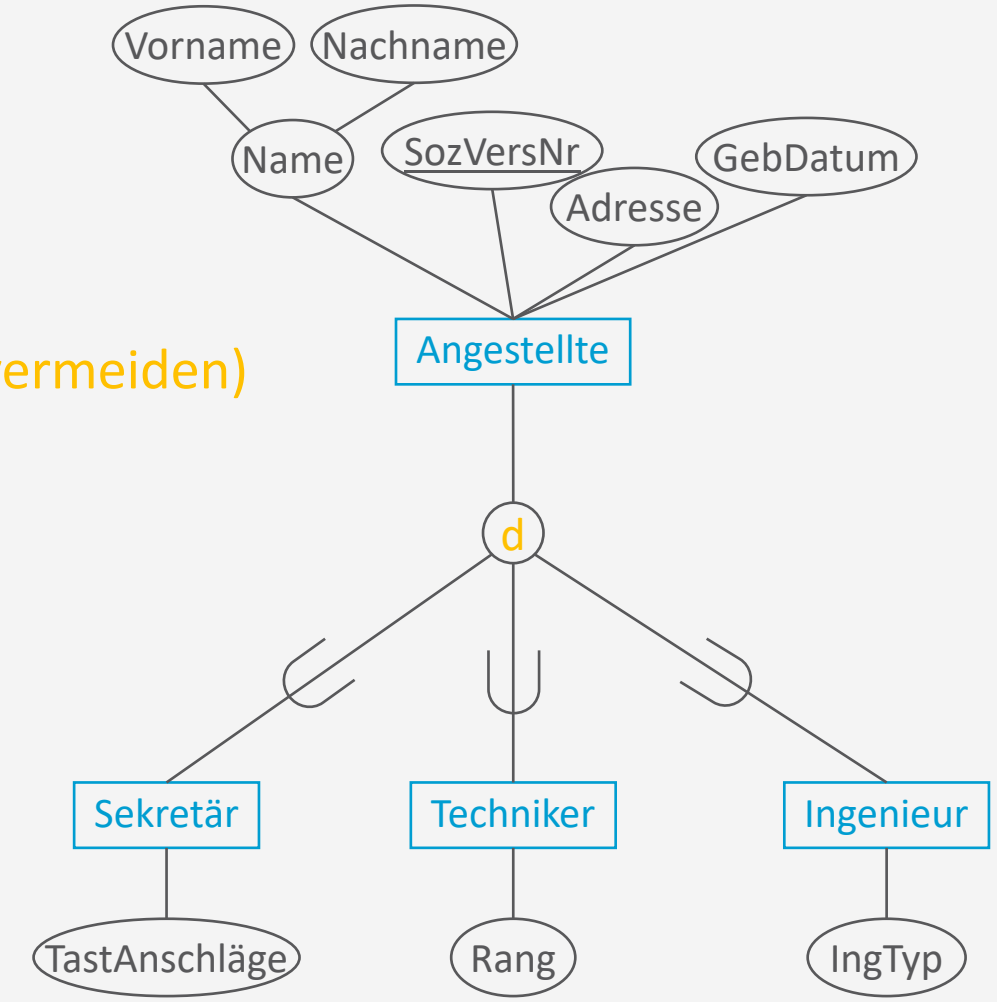
Partielle Schlüssel der schwachen Entität werden zusammen mit den Fremdschlüsseln der starken Entität zum Schlüssel.

ANGEHÖRIGE

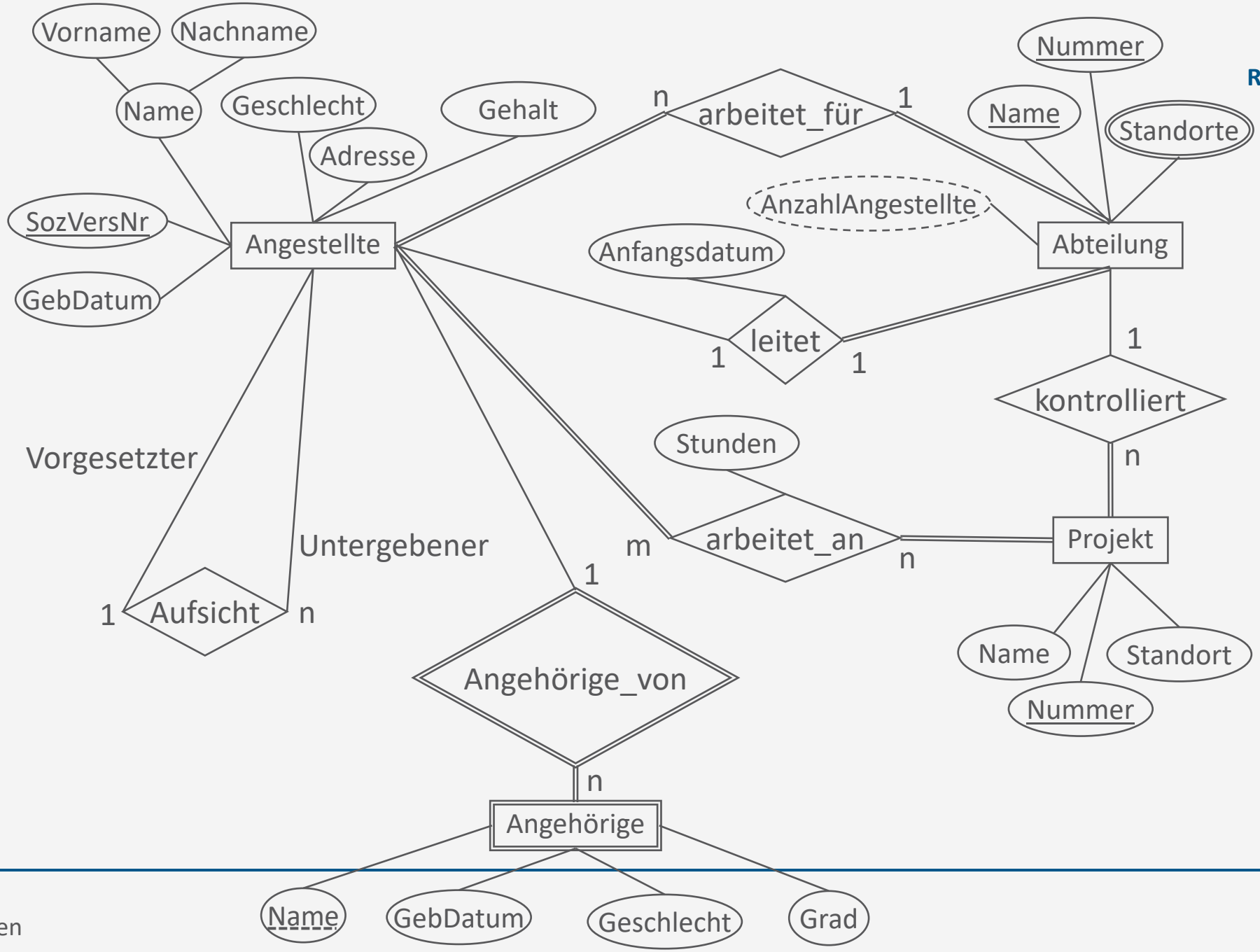
Name	Geb.	Geschl.	Grad	Soz.
Alice	...	...	...	A1
Eve	...	...	...	A1
Bob	...	...	...	A1
Charlie	...	...	...	B2
Judy	...	...	...	C4

Relationale Darstellung: EER-Modell

- EER-Konstrukte: Spezialisierung, Generalisierung
  - Subklasse wird zur Relation
  - Fremdschlüssel von Superklasse
  - Keine Vererbung der Attribute (doppelte Speicherung vermeiden)



ER-Diagramm


Vorname
Nachname
Name
Geschlecht
Gehalt
Adresse
SozVersNr
GebDatum
Angestellte
n
arbeitet_für
1
Nummer
Name
Standorte
AnzahlAngestellte
Abteilung
Anfangsdatum
leitet
1
1
1
kontrolliert
n
Stunden
Vorgesetzter
Untergebener
1
Aufsicht
n
m
arbeitet_an
n
Projekt
1
Angehörige_von
n
Name
Standort
Nummer
Angehörige
Name
GebDatum
Geschlecht
Grad

Entsprechendes relationales Schema

ANGESTELLTE	SozVersNr	Nachname	Vorname	Geschlecht	Adresse	Gehalt	GebDatum

AUFSICHT	Vorgesetzte	Untergebene

ABTEILUNG	Name	Nummer

ARBEITET_FÜR	AbtNr	SozVersNr

LEITET	SozVersNr	AbtNr	AnfDatum

ARBEITET_AN	ProjNr	SozVersNr	Stunden

PROJEKT	Nummer	Name	Standort

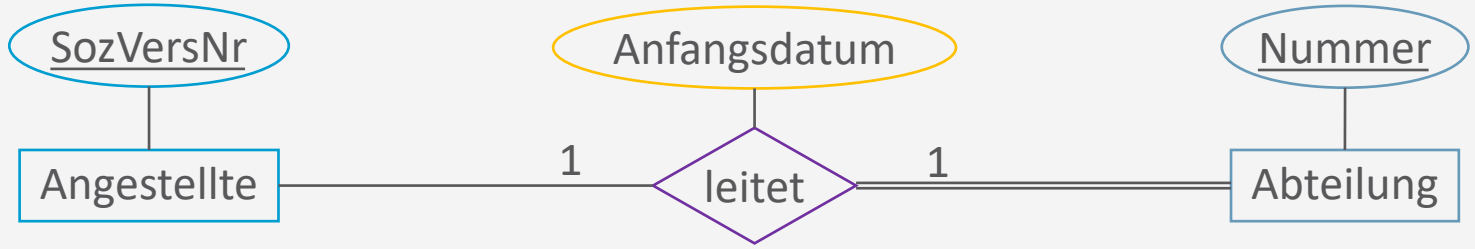
KONTROLLIERT	ProjNr	AbtNr

ANGEHÖRIGE	Name	GebDatum	Geschlecht	Grad	SozVersNr

Neun Relationen

Verfeinerung des relationalen Schemas

- Relationen mit gleichem Schlüssel kann man zusammenfassen
- Aber nur diese und keine anderen!
- Beispiel: Relation LEITET
  - SozVersNr und AbtNr als Schlüssel möglich, jeweils Schlüssel von Angestellte bzw. Abteilung
    1. Wenn „SozVersNr“ Schlüssel → LEITET zu ANGESTELLTE
    2. Wenn „AbtNr“ Schlüssel → LEITET zu ABTEILUNG
    - Da jede Abteilung eine Leitung hat, aber nicht jeder Angestellte leitet, ist 2. hier besser



ANGESTELLTE | SozVersNr | Nachname | Vorname | Geschlecht | Adresse | Gehalt | GebDatum

ABTEILUNG | Name | Nummer

LEITET | SozVersNr | AbtNr | AnfDatum

ABTEILUNG | Name | Nummer | Leitung | AnfDatum

Verfeinerung des relationalen Schemas

- Relationen mit gleichem Schlüssel kann man zusammenfassen
- Aber nur diese und keine anderen!
- Vor allem sinnvoll bei totaler Partizipation (auch der Fall auf vorheriger Folie)
  - Jede angestellte Person arbeitet für genau eine Abteilung

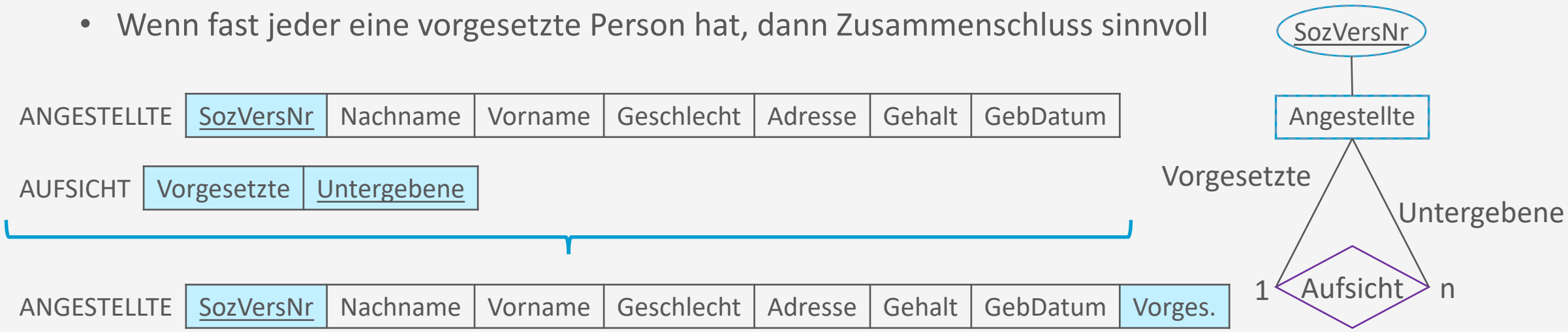
ANGESTELLTE | SozVersNr | Nachname | Vorname | Geschlecht | Adresse | Gehalt | GebDatum

ARBEITET_FÜR | AbtNr | SozVersNr

ANGESTELLTE | SozVersNr | Nachname | Vorname | Geschlecht | Adresse | Gehalt | GebDatum | AbtNr

Verfeinerung des relationalen Schemas

- Relationen mit gleichem Schlüssel kann man zusammenfassen
- Aber nur diese und keine anderen!
- Ohne totale Partizipation können NULL Werte erforderlich werden
  - Also Achtung: wenn viele Zellen mit NULL Werten nötig, Beziehung nicht so schnell ersichtlich
  - Beispiel:
    - Wenn fast jeder eine vorgesetzte Person hat, dann Zusammenschluss sinnvoll



Verfeinertes relationales Schema

ANGESTELLTE	SozVersNr	Nachname	Vorname	Geschlecht	Adresse	Gehalt	GebDatum	AbtNr	Vorgesetzte

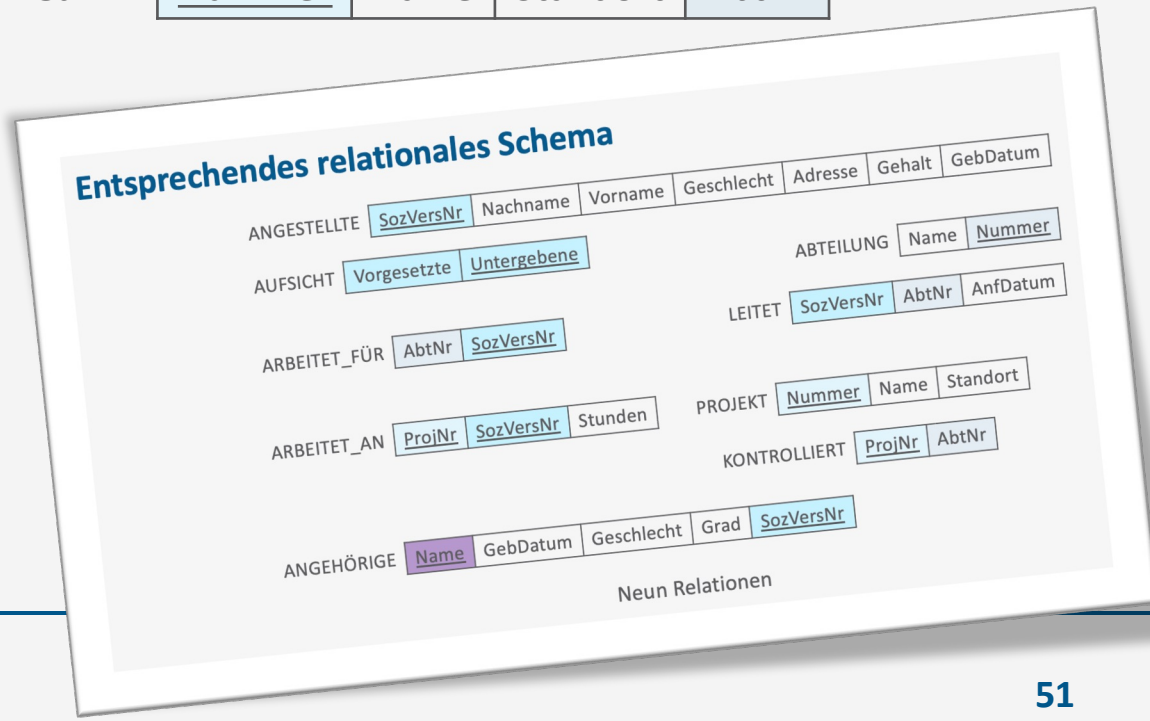
ABTEILUNG	Name	Nummer	Leitung	AnfDatum

ARBEITET_AN	ProjNr	SozVersNr	Stunden

PROJEKT	Nummer	Name	Standort	AbtNr

ANGEHÖRIGE	Name	GebDatum	Geschlecht	Grad	SozVersNr

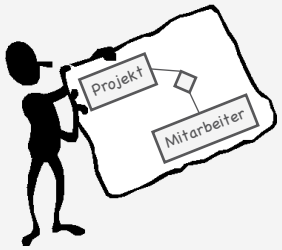
Fünf Relationen

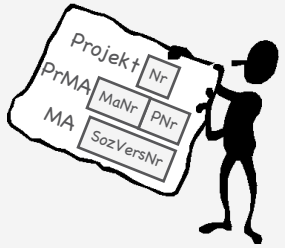


Entwurf relationaler Schemata

- Zwei alternative Methoden
  1. Entwickle zunächst ein ER-Diagramm, leite daraus ein relationales Schema mit Entitäten- und Beziehungstabellen ab
     - (Vgl. C. Batini, S. Ceri, S.B. Navathe. Conceptual Database Design - An Entity Relationship Approach, Benjamin/Cummings, Redwood City, Kalifornien, 1992)
  2. Sammle so genannte funktionale Abhängigkeiten aus der Anforderungsdefinition und erzeuge daraus ein relationales Schema in Normalform
     - (Im Trend 1970...80).
     - Ausführlich in der Literatur beschrieben (Vgl. S.M. Lang, P.C. Lockemann. Datenbankeinsatz. Springer, Berlin u.a., 1995)

Zwischenzusammenfassung





- Entität mit Attributen wird zu Relationenschema
  - Entität als Tabelle mit Attributen als Spaltennamen
  - Schlüssel ausgehend von identifizierenden Attributen, aber nun minimaler Superschlüssel
  - Mehrwertige Attribute als eigene Relation (so auch schon im ER-Diagramm möglich)
  - Zusammengesetzte Attribute nicht direkt übersetzbar
- Assoziation mit Attributen wird zu Relationenschema mit Schlüsselattributen der verknüpften Entitäten als zusätzliche Attribute
  - Schlüssel gemäß Kardinalitäten wählen
  - Abgeleitete Attribute: Anfrage
  - Schwache Entität: eigene Schlüssel plus Primärschlüssel der starken Entität
- Spezialisierung: eigenes Relationenschema mit Primärschlüsseln der spezialisierten Entität
- Verfeinerung durch Zusammenfassung von Relationen mit gleichem Schlüssel

Übersicht: 3. Das relationale Datenmodell

A. *Relationales Datenmodell*
- Relationen, Attribute, relationale Datenbanken und –schemata
- Schlüssel: Primärschlüssel, Fremdschlüssel, referentielle Integrität

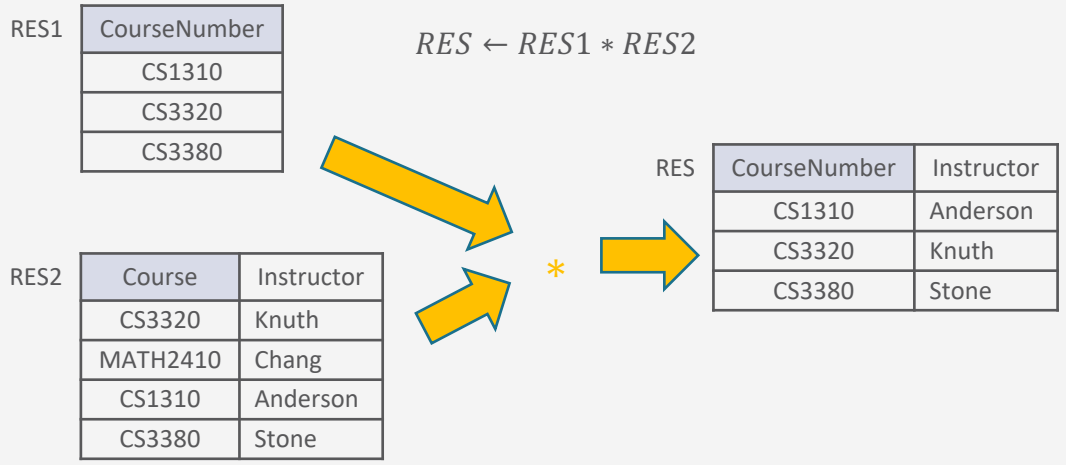
B. *Entwurf relationaler Schemata*
- Vom ER-Diagramm zum relationalen Modell

C. *Relationale Algebra*
- $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$
- Minimalität
- Aggregieren, gruppieren
- Einfügen, löschen, aktualisieren

WWU MÜNSTER

Das relationale Modell: Relationale Algebra

Datenbanken



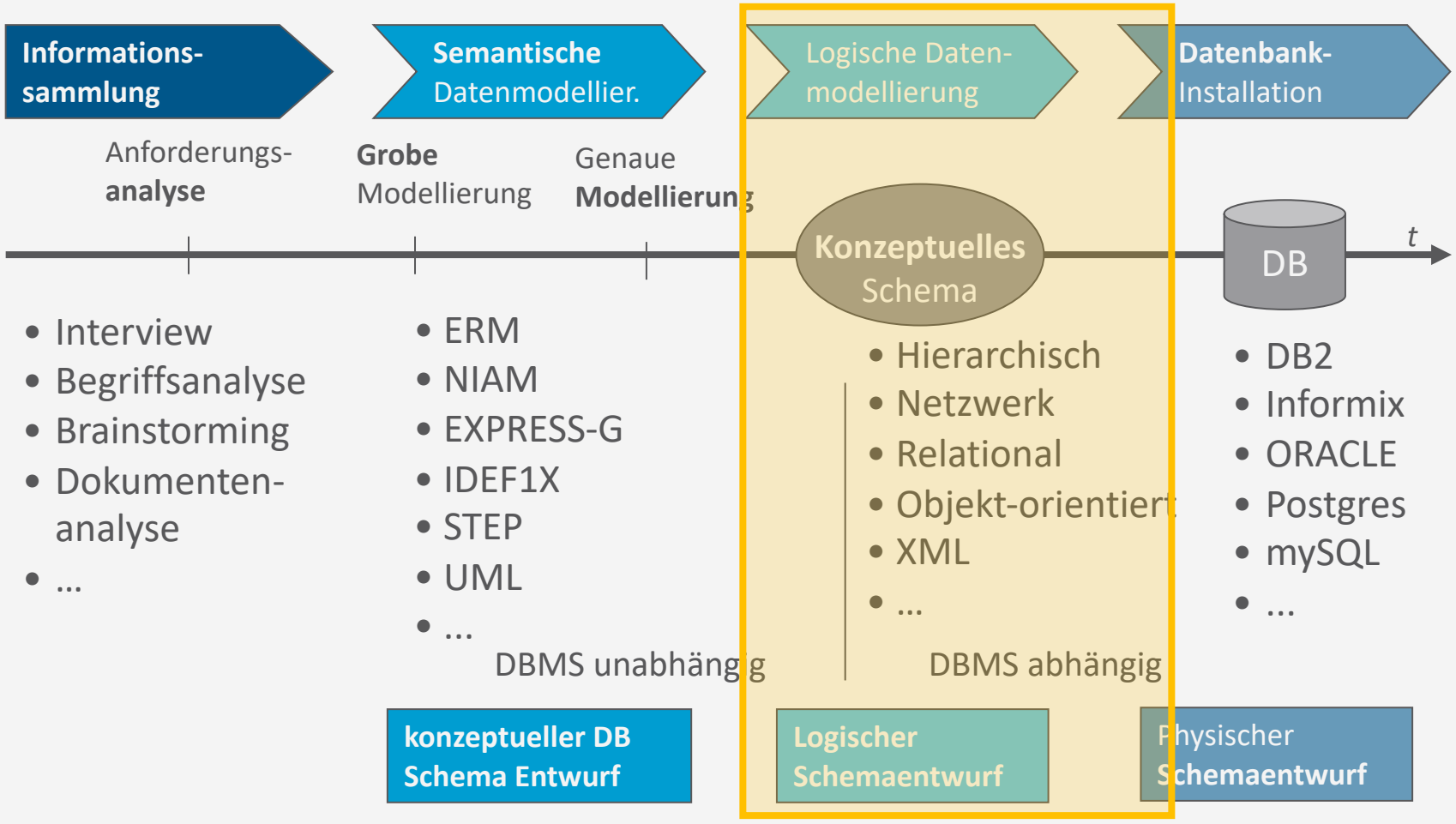
Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Übersicht: 3. Das Relationale Datenmodell

A. Relationales Datenmodell

- Relationen, Attribute, relationale Datenbanken und –schemata
- Schlüssel: Primärschlüssel, Fremdschlüssel, referentielle Integrität

B. Entwurf relationaler Schemata

- Vom ER-Diagramm zum relationalen Modell

C. Relationale Algebra

- $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$
- Minimalität
- Aggregieren, gruppieren
- Einfügen, löschen, aktualisieren

Relationenschemata und Relationen

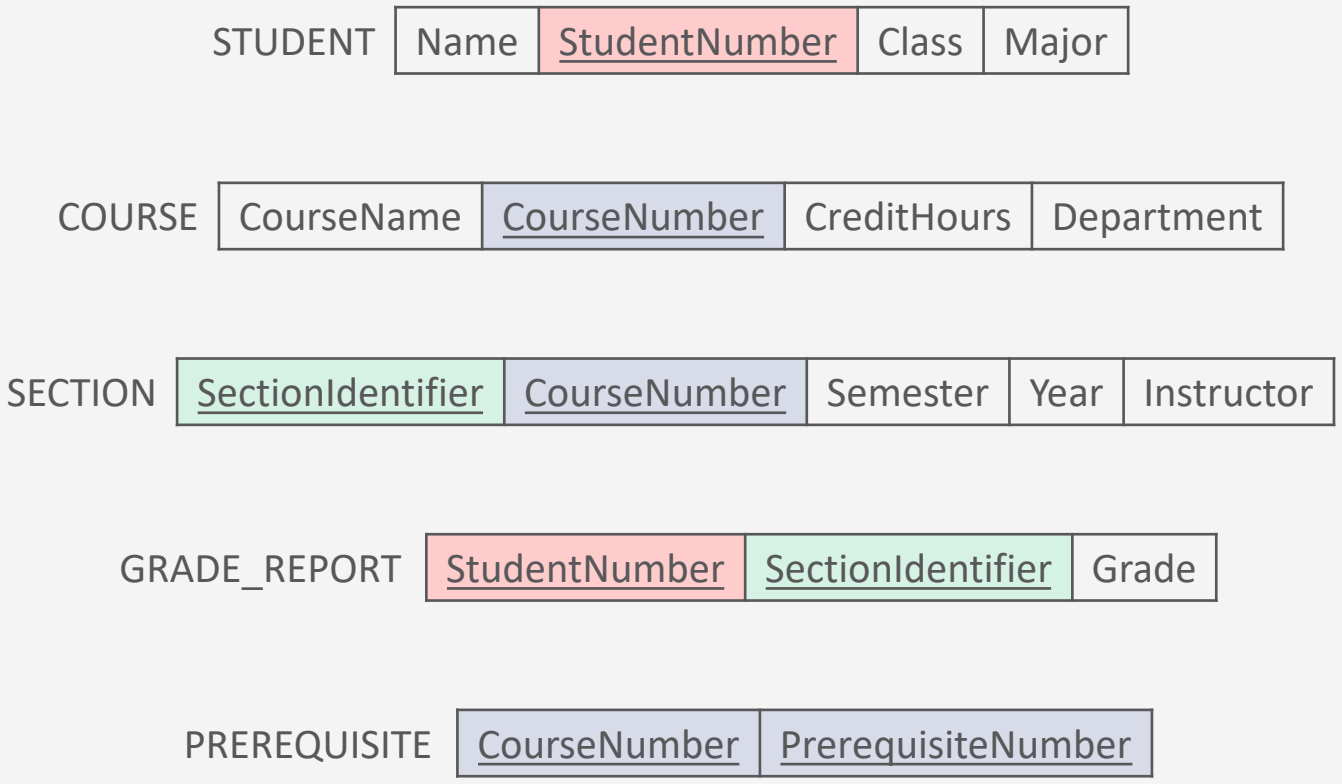
- $R(A_1, \ldots, A_n)$: Relationenschema n-ten Grades
- $t = \langle v_1, \ldots, v_n \rangle$: ein Tupel der Relation $r(R)$
  - v_i ist der Wert, der im Tupel t dem Attribut A_i entspricht.
- $R.A$: ein Attribut A des Relationenschemas R
- Für einzelne Komponentenwerte von einem Tupel t gilt:
  - $t[A_i]$ bzw. $t.A_i$ beziehen sich auf den Wert v_i in t für Attribut A_i
  - $t[A_u, \ldots, A_z]$ bzw. $t.(A_u, \ldots, A_z)$ beziehen sich auf Werte $\langle v_u, \ldots, v_z \rangle$ von Subtupeln von t, die den Attributen $A_u, \ldots, A_z$ von R entsprechen

Relation r — Relationenschema R, Grad 4

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Beispiel eines DB-Schemas



Beispiel eines DB-Zustands

Was für Fragen würde man stellen wollen?

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

STUDENT

Name	StudentNumber	Class	Major
Smith	17	1	CS
Brown	8	2	CS

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

GRADE_REPORT

StudentNumber	SectionIdentifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

CourseNumber	PrerequisiteNumber
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Anfragen an Relationen

- Entfernende Operatoren
  - Selektion σ
  - Projektion π
- Umbenennung ρ
- Klassische Mengenoperatoren (kombinieren Relationen)
  - Vereinigung $\cup$
  - Schnitt $\cap$
  - Differenz $-$
- Weitere kombinierende Operatoren
  - Kartesisches Produkt $\times$
  - Join $\bowtie$ und weitere Join-Arten
  - Outer Union
  - Division
- Aggregieren, gruppieren
  - Über die klassische relationale Algebra hinaus
- *Spotlight*: Relationenzustände ändern
  - Einfügen, löschen, aktualisieren

Selektion und Projektion

Entfernende Operatoren*

* Entfernen in dem Sinne, dass in einem Zwischenergebnis weniger Tupel oder Attribute vorkommen

Selektion σ

- Bildet eine Teilmenge von Tupeln einer Relation, die (jeweils) eine bestimmte Auswahlbedingung erfüllen:
  - $R' = \sigma_{\langle Auswahlbedingung \rangle}(R)$
  - Unär: wird auf genau eine Relation angewendet
  - Grad von R' = Grad von R
    - D.h., alle Attribute bleiben erhalten
  - Kardinalität wird möglicherweise kleiner
- Beispiele:
  - $\sigma_{Department=CS}(COURSE)$
  - $\sigma_{CreditHours \geq 4}(COURSE)$
  - $\sigma_{CreditHours \leq 3 \vee Deparment=MATH}(COURSE)$

Selektion = Auswahl
Selektion → sigma (σ)

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Selektion σ

- Kommutativ:

$$\sigma_{\langle Bedingung1\rangle}\left(\sigma_{\langle Bedingung2\rangle}(R)\right) = \sigma_{\langle Bedingung2\rangle}\left(\sigma_{\langle Bedingung1\rangle}(R)\right)$$

  - Beispiel:

$$\sigma_{CreditHours\leq 3}\left(\sigma_{Department=CS}(COURSE)\right) = \sigma_{Department=CS}\left(\sigma_{CreditHours\leq 3}(COURSE)\right)$$

3 Tupel nach erster σ — 2 Tupel nach erster σ

  - Aber: Reihenfolge hat eine Auswirkung auf Größe des Zwischenergebnisses
    - Zu nutze machen für effiziente Beantwortung (→ Kapitel 6: Anfragebeantwortung)

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Selektion σ

- Kaskade von σ = Und-Verknüpfung der Bedingungen:

$$\sigma_{\langle Bed1\rangle}\left(\sigma_{\langle Bed2\rangle}\left(\ldots\sigma_{\langle Bedn\rangle}(R)\right)\right) = \sigma_{\langle Bed1 \wedge Bed2 \wedge \cdots \wedge Bedn\rangle}(R)$$

  - Beispiel:

$$\sigma_{CreditHours \leq 3}\left(\sigma_{Department=CS}(COURSE)\right) = \sigma_{Department=CS \wedge CreditHours \leq 3}(COURSE)$$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Projektion π

- Wählt aus einer Relation bestimmte Attribute aus und verwirft die anderen:
  - $R' = \pi_{\langle Attributliste \rangle}(R)$
  - Unär: wird auf genau eine Relation angewendet
  - Grad R' ≤ Grad R
    - idR fehlen nach π Attribute
- Beispiele:
  - $\pi_{CourseName,CourseNumber}(COURSE)$
  - $\pi_{CourseName}(COURSE)$

Projektion = Abbildung
Projektion → pi (π)

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Projektion π

- Es gilt:

$$\text{Wenn } \langle Liste1 \rangle \subseteq \langle Liste2 \rangle: \pi_{\langle Liste1 \rangle}\left(\pi_{\langle Liste2 \rangle}(R)\right) = \pi_{\langle Liste1 \rangle}(R)$$

  - Beispiel:

$$\pi_{CourseName}\left(\pi_{CourseName,CourseNumber}(COURSE)\right) = \pi_{CourseName}(COURSE)$$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Projektion π

- Anzahl der Tupel (Kardinalität) kann sich verringern
  - Mengeneigenschaft entfernt Duplikate
- Beispiele:
  - $\pi_{CreditHours,Department}(COURSE)$
  - Projektionen der vorherigen Folien
    - $\pi_{CourseName,CourseNumber}(COURSE)$
    - $\pi_{CourseName}(COURSE)$
    - Keine doppelten Einträge
    - CourseName, CourseNumber eindeutig (Primärschlüssel bzw. Schlüsselkandidaten)

CreditHours	Department
4	CS
3	MATH
3	CS

CreditHours	Department
4	CS
4	CS
3	MATH
3	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Operationssequenzen und Renaming

Relationale Algebra

Sequenzen von Operationen

- Im Allgemeinen werden mehrere Operationen nacheinander ausgeführt
  - Einzelner Ausdruck oder Sequenz mit explizit benanntem Zwischenergebnis
- Beispiel
  - $\pi_{CreditHours,Department}(\sigma_{CreditHours\geq 4}(COURSE))$
  - $MIN4 \leftarrow \sigma_{CreditHours\geq 4}(COURSE)$
    $RESULTAT \leftarrow \pi_{CreditHours,Department}(MIN4)$
- Relationen umbenennen möglich:
  - $MIN4 \leftarrow \sigma_{CreditHours\geq 4}(COURSE)$
    $COURSE4(Hours, Department)$
    $\leftarrow \pi_{CreditHours,Department}(MIN4)$

COURSE4

Hours	Department
4	CS

CreditHours	Department
4	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Umbenennung ρ

- Erlaubt die explizite Umbenennung von Relationen und Attributen
- Gegeben Ausgangsrelation $R(A_1, \ldots, A_n)$:
- Umbenennung von R in S und $A_1, \ldots, A_n$ in $B_1, \ldots, B_n$
  - $\rho_{S(B_1,\ldots,B_n)}(R)$
    - Verändert nicht Tupel, sondern Schemata
    - Unär: wird auf ein Schema angewendet
    - Häufig Hilfsoperation in Operationssequenzen
    - Auch bekannt als RENAME
  - Beispiel:
    - $\rho_{KURS(KursName,KursNr,SWS,Institut)}(COURSE)$

KURS

KursName	KursNr	SWS	Institut
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Vereinigung, Schnitt, Differenz

Mengenoperatoren

Mengenoperationen auf Relationen

- Vereinigung:
  - $R \cup S$ enthält alle Tupel, die in R, in S oder in beiden Relationen auftauchen
  - Duplikat-Tupel werden eliminiert
- Schnitt:
  - $R \cap S$ enthält nur Tupel, die in R und in S auftauchen
- Differenz:
  - $R - S$ enthält alle Tupel, die in R, jedoch nicht in S enthalten sind
- Auch bekannt als UNION, INTERSECTION, DIFFERENCE
- Binär: werden auf zwei Relationen angewendet
  - R und S können dieselbe Relation sein, d.h., $R = S$
    - Dann auch durch UND, ODER, NICHT Verknüpfungen darstellbar

Vereinigung: Beispiel

- $RESULT1 \leftarrow \pi_{CreditHours,Deparment}\left(\sigma_{Department=MATH}(COURSE)\right)$
- $RESULT2 \leftarrow \pi_{CreditHours,Department}\left(\sigma_{CreditHours\leq 3}(COURSE)\right)$
- $RESULT \leftarrow RESULT2 \cup RESULT1$

RESULT

CreditHours	Department
3	MATH
3	CS

U

RESULT2

CreditHours	Department
3	MATH
3	CS

RESULT1

CreditHours	Department
3	MATH

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Schnitt: Beispiel

- $RESULT1 \leftarrow \pi_{CreditHours,Deparment}\left(\sigma_{Department=MATH}(COURSE)\right)$
- $RESULT2 \leftarrow \pi_{CreditHours,Department}\left(\sigma_{CreditHours\leq 3}(COURSE)\right)$
- $RESULT \leftarrow RESULT2 \cap RESULT1$

RESULT

CreditHours	Department
3	MATH

RESULT2

CreditHours	Department
3	MATH
3	CS

∩

RESULT1

CreditHours	Department
3	MATH

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Differenz: Beispiel

- $RESULT1 \leftarrow \pi_{CreditHours,Deparment}\left(\sigma_{Department=MATH}(COURSE)\right)$
- $RESULT2 \leftarrow \pi_{CreditHours,Department}\left(\sigma_{CreditHours\leq 3}(COURSE)\right)$
- $RESULT \leftarrow RESULT2 - RESULT1$

$RESULT \leftarrow RESULT1 - RESULT2?$

RESULT

CreditHours	Department
3	CS

RESULT2

CreditHours	Department
3	MATH
3	CS

–

RESULT1

CreditHours	Department
3	MATH

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Mengenoperatoren: Eigenschaften

- Können nur auf UNION-kompatible Relationen angewendet werden:
  - R und S haben gleichen Grad n
  - Attribute haben gleiche Wertebereiche
    - $\mathrm{dom}(A_i) = \mathrm{dom}(B_i)$ für alle $1 \leq i \leq n$
  - Attribute müssen aber nicht gleich heißen → Umbenennung ρ
    - Konvention: Namen aus der ersten Relation R

- Vereinigung und Schnitt sind kommutativ

$$R \cup S = S \cup R$$
$$R \cap S = S \cap R$$

- Vereinigung und Schnitt sind assoziativ

$$(R \cup S) \cup T = S \cup (R \cup T)$$
$$(R \cap S) \cap T = S \cap (R \cap T)$$

- Differenz?
  - Im Allgemeinen nicht kommutativ:

$$R - S \neq S - R$$

  - Im Allgemeinen nicht assoziativ:

$$(R - S) - T \neq S - (R - T)$$

Kartesisches Produkt, Join, Outer Union, Division

Kombinierende Operatoren
Relationale Algebra

Kartesisches Produkt

- Alle Tupel zweier Relationen R und S werden kombinatorisch („vollständig“) miteinander verbunden
  - Gegeben $R(A_1, \dots, A_n)$ und $S(B_1, \dots, B_m)$
  - $R \times S = Q(A_1, \dots, A_n, B_1, \dots, B_m)$
  - Binär: wird auf zwei Relationen angewendet
  - R und S müssen nicht UNION-kompatibel sein
  - Um eindeutige Attributbezeichnungen in der Ergebnisrelation zu gewährleisten, müssen Attribute, die in R und S gleich bezeichnet sind, vorher umbenannt werden
- Resultat
  - Grad: R hat n Spalten, S hat m Spalten $\rightarrow$ $R \times S$ hat $(n + m)$ Spalten
  - Kardinalität: R hat k Zeilen, S hat l Zeilen $\rightarrow$ $R \times S$ hat $(k \cdot l)$ Zeilen

Kartesisches Produkt: Beispiel

- $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
- $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
- $CROSS \leftarrow RES1 \times RES2$
- $RESULT \leftarrow \pi_{CourseNumber,Instructor}\left(\sigma_{CourseNumber=Course}(CROSS)\right)$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

RES1

CourseNumber
CS1310
CS3320
CS3380

Kartesisches Produkt: Beispiel

- $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
- $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
- $CROSS \leftarrow RES1 \times RES2$
- $RESULT \leftarrow \pi_{CourseNumber,Instructor}\left(\sigma_{CourseNumber=Course}(CROSS)\right)$

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

Kartesisches Produkt: Beispiel

- $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
- $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
- $CROSS \leftarrow RES1 \times RES2$
- $RESULT \leftarrow \pi_{CourseNumber,}$

RES1

CourseNumber
CS1310
CS3320
CS3380

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

CROSS

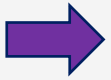
CourseNumber	Course	Instructor
CS1310	CS3320	Knuth
CS1310	MATH2410	Chang
CS1310	CS1310	Anderson
CS1310	CS3380	Stone
CS3320	CS3320	Knuth
CS3320	MATH2410	Chang
CS3320	CS1310	Anderson
CS3320	CS3380	Stone
CS3380	CS3320	Knuth
CS3380	MATH2410	Chang
CS3380	CS1310	Anderson
CS3380	CS3380	Stone

Kartesisches Produkt: Beispiel

- …
- $CROSS \leftarrow RES1 \times RES2$
- $RESULT \leftarrow \pi_{CourseNumber,Instructor}(\sigma_{CourseNumber=Course}(CROSS))$

CROSS

CourseNumber	Course	Instructor
CS1310	CS3320	Knuth
CS1310	MATH2410	Chang
CS1310	CS1310	Anderson
CS1310	CS3380	Stone
CS3320	CS3320	Knuth
CS3320	MATH2410	Chang
CS3320	CS1310	Anderson
CS3320	CS3380	Stone
CS3380	CS3320	Knuth
CS3380	MATH2410	Chang
CS3380	CS1310	Anderson
CS3380	CS3380	Stone



Geht das auch einfacher?

RES2

CourseNumber	Instructor
CS1310	Anderson
CS3320	Knuth
CS3380	Stone

Join ⋈: Intuition

- $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
- $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
- Die Tupel miteinander verbinden, wo
  $RES1.CourseNumber = RES2.Course$

RES1

CourseNumber
CS1310
CS3320
CS3380

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

CROSS'

CourseNumber	Course	Instructor
CS1310	CS1310	Anderson
CS3320	CS3320	Knuth
CS3380	CS3380	Stone

$RES1 \bowtie_{CourseNumber=Course} RES2$

CROSS

CourseNumber	Course	Instructor
~~CS1310~~	~~CS3320~~	~~Knuth~~
~~CS1310~~	~~MATH2410~~	~~Chang~~
CS1310	CS1310	Anderson
~~CS1310~~	~~CS3380~~	~~Stone~~
CS3320	CS3320	Knuth
~~CS3320~~	~~MATH2410~~	~~Chang~~
~~CS3320~~	~~CS1310~~	~~Anderson~~
~~CS3320~~	~~CS3380~~	~~Stone~~
~~CS3380~~	~~CS3320~~	~~Knuth~~
~~CS3380~~	~~MATH2410~~	~~Chang~~
~~CS3380~~	~~CS1310~~	~~Anderson~~
CS3380	CS3380	Stone

Join ⋈

- Verbindet die Tupel zweier Relationen, die die Join-Bedingung erfüllen
  - Gegeben $R(A_1, \dots, A_n)$ und $S(B_1, \dots, B_m)$
  - $R \bowtie_{\langle Bedingung \rangle} S = Q(A_1, \dots, A_n, B_1, \dots, B_m)$
  - Q enthält alle Kombinationen von Tupeln, die der Bedingung entsprechen
  - Äquivalent zu kartesischem Produkt mit anschließender Selektion
    - $R \bowtie_{\langle Bedingung \rangle} S = \sigma_{\langle Bedingung \rangle}(R \times S)$
    - Zwischenergebnis kleiner bei Join
  - Binär: werden auf zwei Relationen angewendet

Join ⋈: Beispiel

- $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
- $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
- $JOINRES \leftarrow RES1 \bowtie_{CourseNumber=Course} RES2$
- $RES \leftarrow \pi_{CourseNumber,Instructor}(JOINRES)$

RES1

CourseNumber
CS1310
CS3320
CS3380

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

JOINRES

CourseNumber	Course	Instructor
CS1310	CS1310	Anderson
CS3320	CS3320	Knuth
CS3380	CS3380	Stone

RES

CourseNumber	Instructor
CS1310	Anderson
CS3320	Knuth
CS3380	Stone

Join-Arten

- Theta-Join
  - Jede (Teil-)Bedingung $A_i\ \theta\ B_j$ der Join-Bedingung
  - basiert auf einem θ aus $\{=, <, \leq, \geq, >, \neq\}$
- Equi-Join (Spezialfall)
  - θ ist $\{=\}$: es gibt nur eine Join-Bedingung, und sie prüft auf Gleichheit
  - Beispiel von vorher: Equi-Join
    - $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
    - $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
    - $JOINRES \leftarrow RES1 \bowtie_{CourseNumber=Course} RES2$
    - $RES \leftarrow \pi_{CourseNumber,Instructor}(JOINRES)$

Join-Arten (Forts.)

- Natural Join ($*$)
  - Join-Bedingung muss nicht angegeben werden: entspricht einem Equi-Join mit mehreren Attributen, die in beiden Relationen gleich heißen (→RENAME)
  - Doppelte Spalten werden entfernt
  - Beispiel von vorher als Natural Join
    - $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
    - $RES2 \leftarrow \rho_{Course,Instructor}\left(\pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)\right)$
    - $JOINRES \leftarrow RES1 * RES2$
    - $RES \leftarrow \pi_{CourseNumber,Instructor}(JOINRES)$

Join-Arten (Forts.)

- Beispiel von vorher als Natural Join
  - $RES1 \leftarrow \pi_{CourseNumber}\left(\sigma_{Department=CS}(COURSE)\right)$
  - $RES2 \leftarrow \pi_{CourseNumber,Instructor}\left(\sigma_{Year=19}(SECTION)\right)$
  - $JOINRES \leftarrow RES1 * RES2$

RES1

CourseNumber
CS1310
CS3320
CS3380

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

*

RES

CourseNumber	Instructor
CS1310	Anderson
CS3320	Knuth
CS3380	Stone

Join-Arten (Forts.)

- Left-/Right-/Full-Outer-Join
  ($R \bowtie S$, $R \bowtie S$, $R \bowtie S$)
  - Tupel ohne Join-Partner kommen trotzdem ins Ergebnis
    - Fehlende Werte werden mit NULL aufgefüllt
  - Left Outer Join: alle Tupel von R
  - Right Outer Join: alle Tupel von S
  - Full Outer Join: alle Tupel von R, S
- Beispiele:
  - $LEFT \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$
  - $RIGHT \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$
  - $FULL \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$

RES'

CourseNumber
CS1310
CS3320
CS3380
CS3390

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

Join-Arten (Forts.)

- $LEFT \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$
- $RIGHT \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$
- $FULL \leftarrow RES' \bowtie_{CourseNumber=Course} RES2$

RES'

CourseNumber
CS1310
CS3320
CS3380
CS3390

RES2

Course	Instructor
CS3320	Knuth
MATH2410	Chang
CS1310	Anderson
CS3380	Stone

LEFT

CourseNumber	Course	Instructor
CS1310	CS1310	Anderson
CS3320	CS3320	Knuth
CS3380	CS3380	Stone
CS3390	NULL	NULL

RIGHT

CourseNumber	Course	Instructor
CS1310	CS1310	Anderson
CS3320	CS3320	Knuth
CS3380	CS3380	Stone
NULL	MATH2410	Chang

FULL

CourseNumber	Course	Instructor
CS1310	CS1310	Anderson
CS3320	CS3320	Knuth
CS3380	CS3380	Stone
CS3390	NULL	NULL
NULL	MATH2410	Chang

Outer Union $\cup\!\!\!\!\!+$

- Vereinigung von Tupeln, deren Relationen nicht UNION-kompatibel bzw. nur partiell UNION-kompatibel sind
  - Gegeben $R(A_1, \dots, A_n)$ und $S(B_1, \dots, B_m)$
  - $R \cup S = Q(C_1, \dots, C_k)$
    - $C_1, \dots, C_k$ beinhaltet die kompatiblen Attribute sowie die verbliebenen Attribute in R und S
    - Kompatible Attribute müssen nicht gleich heißen → Umbenennung ρ
      - Konvention: Namen aus der ersten Relation R
    - Binär: wird auf zwei Relationen angewendet
    - NULL-Werte für Datenfelder, die dadurch für ein Tupel neu entstehen

Outer Union $\cup$

- Beispiel:
  - $ALL{-}COURSES \leftarrow COURSE \cup \rho_{CourseName,CreditHours,Department,Module}(SEMINAR)$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

SEMINAR

SeminarName	CreditHours	Department	Module
Sorting	2	CS	A1
Indexes	3	CS	A2
Hashing	2	CS	A1

ALL-COURSES

CourseName	CourseNumber	CreditHours	Department	Module

Outer Union $\cup\!\!\!\!-$

- Beispiel:
  - $ALL{-}COURSES \leftarrow COURSE \cup \rho_{CourseName,CreditHours,Department,Module}(SEMINAR)$

ALL_COURSES

CourseName	CourseNumber	CreditHours	Department	Module
Introduction to CS	CS1310	4	CS	NULL
Data Structures	CS3320	4	CS	NULL
Discrete Mathematics	MATH2410	3	MATH	NULL
Databases	CS3380	3	CS	NULL
Sorting	NULL	2	CS	A1
Indexes	NULL	3	CS	A2
Hashing	NULL	2	CS	A1

Division ÷

- $T(Y) = R(Z) \div S(X)$
  - Geht nur, wenn gilt: Attributmengen $X \subseteq Z$
  - Binär: auf zwei Relationen angewendet
  - Nicht sehr intuitiv → selten verwendet
- Sei $Y = Z - X$
- $T(Y)$ enthält ein Tupel t, wenn für jedes Tupel t_S in S ein Tupel t_R in R existiert, so dass gilt:

$$t_R[Y] = t \text{ und } t_R[X] = t_S$$

  - Jedes Ergebnistupel t muss mit jedem Tupel t_S aus S ein Tupel t_R in R erzeugen

- Beispiel
  - $R(A, B), Z = \{A, B\}, S(A), X = \{A\}$
  - $Y = \{B\} \rightarrow T(B)$

R	A	B
	a1	b1
	a2	b1
	a3	b1
	a4	b1
	a1	b2
	a3	b2
	a2	b3
	a3	b3
	a4	b3
	a1	b4
	a2	b4
	a3	b4

S	A
	a1
	a2
	a3

T	B
	b1
	b4

„Sammle die B's ein, die in R mit allen A's auftreten, die in S vorkommen (a1, a2, a3)."

- b1: taucht mit a1, a2, a3 auf: ✓
- b2: taucht mit a1, a3 auf: ↘
- b3: taucht mit a2, a3 auf: ↘
- b4: taucht mit a1, a2, a3 auf: ✓

a4 irrelevant, da nicht in S.

Beispiel

- Ermittle alle Namen von Studenten, die in allen Kursen, die das Department CS anbietet, im Jahr 19 Prüfungen abgelegt haben
  1. $CS{-}COURSE \leftarrow \sigma_{Department=CS}(COURSE)$
  2. $19{-}SECTION \leftarrow \sigma_{Year=19}(SECTION)$
  3. $CS19{-}SECTION \leftarrow \pi_{SectionIdentifier}(19{-}SECTION * CS{-}COURSE)$
  4. $STN{-}SID \leftarrow \pi_{StudentNumber,SectionIdentifier}(GRADE{-}REPORT)$
  5. $CS19{-}STUDENT(StudentNumber) \leftarrow STN{-}SID \div CS19{-}SECTION$
  6. $RESULTAT \leftarrow \pi_{Name}(CS19{-}STUDENT * STUDENT)$

Beispiel

1. $CS{-}COURSE \leftarrow \sigma_{Department=CS}(COURSE)$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

CS-COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Databases	CS3380	3	CS

Beispiel

2. $19\text{–}SECTION \leftarrow \sigma_{Year=19}(SECTION)$

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

19-SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

Beispiel

1. $CS{-}COURSE \leftarrow \sigma_{Department=CS}(COURSE)$
2. $19{-}SECTION \leftarrow \sigma_{Year=19}(SECTION)$
3. $CS19{-}SECTION \leftarrow \pi_{SectionIdentifier}(19{-}SECTION * CS{-}COURSE)$

CS-COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Databases	CS3380	3	CS

19-SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
102	CS3320	Spring	19	Knuth
112	MATH2410	Fall	19	Chang
119	CS1310	Fall	19	Anderson
135	CS3380	Fall	19	Stone

CS19-SECTION

SectionIdentifier
102
119
135

Beispiel

4. $STN{-}SID \leftarrow \pi_{StudentNumber,SectionIdentifier}(GRADE{-}REPORT)$

GRADE-REPORT

StudentNumber	SectionIdentifier	Grade
17	112	B
17	119	C
8	85	A
8	119	A
8	102	B
8	135	A

STN-SID

StudentNumber	SectionIdentifier
17	112
17	119
8	85
8	119
8	102
8	135

Beispiel

3. $CS19{-}SECTION \leftarrow \pi_{SectionIdentifier}(19{-}SECTION * CS{-}COURSE)$
4. $STN{-}SID \leftarrow \pi_{StudentNumber,SectionIdentifier}(GRADE{-}REPORT)$
5. $CS19{-}STUDENT(StudentNumber) \leftarrow STN{-}SID \div CS19{-}SECTION$

STN-SID

StudentNumber	SectionIdentifier
17	112
17	119
8	85
8	119
8	102
8	135

CS19-SECTION

SectionIdentifier
102
119
135

CS19-STUDENT

StudentNumber
8

Beispiel

- Ermittle alle Namen von Studenten, die in allen Kursen, die das Department CS anbietet, im Jahr 19 Prüfungen abgelegt haben
  1. $CS{-}COURSE \leftarrow \sigma_{Department=CS}(COURSE)$
  2. $19{-}SECTION \leftarrow \sigma_{Year=19}(SECTION)$
  3. $CS19{-}SECTION \leftarrow \pi_{SectionIdentifier}(19{-}SECTION * CS{-}COURSE)$
  4. $STN{-}SID \leftarrow \pi_{StudentNumber,SectionIdentifier}(GRADE{-}REPORT)$
  5. $CS19{-}STUDENT(StudentNumber) \leftarrow STN{-}SID \div CS19{-}SECTION$
  6. $RESULTAT \leftarrow \pi_{Name}(CS19{-}STUDENT * STUDENT)$

CS19-STUDENT

StudentNumber
8

STUDENT

Name	StudentNumber	Class	Major
Smith	17	1	CS
Brown	8	2	CS

RESULTAT

Name
Brown

Minimalität

Relationale Algebra

Minimalität der relationalen Algebra

- Minimale Operatormenge
  - Selektion (σ) und Projektion (π)
  - Umbenennung (ρ)
  - Vereinigung ($\cup$) und Differenz ($-$)
  - Kartesisches Produkt ($\times$)
- Weitere Operatoren durch minimale Operatormenge ausdrückbar
  - Schnitt ($\cap$): $A - (A - B)$
  - Join ($\bowtie$): $\sigma_{\langle \dots \rangle}(A \times B)$

Minimalität der relationalen Algebra

- Argumentation für die Minimalität – kein Beweis
  - Die Umbenennung (ρ) kann nicht durch eine der anderen fünf Operatoren ersetzt werden
  - ρ hat zudem keinen Einfluss auf die Darstellung eines der anderen Operatoren mithilfe der noch verbleibenden fünf Operatoren ($\sigma, \pi, \cup, -, \times$)
    Daher wird auf ρ im Folgenden nicht weiter eingegangen
  - Es verbleiben also die Operatoren $\sigma, \pi, \cup, -, \times$ bzgl. der Untersuchung, ob man welche davon ohne Verlust der Ausdrucksmöglichkeiten streichen kann

Untersuchung der verbliebenen Operatoren

- Kartesisches Produkt ($\times$):
  - Kann nicht simuliert werden, da $\{\sigma, \pi, \cup, -\}$ ein Schema nicht erweitern können
- Projektion (π):
  - Analoges Argument: keiner der Operatoren $\{\sigma, \cup, -, \times\}$ kann ein Schema reduzieren
- Selektion (σ):
  - Kann höchstens durch $-$ simuliert werden; Differenz testet jedoch nur auf Gleichheit ganzer Tupel und nicht auf beliebige Vergleiche durch Formeln, die sich auf Komponenten von Tupeln beziehen
- Differenz ($-$)
  - Kann die Selektion nicht simulieren, da σ die „Negation“ auf Relationen nicht darstellen kann.
- Vereinigung ($\cup$)
  - Kann nicht durch durch die Operatoren $\{\sigma, \pi, -, \times\}$ dargestellt werden.

Aggregatfunktion und Gruppierung

Relationale Algebra

Aggregatfunktionen

- Aggregiert mehrere Tupel zu einem Tupel bzgl. eines Attributes A einer Relation R
  - $\mathcal{F}_{\langle Liste\ von\ (Funktion, Attribut\ A)\ Paaren \rangle}(R)$
  - Abbildung in den Wertebereich von A
- Standard-Aggregationsfunkionen:
  - $\mathcal{F}_{\text{MIN } A}(R)$ Minimaler Wert, den A in $r(R)$ annimmt
  - $\mathcal{F}_{\text{MAX } A}(R)$ Maximaler Wert, den A in $r(R)$ annimmt
  - $\mathcal{F}_{\text{AVG } A}(R)$ Durchschnittlicher Wert von A über alle Tupel in $r(R)$
  - $\mathcal{F}_{\text{SUM } A}(R)$ Summe der Werte von A über alle Tupel in $r(R)$
  - $\mathcal{F}_{\text{COUNT } A}(R)$ Anzahl der Tupel, bei denen $A \neq$ NULL in $r(R)$
    - $\mathcal{F}_{\text{COUNT}*}(R)$ ohne Attribut $(*)$ = Kardinalität von R Häufig im Anschluss an eine Selektion
  - Setzt voraus, dass im Wertebereich des aggregierten Attributs eine Ordnung (bei MIN, MAX) bzw. Rechenoperationen (bei AVG, SUM) definiert sind

Aggregatfunktionen: Beispiele

- $RMIN(Min) \leftarrow \mathcal{F}_{\text{MIN } CreditHours}(COURSE)$
- $RMAX(Max) \leftarrow \mathcal{F}_{\text{MAX } CreditHours}(COURSE)$
- $RAVG(Avg) \leftarrow \mathcal{F}_{\text{AVG } CreditHours}(COURSE)$
- $RSUM(Sum) \leftarrow \mathcal{F}_{\text{SUM } CreditHours}(COURSE)$
- $RCNT(Cnt) \leftarrow \mathcal{F}_{\text{COUNT } CreditHours}(COURSE)$
  - $\mathcal{F}_{\text{COUNT}*}(COURSE)$?
    - Mit Projektion:
      Anzahl an Kursen vom Department CS
      $\mathcal{F}_{\text{COUNT}*}\left(\sigma_{Department=CS}(COURSE)\right)$
  - $\mathcal{F}_{\text{COUNT } Department}(COURSE)$?

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS
Algorithms	CS3390	4	NULL

RMIN

Min
3

RMAX

Max
4

RAVG

Avg
3.5

RSUM

Sum
14

RCNT

Cnt
4

Gruppierung

- Bildet Gruppen von Tupeln, die in einer Attributmenge die gleichen Werte haben
  - Häufig zur Vorbereitung einer Aggregation
  - Notation: Attributliste vor den Ausdruck
    - $\langle B_1, \dots, B_m \rangle \mathcal{F}_{\langle Liste\ von\ (Funktion, Attribut)\ Paaren \rangle}(R)$
  - Beispiel:
    - Bestimme die durchschnittliche Stundenzahl der Kurse der Departments

$$GAVG(Department, Cnt, Avg)$$
$$\leftarrow Department\ \mathcal{F}_{\text{COUNT } CourseNumber, \text{AVG } CreditHours}(COURSE)$$

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

GAVG

Department	Cnt	Avg
CS	3	3.7
MATH	1	3

Gruppierung: Unterschiede

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

- Beispiel:
  - $\mathcal{F}_{\text{COUNT } CourseNumber,\text{AVG } CreditHours}(COURSE)$
  - $Department\ \mathcal{F}_{\text{COUNT } CourseNumber,\text{AVG } CreditHours}(COURSE)$
  - $GAVG(Department, Cnt, Avg)$
    $\leftarrow Department\ \mathcal{F}_{\text{COUNT } CourseNumber,\text{AVG } CreditHours}(COURSE)$

COUNT_CourseNumbers	AVG_CreditHours
4	3.5

Department	COUNT_CourseNumbers	AVG_CreditHours
CS	3	3.7
MATH	1	3

GAVG

Department	KursAnzahl	StundenSchnitt
CS	3	3.7
MATH	1	3

Einfügen, Löschen, Aktualisieren

Änderung von Relationenzuständen

Mengenorientierte Spezifizierung von Änderungsoperatoren

- Änderungen am Relationenzustand über Mengenoperationen realisierbar
  - Gegeben R, S, T über die gleichen Attribute
    - S, T kann eine Menge von Tupeln sein oder ein komplexer relationaler Ausdruck, der in die gleichen Attribute endet
  - *Einfügen*: Vereinigung $R \leftarrow R \cup S$
  - *Löschen*: Differenz $R \leftarrow R - S$
  - *Aktualisieren*: $R \leftarrow R - S \cup T$ oder ein relationaler Ausdruck, der Tupel aktualisiert: $R \leftarrow \xi(R)$
- Änderungen
  - Einfügen, löschen: ganze Tupel betroffen
  - Aktualisieren: Werte einzelner Attribute ändern, Attribute hinzufügen (z.B. über Funktionen)
- Deklarative Spezifizierung auch mittels INSERT, DELETE, UPDATE bekannt

Einfügen von Tupeln

- **INSERT INTO** $R(A_1, \ldots, A_n)$ **VALUES** $\langle v_1, \ldots, v_n \rangle$ // oder: **INSERT INTO** R . . .
  - Eingabe:
    eine Liste von Attributwerten für ein neues Tupel $t = \langle v_1, \ldots, v_n \rangle$, das in die Relation r bzw. $r(R)$ eingefügt werden soll
  - Beispiel:
    - **INSERT INTO** COURSE
      VALUES <Algorithms, CS3390, 4, CS>
      - Primärschlüssel noch nicht vorhanden
      - Werte der Attribute liegen in den Domänen der Attribute
      - Kein zu prüfender Fremdschlüssel in Tupel

Was kann schief gehen?

Eigentlich sind Strings als solche zu markieren. Zur Übersicht lassen wir Anführungszeichen in dieser Vorlesung weg.

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS
Algorithms	CS3390	4	CS

Einfügen von Tupeln: Fehlersituationen

- **INSERT** t erzeugt Fehler
  → Führt zur Abweisung der INSERT Operation (Konsistenz bewahren!)
- Fehlersituationen
  - *Wertebereichseinschränkungen*: v_i entspricht nicht dem für A_i festgelegten Wertebereich
    - **INSERT** ... <Algorithms, CS3390, four, CS> bei z.B. dom(CreditHours) = Integer
  - *Schlüsseleinschränkungen*: Primärschlüsselwert in t existiert schon in r(R)
    - **INSERT** ... <Algorithms, CS3380, 4, CS>
  - *Entitätsintegrität*: Primärschlüssel / Teil des Primärschlüssels in t hat den Wert NULL
    - **INSERT** ... <Algorithms, NULL, 4, CS>

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Einfügen von Tupeln: Fehlersituationen

- **INSERT** t erzeugt Fehler
  → Führt zur Abweisung der INSERT Operation (Konsistenz bewahren!)
- Fehlersituationen (Forts.)
  - *Referenzielle Integrität*:
    Wert eines Fremdschlüssels in t referenziert ein Tupel s in einer Relation S, welches dort gar nicht existiert
    - **INSERT INTO** SECTION
      VALUES (142, CS3390, Fall, 19, Anderson)
      - Vorherige Fehler können also spätere Fehler nach sich ziehen

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor
85	MATH2410	Fall	18	King
92	CS1310	Fall	18	Anderson
102	CS3320	Spring	19	Knuth
142	CS3390	Fall	19	Anderson

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Löschen von Tupeln

- **DELETE FROM** R
  [**WHERE** <Bedingung>]
  - Löscht eine Menge von Tupeln $\{t_k\}_{k=1}^{K}$ aus einer Relation $r(R)$
  - Spezifiziert über Bedingungen
  - Beispiele:
    - Bestimmtes Tupel über Wert v des Primärschlüssels P referenziert (löscht ein Tupel)
      - **DELETE FROM** COURSE
        WHERE COURSE.CourseNumber = CS3380
    - Alle Tupel, bei denen z.B. ein Attribut A größer einem Wert w ist (löscht mehrere Tupel)
      - **DELETE FROM** COURSE
        WHERE COURSE.CreditHours >= 4

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

wird gelöscht

wird gelöscht

Löschen von Tupeln: Fehlersituation

- Fehlersituation: *Referenzielle Integrität*
  - In einer anderen Relation S gibt es einen Fremdschlüssel auf R und ein Tupel s in S referenziert das zu löschende Tupel t_k
    - **DELETE FROM** COURSE
      WHERE COURSE.CourseNumber=CS3380

Was können wir machen um die Konsistenz zu bewahren?

Fremdschlüssel

SECTION

SectionIdentifier	CourseNumber
85	MATH2410
92	CS1310
102	CS3320
112	MATH2410
119	CS1310
135	CS3380

Primärschlüssel

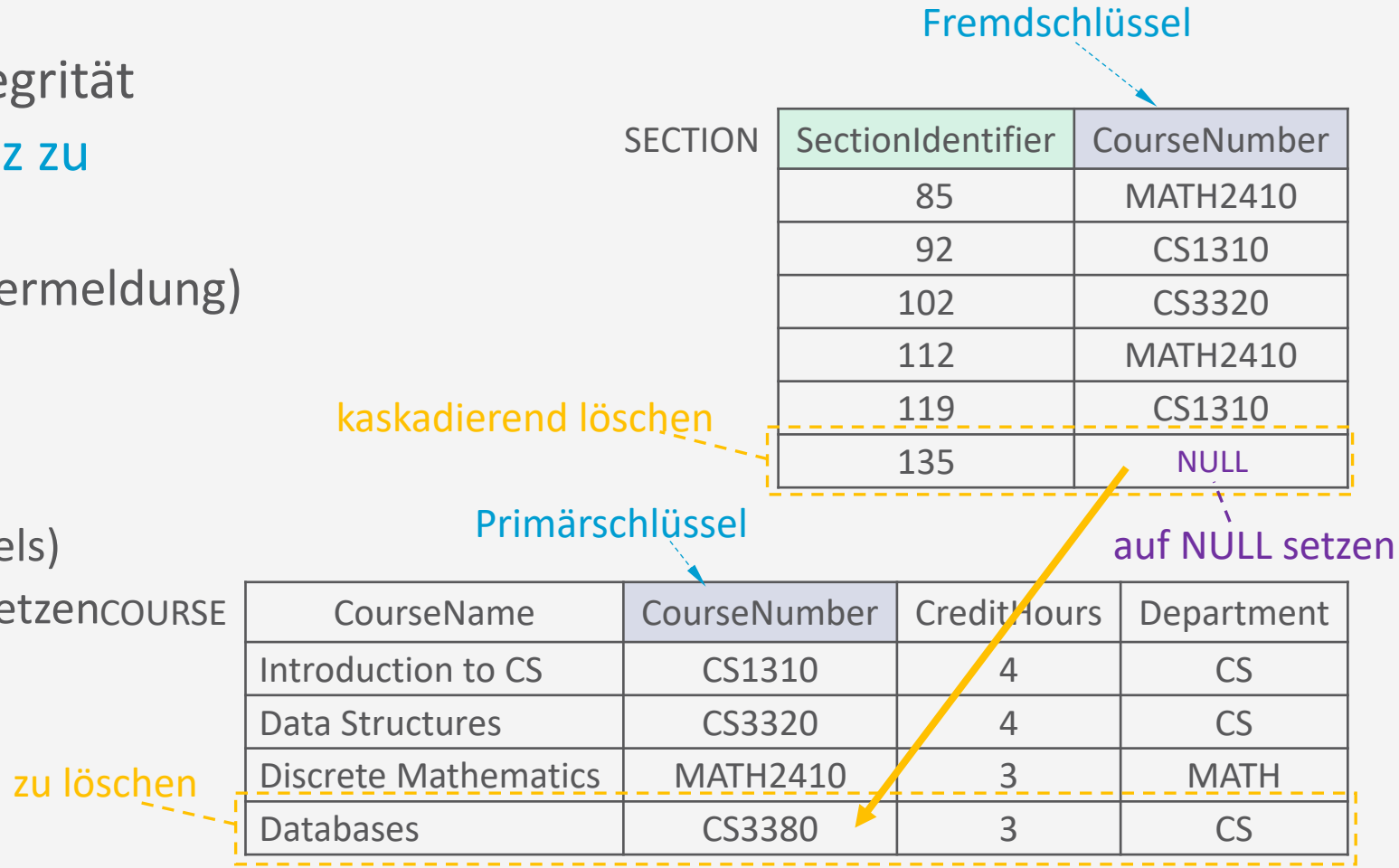
COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

zu löschen

Löschen von Tupeln: Fehlersituation

- Fehlersituation: Referenzielle Integrität
- Lösungsansätze um die Konsistenz zu bewahren
  - DELETE Operation abweisen (Fehlermeldung)
  - Kaskadierend löschen
  - Betroffene Tupel korrigieren
    - Fremdschlüssel auf NULL setzen (wenn nicht Teil des Primärschlüssels)
    - Fremdschlüssel auf Default-Wert setzen (wenn vorhanden)
    - Fremdschlüssel auf existierenden Schlüsselwert setzen



Aktualisieren von Tupeln

- **UPDATE** R
  SET ...
  [**WHERE** <Bedingung>]
  - Aktualisiert / ändert eine Menge von Tupeln aus einer Relation
  - Identifikation bestimmter Tupel über Schlüsselwerte/Bedingungen
    - **UPDATE** COURSE
      SET Course.CreditHours = 4
      WHERE Course.CourseNumber = CS3380
    - **UPDATE** COURSE
      SET Course.CreditHours = 4
      WHERE Course.CreditHours = 3

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete MathEmatics	MATH2410	4	MATH
Databases	CS3380	4	CS

Aktualisieren von Tupeln

- Fehlersituationen
  - Wenn keine Primär- oder Fremdschlüssel geändert werden:
    - Nur Wertebereichseinschränkungen
      - **UPDATE** COURSE
        SET Course.CreditHours = four
        WHERE Course.CourseNumber = CS3380
    - Kein Tupel durch Bedingung angesprochen (keine Auswirkung!)
      - **UPDATE** COURSE
        SET Course.CreditHours = 4
        WHERE Course.CourseNumber = CS3390
  - Ansonsten alle Probleme von **INSERT** und **DELETE**

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Aktualisieren von Tupeln

- Fehlersituationen
  - Ansonsten alle Probleme von **INSERT** und **DELETE**; Beispiele
    - Primärschlüsseländerungen: Neuer Schlüsselwert schon belegt?
    - **UPDATE** COURSE **SET** Course.CourseNumber = CS1310 ...
    - Referenz auf alten Schlüssel in anderer Relation vorhanden? → Kaskadierend aktualisieren
    - **UPDATE** COURSE **SET** Course.CourseNumber = CS3390 ...
    - Fremdschlüsseländerungen: Existiert neuer Fremdschlüsselwert in referenzierter Relation?
    - **UPDATE** Section.CourseNumber = CS3390 ...
- Lösungen:
  Prinzipiell gleiche Optionen
  wie bei DELETE

COURSE

CourseName	CourseNumber	CreditHours	Department
Introduction to CS	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Databases	CS3380	3	CS

Zwischenzusammenfassung

- Relationale Algebra als Anfragesprache an Relationen
- Entfernende Operatoren
  - Selektion σ, Projektion π
- Umbenennung ρ
- Kombinierende Operatoren
  - Klassisch: Vereinigung $\cup$, Schnitt $\cap$, Differenz $-$
  - Kartesisches Produkt $\times$, Join $\bowtie$ und weitere Join-Arten, Outer Union, Division
- Minimalität der relationalen Algebra
- Aggregieren, gruppieren
- Relationenzustände ändern
  - Einfügen, löschen, aktualisieren

Übersicht: 3. Das Relationale Datenmodell

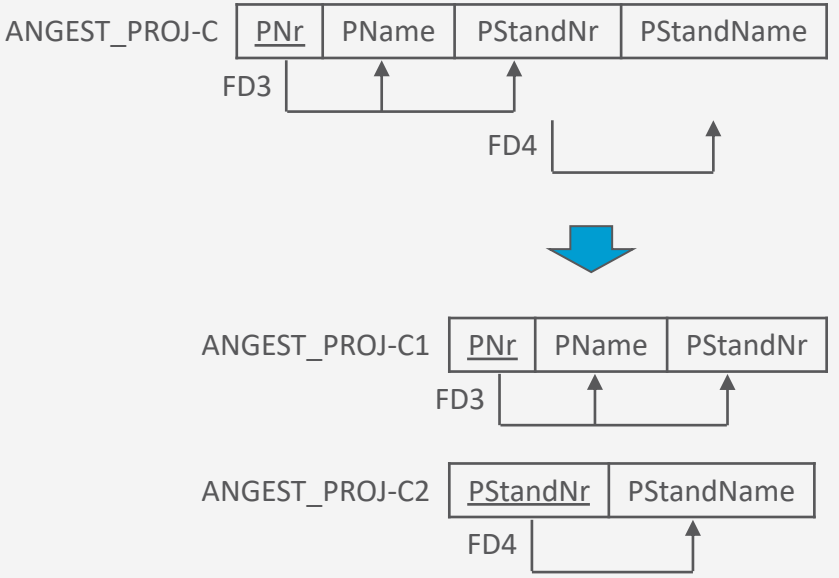
A. Relationales Datenmodell
- Relationen, Attribute, relationale Datenbanken und -schemata
- Schlüssel: Primärschlüssel, Fremdschlüssel, referentielle Integrität

B. Entwurf relationaler Schemata
- Vom ER-Diagramm zum relationalen Modell

C. Relationale Algebra
- $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$
- Minimalität
- Aggregieren, gruppieren
- Einfügen, löschen, aktualisieren

→ Datenbank-Entwurf

WWU
MÜNSTER



Datenbank-Entwurf

Datenbanken

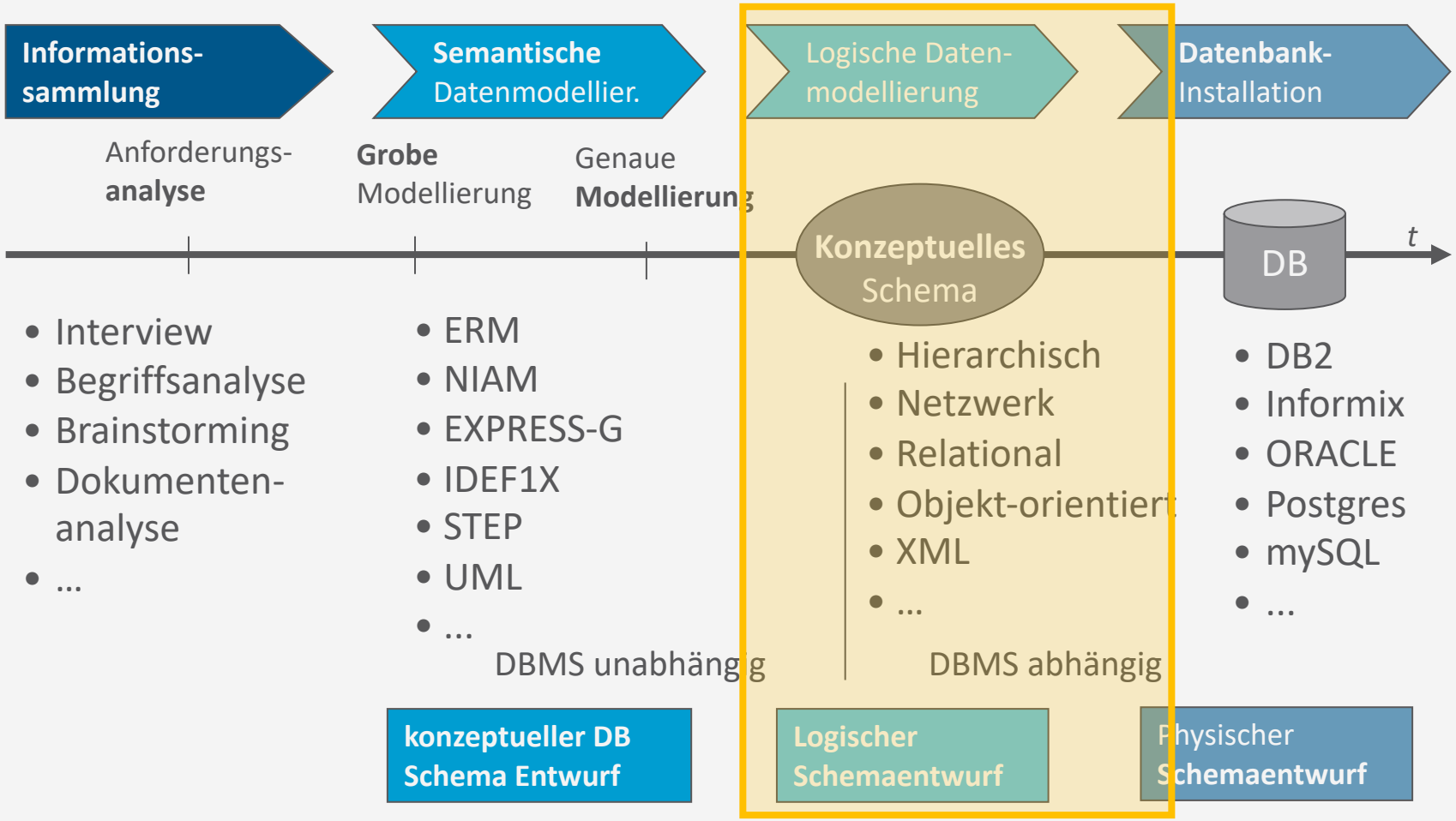
Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



„Schlechte“ Relation

AName	SVNr	Geb	Adr	AbtNr	Abt	ALeitSVNr
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291, Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975, FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	Rice, Houston, TX	5	Research	333445555
Jabber, Ahmad V.	987987987	1969-03-29	Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	Stone, Houston, TX	1	Headquarters	888665555

„Schlechte“ Relation: Einfügeanomalie

AName	SVNr	Geb	Adr	AbtNr	Abt	ALeitSVNr
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291, Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975, FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	Rice, Houston, TX	5	Research	333445555
Jabber, Ahmad V.	987987987					87654321
Borg, James E.	888665555	1937-11-10	Stone, Houston, TX	1	Headquarters	888665555
Grawunder, M	007	1971-02-17	Barßel	1	Hädquarters	886665554
				42	Geheim	007

neuer Angestellter → Information zu Abteilung (konsistent?) → Tippfehler?

keine neue Abteilung ohne Mitarbeiter

„Schlechte“ Relation: Update-Anomalie

AName	SVNr	Geb	Adr	AbtNr	Abt	ALeitSVNr
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston TX	5	Research	732456732
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston TX	5	Research	732456732
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291, Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975, FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	Rice, Houston, TX	5	Research	732456732
Jabber, Ahmad V.	987987987	1969-03-29	Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	Stone, Houston, TX	1	Headquarters	888665555

Abteilung Research bekommt neuen Leiter ... hoffentlich keine Zeile vergessen

„Schlechte“ Relation: Löschanomalie

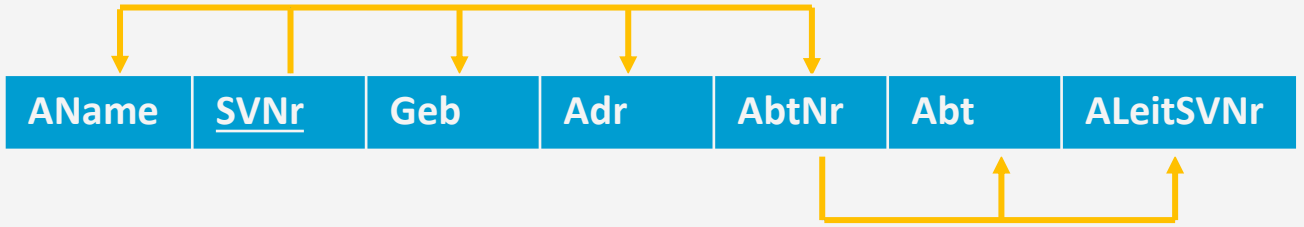
AName	SVNr	Geb	Adr	AbtNr	Abt	ALeitSVNr
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291, Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975, FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	Rice, Houston, TX	5	Research	333445555
Jabber, Ahmad V.	987987987	1969-03-29	Dallas, Houston, TX	4	Administration	987654321
~~Borg, James E.~~	~~888665555~~	~~1937-11-10~~	~~Stone, Houston, TX~~	~~1~~	~~Headquarters~~	~~888665555~~

James E. Borg geht in den Ruhestand. Es gibt aber noch keinen Nachfolger ... Wo ist die Abteilung Headquarters hin??

„Schlechte“ Relation

- Problem: Kombination aus Angestellten- mit Abteilungsinformation
- Einfüge-Anomalie
  - Neuer Angestellter → Information zu Abteilung konsistent?
  - Keine neue Abteilung ohne Mitarbeiter
- Update-Anomalie
  - Änderung des Abteilungsleiters → Update in allen Angestellten-Einträgen?
- Lösch-Anomalie
  - Letzter Angestellter einer Abteilung gelöscht → Abteilung verschwindet

- Was fällt auf?
  - Es tauchen Redundanzen in den Tupeln auf
  - Verschiedene Abhängigkeiten in der Relation
  - Änderungen an den Daten können zu Anomalien führen, da
    - Informationen an mehreren Stellen verändert werden müssen
    - Abhängigkeiten nicht berücksichtigt werden



Ziele der relationalen Entwurfstheorie

- Bewertung der Qualität eines Relationenschemas
  - Redundanz
    - Viel Redundanz erhöht Gefahr für Anomalien, vor allem bei Updates
  - Einhaltung von Konsistenzbedingungen
    - Funktionale Abhängigkeiten, anhand derer Konsistenz geprüft werden kann
- Normalformen als Gütekriterium
- Gegebenenfalls Verbesserung eines Relationenschemas durch
  - Synthesealgorithmus
  - Zerlegungsalgorithmus

- Meist gute Qualität bei aus validen ER-Diagrammen erstellte DB-Schemata

Überblick: 4. Datenbankentwurf

A. Funktionale Abhängigkeiten

- Definition, Schlüssel, Ableitungen
- Attributhülle, kanonische Überdeckung

B. Normalformen

- Zerlegung von Relationen
- Exkurs: NF^2; 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
- Synthesealgorithmus, Zerlegungsalgorithmus

C. Datenqualität

- Datenqualitätsprobleme, Dimensionen der Datenqualität

Funktionale Abhängigkeiten

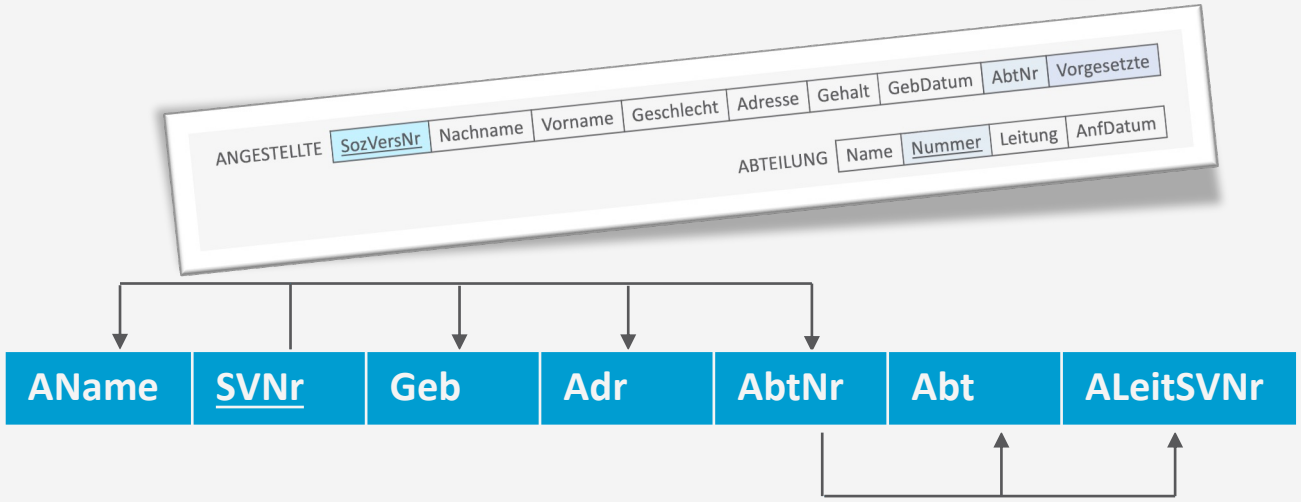
- $R(A_1, \ldots, A_n)$ sei ein Schema, α und β seien Attributteilmengen von R
- FD: $\alpha \rightarrow \beta$ sei eine funktionale Abhängigkeit (*functional dependency*), genau dann wenn

$$\forall t_1, t_2 \in r(R) : \text{Wenn } t_1[\alpha] = t_2[\alpha] \text{ gilt, gilt auch } t_1[\beta] = t_2[\beta]$$

  - D.h., Werte von α bestimmen eindeutig Werte von β
  - FD heißt *trivial*, wenn $\beta \subseteq \alpha$
  - Für ANGEST_ABT gilt:
    - F1: $\{SVNr\} \rightarrow \{AName, Geb, Adr, AbtNr\}$
    - F2: $\{AbtNr\} \rightarrow \{Abt, ALeitSVNr\}$
    - Relationen abgeleitet aus ER-Modell
- Wenn α ein Schlüssel ist, dann gilt $\alpha \rightarrow \beta$ für alle möglichen β aus R

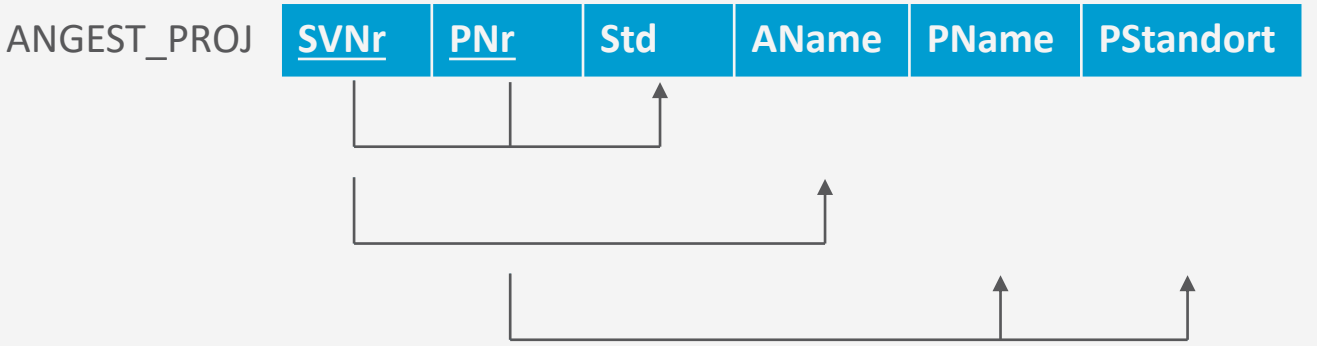
Folgt aus $\alpha \rightarrow \beta$ auch $\beta \rightarrow \alpha$?

Was ist, wenn α ein Schlüssel ist?



Funktionale Abhängigkeiten

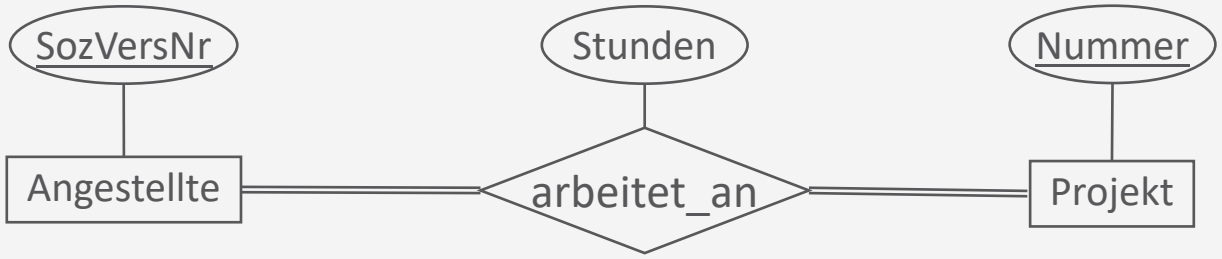
- Für ANGEST_PROJ gilt
  - F3: $\{SVNr, PNr\} \rightarrow \{Std\}$
  - F4: $\{SVNr\} \rightarrow \{AName\}$
  - F5: $\{PNr\} \rightarrow \{PName, PStandort\}$
  - Relationen abgeleitet aus ER-Modell:



ANGESTELLTE	SozVersNr	Nachname	Vorname	Geschlecht	Adresse	Gehalt	GebDatum	AbtNr	Vorgesetzte

ARBEITET_AN	ProjNr	SozVersNr	Stunden

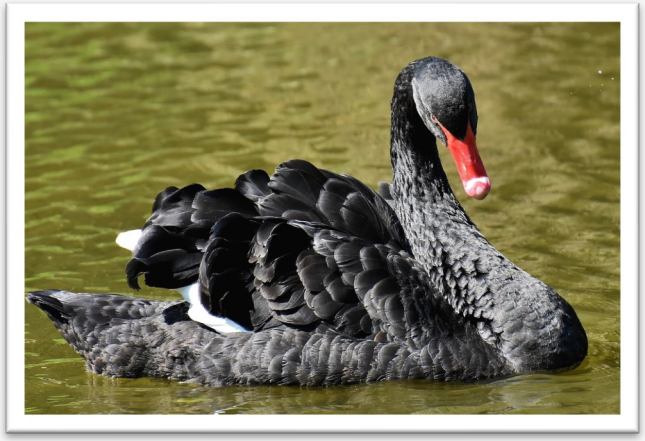
PROJEKT	Nummer	Name	Standort	AbtNr



Wo kommen die FDs her?

- Können die nicht einfach aus den Daten abgeleitet werden?
- Beispiel: Tierart → Farbe?

Datum	Tierart	Farbe
12.3.2010	Schwan	weiß
14.3.2010	Fuchs	rot
17.3.2010	Schwan	weiß



"Not all swans are white."

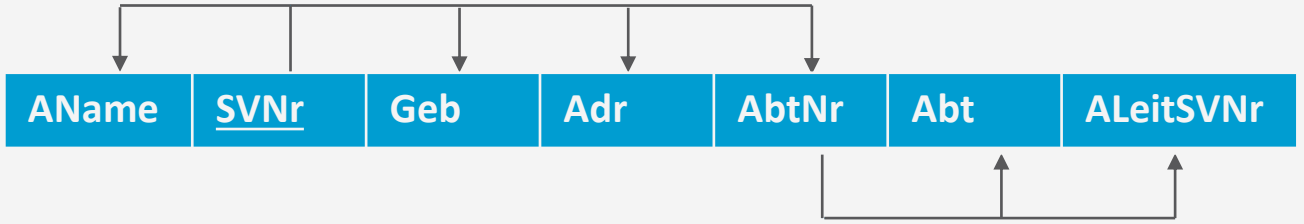
- FDs können nicht aus den Daten (Extension) abgeleitet werden
- FDs sind Eigenschaften des Relationenschemas (Intension)
- Sie müssen vom DB-Designer definiert werden

Schlüssel

- $\alpha \subseteq R$ ist ein Superschlüssel, falls gilt:
  - $\alpha \rightarrow R$
- β ist voll funktional abhängig von α genau dann, wenn gilt
  - $\alpha \rightarrow \beta$ und
  - α kann nicht mehr verkleinert werden (α minimal), d.h.
    - $\forall A \in \alpha$ folgt, dass $(\alpha - \{A\}) \rightarrow \beta$ nicht gilt, oder kürzer
    - $\forall A \in \alpha : \neg\big((\alpha - \{A\}) \rightarrow \beta\big)$
  - Notation für volle funktionale Abhängigkeit: $\alpha \rightarrow\!\!\!\!\cdot\ \beta$
- $\alpha \subseteq R$ ist ein Kandidatenschlüssel, falls gilt:
  - $\alpha \rightarrow\!\!\!\!\cdot\ R$

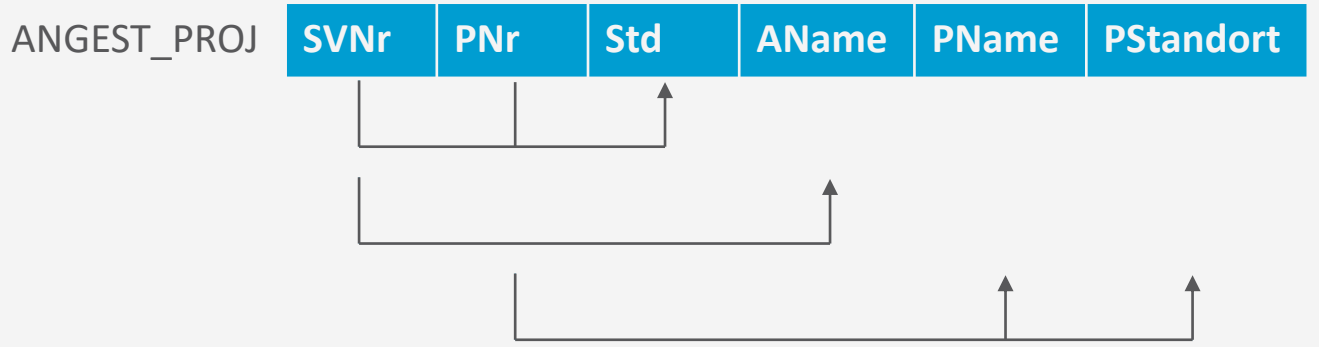
Schlüssel: Beispiel 1

- Beispiel: Für ANGEST_ABT gelten $F = \{F1, F2\}$
  - F1: $\{SVNr\} \rightarrow \{AName, Geb, Adr, AbtNr\}$
  - F2: $\{AbtNr\} \rightarrow \{Abt, ALeitSVNr\}$
- $\{SVNr, AbtNr\} \rightarrow \{SVNr, AbtNr, AName, Geb, Adr, Abt, ALeitSVNr\}$
  - $\{SVNr, AbtNr\}$ ein Superschlüssel?
    - Ja, da alle Attribute eindeutig durch $\{SVNr, AbtNr\}$ bestimmt werden
  - $\{SVNr, AbtNr, AName, Geb, Adr, Abt, ALeitSVNr\}$ voll funktional abhängig von $\{SVNr, AbtNr\}$?
    - Nein, da $\{SVNr, AbtNr\}$ verkleinert werden kann: $\{SVNr\}$ reicht aus
  - $\{SVNr, AbtNr\}$ Kandidatenschlüssel?
    - Nein, da volle funktionale Abhängigkeit nicht gilt



Schlüssel: Beispiel 2

- Für ANGEST_PROJ gilt
  - F3: $\{SVNr, PNr\} \rightarrow \{Std\}$
  - F4: $\{SVNr\} \rightarrow \{AName\}$
  - F5: $\{PNr\} \rightarrow \{PName, PStandort\}$
  - Volle funktionale Abhängigkeiten
    - $\{SVNr\} \rightarrow\cdot \{SVNr, AName\}$
      - Kein Superschlüssel, da z.B. Std nicht allein von $SVNr$ abhängig
    - $\{PNr\} \rightarrow\cdot \{PNr, PName, PStandort\}$
      - Kein Superschlüssel, da z.B. Std nicht allein von PNr abhängig
    - $\{SVNr, PNr\} \rightarrow\cdot \{SVNr, PNr, Std, AName, PName, PStandort\}$
      - Super- und Kandidatenschlüssel, da $\{SVNr, PNr\} \rightarrow\cdot$ ANGEST_PROJ



Wie sehen weitere Superschlüssel aus?

Wie sieht es mit deren voller funktionaler Abhängigkeit aus?

Herleitung funktionaler Abhängigkeiten

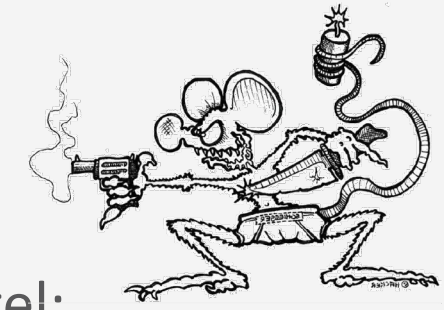
- Aus gegebenen FDs können weitere FDs abgeleitet werden
  - Mittels Inferenzregeln
- Transitive Hülle F^+:
  - Auch manchmal Abschluss (engl. *closure*) genannt
  - Gegeben eine Menge von FDs F
  - F^+: Menge aller FDs, die mit Inferenzregeln abgeleitet werden können

- Es gibt sechs Inferenzregeln (RATZAP)
  - *IR1*: **R**eflexivitätsregel
  - *IR2*: **A**ugmentationsregel
  - *IR3*: **T**ransivitätsregel
  - *IR4*: **Z**erlegungsregel
  - *IR5*: **A**dditive oder Vereinigungsregel
  - *IR6*: **P**seudotransitive Regel

IR1-3 auch unter Armstrong-Axiomen bekannt
- Vollständig: Erzeugen nur gültige FDs
- Korrekt: Jede mögliche FD lässt sich herleiten

IR4-6 lassen sich aus IR1-3 ableiten
- Erleichtern Ableitungen

Die sechs Inferenzregeln (RATZAP)



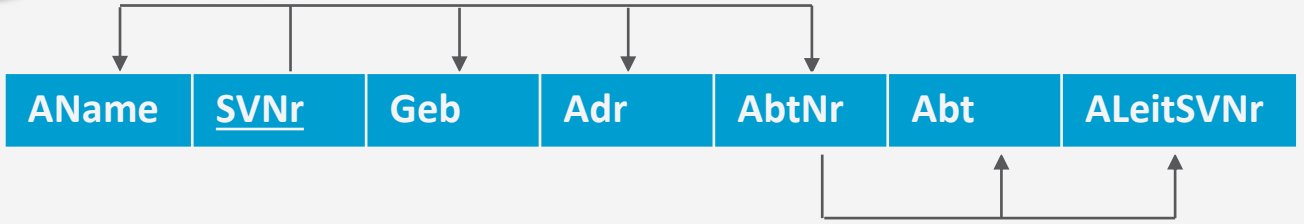
- **R**eflexivitätsregel:
  - *IR1*: Falls $Y \subseteq X$, dann $X \rightarrow Y$
  - Eine Attributmenge bestimmt sich immer selbst oder eine ihrer Teilmengen
- **A**ugmentationsregel:
  - *IR2*: Falls $X \rightarrow Y$, dann $XZ \rightarrow YZ$
  - Hinzufügen von Attributen auf beiden Seiten führt zu weiterer Regel
- **T**ransitivitätsregel:
  - *IR3*: Falls $\{X \rightarrow Y, Y \rightarrow Z\}$, dann $X \rightarrow Z$
- **Z**erlegungsregel:
  - *IR4*: Falls $X \rightarrow YZ$, dann $X \rightarrow Y$ und $X \rightarrow Z$
  - Attribute auf der rechten Seite können entfernt und FDs in Teilmengen zerlegt werden
- **A**dditive oder Vereinigungsregel:
  - *IR5*: Falls $\{X \rightarrow Y, X \rightarrow Z\}$, dann $X \rightarrow YZ$
  - Gegenstück zu **Z** (*IR4*): Regel wieder zusammenfassen
- **P**seudotransitive Regel:
  - *IR6*: Falls $\{X \rightarrow Y, WY \rightarrow Z\}$, dann $WX \rightarrow Z$
  - Transitivität im Kontext

Beispiel für die Anwendung von Inferenzregeln

- Beispiel: $F = \{F1, F2\}$
  - F1: $\{SVNr\} \rightarrow \{AName, Geb, Adr, AbtNr\}$
  - F2: $\{AbtNr\} \rightarrow \{Abt, ALeitSVNr\}$
  - Zusätzlich können u.a. abgeleitet werden
    - F3: $\{SVNr\} \rightarrow \{Abt, ALeitSVNr\}$
      - IR3
    - F4: $\{SVNr\} \rightarrow \{Adr, AbtNr\}$
      - IR4
    - F5: $\{SVNr\} \rightarrow \{SVNr\}$
      - IR1

- Regeln
  - *IR1*: Falls $Y \subseteq X$, dann $X \rightarrow Y$
  - *IR2*: Falls $X \rightarrow Y$, dann $XZ \rightarrow YZ$
  - *IR3*: Falls $\{X \rightarrow Y, Y \rightarrow Z\}$, dann $X \rightarrow Z$
  - *IR4*: Falls $X \rightarrow YZ$, dann $X \rightarrow Y$ und $X \rightarrow Z$
  - *IR5*: Falls $\{X \rightarrow Y, X \rightarrow Z\}$, dann $X \rightarrow YZ$
  - *IR6*: Falls $\{X \rightarrow Y, WY \rightarrow Z\}$, dann $WX \rightarrow Z$

Was für FDs können abgeleitet werden?



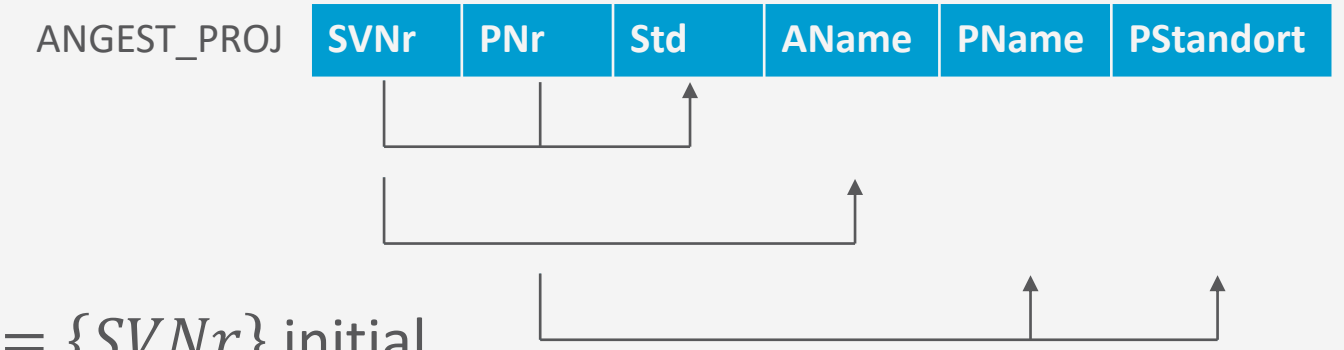
Attributhülle

- Attributhülle α^+ für eine Menge von Attributen α
  - Alle Attribute, die von α funktional abhängen
- Berechnung
  - Eingabe: Menge von FDs F, Menge von Attributen α
  - Ausgabe: Vollständige Menge von Attributen α^+, für die gilt: $\alpha \rightarrow \alpha^+$
  - Vorgehen:
    - Initialisierung: $\alpha^+ \leftarrow \alpha$
    - Für jede FD $\beta \rightarrow \gamma$ in F
      - Wenn β in α^+ vorkommt, dann füge γ zu α^+ hinzu
    - Gebe α^+ aus

```
AttrHülle(F, α)
    α⁺ ← α
    while α⁺ geändert do
        for each β → γ ∈ F do
            if β ⊆ α⁺ then
                α⁺ ← α⁺ ∪ γ
    return α⁺
```

Attributhülle: Beispiel

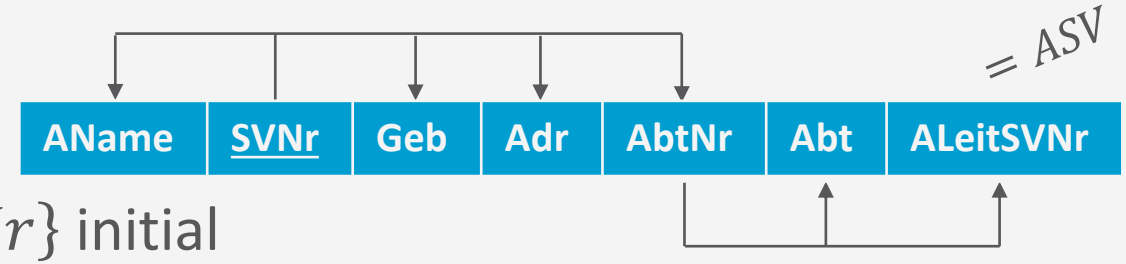
- Für ANGEST_PROJ gilt
  - F3: $\{SVNr, PNr\} \rightarrow \{Std\}$
  - F4: $\{SVNr\} \rightarrow \{AName\}$
  - F5: $\{PNr\} \rightarrow \{PName, PStandort\}$
- Attributhülle $\alpha = \{SVNr\}$ mit $\alpha^+ = \alpha = \{SVNr\}$ initial
  - Betrachte F3, F4, F5:
    - Betrachte F3 mit $\beta = \{SVNr, PNr\}$: $\beta \not\subseteq \alpha^+$ (keine Änderung)
    - Betrachte F4 mit $\beta = \{SVNr\}$: $\beta \subseteq \alpha^+$, damit $\alpha^+ = \{SVNr, AName\}$
    - Betrachte F5 mit $\beta = \{PNr\}$: $\beta \not\subseteq \alpha^+$ (keine Änderung)
  - Änderung in α^+, also nochmal: Betrachte F3, F4, F5
    - Keine Änderung mehr in α^+
  - Ausgabe: $\alpha^+ = \{SVNr, AName\}$



AttrHülle(F, α)
  $\alpha^+ \leftarrow \alpha$
  while α^+ geändert **do**
    for each $\beta \rightarrow \gamma \in F$ **do**
      if $\beta \subseteq \alpha^+$ **then**
        $\alpha^+ \leftarrow \alpha^+ \cup \gamma$
  return α^+

Attributhülle: Beispiel

- F1: $\{SVNr\} \rightarrow \{AName, Geb, Adr, AbtNr\}$
- F2: $\{AbtNr\} \rightarrow \{Abt, ASV\}$
- Attributhülle $\alpha = \{SVNr\}$, mit $\alpha^+ = \alpha = \{SVNr\}$ initial
  - Betrachte F2, F1
    - Betrachte F2 mit $\beta = \{AbtNr\}$: $\beta \not\subseteq \alpha^+$ (keine Änderung)
    - Betrachte F1 mit $\beta = \{SVNr\}$: $\beta \subseteq \alpha^+$, damit $\alpha^+ = \alpha^+ \cup \{AName, Geb, Adr, AbtNr\}$
  - Änderung in α^+; also nochmal: Betrachte F2, F1
    - Betrachte F2 mit $\beta = \{AbtNr\}$: $\beta \subseteq \alpha^+$, damit $\alpha^+ = \alpha^+ \cup \{Abt, ASV\}$
    - Betrachte F1 mit $\beta = \{SVNr\}$: $\beta \subseteq \alpha^+$, aber keine Änderung in α^+
  - Änderung in α^+, also nochmal: Betrachte F2, F1
    - Keine Änderung mehr in α^+
  - Ausgabe: $\alpha^+ = \{SVNr, AName, Geb, Adr, AbtNr, Abt, ASV\}$




```
AttrHülle(F, α)
    α⁺ ← α
    while α⁺ geändert do
        for each β → γ ∈ F do
            if β ⊆ α⁺ then
                α⁺ ← α⁺ ∪ γ
    return α⁺
```

Attributhülle

- Attributhülle α^+ für eine Menge von Attributen α
  - Alle Attribute, die von α funktional abhängen
- Nutzen
  - Berechnung der transitiven Hülle F^+:
    - Für alle $\gamma \subseteq R$ und alle $S \subseteq \gamma^+$ enthält F^+ die FD $\gamma \rightarrow S$
  - Bestimmung von Superschlüsseln:
    - α Superschlüssel, wenn α^+ alle Attribute von R enthält
  - Überprüfung (voll) funktionaler Abhängigkeiten:
    - $\alpha \rightarrow \beta$ gilt, wenn $\beta \subseteq \alpha^+$

```
AttrHülle(F, α)
    α⁺ ← α
    while α⁺ geändert do
        for each β → γ ∈ F do
            if β ⊆ α⁺ then
                α⁺ ← α⁺ ∪ γ
    return α⁺
```

Redundanz in FDs & Kanonische Überdeckung

- Menge von FDs F kann redundante FDs enthalten oder FDs mit überflüssigen Attributen
  - Macht Überprüfung von Kosistenzen bei Änderungen im DB-Zustand aufwendiger als nötig
- Gesucht: Minimale Menge an FDs, die ausreichend sind, um F^+ zu beschreiben

→ Kanonische Überdeckung F_C, welche folgende drei Kriterien erfüllt

1. $F_C \equiv F$, d.h. $F_C^+ = F^+$
2. In F_C existieren keine FDs $\alpha \rightarrow \beta$, die überflüssige Attribute in α oder β enthalten
   - Kein überflüssiges Attribut in α: $\forall A \in \alpha : (F_C - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - \{A\}) \rightarrow \beta\} \not\equiv F_C$
     - Test: Wenn $\beta \subseteq (\alpha - \{A\})^+$, dann A überflüssig (kann aus α entfernt werden)
   - Kein überflüssiges Attribut in β: $\forall B \in \beta : (F_C - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - \{B\})\} \not\equiv F_C$
     - Test: Wenn $B \in \alpha^+$, dann B überflüssig (kann aus β entfernt werden)
3. Jede linke Seite einer FD in F_C ist einzigartig
   - Additive Regel sukzessive anwenden: Wenn FDs $\alpha \rightarrow \beta, \alpha \rightarrow \gamma$, ersetze durch $\alpha \rightarrow \beta\gamma$

Kanonische Überdeckung: Berechnung

- Eingabe: Menge von FDs F
- Ausgabe: Kanonische Überdeckung F_C (Menge von FDs, welche die drei Kriterien erfüllt)
- Initialisiere F_C mit F

1. Führe für jede FD $\alpha \rightarrow \beta \in F_C$ die Linksreduktion durch:
   - Überprüfe für alle $A \in \alpha$:
     - Wenn $\beta \subseteq (\alpha - \{A\})^+$ bzgl. F_C, ersetze $\alpha \rightarrow \beta$ mit $(\alpha - \{A\}) \rightarrow \beta$ in F_C
2. Führe für jede FD $\alpha \rightarrow \beta \in F_C$ die Rechtsreduktion durch:
   - Überprüfe für alle $B \in \beta$:
     - Wenn $B \in \alpha^+$ bzgl. $(F_C - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - \{B\})\}$, ersetze $\alpha \rightarrow \beta$ mit $\alpha \rightarrow (\beta - \{B\})$ in F_C
3. Entferne die FDs der Form $\alpha \rightarrow \emptyset$ aus F_C
4. Ersetze alle FDs der Form $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$ durch $\alpha \rightarrow (\beta_1 \cup \cdots \cup \beta_n)$ in F_C

Kanonische Überdeckung: Pseudocode

- Eingabe:
  Menge von FDs F
- Ausgabe:
  Kanonische Überdeckung F_C
- Initialisiere F_C mit F

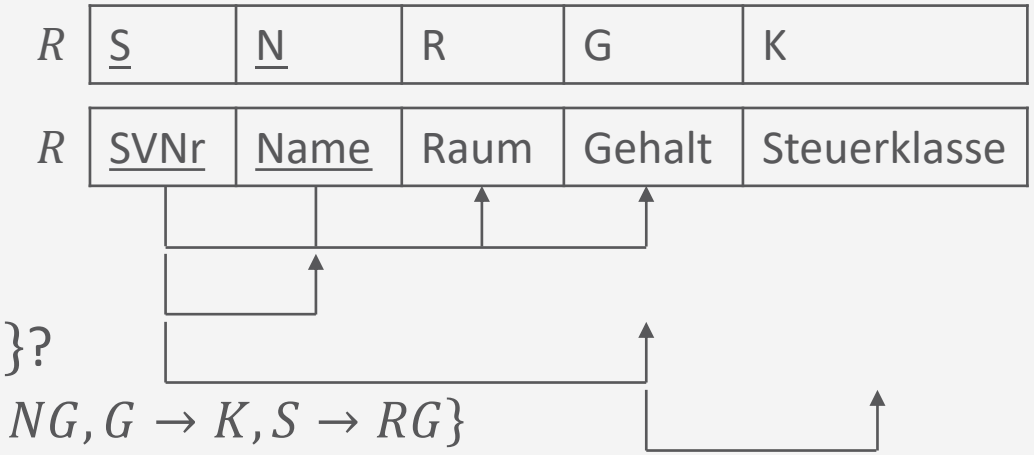
1. Linksreduktion
2. Rechtsreduktion
3. FDs mit leerem β entfernen
4. FDs mit gleichem α zusammenfassen

KANON(F)
  $F_C \leftarrow F$
  for each $\alpha \rightarrow \beta \in F_C$ **do**
    for each $A \in \alpha$ **do**
      if $\beta \subseteq$ ATTRHÜLLE$(F_C, \alpha - \{A\})$ **then**
        $F_C \leftarrow (F_C - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - \{A\}) \rightarrow \beta\}$
  for each $\alpha \rightarrow \beta \in F_C$ **do**
    for each $B \in \beta$ **do**
      if $B \in$ ATTRHÜLLE$\big((F_C - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - \{B\})\}, \alpha\big)$ **then**
        $F_C \leftarrow (F_C - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - \{B\})\}$
  for each $\alpha \rightarrow \beta \in F_C$ **do**
    if $\beta = \emptyset$ **then**
      $F_C \leftarrow F_C - \{\alpha \rightarrow \beta\}$
  while $\exists \alpha \rightarrow \beta_1, \dots, \alpha \rightarrow \beta_n$ **do**
    $F_C \leftarrow (F_C - \{\alpha \rightarrow \beta_1, \dots, \alpha \rightarrow \beta_n\}) \cup \{\alpha \rightarrow (\beta_1 \cup \dots \cup \beta_n)\}$
  return F_C

Kanonische Überdeckung

- $F = \{S \to NG, G \to K, SN \to RG\} = F_C$

1. Linksreduktion: $SN \to RG$ in F_C testen
   - S überflüssig, d.h., $\{R, G\} \subseteq \{N\}^+ = \{N\}$?
     - Nein, daher keine Änderung in F_C
   - N überflüssig, d.h., $\{R, G\} \subseteq \{S\}^+ = \{S\} \cup \{N, G, K, R\}$?
     - Ja, daher $F_C \leftarrow (F_C - \{SN \to RG\}) \cup \{S \to RG\} = \{S \to NG, G \to K, S \to RG\}$



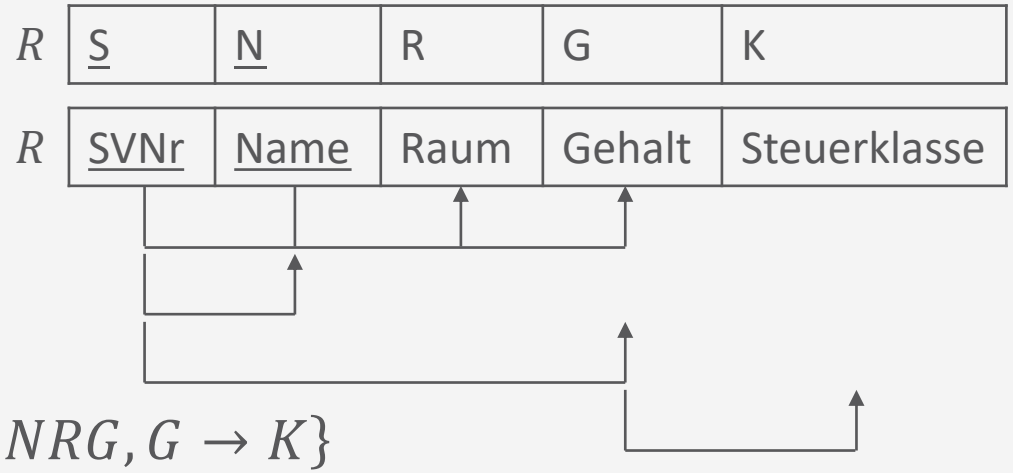
2. Rechtsreduktion: $S \to NG, S \to RG$ in F_C mit reduzierter FD testen
   - $S \to NG$: N überflüssig in $\{S \to G, G \to K, S \to RG\}$, d.h., $\{N\} \subseteq \{S\}^+ = \{S\} \cup \{G, K, R\}$?
     - Nein, daher keine Änderung in F_C
   - $S \to NG$: G überflüssig in $\{S \to N, G \to K, S \to RG\}$, d.h., $\{G\} \subseteq \{S\}^+ = \{S\} \cup \{N, R, G\} \cup \{K\}$?
     - Ja, daher $F_C \leftarrow (F_C - \{S \to NG\}) \cup \{S \to N\} = \{S \to N, G \to K, S \to RG\}$
   - $S \to RG$: beide nicht überflüssig

Kanonische Überdeckung

- $F = \{S \to NG, G \to K, SN \to RG\} = F_C$
- $F_C = \{S \to N, G \to K, S \to RG\}$ nach Schritt 1 + 2

3. FDs mit leerem β entfernen
   - Keine Änderung
4. FDs mit gleichem α zusammenfassen
   - $F_C \leftarrow (F_C - \{S \to N, S \to RG\}) \cup \{S \to NRG\} = \{S \to NRG, G \to K\}$

- Ergebnis: $F_C = \{S \to NRG, G \to K\}$



Ist F_C eindeutig?

Kanonische Überdeckung: Abstraktes Beispiel

- Relation über A, B, C
  - $F = \{A \to BC, B \to CA, C \to A\} = F_C$
- Schritt mit Effekt: Rechtsreduktion
  - $A \to BC$ zuerst betrachten
  - $A \to BC$: $B \in \{A\}^+ = \{A\} \cup \{C\}$?
    - Nein, keine Änderung
  - $A \to BC$: $C \in \{A\}^+ = \{A\} \cup \{C\}$?
    - Ja, daher $F_C = \{A \to B, B \to CA, C \to A\}$
  - $B \to CA$: $C \in \{B\}^+ = \{B\} \cup \{A\}$?
    - Nein, keine Änderung
  - $B \to CA$: $A \in \{B\}^+ = \{B\} \cup \{A\}$?
    - Ja, daher $F_C = \{A \to B, B \to C, C \to A\}$

- Relation über A, B, C
  - $F = \{A \to BC, B \to CA, C \to A\} = F_C$
- Schritt mit Effekt: Rechtsreduktion
  - $B \to CA$ zuerst betrachten
  - $B \to CA$: $C \in \{B\}^+ = \{B\} \cup \{A, C\}$?
    - Ja, daher $F_C = \{A \to BC, B \to A, C \to A\}$
  - $A \to BC$: $B \in \{A\}^+ = \{A\} \cup \{C\}$?
    - Nein, keine Änderung
  - $A \to BC$: $C \in \{A\}^+ = \{A\} \cup \{B\}$?
    - Nein, keine Änderung
  - Ergebnis: $F_C = \{A \to BC, B \to A, C \to A\}$

Kanonische Überdeckung nicht immer eindeutig

Zwischenzusammenfassung

- Probleme bei schlecht gebauten Relationen:
  - Einfüge-Anomalien, Update-Anomalien, Lösch-Anomalien
- Bewertung der Qualität über Konsistenzbedingungen durch FDs und Redundanz
- FDs
  - Erlauben Schlüssel zu bestimmen: Relation dann voll funktional abhängig
  - Transitive Hülle: Herleitung über Inferenzregeln (RATZAP)
  - Attributhülle: Menge von Attributen, die funktional abhängig sind von gegebenen Attributen
    - Erlaubt Berechnung von Superschlüsseln und transitiver Hülle, Test von funktionalen Abhängigkeiten
  - Kanonische Überdeckung zur Minimierung der Menge der FDs durch Entfernung von überflüssigen Attributen und FDs
    - Vorgehen: Rechtsreduktion, Linksreduktion, leere FDs löschen, gleiche α Regeln zusammenfassen
    - Nutzt auch die Attributhülle in ihren Berechnungen

Überblick: 4. Datenbankentwurf

A. Funktionale Abhängigkeiten
- Definition, Schlüssel, Ableitungen
- Attributhülle, kanonische Überdeckung

B. Normalformen
- Zerlegung von Relationen
- (Exkurs: NF^2), 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
- Synthesealgorithmus, Zerlegungsalgorithmus

C. Datenqualität
- Datenqualitätsprobleme, Dimensionen der Datenqualität

Erinnerung: Ziele der relationalen Entwurfstheorie

- Bewertung der Qualität eines Relationenschemas
  - Redundanz
    - Viel Redundanz erhöht Gefahr für Anomalien, vor allem bei Updates
  - Einhaltung von Konsistenzbedingungen
    - Funktionale Abhängigkeiten, anhand derer Konsistenz geprüft werden kann
- Normalformen als Gütekriterium
- Gegebenenfalls Verbesserung eines Relationenschemas durch
  - Synthesealgorithmus
  - Zerlegungsalgorithmus

- Meist gute Qualität bei aus validen ER-Diagrammen erstellten DB-Schemata

Normalisierung von Datenbank-Schemata & Normalformen (NF)

- Normalisierung bietet:
  - Normalformbedingungen, um NF zu testen
  - Vorgehen, um NF zu erreichen
    - Grundsätzliches Vorgehen:
      - Prüfung eines Relationenschemas auf eine Normalform
      - Wenn nicht erfüllt: Zerlegung in neue Relationenschemata, bis gewünschte Normalform erreicht ist
    - Je höher die NF, desto weniger Redundanz

Zerlegung von Schemata

- Zur Herstellung einer NF:
  Zerlegung eines Schemas R in Menge von Schemata $D = \{R_1, \dots, R_n\}$
  - Attributerhaltung der Zerlegung D:
    - Jedes Attribut aus R erscheint in mindestens einem R_i
    - Vereinigung der Attributmengen aller R_i entspricht damit der Attributmenge von R
- Korrektheitskriterien (beide sollen möglichst gelten)
  1. Verlustlosigkeit bzw. Eigenschaft des nicht-additiven JOINs
     - Die im ursprünglichen Relationenzustand $r(R)$ enthaltenen Informationen müssen aus den Zuständen der neuen Schemata D rekonstruierbar sein → keine unechten Tupel entstehen lassen
  2. Abhängigkeitserhaltung
     - Die für R geltenden Abhängigkeiten müssen auf die Schemata in D übertragbar sein

  ❖ Zusätzlich soll gelten, dass möglichst wenig Redundanz in den Daten herrscht

Verlustlosigkeit bzw. nicht-additiver JOIN

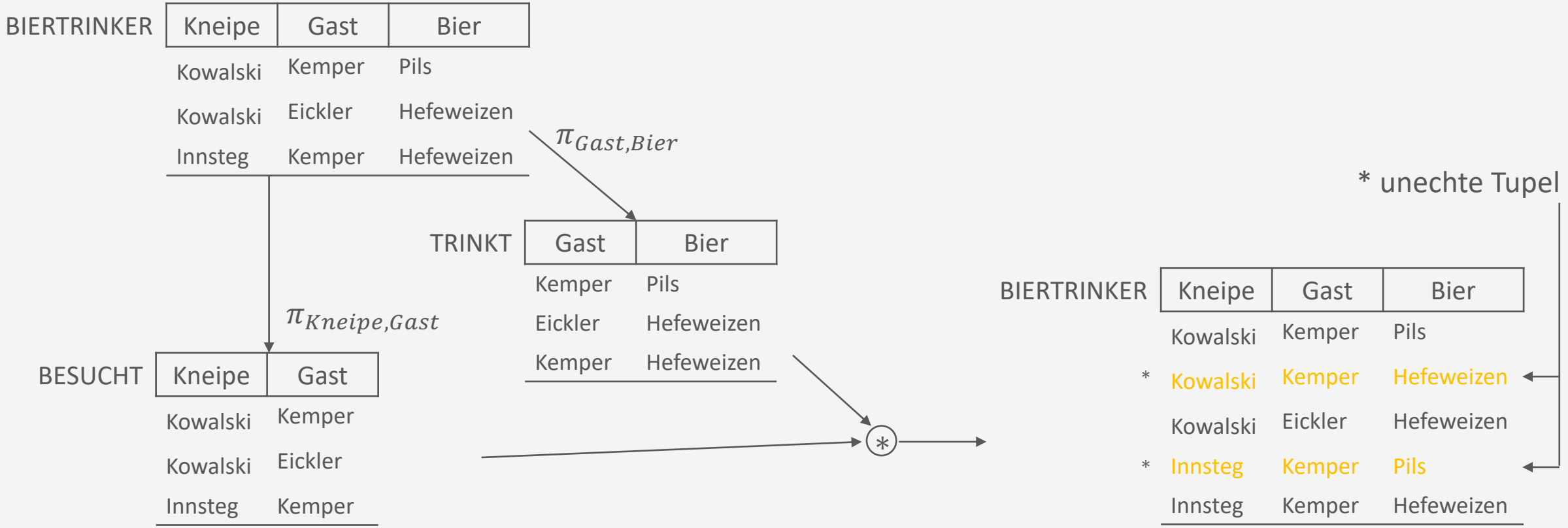
- Zerlegung $D = \{R_1, \dots, R_m\}$ von R ist verlustlos in Bezug auf die Abhängigkeitsmenge F in R, wenn für jede Relation $r(R)$, die F erfüllt, gilt

$$* \left(\pi_{R_1}(r), \dots, \pi_{R_m}(r)\right) = r$$

  - * bezeichnet hier den NATURAL JOIN
  - Hinreichende Bedingung für $D = \{R_1, R_2\}$ von R
    - Wenn $(R_1 \cap R_2) \rightarrow R_1 \in F^+$ oder $(R_1 \cap R_2) \rightarrow R_2 \in F^+$, dann ist D verlustlos
- D.h. JOINs über die zerlegten Relationen erzeugen keine „unechten“ Tupel
  - Informationsgehalt von R und D bleibt gleich

Beispiel

- Zerlegung, die nicht die Eigenschaft des nicht-additiven JOINs erfüllt

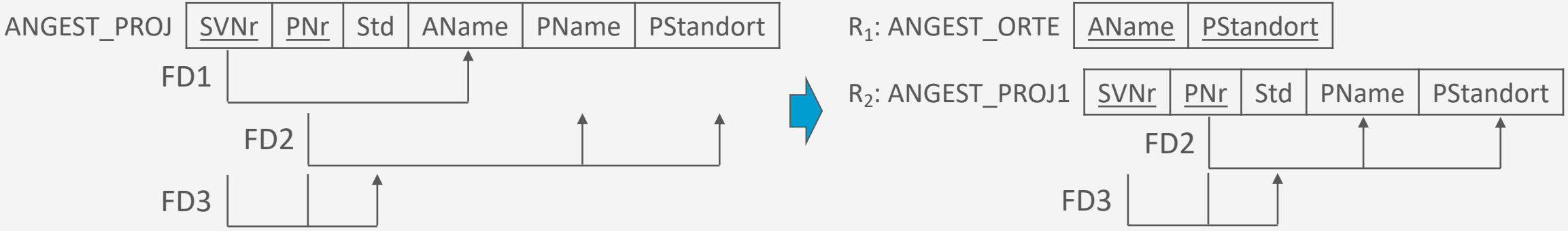


Nicht-additiver JOIN-Test

- Input: Relationenschema R, Zerlegung $D = \{R_1, R_2, \ldots, R_m\}$ von R, Menge von FDs F
  1. Erzeuge Matrix S: Zeile i für jede Relation R_i in D, Spalte j für jedes Attribut A_j in R
  2. Setze $S(i,j) := b_{ij}$ für alle Matrixeinträge
  3. Für jede Zeile i, jede Spalte j: Wenn Attribut A_j zu Relation R_i gehört, dann setze $S(i,j) := a_j$
  4. Wiederhole die folgende Schleife, bis keine Änderung mehr in S:
     - Für jede FD $X \rightarrow Y$ in F, für alle Zeilen in S mit den gleichen Symbolen in den Spalten, die Attributen in X entsprechen: Setze die Symbole in jeder Spalte, die einem Attribut in Y entsprechen, in allen diesen Zeilen wie folgt gleich: Falls eine der Zeilen ein Symbol "a" in der Spalte enthält,
       - Dann setze die Werte der anderen Zeilen in dieser Spalte auf dieses Symbol "a"
       - Sonst: Wähle eines der Symbole "b", die in einer Zeile für das Attribut erscheinen, und setze die Werte in den anderen Zeilen in der Spalte auf dieses Symbol "b";
  5. Besteht eine Zeile vollständig aus Symbolen "a", dann weist die Zerlegung die Eigenschaft des nicht-additiven JOINs auf, im anderen Fall hat die Zerlegung diese Eigenschaft nicht

Nicht-additiver JOIN-Test: Beispiel 1

- Gegeben $R = ANGEST\_PROJ = \{SVNr, AName, PNr, PName, PStandort, Std\}$
  - $F = \{\{SVNr\} \rightarrow \{AName\}, \{PNr\} \rightarrow \{PName, PStandort\}, \{SVNr, PNr\} \rightarrow \{Std\}\}$
- Zerlegung $D = \{R_1, R_2\}$
  - $R_1 = ANGEST\_ORTE = \{AName, PStandort\}$
  - $R_2 = ANGEST\_PROJ1 = \{SVNr, PNr, Std, PName, PStandort\}$

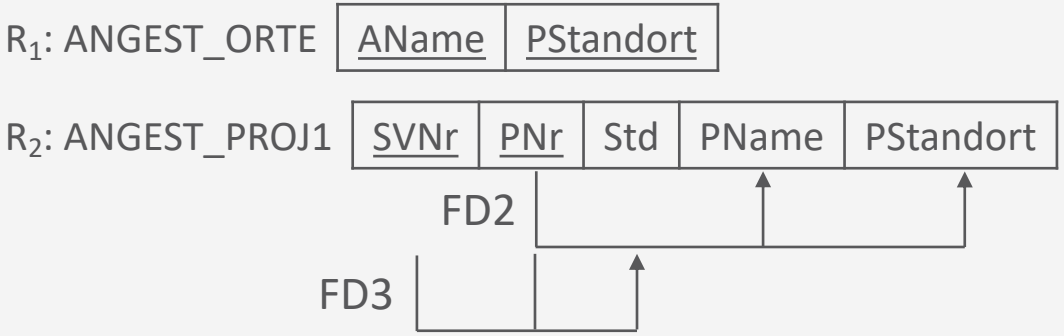


Nicht-additiver JOIN-Test: Beispiel 1

- Matrix:

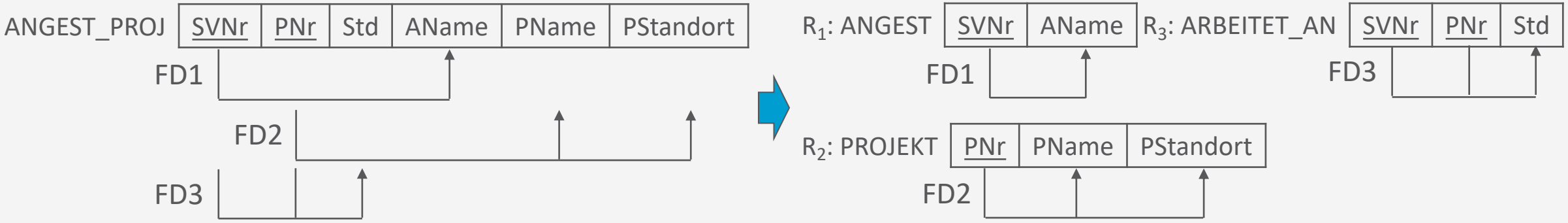
	SVNr	AName	PNr	PName	PStandort	Std
R_1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R_2	a_1	b_{22}	a_3	a_4	a_5	a_6

  - Keine Änderung über die Schleife
  - Keine Zeile nur mit „a“, d.h. keine Zerlegung mit gewünschter Eigenschaft
    → Additiver JOIN!



Nicht-additiver JOIN-Test: Beispiel 2

- Alternative Zerlegung von $ANGEST\_PROJ$
  - Gegeben $R = ANGEST\_PROJ = \{SVNr, AName, PNr, PName, PStandort, Std\}$
    - $F = \{\{SVNr\} \rightarrow \{AName\}, \{PNr\} \rightarrow \{PName, PStandort\}, \{SVNr, PNr\} \rightarrow \{Std\}\}$
  - Zerlegung $D = \{R_1, R_2, R_3\}$
    - $R_1 = ANGEST = \{SVNr, AName\}$
    - $R_2 = PROJEKT = \{PNr, PName, PStandort\}$
    - $R_3 = ARBEITET\_AN = \{SVNr, PNr, Std\}$



Nicht-additiver JOIN – Test: Beispiel 2

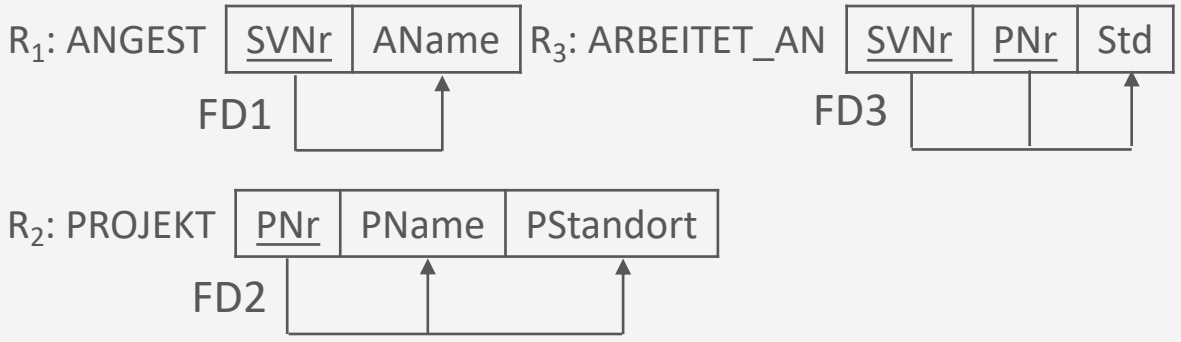
- Matrix
  - Nach Schritt 3:

	SVNr	AName	PNr	PName	PStandort	Std
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

  - Nach FD1 und FD2 in Schritt 4

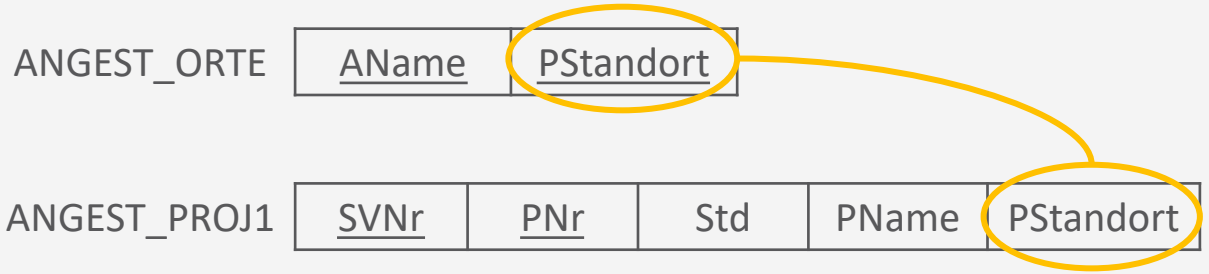
	SVNr	AName	PNr	PName	PStandort	Std
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	a_2	a_3	a_4	a_5	a_6

Zeile R_3 nur mit „a“ Symbolen
→ nicht-additive JOIN-Zerlegung



Nebenbemerkung: Unechte Tupel

- Auch aus Equijoins über Nicht-Schlüssel-Attribute können unechte (*spurious*) Tupel entstehen
  - Abhilfe:
    - Vermeidung gleichlautender Attribute, die keine Schlüssel sind
    - Wenn dies unvermeidbar ist: kein Join darüber bzw. Join nur über Schlüsselattribute
- Beispiel:
  - $ANGEST\_ORTE * ANGEST\_PROJ1$ (Natural Join)



Beispiel

ANGEST_ORTE

AName	PStandort
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
	Stafford
	Stafford
	Stafford
	Stafford
	Houston
	Houston

AName	SVNr
Smith, John B.	123456789
Wong, Franklin T.	333445555
Wallace, Jennifer S.	987654321
Narayan, Ramesh K.	666884444
English, Joyce A.	453453453
Borg, James E.	888665555

ANGEST_PROJ1

SVNr	PNr	Std	PName	PStandort
123456789	1	32.5	Product X	Bellaire
123456789	2	7.5	Product Y	Sugarland
666884444	3	40.0	Product Z	Houston
453453453	1	20.0	Product X	Bellaire
453453453	2	20.0	Product Y	Sugarland

* unechte Tupel

	SVNr	PNr	Std	PName	PStandort	AName
	123456789	1	32.5	Product X	Bellaire	Smith, John B.
*	123456789	1	32.5	Product X	Bellaire	English, Joyce A.
	123456789	2	7.5	Product Y	Sugarland	Smith, John B.
*	123456789	2	7.5	Product Y	Sugarland	English, Joyce A.
*	123456789	2	7.5	Product Y	Sugarland	Wong, Franklin T.
	666884444	3	40.0	Product Z	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	Product Z	Houston	Wong, Franklin T.
*	666884444	3	40.0	Product Z	Houston	Wallace, Jennifer S.
*	666884444	3	40.0	Product Z	Houston	Borg, James E.
*	453453453	1	20.0	Product X	Bellaire	Smith, John B.
	453453453	1	20.0	Product X	Bellaire	English, Joyce A.
	⋮					

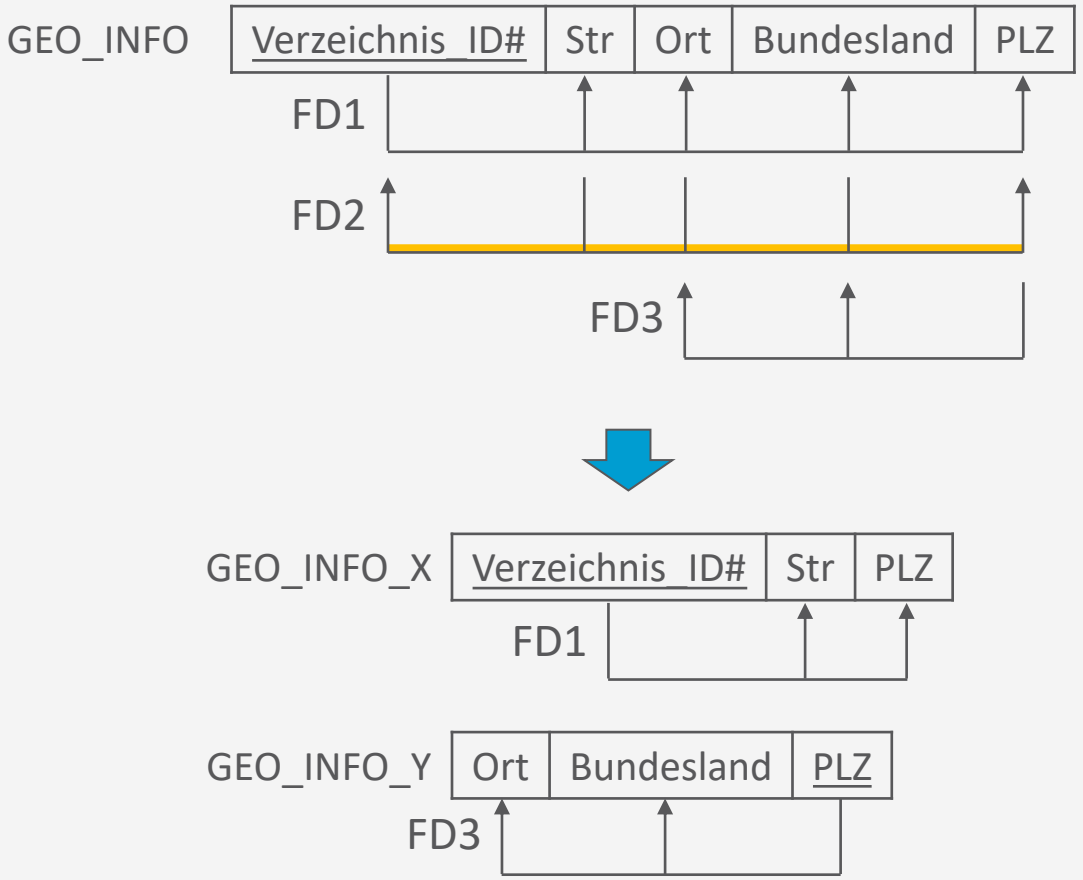
Abhängigkeitswahrung

- Jede in F spezifizierte FD $X \rightarrow Y$
  - Soll direkt in einem der R_i aus der Zerlegung D erscheinen oder
  - (Indirekt) aus den Abhängigkeiten, die in den R_i gelten, abgeleitet werden können
  - Warum ist das wichtig?
    - Wenn Abhängigkeiten verloren gehen, dann ist das Einfügen inkonsistenter Tupel möglich
- Formale Betrachtung
  - Gegeben sei eine Menge von FDs F in R
  - Dann ist die Projektion $\pi_{R_i}(F)$ von F auf R_i (für alle R_i aus D) die Menge von Abhängigkeiten $X \rightarrow Y$ in F^+, für die alle Attribute $X \cup Y$ in R_i vorkommen
  - Die Zerlegung $D = \{R_1, \ldots, R_m\}$ von R heißt in Bezug auf F abhängigkeitswahrend, wenn die Vereinigung der Projektionen von F auf jedes R_i in D mit F äquivalent ist, i.e.,

$$\left(\pi_{R_1}(F) \cup \cdots \cup \pi_{R_m}(F)\right)^+ = F^+$$

Abhängigkeitswahrung: Beispiel 1

- Zerlegung, die nicht alle FDs erhält …
  - Bei der unten gezeigten Zerlegung von GeoInfo in die Relationenschemata GeoInfoX und GeoInfoY geht FD2 verloren
    - (Verzeichnis_ID# könnte z.B. für GPS-Koordinaten stehen)
    - (FD3 stimmt nur unter der Annahme, dass PLZ nicht länder- bzw. ortsübergreifend sind)



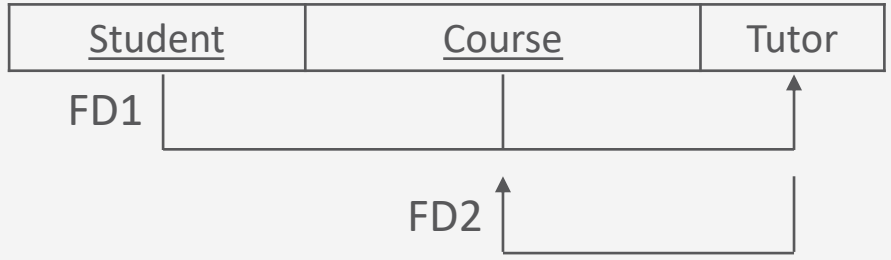
Abhängigkeitswahrung: Beispiel 2

- Gegeben ist die Relation $EXERCISE$ mit den FDs
  - FD1 : $\{Student, Course\} \rightarrow \{Tutor\}$
  - FD2 : $\{Tutor\} \rightarrow \{Course\}$
- Mögliche Zerlegungen
  - $(Student, Tutor)$ und $(Student, Course)$
  - $(Course, Tutor)$ und $(Student, Course)$
  - $(Course, Tutor)$ und $(Student, Tutor)$
  - FD1 geht natürlich bei allen möglichen Zerlegungen verloren, denn man bräuchte für sie alle drei Attribute in genau einem Relationenschema

EXERCISE

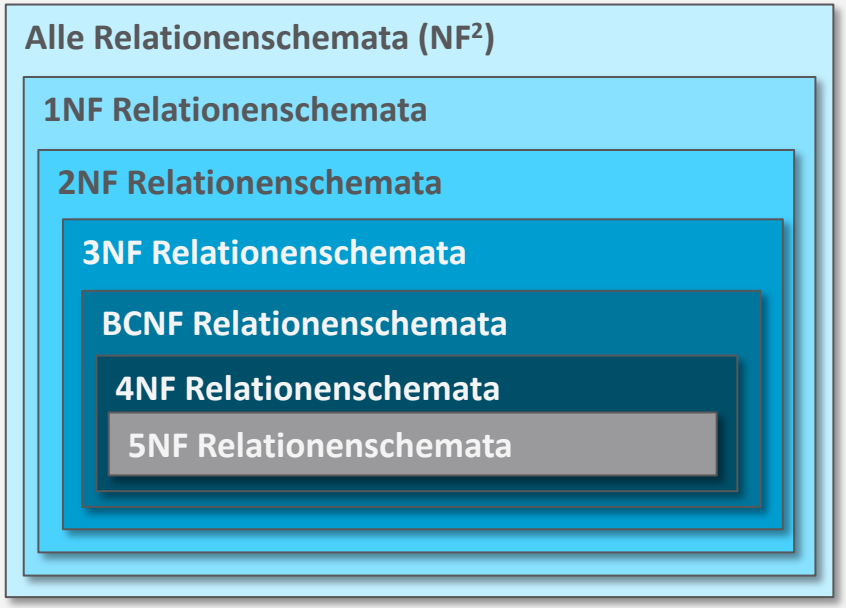
Student	Course	Tutor
Narayan	Information Systems	Mark
Smith	Information Systems	Navathe
Smith	Operating Systems	Ammar
Smith	Formal Languages	Schulz
Wallace	Information Systems	Mark
Wallace	Operating Systems	Ahamad
Wong	Information Systems	Otte
Zelaya	Information Systems	Navathe

EXERCISE



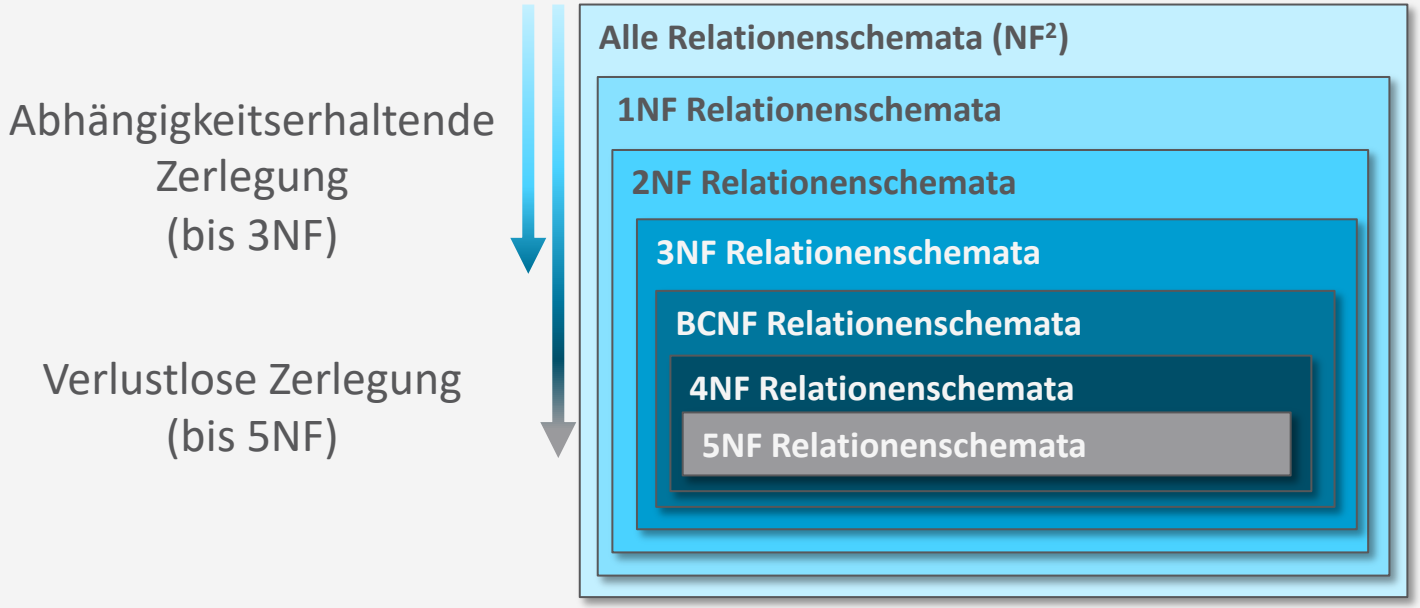
Normalformen

Zerlegungen von Relationen



Historie

- Normalisierungsverfahren: 1972 von Codd vorgeschlagen
  - Unterzieht DB-Schemata einer Reihe von Tests, ob sie einer bestimmten NF genügen
  - Ursprünglich 1. – 3. Normalform (1NF, 2NF, 3NF)
  - Dann Verschärfung der 3. NF: Boyce Codd Normalform (BCNF)
  - Später: 4. und 5. Normalform
- NFs enthalten einander
  - Siehe Abbildung rechts



Nebenbemerkung: Non-first Normal Form (NF²)

- Relationen mit
  - Nicht-atomaren Einträgen
    - Listen
    - Zusammengesetzte Einträge
  - Geschachtelten Relationen

ELTERN

Vater	Mutter	Kinder	
		KName	KAlter
Johann	Martha	Else	5
		Lucie	3
Johann	Maria	Theo	3
		Josef	1
Heinz	Martha	Cleo	9

ADR

Straße	PLZ+Ort
Baumstr	2500 Biel
Parkstr	3000 Bern
Wiesenstr	3018 Bern
Baumstr	3018 Bern

ABT

Abt	AbtNr	ALeitSVNr	AStandort
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

1. Normalform

- Relationenschema R ist in 1. Normalform (1NF), wenn
  - Domänen der Attribute von R nur atomare Werte enthalten
- Erzeugung:
  - Erstelle für jedes nicht-atomare Attribut oder für verschachtelte Schemata ein neues Relationenschema
    - Bei Übersetzung aus einem ER-Diagramm
      - *Zusammengesetzte* Attribute in seinen Einzelteilen als Attribute übernehmen
      - *Mehrwertige* Attribute in eigene Relation auslagern

ADR

Straße	PLZ	Ort
Baumstr	2500	Biel
Parkstr	3000	Bern
Wiesenstr	3018	Bern
Baumstr	3018	Bern

ADR

Straße	PLZ+Ort
Baumstr	2500 Biel
Parkstr	3000 Bern
Wiesenstr	3018 Bern
Baumstr	3018 Bern

ABT

Abt	AbtNr	ALeitSVNr	AStandort
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

ABT

Abt	AbtNr	ALeitSVNr
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

ABTSTAN

AbtNr	AStandort
5	Bellaire
5	Sugarland
5	Houston
4	Stafford
1	Houston

1. Normalform

- Effekt:
  - Keine zusammengesetzten, mengenwertige oder geschachtelten Werte
- Praktischer Nutzen:
  - Erleichtert oder ermöglicht sogar erst Anfragen an die DB durch atomare Einträge
    - Beispiel: Alle Straßen im Ort „Bern“
      - Sonst nicht möglich, da nur Anfragen an alle Straßen in „3000 Bern“, „3018 Bern“ möglich oder mittels aufwendiger Substring-Suche
    - Beispiel: Alle Abteilungen am Standort „Bellaire“
      - Erleichtert, da sonst Suche in Liste für jede Abteilung nötig

ADR

Straße	PLZ	Ort
Baumstr	2500	Biel
Parkstr	3000	Bern
Wiesenstr	3018	Bern
Baumstr	3018	Bern

ADR

Straße	PLZ+Ort
Baumstr	2500 Biel
Parkstr	3000 Bern
Wiesenstr	3018 Bern
Baumstr	3018 Bern

ABT

Abt	AbtNr	ALeitSVNr	AStandort
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

ABT

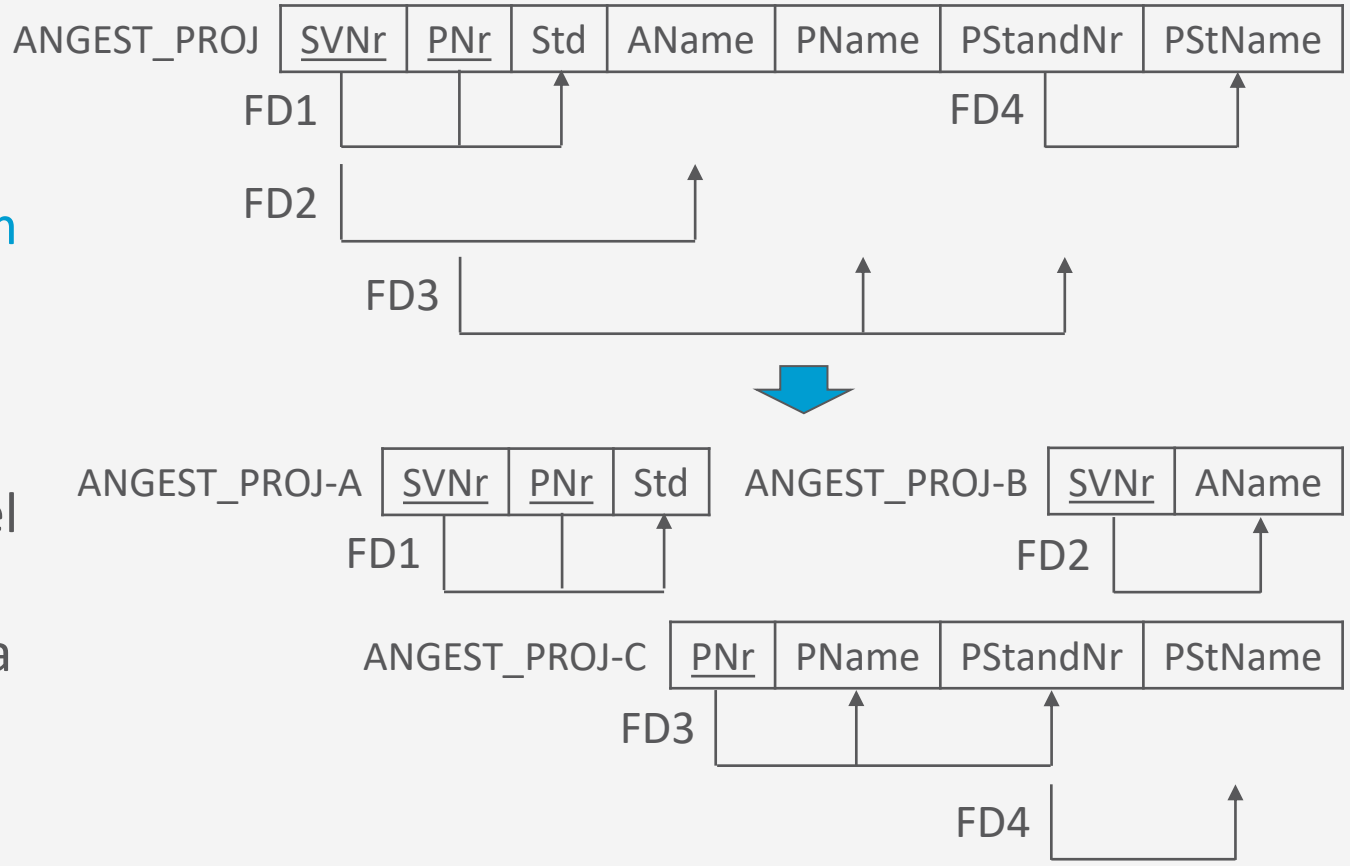
Abt	AbtNr	ALeitSVNr
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

ABTSTAN

AbtNr	AStandort
5	Bellaire
5	Sugarland
5	Houston
4	Stafford
1	Houston

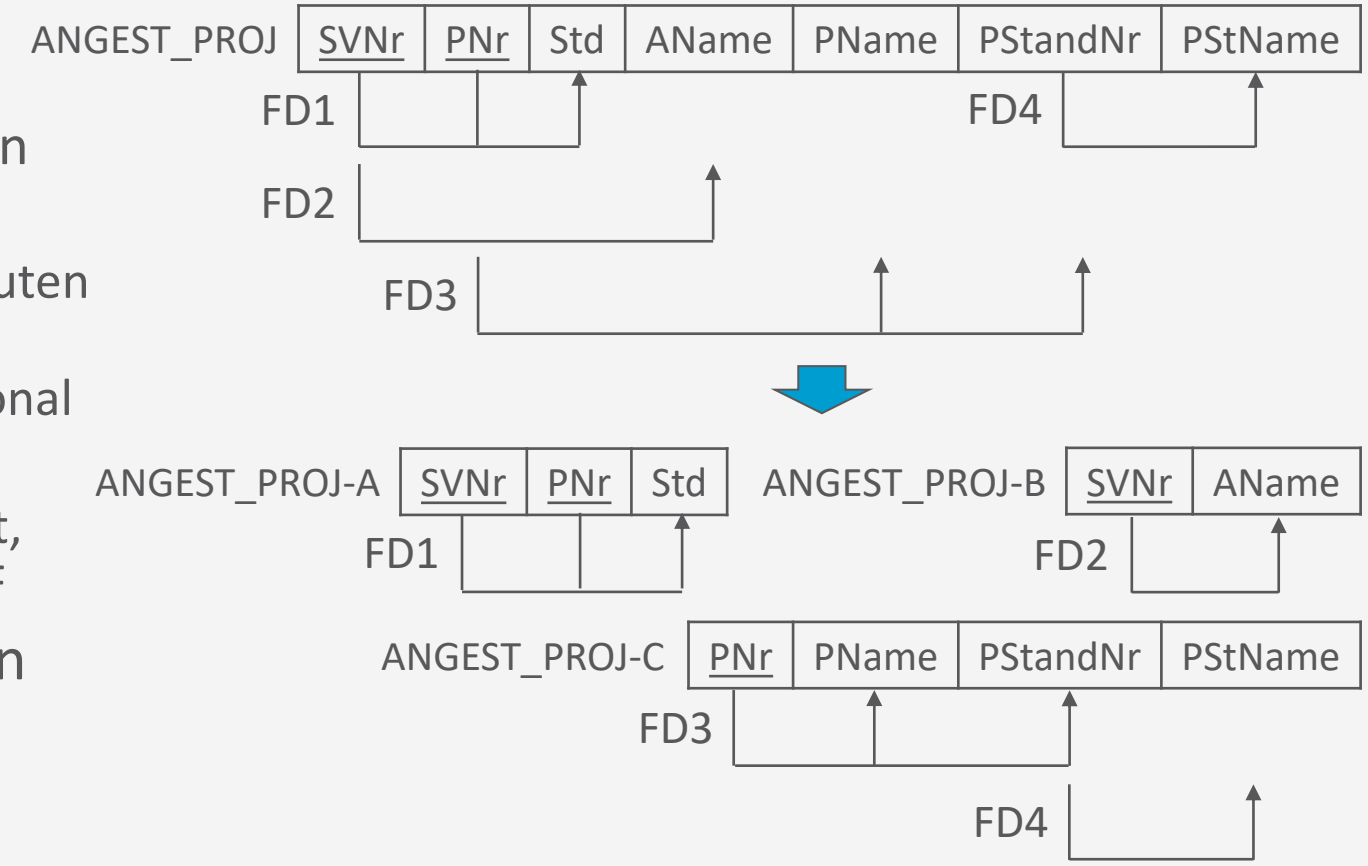
2. Normalform

- Relationenschema R ist in 2. Normalform (2NF), wenn
  - R in 1NF ist und
  - Kein nicht-primes Attribut von nur einem Teil eines Schlüsselkandidaten abhängt
- Erzeugung:
  - Zerlege das Relationenschema und erstelle ein neues für jeden Teil-Schlüssel mit seinen abhängigen Attributen
  - Erhalte das (restliche) Relationenschema mit dem ursprünglichen Primärschlüssel und Attributen, die von diesem voll funktional abhängig sind



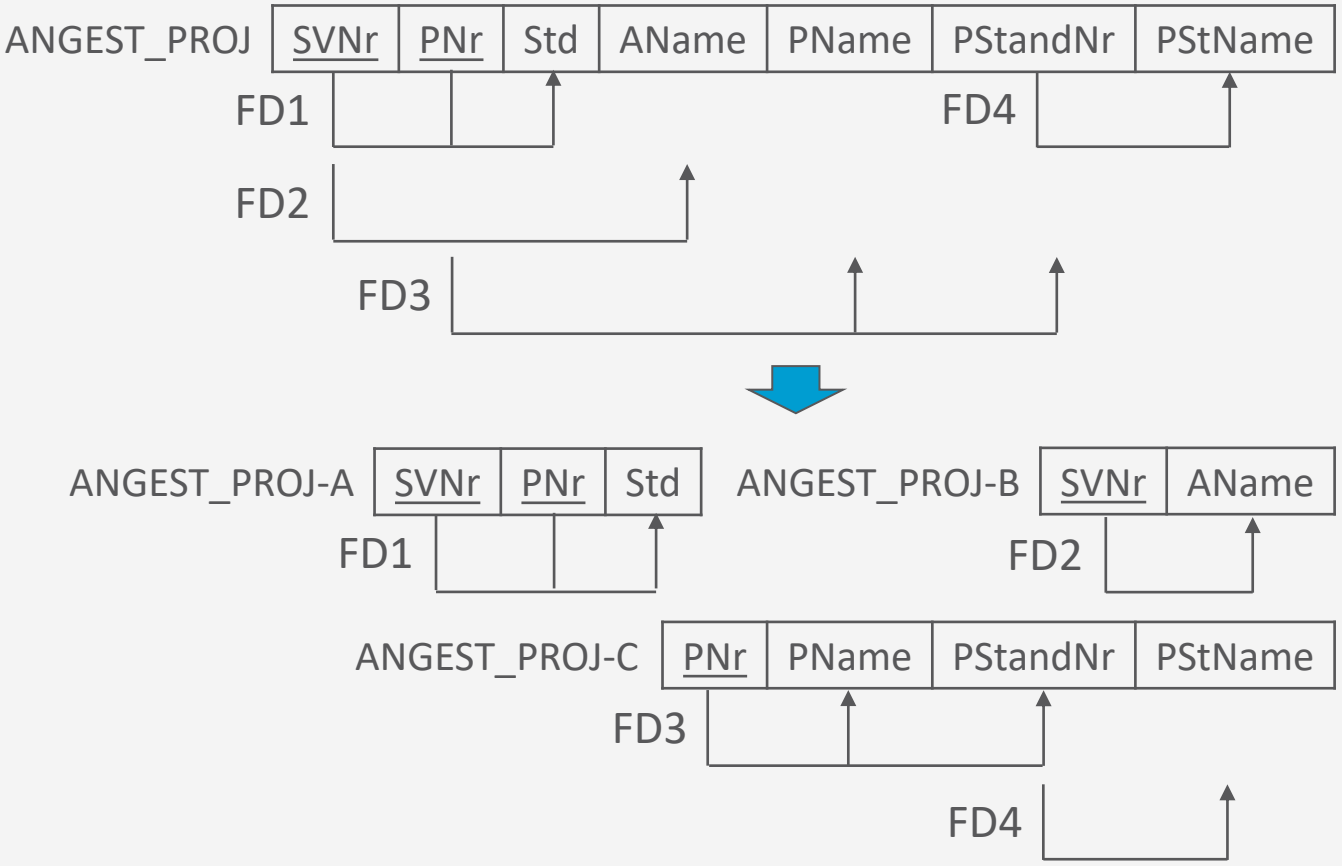
2. Normalform

- Effekt:
  - Alle nicht-primen Attribute sind voll funktional von allen Schlüsselkandidaten abhängig
    - Voll funktional: Von allen Schlüsselattributen (und nicht nur einer Teilmenge eines zusammengesetzten Kandidaten) funktional abhängig
    - Heißt auch, wenn nur ein primes Attribut, dann ist eine Relation in 1NF auch in 2NF
- 2NF bei Erzeugung aus ER-Diagramm in der Regel gegeben



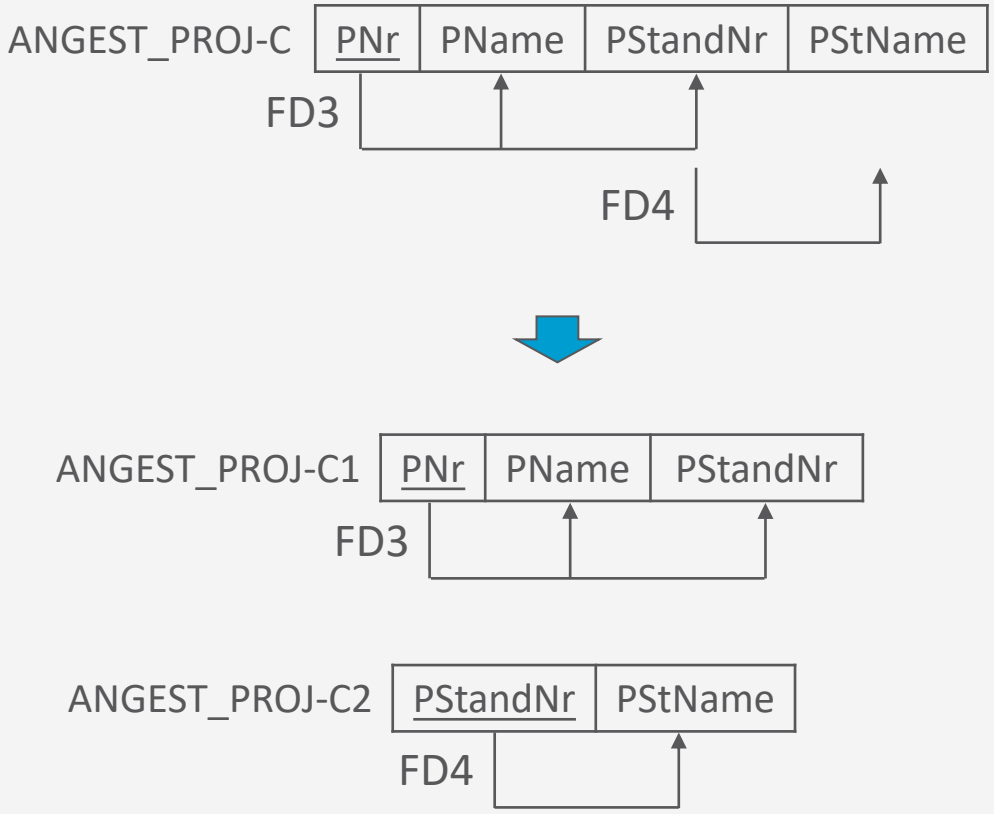
2. Normalform

- Praktischer Nutzen
  - Relationen enthalten logisch zusammengehörige Informationen (*Abgrenzung*)
  - *Reduktion redundanter Werte*
    - Anomalien bei Veränderungen vermeiden
    - Beispiel: $AName$ einmal pro $SVNr$
      - Gleiches gilt für Werte pro PNr
- Probleme:
  - Immer noch redundante Werte möglich, was zu Anomalien führen kann
    - Beispiel: Standortnamen werden jedes Mal für einen Standort wiederholt



3. Normalform

- Relationenschema R ist in 3. Normalform (3NF), wenn
  - R in 2NF ist und
  - Kein nicht-primes Attribut von R transitiv von einem Kandidatenschlüssel abhängt
    - Keine so genannten *transitiven Abhängigkeiten*
- Erzeugung:
  - Zerlege das Relationenschema und erstelle ein neues, welches das nicht-prime Attribut bzw. die nicht-primen Attribute beinhaltet, die funktional von anderen nicht-primen Attributen bestimmt werden
    - Synthesealgorithmus



Synthesealgorithmus

- Ziel: abhängigkeitswahrende, verlustlose Zerlegung $D = \{R_1, \dots, R_m\}$ in Bezug auf eine Menge von FDs F eines Relationenschemas R, so dass jede Relation R_i in 3NF ist
- Input: Universal-Relationenschema R, Menge von FDs F für die Attribute von R
  - Universal-Relationenschema R als eine abstrakte Relation über alle betrachteten Attribute
- Output: Zerlegung $D = \{R_1, \dots, R_m\}$ von R
- 4 Schritte
  1. Bestimme eine kanonische Überdeckung F_C
  2. Erzeuge Relationenschemata für jede FD in F_C
  3. Stelle sicher, dass es ein Schema mit einem Schlüssel für R gibt
  4. Eliminiere doppelte Schemata

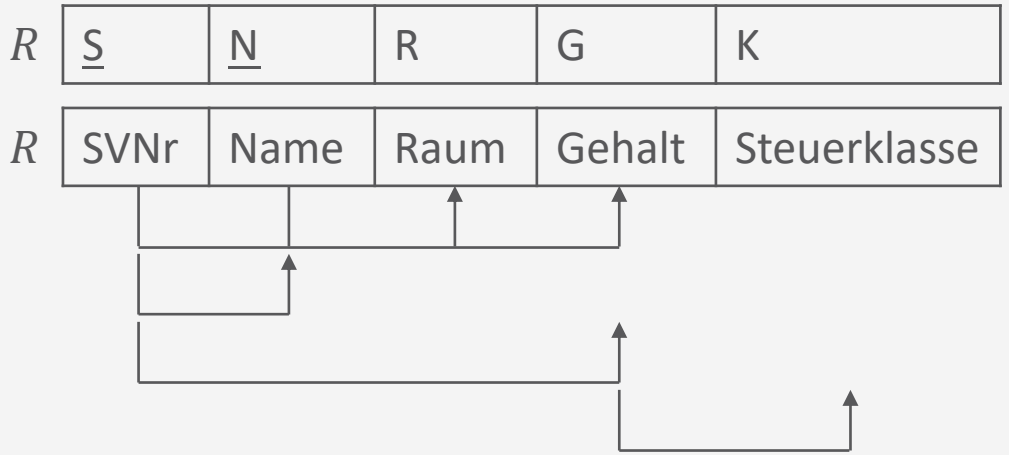
Synthesealgorithmus: Schritte

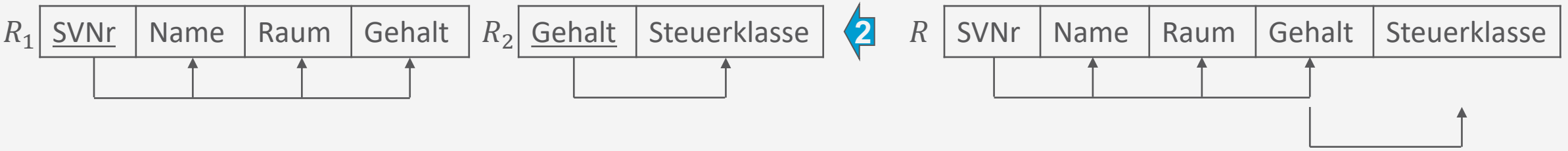
- Input: Universal-Relationenschema R, Menge von FDs F für die Attribute von R
- Output: Zerlegung $D = \{R_1, \dots, R_m\}$ von R
  1. Bestimme eine kanonische Überdeckung F_C für F
     - Linksreduktion, Rechtsreduktion, FDs mit leerem β löschen, FDs mit gleichem α zusammenfassen
  2. Für jede FD $\alpha \rightarrow \beta$ in F_C, erzeuge ein Relationenschema R_α in D mit Attributen $\alpha \cup \beta$
     - Ordne R_α die FDs aus F_C zu, deren Attribute in $\alpha \cup \beta$ sind: $F_\alpha = \{\alpha' \rightarrow \beta' | \alpha' \cup \beta' \subseteq \alpha \cup \beta\}$
     - α ist Schlüssel dieses Relationenschemas
  3. Wenn keines der resultierenden Relationenschemata in D einen Schlüssel von R enthält, dann erzeuge ein weiteres Relationenschema in D, das die Attribute enthält, die einen Schlüssel von R bilden: Wähle Kandidatenschlüssel κ und definiere R_κ über κ mit $F_\kappa = \emptyset$
  4. Eliminiere diejenigen Schemata $R_{\alpha'}$ in D, die in einem anderen Schema R_α aus D enthalten sind, d.h. $R_{\alpha'} \subseteq R_\alpha$

Synthesealgorithmus: Beispiel

- Sei unten stehendes Relationenschema R mit funktionalen Abhängigkeiten gegeben
  - $F = \{S \rightarrow NG, G \rightarrow K, SN \rightarrow RG\}$

1. Bestimme eine kanonische Überdeckung F_C von F
   - Ergebnis aus vorherigen Folien:
     $F_C = \{S \rightarrow NRG, G \rightarrow K\}$
2. Erzeuge Schemata für die FDs in F_C
   - $D = \{R_1, R_2\}$





Synthesealgorithmus: Beispiel

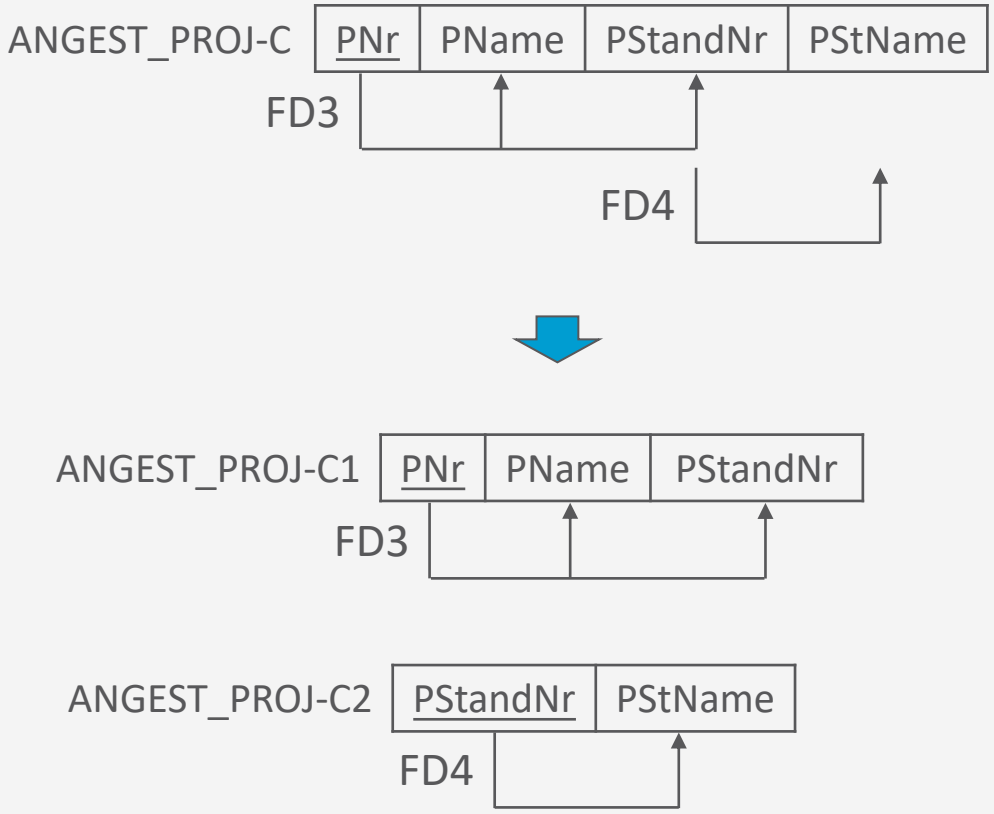
- D nach Schritt 2

R_1	SVNr	Name	Raum	Gehalt	R_2	Gehalt	Steuerklasse

- Schritt 3: Abschließendes Relationenschema bei fehlendem Schlüssel
  - Wenn keines der Relationenschemata in D einen Schlüssel von R enthält, dann erzeuge ein weiteres Relationenschema in D, das Attribute enthält, die einen Schlüssel von R bilden.
    - Betrifft Attribute, die in keiner FD vorkommen
  - Beispiel: Schlüssel von R enthalten (keine Änderungen in D)
- Schritt 4: G reduzieren
  - Eliminiere diejenigen Schemata in D, die in einem anderen Schema aus D enthalten sind
  - Beispiel: R_1 und R_2 sind nicht im jeweils anderen enthalten (keine Änderungen in D)
    - D.h., der Algorithmus terminiert

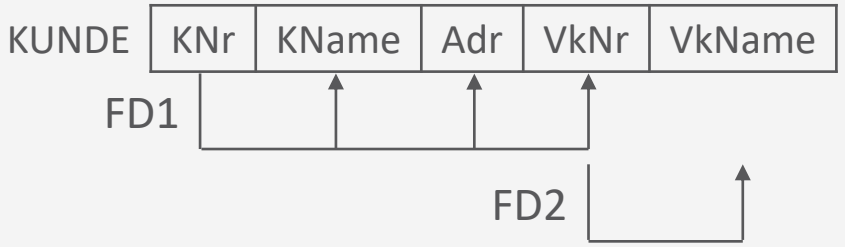
3. Normalform

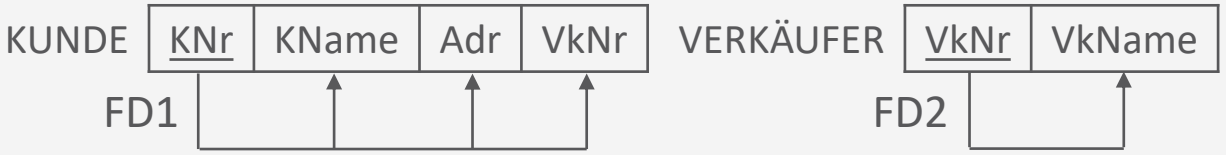
- Effekt:
  - Es gibt keine anderen Nicht-Schlüsselattribute, von denen ein nicht primes Attribut funktional abhängig ist
- Praktischer Nutzen:
  - Verbliebene thematische Durchmischung behoben (*Abgrenzung*)
  - Weitere *Reduktion redundanter Werte*
    - Weitere Vermeidung von Anomalien
    - Beispiel: Standortnamen nur einmal pro Standortnummer
- Die meisten 3NF-Schemata weisen in der Praxis keine dramatischen Probleme auf
  - Sorgfältige Erstellung, z.B., über ER-Diagramm



3. Normalform: Probleme

- Immer noch redundante Werte möglich → Anomalien möglich
  - Beispiel: $KUNDE(KNr, KName, Adr, VkNr, VkName)$
    - Abhängigkeiten:
      - FD1: $\{KNr\} \rightarrow \{KName, Adr, VkNr\}$
      - FD2: $\{VkNr\} \rightarrow \{VKName\}$
    - In 3NF:
      - $KUNDE(KNr, KName, Adr, VkNr)$
      - $VERKÄUFER(VkNr, VkName)$
    - Problem: Folgende Zerlegung ebenfalls in 3NF
      - $KUNDE(KNr, KName, Adr, VkNr)$
      - $VERKÄUFER1(KNr, VkName)$

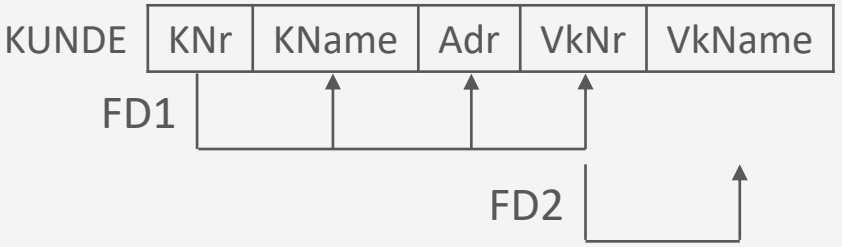


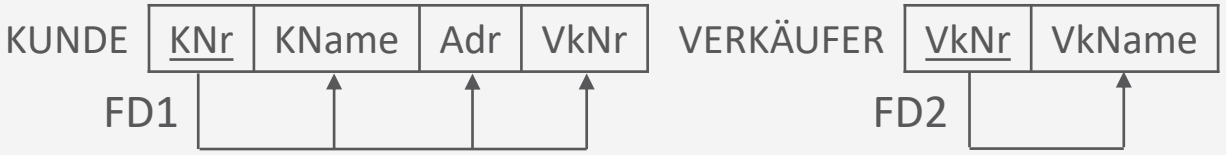


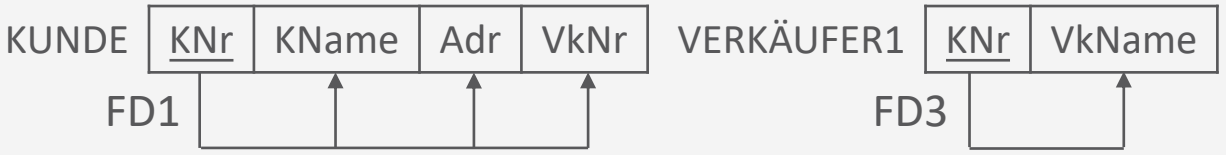
Was für Anomalien können entstehen?

3. Normalform: Probleme

- Fortsetzung Beispiel:
  - Erzeugt u.a. folgende Anomalien:
    - Änderungsaufwand: Änderung eines $VkName$ zu einer $VkNr$
    - Unvollständiges Einfügen: Einfügen eines neuen Verkäufers mit Nummer und Name, der noch keinen Kunden betreut
  - Ursache: FDs $\{KNr\} \rightarrow \{VkNr\}$ und $\{VkNr\} \rightarrow \{VkName\}$ werden auf zwei Relationen „transitiv“ verteilt
    - *IR3*: Wenn
      - $\{\{KNr\} \rightarrow \{VkNr\}, \{VkNr\} \rightarrow \{VKName\}\}$, dann
      - $\{KNr\} \rightarrow \{VKName\}$ (FD3)
  - Reine Erfüllung der NF kein gutes Maß







3. Normalform: Probleme

- Nicht-triviale Abhängigkeiten zwischen Attributen von Schlüsselkandidaten → Erzeugen ebenfalls Redundanzen
  - Beispiel: $RECHNUNG(RNr, KostenArt, KANr, Anzahl)$
    - Abhängigkeit zwischen $KostenArt, KANr$ (1:1-Beziehung)
    - Schlüsselkandidaten: $\{RNr, KostenArt\}, \{RNr, KANr\}$
    - In 3NF, da
      - In 2NF: jedes nicht-prime Attribut ($Anzahl$) ist voll funktional abhängig von allen Attributen der jeweiligen Schlüsselkandidaten
      - Keine transitiven Abhängigkeiten, da kein weiteres nicht-primes Attribut vorhanden
    - Aber:
      - Redundanzen durch Abhängigkeit zwischen $KostenArt, KANr$ → Anomalien möglich, z.B. wenn man eine Nummer zu einer Kostenart ändert

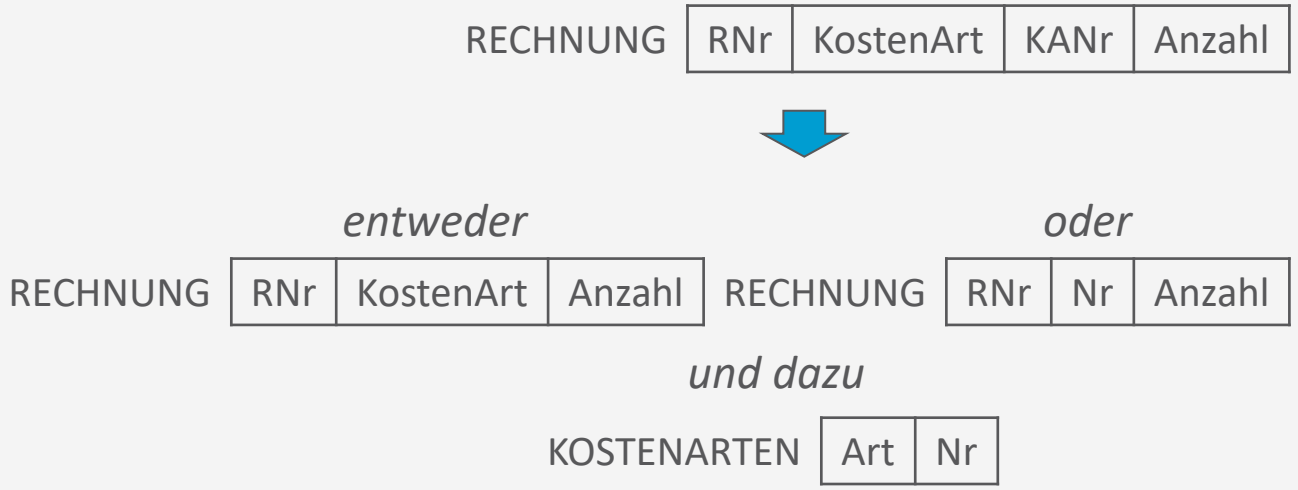
RECHNUNG	RNr	KostenArt	KANr	Anzahl

In 3NF ist es möglich, dass ein Teil eines (zusammengesetzten) Schlüsselkandidaten funktional abhängig ist von einem Teil eines anderen Schlüsselkandidaten.

Boyce-Codd-Normalform (BCNF)

- Relationenschema R ist in Boyce-Codd-Normalform (BCNF), wenn
  - R in 3NF ist und
  - Für jede nicht-triviale funktionale Abhängigkeit $X \rightarrow A$ in R gilt:

    X ist ein Superschlüssel von R
- Erzeugung:
  - Für eine nicht-triviale Abhängigkeit $X \rightarrow A$, die die BCNF verletzt (X kein Superschlüssel), erzeuge eine neue Relation über Attribute X, A mit X als Schlüssel und lösche A aus R
    - Zerlegungsalgorithmus

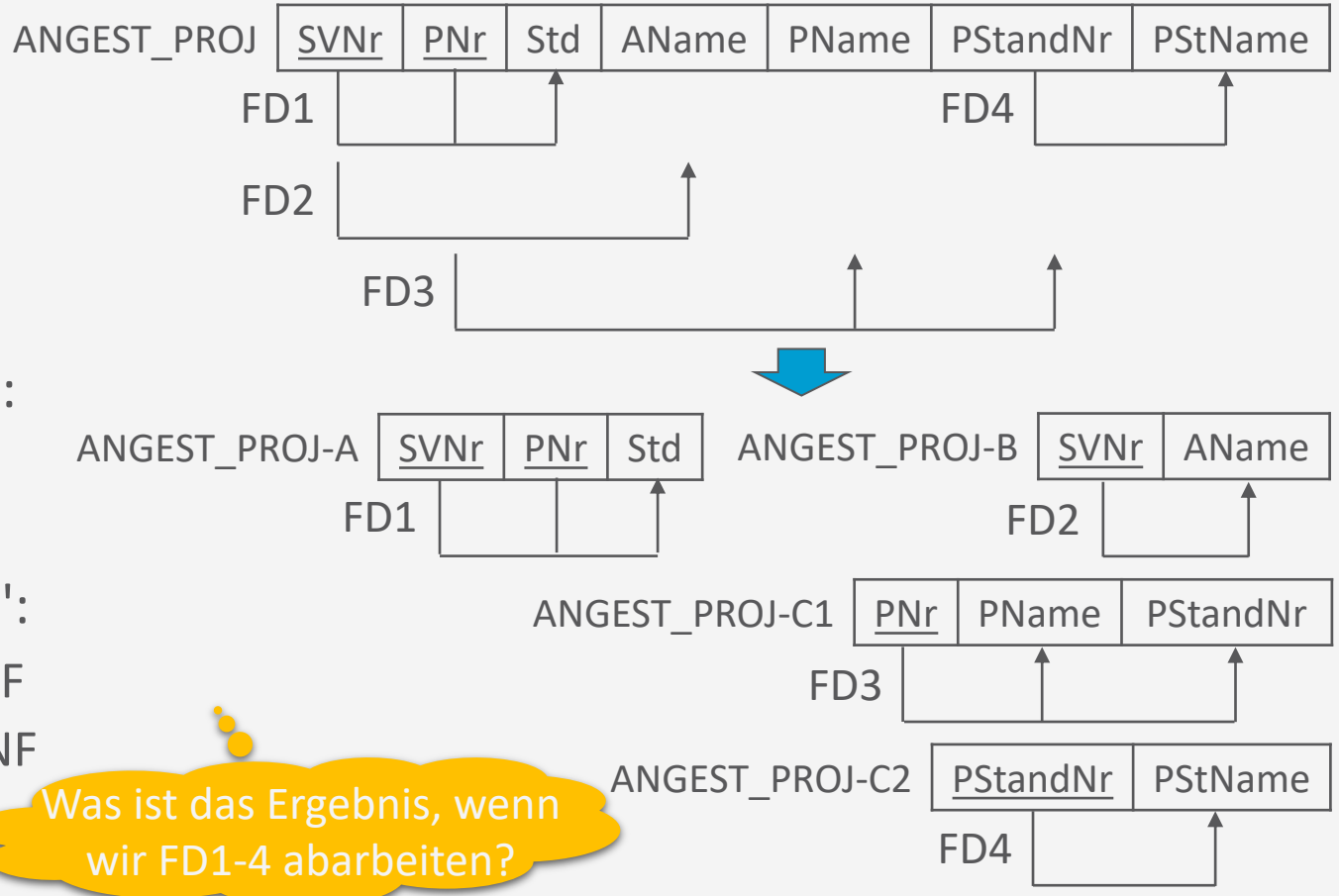


Zerlegungsalgorithmus

- Eingabe: Relationenschema R und Menge von zugehörigen FDs F
- Ausgabe: Zerlegung $D = \{R_1, \dots, R_n\}$, wobei jedes R_i in BCNF ist
- Starte mit $D = \{R\}$
- Solange es noch ein Relationenschema R_i in D gibt, das nicht in BCNF ist:
  - Finde eine FD $\alpha \rightarrow \beta$, die dem R_i zugehörig ist, mit
    1. $\alpha \cap \beta = \emptyset$
    2. $\neg(\alpha \rightarrow R_i)$
    - Man sollte $\alpha \rightarrow \beta$ so wählen, dass β alle von α funktional abhängigen Attribute $B \in (R_i - \alpha)$ enthält, damit der Dekompositionsalgorithmus möglichst schnell terminiert
  - Zerlege R_i in $R_{i1} \leftarrow \alpha \cup \beta$ und $R_{i2} \leftarrow R_i - \beta$
  - Entferne R_i aus D und füge R_{i1} und R_{i2} ein, also
    - $D \leftarrow (D - \{R_i\}) \cup \{R_{i1}, R_{i2}\}$

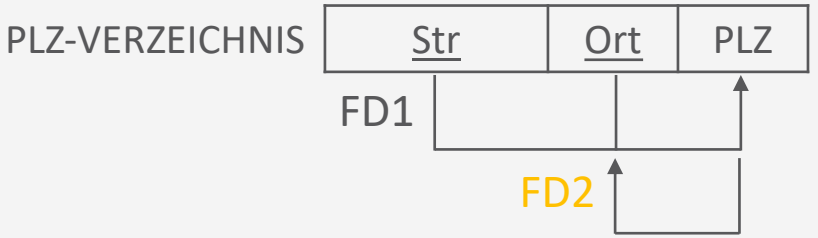
Beispiele

- Zerlegung ANGEST_PROJ
  - FD4 erfüllt $\neg(\alpha \rightarrow R_i)$:
    - ANGEST_PROJ-C2 → in BCNF
    - ANGEST_PROJ'(SVNr, PNr, Std, AName, Pname, PStandNr)
  - FD3 erfüllt $\neg(\alpha \rightarrow R_i)$ in ANGEST_PROJ':
    - ANGEST_PROJ-C1 → in BCNF
    - ANGEST_PROJ''(SVNr, PNr, Std, AName)
  - FD2 erfüllt $\neg(\alpha \rightarrow R_i)$ in ANGEST_PROJ'':
    - ANGEST_PROJ-B(SVNr, AName) → in BCNF
    - ANGEST_PROJ-A(SVNr, PNr, Std) → in BCNF



Boyce-Codd-Normalform (BCNF)

- Relation lässt sich immer in BCNF bringen
  - Dann keine Redundanzen mehr
  - Verlustlos
- Aber: Nicht jedes 3NF Schema in BCNF überführbar bei gleichzeitiger Wahrung der Abhängigkeiten
  - Damit keine effiziente Überprüfung der Abhängigkeiten möglich
    - Beispiel: Man erhält nicht mehr ohne Join die PLZ zu Straße und Ort

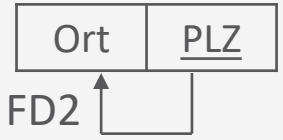


PLZ kein Superschlüssel

Str	Ort	PLZ
Baumstr	Biel	2500
Parkstr	Bern	3000
Wiesenstr	Bern	3018
Baumstr	Bern	3018

FD1 geht verloren!

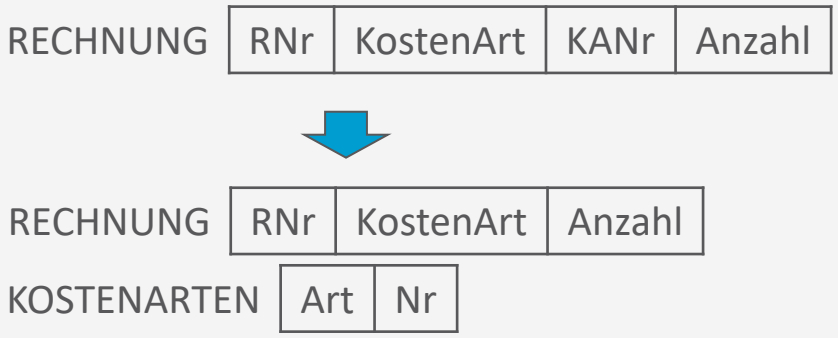
Str	PLZ
Baumstr	2500
Parkstr	3000
Wiesenstr	3018
Baumstr	3018

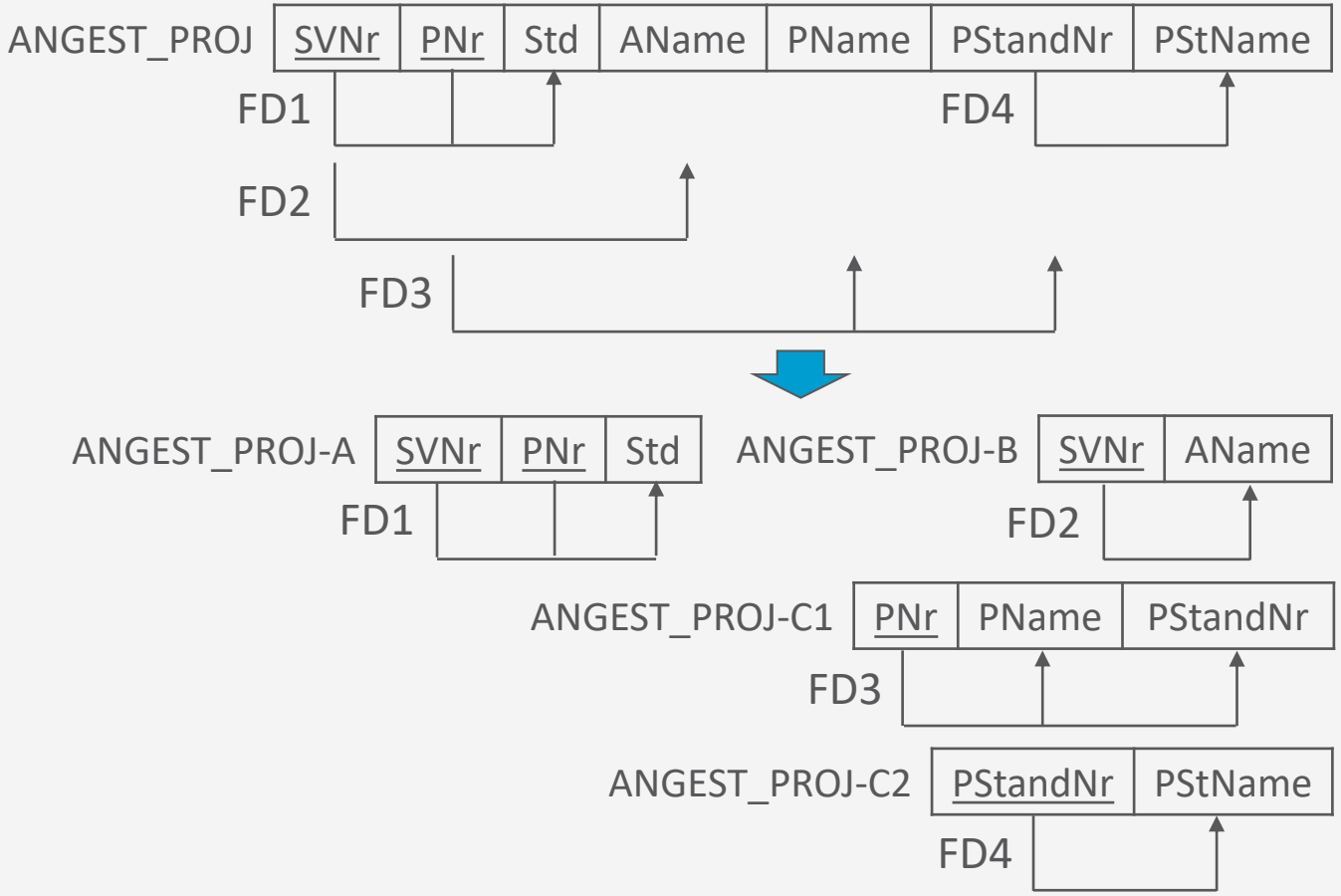


Ort	PLZ
Biel	2500
Bern	3000
Bern	3018

BCNF und 3NF

- Unterschied zwischen 3NF und BCNF nur bei mehreren Schlüsselkandidaten mit überlappenden Attributen
- Relation in BCNF hat garantiert keine Redundanzen
  - Im Gegensatz zu Relation in 3NF





Höhere Normalformen

- 4. Normalform (4NF): Keine mehrwertigen Abhängigkeiten (*multi-valued dependencies*, MVDs)
  - Falls in einem Relationenschema $R(A_1, \ldots, A_n)$ wenigstens zwei 1:n-Beziehungen der Form $A_i: A_j$ und $A_i: A_k$ existieren, bei denen A_j und A_k „unabhängig“ sind, dann kann eine MVD entstehen
    → zu zerlegen für 4NF
  - Beispiel
    - Ein Angestellter arbeitet an Projekten und hat Angehörige. Er kann an mehreren Projekten arbeiten und ebenfalls mehrere Angehörige haben. Projekte und Angehörige sind aber voneinander unabhängig.
    - In $ANGEST$ sind die beiden 1:n-Beziehungen in einer Relation, bei $ANGEST\_PROJ, ANGEST\_ANG$ auf zwei Relationen verteilt

ANGEST

AName	PName	AAName
Smith	P1	John
Smith	P2	Anna
Smith	P1	Anna
Smith	P2	John

ANGEST_PROJ

AName	PName
Smith	P1
Smith	P2

ANGEST_ANG

AName	AAName
Smith	John
Smith	Anna

Höhere Normalformen

- 5. Normalform (5NF)
  - Es gibt Fälle, in denen eine nicht-additive JOIN-Zerlegung eines Schemas R in zwei Relationenschemata nicht möglich ist, aber in drei oder mehr
  - Beispiel:
    - Zerlegung von R in zwei Schemata nicht möglich (unechte Tupel)
    - Nicht-additive JOIN-Zerlegung in R_1, R_2 und R_3 möglich
      - Man sagt auch: „JOIN-Abhängigkeit“ verhindern

R

Repräsentant	Firma	Produkt
Schmidt	Ford	PKW
Schmidt	Ford	LKW
Schmidt	VW	Transporter
Müller	Porsche	PKW
Müller	Ford	PKW

R_1

Repräsentant	Firma
Schmidt	Ford
Schmidt	VW
Müller	Porsche
Müller	Ford

R_2

Repräsentant	Produkt
Schmidt	PKW
Schmidt	LKW
Schmidt	Transporter
Müller	PKW

R_3

Firma	Produkt
Ford	PKW
Ford	LKW
VW	Transporter
Porsche	PKW

Die 5. Normalform – Motivation

- Beispiel (Test 1): JOIN über R_1 und R_2 (Join-Prädikat Repräsentant)

R

	Repräsentant	Firma	Produkt
	Schmidt	Ford	PKW
	Schmidt	Ford	LKW
*	Schmidt	Ford	Transporter
*	Schmidt	VW	PKW
*	Schmidt	VW	LKW
	Schmidt	VW	Transporter
	Müller	Porsche	PKW
	Müller	Ford	PKW

* unechte Tupel

$R_1 * R_2$

R_1

Repräsentant	Firma
Schmidt	Ford
Schmidt	VW
Müller	Porsche
Müller	Ford

R_2

Repräsentant	Produkt
Schmidt	PKW
Schmidt	LKW
Schmidt	Transporter
Müller	PKW

R_3

Firma	Produkt
Ford	PKW
Ford	LKW
VW	Transporter
Porsche	PKW

Die 5. Normalform – Motivation

- Beispiel (Test 2): JOIN über R_1 und R_3 (Join-Prädikat Firma)

R

	Repräsentant	Firma	Produkt
	Schmidt	Ford	PKW
	Schmidt	Ford	LKW
	Schmidt	VW	Transporter
	Müller	Porsche	PKW
	Müller	Ford	PKW
*	Müller	Ford	LKW

* unechte Tupel

$R_1 * R_3$

R_1

Repräsentant	Firma
Schmidt	Ford
Schmidt	VW
Müller	Porsche
Müller	Ford

R_2

Repräsentant	Produkt
Schmidt	PKW
Schmidt	LKW
Schmidt	Transporter
Müller	PKW

R_3

Firma	Produkt
Ford	PKW
Ford	LKW
VW	Transporter
Porsche	PKW

Die 5. Normalform – Motivation

- Beispiel (Test 3): JOIN über R_2 und R_3 (Join-Prädikat Produkt)

R

Repräsentant	Firma	Produkt
Schmidt	Ford	PKW
* Schmidt	Porsche	PKW
Schmidt	Ford	LKW
Schmidt	VW	Transporter
Müller	Porsche	PKW
Müller	Ford	PKW

* unechte Tupel

$R_2 * R_3$

R_1

Repräsentant	Firma
Schmidt	Ford
Schmidt	VW
Müller	Porsche
Müller	Ford

R_2

Repräsentant	Produkt
Schmidt	PKW
Schmidt	LKW
Schmidt	Transporter
Müller	PKW

R_3

Firma	Produkt
Ford	PKW
Ford	LKW
VW	Transporter
Porsche	PKW

Die 5. Normalform – Motivation

- Beispiel (Test 4): JOIN über R_1, R_2 und R_3

R

Repräsentant	Firma	Produkt
Schmidt	Ford	PKW
Schmidt	Ford	LKW
Schmidt	VW	Transporter
Müller	Porsche	PKW
Müller	Ford	PKW

$R_1 * R_2 * R_3$

R_1

Repräsentant	Firma
Schmidt	Ford
Schmidt	VW
Müller	Porsche
Müller	Ford

R_2

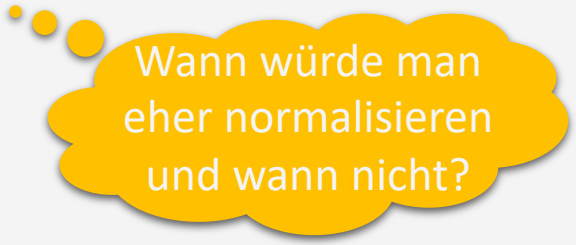
Repräsentant	Produkt
Schmidt	PKW
Schmidt	LKW
Schmidt	Transporter
Müller	PKW

R_3

Firma	Produkt
Ford	PKW
Ford	LKW
VW	Transporter
Porsche	PKW

Normalisierung und die Praxis

- Aus Performance-Gründen manchmal keine Normalisierung

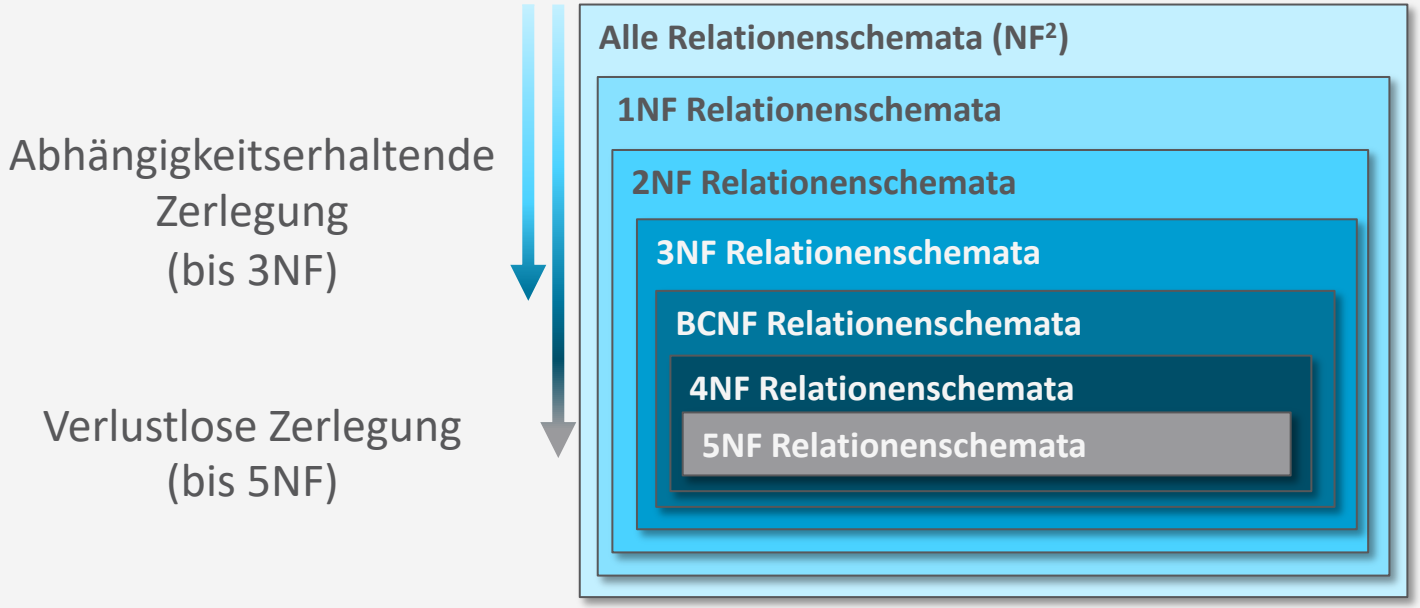


- Gründe gegen Normalisierung, z.B.
  - Zerlegung kann häufige JOINs erfordern
    - Wenn Dinge immer wieder vorkommen, ist es vielleicht sinnvoll die JOIN Tabelle zu speichern, auch wenn es Redundanzen gibt
  - Wenn es kaum Änderungen am Datensatz gibt, dann sind Redundanzen nicht so tragisch

- Gründe für Normalisierung, z.B.
  - Häufiger Änderungen
    - Abwägung zwischen Redundanzfreiheit und Abhängigkeitswahrung, sollte nur eins von beiden gehen

Zwischenzusammenfassung

- Korrektheitskriterien für Zerlegungen: Verlustlose Zerlegung, Abhängigkeitswahrung
  - Dazu: Vermeidung von Redundanzen
- Normalformen
  - (NF^2), 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
  - Synthesealgorithmus liefert verlustfreie, abhängigkeitswahrende Zerlegung mit Schemata in 3NF; Redundanzen möglich
  - Zerlegungsalgorithmus liefert verlustfreie Zerlegung ohne Redundanzen mit Schemata in BCNF, aber nicht immer abhängigkeitswahrend



Überblick: 4. Datenbankentwurf

A. Funktionale Abhängigkeiten
- Definition, Schlüssel, Ableitungen
- Attributhülle, kanonische Überdeckung

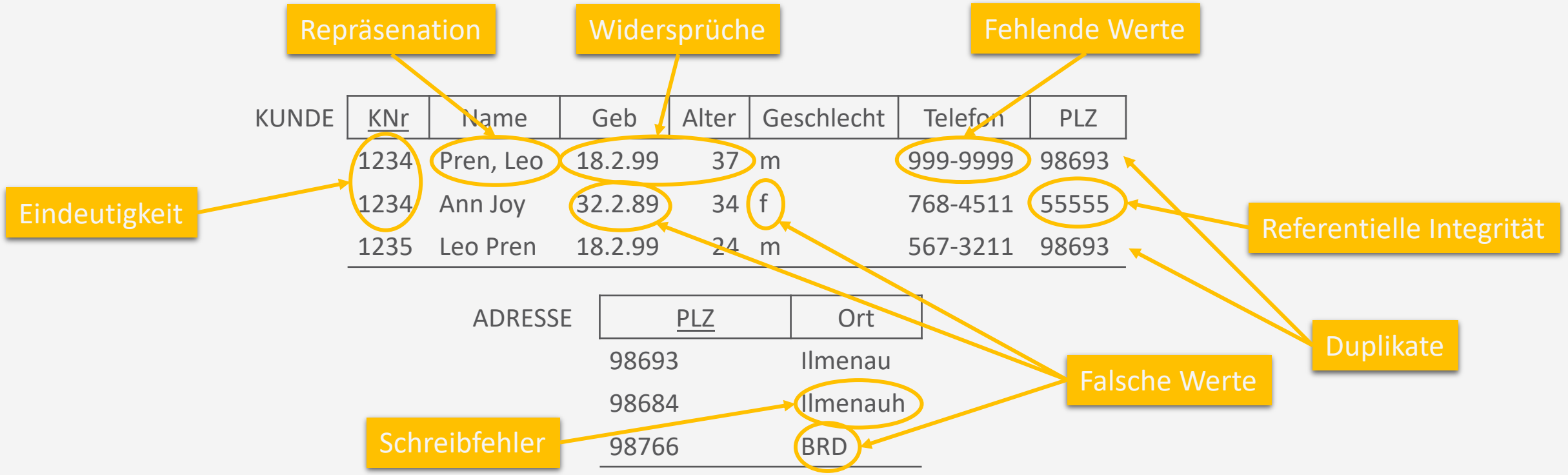
B. Normalformen
- Zerlegung von Relationen
- Exkurs: NF^2; 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
- Synthesealgorithmus, Zerlegungsalgorithmus

C. Datenqualität
- Datenqualitätsprobleme, Dimensionen der Datenqualität

Problemfeld Datenqualität

- Normalformen zielen auf die strukturelle Qualität von Relationenschemata
  - „Intension“
- Datenqualität umfasst Beiträge zur Verbesserung der konkreten „inhaltlichen“ Dateninstanzen auf der Schemaebene
  - „Extension“
- Der Aspekt der Datenqualität ist als problematisch einzuschätzen, wenn Daten vor dem Hintergrund einer Anwendungssituation, z.B.
  - Nicht die angenommene Bedeutung haben
  - Nicht der Spezifikation entsprechen
  - Unverständlich sind

Beispiele für mindere Datenqualität



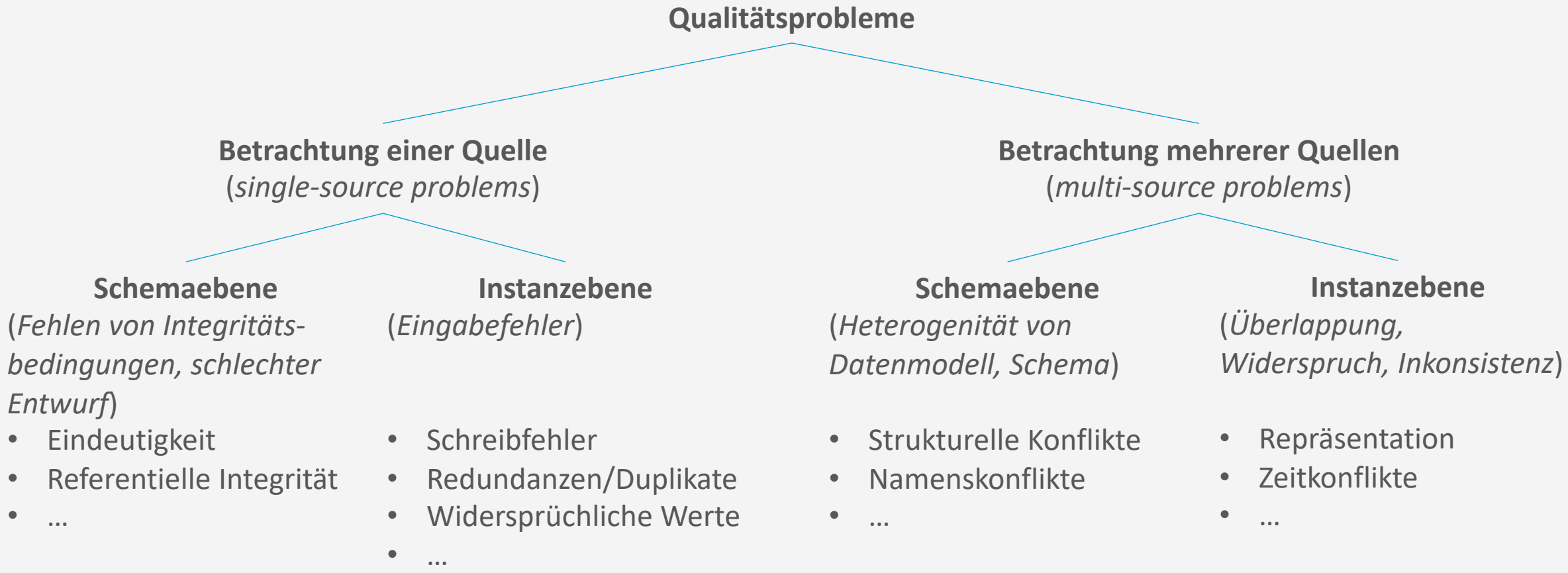
(Quelle: K. Sattler; DB-Tutorientage 2005)

Ursachen für Datenqualitätsprobleme

- Ursachen liegen auf verschiedenen Ebenen, u.a. bei der
  - Datenproduktion, z.B.
    - Verschiedene Quellen repräsentieren gleiche Realwelt-Objekte in unterschiedlicher Form
    - Datenerfassung unterliegt „subjektiven Einflüssen“
    - Systematische Probleme bei Datenerfassung (Messung, Kodierung etc.)
  - Datenspeicherung, z.B.
    - Unterschiedliche oder ungeeignete Formate
  - Datennutzung, z.B.
    - Unzureichende Analyse- und Verarbeitungsmöglichkeiten
    - Veränderung der Nutzungsbedürfnisse
    - Sicherheits- und Zugriffsprobleme

(Quelle: K. Sattler; DB-Tutorientage 2005)

Arten von Datenqualitätsproblemen



(Quelle: K. Sattler; DB-Tutorientage 2005)

Dimensionen der Datenqualität

- Datenqualität wird anhand verschiedener Dimensionen beurteilt, z.B.
  - Vollständigkeit: Verhältnis tatsächlicher Werte zu gespeicherten Werten, u.a.
    - Wertebelegungen verschieden von Null
    - Repräsentation aller in der Realwelt vorkommenden Objekte
  - Genauigkeit: Verhältnis der Anzahl der korrekten Werte zur Gesamtanzahl, d.h. prozentualer Anteil an Daten ohne Datenfehler
    - Umfang, in dem Attributwerte im jeweils „optimalen“ Detaillierungsgrad vorliegen
    - Nähe eines Wertes zum korrekten Wert innerhalb der Realwelt
  - Zeitnähe: Aktualität, in der Attributwerte dem sich dynamisch ändernden Realwelt-Zustand entsprechen
    - Alter: Zeit seit dem Erfassen / Laden der Daten
    - Volatilität: Häufigkeit der Änderungen

(Quelle: K. Sattler; DB-Tutorientage 2005)

Dimensionen der Datenqualität (Fortsetzung)

- Weitere Dimensionen, u.a.
  - Relevanz: Grad, in dem der Informationsgehalt den Nutzungsbedürfnissen entspricht
  - Verständlichkeit: Grad, in dem Daten in Inhalt und Struktur mit der „Vorstellungswelt“ der nutzenden Personen übereinstimmen
  - Konsistenz: Grad, in dem Daten frei von logischen Widersprüchen sind
    - Integritätsbedingungen, Geschäftsregeln, ...
  - Verfügbarkeit: Grad, in dem Daten für nutzende Personen in einem bestimmten Zeitraum nutzbar sind
  - Glaubwürdigkeit: Grad, in dem Daten von nutzenden Personen als korrekt akzeptiert werden
  - Kosten: Preis für Datenzugriff, Anfrage, Datenübertragung, ...

(Quelle: K. Sattler; DB-Tutorientage 2005)

Zwischenzusammenfassung

- Normalformen: strukturelle Qualität, Datenqualität: Qualität der Dateninstanzen
- Ursachen von Datenqualitätsproblemen
  - Verschiedene Quellen, subjektive oder systematische Einflüsse bei der Datenerfassung
  - Unterschiedliche / falsche Formate
  - Veränderte Anforderungen, Sicherheits- und Zugriffsprobleme
- Arten von Datenqualitätsproblemen
  - Eindeutigkeit, referentielle Integrität, Schreibfehler, Redundanzen / Duplikate, Widersprüche
  - Namenskonflikte, strukturelle Konflikte, Repräsentation
- Dimension Beurteilung
  - Vollständigkeit, Genauigkeit, Zeitnähe, Relevanz, Verständlichkeit, Konsistenz, Verfügbarkeit, Glaubwürdigkeit, Kosten

Überblick: 4. Datenbankentwurf

A. *Funktionale Abhängigkeiten*
   - Definition, Schlüssel, Ableitungen
   - Attributhülle, kanonische Überdeckung

B. *Normalformen*
   - Zerlegung von Relationen
   - Exkurs: NF^2; 1NF, 2NF, 3NF, BCNF, 4NF, 5NF
   - Synthesealgorithmus, Zerlegungsalgorithmus

C. *Datenqualität*
   - Datenqualitätsprobleme, Dimensionen der Datenqualität

→ Structured Query Language (SQL)

WWU MÜNSTER

```
select <Attribut- und Funktionsliste>
from <Relationenliste>
[where <Bedingung>]
[group by <Gruppierungsattribut(e)>]
[having <Gruppenbedingung>]
[order by <Attributliste>];
```

Structured Query Language (SQL)

Datenbanken

wissen.leben

Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Unternehmen → DB anlegen

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

ArbeitetAn	ProjNr	SVN	Std

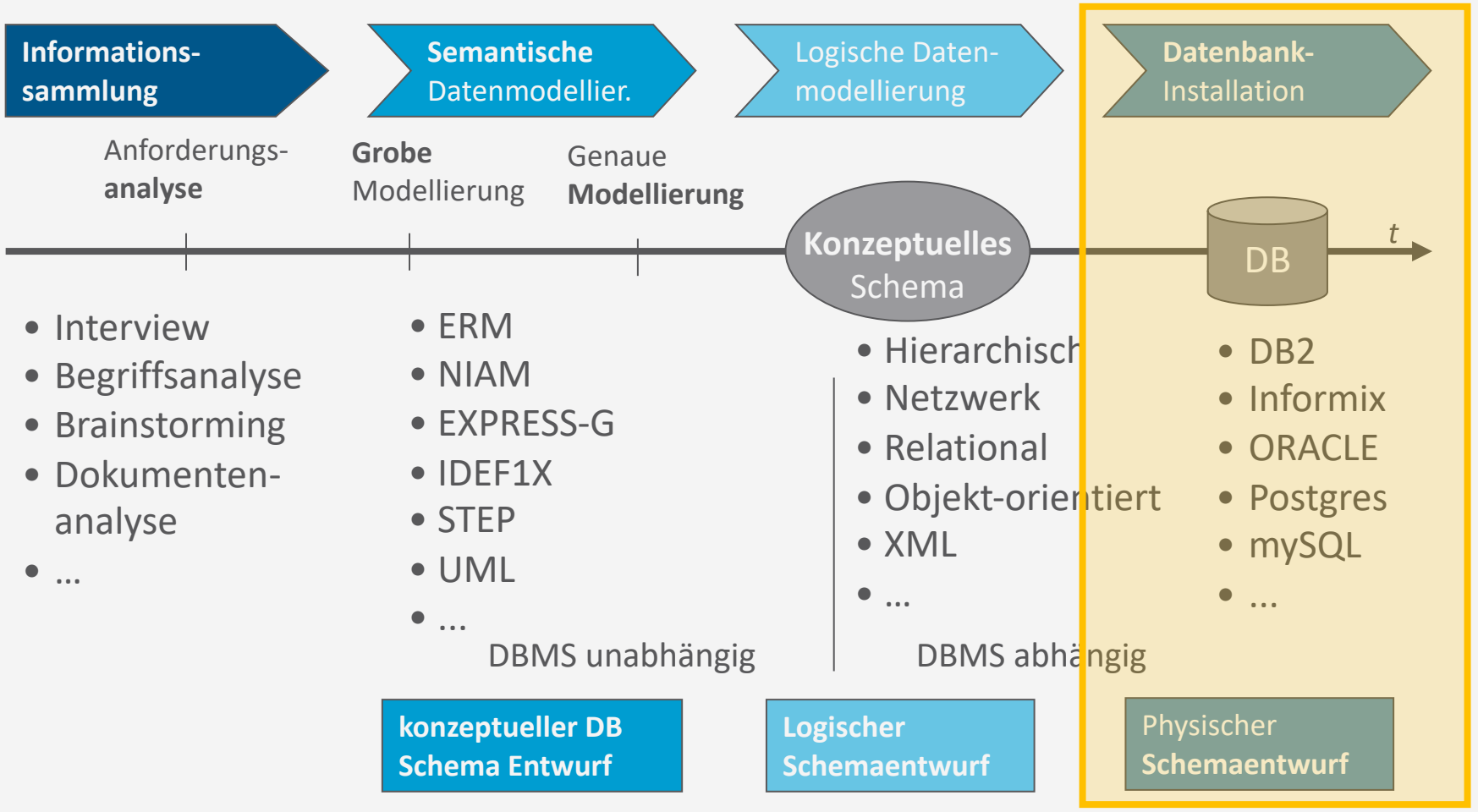
AbtStandort	AbtNr	Standort

Angehörige	Name	Geb	Geschlecht	Grad	SVN

- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



SQL - Historie

- 1974: SEQUEL
  - Erster Vorschlag für die Sprache SQL, Entwicklung durch IBM
  - Implementierung für (experimentelles) relationales DBMS: System-R
- 1983: SQL ist de facto Standard
- 1986: SQL-86 / SQL 1
  - Erster offizieller Standard durch ANSI und ISO
- 1989: SQL-89
  - Revision des ersten Standards
- 1992: SQL-92 / SQL 2
  - zweite, deutlich erweiterte Revision
- 2000: SQL 3
  - mit OO-Konzepten, Multimedia, …
- Weitere Revisionen 2003, 2006 (mit XML), 2008, 2011, 2016 (aktuell)

DB-Sprachen

- Definition von DBs:
  - View Definition Languages (VDLs): extern
  - Data Definition Languages (DDLs): logisch
  - Storage Definition Languages (SDLs): intern

- Zugriff auf DBs (Einfügen, Ändern, Löschen und Anfragen von Datensätzen):
  - Data Manipulation Languages (DMLs)
    - Einfüge-, Änderungs- und Löschoperationen: Updates
    - Reine Anfragen: „Queries“
    - Alle Zugriffsarten: „Manipulation“
  - *Data Control Languages* (DCLs)

SQL : Structured Query Language
VDL, DDL, SDL, DML, DCL in einem

Überblick: 5. Structured Query Language (SQL)

A. ***Datendefinition (SQL als DDL)***
   - Schema, Tabellen, Datentypen, Constraints definieren
   - Strukturelle Änderungen mittels drop, alter

B. *Datenmanipulation (SQL als DML)*
   - Anfragen
   - Datenänderungen

C. *Und der Rest*
   - Sichten (SQL als VDL)
   - Rechtevergabe (SQL als DCL)
   - Programmiermethoden

Das SCHEMA Konstrukt

- SCHEMA: Namensraum in einer Datenbank
  - Die meisten Implementierungen akzeptieren auch DATABASE
- Enthält:
  - Eindeutigen Namen
  - Autorisierungsbezeichner, um Nutzer*innen oder Inhaber*innen des Schemas zu identifizieren
  - Deskriptoren für jedes im Schema enthaltene Element:
    - Relationen
    - Wertebereiche
    - Einschränkungen
    - Sichten
    - Autorisierungsinformationen bzw. Zugriffsrechte
    - etc.

Definition eines SCHEMAs

- CREATE SCHEMA definiert eine Hülle für eine DB:
  - Name: `SchemaName`
  - Autorisierung: `[ authorization ] Authorization`
    - Identifiziert Nutzer*in, dem*r das Schema gehört; wenn nicht genannt, wird der*ie derzeitige Nutzer*in als Inhaber*in gesetzt
  - Mögliche Liste von Elementen in Schema: `SchemaElementDefinition`
    - Liste von Tabellen innerhalb des Schemas
      - Über CREATE TABLE (nächste Folie)

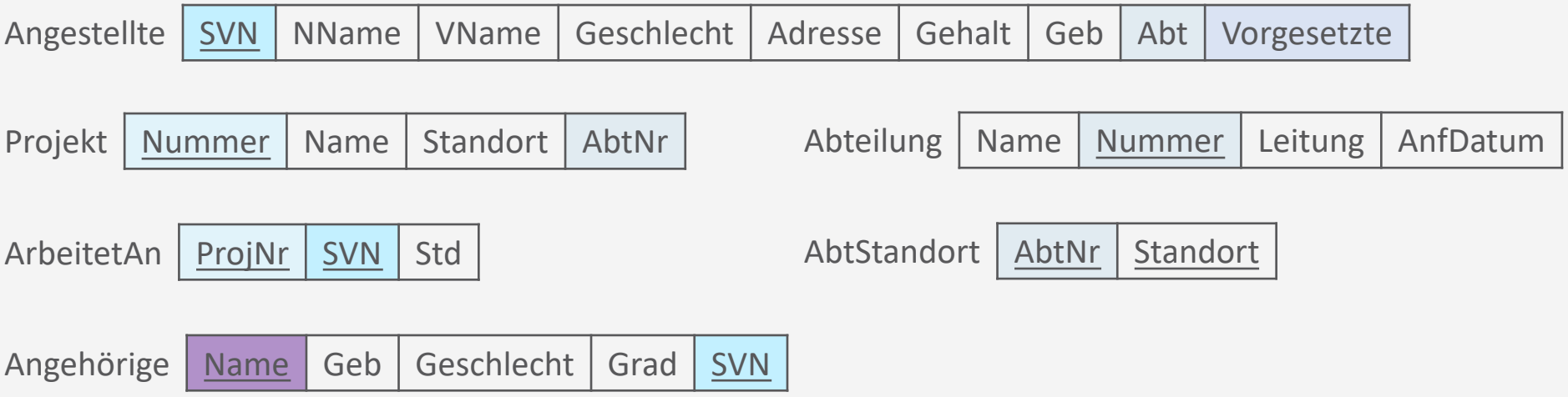
```
create schema [ SchemaName ]
[ [ authorization ] Authorization ]
{ SchemaElementDefinition };
```

Eingeschränkte SQL-Grammatik
(nicht vollständig)

- Beispiele:
  - `create schema Unternehmen authorization JSmith;`
  - `create schema Unternehmen authorization JSmith create table Projekt;`

Relationales Schema: CREATE Beispiele

- Relationales Schema (in 3NF/BCNF): Unternehmen → DB anlegen



Praxis-Test mit DBeaver

- Tabellen definieren

Dbeaver Screenshot

Weitere Online-Ressource:
https://www.w3schools.com/sql/default.asp
Achtung: SQL Befehle teilweise implementierungsabhängig

Beispiele für CREATE TABLE

Einrückung, erweiterter Leeraum dient nur der Übersicht und ist nicht erforderlich.

```
create table Angestellte (
     SVN            char(12)      not null,
     NName          varchar(25)   not null,
     VName          varchar(25)   not null,
     Geschlecht     char,
     Adresse        varchar(60),
     Gehalt         decimal(10,2),
     Geb            date,
     Abt            int           default 1,
     Vorgesetzte    char(12),

     primary key  (SVN),
     foreign key  (Vorgesetzte) references Angestellte(SVN)
);
```

`Abt` ist Fremdschlüssel auf Relation ABTEILUNG, die noch nicht definiert ist → noch nicht definierbar

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Beispiele für CREATE TABLE

```
create table Abteilung (
    Name          varchar(25) not null,
    Nummer        serial,
    Leitung       char(12),
    AnfDatum      date,

    primary key (Nummer),
    unique (Name),
    foreign key (Leitung) references Angestellte(SVN)
);
```

Für eine inkrementell wachsende ID nutzt
- *PostgreSQL* den Datentyp SERIAL, womit ein Int-Attribut angelegt wird, welches mit jeder Einfügung um 1 inkrement wird (Start bei 1)
- *MySQL* den Datentyp INT zusammen mit AUTO_INCREMENT
- *SQL Server* den Datentyp INT zusammen mit IDENTITY(m,n), wobei m=1 der Startwert und n=1 das Inkrement

Abteilung	Name	Nummer	Leitung	AnfDatum

Beispiele für CREATE TABLE

```
create table Angestellte (
    ...
    primary key (SVN),
    foreign key (Vorgesetzte) references Angestellte(SVN)
);
create table Abteilung (
    ...
    primary key (Nummer),
    ...
);
alter table Angestellte
    add foreign key (Abt) references Abteilung(Nummer)
```

Beispiele für CREATE TABLE

```
create table Projekt (
     Name          varchar(25) not null     unique,
     Nummer        int         not null     primary key,
     Standort      varchar(25),
     AbtNr         int         not null     references
                                            Abteilung(Nummer)
);
```

Projekt | Nummer | Name | Standort | AbtNr

Beispiele für CREATE TABLE

```
create table Abteilung (
    ...
    primary key (Nummer),
    ...
);
create table AbtStandort(
    AbtNr       int          not null,
    Standort    varchar(25)  not null,

    primary key (AbtNr, Standort),
    foreign key (AbtNr) references Abteilung(Nummer)
);
```

AbtStandort | AbtNr | Standort

Beispiele für CREATE TABLE

```
create table ArbeitetAn (
    ProjNr   int       not null  references Projekt(Nummer),
    SVN      char(12)  not null  references Angestellte(SVN),
    Std      decimal(3,1),

    primary key (SVN, ProjNr)
);
```

ArbeitetAn | ProjNr | SVN | Std

Beispiele für CREATE TABLE

```
create table Angehoerige (
    Name          varchar(25) not null,
    Geschlecht    char,
    Geb           date,
    Grad          varchar(8),
    SVN           char(12)    not null,

    primary key (SVN, Name),
    foreign key (SVN) references Angestellte(SVN)
);
```

Häufig gilt: keine Umlaute

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Definition von Relationen: CREATE TABLE

- CREATE TABLE spezifiziert eine neue Relation
  - Name der Relation: `TableName`
  - Liste von Attributen: `AttributeList`
    - Name der Attribute: `AttrName`
    - Name der Wertebereiche: `Domain`
  - Intrarelationale Einschränkungen
    - NOT NULL, UNIQUE, PRIMARY KEY
    - DEFAULT-Werte zusätzlich angebbar
    - SERIAL für automatisches inkrementieren eines INT (Start bei 1)
  - Interrelationale Einschränkungen
    - REFERENCES, FOREIGN KEY, CHECK (komplexe Bedingung über mehrere Relationen)

```
create table [ TableName ] [(
   [AttributeList]
   [RConstrList]
)];
```

```
AttrName   Domain   [AConstrList]
```

AConstrList:

```
  not null
| default Value
| unique
| primary key AttrName
| references TableName1(AttrName1)
```

RConstrList:

```
  primary key (AttrNameList)
| unique (AttrNameList)
| foreign key (AttrName)
      references TableName1(AttrName1)
| check [Bedingung]
```

Definition von Relationen: CREATE TABLE

```
create table [ TableName ] [(
   [AttributeList]
   [RConstrList]
)];
```

- Jede Relation ist einem Schema implizit (Defaultschema) oder explizit (Punktnotation) zugeordnet
  - Verwendung (und Erweiterung) eines Schemas über Punktnotation:
    - `create table Unternehmen.Angestellte`

Vordefinierte Datentypen in SQL

Achtung: hier große Unterschiede zwischen DBMS

- Numerische Typen:
  - Ganze Zahlen:
    INTEGER / INT, SMALLINT
  - Reelle Zahlen:
    FLOAT, REAL, DOUBLE PRECISION
    - Approximativ
      - Achtung bei Test auf Gleichheit
  - Formatierte Zahlen:
    DECIMAL(i,j), NUMERIC(i,j)
    - Exakt
    - i und j: Dezimalstellen und Nachkommastellen
  - SERIAL als INT mit Auto-Inkrement
- Text-Typen
  - Zeichenketten mit fester Länge:
    CHAR(n)
    - Bei Input mit weniger als n Zeichen wird aufgefüllt
    - Max. n = 255
  - Zeichenketten mit variabler Länge:
    VARCHAR(n)
    - n maximale Länge; kein Auffüllen
    - Max. n systemabhängig
  - In manchen DBMS in der Zwischenzeit synonym

Vordefinierte Datentypen in SQL

Achtung: hier große Unterschiede zwischen DBMS

- Datum, Zeit, Zeitstempel, Intervall
  - DATE ('YYYY-MM-TT')
  - TIME ('hh:mm:ss[.nnnnnnn]')
  - DATETIMEOFFSET ('YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+|-]hh:mm‘)
  - INTERVAL
    - YEAR, DAY, SECOND…
  - Beispiele
    - `select to_number(to_char(current_date,'J'),'9999999');`
    - `select to_char(to_date('1','J'),'DD-MM-YYYY');`
    - `select date '2020-01-01' + interval '1' day;`

Julianisches Datum für Daten vor 1841, Anzahl an Tagen seit 1. Januar 4712 v.Chr.

dual: In manchen Implementierungen gibt es eine so genannte Default Tabelle **dual** (1 Zeile, 1 Reihe) für Anfragen, die eigentlich keine Tabelle benötigten, wenn das DBMS das FROM Konstrukt zwingend vorsieht

Benutzerdefinierte Datentypen

- Benannte Definitionen (Domains)
  - Können im Schema wie Basisdatentypen verwendet werden
- Warum verwenden?
  - Bessere Lesbarkeit
  - Bessere Wartbarkeit

```
create domain [ DomName ]
    as Domain
   [default Value];
```

- Beispiel:
  - `create domain svn_dom as char(12)`
  - Zusätzlich Default-Wert möglich:
    - `create domain abt_name as char(25) default 'invalid';`

Constraints: Intrarelationale Constraints

- NOT NULL
  - Verbietet die Speicherung von NULL-Werten
- UNIQUE
  - Verlangt Eindeutigkeit von Attributen (NULL max. einmal)
- PRIMARY KEY
  - Verlangt Eindeutigkeit von Attributen (NULL nicht erlaubt)
- CHECK
  - Ermöglicht die Formulierung komplexer Einschränkungen

AConstrList

```
  not null
| default Value
| unique
| primary key AttrName
```

```
  primary key (AttrNameList)
| unique (AttrNameList)
| check [Bedingung]
```

RConstrList

- *Einige SQL Implementierungen:* AUTO_INCREMENT
  - Eindeutige ID beginnend bei 1, erhöht sich automatisch mit jedem INSERT

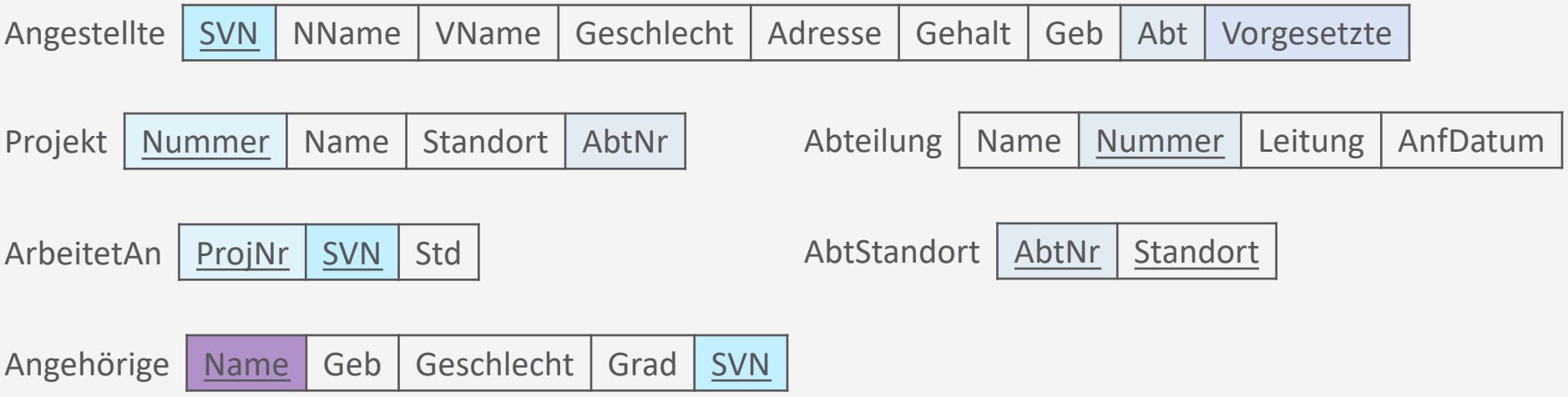
Constraints: Interrelationale Constraints

- REFERENCES und FOREIGN KEY
  - Spezifikation von Bedingungen zur referenziellen Integrität:
  - Auch für einzelne Attribute nach der Domain-Angabe:
    - REFERENCES Relation (Attrib)
  - Für mehrere Attribute:
    - FOREIGN KEY(Attrib1, Attrib2)
      REFERENCES Relation (Attrib1, Attrib2)

```
references TableName1(AttrName1)
```

```
foreign key (AttrName)
   references TableName1(AttrName1)
```

Constraints: Namen

- CONSTRAINT
  - Erlaubt intra- und interrelationalen Constraints eigene Namen zu geben
  - Damit leichter änderbar oder löschbar
  - Beispiel:
    - ```
      constraint   Angest_PK
      primary key (SVN),

      constraint   Angest_Vorgesetzt_FK
      foreign key (Vorgesetzte) references Angestellte(SVN)
      ```
      ```

Constraints: Reaktion auf Constraintverletzung

- Um Constraintverletzungen bei Änderungen zu vermeiden, Aktion spezifizieren
  - CASCADE
    - Propagiert die durchgeführte Änderung in die referenzierende Relation
  - SET NULL
    - Setzt die (wertebasierte) Referenz auf NULL
  - SET DEFAULT
    - Setzt die (wertebasierte) Referenz auf den für das Attribut vorgesehenen Default-Wert.
  - NO ACTION (manchmal auch: RESTRICT)
    - Verbietet Änderungen in einer referenzierten Relation – solange Abhängigkeiten bestehen
- Für DELETE und UPDATE getrennt spezifizierbar:
  - ON DELETE [Action]
    ON UPDATE [Action]

Constraints: Beispiel 1

```
create table Abteilung (
    Abt             int             not null    default 1,
    ...,
    constraint PK__Angest PRIMARY KEY (SVN),
    constraint FK__Angest_VorgesSVN
       foreign key (Vorgesetzte) references Angestellte(SVN)
       on delete set null
       on update cascade,
    constraint FK__Angest_Abt
       foreign key (Abt) references Abteilung(Nummer)
       on delete set default
);
```

Constraints: Beispiel 2

```
create table Abteilung (
    ...,
    Leitung      char(12)    not null    default '000000000000',
    ...,
    constraint PK__Abt
       primary key (Nummer),
    constraint UQ__AbtName
       unique (Name),
    constraint FK__Abt_Mgr
       foreign key (Leitung) references Angestellte(SVN)
       on delete set default
);
```

Constraints: Beispiel 3

```
create table AbtStandort (
    ...,
    primary key (AbtNummer, AStandort),
    ...,
    constraint FK__Stand_Abt_Nummer
       foreign key (AbtNummer) references Abteilung(AbtNummer)
       on delete cascade
);
```

- Oder nach der Definition der Tabelle(n)

```
alter table Angestellte
add constraint
      Angest_CK check (Gehalt/168 > 11.50);  -- Mindestlohn
```

Katalog: INFORMATION_SCHEMA

- Katalog eines DBMS
  - Beinhaltet Informationen über
    - Namen und Größe von Dateien
    - Namen und Datentypen von Datenelementen
    - Speicherdetails für jede Datei
    - Mappinginformation zwischen Schemata
    - Constraints

- SQL: Spezielles, vordefininiertes Schema **`information_schema`**
  - Enthält Metadaten zur Datenbank:
    - Welche Schemata
    - Welche Tabellen
    - Welche Attribute
    - …
  - Können auch mit SQL abgefragt werden, z.B.:
    - `select * from information_schema.tables;`

Schemata ändern

ALTER, DELETE

Löschen von DB-Konstrukten: DROP

- DROP [SCHEMA | TABLE | VIEW] Name;
  - Löscht ein Schema / Relation / Sicht
- ALTER TABLE TableName
  DROP [COLUMN | CONSTRAINT] Name;
  - Löscht eine Spalte / Einschränkung einer Tabelle

```
drop [schema | table | view] Name;
```

```
alter table TableName
drop [column | constraint | view] Name;
```

- Liste von Objekten möglich
- Kann durch CASCADE oder RESTRICT kontrolliert werden

Sichten (VIEWs) kommen später dran,
da wir dafür SQL Queries brauchen.

Ändern einer Relation: ALTER

- Mögliche Änderungen:
  - Hinzufügen eines neuen Attributs
  - Entfernen eines Attributs
  - Ändern einer Attributdefinition
  - Hinzufügen von zusätzlichen Relationeneinschränkungen
  - Entfernen von Relationeneinschränkungen
- Werte für neue Attribute:
  - Default-Wert, manuelle Zuweisung, NULL

```
alter TableName
[add AttrName Domain]
[alter AttrName drop constr]
[rename TableName1]
```

Ändern einer Relation: ALTER – Beispiele

- Beispiele:
  - `alter table Unternehmen.Angestellte add Job varchar(12);`
  - `alter table Unternehmen.Angestellte alter VorgesSVN drop default;`
  - `alter table Unternehmen.Angestellte rename Unternehmen.Mitarbeitende;`
  - Vorherige Folien: Primärschlüssel / CHECK-Constraint hinzugefügt, Attribut gelöscht

```
alter TableName
[add AttrName Domain]
[alter AttrName drop constr]
[rename TableName1]
```

Zwischenzusammenfassung

- Schema als Menge von Relationen: CREATE SCHEMA
- Relationen: CREATE TABLE
  - Liste von Attributen mit Wertebereichen und Constraints
- Wertebereiche
  - Basisdatentypen: INT, FLOAT, …, CHAR(n), VARCHAR(n), DATE, …
  - Benutzerdefiniert zur besseren Lesbarkeit / Wartbarkeit: CREATE DOMAIN
- Constraints
  - UNIQUE, NOT NULL, DEFAULT, AUTO_INCREMENT, PRIMARY KEY, FOREIGN KEY … REFERENCES, CHECK
  - Namen für Constraints über CONSTRAINT für Lesbarkeit / Wartbarkeit; Aktionen zur Vermeidung von Constraintverletzungen: CASCADE, SET NULL, SET DEFAULT, NO ACTION
- Strukturelle Änderungen: DROP, ALTER

Praxis-Test mit DBeaver

- Tabellen ändern

Dbeaver Screenshot

Weitere Online-Ressource:
https://www.w3schools.com/sql/default.asp
Achtung: SQL Befehle teilweise implementierungsabhängig

Überblick: 5. Structured Query Language (SQL)

A. Datendefinition (SQL als DDL)
- Schema, Tabellen, Datentypen, Constraints definieren
- Strukturelle Änderungen mittels drop, alter

B. Datenmanipulation (SQL als DML)
- Anfragen
- Datenänderungen

C. Und der Rest
- Sichten (SQL als VDL)
- Rechtevergabe (SQL als DCL)
- Programmiermethoden

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Firma → Anlegen



- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Anfragen: SELECT-Anweisung

$R \bowtie_{\langle AttrList \rangle} S$
$R \times S$

$\pi_{\langle AttrList \rangle}(R)$

```
select <Attribut- und Funktionsliste>
from <Relationenliste>
[where <Bedingung>]
[group by <Gruppierungsattribut(e)>]
[having <Gruppenbedingung>]
[order by <Attributliste>];
```

$\sigma_{\langle Bedingung \rangle}(R)$
$R \bowtie_{\langle Bedingung \rangle} S$

$\langle Gruppierungsattribute \rangle \mathcal{F}_{\langle Funktion\ Attribut \rangle}(R)$

SELECT …
FROM …
WHERE …

Selektion, Projektion
Mengeneigenschaft und Reihenfolge

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

SELECT: Grundstruktur

- Umsetzung der Projektion (π)
  - RA: $\pi_{\langle Attributliste \rangle}(R)$
  - SQL: SELECT$\langle Attributliste \rangle$ FROM R;
    - Wählt die Attribute aus der Relation aus, die in der Liste genannt sind
    - Sonderzeichen: * wählt alle Attribute einer Relation (keine Projektion)
  - Achtung: In SQL keine automatische Duplikatseliminierung

```
select List
from Table;
```

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

SELECT: Grundstruktur – Beispiele

```
select List
from Table;
```

- Liefere alle Attributausprägungen der Relation ANGESTELLTE, i.e., den Relationenzustand:
  - $\pi_{SVN,NName,VName,Geschlecht,Adresse,Gehalt,Geb,Abt,Vorgesetzte}(Angestellte)$
  - ```
    select *
    from Angestellte;
    ```
- Liefere Vornamen und Nachnamen aller Angestellten der Firma:
  - $\pi_{VName,NName}(Angestellte)$
  - ```
    select VName, NName
    from Angestellte;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

WHERE-Klausel

- Umsetzung der Selektion (σ) mit anschließender Projektion:
  - RA: $\sigma_{\langle Bedingung \rangle}(R)$
  - SQL: SELECT *
    FROM R
    [WHERE $\langle Bedingung \rangle$];
    - Mit anschließender Projektion, i.e., $\pi_{\langle Attributliste \rangle}\left(\sigma_{\langle Bedingung \rangle}(R)\right)$: $\langle Attributliste \rangle$ anstatt *
  - Beispiel: Liefere Geburtsdatum und Adresse der Angestellten mit Namen John Smith:
    - $\pi_{Geb,Adresse}\left(\sigma_{VName=„John“ \wedge NName=„Smith“}(Angestellte)\right)$
    - ```
      select Geb, Adresse
      from Angestellte
      where VName='John' and NName='Smith';
      ```

```
select List
from Table
[where <Condition>];
```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
```

Relationen als Mengen in SQL

- SQL behandelt Relationen als Multimengen
  - Effekt: Keine automatische Duplikatsentfernung
  - Wenn gewünscht: SELECT DISTINCT $\langle Attributliste \rangle$…
- Gründe
  - Entfernung von Duplikaten ist eine teure Operation
    - Möglichkeit der Implementierung: alle Tupel zunächst zu sortieren, anschließend/dabei Duplikate entfernen
  - Oft nicht nötig
    - Viele praktische Anwendungen von DBs sind so, dass Nutzer*innen Duplikate in der Ergebnismenge wünschen
  - Manchmal falsch
    - Aggregatfunktionen: Duplikatseliminierung könnte Ergebnis verfälschen zu intendiertem Ergebnis

```
select [distinct] List
from Table
[where <Condition>];
```

Gründe für keine automatische Duplikatsentfernung?

Ordnen durch ORDER BY

- Resultatmenge kann für die Ausgabe sortiert werden
  - SELECT … ORDER BY <Attribut> ASC
  - SELECT … ORDER BY <Attribut> DESC
  - ASC / DESC muss nicht angegeben werden:
    - SELECT … ORDER BY <Attribut>
    - Dann greift ein Default-Wert: ASC
      - Wenn nichts angegeben ist, wird aufsteigend sortiert
  - Ordnung auf <Attribut> muss definiert sein
  - Auch mehrere Sortierkriterien möglich

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

Ordnen durch ORDER BY: Beispiel

```
select [distinct] List
from Table
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Gebe für alle Angestellte deren Nachname, Vorname und Abteilung sortiert nach Abteilungsnummer, dann Nachname und dann Vorname (lexikographisch aufsteigend) an:
  - ```
    select Abt, NName, VName
    from Angestellte
    order by Abt, NName, VName;
    ```
- Bei absteigender Sortierung der Abteilungsnamen und aufsteigender Sortierung des Rests:
  - ```
    select Abt, NName, VName
    from Angestellte ang
    order by AName DESC, NName ASC, VName ASC;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Relationen kombinieren 1

Kombinierende Operationen: Kartesisches Produkt, Join und Join-Arten, klassische Mengenoperationen (Vereinigung, Schnitt, Differenz)

Umbenennung

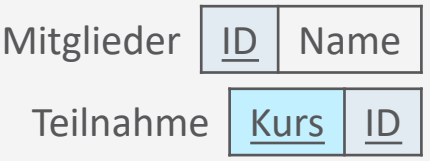
Relationen kombinieren: Kartesisches Produkt

- Umsetzung des kartesischen Produkts (×) durch Auflistung der beteiligte Tabellen in FROM
  - RA: $R \times S$
  - SQL: SELECT *
    FROM R, S;

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Beispiel:
  - Liefere alle möglichen Kombinationen von Nachnamen und Abteilungsnamen der Firma geordnet erst nach Abteilung und dann nach Nachname:
    - $\pi_{NName,Name}(ANGESTELLTE \times ABTEILUNG)$
    - ```
      select distinct Name, NName
      from Angestellte, Abteilung
      order by Name, NName;
      ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Relationen kombinieren: JOIN

- Umsetzung des JOIN (⋈) über zwei Wege möglich:
  1. Implizit über kartesisches Produkt:
     - RA: $\sigma_{\langle Bedingung \rangle}(R \times S)$
     - SQL: SELECT *
       FROM R, S
       WHERE $\langle Bedingung \rangle$;
       - Effiziente Umsetzung als JOIN durch DBMS, auch wenn als $\sigma_{\langle Bedingung \rangle}(R \times S)$ formuliert
  2. Explizit über Schlüsselworte [INNER] JOIN mit Spezifizierung der JOIN-Bedingung über ON:
     - RA: $R \bowtie_{\langle Bedingung \rangle} S$
     - SQL: SELECT *
       FROM R INNER JOIN S ON $\langle Bedingung \rangle$;
- Häufig Verknüpfung über Fremdschlüssel

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

Abteilung

Name	Nummer	Leitung	AnfDatum

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Relationen kombinieren: JOIN – Beispiele

- Gebe die Nachnamen aller Angestellten mit dem Namen der zugeordneten Abteilung geordnet erst nach Abteilung und dann nach Nachname aus

```
select [distinct] List
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

1. $\pi_{Name,NName}(\sigma_{Nummer=Abt}(Abteilung \times Angestellte))$
   - ```
     select Name, NName
     from Abteilung, Angestellte
     where Nummer = Abt
     order by Name;
     ```
2. $\pi_{Name,NName}(Abteilung \bowtie_{Nummer=Abt} Angestellte)$
   - ```
     select Name, NName
     from Abteilung inner join Angestellte on Nummer = Abt
     order by Name;
     ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Uneindeutigkeit in Statements

- Problem: Gleiche Attributnamen in zu verknüpfenden Tabellen
- Beispiel:
  - Relationen mit Schemata $Mitglieder(\underline{ID}, Name), Teilnahme(\underline{Kurs, ID})$
  - Liefere Namen der Mitglieder, die den Kurs 'Sport101' belegen:
    - $\pi_{Name}(\sigma_{Kurs='Sport101'}(Teilnahme) * Mitglieder)$
    - ```
      select Name
      from Mitglieder, Teilnahme
      where Kurs='Sport101' and ID = ID;
      ```

```
create table Mitglieder (
   ID     int         not null,
   Name   char(20)    not null,
   primary key (ID)
);
create table Teilnahme (
   Kurs   char(25)    not null,
   ID     int         not null,
   primary key(Kurs,ID)
);
```


```

Qualifizierung von Attributen & Aliases

- Über Punktnotation Attributname mit Tabellennamen qualifizieren zur eindeutigen Referenzierung
  - Verbessert Lesbarkeit, da im Statement angegeben ist, woher welches Attribut kommt
  - Beispiel
    - ```
      select Name
      from Mitglieder, Teilnahme
      where Kurs = 'Sport101' and Mitglieder.ID = Teilnahme.ID;
      ```
- Aliases für lange Tabellennamen
  - Dient auch der Übersicht bei komplexen Ausdrücken
  - Beispiel
    - ```
      select Name
      from Mitglieder m, Teilnahme k
      where k.Kurs = 'Sport101' and m.ID = k.ID;
      ```

Mitglieder | ID | Name

Teilnahme | Kurs | ID

Zurück zu Relationen kombinieren: Natürlicher JOIN

- Umsetzung des natürlichen JOINs ($*$) über Schlüsselworte NATURAL JOIN
  - RA: $R * S$
  - SQL: SELECT *
    FROM R NATURAL JOIN S
    - Wie auch in relationaler Algebra: Keine Spezifizierung der JOIN Bedingung, nur je eine Spalte behalten
- Beispiel
  - Liefere Namen der Mitglieder, die den Kurs 'Sport101' belegen:
    - $\pi_{Name}\left(\sigma_{Kurs='Sport101'}(Teilnahme) * Mitglieder\right)$
      bzw. $\pi_{Name}\left(\sigma_{Kurs='Sport101'}(Teilnahme * Mitglieder)\right)$
    - ```
      select Name
      from Mitglieder natural join Teilnahme
      where Kurs='Sport101';
      ```

Mitglieder | ID | Name

Teilnahme | Kurs | ID
      ```

Relationen kombinieren: OUTER JOIN

- Umsetzung der Outer-Join Varianten ($⟕$, $⟖$, $⟗$) über Schlüsselworte RIGHT / LEFT / FULL OUTER JOIN mit Spezifizierung der JOIN Bedingung wieder über ON
  - RA: $R ⟕_{\langle Bedingung \rangle} S$ | $R ⟖_{\langle Bedingung \rangle} S$ | $R ⟗_{\langle Bedingung \rangle} S$
  - SQL: SELECT *
    FROM R [RIGHT | LEFT | FULL] OUTER JOIN S ON $\langle Bedingung \rangle$;
- Beispiel
  - Gebe die Nachnamen aller Angestellten aus und wenn existent, dazu den Namen ihrer Abteilung
    - $\pi_{NName,AbtName}(Abteilung ⟕_{Nummer=Abt} Angestellte)$
    - ```
      select NName, Name
      from Abteilung right outer join Angestellte on Nummer = Abt
      ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Mehrfach-JOINs

- Umsetzung durch Verkettung
  1. Mittels AND in der WHERE Klausel
  2. Liste expliziter JOIN-Klauseln
     - Exemplarisch für drei Tabellen:
       FROM Table1 INNER JOIN Table2 ON <Join-Bedingung> INNER JOIN Table3 ON <Join-Bedingung2>
       - Man kann auch Klammern setzen, aber schränkt damit das DBMS in seinen Möglichkeiten zur Optimierung ein
     - Ergebnis unabhängig von Auswertungsreihenfolge
       - Aber: Größe der Zwischenergebnisse kann durch Reihenfolge beeinflusst werden

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Mehrfach-JOINs: Beispiel

- Gebe die Projektnummern, die Nummer der verantwortlichen Abteilung sowie den Namen, die Adresse und das Geburtsdatum der jeweiligen Abteilungsleitung für die Projekte am Standort Stafford aus

1.
```
select p.Nummer, p.AbtNr, ang.NName, ang.Adresse, ang.Geb
from Projekt p, Abteilung abt, Angestellte ang
where p.AbtNr = abt.Nummer and abt.Leitung = ang.SVN
            and p.Standort = 'Stafford';
```

2.
```
select p.PNummer, p.Abt, ang.NName, ang.Adresse, ang.Gdatum
from Projekt p inner join Abteilung abt on p.AbtNr = abt.Nummer
               inner join Angestellte ang on abt.Leitung = ang.SVN)
where p.Standort = 'Stafford';
```

Was für Effekte hat es, wenn man zuerst
- nach p.PStandort='Stafford' selektiert?
- die Joins durchführt?

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Self-Join

- Wenn im JOIN die gleiche Tabelle mehrfach vorkommt
  - Nutzung von Aliases um die Vorkommen zu unterscheiden
- Beispiel
  - Führe für jeden Angestellten seinen Nachnamen sowie den Nachnamen seines unmittelbaren Vorgesetzten auf:
    - ```
      select a.NName, v.NName
      from Angestellte a, Angestellte v
      where a.Vorgesetzte = v.SVN;
      ```
  - Problem: Gleiche Attributnamen im Endresultat: `a.NName, v.NName...`
    - Nicht nur bei Self-Joins ein Problem:
      - `select Produkt.Name, Projekt.Name...`

Projekt	Nummer	Name	Standort	AbtNr

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Umbenennung mittels Namen in SQL-Statement

- Umsetzung der Umbenennung (ρ) von Attributen im *Endresultat* über AS
  - RA: $\rho_{(B_1,\ldots,B_n)}(R)$
  - SQL: SELECT A_1 AS B_1, …, A_n AS B_n FROM R

```
select [distinct] {Attr as Name}
from TableList
[where <Condition>]
[order by {Attr [asc | desc]}];
```

- Beispiele
  - Beispiel auf vorheriger Folie zu Self-Joins
    - ```
      select a.NName as Untergebene, v.NName as Vorgesetzte,
      from Angestellte a, Angestellte v
      where a.Vorgesetzte=v.SVN;
      ```
  - Nicht-Self-Join-Beispiel von vorheriger Folie:
    - `select Produkt.Name as Produkt, Projekt.Name as Projekt ...`
      ```

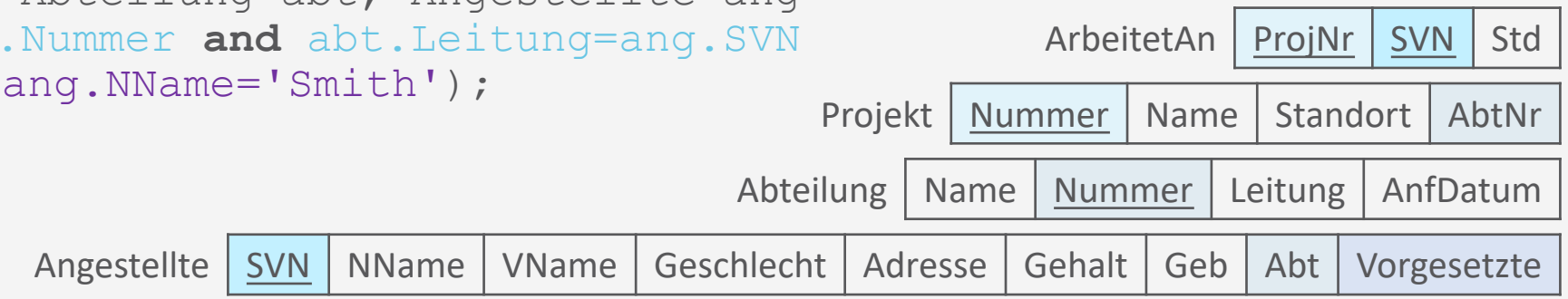
Mengenoperationen in SQL

- Umsetzung der Mengenoperationen (∪,∩, −) über entsprechende Schlüsselworte in Kombination mit SELECT-Ausdrücken zur Bestimmung der zu verarbeitenden Relationen
  - RA: $R \cup S$ $R \cap S$ $R - S$
  - SQL:
    (SELECT * FROM R) UNION (SELECT * FROM S)
    (SELECT * FROM R) INTERSECT (SELECT * FROM S)
    (SELECT * FROM R) MINUS (SELECT * FROM S)
    - Statt MINUS geht auch EXCEPT
  - Ergebnis von Mengen-Operationen: Tupel-Mengen
    - D.h., hier werden Duplikate aus dem Ergebnis entfernt
    - Sollen Duplikate erhalten bleiben, so muss das Schlüsselwort ALL ergänzt werden:
      - UNION ALL, INTERSECT ALL, EXCEPT ALL

```
select-statement
[  union
 | intersect
 | except
 | minus] [all]
select-statement;
```

Mengenoperationen in SQL: Beispiel

- Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert
  -
```
(select p.Nummer
 from Projekt p, ArbeitetAn arb, Angestellte ang
 where p.Nummer=arb.ProjNr and arb.SVN=ang.SVN
            and ang.NName='Smith')
union
(select p.Nummer
 from Projekt p, Abteilung abt, Angestellte ang
 where p.Abt=abt.Nummer and abt.Leitung=ang.SVN
            and ang.NName='Smith');
```



Aggregatfunktion und Gruppierung

Aggregatfunktion und Gruppierung

Aggregatfunktionen

- Umsetzung der Aggregation über Angabe des Funktionsnamens in der Attributliste des SELECT
  - RA: $\mathcal{F}_{\langle Liste\ von\ (Funktion,Attribut)\ Paaren \rangle}(R)$
  - SQL: SELECT $\langle Liste\ von\ Funktion(Attribut) \rangle$ FROM R
  - Standardfunktionen:
    - COUNT($Attribut$) Anzahl der Tupel; mit COUNT(DISTINCT($Attribut$)) Anzahl der verschiedenen Tupel
    - SUM($Attribut$) Summe der Werte der Tupelattribute
    - MIN($Attribut$) Wert des minimalen Tupelattributs
    - MAX($Attribut$) Wert des maximalen Tupelattributs
    - AVG($Attribut$) Durchschnittlicher Wert der Tupelattribute; mit AVG(DISTINCT($Attribut$)) Durchschnitt der verschiedenen Tupel

```
select {Funktion([distinct] Attr)}
from Table;
```

Aggregatfunktionen: Beispiel

```
select {Funktion([distinct] Attr)}
from Table;
```

- Liefere
  - die Summe der Gehälter,
  - das maximale Gehalt,
  - das durchschnittliche Gehalt und
  - das minimale Gehalt

  aller Angestellten
  - $\mathcal{F}_{\text{SUM}(Gehalt),\text{MAX}(Gehalt),\text{AVG}(Gehalt),\text{MIN}(Gehalt)}(Angestellte)$
  - ```
    select sum(Gehalt), max(Gehalt), avg(Gehalt), min(Gehalt)
    from Angestellte;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Gruppierung mit GROUP BY

- In der Praxis: Aggregatfunktionen häufig auf Gruppen von Tupeln
  - Beispiel: Liefere die Summe der Gehälter, das maximale Gehalt, das durchschnittliche Gehalt und das minimale Gehalt *pro Abteilung*

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

- Umsetzung von Gruppierungen
  - RA: $\langle B_1, \dots, B_m \rangle \mathcal{F}_{\langle Liste\ von\ (Funktion, Attribut)\ Paaren \rangle}(R)$
  - SQL: SELECT $\langle B_1, \dots, B_m \rangle\ \langle Liste\ von\ Funktion(Attribut) \rangle$
    FROM R
    GROUP BY $B_1, \dots, B_m$
    - Nur was in der GROUP BY Klausel vorkommt, kann im SELECT als weiteres Attribut vorkommen
      - Es muss nicht jedes Attribut im SELECT vorkommen, macht aber die Ergebniszeilen weniger interpretierbar
    - Abarbeitungsreihenfolge: Erst gruppieren, dann aggregieren

Gruppierung mit GROUP BY: Beispiel

- Liefere
  - die Summe der Gehälter,
  - das maximale Gehalt,
  - das durchschnittliche Gehalt und
  - das minimale Gehalt

  der Angestellten *pro Abteilung*
  - $Abt\ \mathcal{F}_{\text{SUM}(Gehalt),\text{MAX}(Gehalt),\text{AVG}(Gehalt),\text{MIN}(Gehalt)}(Angestellte)$
  - **select** Abt, **sum**(Gehalt), **max**(Gehalt), **avg**(Gehalt), **min**(Gehalt)
    from Angestellte
    group by Abt;

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Gruppierung mit GROUP BY: Beispiel

Wie sortieren wir das Ergebnis absteigend nach Abteilungsgröße?

- Liefere für jede Abteilung die Nummer, die Anzahl der zugehörigen Angestellten und deren Durchschnittsgehalt

- $Abt\,\mathcal{F}_{\mathrm{COUNT}(*),\mathrm{AVG}(Gehalt)}(Angestellte)$
- ```
  select Abt, count(*), avg(Gehalt)
  from Angestellte
  group by Abt;
  ```

VName	NName	SVN	...	Gehalt	Vorgesetzte	Abt
John	Smith	01234567X890		30000	12345678Y901	5
Franklin	Wong	12345678Y901		40000	78901234E567	5
Ramesh	Narayan	23456789Z012		38000	12345678Y901	5
Joyce	English	34567890A123		25000	12345678Y901	5
Alicia	Zelaya	45678901B234	...	25000	56789012C345	4
Jennifer	Wallace	56789012C345		43000	78901234E567	4
Ahmad	Jabbar	67890123D456		25000	56789012C345	4
James	Bong	78901234E567		55000	null	1

Abt	COUNT(*)	AVG(Gehalt)
5	4	33250
4	3	31000
1	1	55000

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
  ```

Abarbeitungsreihenfolge von Aggregation, Gruppierung und JOIN

- Aggregation und Gruppierung werden nach einem JOIN angewendet
- Beispiel:
  - Liefere für jedes Projekt die Projektnummer, den Projektnamen und die Anzahl der daran arbeitenden Angestellten
    - $Nummer, Name\ \mathcal{F}_{\mathrm{COUNT}(*)}(Projekt \bowtie_{Nummer=ProjNr} ArbeitetAn)$
    - ```
      select Nummer, Name, count(*)
      from Projekt, ArbeitetAn
      where Nummer = ProjNr
      group by Nummer, Name;
      ```
  - Erst JOIN, dann Gruppierung, dann Aggregat (COUNT)

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList]
[order by {Attr [asc | desc]}];
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr
```

Bedingungen auf Gruppierungen: HAVING

- HAVING für Bedingungen auf Gruppen in Gruppierungen
  - Entspricht WHERE auf Einzel-Tupeln
  - Tritt nur zusammen mit GROUP BY auf
- Beispiel:
  - Liefere für jedes Projekt, an dem mehr als zwei Angestellte arbeiten, die Projektnummer, den Projektnamen und die Anzahl der daran jeweils arbeitenden Angestellten

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Beispiel: HAVING

Tabelle ist das Ergebnis nach JOIN und Anwendung der GROUP BY Klausel

Name	Nummer	...	SozVersNr	Stunden
ProduktX	1		01234567X890	32,5
ProduktX	1		34567890A123	20,0
ProduktY	2		01234567X890	7,5
ProduktY	2		34567890A123	20,0
ProduktY	2		12345678Y901	10,0
ProduktZ	3		89012345F678	40,0
ProduktZ	3		12345678Y901	10,0
Computerisation	10		12345678Y901	10,0
Computerisation	10	...	45678901B234	10,0
Computerisation	10		67890123D456	35,0
Reorganisation	20		12345678Y901	10,0
Reorganisation	20		56789012C345	15,0
Reorganisation	20		78901234E567	null
Newbenefits	30		67890123D456	5,0
Newbenefits	30		56789012C345	20,0
Newbenefits	30		45678901B234	30,0

Diese Gruppen fliegen raus

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Beispiel: HAVING - Ergebnisse

Name	Nummer	...	SozVersNr	Stunden
ProduktY	2		01234567X890	7,5
ProduktY	2		34567890A123	20,0
ProduktY	2		12345678Y901	10,0
Computerisation	10		12345678Y901	10,0
Computerisation	10		45678901B234	10,0
Computerisation	10		67890123D456	35,0
Reorganisation	20		12345678Y901	10,0
Reorganisation	20		56789012C345	15,0
Reorganisation	20		78901234E567	null
Newbenefits	30		67890123D456	5,0
Newbenefits	30		56789012C345	20,0
Newbenefits	30		45678901B234	30,0

Nach Anwendung der HAVING Klausel

Nummer	COUNT(*)
2	3
10	3
20	3
30	3

Nach COUNT und SELECT (PName nicht dargestellt)

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

```
select Nummer, Name, count(*)
from Projekt, ArbeitetAn
where Nummer = ProjNr
group by Nummer, Name
having count(*) > 2;
```

ArbeitetAn (ProjNr, SVN, Std)

Projekt (Nummer, Name, Standort, AbtNr)

Weiteres Beispiel für Gruppierung

- Liefere für jedes Projekt
  - die Nummer,
  - den Namen und
  - die Anzahl der Angestellten,

  die aus Abteilung 5 an dem betreffenden Projekt arbeiten
  - ```
    select Nummer, Name, count(*)
    from Projekt p, ArbeitetAn arb, Angestellte ang
    where p.Nummer=arb.ProjNr and arb.SVN=ang.SVN and ang.Abt=5
    group by Nummer, Name;
    ```

```
select Attr_Fct_List
from TableList
[where <Condition>]
[group by AttrList
[having <GroupCondition>]]
[order by {Attr [asc | desc]}];
```

ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
```

Weitere Klauseln, Tests und Vergleichsmöglichkeiten

String-Tests

Element-Test

Zeichenvergleiche

- Gleichheit von Substrings: LIKE und zwei Sonderzeichen/Platzhalter
  - % : beliebige Anzahl von Zeichen
  - _ : genau ein Zeichen
- Beispiele:
  - Liefere eine Liste der Vor- und Nachnamen aller Angestellten, die in Houston/Texas wohnen
    - ```
      select VName, NName
      from Angestellte
      where Adresse like '%Houston, TX%';
      ```
  - Liefere eine Liste der Vor- und Nachnamen aller Angestellten, deren SVN an der dritten Stelle die Ziffer 8 besitzt
    - ```
      select VName, NName
      from Angestellte
      where SVN like '_ _ 8 _ _ _ _ _ _ _ _ _';
      ```

Operatoren

- Für Zahlen: Arithmetische Operatoren (+, -, *, /)
- Für Zeichenketten: Verbindungsoperator (||)
- Für Datum, Zeit, Zeitstempel und Intervall: Plus und Minus (+, -)
- Vergleichsoperator BETWEEN für Intervall-Prüfung

- Beispiele:
  - 10% Lohnerhöhung testen, i.e., eine Liste aller Angestellten mit Vor- und Nachname und deren Gehalt, um 10% erhöht
  - E-Mail-Liste für Angestellte der Abteillung 5, die Gehalt zwischen 30.000 und 40.000 beziehen (Annahme: Angestellte haben ein Email-Attribut):

Operatoren

- Beispiele:
  - 10% Lohnerhöhung testen:
    - ```
      select ang.VName, ang.NName, 1.1*ang.Gehalt AS PlusGehalt
      from Angestellte ang, ArbeitetAn arb, Projekt p
      where ang.SVN=arb.SVN and arb.PNr=p.Nummer and p.Name='ProductX';
      ```
  - E-Mail-Liste (Annahme: Angestellte haben ein Email-Attribut):
    - ```
      select  VName || '  ' || NName || '  <' || Email || '>'
      from Angestellte
      where (Gehalt between 30000 and 40000) and Abt=5;
      ```
    - Ausgabe bei VName='John', NName='Smith‘, Email='js@blub.de‘, Gehalt=35000, Abt=5:
      'John Smith <js@blub.de>'

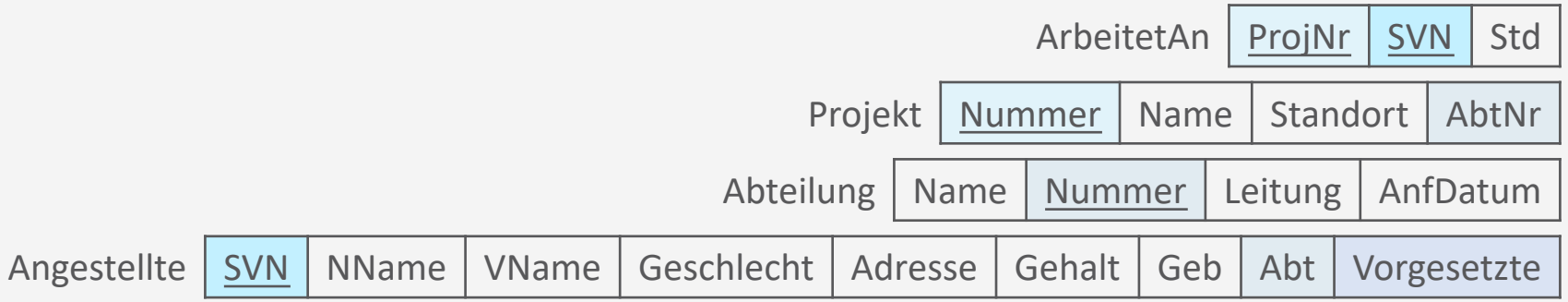
ArbeitetAn	ProjNr	SVN	Std

Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Verschachtelte Anfragen und disjunktiver Elementtest: IN

- Innerhalb des WHERE Ausdrucks können wiederum SELECT Ausdrücke formuliert werden, deren Resultate dann im WHERE Ausdruck weiterverwendet werden
  - Üblicherweise für weitere Vergleiche
- Möglichkeit des Vergleichs: disjunktiver Elementtest mittels IN
- Beispiel: Formulierung der UNION-Query (vgl. Folie 60) ohne UNION
  - Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert



Verschachtelte Anfragen und disjunktiver Elementtest: Beispiel

- Erstelle eine Liste aller Projektnummern von Projekten, an denen ein Mitarbeiter mit Nachnamen 'Smith' als Mitarbeiter oder Leiter der Abteilung arbeitet, die das Projekt kontrolliert
  - ```
    select distinct Nummer
    from Projekt
    where Nummer in (select p.Nummer
                     from Projekt p, Abteilung abt, Angestellte ang
                     where p.AbtNr=abt.Nummer and abt.Leitung=ang.SVN
                                  and ang.NName='Smith‘)
                OR
          Nummer in (select arb.ProjNr
                     from ArbeitetAn arb, Angestellter ang
                     where arb.SVN=ang.SVN
                                  and ang.NName='Smith');
    ```
    ```

Verschachtelte Anfragen und disjunktiver Elementtest mit Gruppierung: Bsp.

- Liefere für jede Abteilung mit mehr als fünf Angestellten die Nummer und die Anzahl der Angestellten, die mehr als 40.000 verdienen.
  - ```
    select Abt, COUNT(*)
    from Angestellte
    where Gehalt>=40000 and Abt in (select Abt
                                    from Angestellte
                                    group by Abt
                                    having count(*) >= 5 )
    group by Abt;
    ```

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

IN-Operator mit Tupeln

- IN-Operator kann auch zum Testen mit Tupeln verwendet werden
  - Tupel-Ausdruck muss UNION-kompatibel zum Resultat des inneren Ausdrucks sein
- Beispiel:
  - Zeige die Sozialversicherungsnummern aller Angestellten, die in einer gleichen Kombination von Projekt und Stunden an einem Projekt arbeiten, an dem auch der Angestellte ‘John Smith‘ mit der SVN 01234567X890 beschäftigt ist.
    - ```
      select distinct SVN
      from ArbeitetAn
      where (ProjNr, Std) IN (select ProjNr, Std
                              from ArbeitetAn
                              where SVN='01234567X890');
      ```

ArbeitetAn

ProjNr	SVN	Std
      ```

Sichtbarkeit von Variablen in verschachtelten Anfragen

- Variablen der äußeren Anfragen in der inneren Anfrage sichtbar (aber nicht anders herum)
  - *Korrelierte* Anfrage: innere Anfrage referenziert auf äußere Anfrage
  - Semantik: Innere Anfrage wird einmal für jedes Tupel ausgewertet
  - Beispiel
    - Liefere die Namen der Angestellten, die einen Angehörigen mit gleichem Vornamen und gleichem Geschlecht wie der Angestellte selbst haben
      - ```
        select a.NName, a.VName
        from Angestellte a
        where a.SVN in (select fam.SozVersNr
                        from Angehoerige fam
                        where a.VName=fam.Name and a.Geschlecht=fam.Geschlecht
        ```

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
        ```

Vergleichsoperatoren ANY und ALL

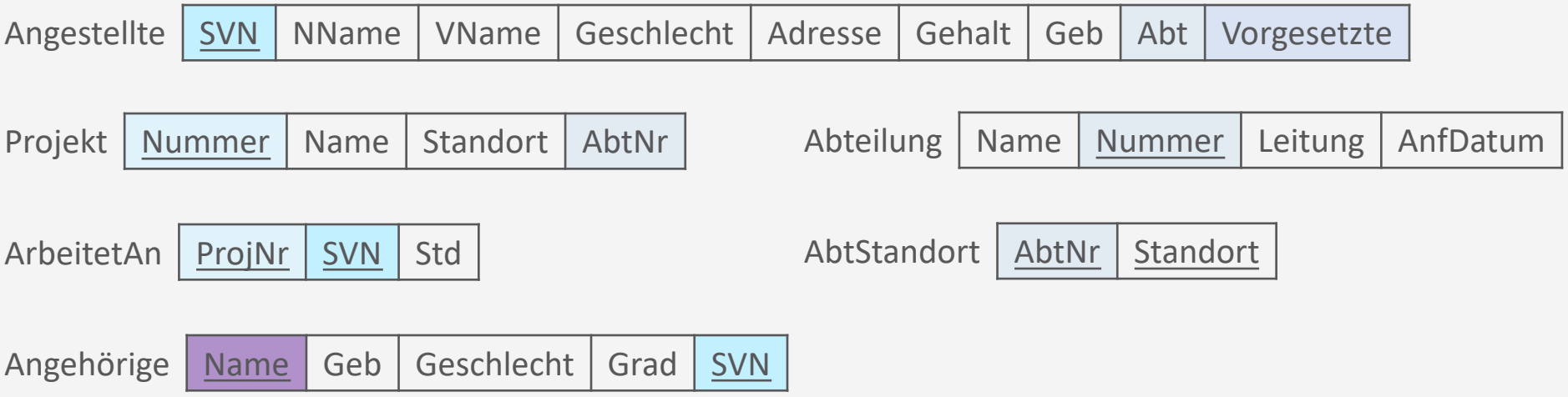
- Weitere Vergleichsoperatoren für ganze Mengen:
  - <op> ANY | SOME
  - <op> ALL
    - <op> ersetzbar durch ein der Operatoren {=,>, >=, <=, <, <>}
    - v > ALL V liefert bspw. dann TRUE, wenn v größer ist als alle Werte der Menge V
    - = ANY [oder auch = SOME] ist äquivalent zu IN
- Beispiel:
  - Gebe Nach- und Vorname aller aus, die ein höheres Gehalt haben als alle aus Abteilung 5
  - ```
    select NName, VName
    from Angestellte
    where Gehalt > all(select Gehalt
                       from Angestellte
                       where Abt=5 );
    ```
    ```

EXISTS-Test

- EXISTS überprüft, ob das das Resultat einer korrelierten verschachtelten Anfrage leer (FALSE) ist – also kein Tupel enthält – oder nicht (TRUE)
  - Kann auch mit NOT kombiniert werden
- Beispiel:
  - Liefere die Namen der Angestellten, die einen Angehörigen mit gleichem Vornamen und gleichem Geschlecht wie der Angestellte selbst haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where exists (select fam.SozVersNr
                    from Angehoerige fam
                    where a.SVN=fam.SozVersNr and a.VName=fam.Name and
      a.Geschlecht=fam.Geschlecht );
      ```

Angehörige

Name	Geb	Geschlecht	Grad	SVN

Angestellte

SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

UNIQUE zum Duplikattest

- Überprüft, ob eine Multimenge Duplikate enthält (FALSE) oder nicht (TRUE).
- Beispiel:
  - Liefere die Namen der Angestellten, die einen eindeutigen Vornamen haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where unique (select b.VName
                    from Angestellte b
                    where a.VName=b.VName );
      ```

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Junktoren bei Mengenvergleichen

- Mengenvergleiche können durch Junktoren AND und OR miteinander verknüpft werden
  - Mehrere korrelierte Anfragen
- Beispiel:
  - Erstelle eine Liste mit den Namen der Manager, die mindestens einen Angehörigen haben
    - ```
      select a.NName, a.VName
      from Angestellte a
      where exists (select *
                    from Angehoerige fam
                    where a.SVN=fam.SozVersNr)
            and
            exists (select *
                    from Abteilung abt
                    where a.SVN=abt.Leitung);
      ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
      ```

Explizite Mengenangaben

- Menge explizit angeben und in der WHERE-Klausel verwenden
- Beispiel:
  - Gib die Sozialversicherungsnummern aller Angestellten aus, die an Projekten mit den Nummern 1, 2 oder 3 arbeiten.
    - ```
      select distinct SVN
      from ArbeitetAn
      where ProjNr in (1, 2, 3);
      ```
      ```

NULL-Test

- Prüft, ob der Wert eines Attributs NULL ist
  - Hier kein = möglich! (IS NULL statt = NULL)
- Beispiel:
  - Liefere die Namen der Angestellten, die keinen Vorgesetzten haben.
    - ```
      select NName, VName
      from Angestellte
      where Vorgesetzte is null;
      ```
  - Liefere die Namen der Angestellten, die einen Vorgesetzten haben.
    - ```
      select NName, VName
      from Angestellte
      where Vorgesetzte is not null;
      ```

Daten ändern

INSERT, DELETE, UPDATE

Relationales Schema

- Relationales Schema (in 3NF/BCNF): Firma → Anlegen



- Relationale Algebra: Anfragen und Datenmanipulation
  - $\pi, \rho, \sigma, \cup, \cap, -, \times, \bowtie$; gruppieren, aggregieren
  - Insert, delete, update

Tupel einfügen

- INSERT benötigt:
  - Ziel-Relation
  - Ggf. Attribute (sonst Reihenfolge wie in Schemadefinition)
  - Werteliste oder Anfrage, die Werteliste produziert

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel einfügen: Beispiele für Werteliste

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

- Neuen Angestellten mit allen Werten einfügen
  - ```
    insert into Angestellte
    values ('90123456G789', 'Marini', 'Richard', 'M', '98 Oak
    Forest, Katy, TX', 37000, '30.12.1962', 4, '67890123D456' );
    ```
- Neuen Angestellten mit Teilinformationen einfügen
  - ```
    insert into Angestellte (VName, NName, Abt, SVN)
    values ('Richard', 'Marini', 4, '90123456G789‘ );
    ```
    - Rest wird auf NULL oder DEFAULT gesetzt, wenn kein Constraint verletzt wird
- Neue Abteilung anlegen: Nummer wird automatisch eingefügt (SERIAL)
  - ```
    insert into Abteilung(Name, Leitung, AnfDatum)
    values ('Facilities', '90123456G789',
    '2019-06-14');
    ```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Tupel einfügen: Beispiele

- Einfügen von Daten aus einer anderen Tabelle:
  - Neue Tabelle:
    - ```
      create table AbtInfo (
      AbtName            varchar(15),
      AnzahlAngest       integer,
      GehaltGesamt       integer
      );
      ```
  - ```
    insert into AbtInfo (AbtName, AnzahlAngest, GehaltGesamt)
        select Name, count(*), sum(Gehalt)
        from (Abteilung inner join Angestellte on Nummer=Abt )
        group by Name;
    ```

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Abteilung	Name	Nummer	Leitung	AnfDatum

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Abweisung von INSERT

- Wenn DB-Integrität verletzt wird
- Mögliche Fehlerquellen
  - Kein Primärschlüssel angegeben
  - Fremdschlüssel existiert nicht in Zieltabelle
  - Kein Wert angegeben trotz NOT NULL
  - CHECK-Constraint verletzt
  - Doppelter Wert trotz UNIQUE

```
insert into TableName[(AttrList)]
[values {ValueList} | select-stmt];
```

Tupel löschen

- DELETE benötigt:
  - Name der Relation, aus der gelöscht werden soll
  - WHERE-Klausel, die bestimmt, welche Tupel gelöscht werden sollen
  - Darf ebenfalls Integrität nicht verletzen
    - Aktionen zur Behandlung (CASCADE, NO ACTION, SET DEFAULT, SET NULL)

```
delete from TableName
where <condition>;
```

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel löschen: Beispiele

```
delete from TableName
where <condition>;
```

- Alle Angehörigen löschen
  - `delete from Angehoerige;`
- Alle Angehörigen des Angestellte mit SVN `34567890A123` löschen
  - ```
    delete from Angehoerige
    where SVN = '34567890A123';
    ```
- Alle Angehörigen der Angestellten löschen, die in Abteilung 4 arbeiten
  - ```
    delete from Angehoerige
    where SVN in(select SVN
                 from Angestellte
                 where Abt = 4 );
    ```

Angehörige	Name	Geb	Geschlecht	Grad	SVN

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Tupel aktualisieren

- UPDATE benötigt:
  - Name der Relation
  - SET – Anweisung für die Änderung
    - Kann auch Berechnungen enthalten
  - WHERE-Anweisung für die zu ändernden Tupel
- Reihenfolge mehrerer UPDATE-Anweisungen ist relevant!

```
update TableName
set <statement>
where <condition>;
```

Tupel aktualisieren: Beispiele

```
update TableName
set <statement>
where <condition>;
```

- Projekt mit der Nummer 5 ist jetzt am Standort Bellaire zu finden und wird von Abteilung mit der Nummer 5 betreut
  - ```
    update Projekt
    set Standort='Bellaire', AbtNr=5
    where Nummer=5;
    ```
- Alle Angestellte der Abteilung mit Namen Research erhalten 10% mehr Gehalt
  - ```
    update Angestellte
    set Gehalt=Gehalt * 1.1
    where Abt in(select Nummer
                 from Abteilung
                 where Name = 'Research');
    ```

Merge von Tabellen

- Eine Tabelle kann in eine andere eingefügt werden, wobei hier im WHEN MATCHED über das Einfügen bestimmt
  - Beispiel:

```
merge into AngestellteAll c
using Angestellte a
on (a.SVN = c.SVN)
when matched then
  update
  set
  c.VName = a.VName,
  c.NName = a.NName,
  c.Geb = a.Geb,    ...
when not matched then
  insert values(a.SVN, ..., a.Abt);
```

```
merge into TableName1
using TableName2
on <condition>
when matched then
   update
   set
   <statement>
when not matched then
   insert values ValueList;
```

Zwischenzusammenfassung

- Grundkonstrukte
  - SELECT, FROM, WHERE
- Sortierung
  - ORDER BY: ASC, DESC
- Relationen kombinieren
  - INNER / NATURAL / OUTER JOIN
  - UNION, INTERSECT, MINUS / EXCEPT
- Aggregationen und Gruppierungen
  - COUNT, SUM, AVG, MIN, MAX
  - GROUP BY
  - HAVING
- Weitere Vergleichsmethoden
  - Operatoren (+ - % / * IN LIKE…)
  - IN, ANY, ALL, EXIST, UNIQUE
  - DISTINCT
  - IS NULL / IS NOT NULL
- Datenänderung
  - INSERT
  - UPDATE
    - WHEN MATCHED
  - DELETE

SQL ist ein Standard, der u.a. eine Grammatik definiert
→ Implementierungen (MySQL, PostgreSQL, etc.) setzen den Standard um, woraus Unterschiede in der Nutzung über Systeme hinweg entstehen können

Überblick: 5. Structured Query Language (SQL)

A. Datendefinition (SQL als DDL)
- Schema, Tabellen, Datentypen, Constraints definieren
- Strukturelle Änderungen mittels drop, alter

B. Datenmanipulation (SQL als DML)
- Anfragen
- Datenänderungen

C. Und der Rest
- Sichten (SQL als VDL)
- Rechtevergabe (SQL als DCL)
- Programmiermethoden

Sichten (Views)

SQL als View Definition Language (VDL)
(Machmal auch als Teil der DML aufgefasst)

Sichten: Virtuelle Relationen

- VIEW: aus anderen Relationen abgeleitete Relation
  - Wird über eine SELECT-Anweisung spezifiziert
  - Werden von DB aktuell gehalten
  - Virtuelle Relation: kann, muss aber nicht in der Datenbank abgespeichert werden
  - VIEWs können für Abfragen wie normale Relationen genutzt werden

```
create view ViewName as
<select-statement>
```

ArbeitetAn	ProjNr	SVN	Std

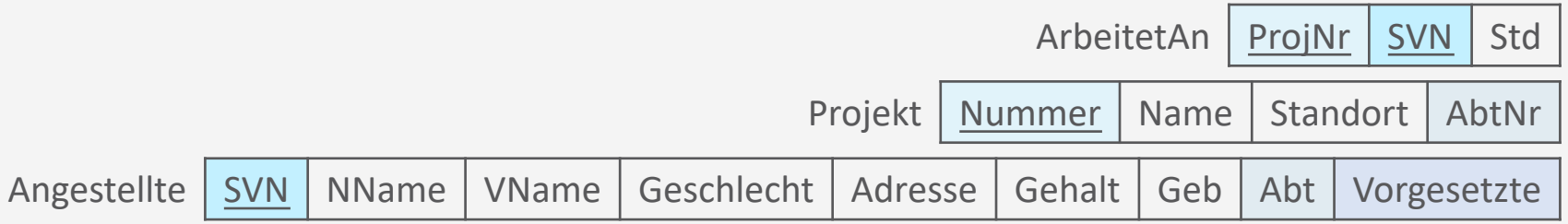
Projekt	Nummer	Name	Standort	AbtNr

Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte

Sichten: Virtuelle Relationen

- Sicht auf Mitarbeitende mit Namen an Projekten
  - ```
    create view ArbeitetAnMitNamen as
    select ang.SVN, VName, NName, Name, Std
    from Angestellte ang, Projekt p, ArbeitetAn arb
    where ang.SVN=arb.SVN and arb.ProjNr=p.Nummer;
    ```
  - Anfrage
    - ```
      select VName, NName, Std as Stunden
      from ArbeitetAnMitNamen
      where SVN = '01234567X890';
      ```

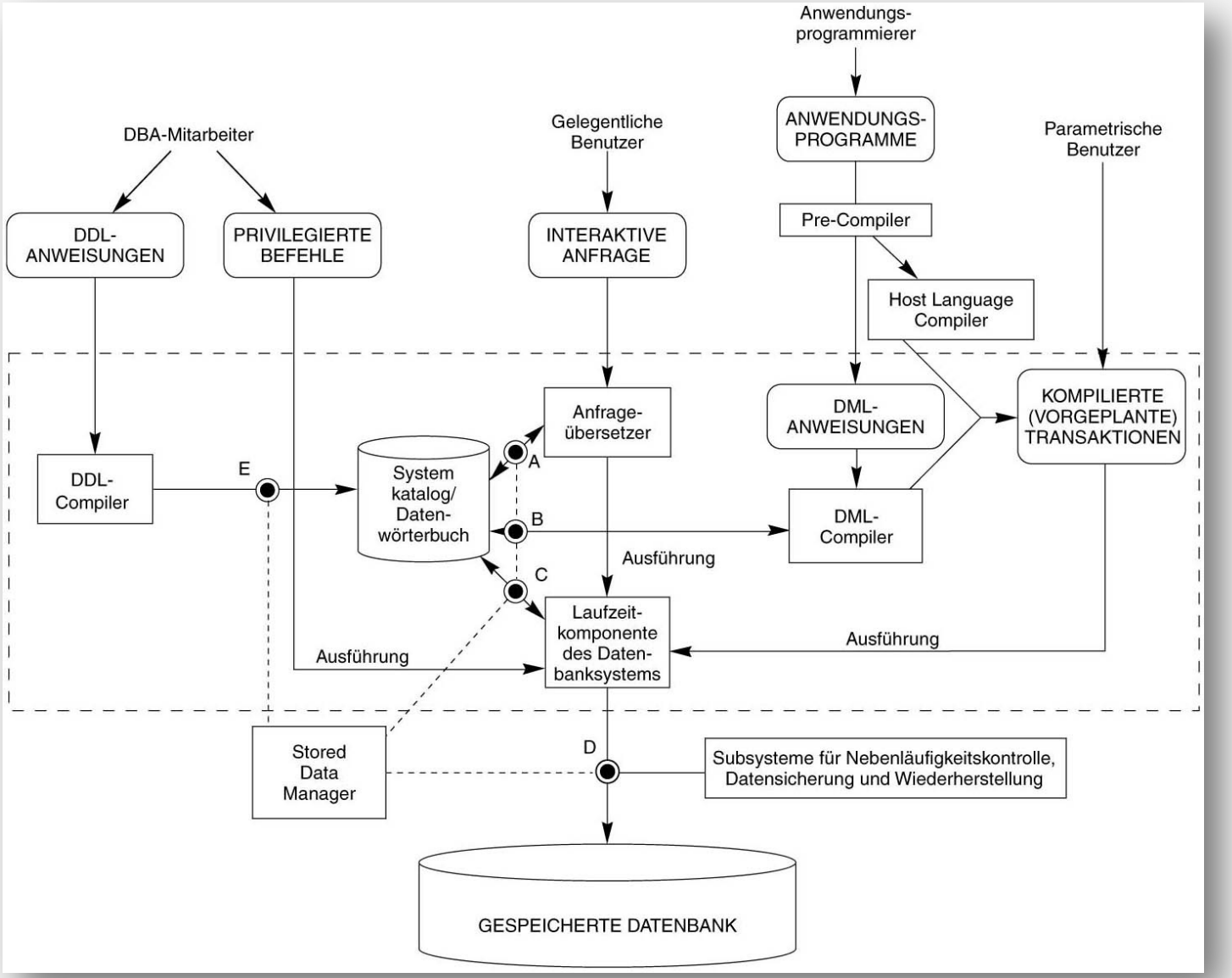
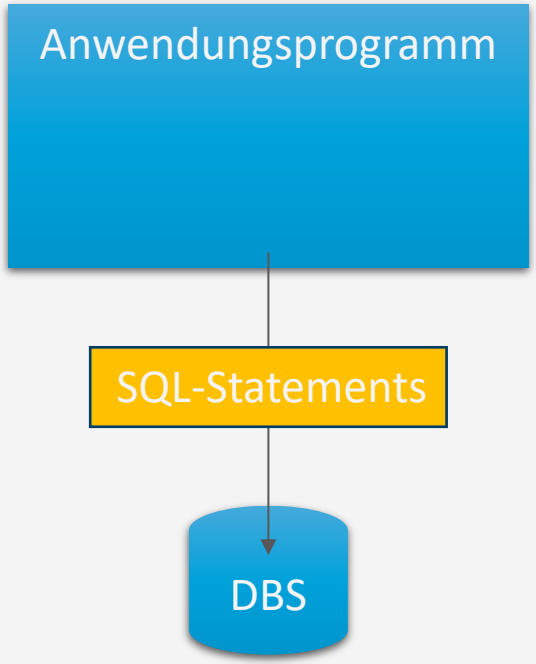
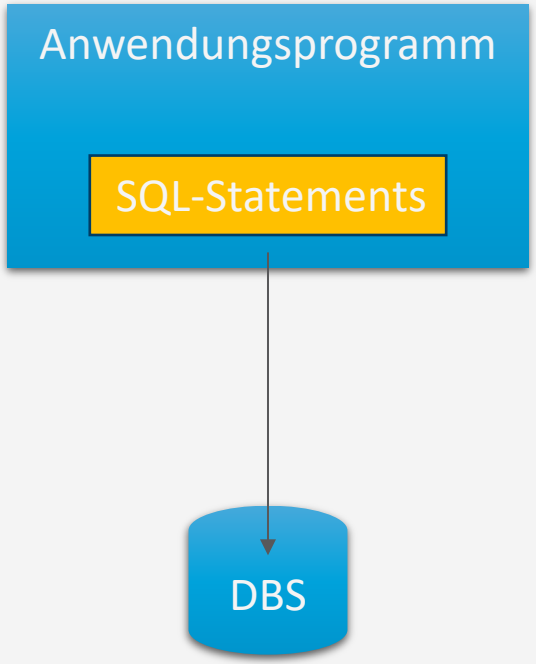
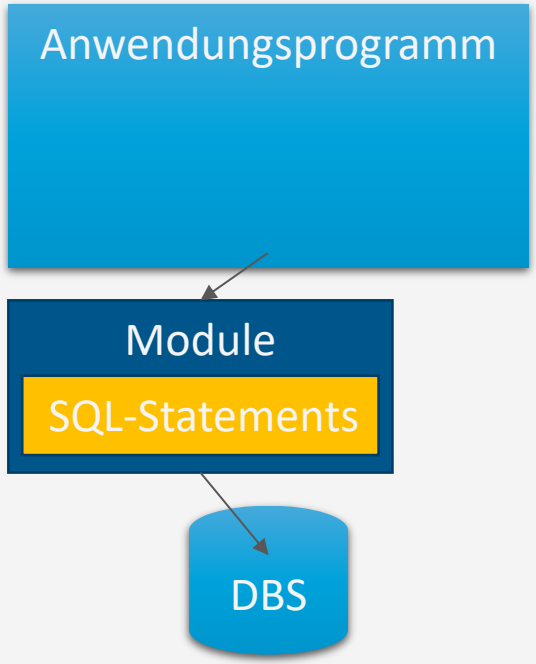
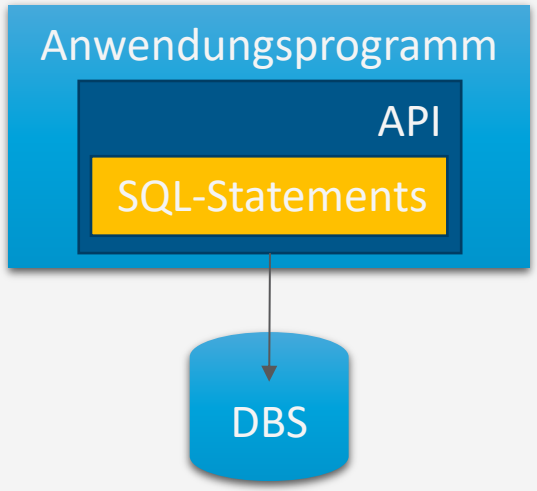
```
create view ViewName as
<select-statement>
```



Verwendung von VIEWs

```
create view ViewName as
<select-statement>
```

- Aktualisierungen/Datenmanipulationen häufig nicht möglich
  - Non-updateable views
- Beispiel: Nutzung von Aggregationsfunktionen:
  - ```
    create view AbtInfo as
       select Abt, count(*) as Cnt, avg(Gehalt) as AvgG
       from Angestellte
       group by Abt;
    ```
  - `insert into AbtInfo values (1,5,20000)` ???
  - `update AbtInfo set Cnt = 5 where Abt = 5` ???
- Beispiel: Sei ein NOT NULL Attribut ohne DEFAULT nicht Teil des Views
  - Kein INSERT

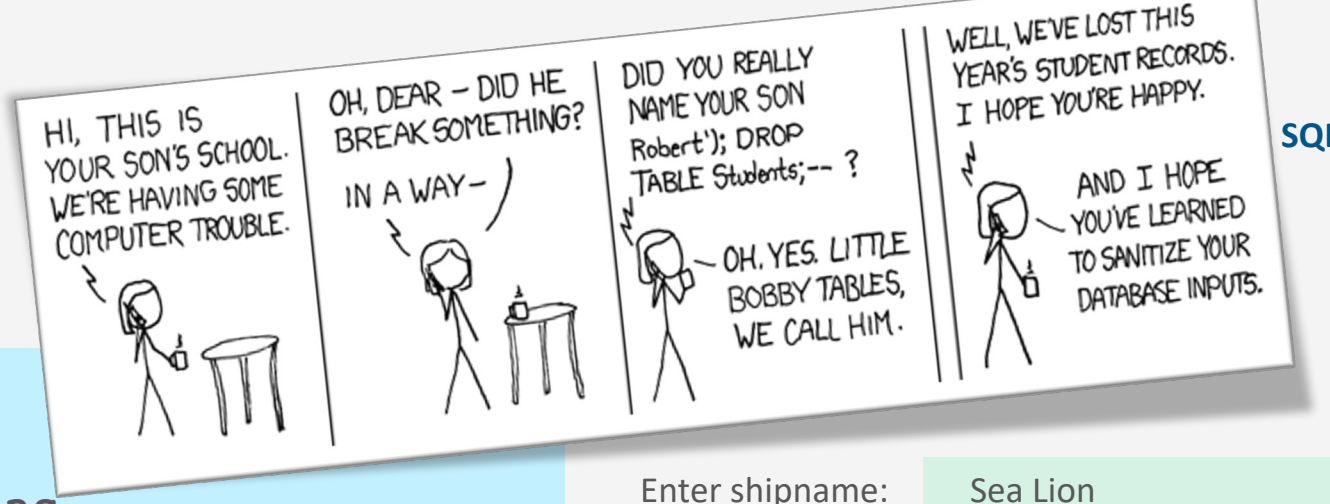
Angestellte	SVN	NName	VName	Geschlecht	Adresse	Gehalt	Geb	Abt	Vorgesetzte
    ```

Rechte

SQL als Data Control Language (DCL)

Rechtevergabe mittels SQL

- Rechtevergabe an DB-Objekten
- Benutzerprivilegien:
  - Lesen oder Ändern von Relationen oder Spalten
  - Anlegen von Relationenschemata oder Datenbankschemata
  - Weitergabe von Privilegien
- Rechte mittels
  - GRANT (ein ausgewähltes Recht) geben
  - REVOKE (entsprechendes Recht) wieder entziehen
    - SELECT, UPDATE, DELETE, INSERT aber auch
    - EXECUTE, ALTER, CREATE, MANAGE,….., etc

```
grant [Right]
   on Table(AttrList)
   to UserList;
grant [Right]
   on TableList
   to UserList;
```

```
revoke [Right]
   on Table(AttrList)
   from UserList;
revoke [Right]
   on TableList
   from UserList;
```

Rechtevergabe mittels SQL: Beispiele

- Rechte auf anfragen und aktualisieren einer Tabelle
  - ```
    grant select, update on Nutzer.Tabelle
    to AndererNutzer;
    ```
- Rechte auf einfügen, anfragen, löschen einer Tabelle; Weitergabe von Rechten
  - ```
    grant insert, select, delete on Angestellter
    to joerg, sabine, harald with grant option;
    ```
- Recht auf aktualisieren von einem Attribut
  - ```
    grant update(Gehalt) on Angestellter
    to chefe;
    ```
- Zurücknehmen des Rechts ein Attribut zu aktualisieren
  - ```
    revoke update(Gehalt) on Angestellte
    from chefe;
    ```

Programmiermethoden

Benutzung von SQL

Erweiterte Systemumgebung

- Zugriffe von
  - DB-Administratoren
  - Anwendungen
  - Nutzer*innen
    - Direkt auf DB
      - Gelegentlich, interaktiv
    - Parametrisch
      - Vorgefertigte Anwendungsprogramme mit beschränktem Kommandovorrat (routinierte, wohldefinierte, formalisierte Befehle)



SQL Programmier-Methoden



Direkter Aufruf
- Von SQL an die DB



Embedded SQL
- SQL wird in Host-Sprache eingebunden

Dynamic SQL
- Zur Programmlaufzeit zusammengebaut



Module Language
- SQL wird in Module ausgelagert, die von Host-Sprache aus angefragt werden



Call-Level APIs
- Schnittstellen, um aus der Host-Sprache DBs anzusprechen, z.B. SQL/CLI, ODBC, JDBC

Mappings
- Programmierer sieht kein SQL mehr

Impedance Mismatch

- Problem: Datenzugriff unterscheidet sich zwischen SQL und anderen Programmiersprachen
  - SQL: mengenorientiert
  - Andere: verarbeiten einzelne Werte

→ sog. *impedance mismatch*

- Relationales Datenmodell wird von den meisten Programmiersprachen nicht unterstützt

Embedded SQL: "Shared" Variablen

- Transferieren Informationen zwischen Datenbank und Anwendungsprogramm
- Werden in einem DECLARE Abschnitt deklariert:
  - Inhalt des Abschnitts hängt von Programmiersprache ab
- Können im SQL-Statement statt einer Konstante verwendet werden
- Variablen-Name wird mit Doppelpunkt gekennzeichnet
- Spezielle Variable SQLSTATE enthält Fehlercodes
  - '00000': no error condition, '02000': no tuple found

```
exec sql begin declare section;
        { Declaration }
exec sql end declare section;
```

Teil der offiziellen SQL-Grammatik

Shared Variable mit INSERT

- Um Daten in die DB über bzw. aus der DB in die Variablen zu bekommen, EXEC SQL mit INSERT bzw. SELECT kombinieren

```
exec sql insert into TableName(AttrList)
values (VarListColonPrepended)
```

```
exec sql select into VarListColonPrepended
from TableName;
```

```
void setParts() {
   exec sql begin declare section;
      char part[4], project[4], version[10], description[50];
      char sqlstate[6];
   exec sql end declare section;

   /* request part, project, version, description */

   exec sql insert into parts(partno, version, projectno, part_description)
         values (:part, :version, :project, :description);
}
```

Beispiel: Single-Row SELECT Statements

```
int getNumProjects(int minBudget) {
   exec sql begin declare section;
      int num, budget;
      char sqlstate[6];
   exec sql end declare section;
   budget := minBudget;
   exec sql select count(*)
            into :num
            from projects
            where budget >= :budget;

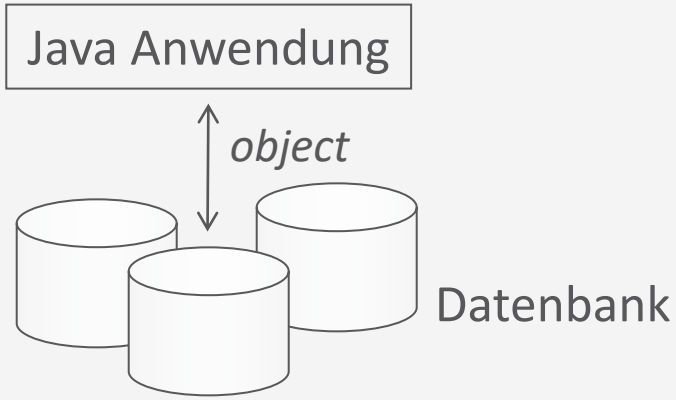
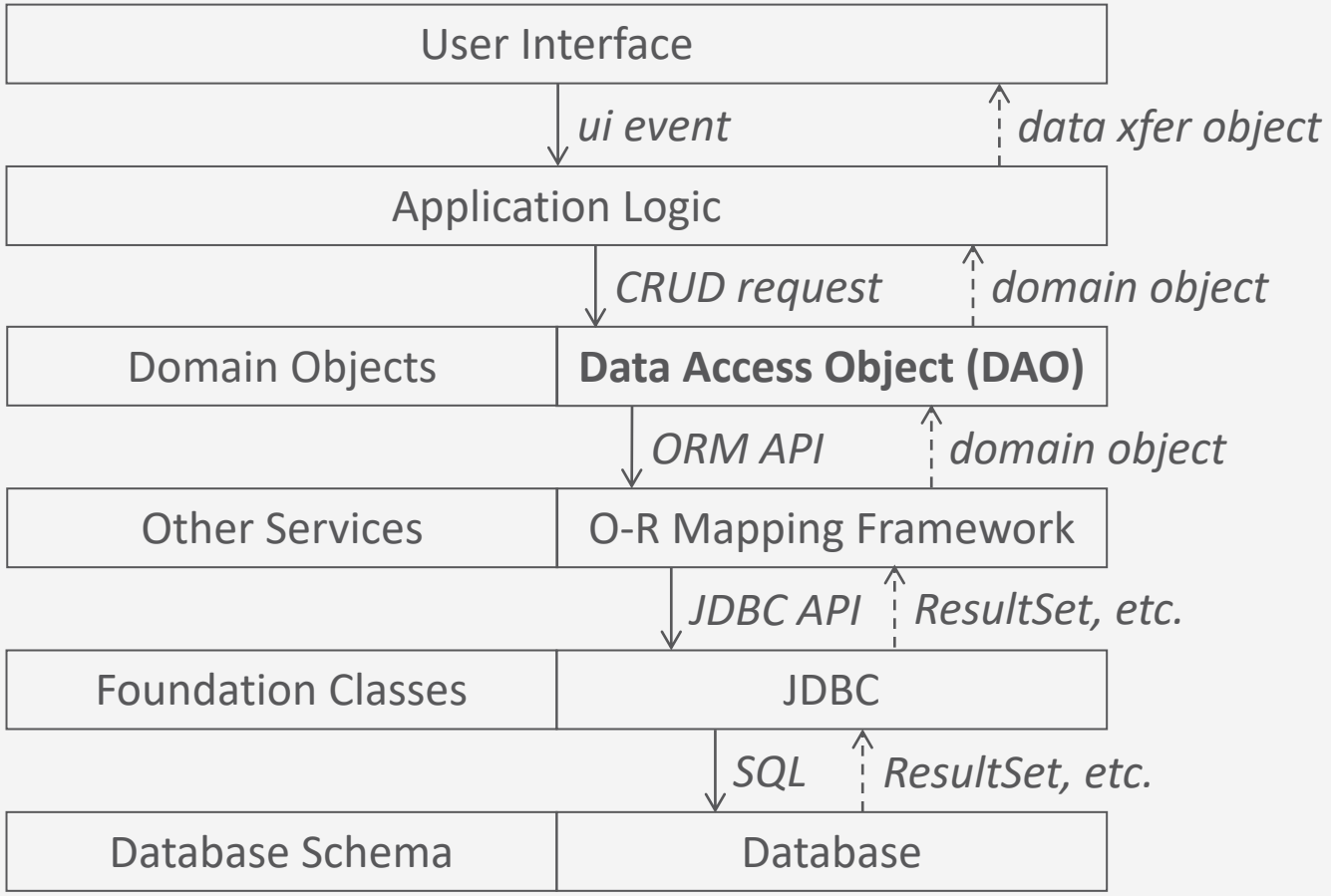
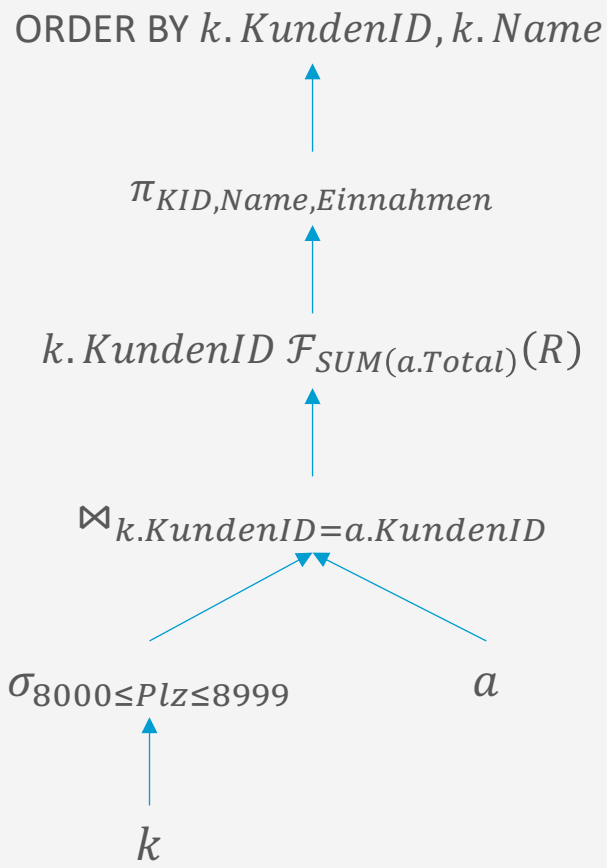
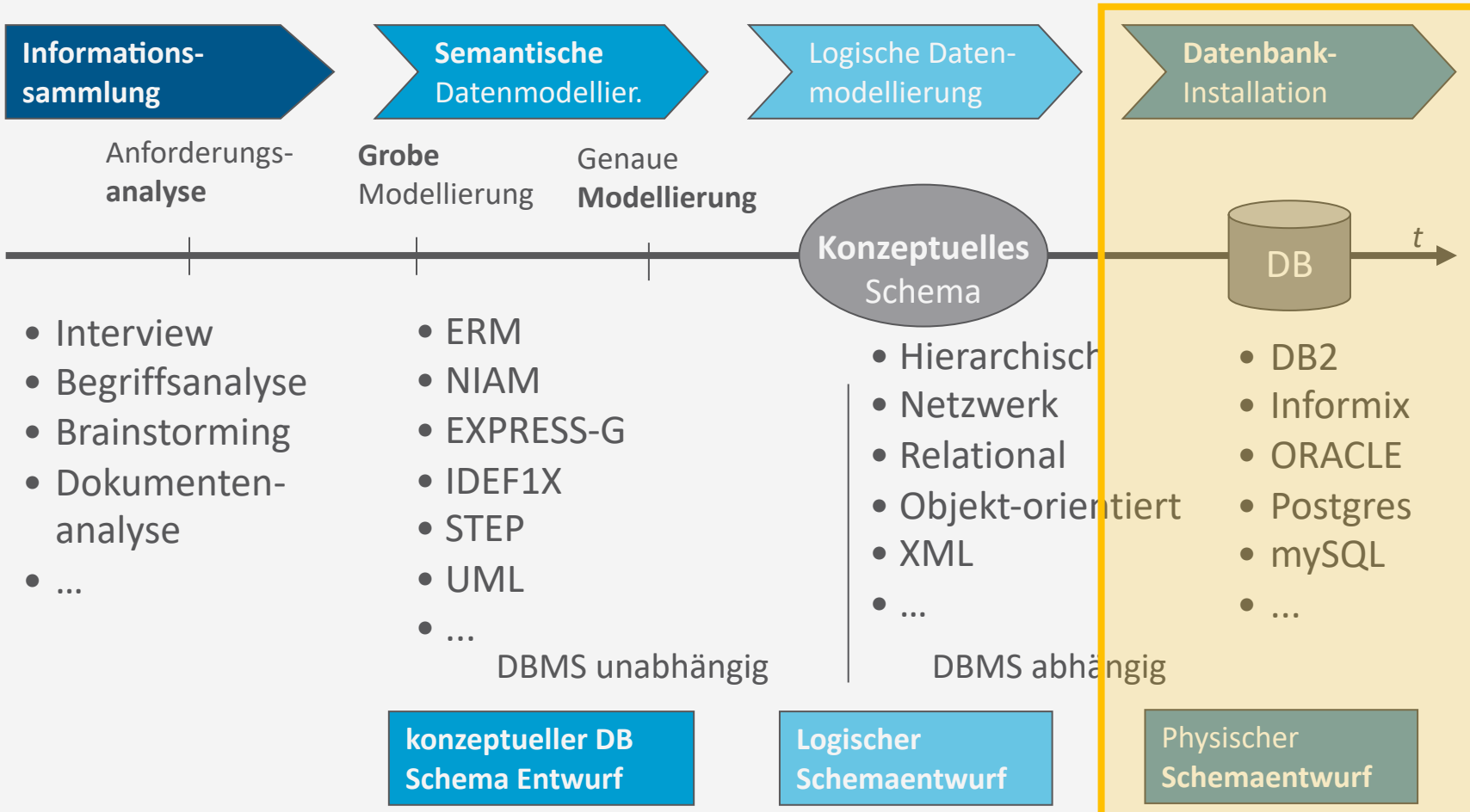
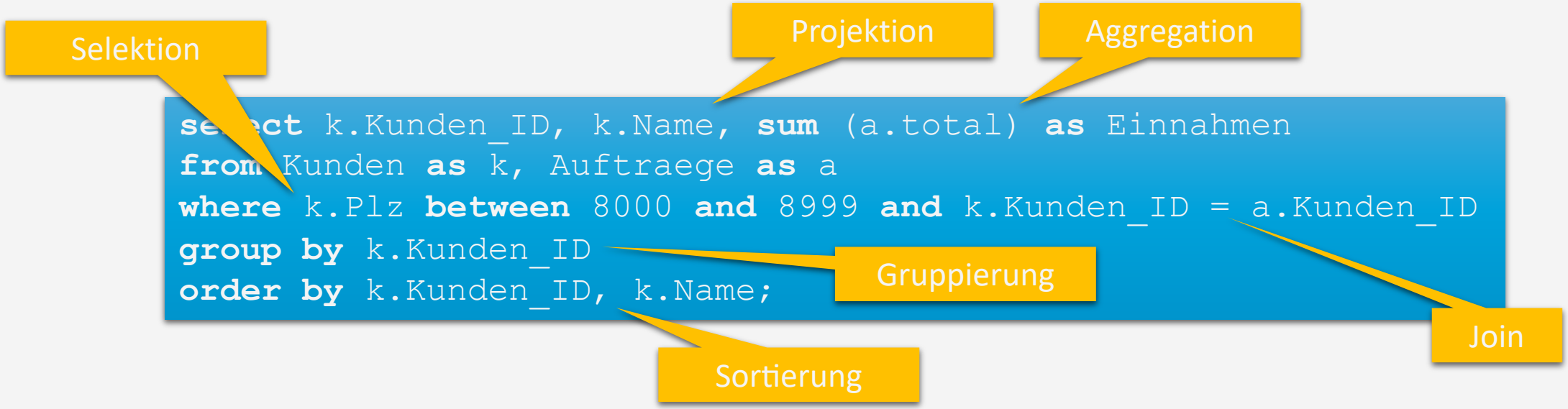
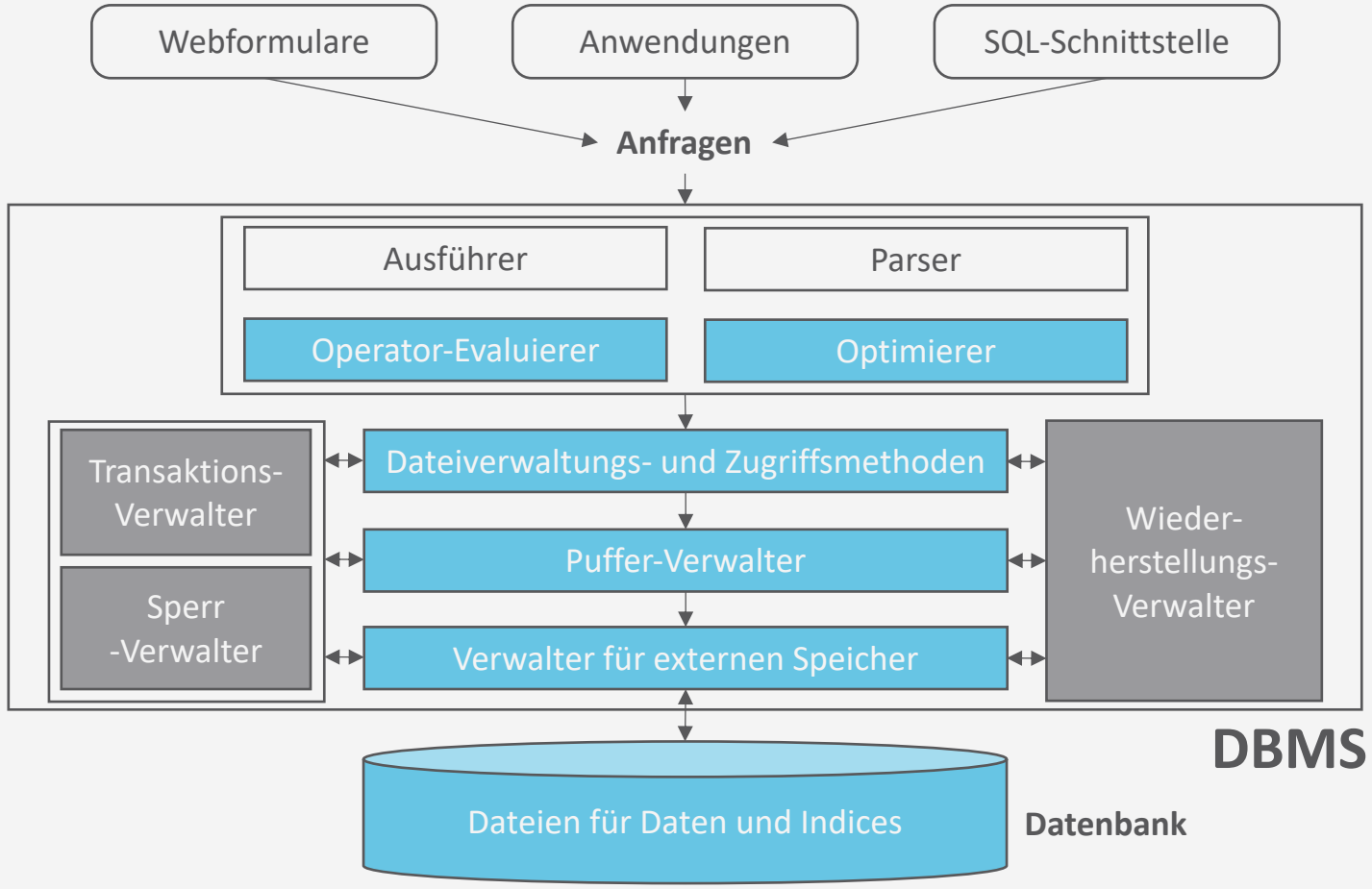
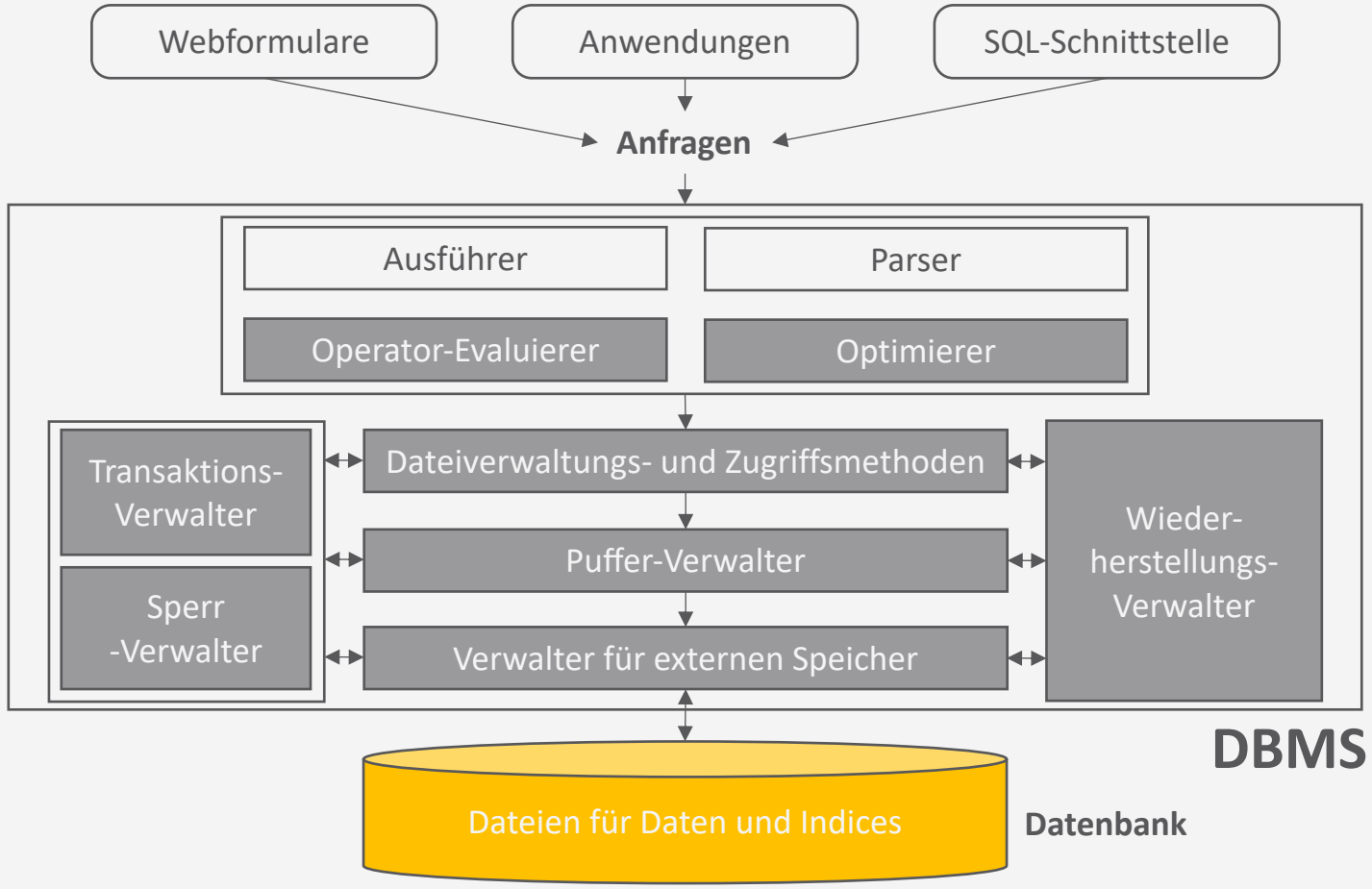
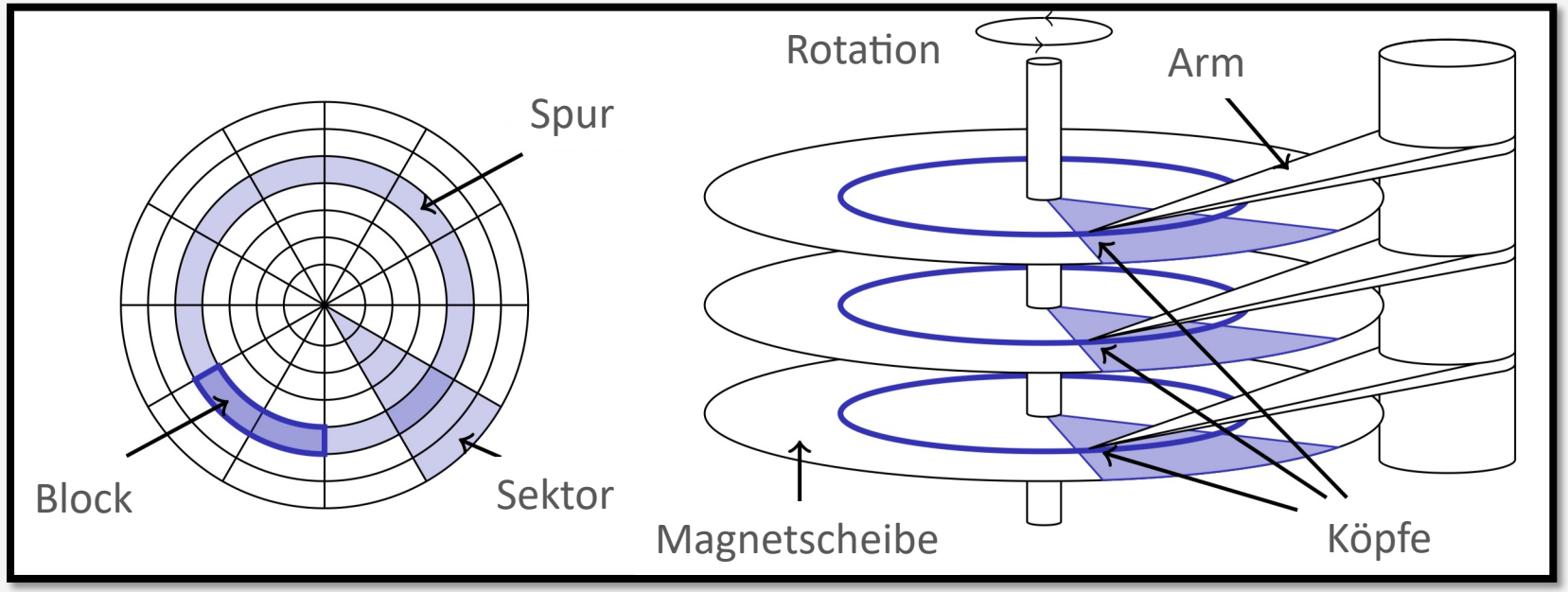
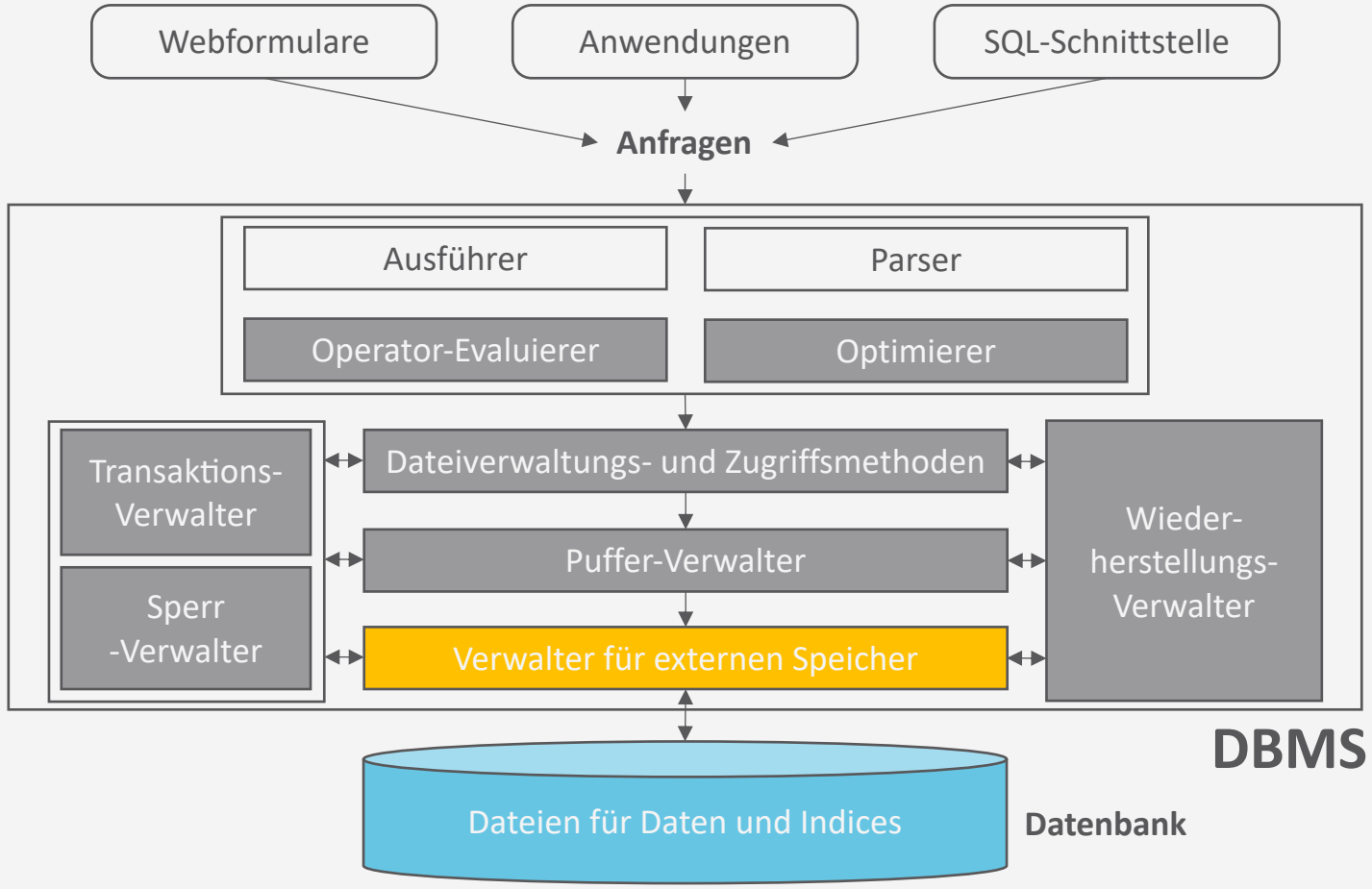
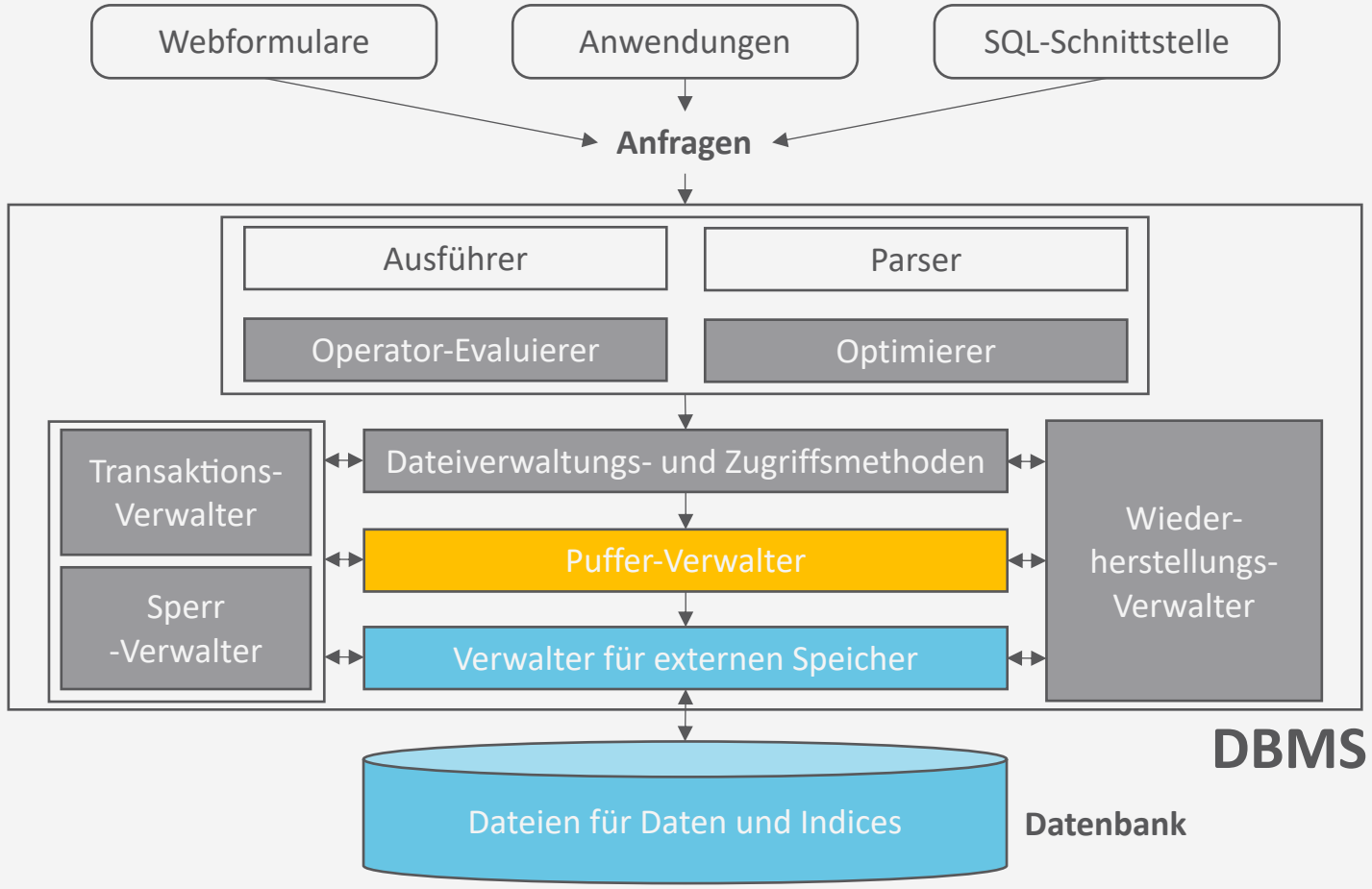
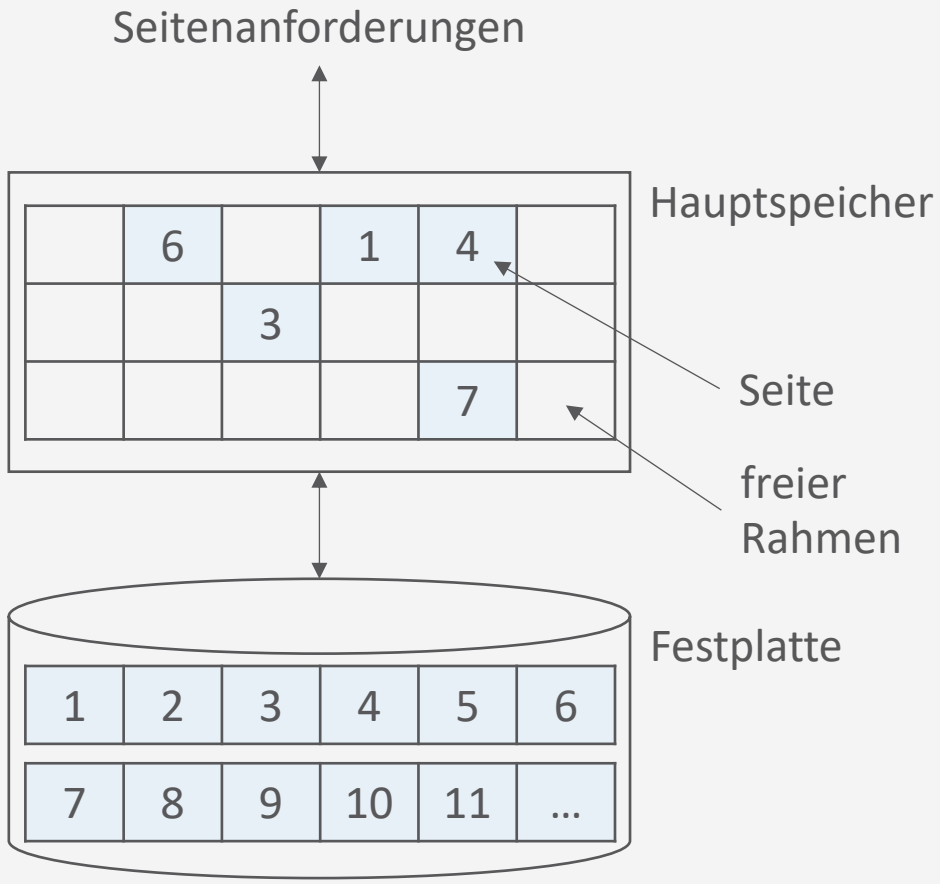
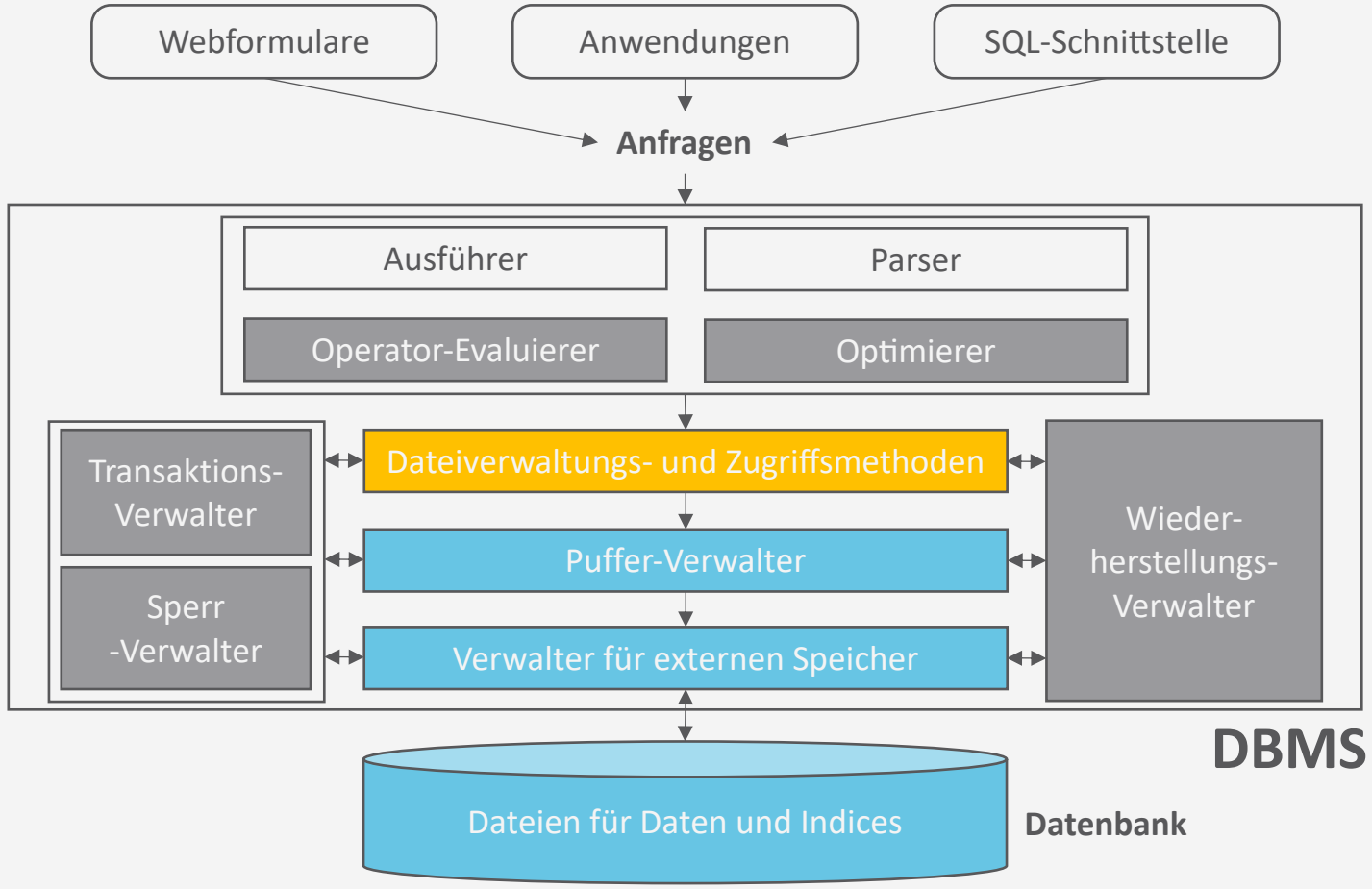
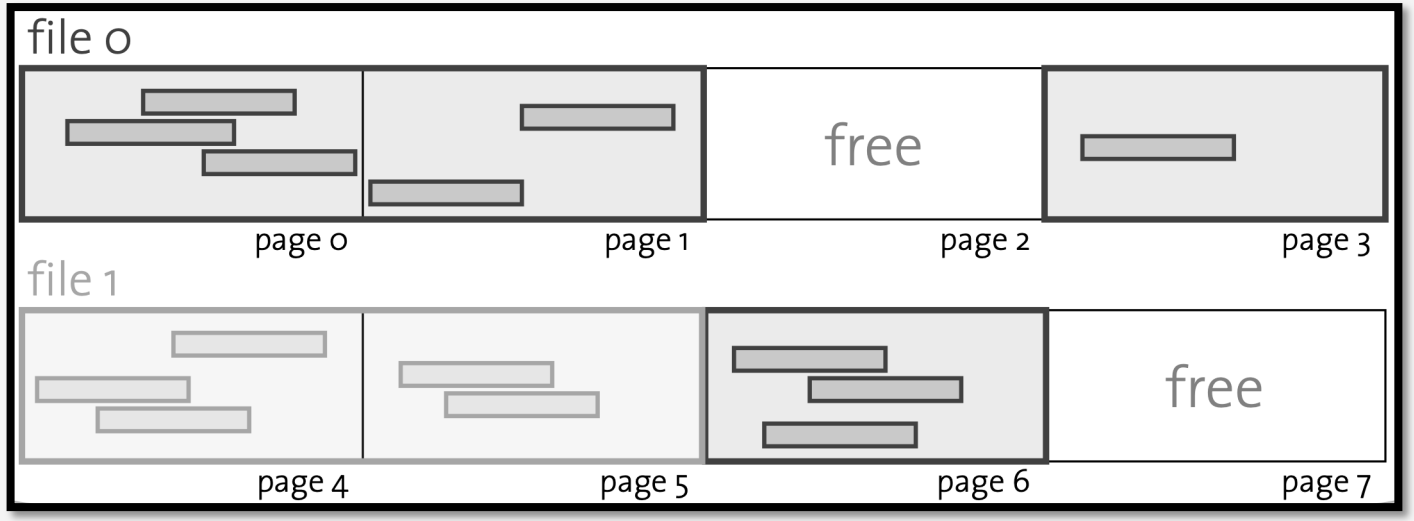
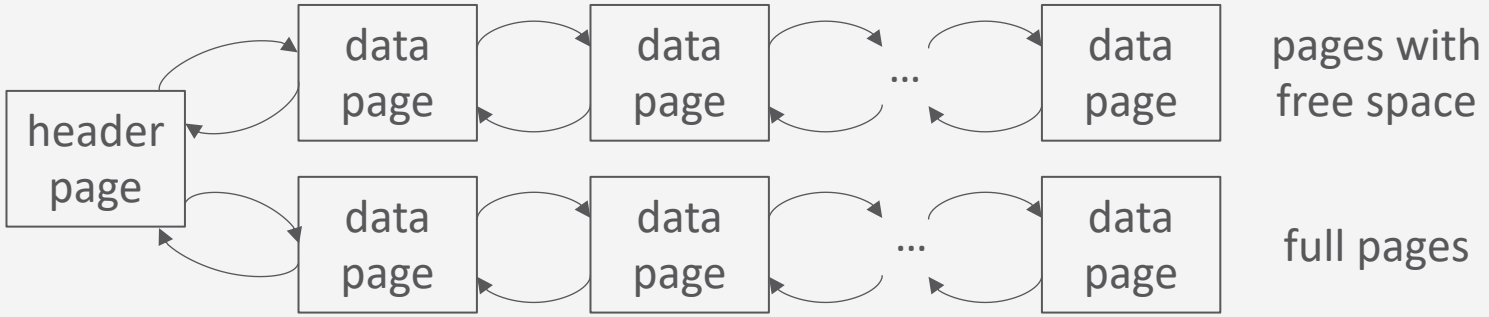
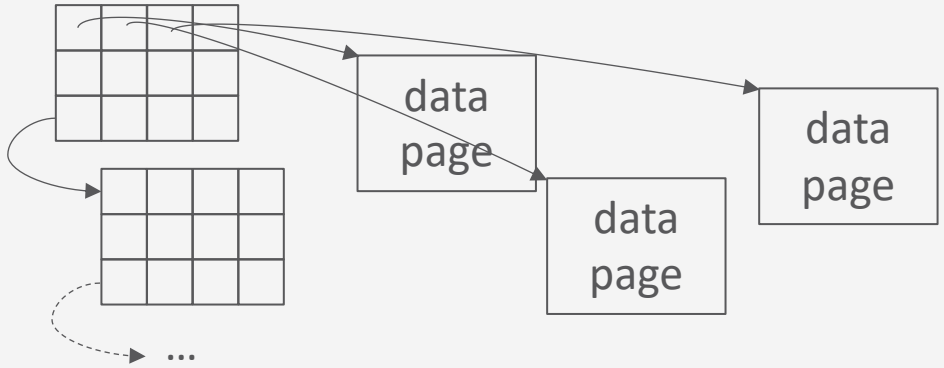
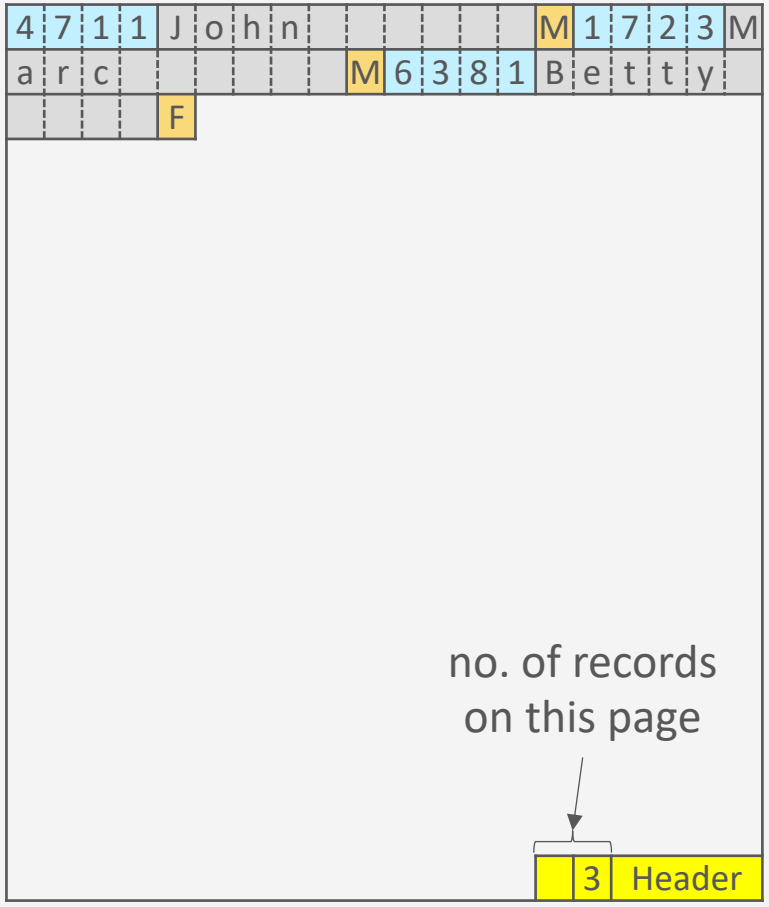
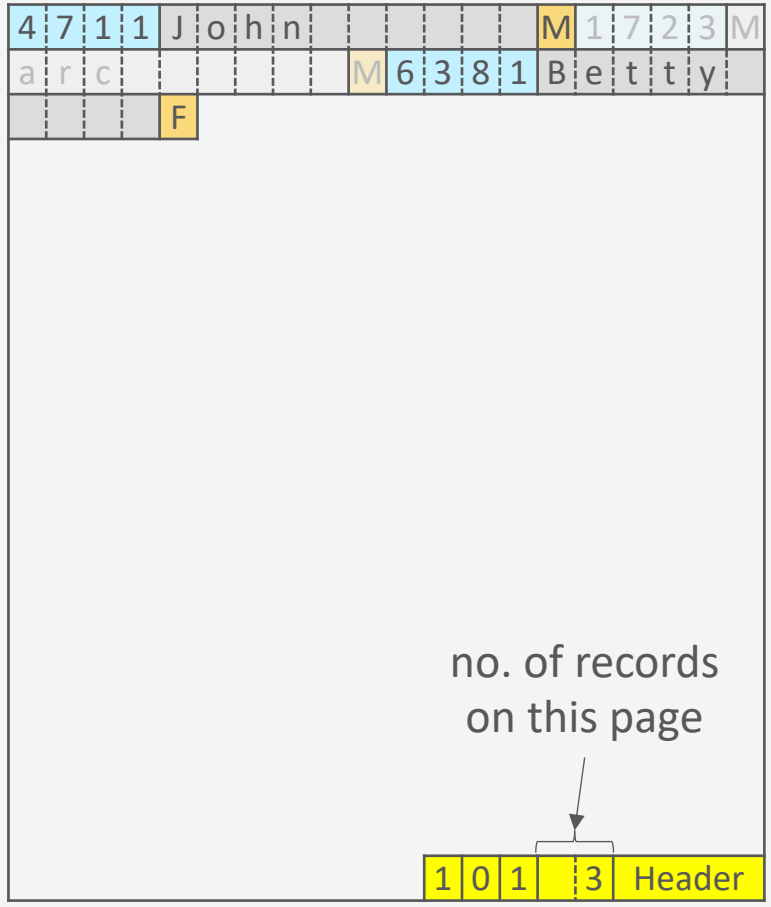
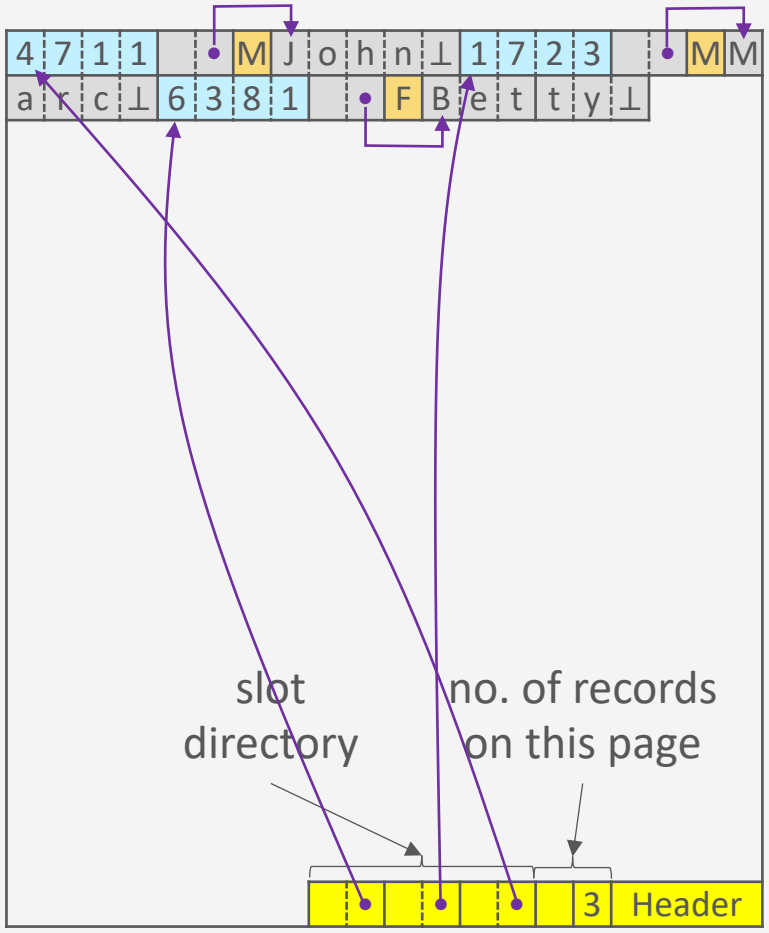
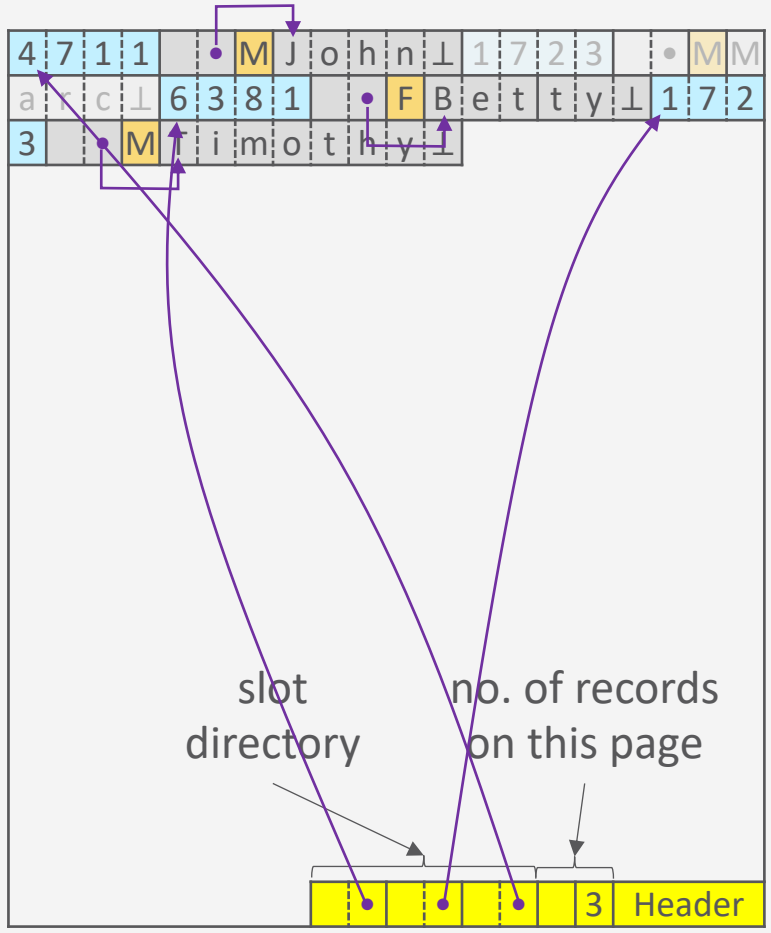
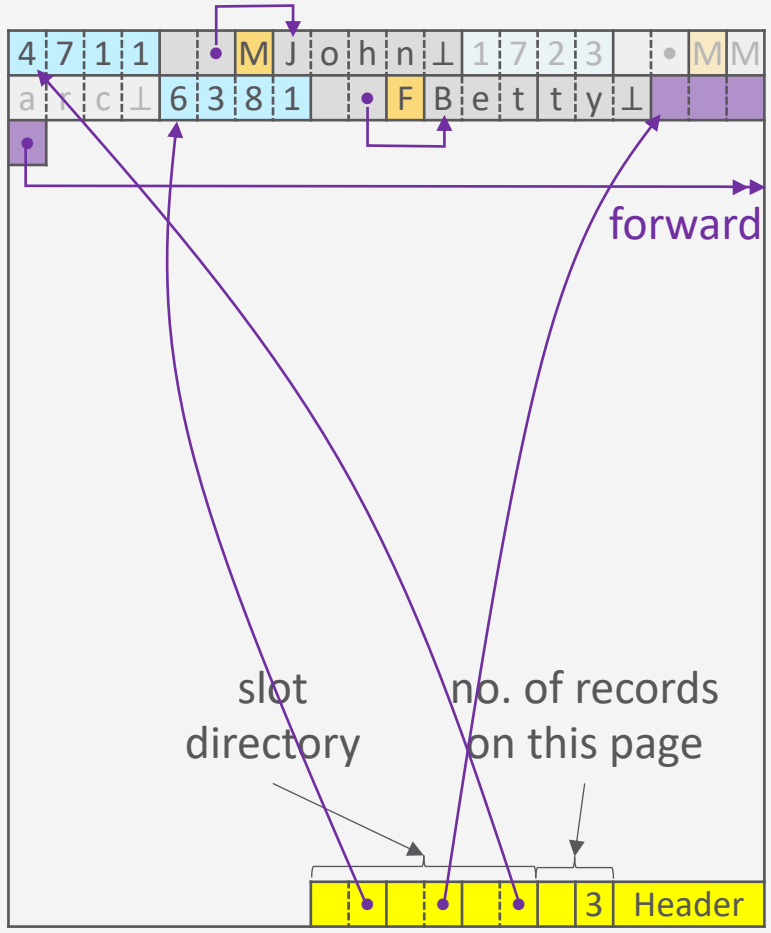
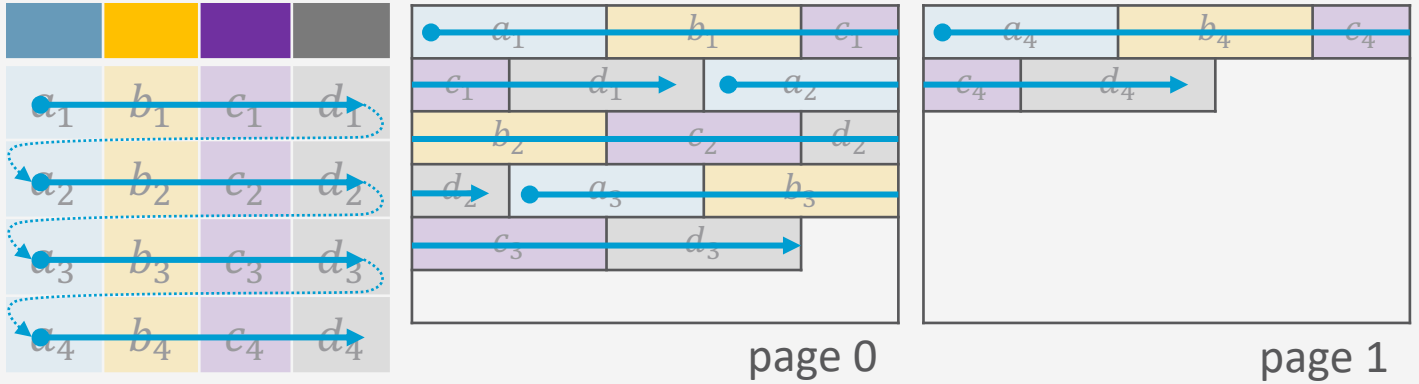
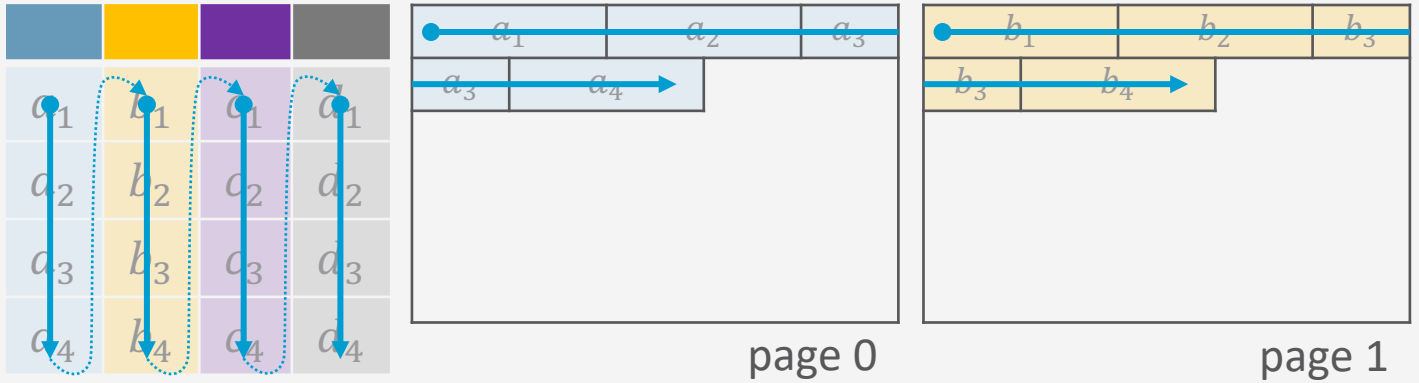
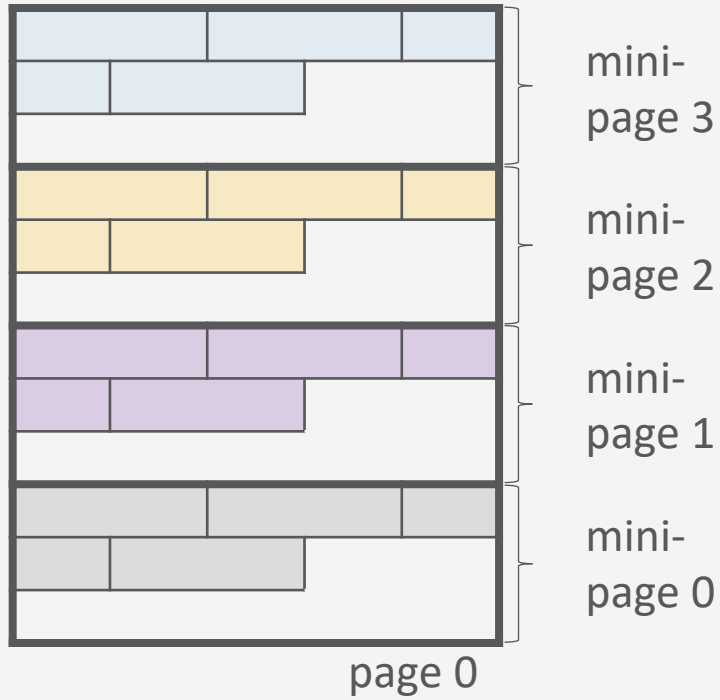
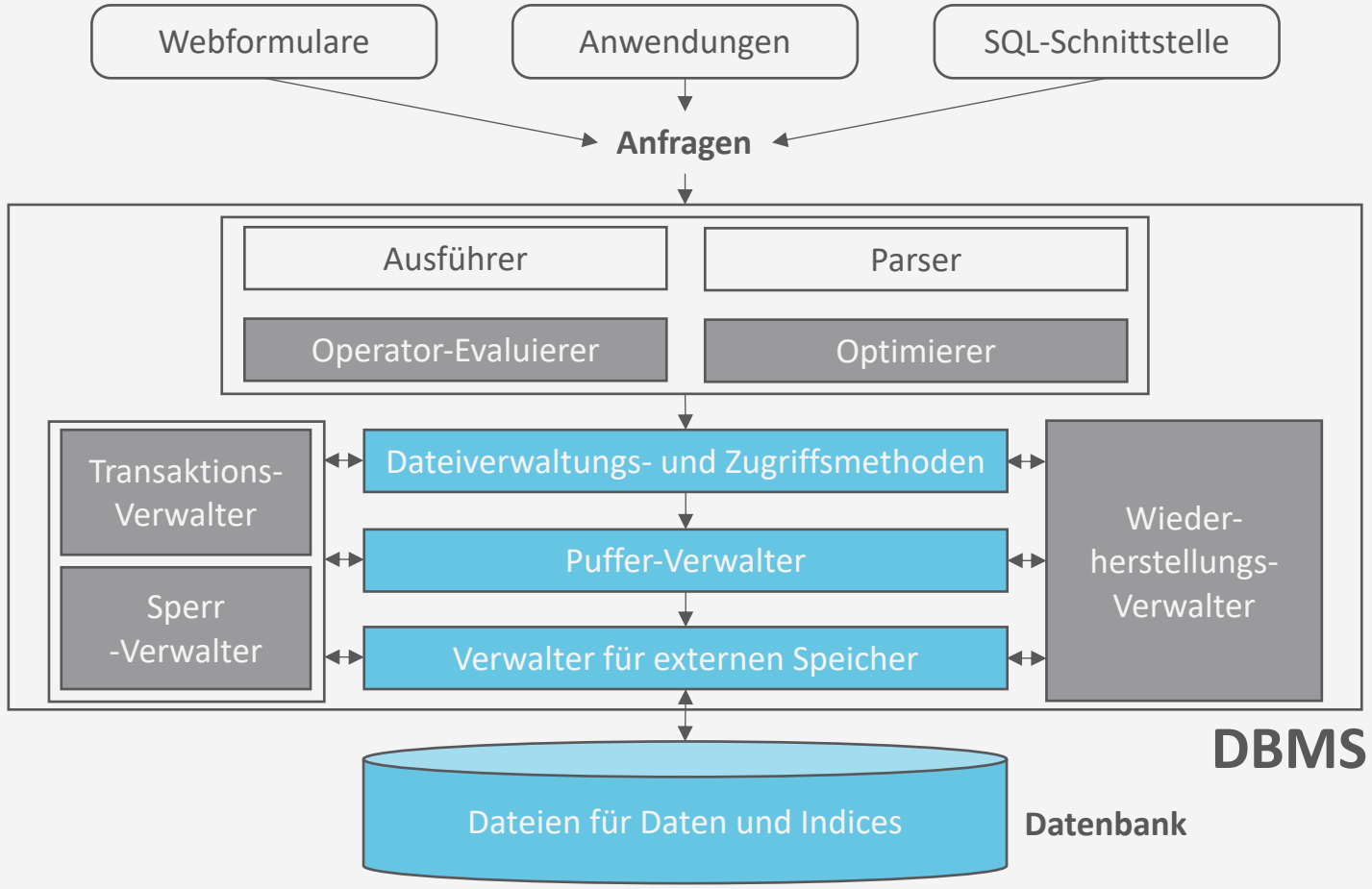
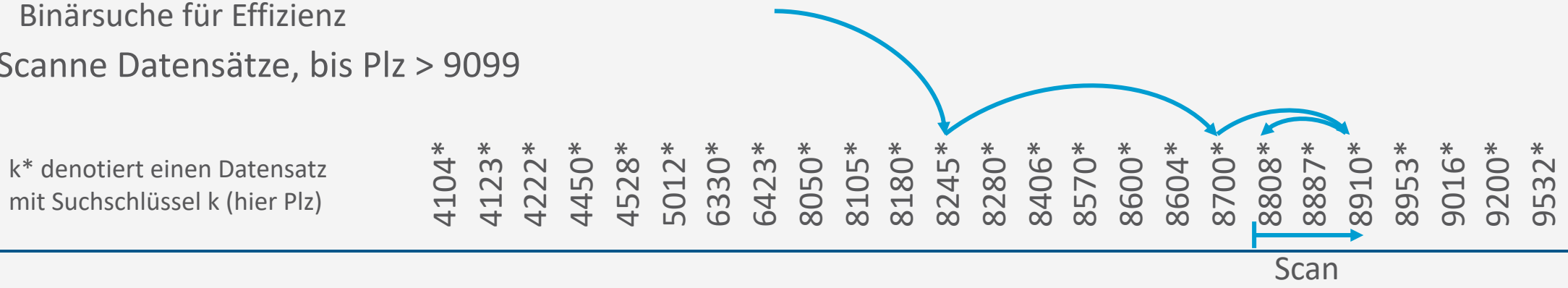
   /* check that SQLSTATE has all 0's    */
   /* and if so print the value of num   */
   if sqlstate == '00000':
      return num;
   else
      return -1:
}
```

Cursor

- Konzept, um durch Ergebnismenge zu navigieren
- 4 Schritte, um einen Cursor zu nutzen:
  1. Cursor Deklaration: **`exec sql declare`** `<cursor>` **`cursor for`** `<query>`
     - <cursor> : Name des Cursor; <query> : SQL-Ausdruck
  2. Cursor Initialisierung: **`exec sql open`** `<cursor>`
     - Initialisiert den Cursor vor dem ersten Tupel, Anfrage wird ausgeführt
  3. Tupel holen: **`exec sql fetch from`** `<cursor>` **`into`** `<variables>`
     - Holt das nächste Tupel und schreibt es in die <variables>
     - kann mehrfach ausgeführt werden
     - wenn keine Tupel mehr da sind: SQLSTATE = '02000'.
  4. Cursor schließen: **`exec sql close`** `<cursor>`

Beispiel: Cursor

```
void getAllProjects() {
   exec sql begin declare section;
      char project[4], description[50];
      char SQLSTATE[6];
   exec sql end declare section;
   exec sql declare execCursor cursor for
      select projectno, description
      from projects;
   exec sql open execCursor;
   while (1) {
      exec sql fetch from execCursor
         into :project, :description;
      if (!(strcmp(SQLSTATE, "02000")) break;
      printf("projectno: %s, description: %s",
            project, description);
   }
   exec sql close execCursor;
}
```

Dynamic SQL

- Standard für Anwendungsprogramme, die SQL-Statements zur Laufzeit erstellen und absenden
- Es ist also der DB vorab unbekannt …
  - Ob ein Statement Daten holen oder speichern will
  - Wie viele Variablen benutzt werden, und welchen Typ sie haben
- Eigenschaften der SQL-Statements durch Deskriptor beschreibbar
- Zwei Möglichkeiten:
  - Execute Immediate:
    - Statement wird direkt ausgeführt
  - Prepare and Execute:
    - Das gleiche Statement wird mehrfach ausgeführt (mit verschiedenen Parametern)
    - Zwischenergebnisse der Vorbereitung werden behalten (z.B. Ausführungsplan)

(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

Dynamic SQL: Beispiel

```
/* execute statement only once */
exec sql execute immediate "UPDATE projects
                            SET budget = 10 000 000
                            WHERE projectno = 'PJ47'";

/* prepare and execute statement */
dynstmt = "DYN1";
temp = "UPDATE projects
        SET budget = 1 000 000
        WHERE projectno = ?";
exec sql prepare :dynstmt from :temp;

prjno = "PJ47";
exec sql execute :dynstmt using :prjno;
```

Nachteile der Dynamik: SQL Code Injection




```
create procedure search_orders
       @custid    nchar(5)   = null,
       @shipname nvarchar(40) = null as
declare @sql nvarchar(4000)

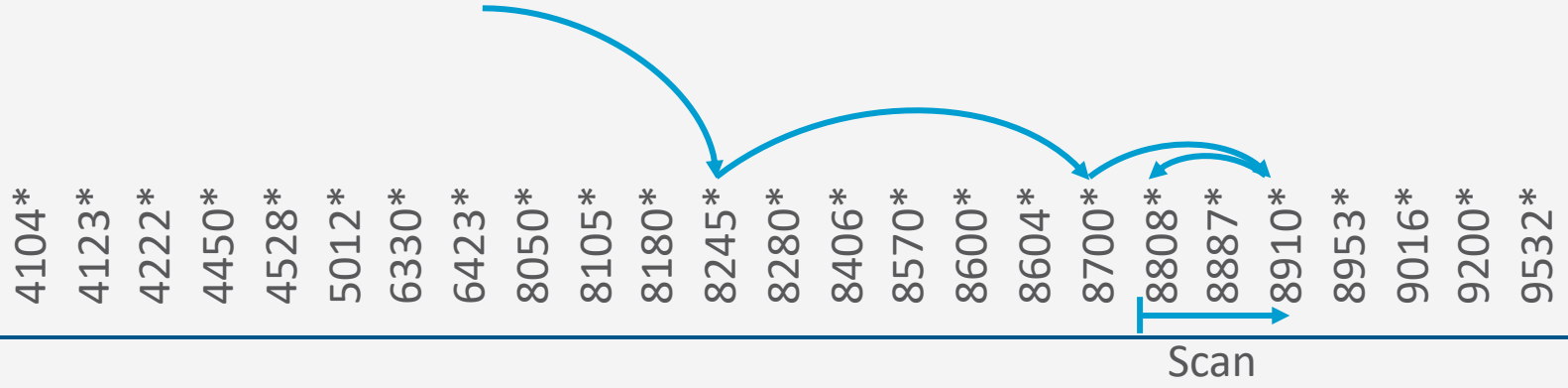
select @sql =' SELECT * ' +
            ' FROM dbo.Orders WHERE 1 = 1 '

if @custid is not null
       select @sql = @sql +
            ' AND CustomerID LIKE ''' + @custid + '''' 
if @shipname is not null
       select @sql = @sql +
            ' AND ShipName LIKE ''' + @shipname + ''''
exec(@sql)
```

Enter shipname: Sea Lion

➔

```
exec(
select * from dbo.Orders
where 1 = 1 and ShipName like
'Sea Lion'
)
```

Enter shipname: `;drop table orders;

➔

```
exec(
select * from dbo.Orders
where 1 = 1 and ShipName like
'';
drop table ORDERS;
)
```

Module Language

- Anwendungsprogramm und SQL-Statements werden getrennt
  - Modul enthält Methoden und Deklarationen von Cursorn und temporären Tabellen, wird in einer Datenbank gespeichert
  - Anwendung kann die Methoden des Moduls aufrufen
  - Sog. Linker kombiniert SQL Statements und Anwendungsprogramm
- Beispiel rechts

```
module projects_module
   names are ascii     language C
   schema user_schema authorization user

procedure num_projects
   (  :budget     integer,
      :num        integer, sqlstate )
   select   count(*)
   into     :num
   from     projects
   where    budget >= :budget;
...
```


(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

Call-Level APIs

- ODBC (Open Database Connectivity)
  - Basiert auf informellem DBMS-Hersteller-Standard aus 1992
  - Microsoft adaptiert die Schnittstelle und nennt es ODBC
- SQL/CLI
  - Formales Konsortium (SQL Access Group) übernimmt die Entwicklung, nennt es CLI (Call-Level Interface)
  - Ergebnis wurde als Teil des SQL-92 Standards 1995 veröffentlicht, ist Teil 3 von SQL:1999
  - ODBC ist SQL/CLI sehr ähnlich
- JDBC (Java Database Connectivity)
  - Schnittstelle speziell für Java Anwendungen
  - JDBC wurde durch allgemeine APIs wie ODBC und SQL/CLI stark beeinflusst

(J. Melton, A. R. Simon: SQL:1999 Understanding Relational Language Components, 2002)

Beispiel: JDBC

```
class Employee {
   public static void main (String args []) throws SQLException {
      // Load the Oracle JDBC driver
      DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
      Connection conn = DriverManager.getConnection
         ("jdbc:oracle:thin:" + "@cip-s.kbs.uni-hannover.de:1521:dbs1",
               "scott", "tiger9i");
      Statement stmt = conn.createStatement();
      // Select the ENAME column from the EMP table
      ResultSet rset = stmt.executeQuery("select ENAME from EMP");
      // Iterate through the result and print the employee names
      while (rset.next())
         System.out.println(rset.getString(1));
      rset.close();
      stmt.close();
      conn.close();
   }
}
```

Object-relational Mappings (ORM)

- Ziel:
  - Persistierung in objektorientierter Anwendung
  - Framework verbirgt SQL vor Entwickler
  - Mapping: OO-Datenobjekte → Datenbankschemata und passende SQL-Statements





Zwischenzusammenfassung

- Views
  - Abgeleitete Relation
- Rechtevergabe
  - GRANT, REVOKE
- Programmiermethoden
  - Direkter Aufruf: Von SQL an die Datenbank
  - Embedded SQL: SQL wird in die Host-Sprache eingebunden
  - Dynamic SQL: Wird zur Programmlaufzeit zusammengebaut
  - Module Language: SQL wird in Module ausgelagert, die von Host-Sprache aus angefragt werden
  - Call-Level APIs: Schnittstellen, um aus der Host-Sprache Datenbanken anzusprechen
  - Mappings: Verbergen SQL vor Programmierer

Überblick: 5. Structured Query Language (SQL)

A. Datendefinition (SQL als DDL)
- Schema, Tabellen, Datentypen, Constraints definieren
- Strukturelle Änderungen mittels drop, alter

B. Datenmanipulation (SQL als DML)
- Anfragen
- Datenänderungen

C. Und der Rest
- Sichten (SQL als VDL)
- Rechtevergabe (SQL als DCL)
- Programmiermethoden

→ Anfragenverarbeitung

WWU
MÜNSTER

Anfrageverarbeitung

Datenbanken



Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Danksagung

- Folien basieren ursprünglich auf dem Kurs

  „Architecture and Implementation of Database Systems“
  von Jens Teubner an der ETH Zürich

- Graphiken wurden mit Zustimmung des Autors aus diesem Kurs übernommen

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Anfragebeantwortung

- DBMS muss eine Menge von Aufgaben erledigen
  - Mit minimalen Ressourcen
  - Über großen Datenmengen
  - So schnell wie möglich



Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
  - Zugriff
- Anfrageverarbeitung
  - Operator-Evaluierer
  - Optimierer
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



Speicherung

- Daten in Datenbanken sind in der Regel
  - Persistent
  - **Zu groß um in den Hauptspeicher zu passen**

- Anforderung an Speicherung
  - Entsprechend große Menge an Speicherplatz
  - Schneller Zugriff vs. vertretbare Kosten
  - Sicherung der Daten gegeben (Totalverlust inakzeptabel)

- Wir schauen uns Festplattenspeicher als Beispiel an
  - Ob Festplattenspeicher, Netzwerkspeicher oder anderes → Bottlenecks / Anforderungen kennen

Speicherhierarchie

	Kapazität	Latenz
• CPU (mit Registern)	Bytes	< 1 ns
• Cache-Speicher	Kilo-/Mega-Bytes	< 10 ns
• Hauptspeicher	Giga-Bytes	20-100 ns
• Flash-Speicher / SSD	Giga/Tera/Peta-Bytes	30-250 µs
• Festplatte	Tera/Peta-Bytes	3-10 ms
• Bandautomat	Peta-Bytes	variierend

- Zur CPU: Schnell aber klein
- Zur Peripherie: Langsam aber groß
- Cache-Speicher zur Verringerung der Latenz
- Blockweises Lesen/Schreiben ab Flash/SSD (Block etwa 4K)

Magnetische Platten / Festplatten

- Schrittmotor positioniert Arme auf bestimmte Spur
- Magnetscheiben rotieren ständig



Zugriffszeit bei Festplatten

- Konstruktion der Platten hat Einflüsse auf Zugriffszeit (lesend und schreibend) auf einen Block
  1. Bewegung der Arme auf die gewünschte Spur (Suchzeit t_s)
  2. Wartezeit auf gewünschten Block bis er sich unter dem Arm befindet (Rotationsverzögerung t_r)
  3. Lese- bzw. Schreibezeit (Transferzeit t_{tr})
- Zugriffszeit: $t = t_s + t_r + t_{tr}$

- Beispiel: Hitachi Travelstar 7K200
  - 4 Köpfe, 2 Magnetplatten, 512 Bytes/Sektor, Kapazität: 200 GB
    - 2 Köpfe pro Platte → max. halbe Runde für Blockanfang → $t_r = 8{,}33/2\ ms = 4{,}17\ ms$
  - Rotationsgeschwindigkeit: 7200 rpm
    - $1r = 1/7200\ m = 1/120\ s = 8{,}33\ ms$
  - Mittlere Suchzeit: $10\ ms$
    - $t_s = 10\ ms$
  - Transferrate: ca. 50 MB/s
    - $t_{tr} = \frac{8KB}{50MB/s} = 0{,}16\ ms$
  - Zugriffszeit auf einen Block von 8 KB?
    - $t = 10\ ms + 4{,}17\ ms + 0{,}16\ ms = 14{,}33\ ms$

Sequentieller vs. Wahlfreier Zugriff

- Fortsetzung Travelstar Beispiel: Lese 1000 Blöcke von je 8 KB (8MB)
  - Wahlfreier Zugriff:
    - $t_{rand} = 1000 \cdot 14{,}33\ ms = 14.330\ ms$
  - Sequentieller Zugriff:
    - 63 Sektoren / Spur, Track-to-Track-Suchzeit $t_{s,t2t}$ (von einer Spur zur nächsten Spur): $1\ ms$
    - Ein Block mit 8 KB benötigt 16 Sektoren (8KB/512 B/Sektor): 16.000 Sektoren lesen

    → Bei 63 Sektoren pro Spur macht das 16000/63 ≈ 254 Spuren
    - $t_{seq} = t_s + t_r + 1000 \cdot t_{tr} + 254 \cdot t_{s,t2t} \approx 10ms + 4{,}17ms + 1000 \cdot 0{,}16ms + 254 \cdot 1\ ms \approx 428ms$
- Einsicht: Sequentieller Zugriff **viel** schneller als wahlfreier → Vermeide wahlfreie I/O, wenn möglich
  - Wenn $428\ ms / 14330\ ms \approx 3\%$ einer 8MB Datei wahlfrei benötigt wird, kann man gleich die ganze Datei lesen, sofern Blöcke hintereinander stehen

Speichernetzwerk (Storage Area Network, SAN)

- Block-basierter Netzwerkzugriff auf Speicher
  - Als logische Platten betrachtet (Suche Block 4711 von Disk 42)
    - SAN-Speichergeräte abstrahieren von RAID oder physikalischen Platten und zeigen sich dem DBMS als logische Platten
    - Hardwarebeschleunigung und einfachere Verwaltung
- Üblicherweise lokale Netzwerke mit multiplen Servern, Speicherressourcen
  - Bessere Fehlertoleranz und erhöhte Flexibilität
- Alternative: Cloud-Speicher
  - Cluster von vielen Standard-PCs (z.B. Google, Amazon)
    - Systemkosten vs. Zuverlässigkeit und Performanz
    - Verwendung massiver Replikation von Datenspeichern
  - CPU-Zyklen und Disk-Kapazität als Service
    - Amazons „Simple Storage System (S3)“
      - Latenz: 100 ms bis 1s!
      - Datenbank auf Basis von S3 entwickelt in 2008

Architektur eines DBMS

- Speicherung
  - Speichermedien
    - Datenbanken in der Regel nicht im Hauptspeicher vorhaltbar
    - Wahlfreier Zugriff teuer im Vergleich zu sequentiellem Zugriff
  - Verwaltung
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



Verwaltung des externen Speichers

- Abstraktion von technischen Details der Speichermedien
- Konzepte der Seite (page) mit typischerweise 4-64KB als Speichereinheiten für die restlichen Komponenten
- Verzeichnis für Abbildung

  Seitennummer → Physikalischer Speicherort

  wobei der physikalische Speicherort
  - eine Betriebssystemdatei inkl. Versatz,
  - eine Angabe Kopf-Sektor-Spur einer Festplatte oder
  - eine Angabe für Bandgerät und -nummer inkl. Versatz

  sein kann

Leere Seiten

- Leere Seiten
  - Insert: Finde leere Seite, die das Datenobjekt speichern kann
  - Delete: Seite wird frei
- Verwaltung leerer Seiten:
  1. Liste der freien Seiten
     - Hinzufügen, falls Seite nicht mehr verwendet
  2. Bitmap mit einem Bit für jede Seite
     - Umklappen des Bits p, wenn Seite p (de-)alloziert wird
     - Finden von hintereinanderliegenden Seiten einfacher
- Persistent als Verwaltungsinformationen zu speichern

Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
    - Seiten zur Referenz von Speichereinheiten
  - Puffer
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



Puffer-Verwalter

- Vermittelt zwischen externem und internem Speicher (Hauptspeicher)
- Verwaltet hierzu einen besonderen Bereich im Hauptspeicher, den Pufferbereich (buffer pool)
- Externe Seiten in Rahmen des Pufferbereichs laden
- Ersetzungsstrategien für den Fall, dass der Pufferbereich voll ist



Schnittstelle zum Puffer-Verwalter

- Funktion pin für Anfragen nach Seiten
  - $\mathrm{pin}(pageno)$
    - Anfrage nach Seitennummer $pageno$
    - Lade Seite in Hauptspeicher falls nötig
    - Rückgabe einer Referenz auf $pageno$
- Funktion unpin für Freistellungen von Seiten nach Verwendung
  - $\mathrm{unpin}(pageno, dirty)$
    - Freistellung einer Seite $pageno$ zur möglichen Auslagerung
    - $dirty = \mathrm{true}$ bei Modifikationen der Seite

Implementation von pin()

```
function pin(pageno)
    if buffer pool already contains pageno then
        pinCount(pageno) ← pinCount(pageno) + 1
        return address of frame holding pageno
    else
        Select a victim frame v using the replacement policy
        if dirty(v) then
            Write v to disk
        Read page pageno from disk into frame v
        pinCount(pageno) ← 1
        dirty(pageno) ← false
        return address of frame v
```

Wofür nötig?

Implementation von unpin()

function unpin($pageno, dirty$)
    $pinCount(pageno) \leftarrow pinCount(pageno) - 1$
    if $dirty$ **then**
        $dirty(pageno) \leftarrow dirty$

Warum werden Seiten nicht gleich beim unpin zurückgeschrieben?

Ersetzungsstrategien

- Least Recently Used (LRU)
  - Verdrängung der Seite mit am längsten zurückliegendem $\mathrm{unpin}()$
- LRU-k
  - Wie LRU, aber k-letztes $\mathrm{unpin}()$, nicht letztes
- Most Recently Used (MRU)
  - Verdrängung der Seite mit jüngstem $\mathrm{unpin}()$
- Random
  - Verdrängung einer beliebigen Seite
- Viele mehr...
- Seite muss pincount = 0 haben, um für Ersetzung zur Verfügung zu stehen

Was, wenn es keine Seite mit pincount = 0 gibt?

Pufferverwaltung in der Praxis

- Prefetching
  - Antizipation von Anfragen, um CPU- und I/O-Aktivität zu überlappen
    - Spekulatives Prefetching: Nehme sequentiellen Seitenzugriff an und lese im Vorwege
    - Prefetch-Listen mit Instruktionen für den Pufferverwalter für Prefetch-Seiten
- Fixierungs- oder Verdrängungsempfehlung
  - Höherer Code kann Fixierung (z.B. für Indexseiten) oder schnelle Verdrängung (bei sequentiellen Scans) empfehlen
- Partitionierte Pufferbereiche
  - Z.B. separate Bereiche für Index und Tabellen

Datenbanken vs. Betriebssysteme

- Haben wir nicht gerade ein Betriebssystem entworfen?
- Ja
  - Verwaltung für externen Speicher und Pufferverwaltung ähnlich
- Aber
  - DBMS weiß mehr über Zugriffsmuster
    - Z.B. für Prefetching
  - Limitationen von Betriebssystemen häufig zu stark für DBMS
    - Obergrenzen für Dateigrößen
    - Plattformunabhängigkeit nicht gegeben
- Gegenseitige Störung möglich
  - Doppelte Seitenverwaltung
  - DMBS-Transaktionen vs. Transaktionen auf Dateien organisiert vom Betriebssystem
  - DBMS Pufferbereiche durch Betriebssystem ausgelagert
- Daher: DBMS schaltet oft Betriebssystemdienste aus
  - Direkter Zugriff auf Festplatten
  - Eigene Prozessverwaltung

Architektur eines DBMS

- Speicherung
  - Speichermedien
  - Verwaltung
  - Puffer
    - Rahmen im Pufferbereich um Seiten aus externem Speicher in den Hauptspeicher zu laden
    - Verdrängungsstrategien
  - Zugriff
- Anfragebeantwortung
- Transaktionsmanagement



Datenbank-Dateien

- Seitenverwaltung unbeeinflusst vom Inhalt
- DBMS verwaltet Tabellen von Tupeln, Indexstrukturen, ...
- Tabellen sind Dateien von Datensätzen (*records*)
  - Datei besteht aus
    einer oder mehrerer Seiten
  - Jede Seite speichert
    einen oder mehrere Datensätze
  - Jeder Datensatz korrespondiert
    zu einem Tupel



Heap-Dateien

- Wichtigster Dateityp: Speicherung von Datensätzen mit willkürlicher Ordnung (konform mit SQL)
- Umsetzung: Verkettete Liste von Seiten



✓ Einfach zu implementieren

✗ Viele Seiten auf der Liste der freie Seiten (haben also noch Kapazität)

✗ Viele Seiten anzufassen bis passende Seite gefunden

Heap-Dateien

- Wichtigster Dateityp: Speicherung von Datensätzen mit willkürlicher Ordnung (konform mit SQL)
- Umsetzung: Verzeichnis von Seiten



  - Verwendung als Abbildung mit Informationen über freie Plätze (Granularität Abwägungssache)

✓ Suche nach freien Plätzen effizient

✗ Zusatzaufwand für Verzeichnisspeicher

Freispeicher-Verzeichnis

- Welche Seite soll für neuen Datensatz gewählt werden?
  - Append Only
    - Immer in letzte Seite einfügen, sonst neue Seite anfordern
  - Best-Fit
    - Alle Seiten müssen betrachtet werden, Reduzierung der Fragmentierung
  - First-Fit
    - Suche vom Anfang, nehme erste Seite mit genug Platz
    - Erste Seiten füllen sich schnell, werden immer wieder betrachtet
  - Next-Fit
    - Verwalte Zeiger und führe Suche fort, wo Suche beim vorigen Male endete

Inhalte einer Seite

- Für jeden Datensatz ergibt sich eine Datensatz-Kennung (*record identifier*, Abkürzung rid), typisch:
$$rid = \langle pageno, slotno \rangle$$
  - Slot: Speicherplatz für einen Datensatz
  - Datensatz-Position (Versatz auf der Seite):
$$slotno \cdot \text{Bytes pro Slot}$$
    - Beginnend bei 0 (sowohl erstes Bit auf der Seite, als auch Slotnummer)
    - Funktioniert so nur bei konstanter Slotgröße
- Beispiel: Angenommen Seite sei 42
  - Referenz Datensatz mit ID 6381:
    - Kennung ist $\langle 42,2 \rangle$

ID	Name	Sex
4711	John	M
1723	Marc	M
6381	Betty	F



Inhalte einer Seite

- Datensatz gelöscht → rid sollte sich nicht ändern
  - Slot-Verzeichnis (Bitmap) markiert durch 1/0, ob Datensatz noch gültig
- Beispiel Fortsetzung:
  - Nach Löschung von Datensatz mit ID 1723
  - Referenz Datensatz mit ID 6381
    - Kennung ist immer noch $\langle 42,2 \rangle$

ID	Name	Sex
4711	John	M
1723	Marc	M
6381	Betty	F



Inhalte einer Seite: Felder variabler Länge

- Felder variabler Länge zum Ende verschoben
  - Platzhalter zeigt auf Position
- Dann brauchen wir ein Slot-Verzeichnis
  - Zeigt auf Start eines Feldes
  - Datensatz-Kennung bleibt bei $rid = \langle pageno, slotno \rangle$, verweist jetzt aber auf Position im Slot-Verzeichnis
  - Damit können auch Datensätze unterschiedlicher Tabellen mit unterschiedlicher Länge pro Datensatz auf einer Seite gespeichert werden

Warum?



Inhalte einer Seite: Felder variabler Länge

- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)
- Beispiel:
  - Datensatz mit ID 1723 hat Kennung $\langle 42,1 \rangle$
  - Name aktualisieren zu Timothy, was nicht in den Platz passt
  - Umkopieren an neue Stelle
    - Referenz in Slot-Verzeichnis aktualisieren
    - Kennung bleibt $\langle 42,1 \rangle$



Inhalte einer Seite: Felder variabler Länge

- Felder können auf Seite verschoben werden (z.B. wenn sich Feldgröße ändert)
- Beispiel:
  - Datensatz mit ID 1723 hat Kennung $\langle 42,1 \rangle$
  - Name aktualisieren zu Timothy, was nicht in den Platz passt
  - Umkopieren an neue Stelle
    - Referenz in Slot-Verzeichnis aktualisieren
    - Kennung bleibt $\langle 42,1 \rangle$
- Einführung einer Vorwärtsreferenz, wenn Feld nicht auf Seite passt



Alternative Seiteneinteilungen

Wann lohnt sich welche Einteilung?

- Im Beispiel wurden Datensätzen zeilenweise angeordnet:



- Spaltenweise Anordnung genauso möglich:



Alternative Seitenanordnungen

Vorgestellte Schemata heißen auch:

- Row-Store
- Column-Store

- Anwendungen für verschiedene Lasttypen und Anwendungskontexte
- Unterschiedliche Kompressionsmöglichkeiten
- Kombination möglich:
  - Unterteilung einer Seite in Miniseiten
  - Mit entsprechender Aufteilung



Zwischenzusammenfassung

- Speichermedien
  - DBs idR zu groß für Hauptspeicher
  - Wahlfreier Zugriff teuer
- Verwaltung
  - Seiten als Referenz auf Speicher
- Puffer
  - Seiten aus externem Speicher in Rahmen des Puffer zu laden
  - Verdrängungsstrategien
- Zugriff
  - Stabile Datensatz-Kennung rid zur Identifikation der Speicherposition



Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

Effiziente Evaluierung einer Anfrage

- Beispiel
  - ```
    select *
    from Kunden
    where Plz between 8800 and 9099;
    ```
  - Auswertung
    1. Sortierung der Tabelle Kunden auf der Platte (nach Plz)
       - Effizienter Sortieralgorithmus benötigt
    2. Suche Startpunkt, an dem PLZ ≥8800
       - Binärsuche für Effizienz
    3. Scanne Datensätze, bis Plz > 9099

```
function binarySearch(x, list)
    mid = list.length/2
    if x = list[mid] then
        return reference to list[mid]
    else if x > list[mid] then
        binarySearch(x, list[mid + 1: end])
    else
        binarySearch(x, list[start: mid − 1])
```


```

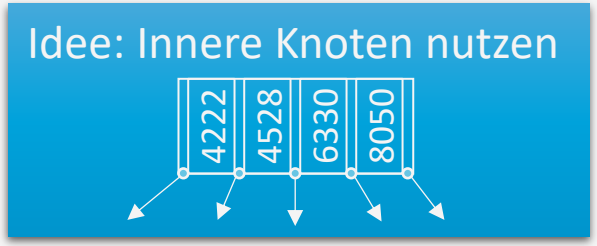
Geordnete Dateien und binäre Suche

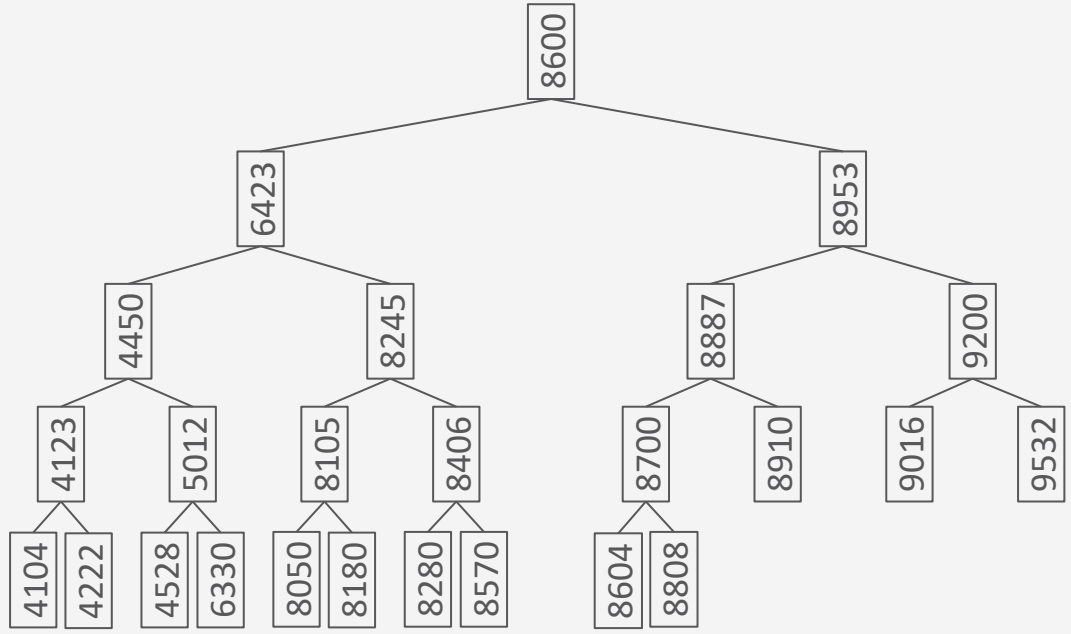
- ✓ Sequentieller Zugriff während der Scan-Phase
- ✓ $\log_2(\#Tupel)$ während der Such-Phase + entsprechende Tupel während der Scan-Phase lesen (statt insgesamt alle bei unsortierten Datensätzen; plus natürlich Sortieraufwand)
- ✗ Für jeden Zugriff während der Such-Phase eine Seite
  - Weite Sprünge sind die Idee der binären Suche, daher Datensätze während der Suche vermutlich nicht auf einer Seite



Helfen Bäume?

- Suchen innerhalb einer Seite im Hauptspeicher effizient machbar
  - Ziel: möglichst wenige Seiten aus Sekundärspeicher laden
- Bäume als Navigationsstruktur
  - Beispiel Binärbaum
    - Innere Knoten: Schlüsselwerte
    - Linker Unterbaum: Alle Schlüsselwerte kleiner
    - Rechter Unterbaum: Alle Schlüsselwerte größer
    - Tiefe bei Ausgeglichenheit: $\log_2(\#Tupel)$

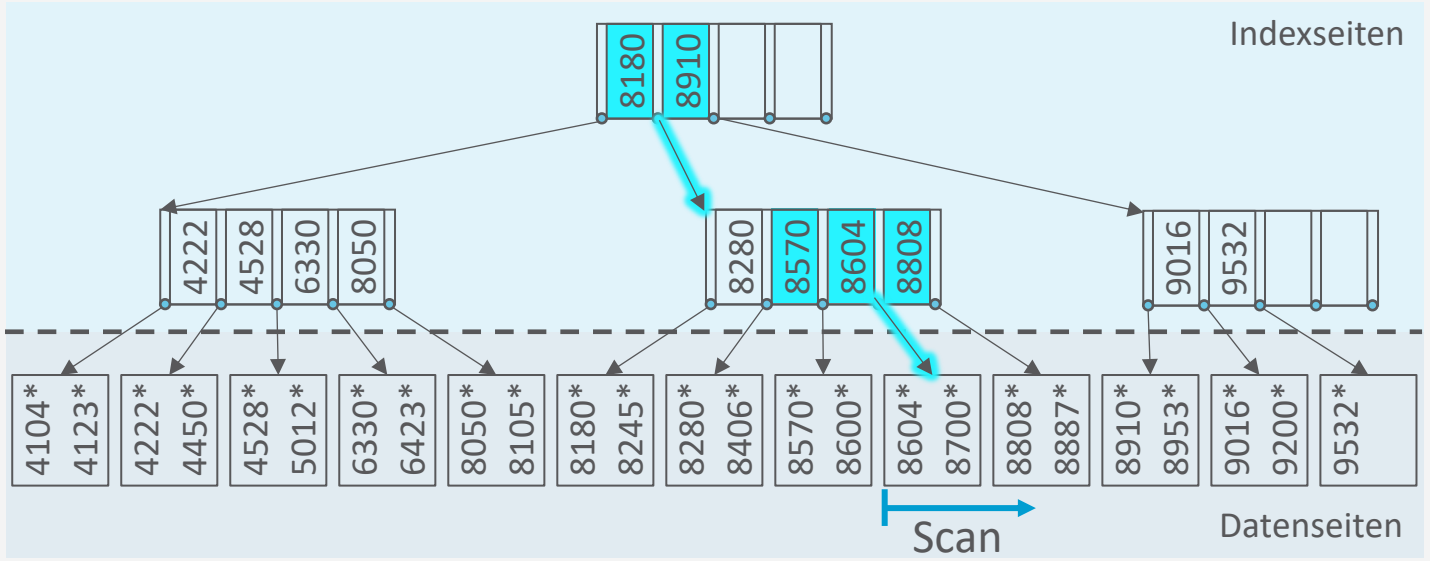
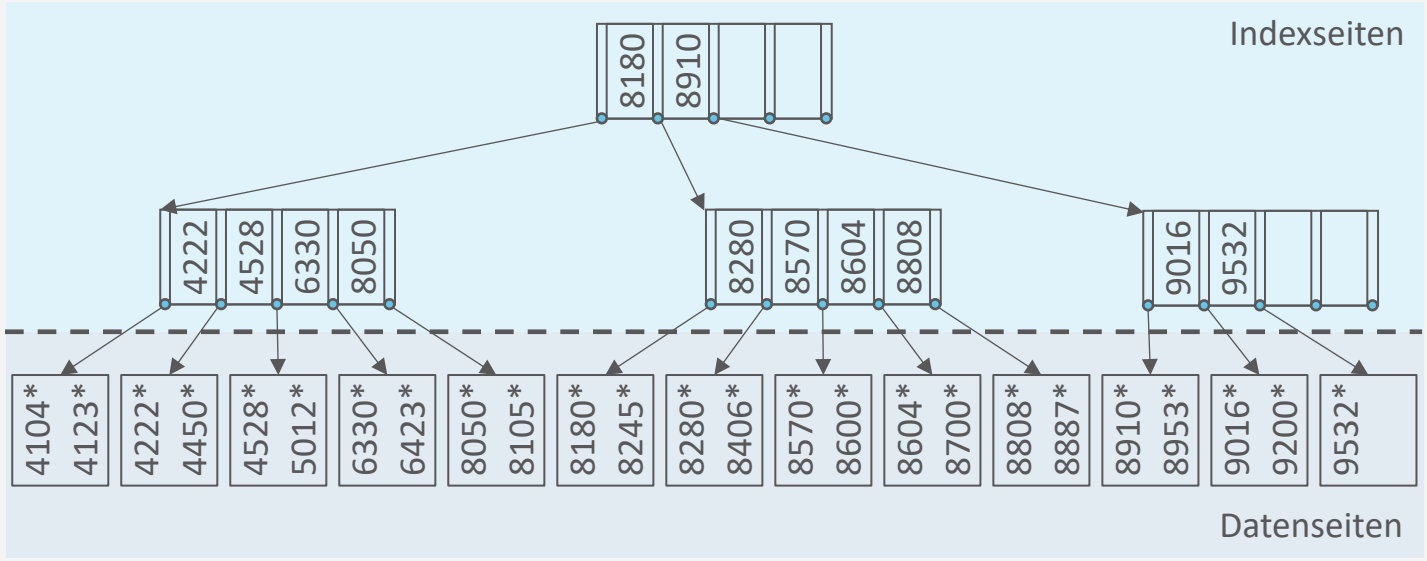
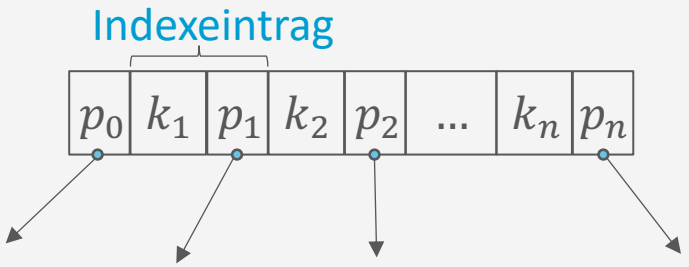
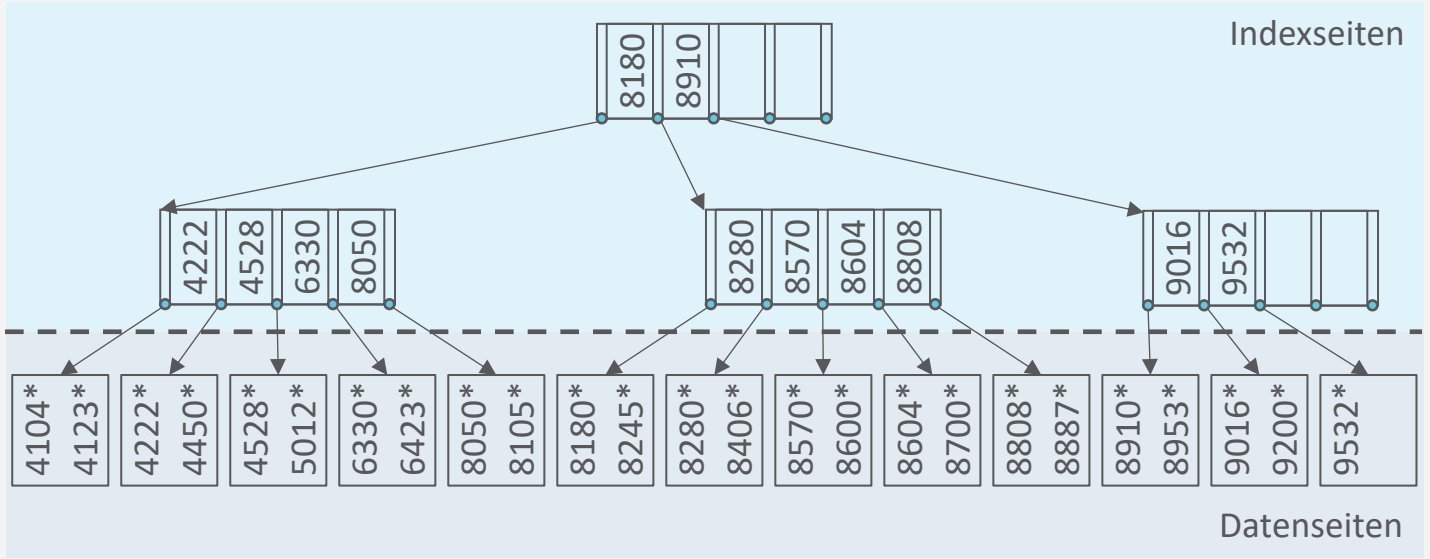
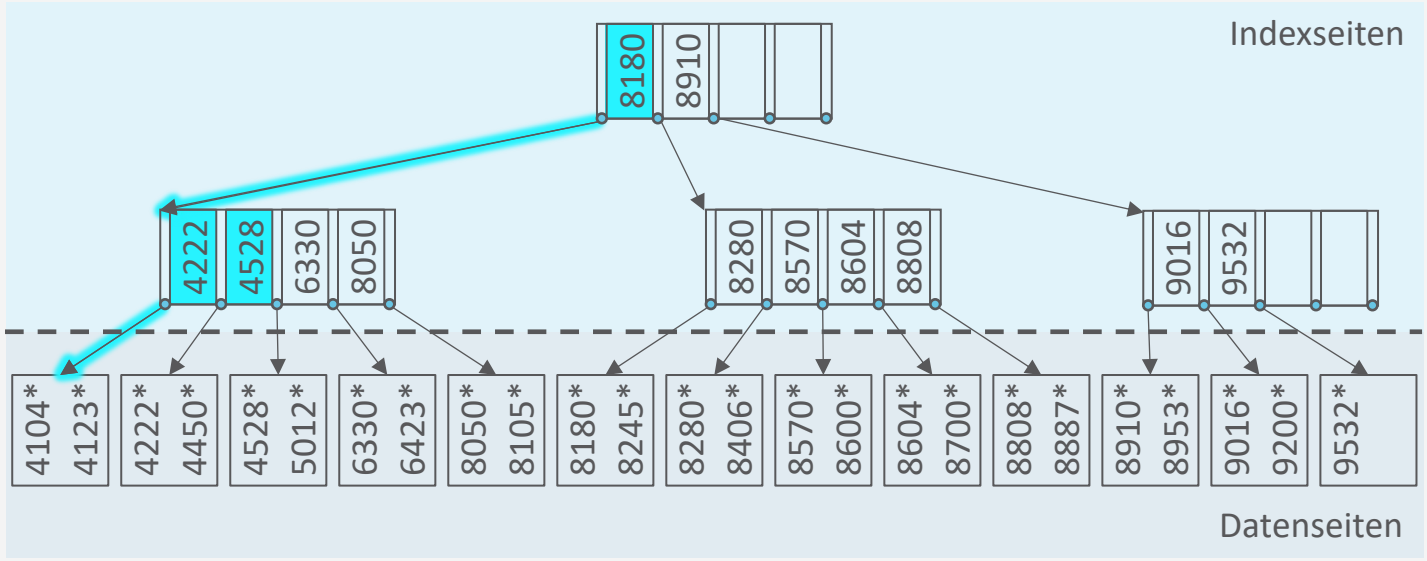
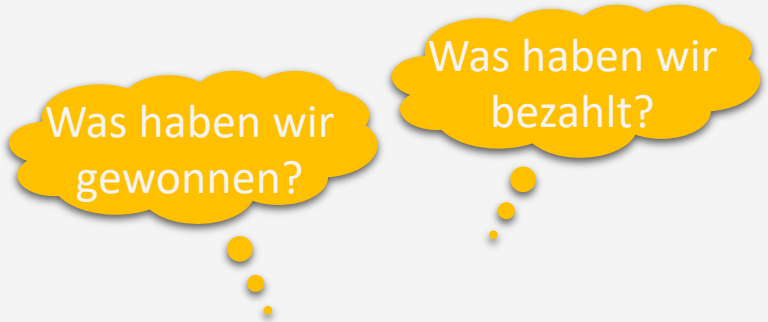
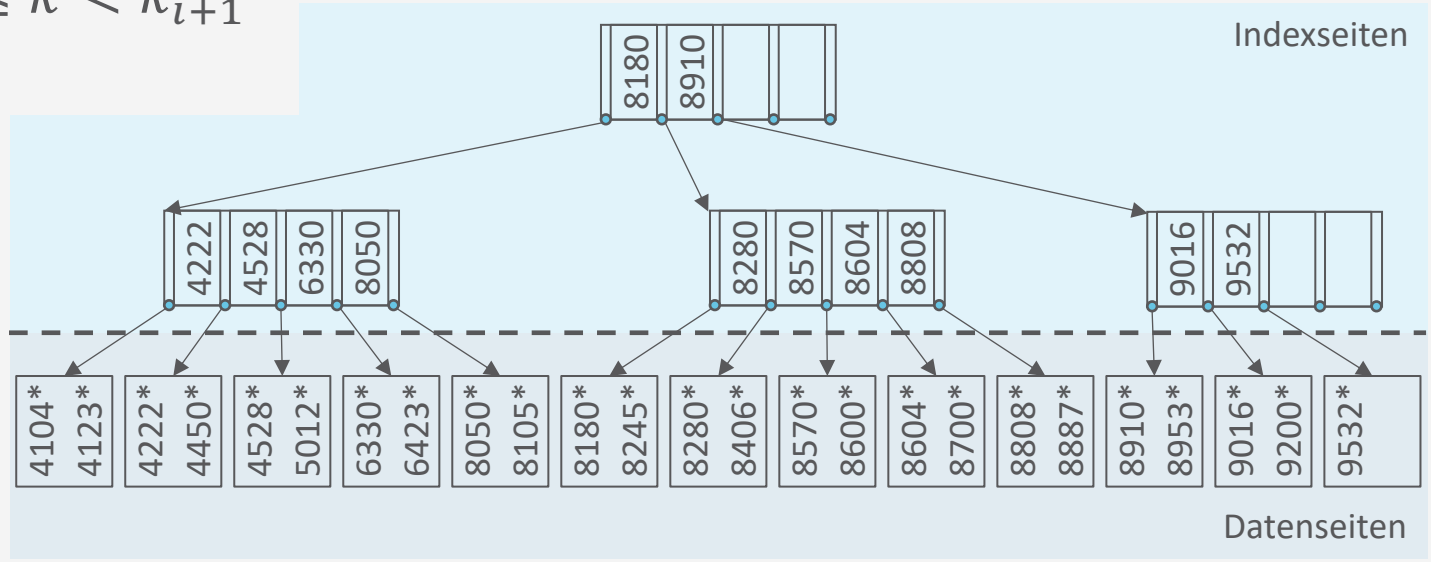
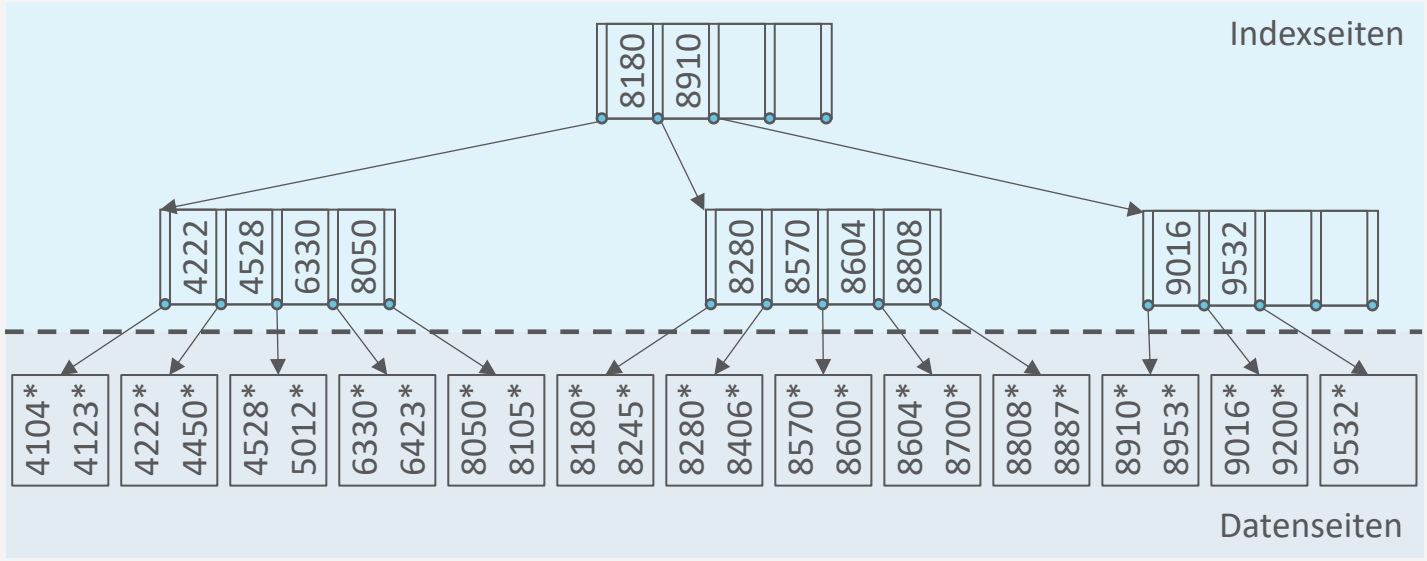
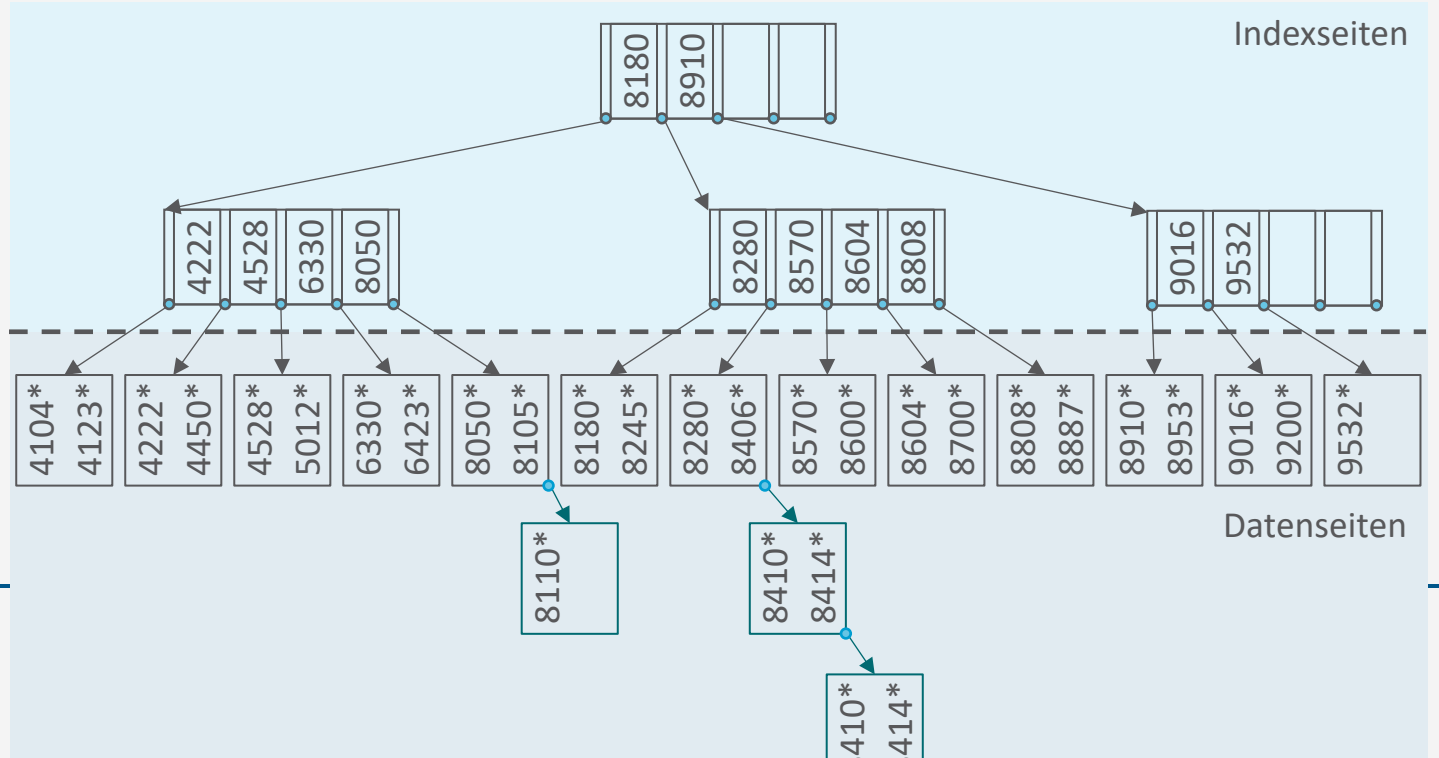
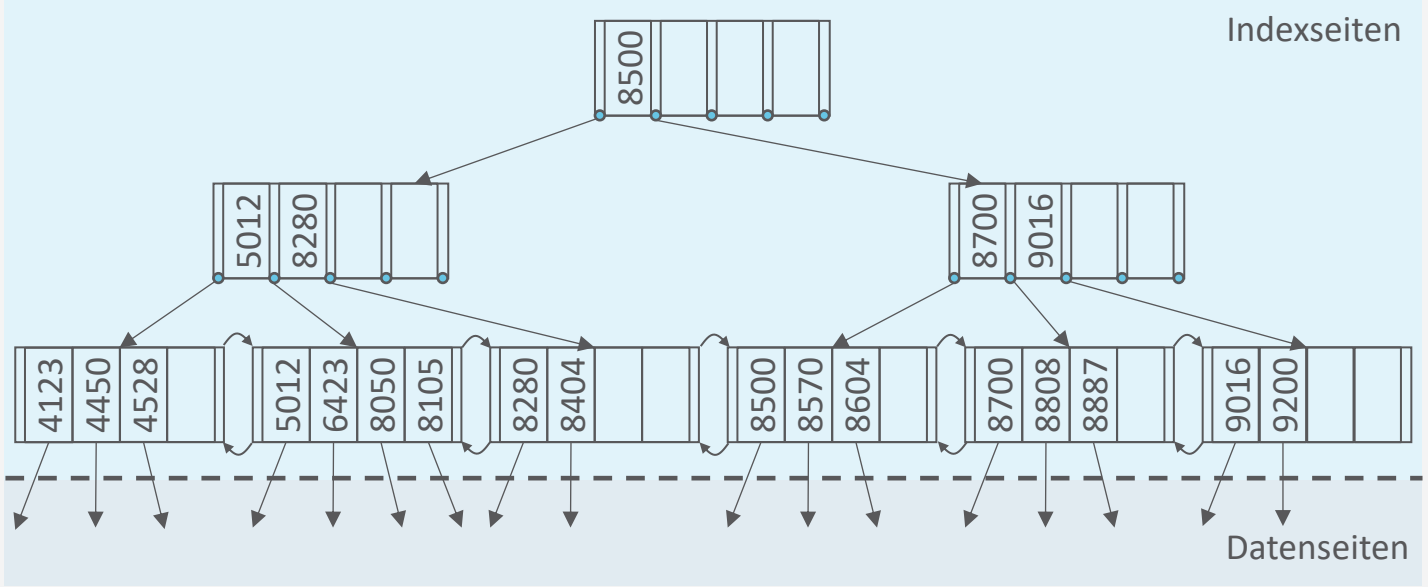
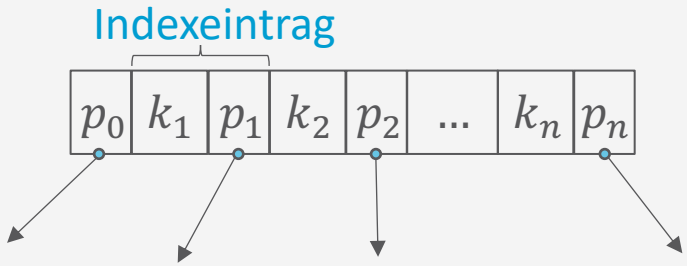
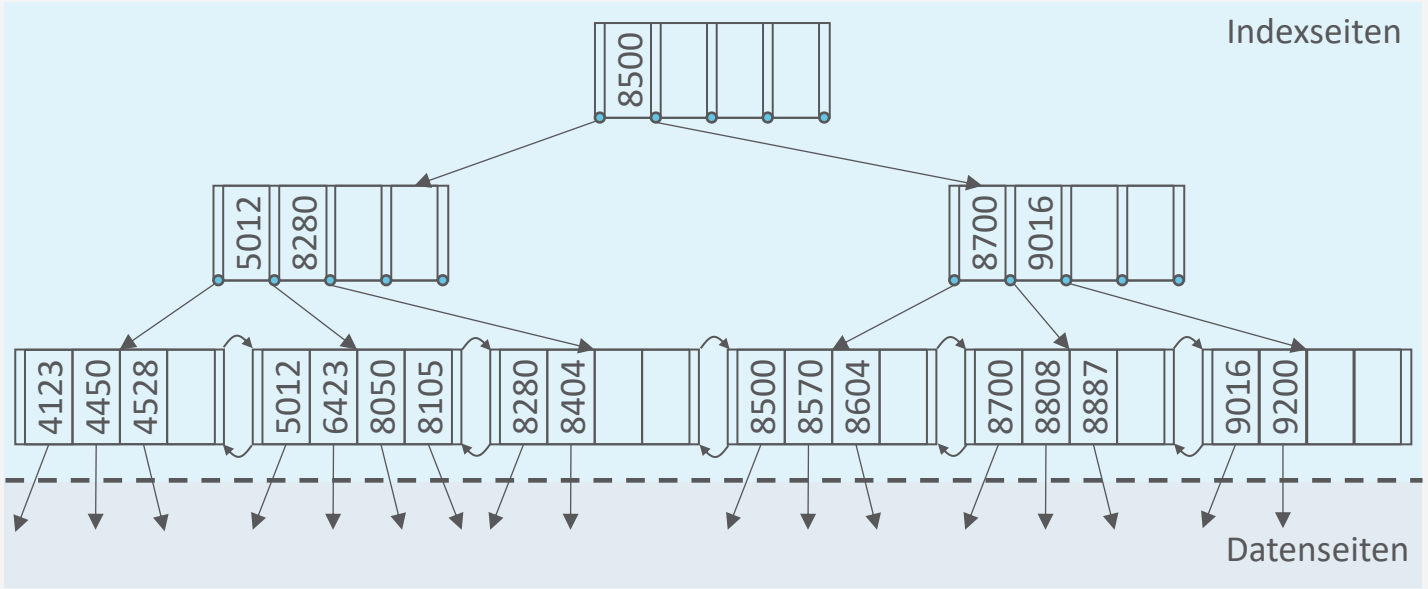
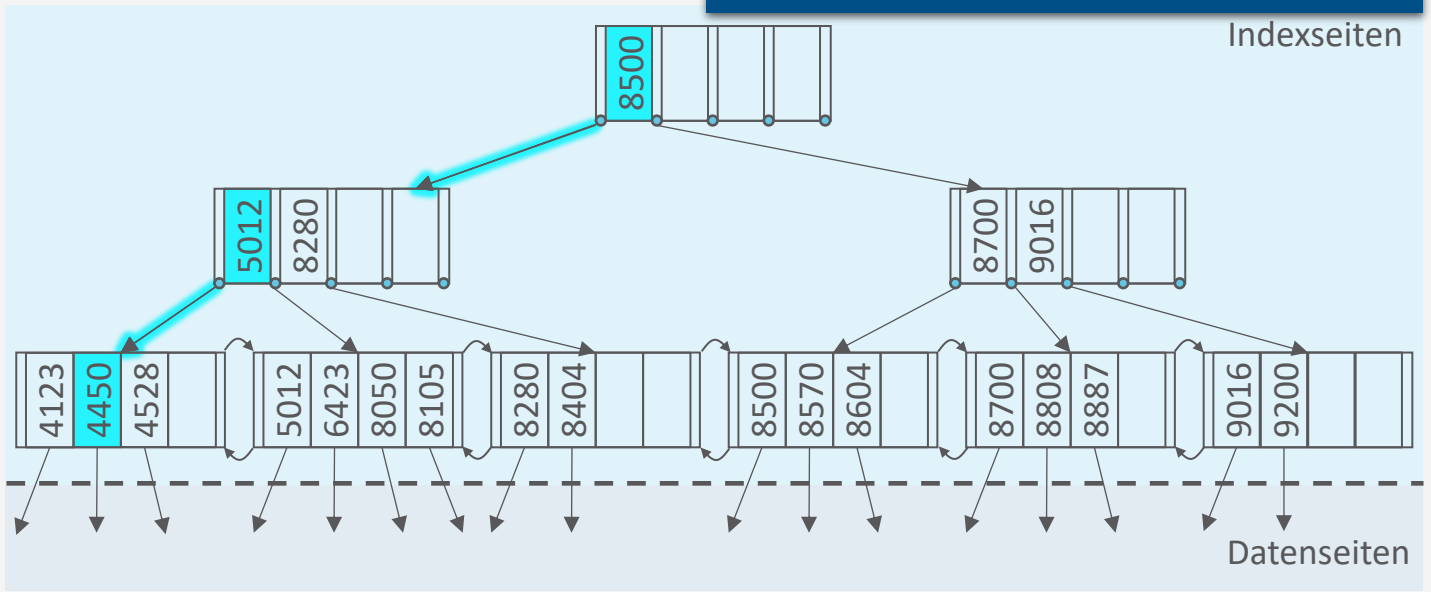
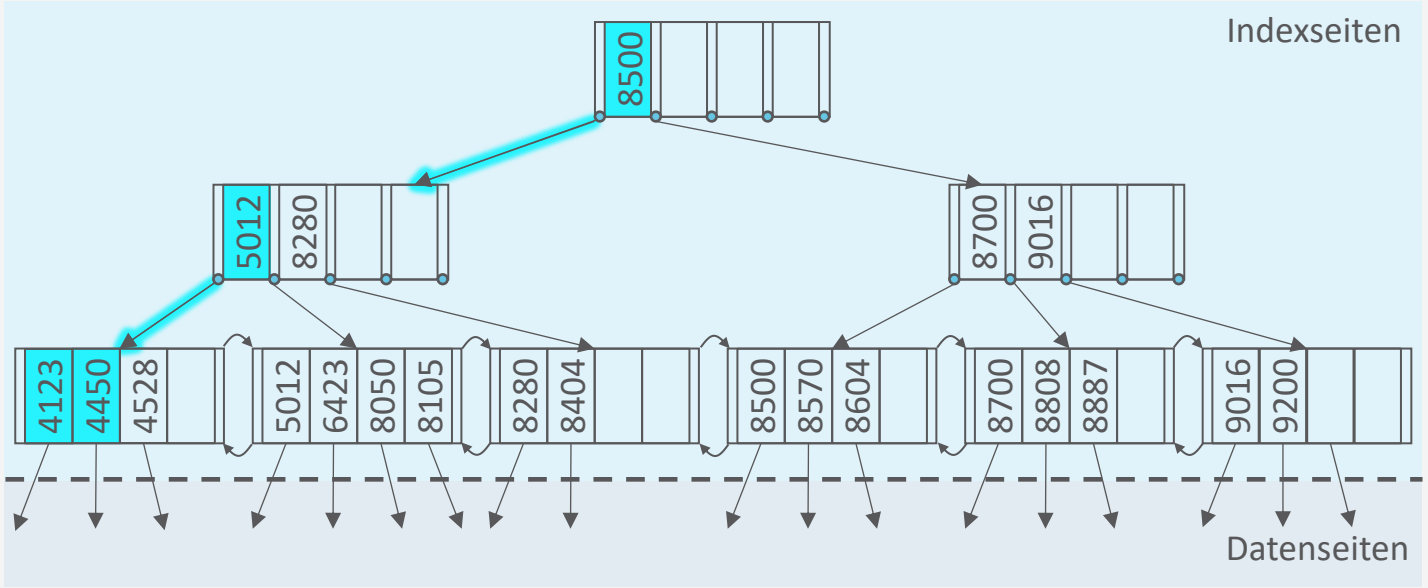
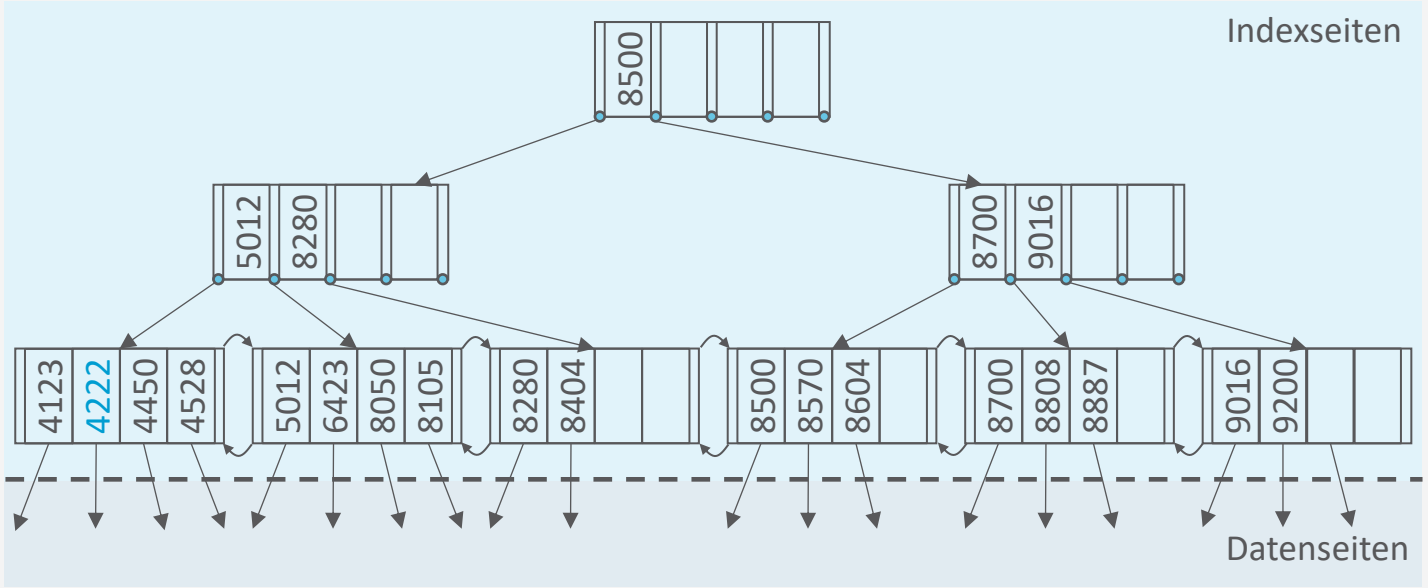
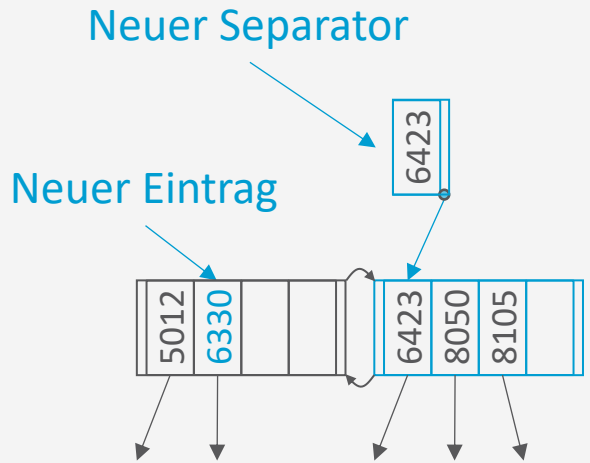
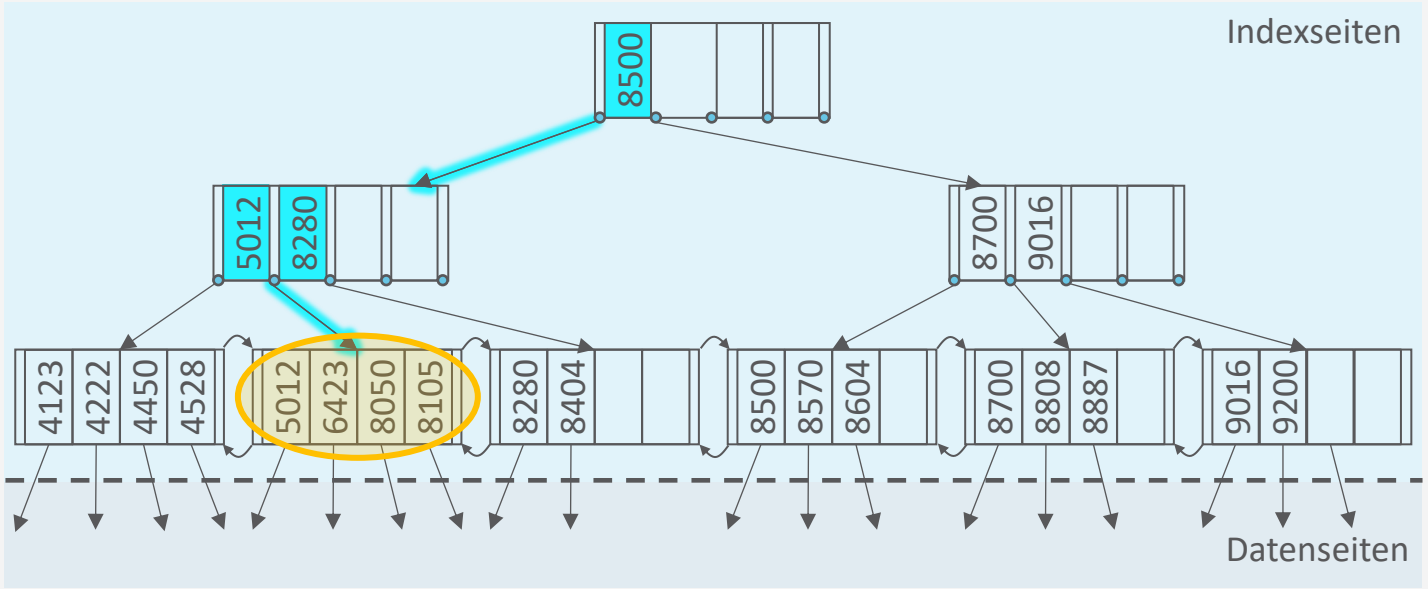
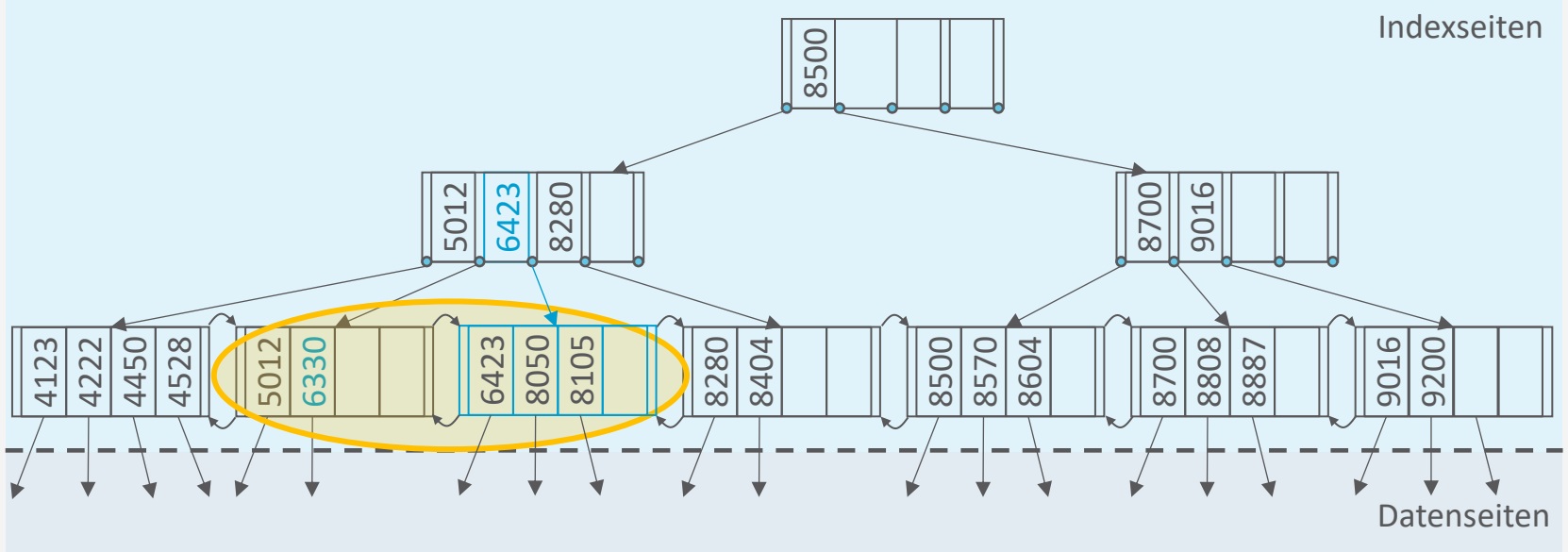
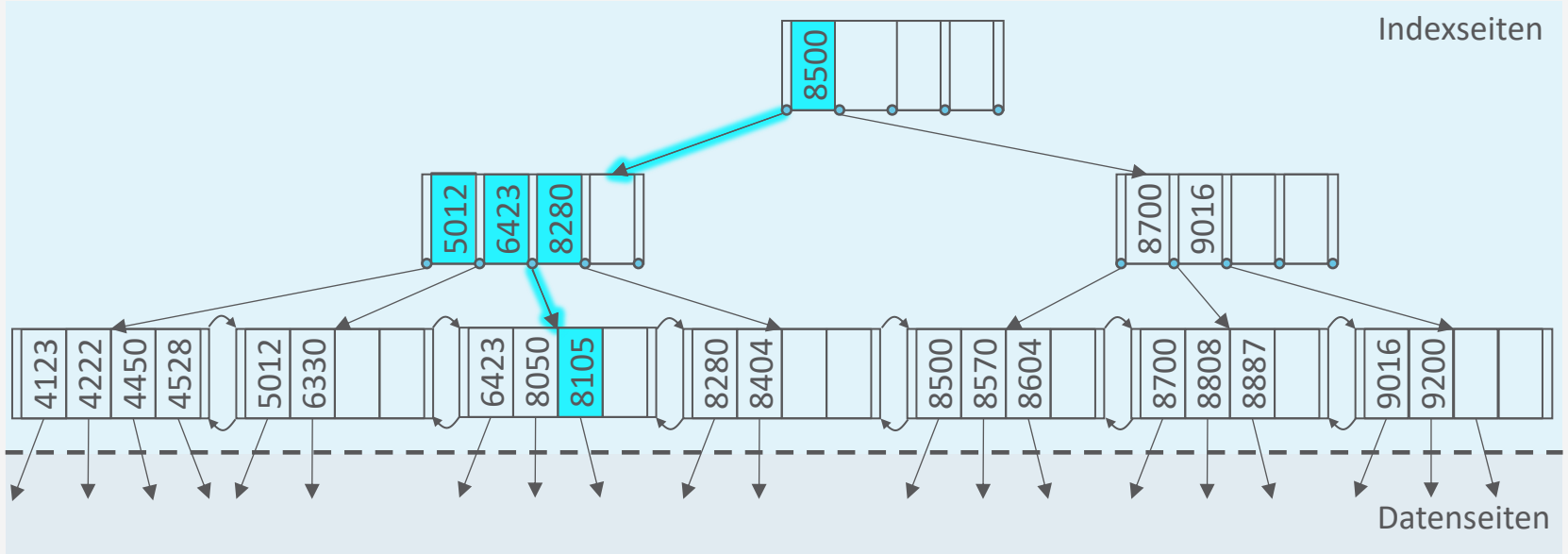
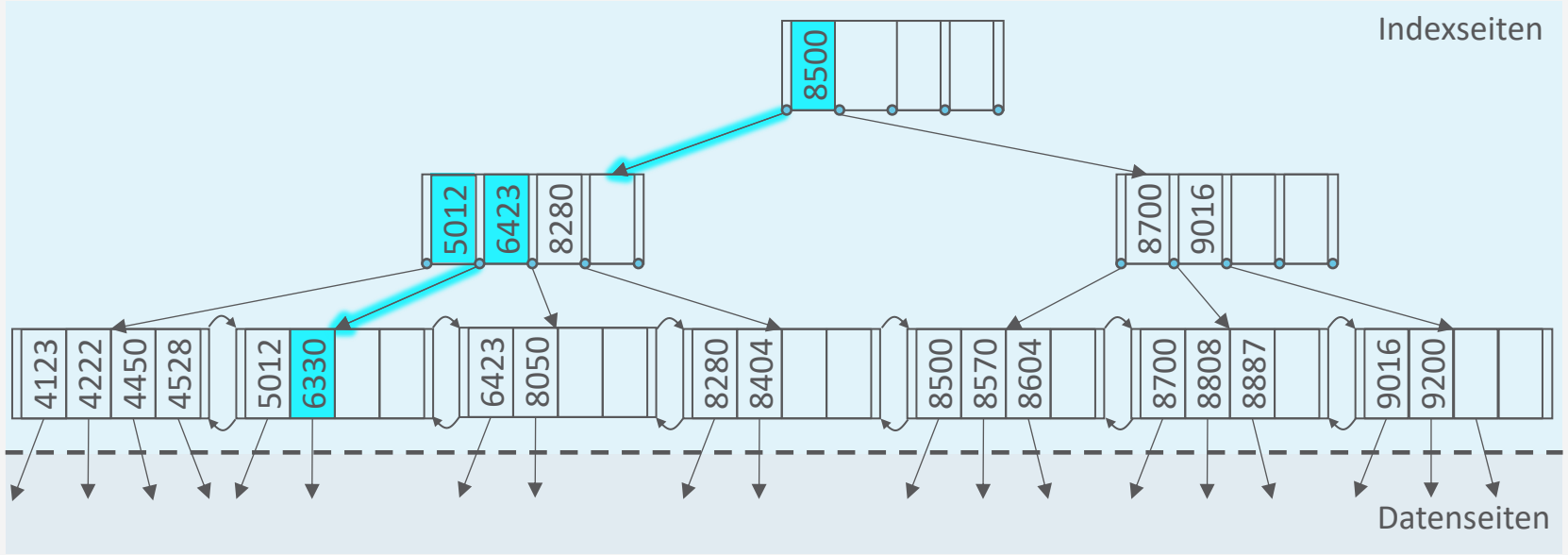
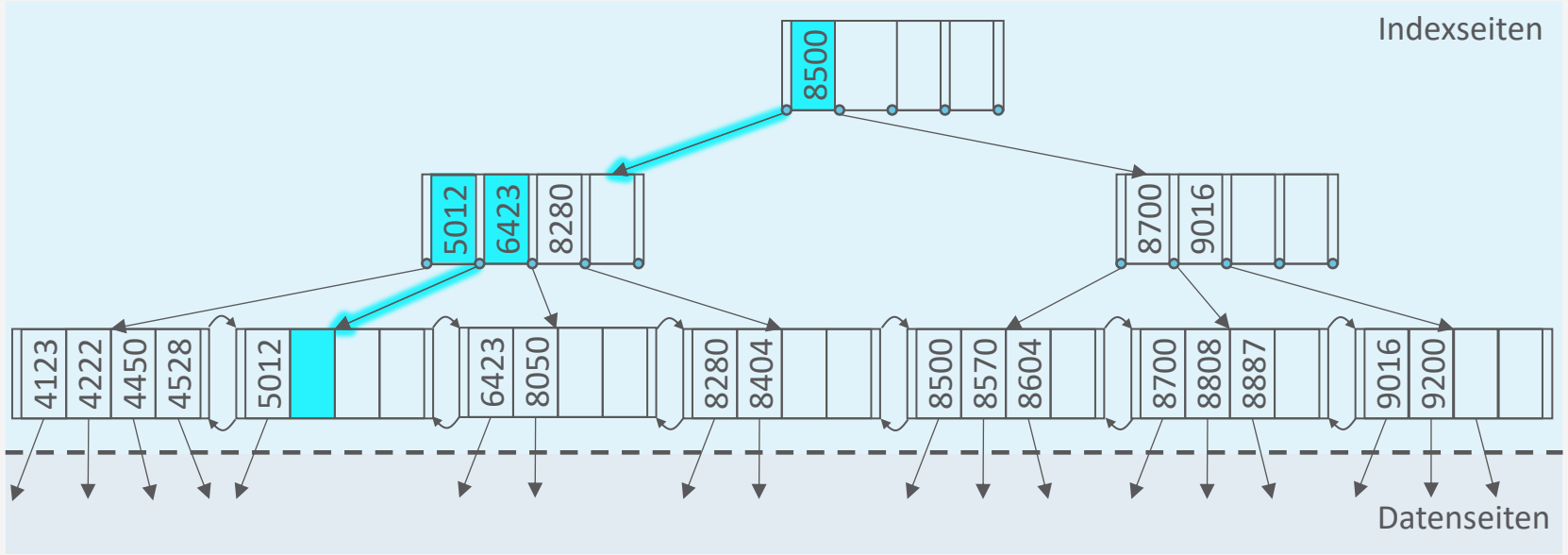
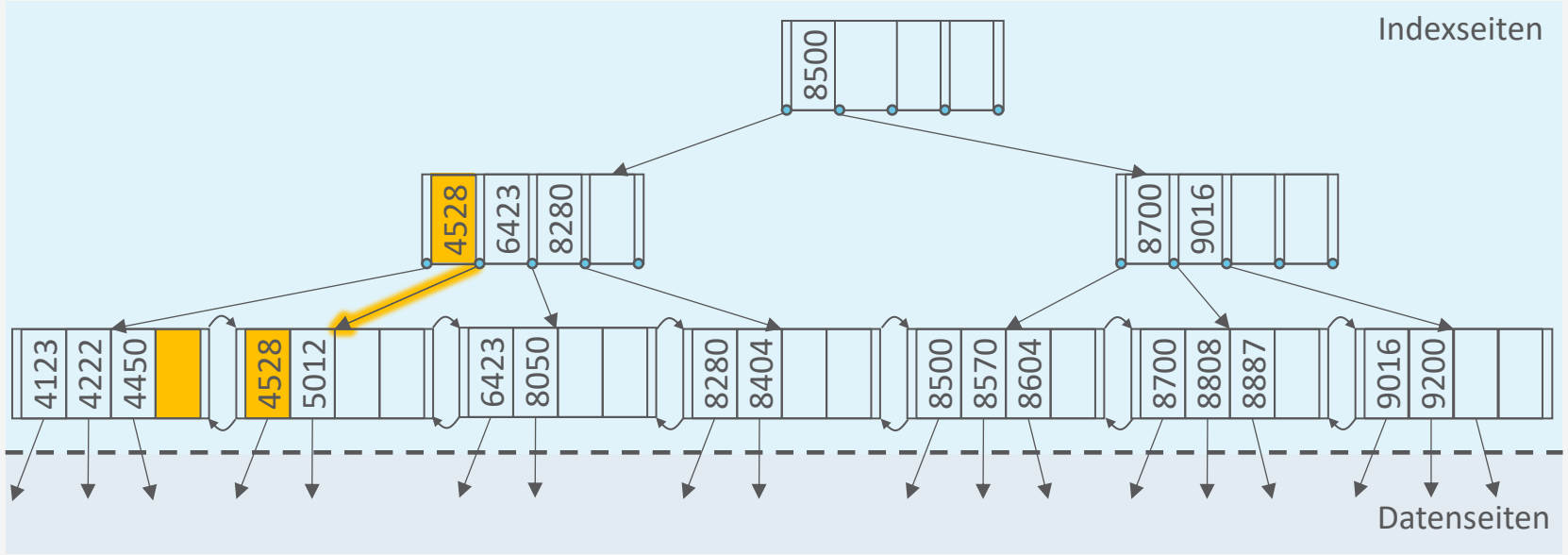
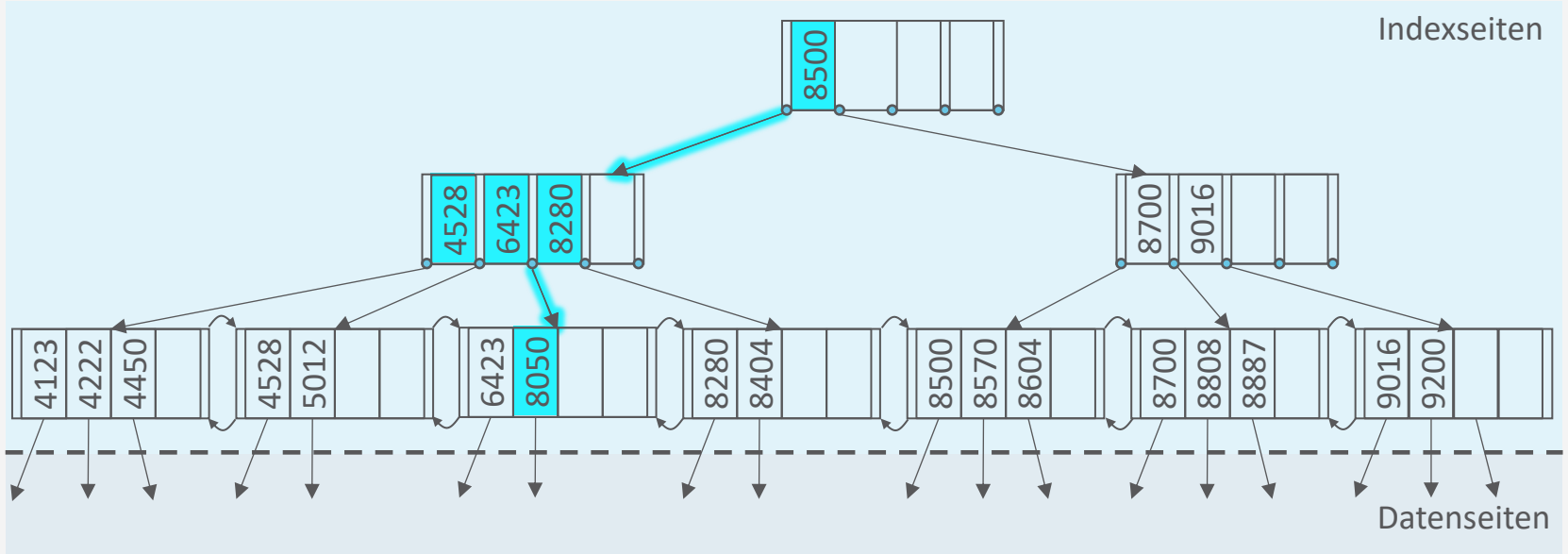
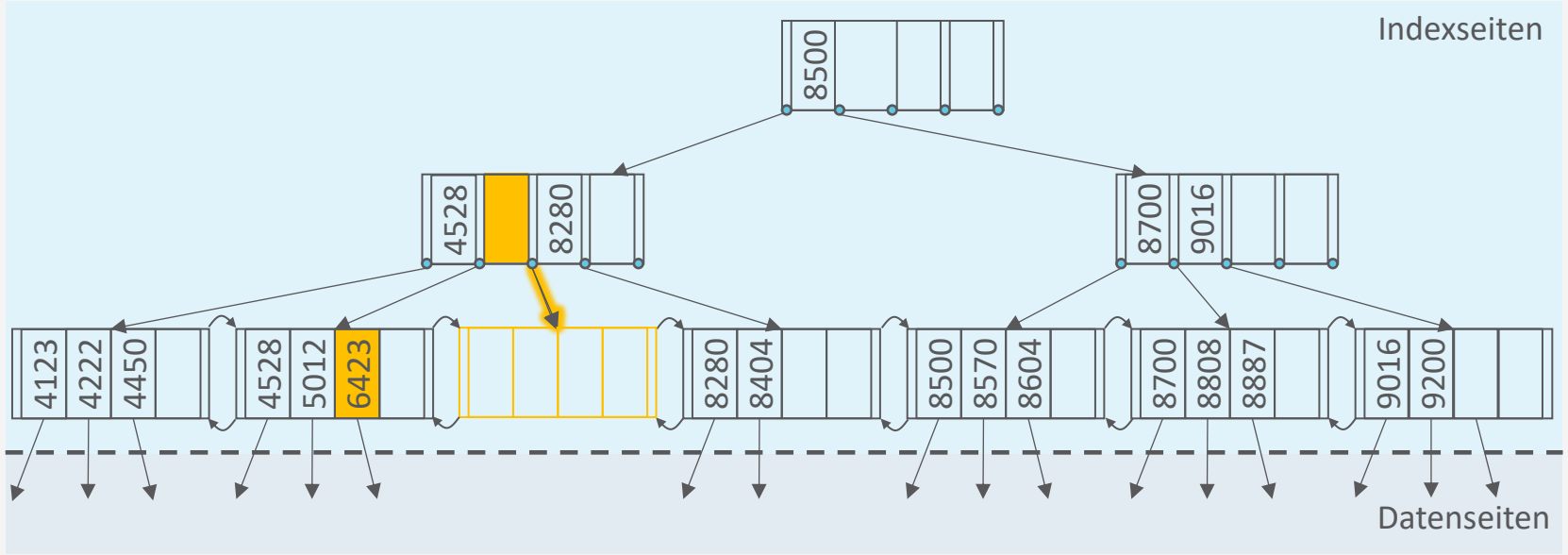
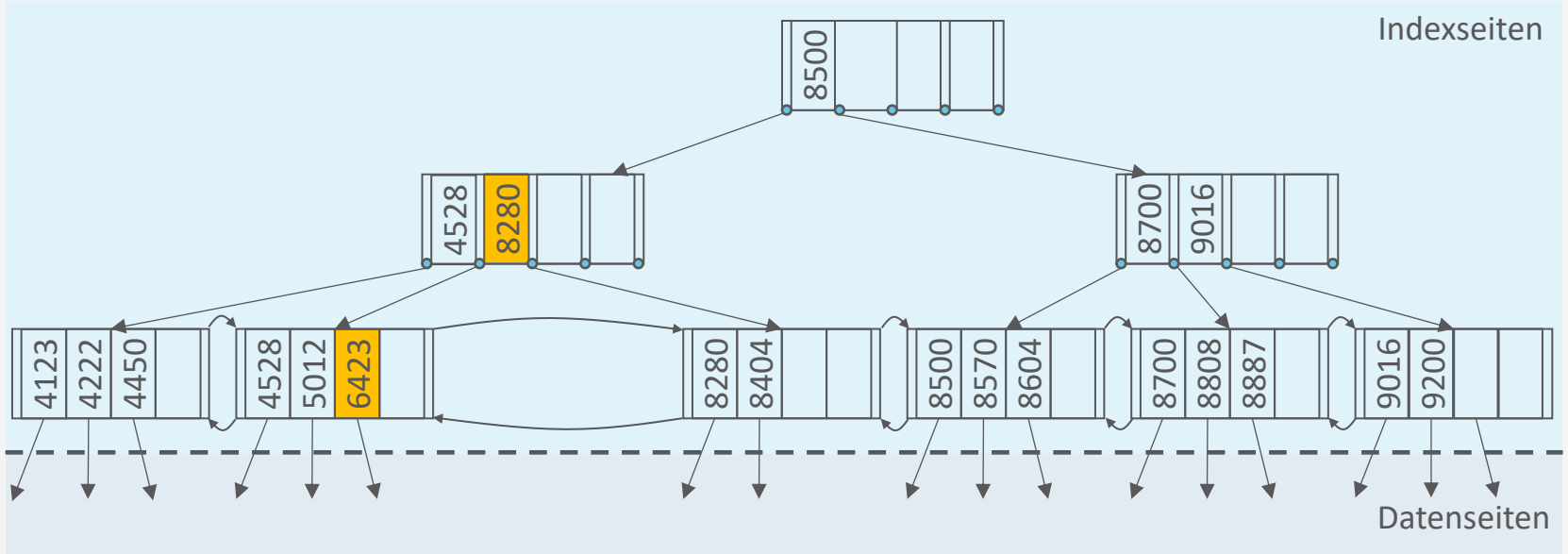
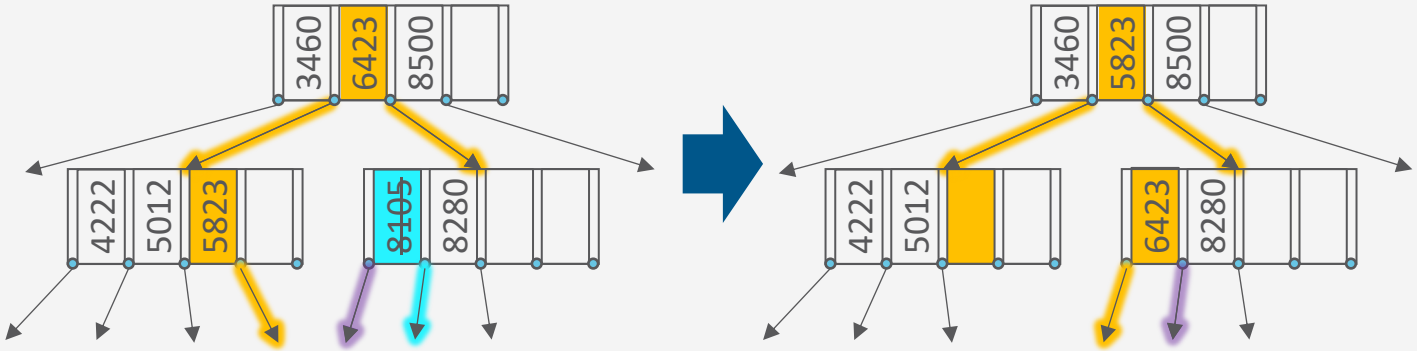
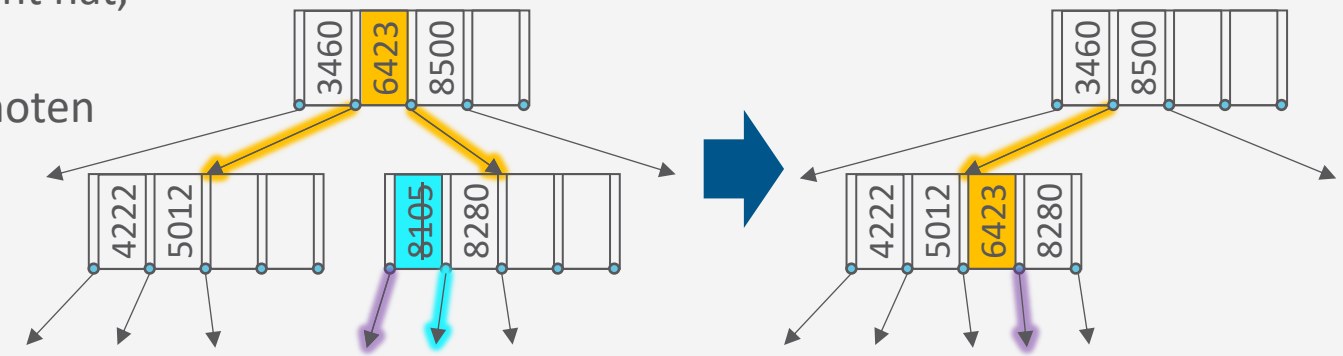




Index-basierte Lösung

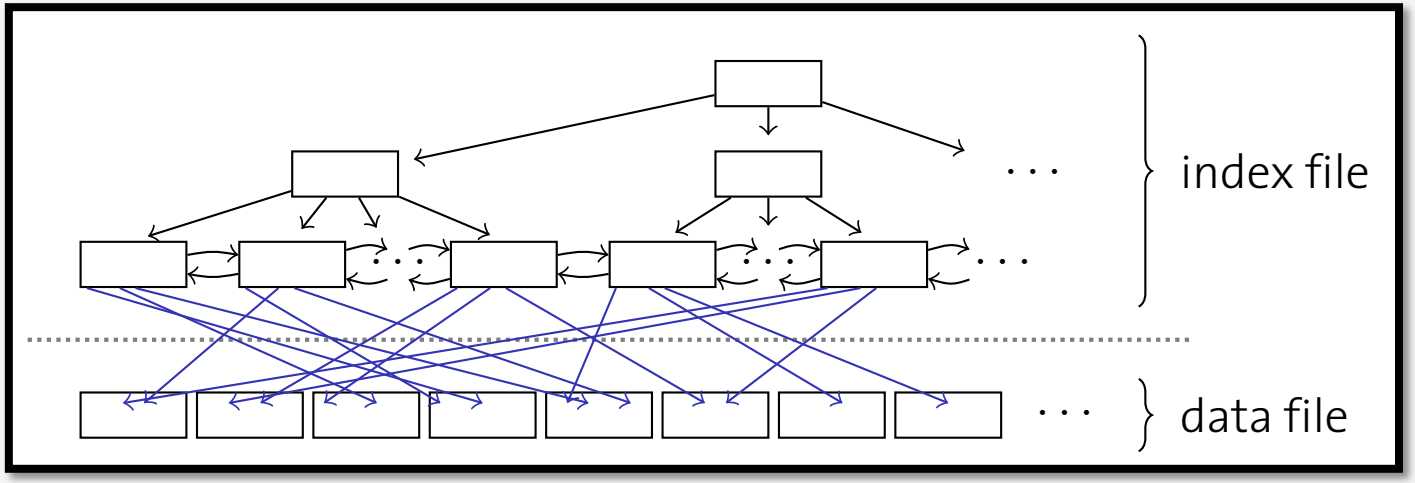
- Zurück zum Beispiel
  - ```
    select *
    from Kunden
    where Plz between 8800 and 9099;
    ```
  - Auswertung:
    1. Suche Startpunkt in Index, an dem PLZ ≥ 8800
    2. Finde Startpunkt auf Datenseite; scanne Datensätze, bis Plz > 9099

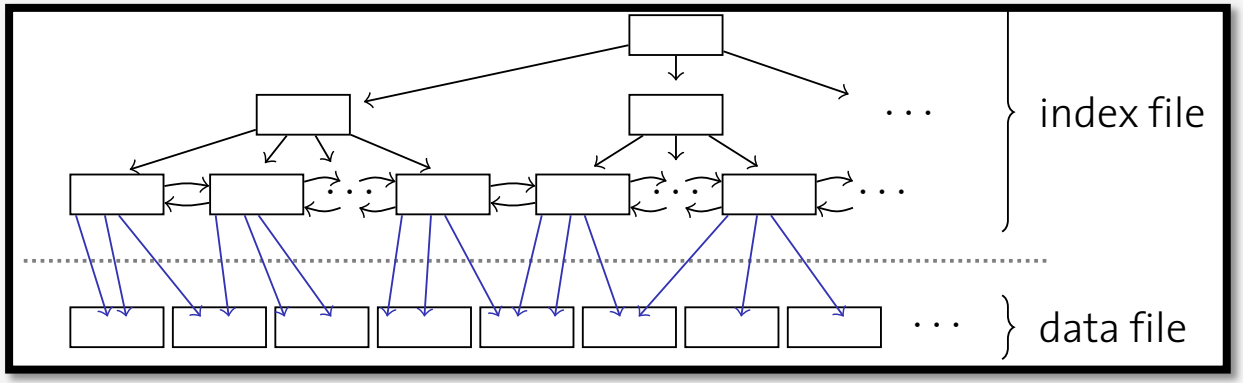
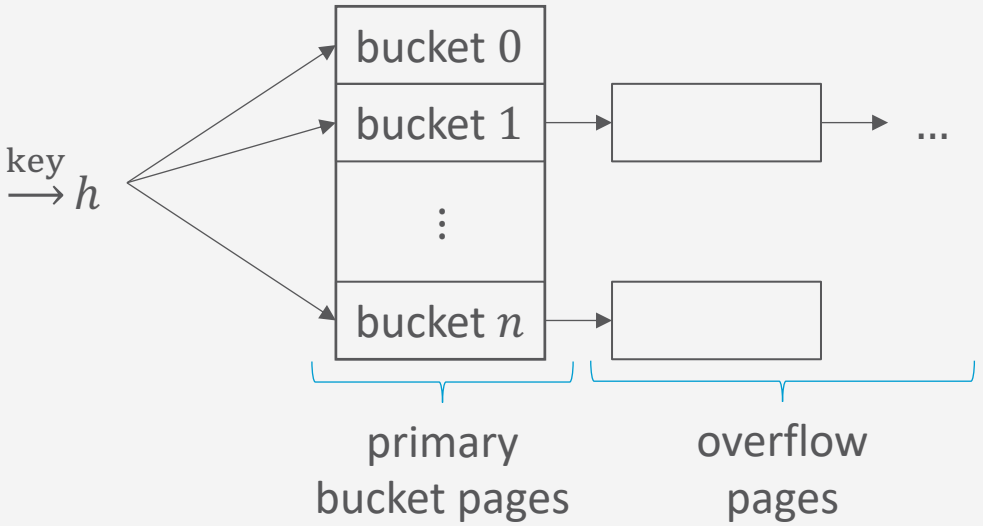
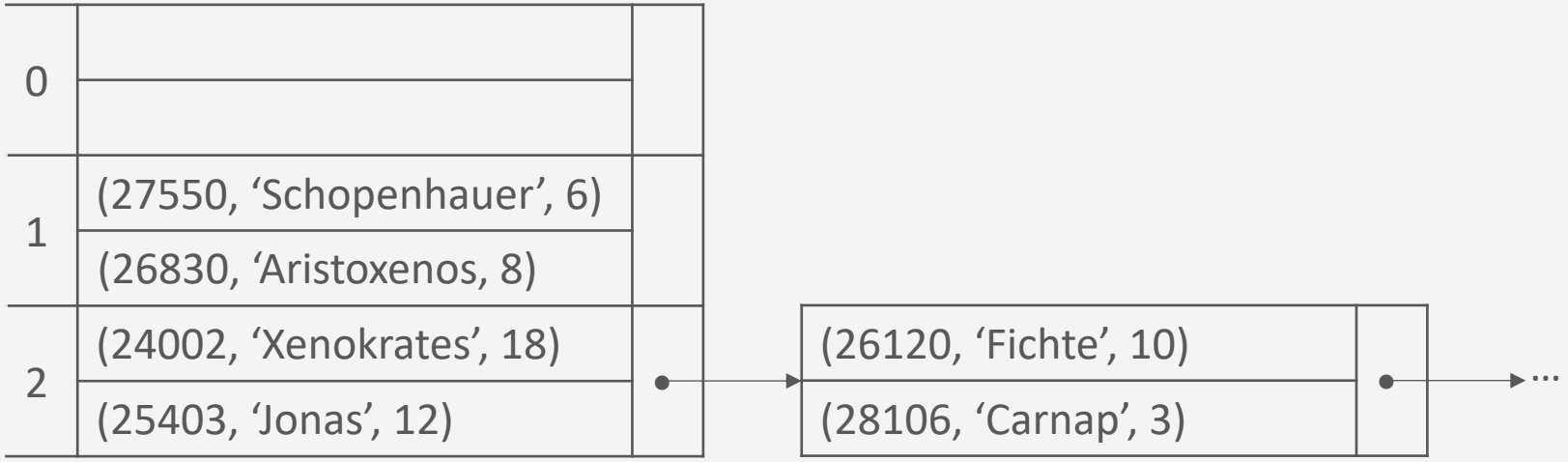
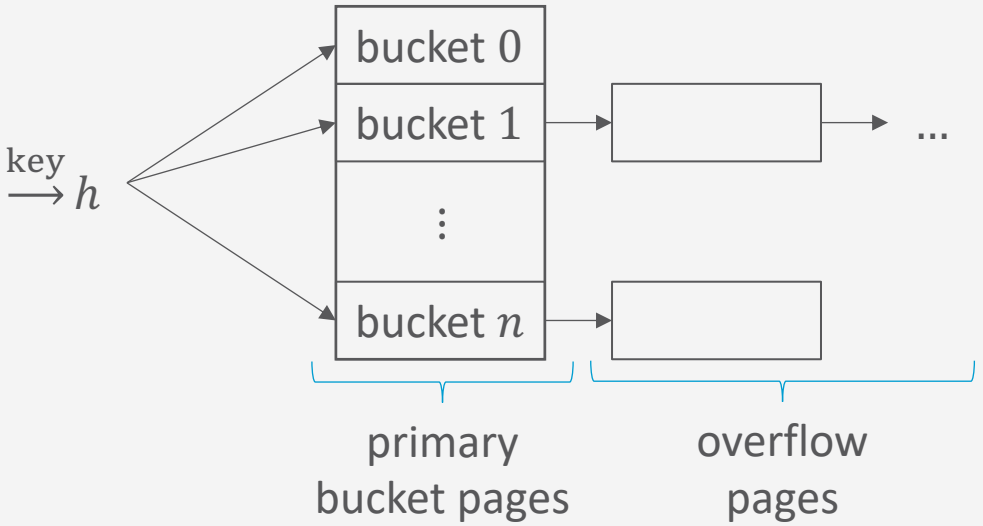
Indexed Sequential Access Method (ISAM)
- Von IBM Ende der 1960er Jahre entwickelt


    ```

Indexed Sequential Access Method (ISAM)

- Datenseiten im Speicher, sortiert gemäß Suchschlüssel (wie für die Binärsuche)
- Indexseiten mit Schlüsselwerten im Baum
  - Hunderte Einträge pro Seite
    - Hohe Verzweigung
    - Kleine Tiefe
  - Felder fester Länge
    - Navigation mittels Versatz
    - Binärsuche möglich



ISAM: Indexeinträge

- Indexeintrag $\langle k, p \rangle$
  - Suchschlüssel k
  - Separator p
    - Referenz auf Index- oder Datenseite
    - Schlüsselwerte in Seite referenziert durch p_i alle $\geq k_i$ und $< k_{i+1}$





ISAM: Anfragetypen

- Anfragetypen
  - Bereichsanfragen
    - Beispiel: `Plz between 8800 and 9099`
  - Punktanfragen
    - Beispiel: `Plz=4123`
- Indexierung möglich, solange eine totale Ordnung definiert ist
  - Zahlenbasierte Datentypen
  - Lexikographisch: String, Char
    - $(A \prec B \prec \ldots)$
    - Manche Systeme erlauben keinen Index über sehr lange Strings





ISAM: Suche

- Suche von Einträgen mit Schlüsselwert k in Indexseite n
  - n ist eine Referenz auf eine Indexseite
    - Suche startet bei Wurzelknoten $root$
      - Implementierung: Funktion $\text{search}(k)$ mit Aufruf $\text{search}(k, root)$
  - Suche Separator p_i in n, so dass $k_i \leq k < k_{i+1}$
    - Wenn $k < k_1$, dann $p_i = p_0$
    - Wenn $k_n < k$, dann $p_i = p_n$
    - Mittels Binärsuche umsetzen
  - Wenn p_i Referenz auf Datenseite, suche nach k auf Datenseite (und bei Bereichsanfrage scanne ab da)
  - Sonst: Rekursiver Aufruf der Suche mit Indexseite n', auf die p_i verweist

function $\text{search}(k, n)$
  $p \leftarrow \text{binarySearch}^*(k, n)$
  if p refers to data page **then**
    return $\text{binarySearch}(k, p)$
  $\text{search}(k, p)$ ▷ p index page



binarySearch^* setzt eine Binärsuche nach k um, wie in binarySearch, berücksichtigt aber die Separatoren bei der Berechnung der Positionen und gibt den passenden Separator zurück

ISAM: Diskussion

- Vorteile
  - Nicht für jeden Zugriff während der Suche eine eigene Seite laden
  - Binärsuche innerhalb einer Indexseite
  - Einstiegspunkt nahe Startpunkt gefunden, ab dem sequentiell gelesen werden kann
- Nachteil
  - Zusätzlicher Speicher für Indexseiten
    - Analyse von Anfragen zum Finden häufiger Suchschlüssel für Indices



ISAM: Aktualisierungsoperationen

- Löschen: Datensatz über Index suchen und anschließend aus Datenseite entfernen
- Einfügen: Auf passende Datenseite navigieren und Datensatz einfügen
  - Problem, wenn Seite voll
    - Im Vorhinein Platz lassen
    - Überlauf-Seite einfügen
- Vorteil: Index statisch, i.e., bleibt gültig
  - Kein Aufwand bei Änderungen
- Nachteil: Index degeneriert
  - Sequentielle Ordnung durch Überlauf-Seiten zerstört



ISAM: Aktualisierungsoperationen

- Freiraum bei der Indexerzeugung vorsehen
  - Reduziert das Einfügeproblem
  - Typisch sind 20% Freiraum
- Da Indexseiten statisch, keine Zugriffskoordination (bei Mehrbenutzerbetrieb) nötig
  - Zugriffskoordination (Sperren) würde gleichzeitigen Zugriff (besonders nahe der Wurzel) für andere Anfragen vermindern
- ISAM ist nützlich für (relativ) statische Daten

B⁺-Bäume

Indexierung



B⁺-Bäume: Eine dynamische Indexstruktur

- B⁺-Bäume vom ISAM-Index abgeleitet, sind aber dynamisch
  - Keine Überlauf-Ketten
  - Balancierung wird aufrechterhalten
  - Behandelt insert und delete angemessen
  - Indexseiten nicht statisch
- Minimale Besetzungsregel für B⁺-Baum-Knoten (außer der Wurzel): 50% (typisch sind 67%, wird dann manchmal auch B*-Baum genannt)
  - Verzweigung nicht zu klein

  vs.
  - Indexknotensuche nicht zu linear

R. Bayer and E. M. McCreight, Organization and Maintenance of Large Ordered Indexes, Acta Informatica, vol. 1, no. 3, **1972**

B⁺-Bäume: Grundlagen

- B⁺-Bäume ähnlich zu ISAM-Index, wobei
  - Referenzierte Datenseiten i.d.R. nicht in sequentieller Ordnung
  - Index-Blätter zu doppelt verketteter Liste verbunden
    - Schlüssel dort noch einmal wiederholt, sortiert
- Jeder Knoten enthält zwischen d und $2d$ Einträge
  - d heißt Ordnung des Baumes, Wurzel ist Ausnahme
- Es gilt weiterhin für die Indexknoten (Nicht-Blätter): Eintrag $\langle k, p \rangle$



Warum?



B⁺-Bäume: Was wird in den Blättern gespeichert?

- Drei Alternativen
  1. Vollständiger Datensatz k^*:
     - Blatt ist Datenseite (wie bei ISAM Index)
  2. Ein Paar $\langle k, rid \rangle$, wobei rid (record ID) Zeiger auf Datensatz:
     - Blatt enthält Liste von $\langle k, rid \rangle$ Paaren, sortiert nach k
     - Suchschlüssel k kann auf Blatt häufiger auftreten (mehrere rid‘s mit Suchschlüssel k)
     - Bei $rid = \langle pageno, slotno \rangle$ lässt sich der genaue Speicherort des Datensatzes bestimmen
  3. Ein Paar $\langle k, \{rid_1, rid_2, \dots\} \rangle$, wobei alle rid‘s den Suchschlüssel k haben
     - Suchschlüssel k tritt nur einmal auf (weniger Redundanz)
     - Aber: kein regelmäßiger Abstand von einem $\langle k, \{rid_1, rid_2, \dots\} \rangle$ zum nächsten $\langle k', \{rid'_1, rid'_2, \dots\} \rangle$
- Varianten 2. und 3. bedingen, dass rid‘s stabil sein müssen, also nicht (einfach) verschoben werden können

Alternative 2 scheint am meisten verwendet zu werden. Wir nehmen im Folgenden Variante 2 an.

B⁺-Bäume: Suche

- Suche von Einträgen mit Schlüssel k in B⁺-Baum wie beim ISAM-Index, wobei aber die Abbruchbedingung nicht auf Datenseite prüft, sondern auf Index-Blattseite:
  - Suche in Index nach k bis zum Index-Blattknoten e
  - Suche in e nach k
  - Folge der Referenz
- Beispiel:
  - Punktanfrage: Schlüssel 4450
  - Bereichsanfrage: Schlüssel zwischen 4450 und 6500

function search(k, n)
  $p \leftarrow \text{binarySearch}^*(k, n)$
  if p refers to index leaf **then**
    return binarySearch(k, p)
  search(k, p)



binarySearch* setzt eine Binärsuche nach k um, wie in binarySearch, berücksichtigt aber die Separatoren bei der Berechnung der Positionen und gibt den passenden Separator zurück

B⁺-Bäume: Einfügen

- B⁺-Baum soll nach Einfügung balanciert bleiben, i.e., keine Überlauf-Seiten
- Überblick über Vorgehen zum Einfügen mit Eingaben k und rid ($\text{insert}(k, rid)$):
  1. Suche Blattseite n, in der Eintrag für k sein kann
  2. Falls n genug Platz hat (höchstens $2d - 1$ Einträge): Füge Eintrag $\langle k, rid \rangle$ in n ein
     - Suche nach passender Position mittels Binärsuche
  3. Sonst: Teile n auf in n und n' inklusive $\langle k, rid \rangle$ und füge neuen Eintrag $\langle k', n' \rangle$ in Elternknoten e von n ein, wobei n' als Separator für die Referenz auf n' steht und k' der kleinste Schlüssel in n' ist
     - Wenn e keinen Platz mehr hat ($2d$ Einträge), muss e ebenfalls aufgeteilt werden

     → Aufspaltung kann sich rekursiv nach oben fortsetzen, eventuell bis zur Wurzel
     - Wenn die Wurzel aufgeteilt wird, wird ein neuer Wurzelknoten als Elternknoten eingeführt (Baum erhöht sich)
     - Wurzel kann unter 50% gefüllt sein

B⁺-Bäume: Einfügen – Beispiel ohne Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 4222
  1. Suche führt zu erstem Blattknoten in der verketteten Liste ($3 < 2d - 1, d = 2$)
  2. Blattknoten hat genug Platz ($3 < 2d - 1, d = 2$), von daher einfach einfügen
     - Erhalte Sortierung innerhalb der Knoten



B⁺-Bäume: Einfügen – Beispiel ohne Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 4222
  1. Suche führt zu erstem Blattknoten in der verketteten Liste ($3 < 2d - 1, d = 2$)
  2. Blattknoten hat genug Platz ($3 < 2d - 1, d = 2$), von daher einfach einfügen
     - Erhalte Sortierung innerhalb der Knoten



B⁺-Bäume: Einfügen – Beispiel mit Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 6330
  1. Suche führt zu zweitem Blattknoten in verketteter Liste
  2. Blattknoten hat nicht mehr genug Platz
  3. Index anpassen
     - Erste Hälfte (d Einträge) bleibt in Knoten
     - Zweite Hälfte ($d + 1$ Einträg) geht in neuen Knoten
     - Neuer Eintrag in Elternknoten
       - Kleinster Eintrag im neuen Knoten: 6423
       - Platz in Elternknoten, daher direkt einfügen





B⁺-Bäume: Einfügen – Beispiel mit Aufspaltung

- Einfügen eines Eintrags mit Schlüssel 6330
  1. Suche führt zu zweitem Blattknoten in verketteter Liste
  2. Blattknoten hat nicht mehr genug Platz
  3. Index anpassen
     - Angepasster Index:



B⁺-Bäume: Insert

- $\text{insert}(k, rid)$ wird von außen aufgerufen
- Blattknoten enthalten rids, innere Knoten enthalten Zeiger auf andere B⁺-Baum-Knoten

function $\text{insert}(k, rid)$
    $\langle key, ptr \rangle \leftarrow \text{treeInsert}(k, rid, root)$ ▷ root contains the root of the index tree
    if key is not **null then**
        Allocate new root page r
        Populate r with $p_0 \leftarrow root, k_1 \leftarrow key, p_1 \leftarrow ptr$
        $root \leftarrow r$

B⁺-Bäume: Tree-Insert für Insert

- Einträge pro Seite maximal: $2d = n$

function treeInsert$(k, rid, node)$
  if $node$ is leaf **then**
    return leafInsert$(k, rid, node)$
  if $k < k_0$ **then**
    $i \leftarrow 0$
  else if $k_n < k$ **then**
    $i \leftarrow n$
  else
    Find i such that $k_i \leq k < k_{i+1}$ ▷ Use binary search
  $\langle sep, ptr \rangle \leftarrow$ treeInsert(k, rid, p_i) ▷ p_i contains the reference to next node
  if sep is **null then**
    return $\langle null, null \rangle$
  else
    return split$(sep, ptr, node)$

B⁺-Bäume: Leaf-Insert für Insert

- Einträge pro Seite maximal: $2d$
- Einträge auf Seite: $\{\langle k_1, rid_1\rangle, \ldots, \langle k_{2d}, rid_{2d}\rangle\}$

function leafInsert$(k, rid, leaf)$
  if another entry fits into $leaf$ **then**
    Insert $\langle k, rid\rangle$ into $leaf$
    return $\langle null, null\rangle$
  Allocate new leaf node l
  Let $\{\langle k'_1, rid'_1\rangle, \ldots, \langle k'_{2d+1}, rid'_{2d+1}\rangle\} \leftarrow \{\langle k_1, rid_1\rangle, \ldots, \langle k_{2d}, rid_{2d}\rangle\} \cup \{\langle k, rid\rangle\}$
  Store entries $\langle k'_1, rid'_1\rangle, \ldots, \langle k'_d, rid'_d\rangle$ in $leaf$
  Store entries $\langle k'_{d+1}, rid'_{d+1}\rangle, \ldots, \langle k'_{2d+1}, rid'_{2d+1}\rangle$ in l
  return $\langle k'_{d+1}, l\rangle$

B⁺-Bäume: Split für Insert

- Einträge pro Seite maximal: $2d$
- Einträge auf Seite: $\{\langle k_1, p_1\rangle, \ldots, \langle k_{2d}, p_{2d}\rangle\}$, dazu noch p_0

function split$(k, ptr, node)$
  if another entry fits into $node$ **then**
    Insert $\langle k, ptr\rangle$ into $node$
    return $\langle null, null\rangle$
  Allocate new node p
  Let $\{\langle k'_1, p'_1\rangle, \ldots, \langle k'_{2d+1}, p'_{2d+1}\rangle\} \leftarrow \{\langle k_1, p_1\rangle, \ldots, \langle k_{2d}, p_{2d}\rangle\} \cup \{\langle k, ptr\rangle\}$
  Store entries $\langle k'_1, p'_1\rangle, \ldots, \langle k'_d, p'_d\rangle$ in $node$, keeping p_0
  Store entries $\langle k'_{d+2}, p'_{d+2}\rangle, \ldots, \langle k'_{2d+1}, p'_{2d+1}\rangle$ in p
  $p_0 \leftarrow p'_{d+1}$ in p
  return $\langle k'_{d+1}, p\rangle$

B⁺-Baum: Löschen

- B⁺-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d + 1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d - 1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105



B⁺-Baum: Löschen

- B⁺-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d + 1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d - 1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 6330



B⁺-Baum: Löschen

- B⁺-Baum soll nach Löschung balanciert bleiben
  - Wenn Löschen nicht für einen Unterlauf sorgt (min. $d + 1$ Einträge), einfach löschen
    - Innere Knoten können dann alte Schlüssel enthalten → ist ok, da die Ordnung erhalten bleibt
  - Sonst, d.h., wenn nach löschen nur $d - 1$ Einträge übrig bleiben: Index wieder balancieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8105
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 6330

Was nun?



B⁺-Baum: Löschen – Unterlauf

- Vorgehen, wenn eines der Nachbarblätter genug Einträge ($> d$) hat
  - Blatt mit Eintrag aus Nachbarknoten auffüllen und Indexeintrag in Vorfahrknoten aktualisieren
    - Wenn Nachbarblatt gleichen Elternknoten wie unterlaufener Knoten hat, dann Elternknoten anpassen, sonst Großelternknoten anpassen
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 6330 führt zu Unterlauf, welcher mit Nachbar-Eintrag aufgefüllt werden kann



B⁺-Baum: Löschen – Unterlauf

- Vorgehen, wenn eines der Nachbarblätter genug Einträge ($> d$) hat
  - Blatt mit Eintrag aus Nachbarknoten auffüllen und Indexeintrag in Vorfahrknoten aktualisieren
    - Wenn Nachbarblatt gleichen Elternknoten wie unterlaufener Knoten hat, dann Elternknoten anpassen, sonst Großelternknoten anpassen
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 6330 führt zu Unterlauf, welcher mit Nachbar-Eintrag aufgefüllt werden kann
  - Anschließendes Löschen eines Eintrags mit Suchschlüssel 8050



Und nun?

B⁺-Baum: Löschen – Unterlauf

- Vorgehen, wenn kein Nachbarblatt genug Einträge hat (beide nur d Einträge)
  - Blattseite mit einem der Nachbarblätter verschmelzen und verweisenden Indexeintrag in Elternknoten löschen
    - Verschmelzen bei unterschiedlichen Elternknoten: Zusätzlich Großeltern-Indexeintrag aktualisieren
- Beispiel:
  - Löschen eines Eintrags mit Suchschlüssel 8050 führt zu Unterlauf
    - Einträge in vorheriges Blatt überführen
    - Dritte Blattseite und Indexeintrag $\langle 6423, leaf_3 \rangle$ löschen



B⁺-Baum: Löschen – Unterlauf

- Vorgehen, wenn kein Nachbarblatt genug Einträge hat (beide nur d Einträge)
  - Blattseite mit einem der Nachbarblätter verschmelzen und verweisenden Indexeintrag in Elternknoten löschen
    - Verschmelzen bei unterschiedlichen Elternknoten: Zusätzlich Großeltern-Indexeintrag aktualisieren
- Beispiel:
  - Ergebnis
- Verbleibende Aufgabe: Was machen, wenn durch Löschen des Indexeintrags im Elternknoten der Elternknoten unterläuft?



B⁺-Baum: Löschen – Indexeintrag

- Löschen von einem Indexeintrag $\langle k, p \rangle$ in Knoten n (ähnlich zu vorher):
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p \rangle$ einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a. Wenn Nachbarknoten genügend Einträge ($> d$) hat, umverteilen:
        - Rotiere Eintrag $\langle k', p' \rangle$ aus Nachbarknoten über den Elternknoten in den unterlaufenen Knoten
        i. Suchschlüssel k' ersetzt k'' im Elternknoten
        ii. Eintrag $\langle k'', p' \rangle$ wird im unterlaufenen Knoten eingefügt
     - Beispiel
       - Indexeintrag $\langle 8105, p \rangle$ löschen
       - Rotieren des Schlüssels 5823 in den Elternknoten mit Weiterrotation des Eintrags 6423 im Elternknoten in den unterlaufenen Kindknoten



B⁺-Baum: Löschen – Indexeintrag

- Löschen von einem Indexeintrag mit Schlüssel k in Knoten n ähnlich zu vorher):
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p \rangle$ einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a. Wenn Nachbarknoten genügend Einträge ($> d$) hat, umverteilen:
        - Rotiere Eintrag über den Elternknoten
     b. Sonst (d Einträge der Nachbarknoten):
        - Verschmelze Knoten n und Nachbarknoten n'
        - Ziehe Schlüssel k', der n und n' getrennt hat, in den verschmolzenen Knoten
        - Lösche Indexeintrag $\langle k', p' \rangle$ in Elternknoten (rekursiver Aufruf)



B+-Baum: Delete

Bei der Umsetzung zu beachten:
- Wo liegt der Nachbarknoten gemäß der Ordnung?
- Hat der Nachbarknoten den gleichen Elternknoten?

Versuchen Sie sich gern selbst mal am Pseudocode dafür.

- Eintrag $\langle k, rid \rangle$ löschen
  1. Finde Blattseite l, in der Eintrag für $\langle k, rid \rangle$ steht
  2. Falls l genügend gefüllt (min. $d + 1$ Einträge), Eintrag löschen
  3. Sonst (d Einträge):
     a. Wenn Nachbarblatt n genügend Einträge ($> d$) hat: Mit Eintrag aus n auffüllen und Indexeintrag im Vorfahren aktualisieren
     b. Sonst (Nachbarblätter haben auch nur d Einträge): Nachbarblatt n und l verschmelzen und Indexeintrag in Vorfahren löschen

- Indexeintrag $\langle k, p \rangle$ in Knoten n löschen
  1. Wenn n genügend Einträge ($> d$) hat, dann $\langle k, p \rangle$ einfach löschen
  2. Sonst: (Umverteilen oder verschmelzen)
     a. Wenn Nachbarknoten genügend Einträge ($> d$) hat, umverteilen:
        - Rotiere Eintrag über den Elternknoten
        - Eventuell Großelternknoten aktualisieren
     b. Sonst (d Einträge der Nachbarknoten):
        - Verschmelze Knoten n und Nachbarknoten n'
        - Ziehe Schlüssel k', der n und n' getrennt hat, in den verschmolzenen Knoten
        - Lösche Indexeintrag $\langle k', p' \rangle$ in Elternknoten (rekursiver Aufruf)

B^+-Bäume in realen Systemen

- Implementierungen verzichten auf die Kosten der Verschmelzung und der Neuverteilung und weichen die Regel der Minimumbelegung auf
  - Beispiel: IBM DB2 UDB
    - MIN**PCT**USED als Parameter zur Steuerung der Blattknotenverschmelzung (Online-Indexreorganisation)
    - Innere Knoten werden niemals verschmolzen (nur bei Reorganisation der gesamten Tabelle)
- Zur Verbesserung der Nebenläufigkeit evtl. nur Markierung von Knoten als gelöscht (keine aufwendige Neuverzeigerung)

PCT = Partition Change Tracking

Erzeugung von Indexstrukturen in SQL

- Implizite Indexe
  - Indexe automatisch erzeugt für Primärschlüssel und Unique-Integritätsbedingungen
    - Werden z.B. bei Einfügeoperationen genutzt für effiziente Prüfung, ob Wert schon existiert
      - Kein Zugriff auf Datensatz selbst nötig
    - Schnelle Findung von Einträgen anhand des Primärschlüssels
- Explizite Indexe: CREATE INDEX $Name$ ON $R(A_1, \ldots, A_n)$
  - Wissen über häufige Anfragen nutzen
  - Einfache Indexe über ein Attribut, zusammengesetzte Indexe (Verbundindexe) über mehrere Attribute einer Tabelle
  - Beispiel
    - Bei häufigen Anfragen mit Selektion der PLZ
    - **create index** PlzIndex **on** Kunden(Plz)

```
create index IndexName
on TableName(Attr);
```

```
create index IndexName
on TableName(Attr1, Attr2, ..., Attrn);
```

Indexe mit zusammengesetzten Schlüsseln

- Voraussetzung: Dinge haben eine definierte **totale Ordnung**
  - Integer, Zeichenketten, Datumsangaben, ...
    - In einigen Implementierungen können lange Zeichenketten nicht als Index verwendet werden
  - Auch Hintereinandersetzung möglich
    - Z.B. basierend auf einer lexikographischen Ordnung
- Beispiel:
  - Bei häufigen Anfragen nach Kundennach- und –vornamen (in der Reihenfolge)
  - ```
    create index idx_nname_vname
    on Kunden(Nachname,Vorname)
    ```
    ```

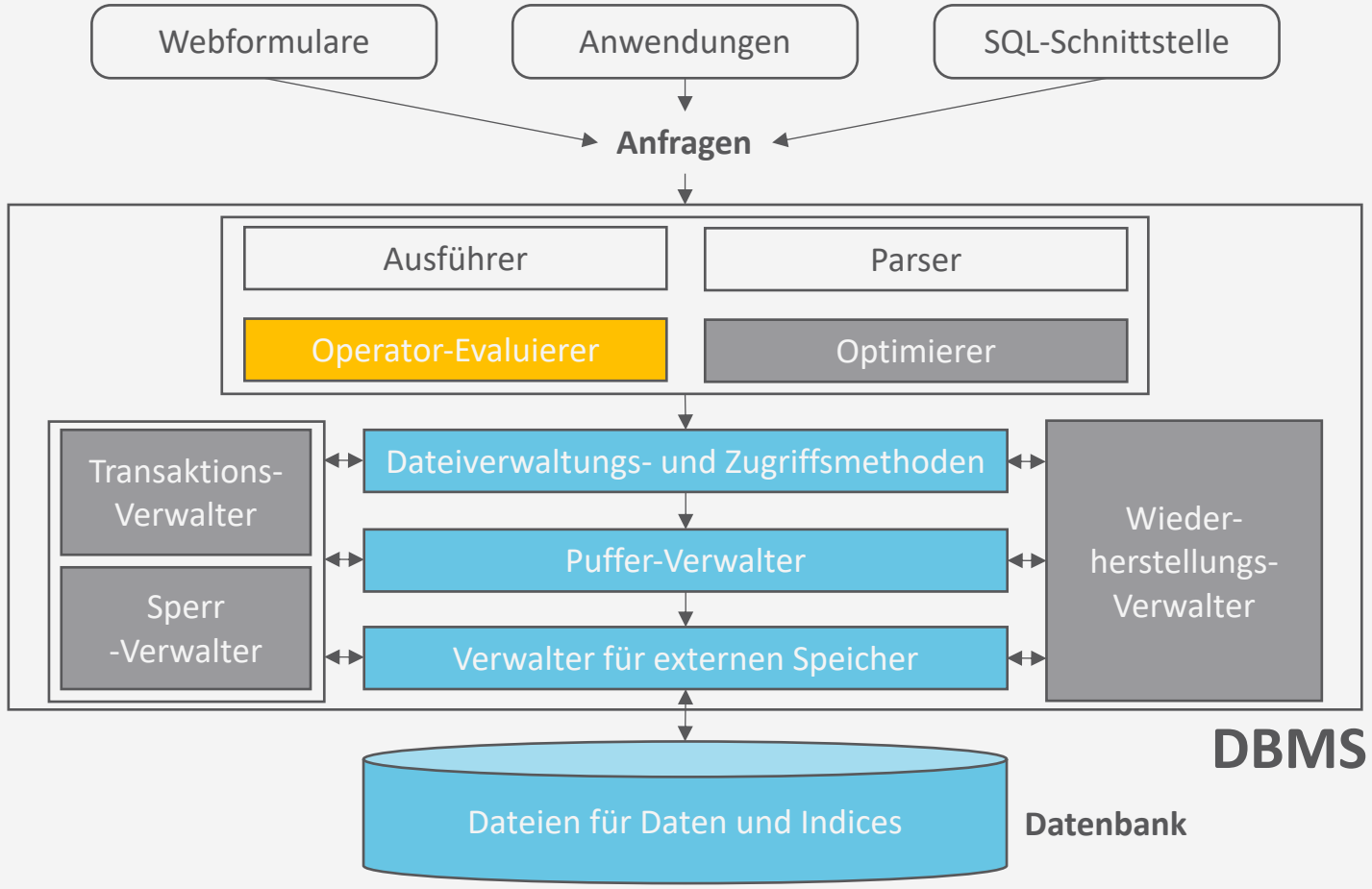
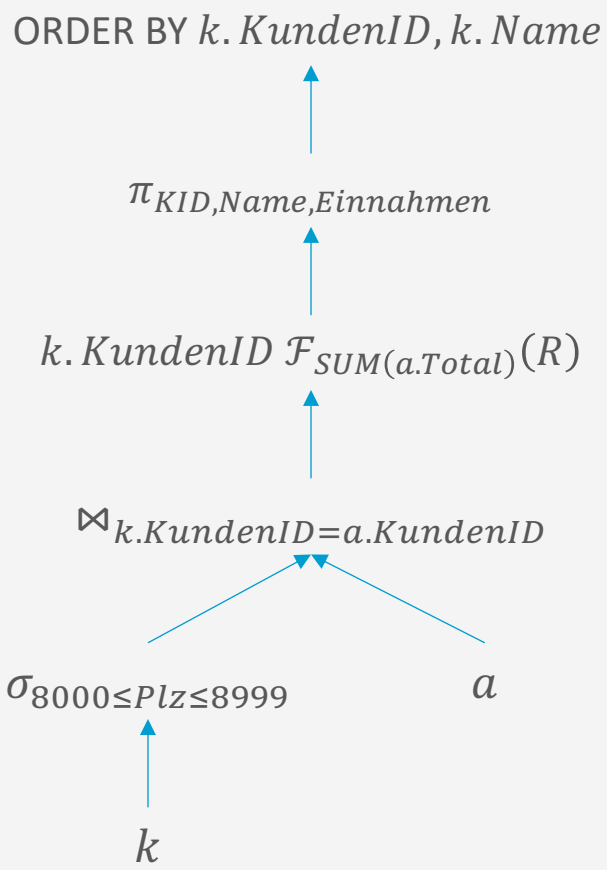
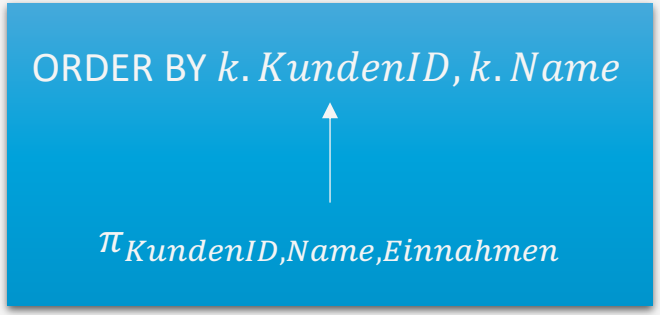
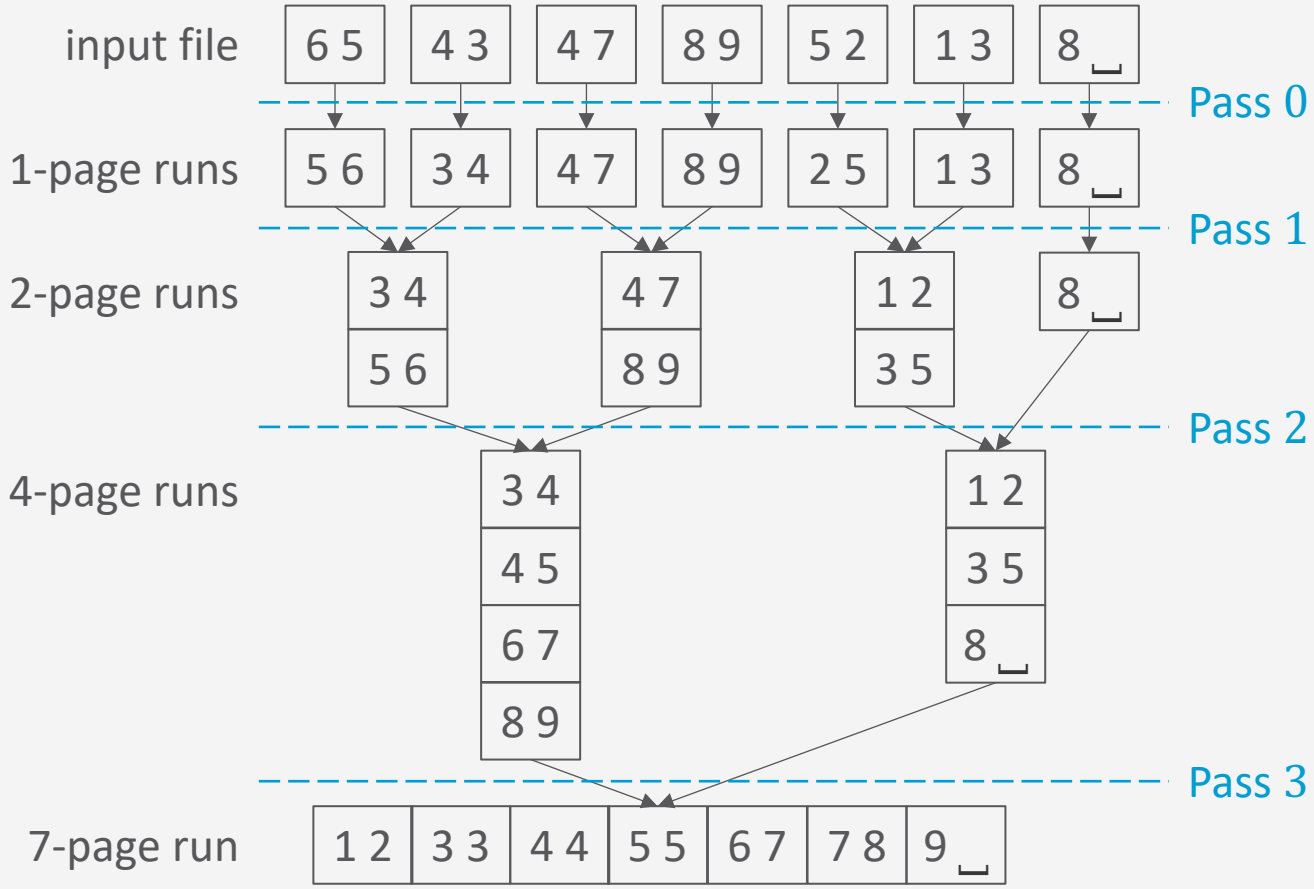
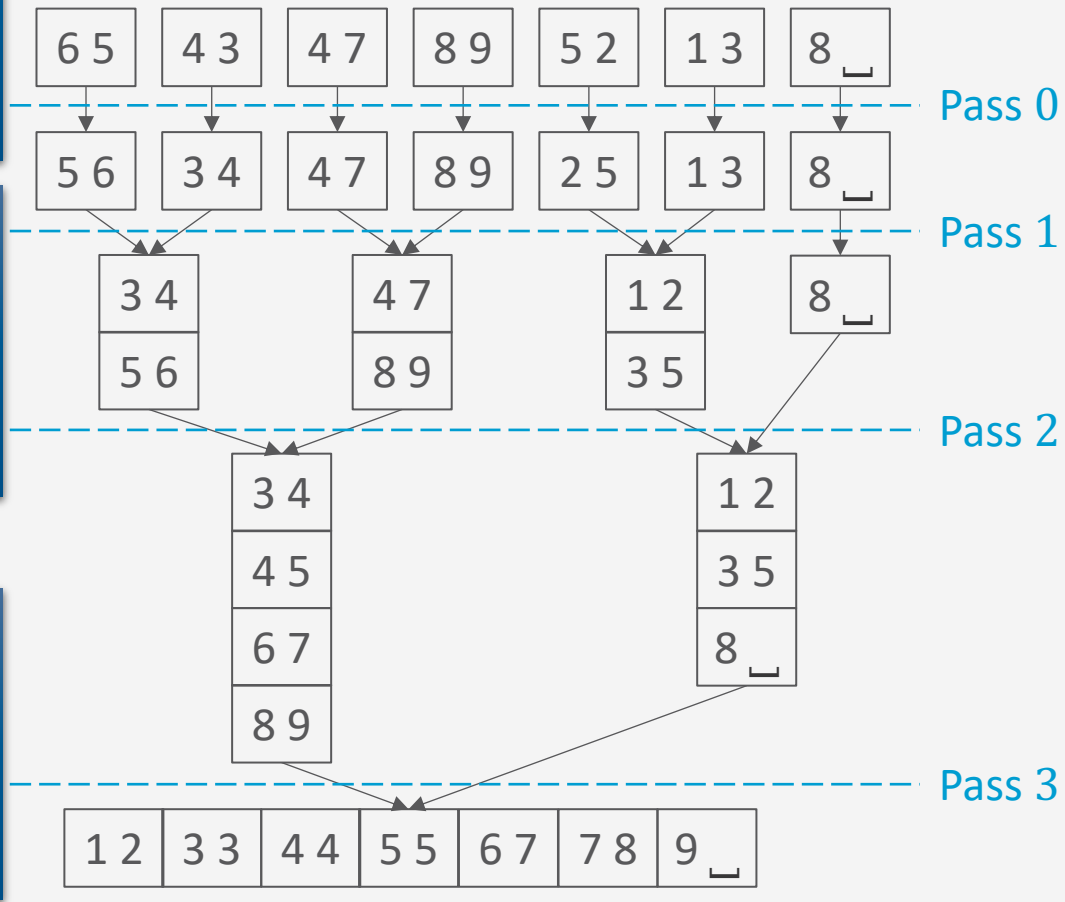
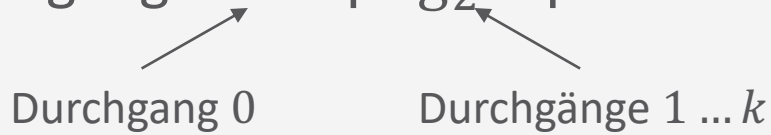
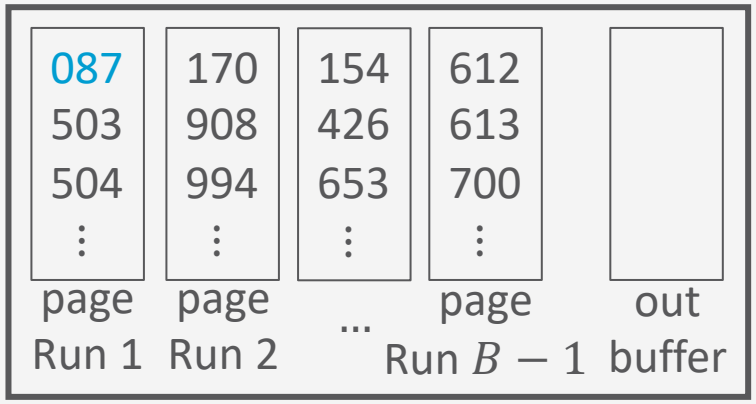
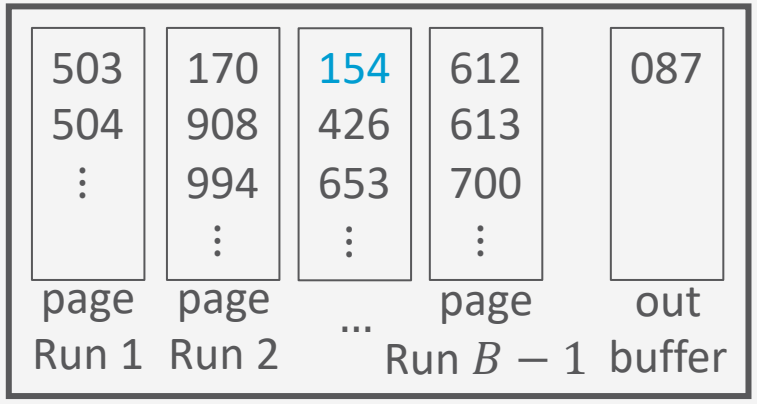
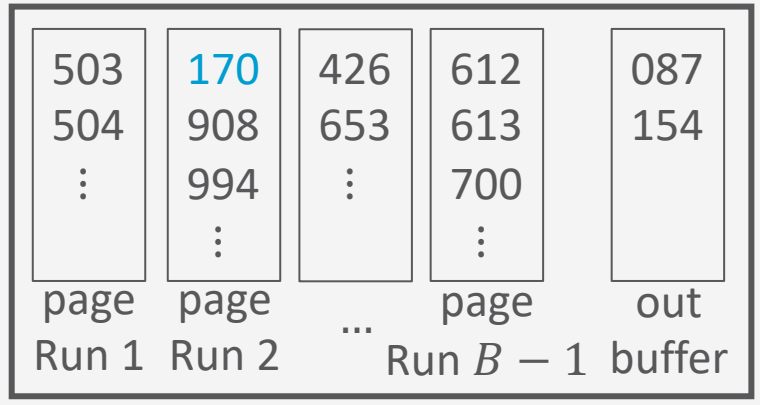
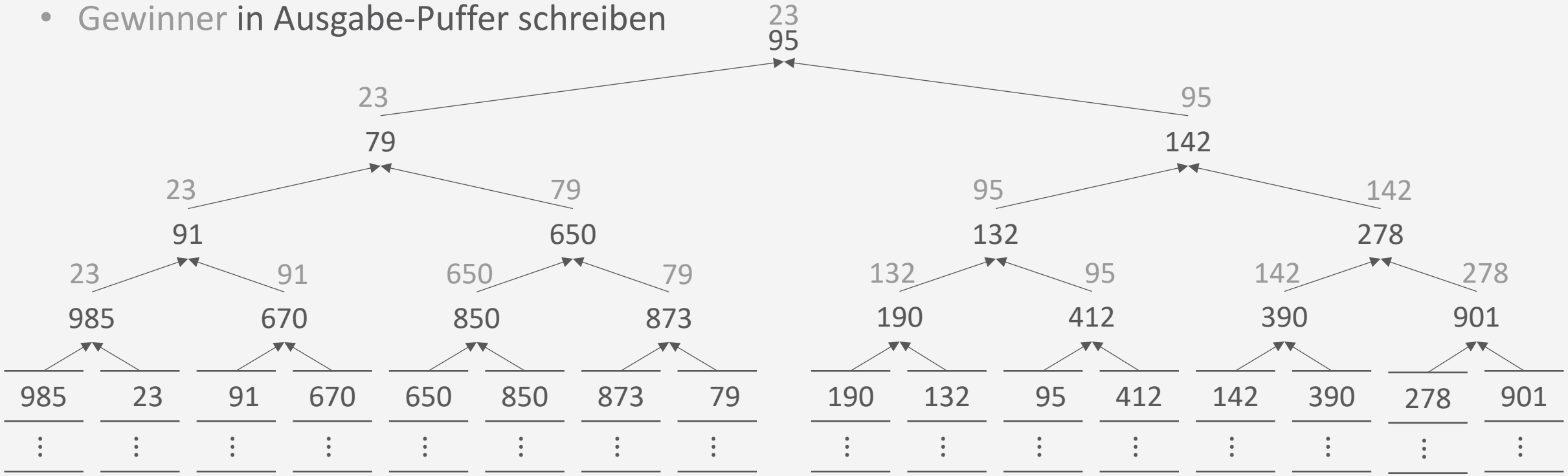
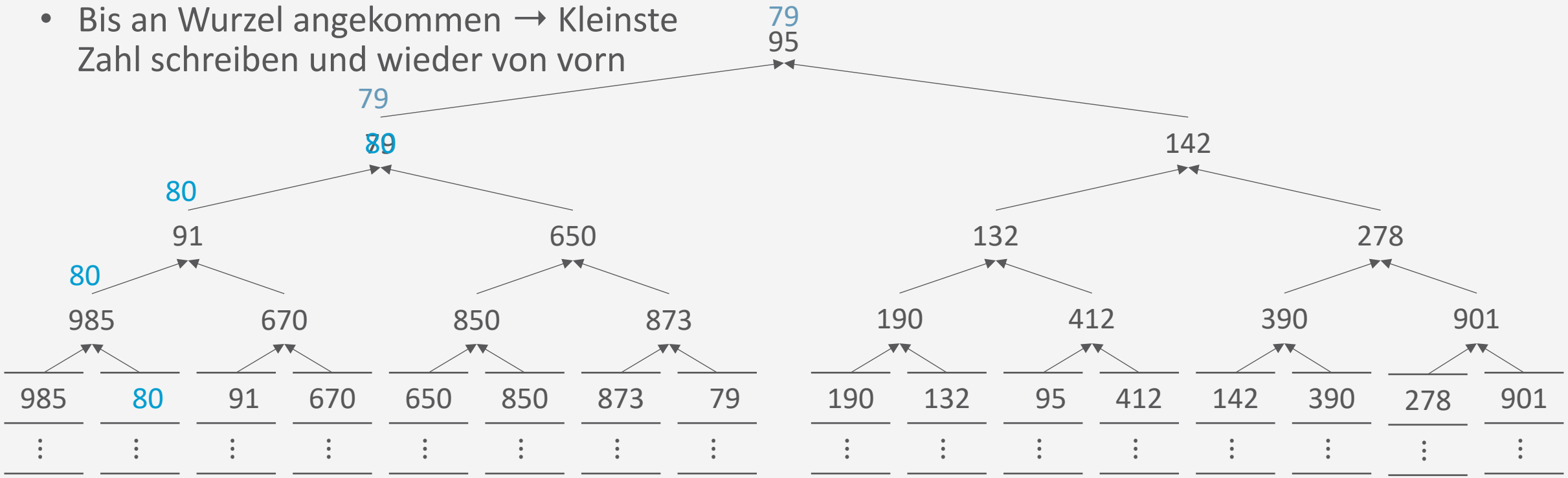
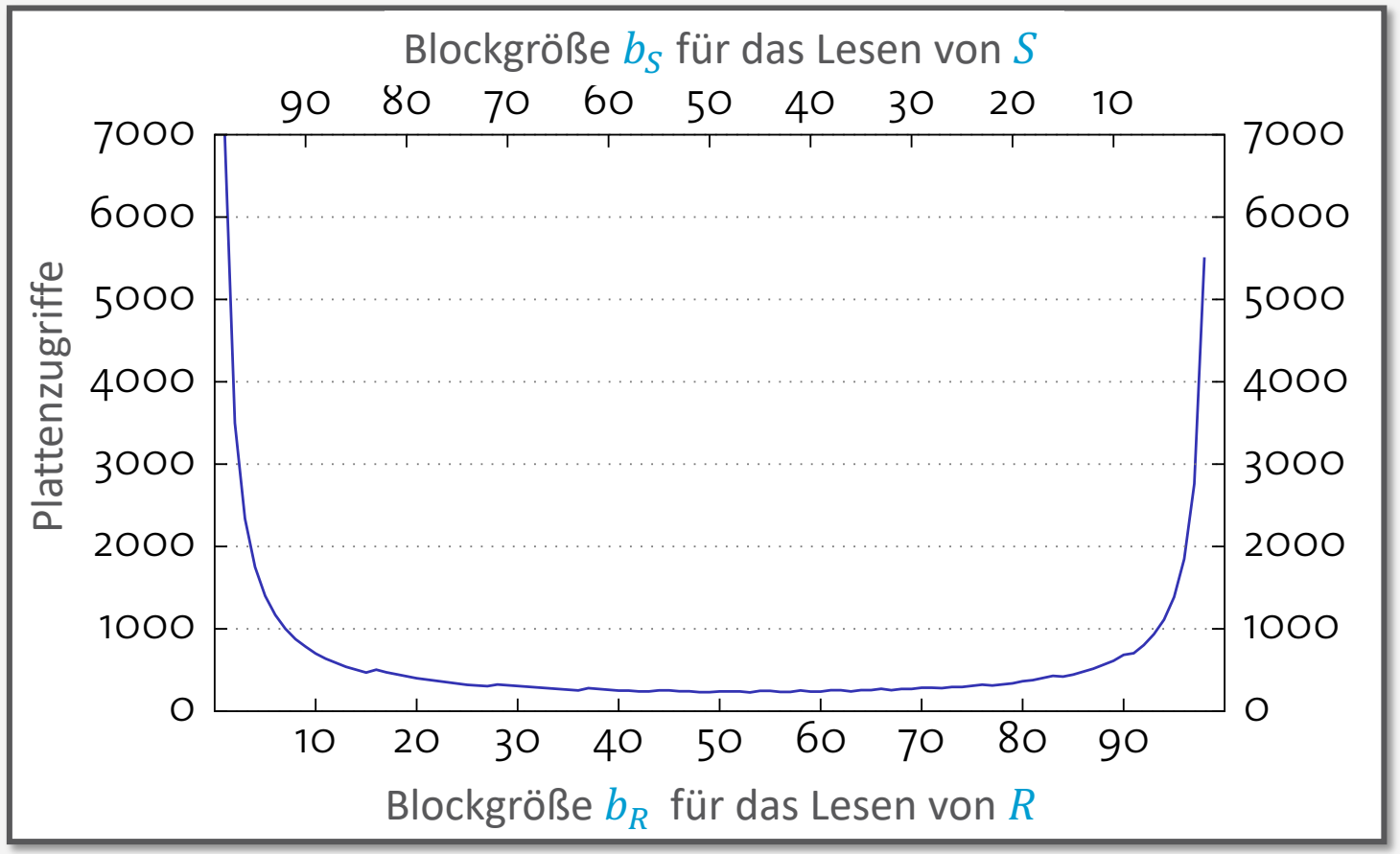
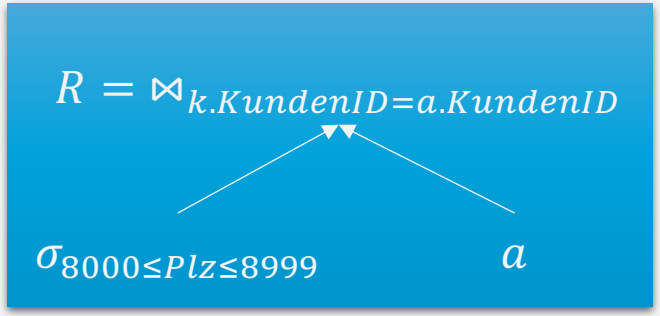
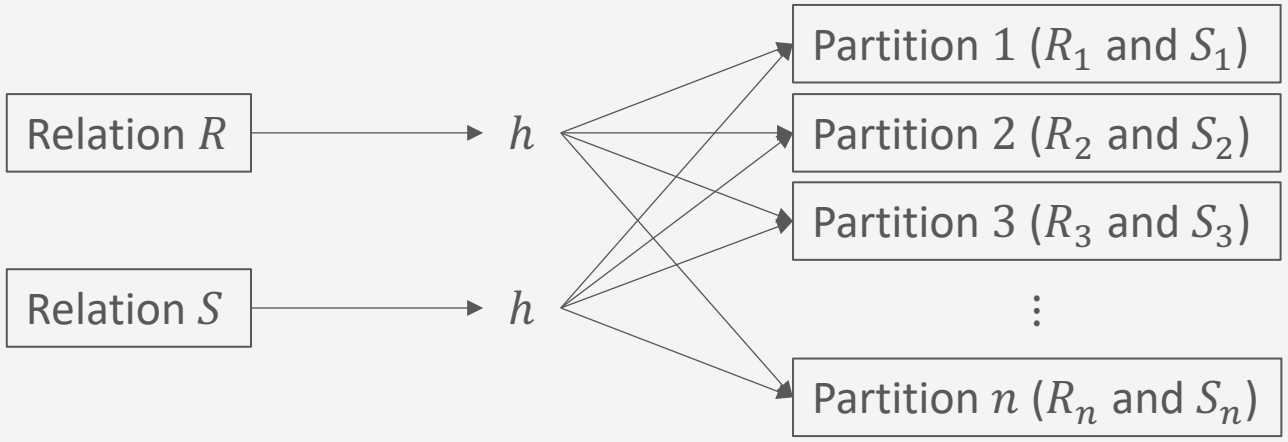
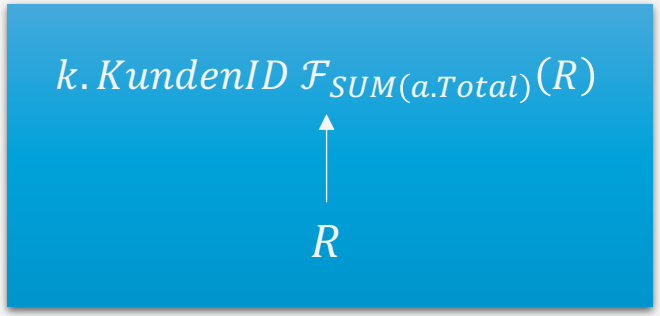
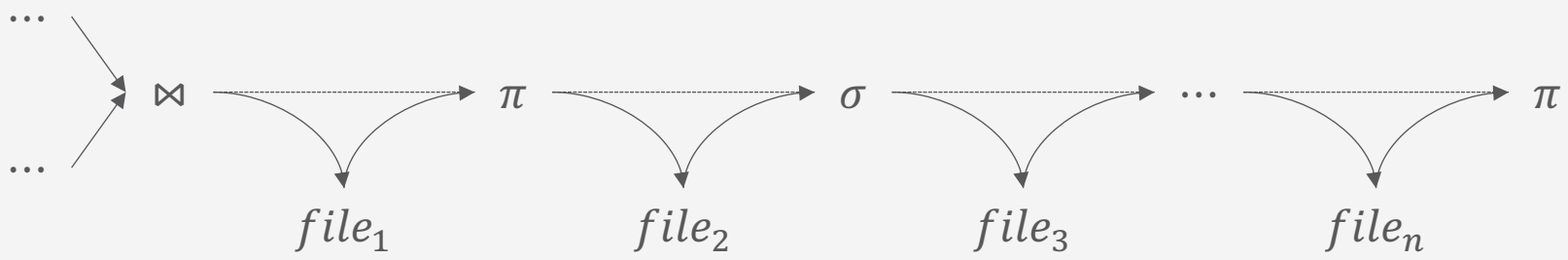
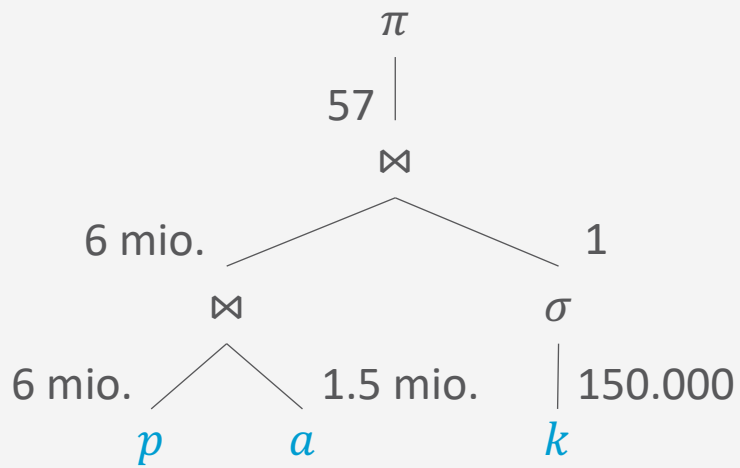
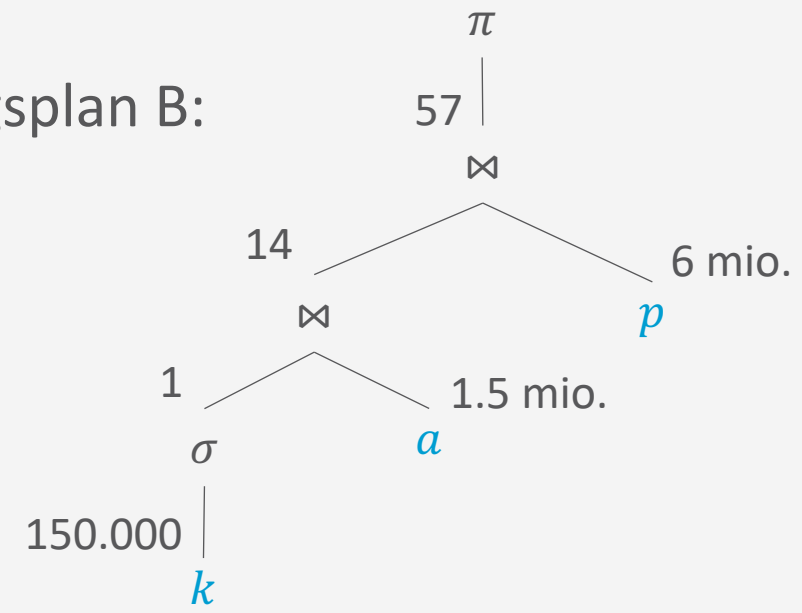
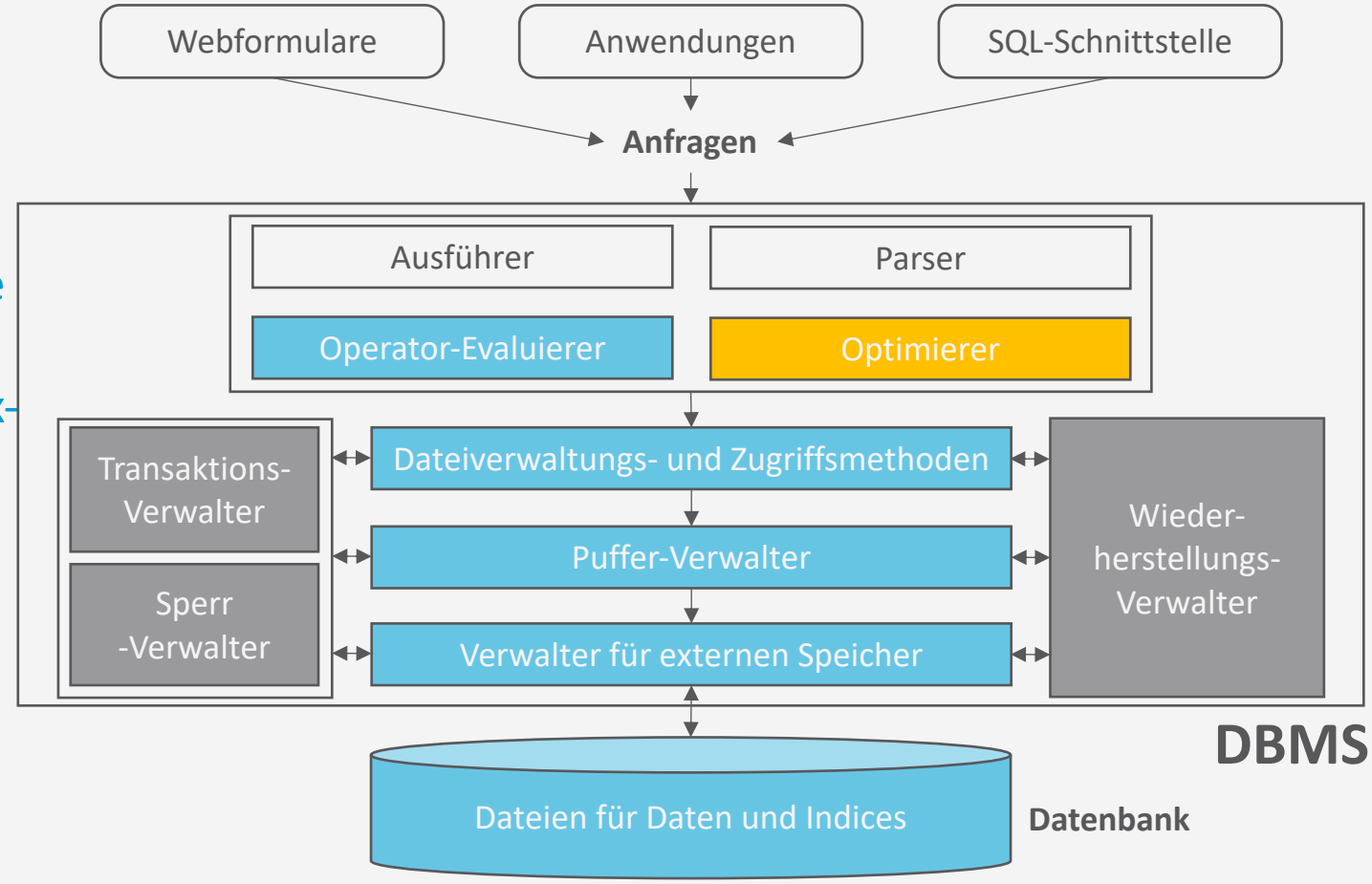
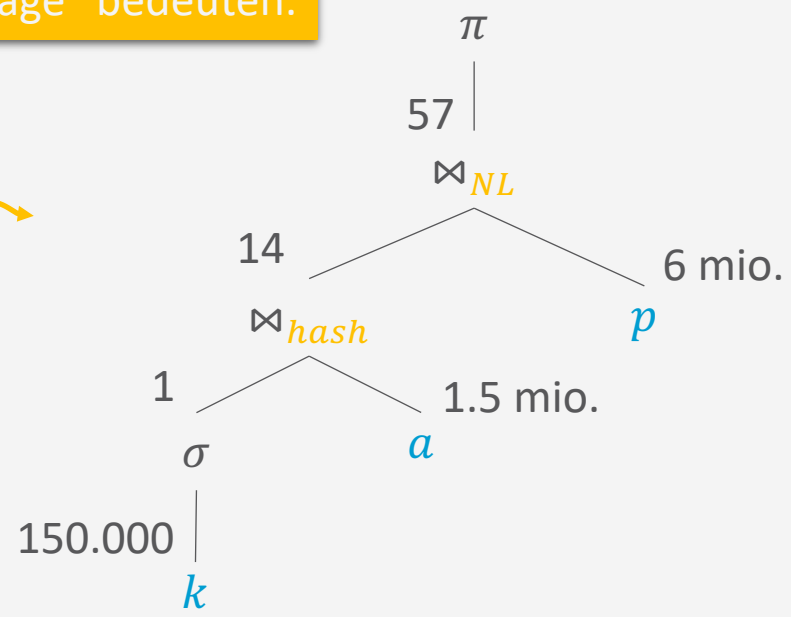
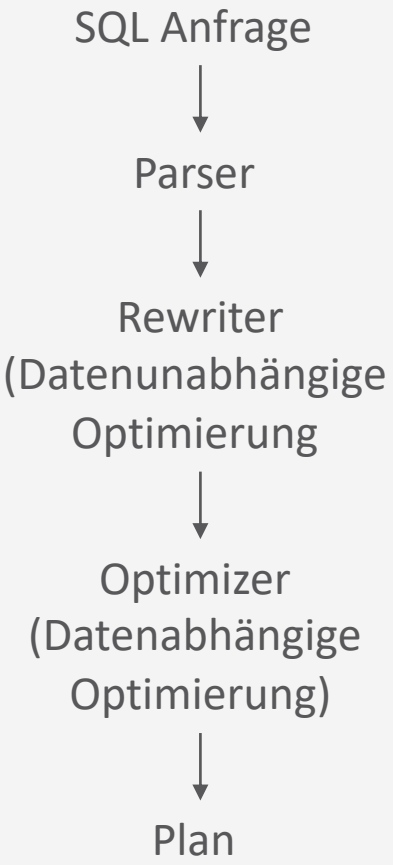
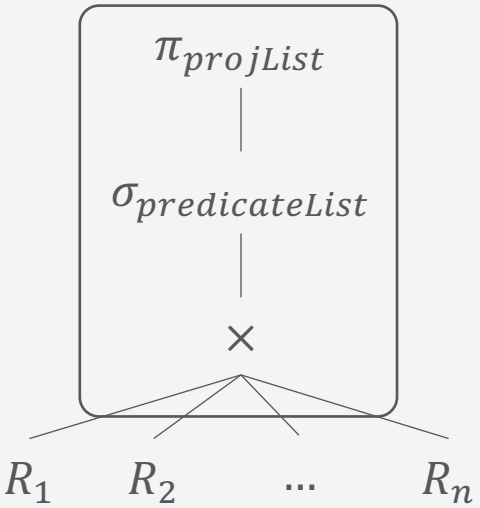
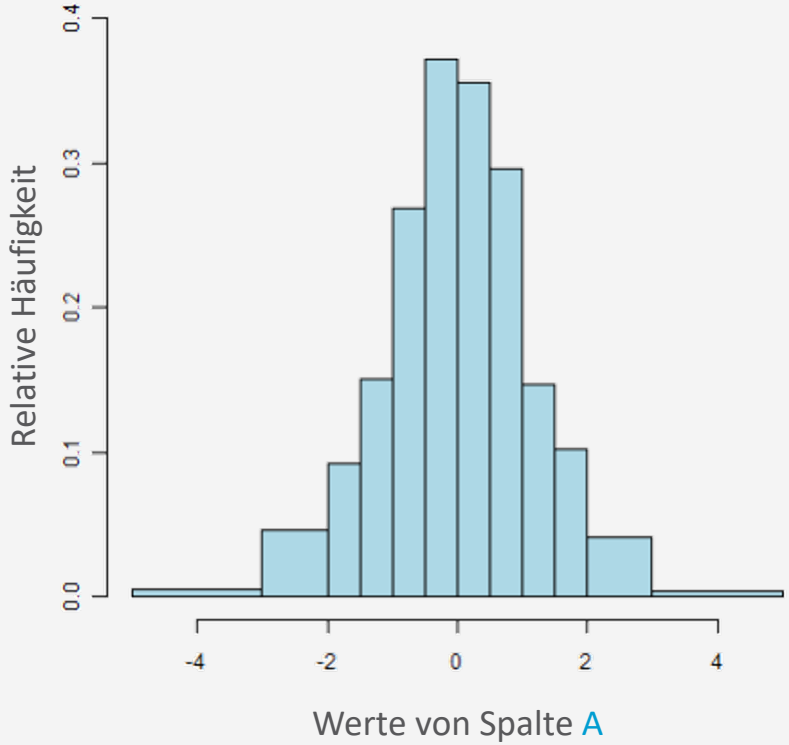
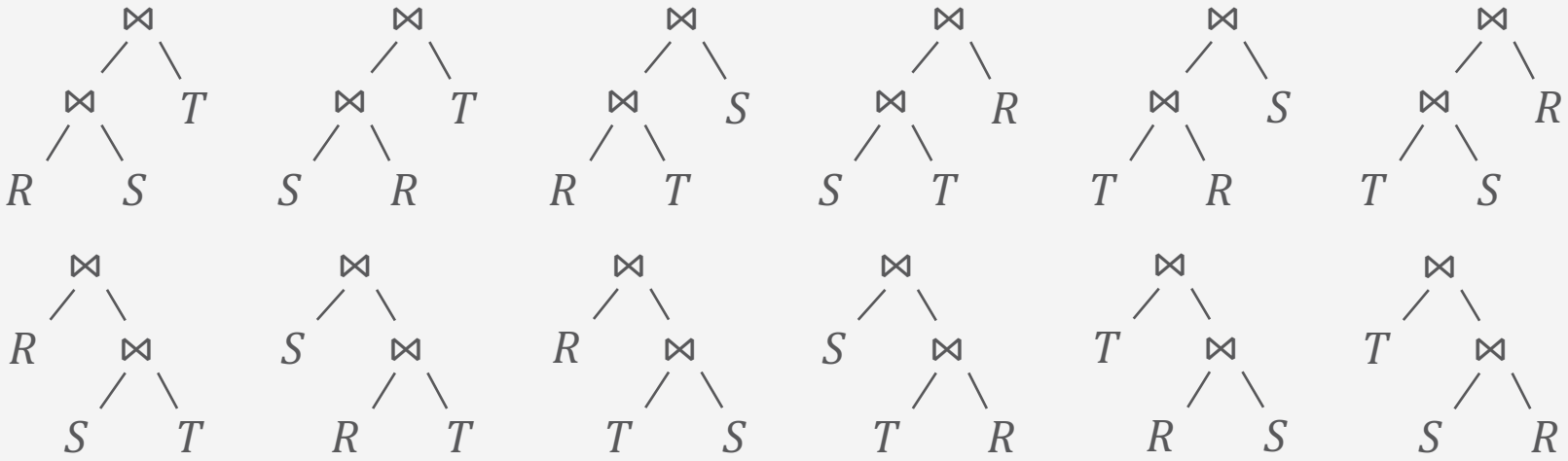
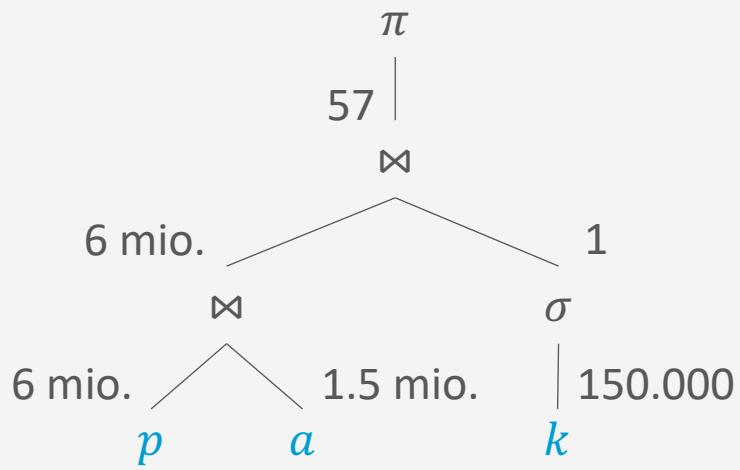
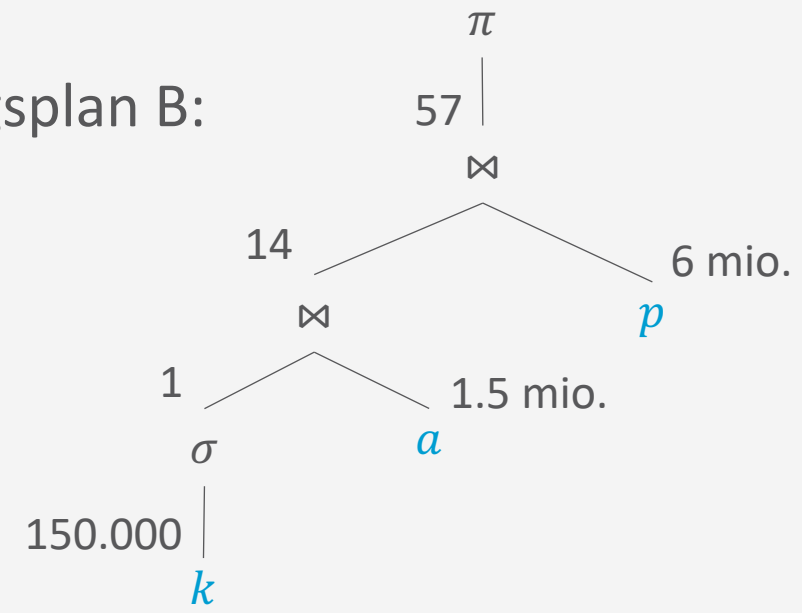
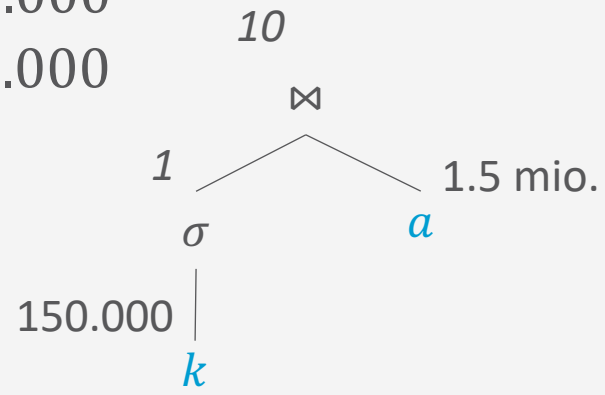
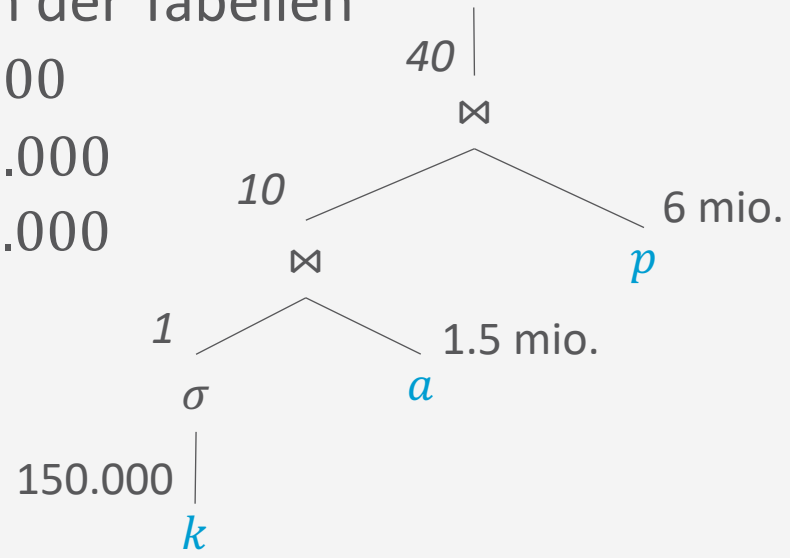
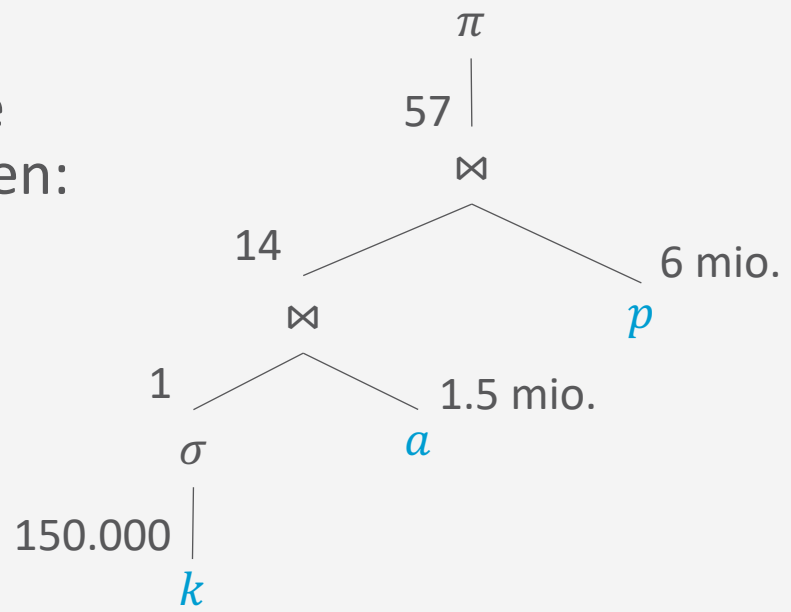
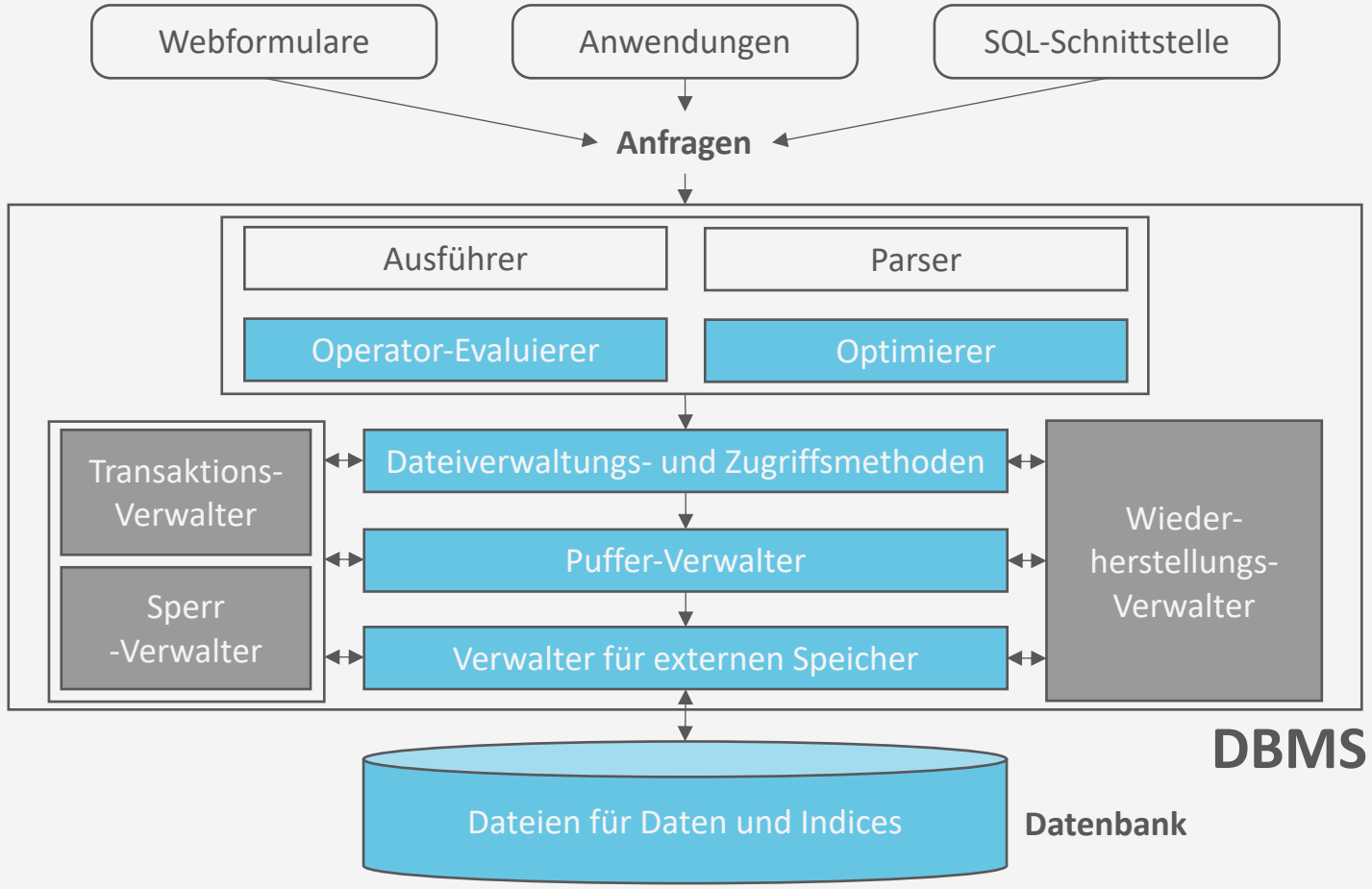
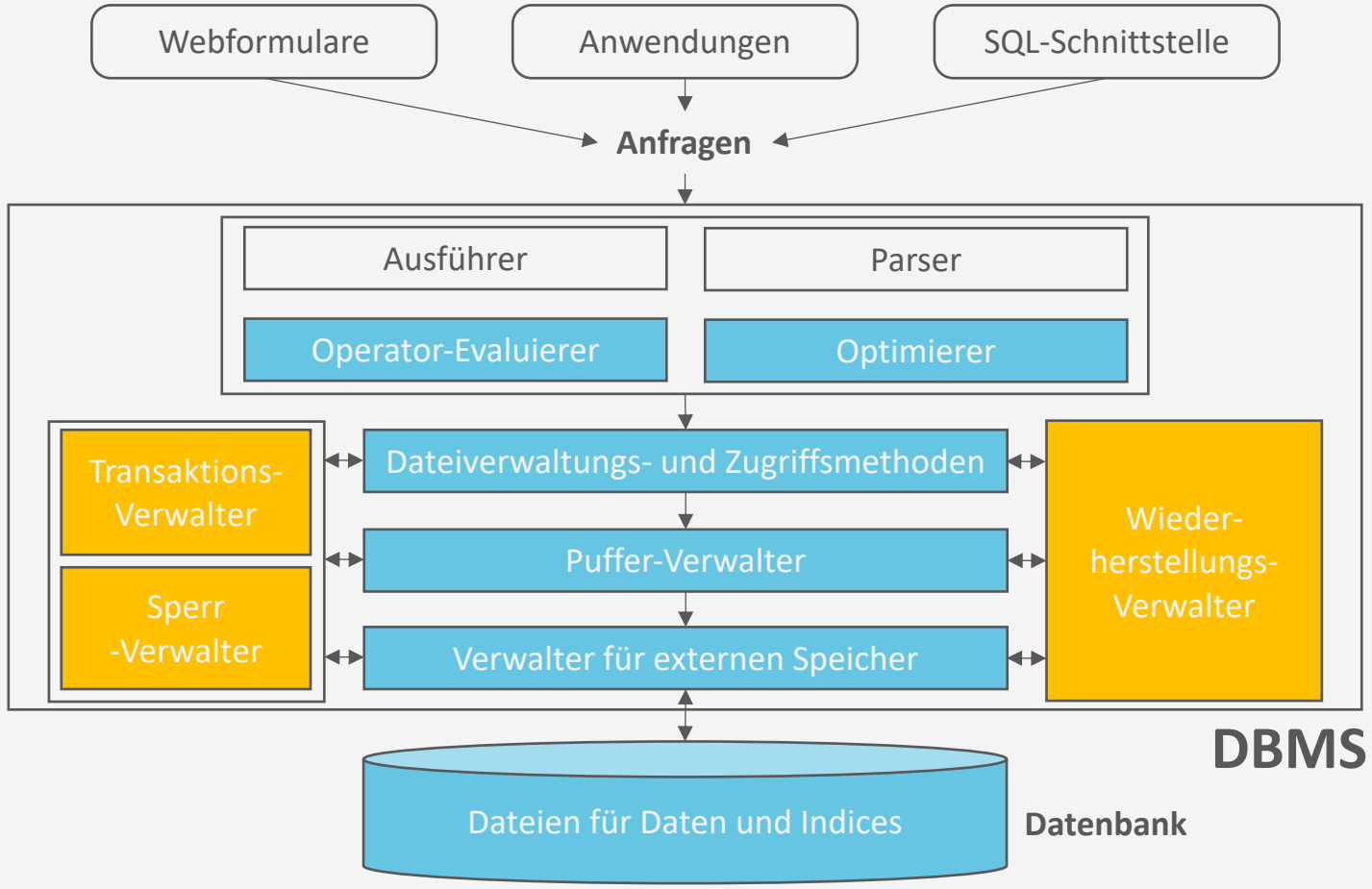
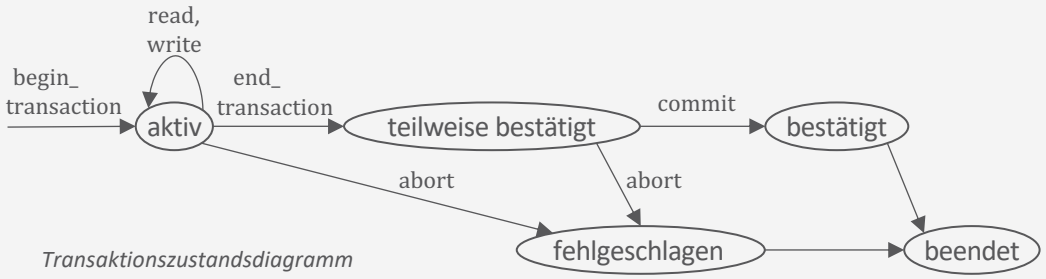
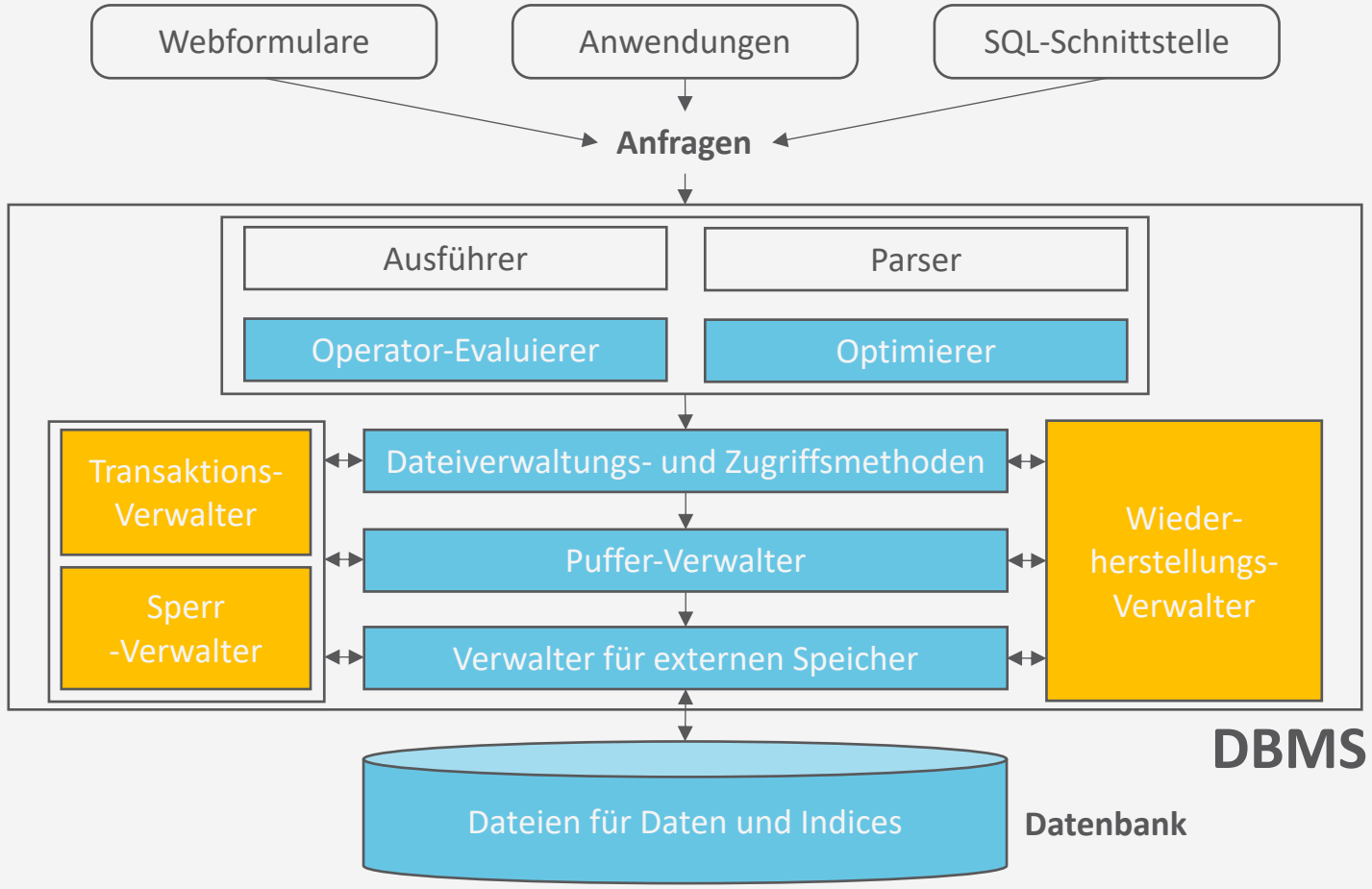
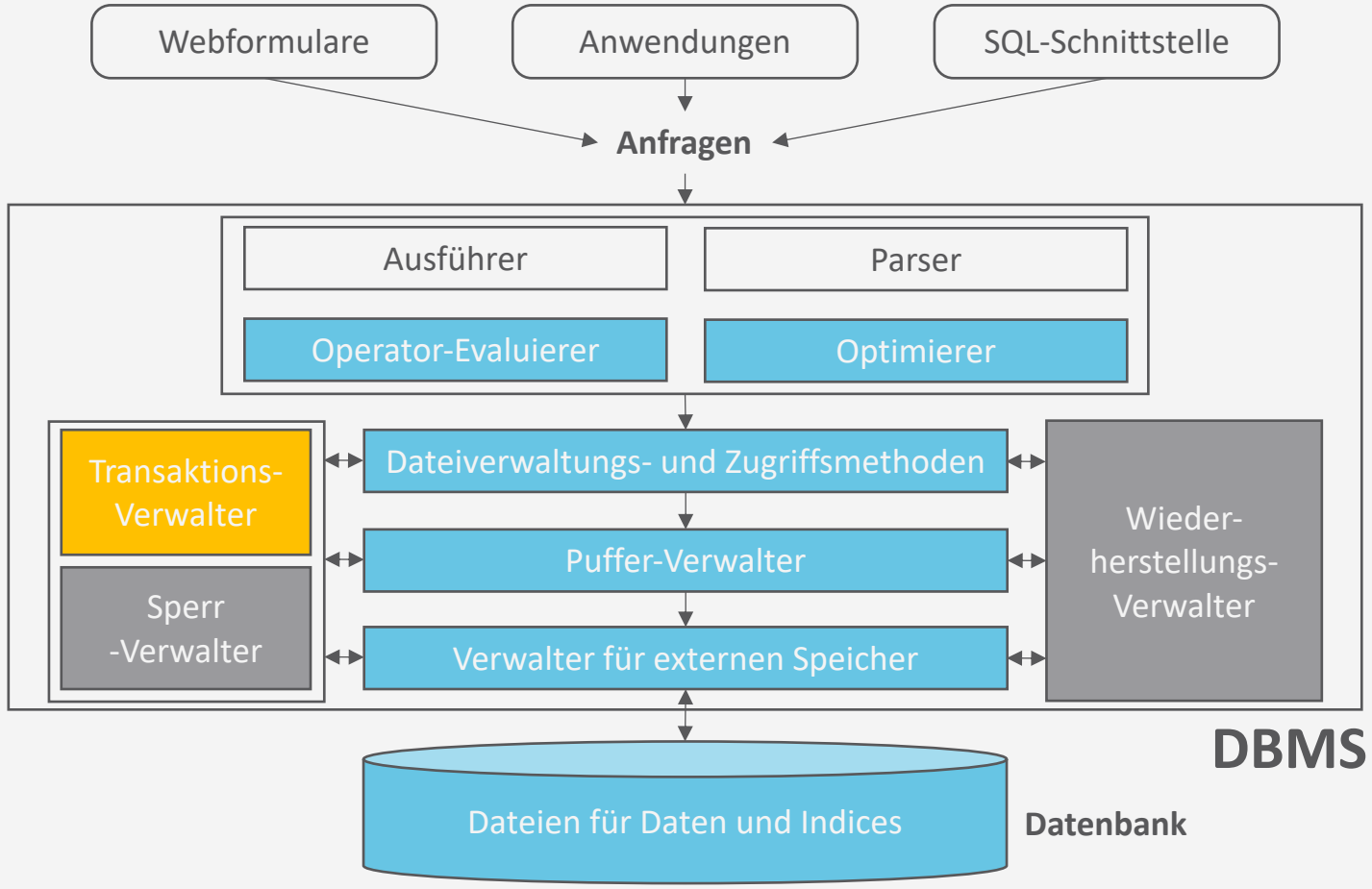
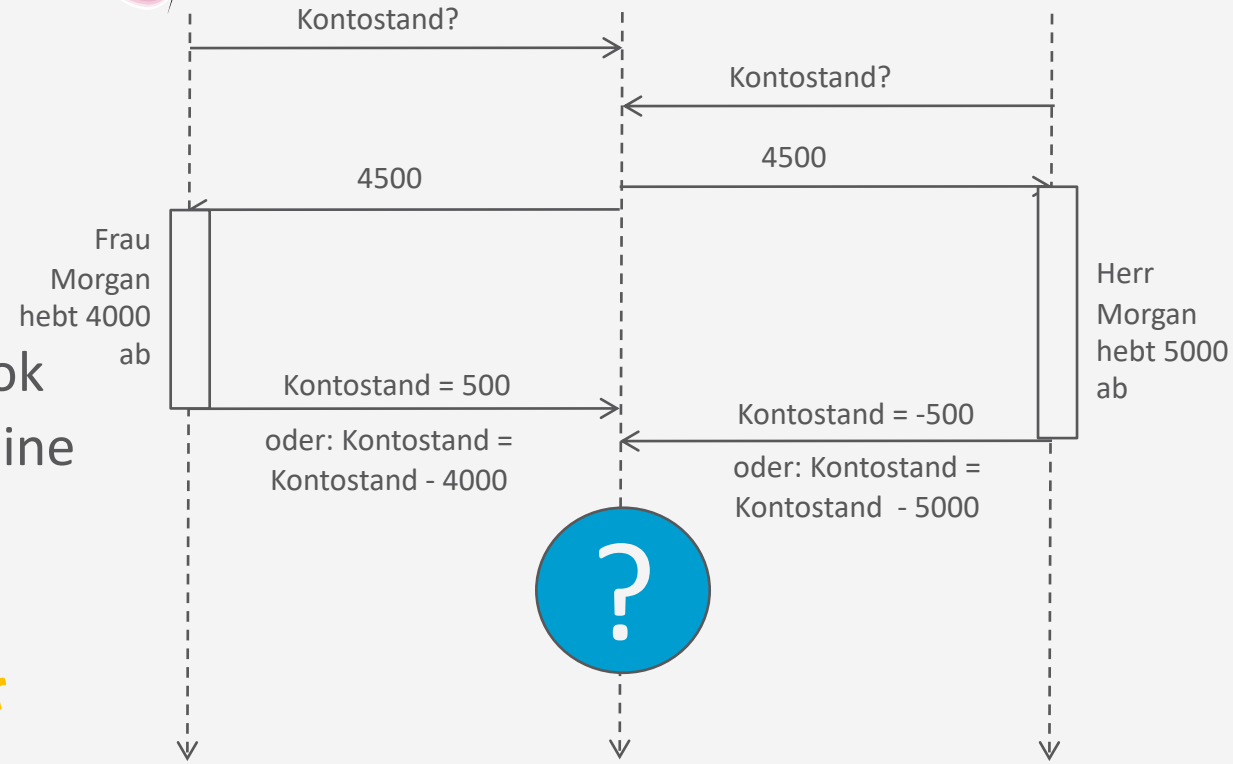
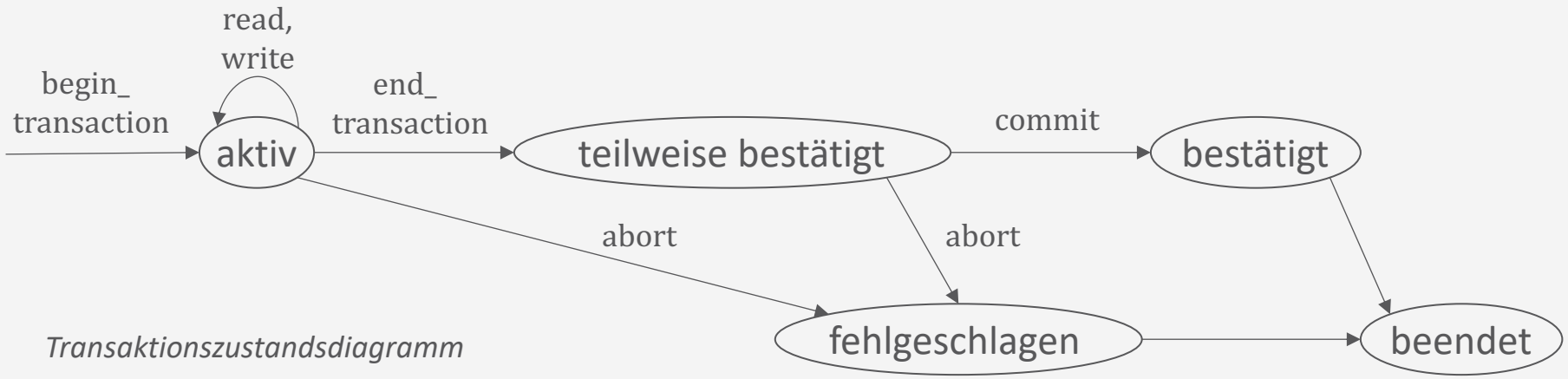
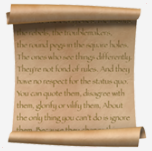
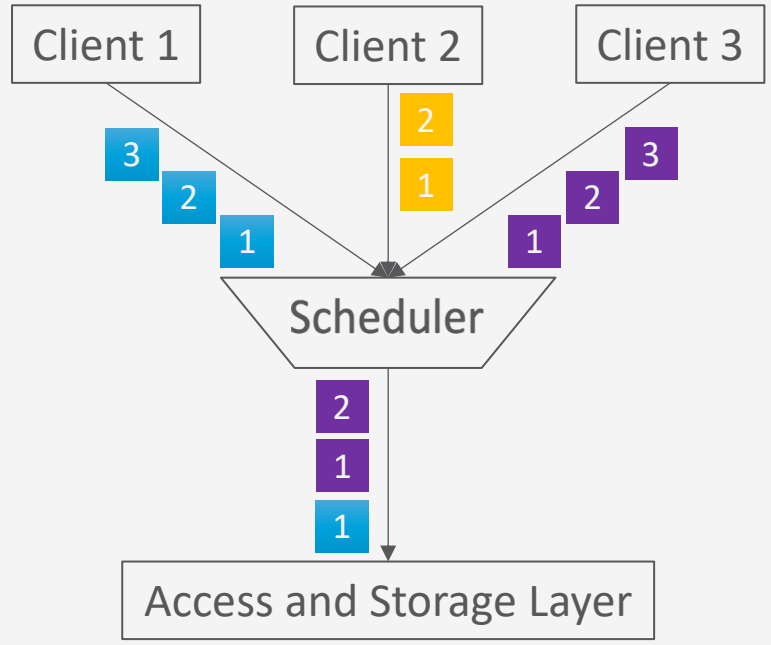
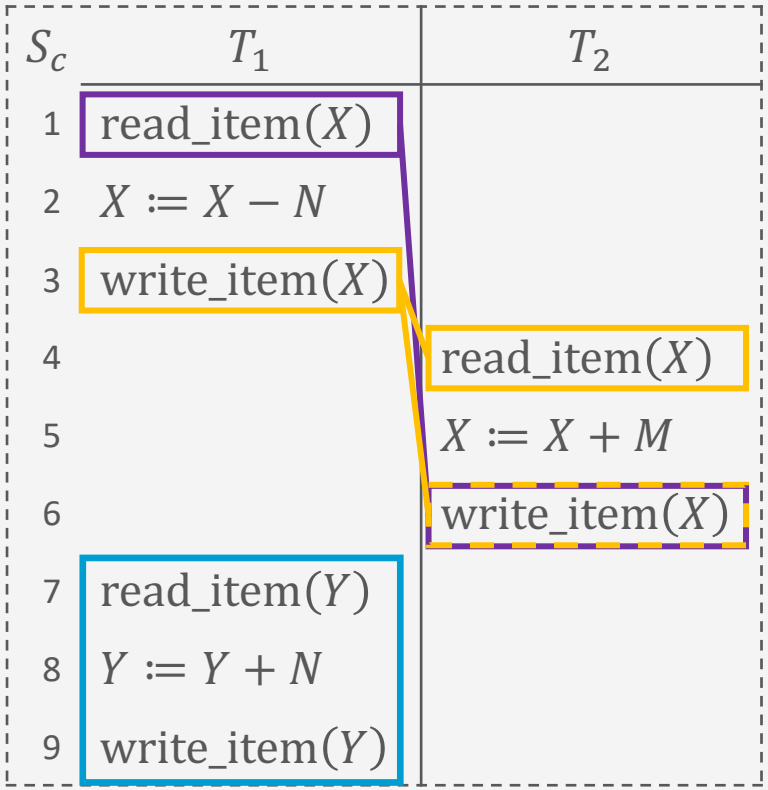
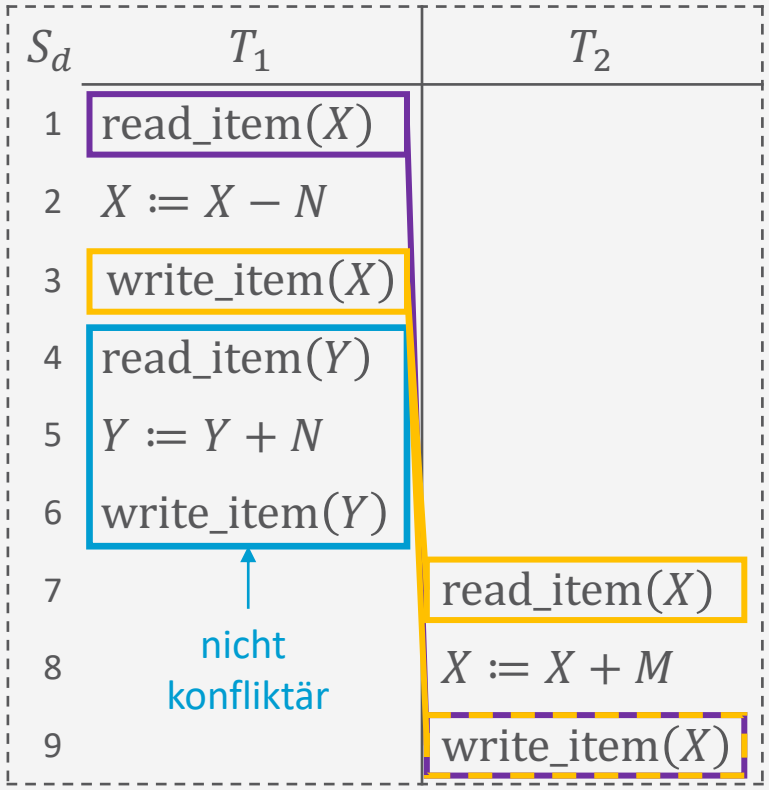
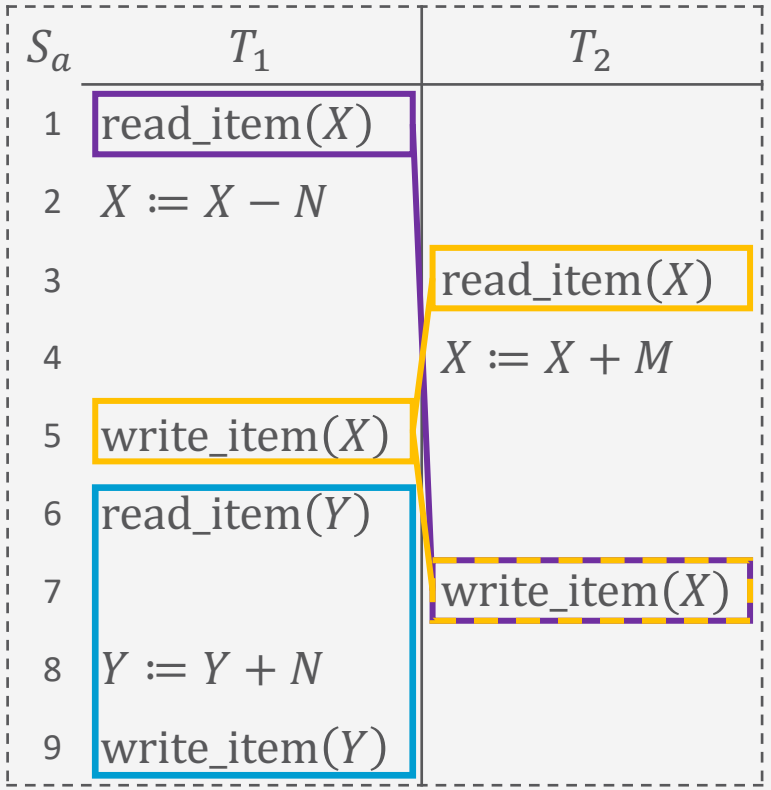
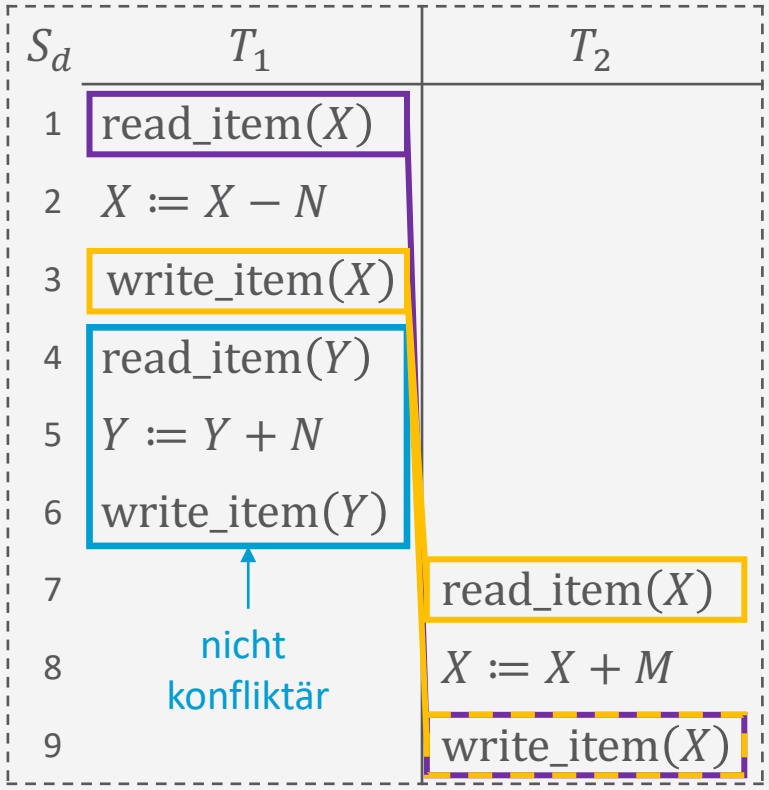
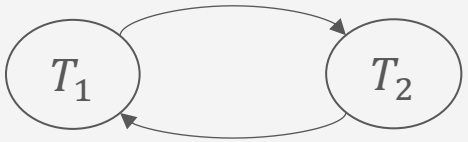
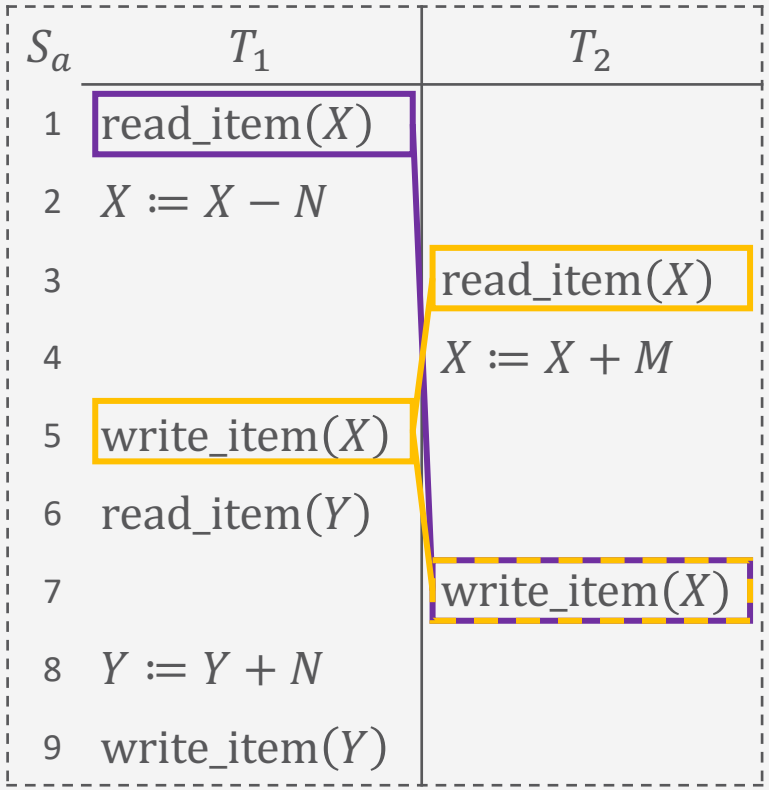
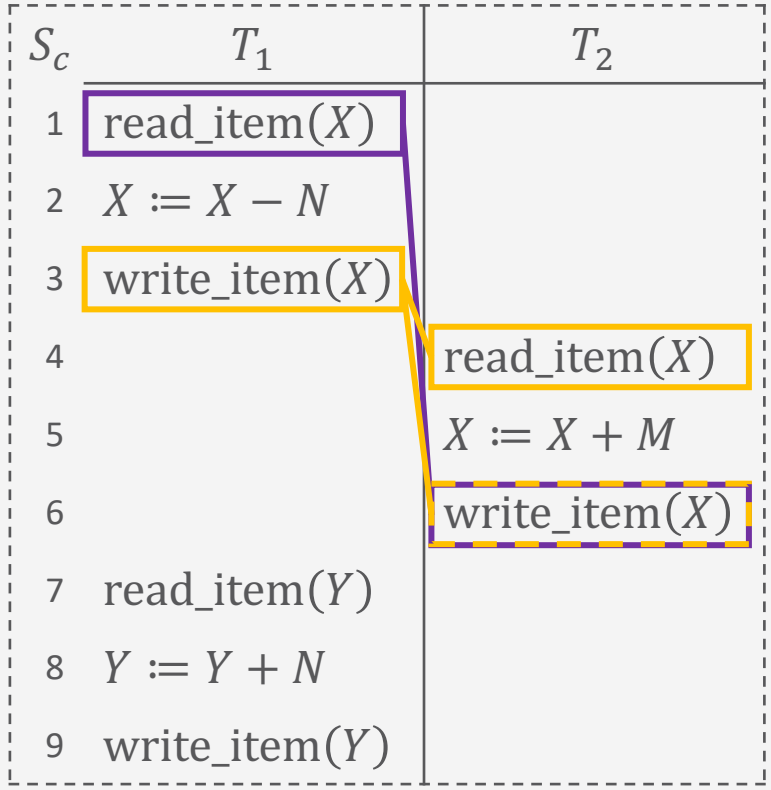
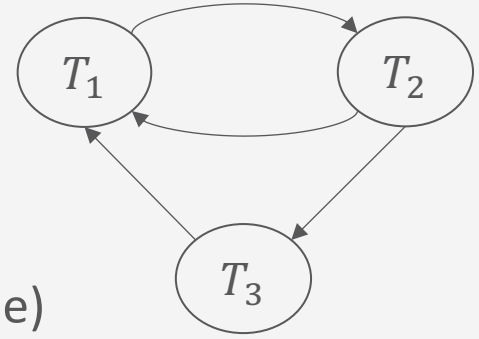
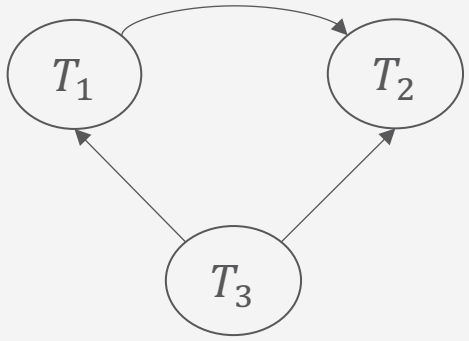
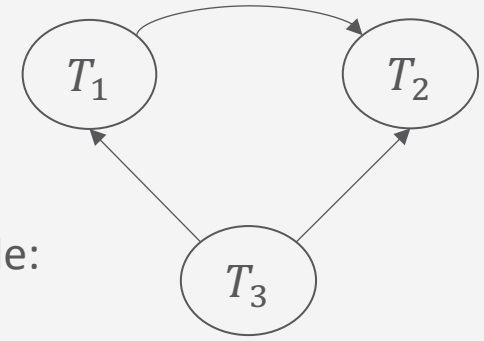
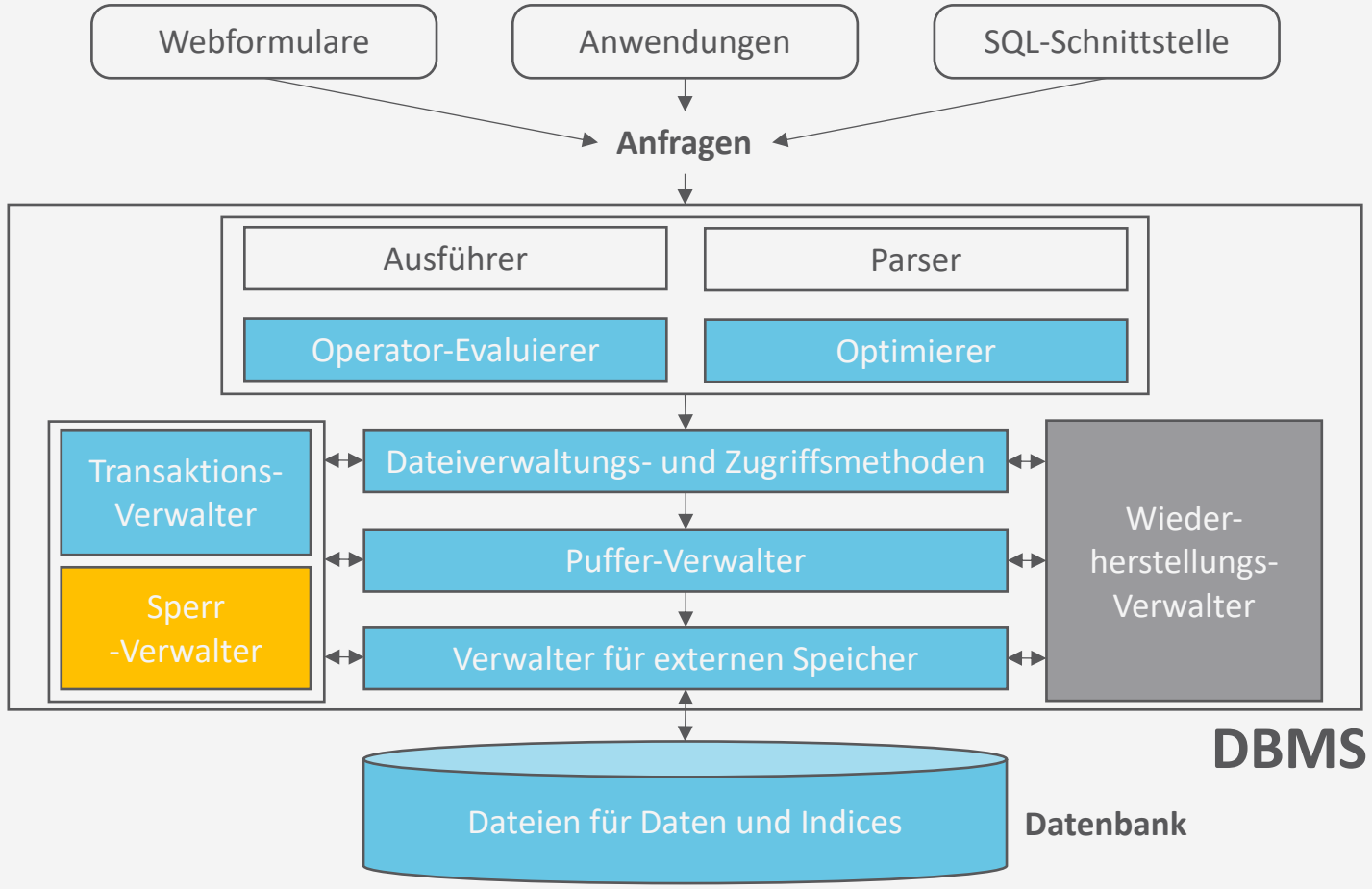
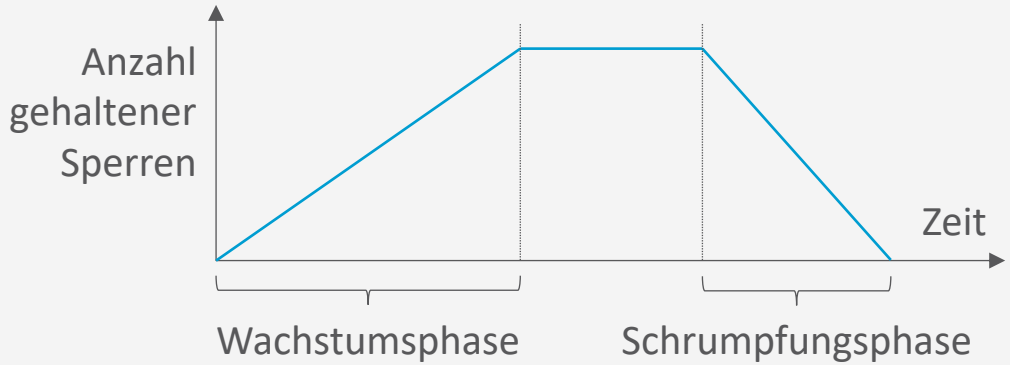
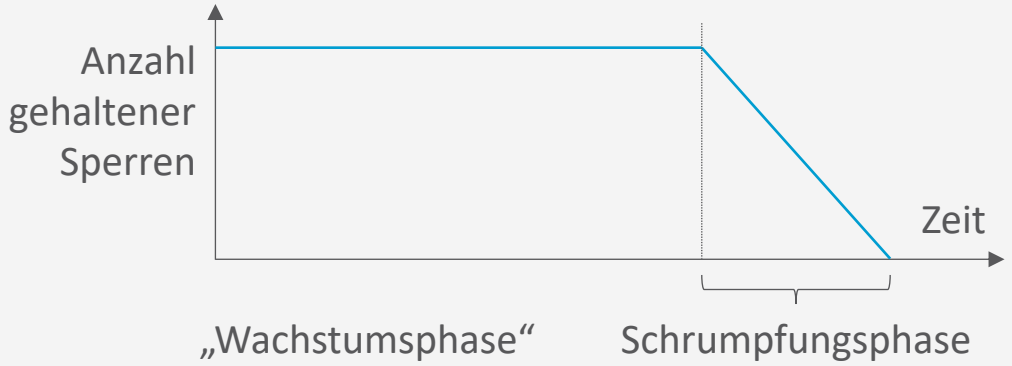
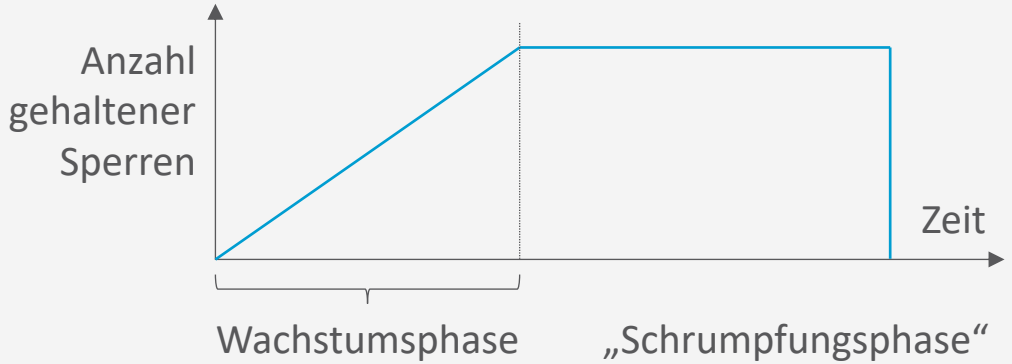
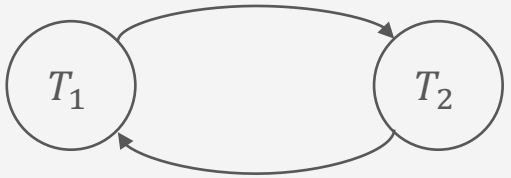
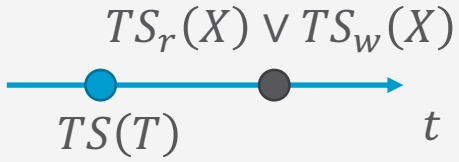
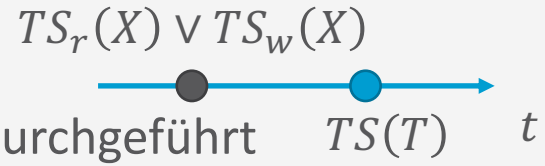
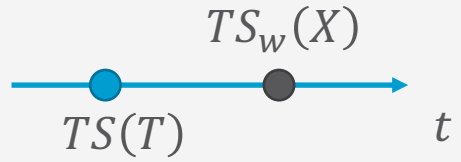
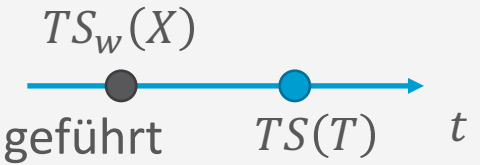
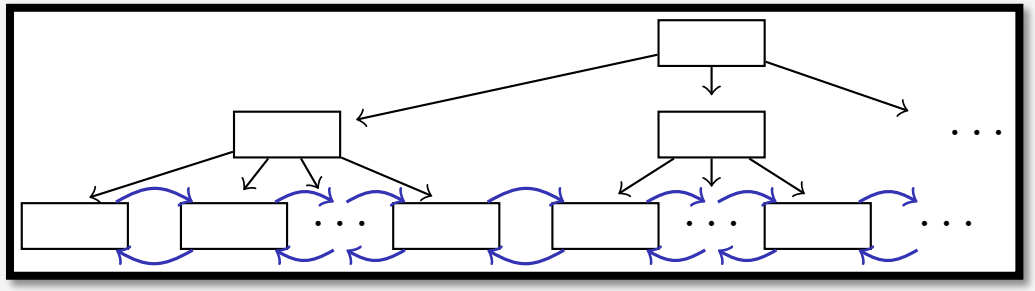
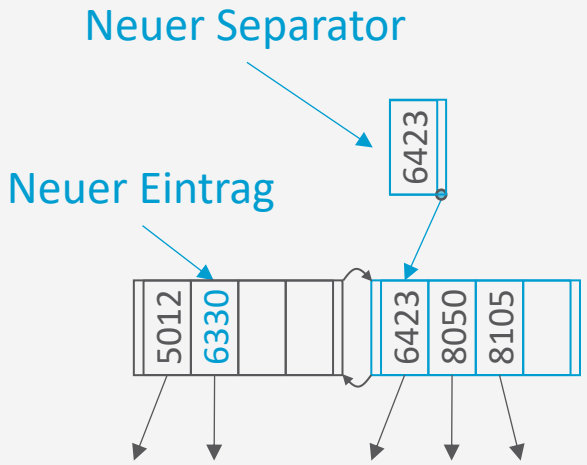
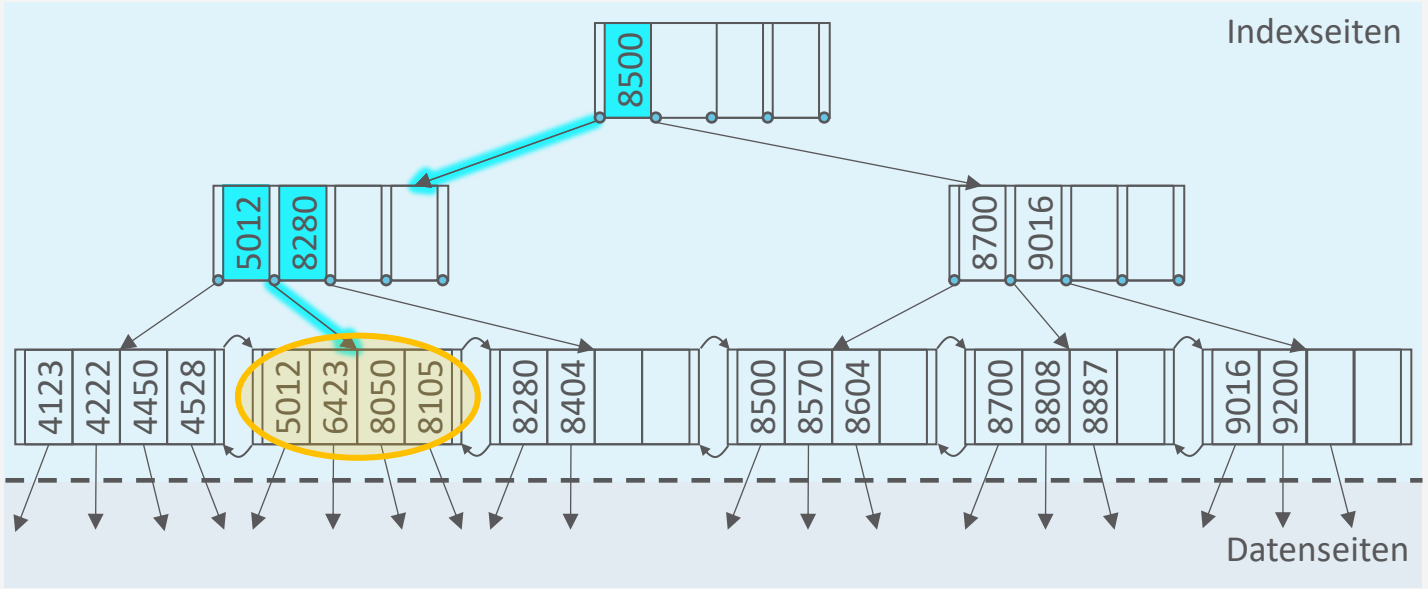
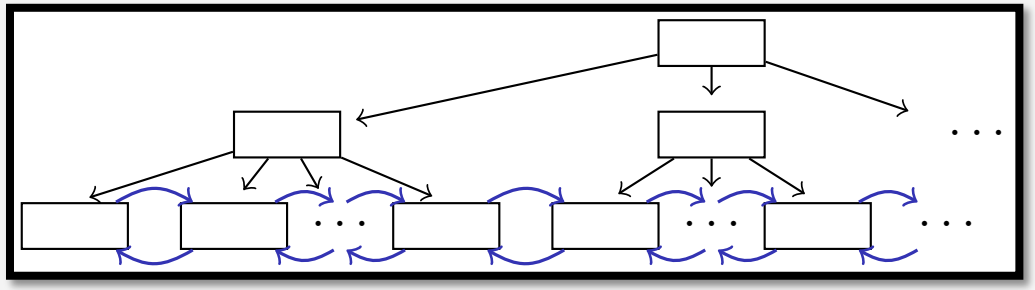
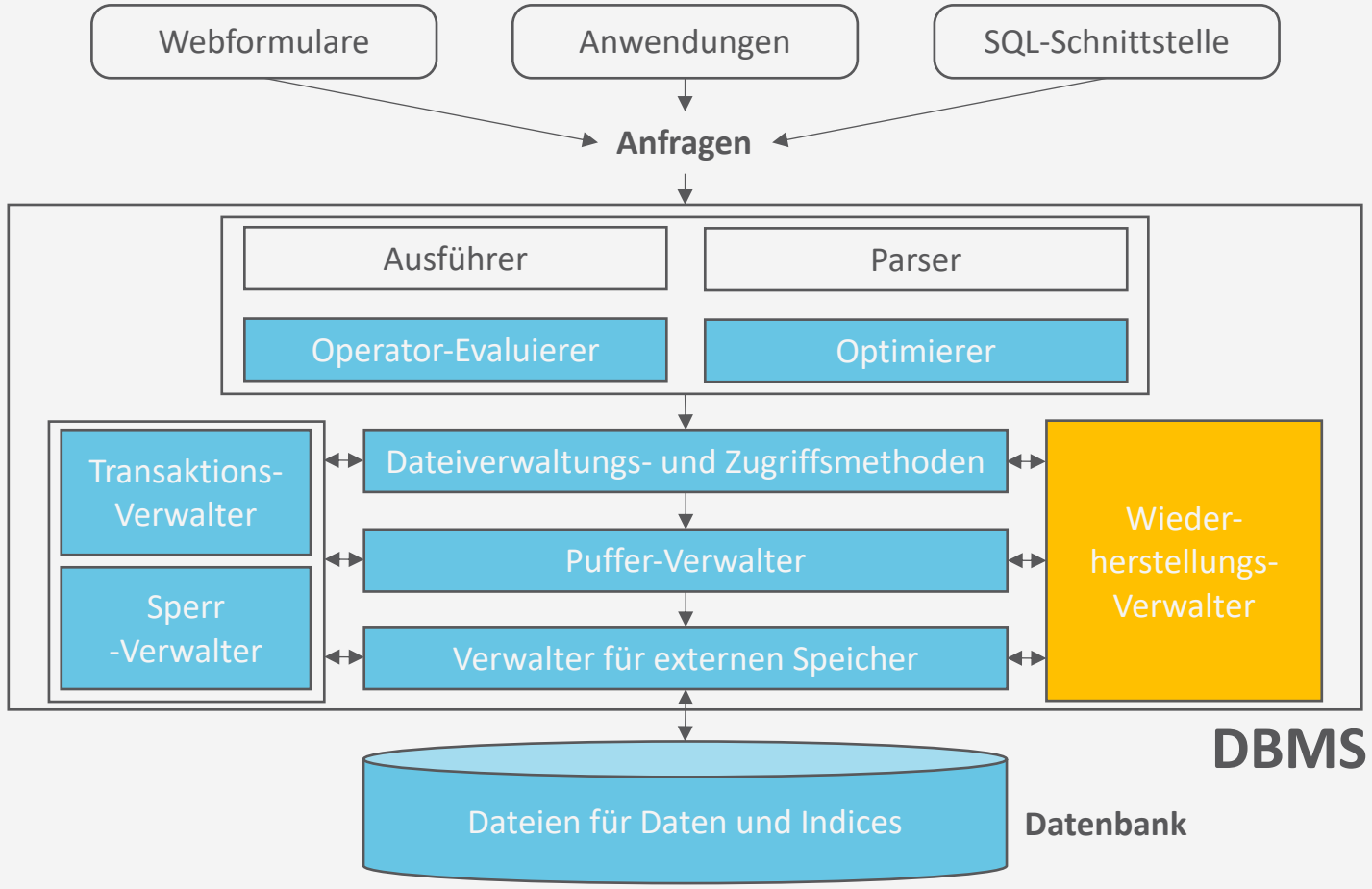
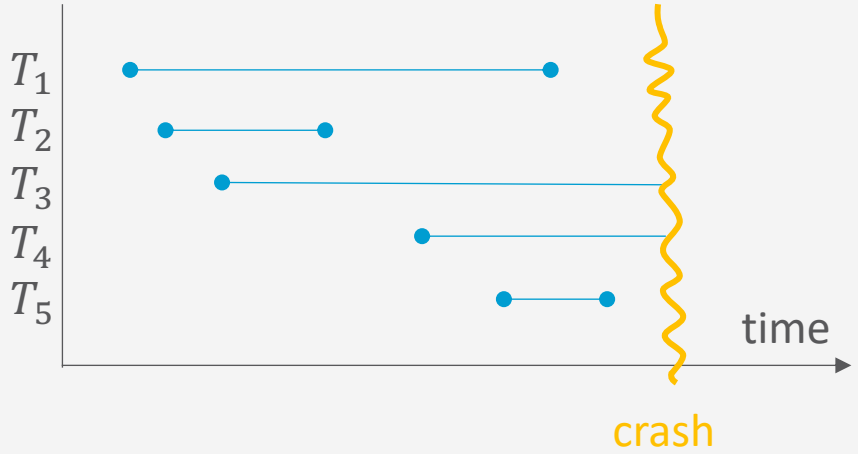
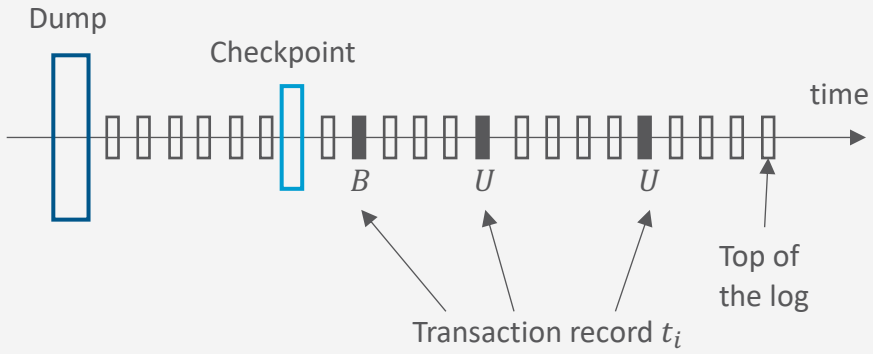
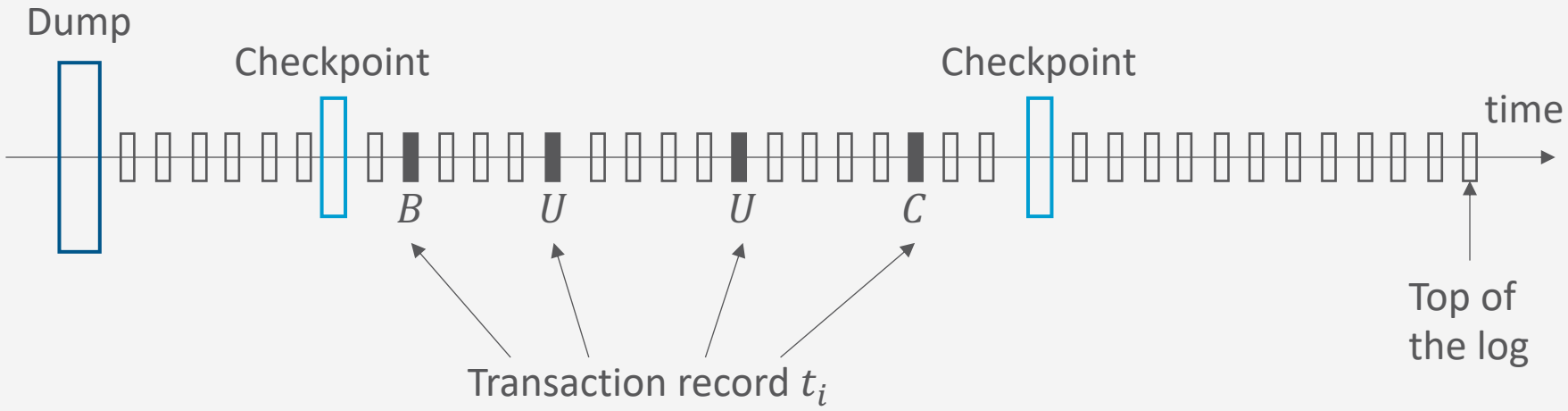
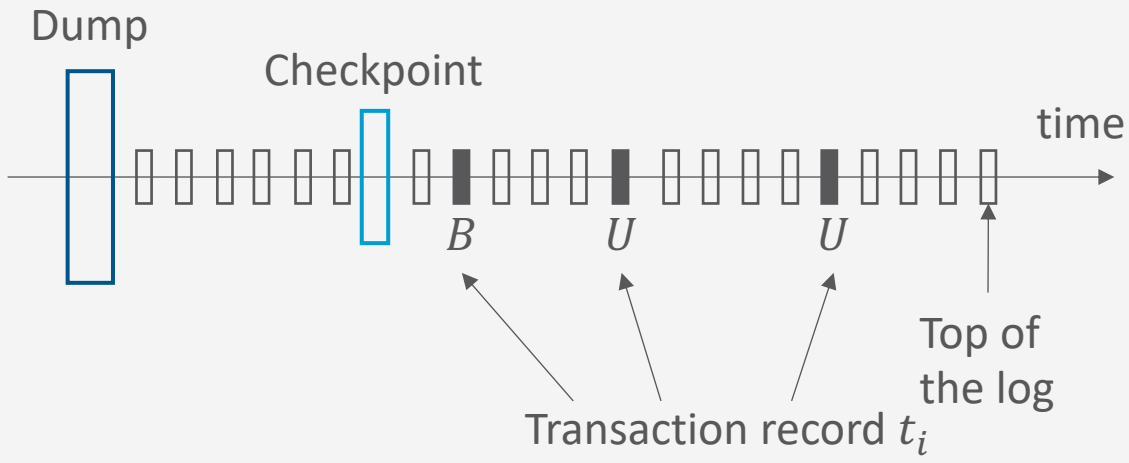
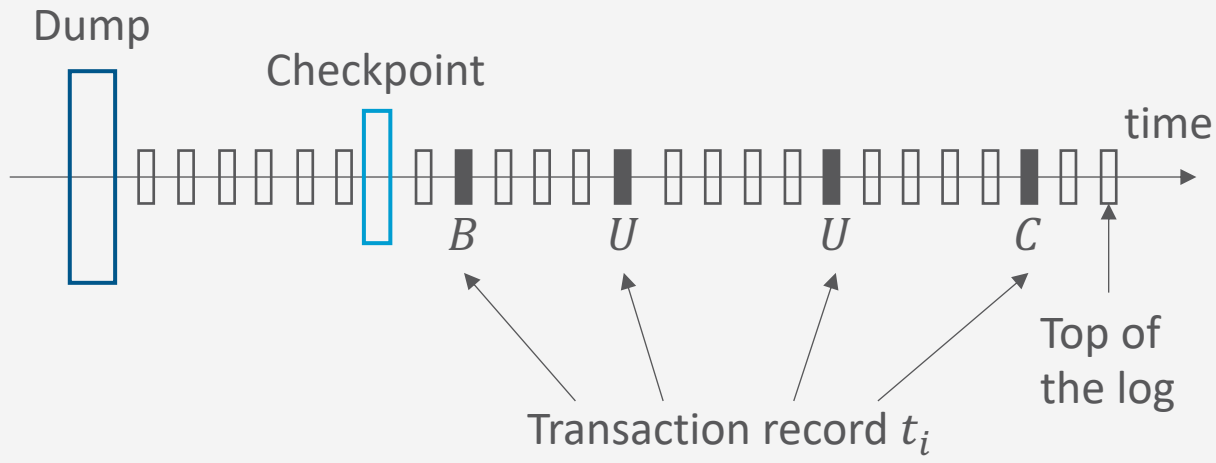
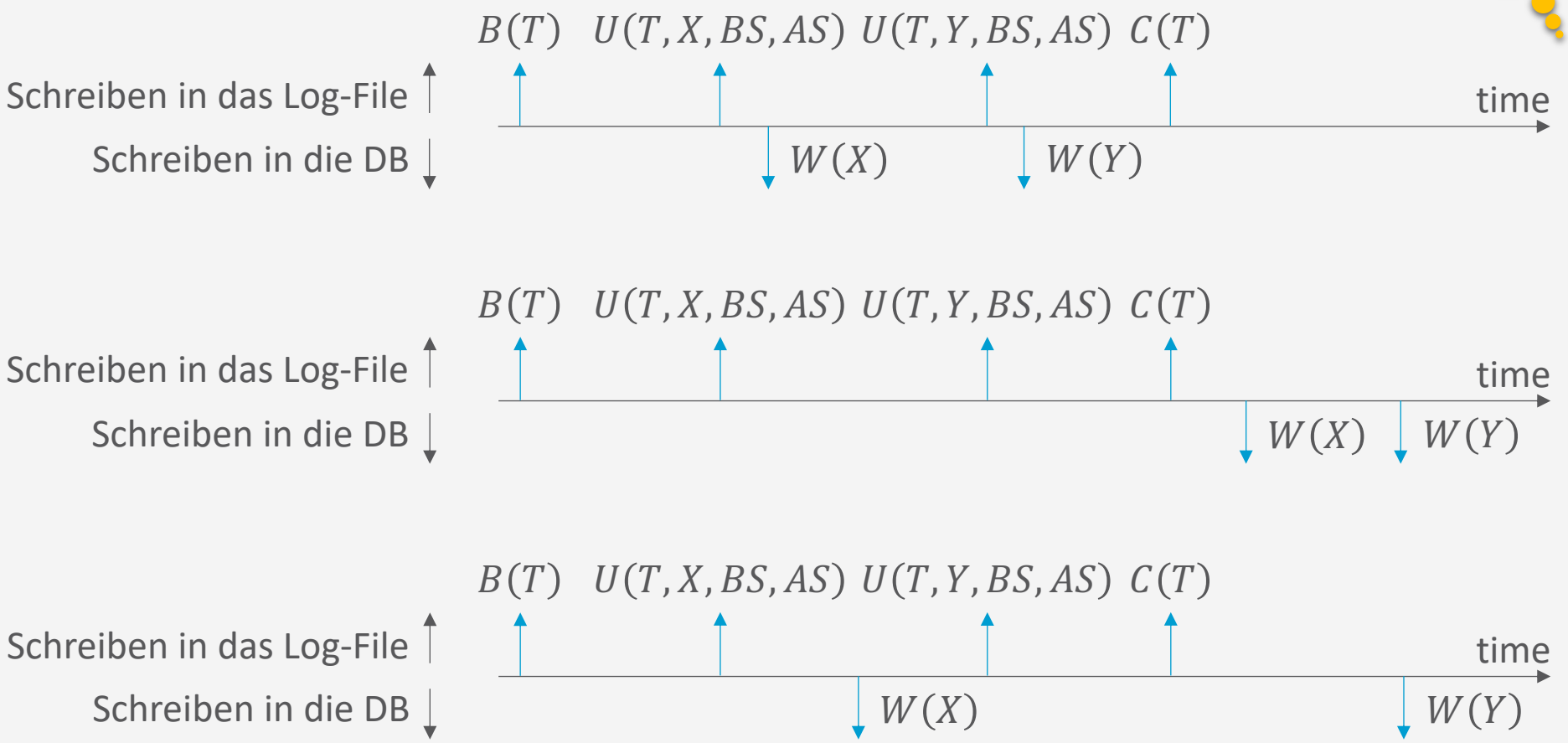
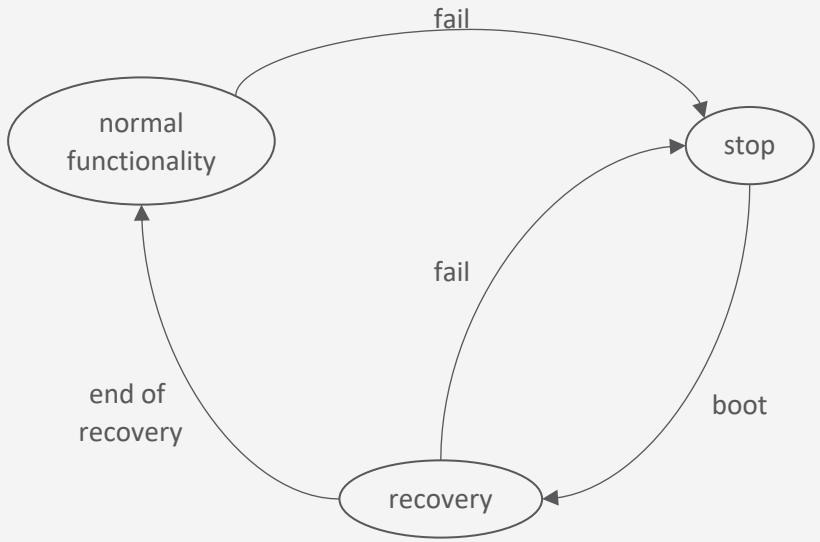
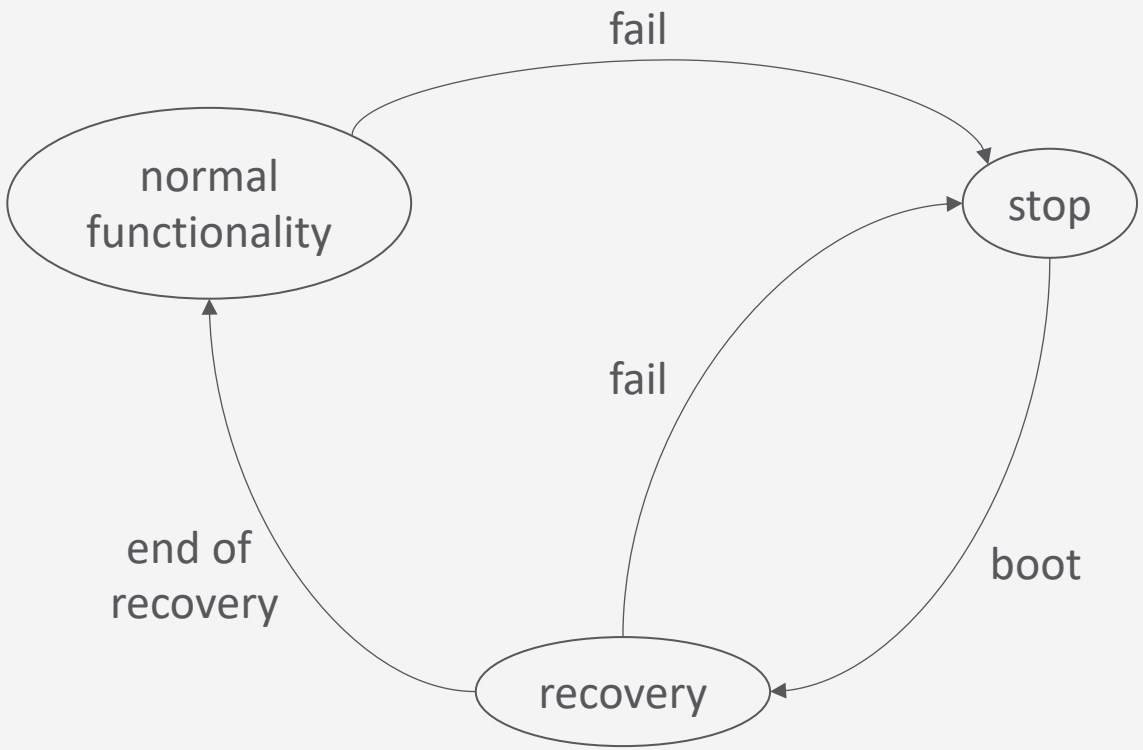
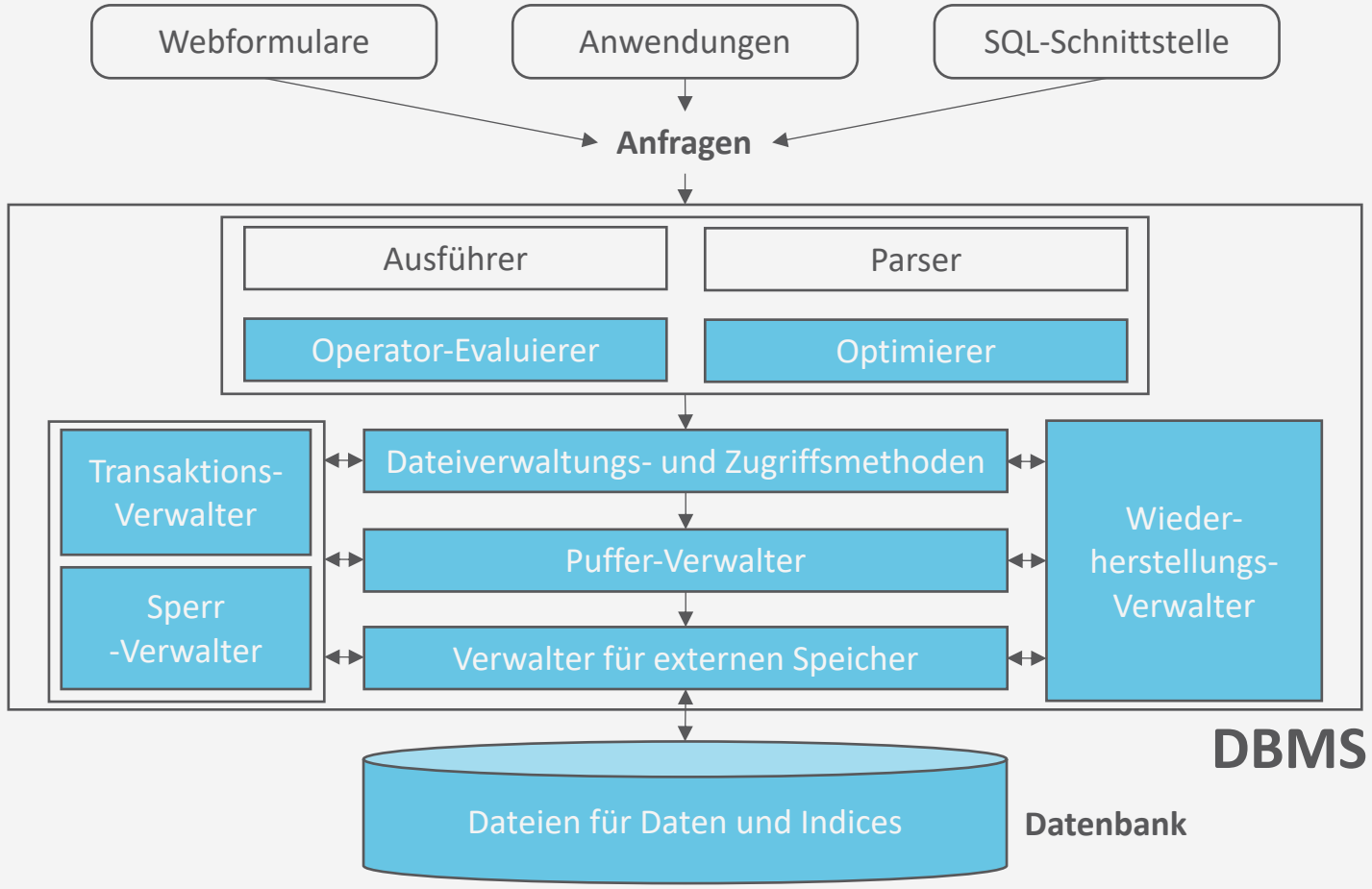
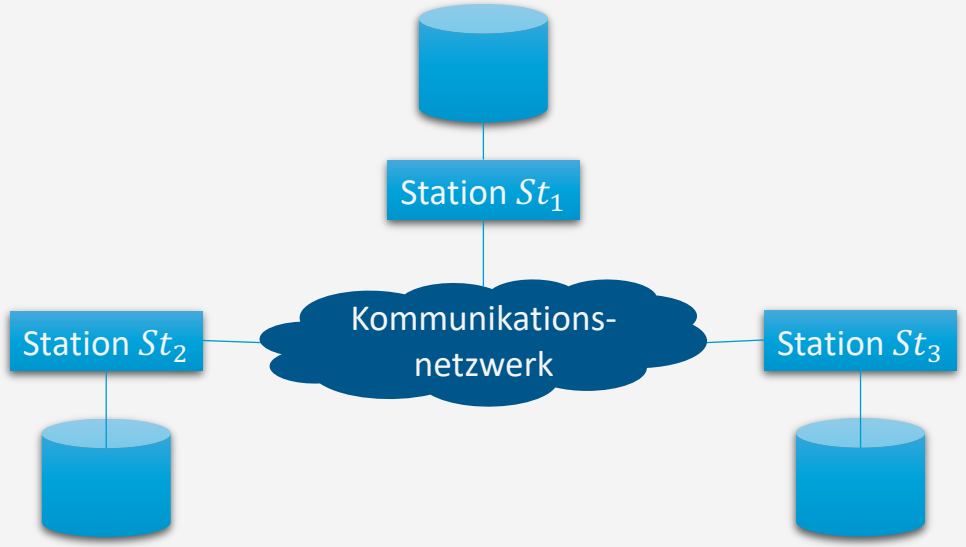
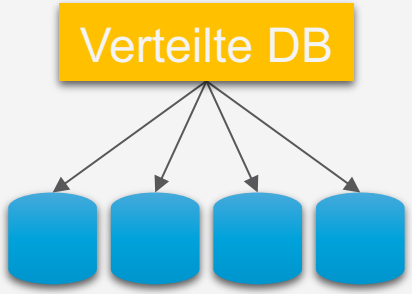
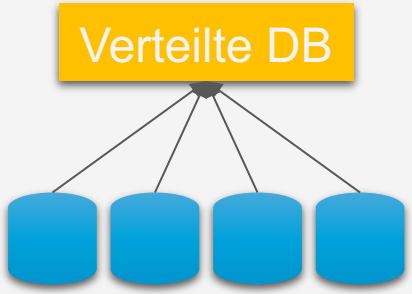
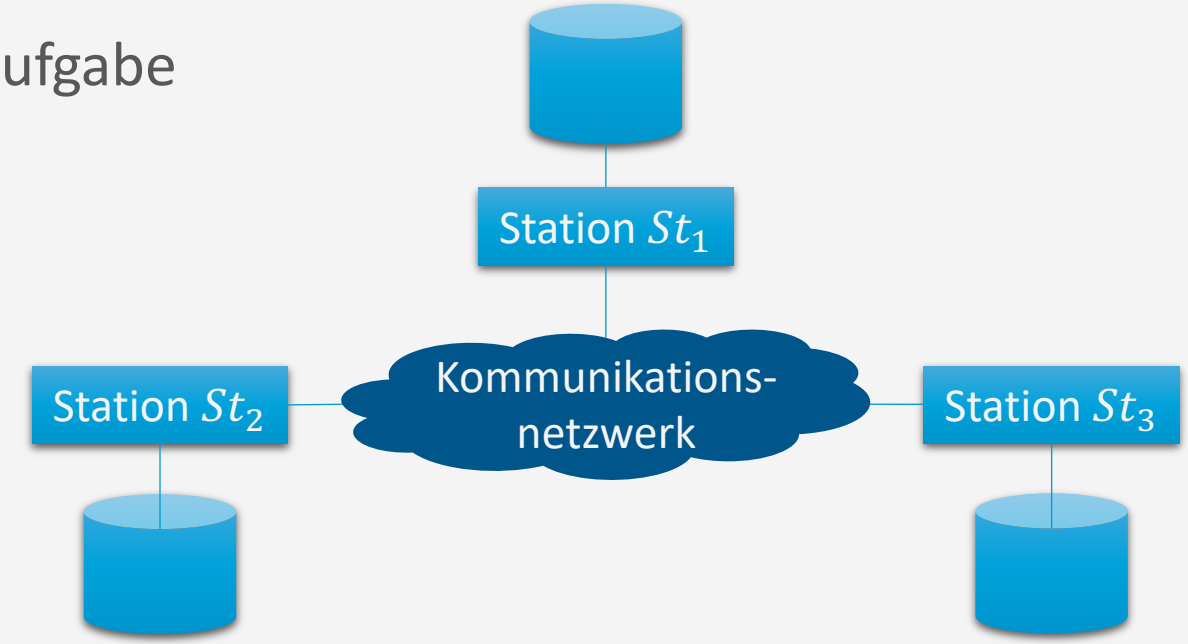
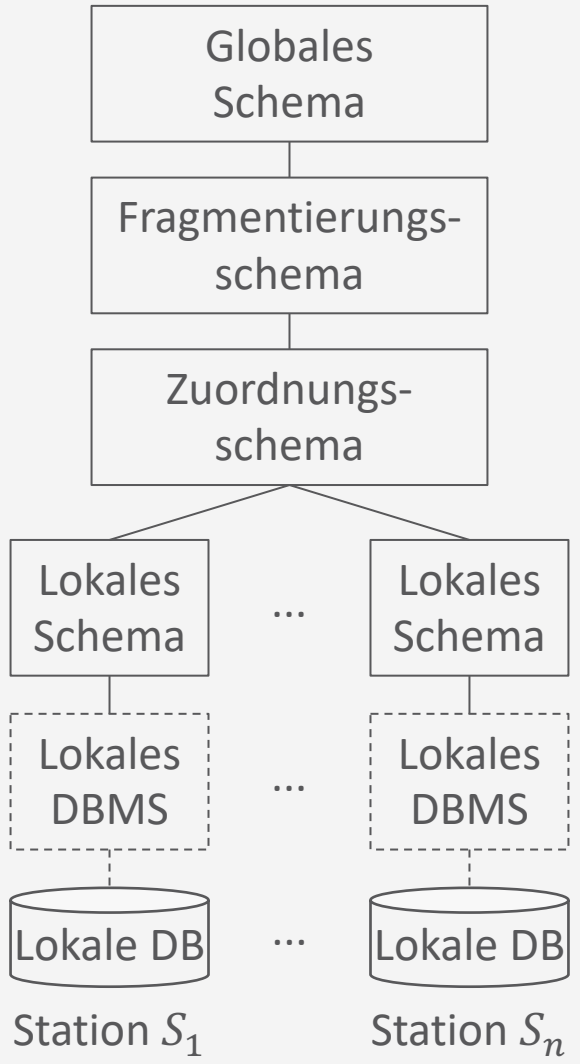
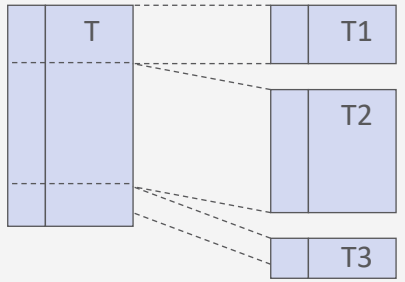
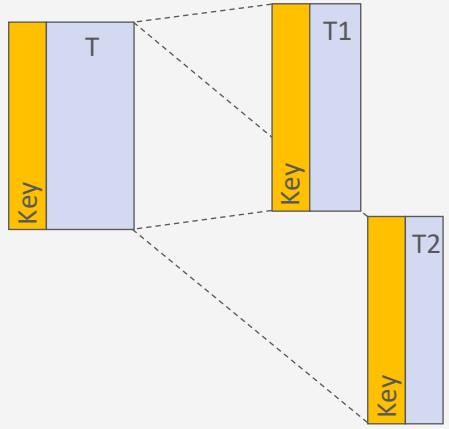
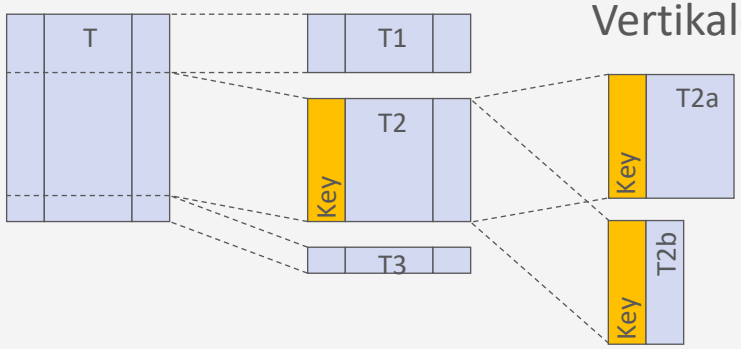
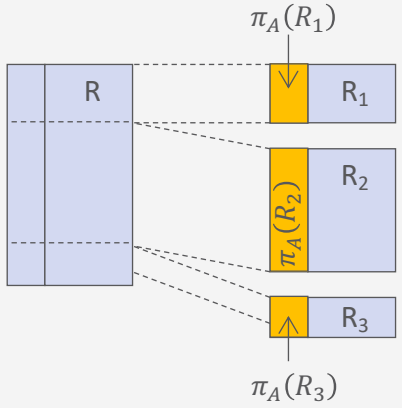
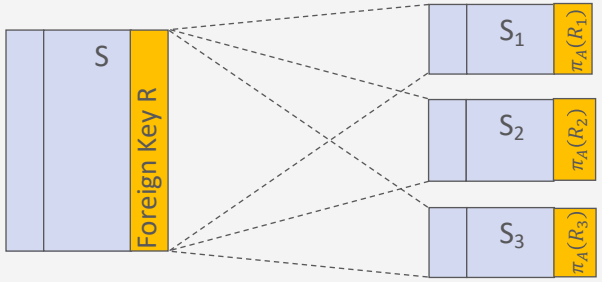
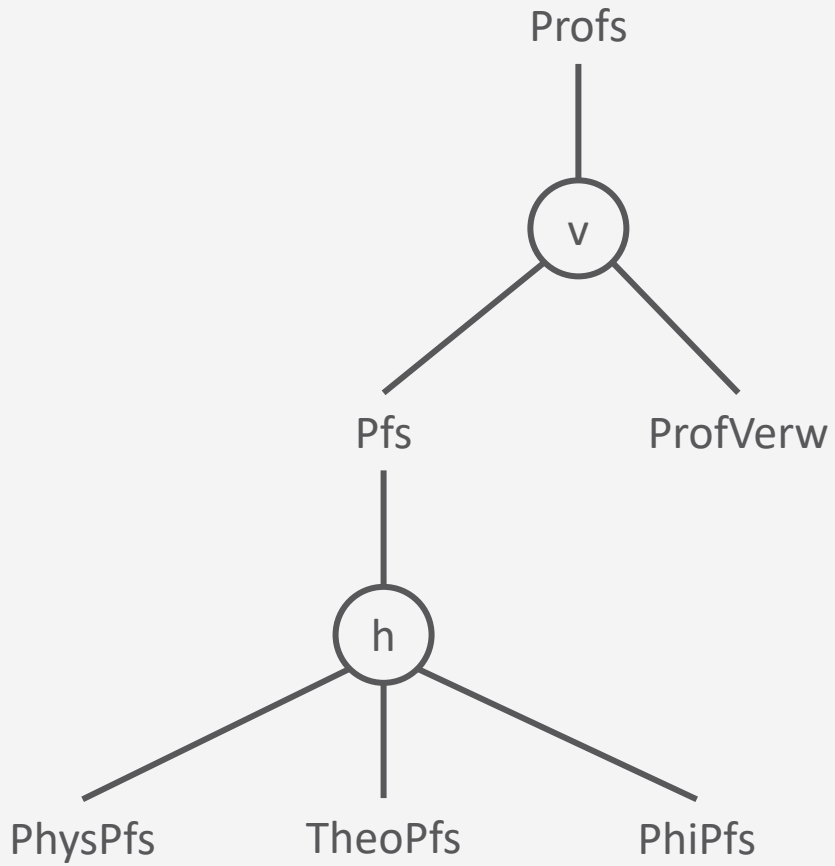
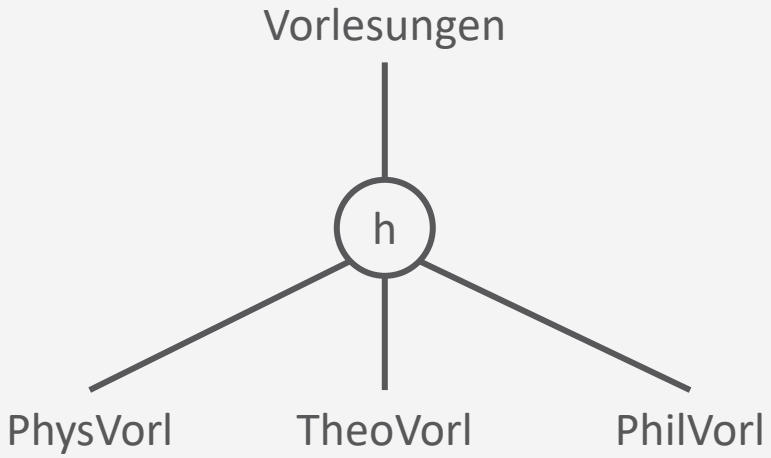
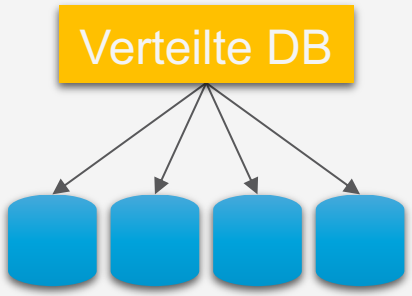
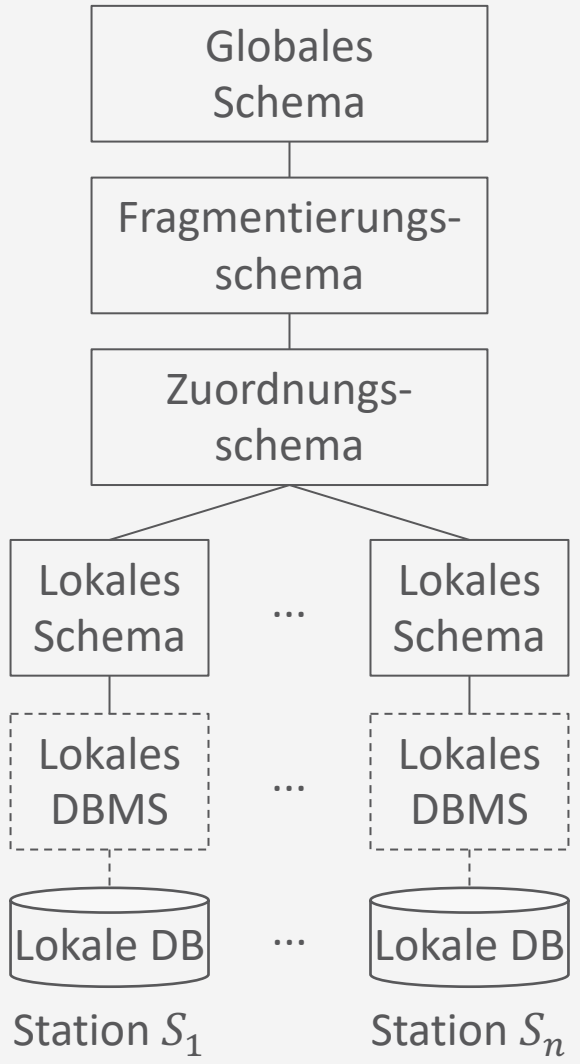
B⁺-Bäume und Sortierung

- Eine typische Situation mit $\langle k, rid \rangle$ Paaren in Blättern sieht so aus:



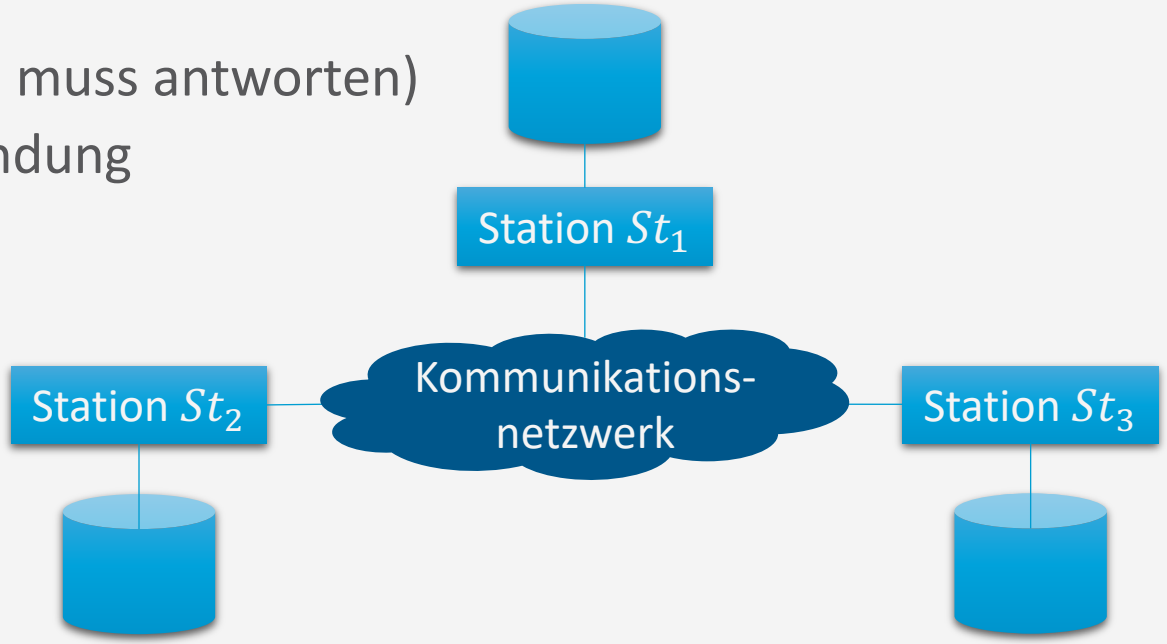
- Was passiert, wenn man Folgendes ausführt?
  - ```
    select *
    from Kunden
    where Plz between 8800 and 9099
    order by Plz;
    ```
    ```

Geclusterte B⁺-Bäume

- Wenn die Datei mit den Datensätzen sortiert und sequentiell gespeichert ist, erfolgt der Zugriff schneller



- Ein so organisierter Index heißt geclusterter Index
  - Sequentieller Zugriff während der Scan-Phase
  - Besonders für Bereichsanfragen geeignet

Warum macht man Indexe nicht immer geclustert?

Hash-basierte Indexierung

Effiziente Indexierung bei *Gleichheitsprädikaten*

Hash-basierte Indexierung

- B⁺-Bäume dominieren in Datenbanken
- Alternative: hash-basierte Indexierung für Gleichheitsprädikate
  - Aufteilung von möglichen Eingabewerten auf n Buckets mittels Hash-Funktion h
    - $h : \mathrm{dom}(key) \rightarrow [0, \ldots, n-1]$
    - Anzahl an Buckets n: vorher festzulegen
      - $n \ll$ Anzahl an Eingabewerten (Werte eines Attributs oder einer Kombination von Attributen)
    - Inhalt Bucket-Seite: Datensätze oder rid Liste
      - Wird gefüllt durch Anwendung der Hash-Funktion auf Datensatz und anschließende Einsortierung in Bucket
      - Wenn Seite voll: Kollisionslisten (overflow pages), lineares Sondieren o.ä.
  - Suche nach k: Suche nach k in Bucket $h(k)$



Statisches Hashing

- Relation $Student(Matrikelnummer, Name, Semester)$
- Hashfunktion $h(x) = x \bmod 3$, x Matrikelnummer
  - Anzahl an Buckets $n = 3$, daher $\bmod 3$
  - Kollisionsbehandlung mit Kollisionslisten
- Beispiel: (26830, ‘Aristoxenos, 8), (26120, ‘Fichte’, 10), (28106, ‘Carnap’, 3), … einfügen



Hash-basierte Indexierung

- Hash-Indexe eignen sich nur für Gleichheitsprädikate
  - Insbesondere für (lange) Zeichenketten
  - Beispielanfragen, für die sich ein Hash-Index lohnen könnte
    - ```
      select *
      from AbtStandort
      where Standort='Stafford';
      ```
    - ```
      select *
      from Kunden k, Auftraege a
      where k.Name='IBM Corp.'
            and k.KundenID=a.KundenID;
      ```



Dynamisches Hashing

- Statisches Hashing ineffizient bei unvorhersehbaren Daten und langen Kollisionslisten
- Problem: Wie groß soll die Anzahl n der Buckets sein?
  - n zu groß → schlechte Platznutzung und –Lokalität
  - n zu klein → viele Überlaufseiten, lange Listen
- Datenbanken verwenden daher dynamisches Hashen (dynamisch wachsende und schrumpfende Bildbereiche)
  - Erweiterbares Hashen
    (Vermeidung des Umkopierens)

Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R., Extendible Hashing - A Fast Access Method for Dynamic Files, *ACM Transactions on Database Systems* **4** (3): 315–344, **1979**

Vor- und Nachteile von Indices

- Zugriff auf Daten von $O(n)$ ungefähr auf $O(\log n)$
- Kosten der Indexierung aber nicht zu vernachlässigen
  - Nicht bei kleinen Tabellen
  - Nicht bei häufigen Update- oder Insert-Anweisungen
  - Nicht bei Spalten mit vielen Null-Werten
- Standardisierung nicht gegeben
  - Die meisten Implementierungen legen Indices für Schlüssel und Unique-Eigenschaften zur schnellen Überprüfung automatisch an

```
create [unique|fulltext|spatial] index index_name [index_type]
    on tbl_name (index_col_name,...) [index_type]

index_col_name:
    col_name [(length)] [asc | desc]

index_type:
    using {btree | hash}
```

Zwischenzusammenfassung

- Index-Sequentielle Zugriffsmethode (ISAM-Index)
  - Statisch, baum-basierte Indexstruktur
- B^+-Bäume
  - Die Datenbank-Indexstruktur
  - Auf linearer Ordnung basierend
  - Dynamisch
  - Kleine Baumhöhe für fokussierten Zugriff auf Bereiche
  - Geclusterte vs. ungeclusterte Indexe
    - Sequentieller Zugriff vs. Verwaltungsaufwand
- Hash-basierte Indexe
  - Gleichheitsprädikate
- Bei Einsatz von Indices Kosten vs. Nutzen abwegen

Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
  - Optimierer
- Transaktionsmanagement



Ausführungspläne

- Operator zur Umsetzung einer Operation in einer Anfrage
  - Jeder Planoperator führt zur Verarbeitung einer vollständigen Anfrage eine Unteraufgabe aus
  - Zu Ausführungsplänen zusammensetzen
  - Nicht immer eindeutige Ausführungspläne

```
select k.KundenID, k.Name,
       sum (a.total) as Einnahmen
from Kunden k, Auftraege a
where k.Plz between 8000 and 8999
       and k.KundenID = a.KundenID
group by k.KundenID
order by k.KundenID, k.Name;
```





Sortieren

Operator-Evaluierer

Effizientes Sortieren

- Häufiges Vorkommen von Sortieroperationen
  - SQL-Anweisung ACS, DESC
  - B^+-Baum bauen: einfach bei sortierter Eingabe
  - Duplikate-Eliminierung einfach
  - Andere Operatoren erfordern manchmal sortierte Eingaben (mehr dazu später)

Zwei-Wege-Mischsortieren (*Two-way Merge Sort*)

- Implementierung mit nur drei Pufferseiten möglich
  → Zwei-Wege-Mischsortieren
  - Sortierung von N Datensätzen in mehreren Durchgängen
  - Aufwand innerhalb von $N \log N$ bei N Datensätzen
    - Schneller bei mehr Pufferseiten, Parallelverarbeitung, ...; weitere Tricks anwendbar
    - Gewählte Implementierung abhängig von verfügbaren Ressourcen und Größe der Daten



Zwei-Wege-Mischsortieren

$N \in \{2^{k-1}+1, \ldots, 2^k\}$

Pass 0 (Input: $N = 2^k$ unsorted pages, output: 2^k sorted runs)

1. Read N pages, one page at a time
2. Sort page records in main memory
3. Write sorted pages to disk (each page results in a *run*)

▹ This pass requires one page of buffer space.

Pass 1 (Input: $N = 2^k$ sorted runs, output: 2^{k-1} sorted runs)

1. Open two runs r_1 and r_2 from Pass 0 at a time for reading
2. Merge records from r_1 and r_2, reading input page-by-page
3. Write new two-page run to disk (page-by-page)

▹ This pass requires *three pages* of buffer space.

⋮

Pass n (Input: $N = 2^{k-n+1}$ sorted runs, output: 2^{k-n} sorted runs)

1. Open two runs r_1 and r_2 from Pass $n-1$ for reading
2. Merge records from r_1 and r_2, reading input page-by-page
3. Write new two-page run to disk (page-by-page)

▹ This pass requires *three pages* of buffer space.



Zwei-Wege-Mischsortieren: I/O-Verhalten

- Um eine Datei mit N Seiten zu sortieren, werden in jedem Durchgang N Seiten gelesen und geschrieben $\rightarrow 2 \cdot N$ I/O-Operationen pro Durchgang
- Anzahl der Durchgänge: $1 + \lceil \log_2 N \rceil$



- Anzahl der I/O-Operationen: $2 \cdot N \cdot (1 + \lceil \log_2 N \rceil)$
  - Beispiel: Sortierung einer 8GB Datei bei einer Travelstar?
    - 8KB pro Seite, wahlfreier Zugriff mit Zugriffszeit $t = 14{,}33ms$ (siehe Folie 12)
    - Anzahl an Seiten $N = \frac{8GB}{8KB} = 10^6$
    - I/O-Operationen: $2 \cdot 10^6 \cdot (1 + \lceil 6 \cdot \log_2 10 \rceil) = 42 \cdot 10^6$
    - Zeit: $42 \cdot 10^6 \cdot 14{,}33ms \approx 7d$

Externes Mischsortieren

- Bisher freiwillig nur drei Seiten verwendet
- Wie kann ein großer Pufferbereich genutzt werden:
  B Seiten auf einmal bearbeiten
  - Mit B Seiten im Puffer können B Seiten eingelesen und im Hauptspeicher sortiert werden
  - Zwei wesentliche Stellgrößen
    - Reduktion der initialen Runs durch Verwendung des Pufferspeichers beim Sortieren im Hauptspeicher
    - Reduktion der Anzahl der Durchgänge durch Mischen von mehr als zwei Seiten

Externes Mischsortieren: Reduktion der initialen Runs

- Mit B Seiten im Puffer können B Seiten eingelesen und im Hauptspeicher sortiert werden
- Anzahl der I/O-Operationen: $2 \cdot N \cdot \left(1 + \left\lceil \log_2 \frac{N}{B} \right\rceil\right)$
  - Zugriffsmuster auf diese Ein-Ausgaben: Chunks von B Seiten sequenziell gelesen
  - Beispiel (Fortsetzung)
    - $B = 1000$
    - Durchgänge: $1 + \lceil \log_2 10^6/10^3 \rceil = 1 + \lceil \log_2 10^3 \rceil = 11$
    - $t_s + t_r$ bei 10 Merge-Durchgängen
      - Im ersten Durchgang t_s, t_r vernachlässigbar
      - Rest: $2 \cdot 10^6 \cdot 10 \cdot 14{,}33ms \approx 3d$
    - Transferzeit $t_{tr} = 2 \cdot 10^6 \cdot 11 \cdot 0{,}16ms \approx 1h$

Pass 0 (Input: N unsorted pages, output: $\lceil N/B \rceil$ sorted runs)
1. Read N pages, B pages at a time
2. Sort records in main memory
3. Write sorted pages to disk (resulting in $\lceil N/B \rceil$ *runs*)

▹ This pass uses B pages of buffer space.

Externes Mischsortieren: Reduktion der Anzahl der Durchgänge

- Mit B Seiten im Puffer können auch $B-1$ Seiten gemischt werden (eine Seite dient als Schreibpuffer)
- Anzahl der I/O-Operationen: $2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil\right)$
  - Zugriffsmuster auf diese Ein-Ausgaben: Random access
    - In den Mergephasen muss entschieden werden, welcher der minimalen Elemente auf Outputbuffer gehen.
  - Beispiel (Fortsetzung)
    - Durchgänge: $1 + \lceil \log_{999} 10^3 \rceil = 3$
    - 2 Merge-Durchgänge: $t_s + t_r$
      $= 2 \cdot 10^6 \cdot 2 \cdot 14{,}33ms \approx 16h$
    - Transferzeit t_{tr}
      $= 2 \cdot 10^6 \cdot 3 \cdot 0{,}16ms \approx 1min$

Pass n (Input: $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs, output: $\frac{\lceil N/B \rceil}{(B-1)^n}$ sorted runs)

1. Open $B-1$ runs $r_1, \ldots, r_{B-1}$ from Pass $n-1$ for reading
2. Merge records from $r_1, \ldots, r_{B-1}$, reading input page-by-page
3. Write new $B \cdot (B-1)^n$-page run to disk (page-by-page)

▹ This pass uses B pages of buffer space.

Externes Mischsortieren: Blockweise Ein-Ausgabe

- Man kann das I/O-Muster verbessern, in dem man Blöcke von b Seiten in den Mischphasen verarbeitet
  - Alloziere b Seiten für jede Eingabe (statt nur eine)
  - Reduktion der Ein-Ausgabe um Faktor b pro Seite
    - Anzahl der I/O-Operationen: $2 \cdot \left\lceil \frac{N}{b} \right\rceil \cdot \left(1 + \left\lceil \log_{\lfloor \frac{B}{b} \rfloor - 1} \left\lceil \frac{N}{\lfloor \frac{B}{b} \rfloor} \right\rceil \right\rceil\right)$
  - Preis: Reduzierte Einfächerung (was in mehr Durchgängen und damit in mehr I/O-Operationen resultiert)
  - Beispiel (Fortsetzung)
    - 63 Sektoren pro Spur, Track-to-Track-Suchzeit $t_{s,t2t} = 1ms$
    - Blockweises I/O mit $b = 32$
    - 1 Block mit 8KB benötigt 16 Sektoren
    - Ca. $10 \cdot 31.250$ Plattenzugriffe mit je $27.42ms \rightarrow 2{,}38h$

Externes Mischsortieren: Hauptspeicher als Ressource

- In der Praxis meist genügend Hauptspeicher vorhanden, so dass Dateien in einem Mischdurchgang sortiert werden kann (mit blockweisem I/O)
  - Beispiel (Fortsetzung)
    - Benötigter Speicher um mit einem Mischvorgang auszukommen: 256MB

Bisher nur IO-Zeit betrachtet

- Auswahl des nächsten Datensatzes für den Output unter $B-1$ (oder ${}^{B}/_{b}-1$) Eingabeläufen kann CPU-intensiv sein ($B-2$ Vergleiche)
- Beispiel: $B-1=4$, Ordnung: $<$







D. Knuth, The Art of Computer Programming,vol.3, **1973**

Tree of Losers (and Hidden Winners)

- Verwende Auswahlbaum zur Kostenreduktion
  - Reduktion der Vergleiche auf $\log_2(B-1)$ bzw. $\log_2({}^{B}/_{b}-1)$
  - Gewinner in Ausgabe-Puffer schreiben



$B-1=16$ Runs

D. Knuth, The Art of Computer Programming,vol.3, **1973**

Tree of Losers (and Hidden Winners): Nächstes Element

Aktualisierungsaufwand nur im Gewinner-Pfad

- Im Gewinner-Run: Nächste Zahl nach oben propagieren
  - Solange bis andere Zahl v kleiner ist, dann Zahl ersetzen und v nach oben propagieren
  - Bis an Wurzel angekommen → Kleinste Zahl schreiben und wieder von vorn



D. Knuth, The Art of Computer Programming,vol.3, **1973**

Join-Implementierung

Operator-Evaluierer

Verbundoperator (Join) $R \bowtie S$

- Join $R \bowtie_p S$ ist Abkürzung für Kreuzprodukt mit anschließender Selektion $\sigma_p(R \times S)$

- Daraus ergibt sich eine einfache Implementierung von $\bowtie_p$
  1. Enumeriere alle Datensätze aus $R \times S$
  2. Wähle die Datensätze, die p erfüllen

- Aber:
  Größe des Zwischenresultats aus Schritt 1: $|R| \cdot |S|$
  - Ineffizienz kann überwunden werden

Join-als-geschachtelte-Schleifen

- Einfache Implementierung des Joins
  - nl = nested loop
- Sei N_R und N_S die Seitenzahl von R und S
- Sei p_R und p_S die Anzahl der Datensätze pro Seite in R und S
- Anzahl an Plattenzugriffen

$$N_R + \underbrace{p_R \cdot N_R}_{\text{\#Tupel in } R} \cdot N_S$$

  - Jede der N_R Seiten muss geladen werden
  - Für jedes Tupel auf den N_R muss jede der N_S Seiten geladen werden

function $\mathrm{nljoin}(R, S, p)$
  $result \leftarrow \emptyset$
  for each record $r \in R$ **do**
    for each record $s \in S$ **do**
      if $\langle r, s \rangle$ satisfies p **then**
        Append $\langle r, s \rangle$ to $result$
  return $result$

Join-als-geschachtelte-Schleifen

- Geringer Speicherbedarf
  - Nur drei Seiten nötig
    - Zwei Seiten fürs Lesen von R und S
    - Eine Seite um das Ergebnis zu schreiben

function $\mathrm{nljoin}(R, S, p)$
  $result \leftarrow \emptyset$
  for each record $r \in R$ **do**
    for each record $s \in S$ **do**
      if $\langle r, s \rangle$ satisfies p **then**
        Append $\langle r, s \rangle$ to $result$
  return $result$

- I/O-Verhalten: Sehr viele wahlfreie Zugriffe
  - Annahme $p_R = p_S = 100, N_R = 1000, N_S = 500$:

$$N_R + p_R \cdot N_R \cdot N_S = 1000 + 5 \cdot 10^7$$

  - Mit einer Zugriffszeit von 10ms für jede Seite dauert der Vorgang 140 Stunden
    - Wenn $|S| < |R|$: Vertauschen von R und S verbessert die Situation nur marginal
    - Abwechselndes seitenweises Lesen bedingt volle Plattenlatenz, obwohl beide Relationen in sequenzieller Ordnung verarbeitet werden.

Blockweiser Join mit Schleifen

- Einsparung von Kosten durch wahlfreien Zugriff durch blockweises Lesen von R und S mit b_R und b_S vielen Seiten
- Braucht mehr Seiten als $\text{nljoin}(R, S, p)$

- Plattenzugriffe
  - R wird (einmal) vollständig gelesen, aber mit nur $\left\lceil \frac{N_R}{b_R} \right\rceil$ Plattenzugriffen
  - S wird nur $\left\lceil \frac{N_R}{b_R} \right\rceil$ mal gelesen, mit $\left\lceil \frac{N_R}{b_R} \right\rceil \cdot \left\lceil \frac{N_S}{b_S} \right\rceil$ Plattenzugriffen

function $\text{block\_nljoin}(R, S, p)$
  $result \leftarrow \emptyset$
  for each b_R-sized block in R **do**
    for each b_S-sized block in S **do**
      Find matches in current R and S block
        and append them to $result$
  return $result$

Join im Hauptspeicher ausführbar

Wahl von b_R und b_S

- Pufferbereich mit
  - $B = 100$ Rahmen
  - $N_R = 1000$
  - $N_S = 500$



Performanz des Hauptspeicher-Joins

- Anweisung im Inneren bedingt einen Hauptspeicherverbund zwischen Blöcken aus R und S
- Aufbau einer Hashtabelle kann den Verbund erheblich beschleunigen
  - Funktioniert nur für Gleichheitsprädikate im Join

Warum Hashtabelle für R Block und nicht S Block?

function block_nljoin(R, S, p)
  $result \leftarrow \emptyset$
  for each b_R-sized block in R **do**
    for each b_S-sized block in S **do**
      Find matches in current R and S block
        and append them to $result$
  return $result$

function block_nljoin$'(R, S, p)$
  $result \leftarrow \emptyset$
  for each b_R-sized block in R **do**
    Build an in-memory hash table H for current R block
    for each b_S-sized block in S **do**
      for each record s in current S block **do**
        Probe H and append matching $\langle r, s \rangle$ tuples to $result$
  return $result$

Indexbasierte Verbunde $R \bowtie S$

- Verwendung eines vorhandenen Index für die innere Relation S
  - Ggf. innere und äußere Relation vertauschen (nur Index für R vorhanden)
- Index muss verträglich mit der Join-Bedingung sein
  - Join-Attribute heißen dann *sargable*
    - (SARG = Search ARGument)
  - Hash-Index (nur für Gleichheitsprädikate) oder auch B^+-Baum
- Häufiger Join → Passenden Index aufbauen

Suchschlüssel für Suche im Index

function index_nljoin(R, S, p)
  $result \leftarrow \emptyset$
  for each record $r \in R$ **do**
    Probe index using r and
      append all matching tuples to $result$
  return $result$

$R = \bowtie_{k.KundenID=a.KundenID}$

$\sigma_{8000 \leq Plz \leq 8999}$ a

Sortier-Merge-Join

- Join-Berechnung einfach, wenn Relationen bzgl. Join-Attribut(en) sortiert
  - Kombiniere Eingabetabellen ähnlich wie beim Merge-Sort
    - Nur für Gleichheitsprädikate

A	B
‘foo’	1
‘foo’	2
‘bar’	2
‘baz’	2
‘baf’	4

$\bowtie_{B=C}$

C	D
1	false
2	true
2	false
3	true

function merge_join$(R, S, \alpha = \beta)$ ▹ α, β join columns in R, S
  $result \leftarrow \emptyset$
  $r \leftarrow$ position of first tuple in R
  $s \leftarrow$ position of first tuple in S
  while $r \neq$ **eof** and $s \neq$ **eof do**
    while $r.\alpha < s.\beta$ **do**
      Advance r
    while $r.\alpha > s.\beta$ **do**
      Advance s
    $s' \leftarrow s$ ▹ Remember current position in s
    while $r.\alpha = s'.\beta$ **do** ▹ All R tuples with same α value
      $s \leftarrow s'$ ▹ Rewind s to s'
      while $r.\alpha = s.\beta$ **do** ▹ All S tuples with same β values
        Append $\langle r, s \rangle$ to $result$
        Advance s
      Advance r
  return $result$

Merge-Join: I/O-Verhalten

- Wenn beide Eingaben sortiert und keine außergewöhnlich langen Sequenzen mit identischen Schlüsselwerten vorhanden, dann ist der I/O-Aufwand

  $N_R + N_S$ (das ist dann optimal)

  - Durch blockweises I/O treten fast immer sequenzielle Lesevorgänge auf
- Vorher sortieren kann sich also für Join-Berechnung auszahlen
- Ausgabe weiterhin sortiert
  - Wenn später eine Sortierung der Ausgabe gefordert wird, lohnt sich die vorherige Sortierung noch mehr
  - Zudem weniger Festplattentransfers mit Merge-Join im Vergleich zu erst eine Art von Join ausführen und dann sortieren



Hash-Join

- Sortierung bringt korrespondierende Tupel in eine räumliche Nähe, so dass eine effiziente Verarbeitung möglich ist
- Ähnlicher Effekt erreichbar mit Hash-Verfahren

- Zerlege R und S in Teilrelationen $R_1, \dots, R_n$ und $S_1, \dots, S_n$ mit der gleichen Hashfunktion (angewendet auf die Join-Attribute)
  - $R_i \bowtie S_j = \emptyset$ für alle $i \neq j$



Hash-Join

- Mittels Hashfunktion werden die Tupel aus R und S partitioniert
  - Durch Partitionierung werden kleine Relationen R_i und S_i geschaffen
  - Korrespondierende Datensätze kommen garantiert in korrespondierende Partitionen der Relationen
- Es muss $R_i \bowtie S_i$ (für alle i) berechnet werden (einfacher)
- Die Anzahl der Partitionen n (d.h. die Hashfunktion) sollte mit Bedacht gewählt werden, so dass $R_i \bowtie S_i$ als Hauptspeicher-Join berechnet werden kann
  - Solange mit Hashfunktionen partitionieren bis die Partitionen der kleineren Relation in den Hauptspeicher passen
  - Partitionen der größeren Relation müssen nicht in den Hauptspeicher passen, werden dann block-weise geladen, wenn $R_i \bowtie S_i$ berechnet wird

Hash-Join-Algorithmus

- I/O-Aufwand, wenn $|R \bowtie S|$ klein:

$$3 \cdot (N_R + N_S)$$

  - Lesen und Schreiben beider Relationen für Partitionierung + Lesen beider Relationen für Join

function hash_join$(R, S, \alpha = \beta)$
    $result \leftarrow \emptyset$
    for each record $r \in R$ **do**
        Append r to partition $R_{h(r.\alpha)}$
    for each record $s \in S$ **do**
        Append s to partition $S_{h(s.\beta)}$
    for each partition $i \in 1, \ldots, n$ do
        Build hash table H for R_i using hash function h'
        for each block in S_i **do**
            for each record s in current S_i block **do**
                Probe H and append matching tuples to $result$
    return $result$



Gruppierung + UNIQUE-Behandlung

Operator-Evaluierer

Gruppierung und Duplikate-Elimination

- Herausforderung: Finde *identische* Datensätze in einer Datei
  - Duplikate-Elimination: Identische Datensätze bzgl. aller Attribute zur Eliminierung
  - Gruppierung: Identisch bzgl. Gruppierungsattribut(e) zur Gruppierung

- Umsetzung der Duplikate-Elimination oder Gruppierung mit Hash-Join oder Sortierung
  - Siehe vorherige Folien

$$\pi_{KundenID,Name,Einnahmen}(S)$$

$$\sigma_{8000 \leq Plz \leq 8999}$$

Selektion und Projektion

Anfrageverarbeitung

Andere Anfrage-Operatoren

- Projektion π
  - Implementierung durch
    a. Entfernen nicht benötigter Spalten
    b. Eliminierung von Duplikaten
  - Die Implementierung von
    - a. Bedingt das Ablaufen (scan) aller Datensätze in der Datei
    - b. Siehe vorherigen Abschnitt zu Duplikate-Eliminierung
  - Systeme vermeiden b. sofern möglich
- Selektion σ
  - Ablaufen (scan) aller Datensätze
  - Eventuell Sortierung ausnutzen oder Index verwenden

Pipelining

Anfrageverarbeitung

Organisation der Operator-Evaluierung

- Bisher gehen wir davon aus, dass Operatoren ganze Dateien verarbeiten



- Erzeugt offensichtlich viel I/O
- Außerdem: lange Antwortzeiten
  - Ein Operator kann nicht anfangen, solange nicht seine Eingaben vollständig bestimmt sind (materialisiert sind)
  - Operatoren werden nacheinander ausgeführt

Pipeline-orientierte Verarbeitung

- Alternativ könnte jeder Operator seine Ergebnisse direkt an den nachfolgenden senden, ohne die Ergebnisse erst auf die Platte zu schreiben
- Ergebnisse werden so früh wie möglich weitergereicht und verarbeitet (Pipeline-Prinzip)
- Granularität ist bedeutsam:
  - Kleinere Brocken reduzieren Antwortzeit des Systems
  - Größere Brocken erhöhen Effektivität von Instruktions-Cachespeichern
  - In der Praxis meist tupelweises Verarbeiten verwendet

- Siehe auch Gebiet der Stromverarbeitung

Auswirkungen auf die Performanz

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Ausführungsplan A:



- Ausführungsplan B:



- Ermöglicht
  - Tupelweises Verarbeiten der Relation k
  - Sofortiges Weiterleiten nach der Selektion
  - Kombiniert mit tupelweisem Verarbeiten der Relationen a und p

Blockierende Operatoren

- Pipelining reduziert Speicheranforderungen und Antwortzeiten, da jeder Datensatz gleich weitergeleitet
- Funktioniert so nicht für alle Operatoren

  - Misch-Sortierung
  - Gruppierung, Duplikate-Elimination, Max/Min über einer unsortierten Eingabe
- Solche Operatoren nennt man blockierend
- Blockierende Operatoren konsumieren die gesamte Eingabe in einem Rutsch, bevor die Ausgabe erzeugt werden kann
  - Daten auf Festplatte zwischengespeichert

Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
    - Sortieren: externes Merge-Sort, Tree of Losers
    - Join-Verarbeitung: blockweise, index-basiert, Sortier-Merge, hash-basiert
    - Gruppierung, Duplikate-Eliminierung, Selektion, Projektion möglicherweise unterstützt durch Indices / Sortierung
    - Pipelining
  - Optimierer



Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

Anfrageoptimierung

Bezogen auf die Ausführungszeit können die Unterschiede „Sekunden vs. Tage“ bedeuten.

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```



- Neben der Anordnung in den Ausführungsplänen gibt es weitere Entscheidungen, die auf die Anfragebeantwortung Auswirkung haben
  - Welche Implementation eines Join-Operators?
  - Welche Parameter für Blockgrößen, Pufferallokation, ...
  - Automatisch einen Index aufsetzen?
- Aufgabe, den besten Ausführungsplan zu finden: Heilige Gral der DB-Implementierung

Optimierung

- Optimierungen können unabhängig von den Daten erfolgen: Rewriting
  - Selektionsprädikate vereinfachen / früh anwenden
  - Geschachtelte Anfragen entschachteln bzw. Joins explizit machen
  - Vermeide Duplikate-Elimination, wenn möglich
- Datenabhängige Optimierung: Optimiser
  - Kostenbasiert auf Basis der Daten bzw. statistisch relevanter Größen der DB

- Hier nicht näher besprochen
  - Minimierung einer Anfrage durch Elimination einer Unteranfrage
  - Elimination eines teuren Operators
  - Bestimmung relevanter Tabellen



M. Benedikt: How Can Reasoners Simplify Database Querying (And Why Haven't They Done It Yet)? (Invited Talk at PODS 2018.)

Prädikatsvereinfachung (Rewriting)

- Beispiel: Schreibe
  - ```
    select *
    from Einzelposten p
    where p.Steuern * 100 < 5;
    ```
    Non-Sargable

  um in
  - ```
    select *
    from Einzelposten p
    where p.Steuern < 0.05;
    ```
    Sargable

- Prädikatsvereinfachung ermöglicht Verwendung von Indices und vereinfacht die Erkennung von effizienten Join-Implementierungen

Geschachtelte Anfragen

- SQL bietet viele Wege, geschachtelte Anfrage zu schreiben
  - Unkorrelierte Unteranfragen
    → nur einmal auswerten

```
select *
from Auftraege
where Kunden_ID in
   (select Kunden_ID
    from Kunden
    where Name = 'IBM Corp.');
```

Korrelierte Unteranfragen
→ für jedes Tupel auswerten

```
select *
from Auftraege a
where Kunden_ID in
   (select Kunden_ID
    from Kunden k
    where k.Kontostand = a.Gesamtpreis);
```

- Meist sind Unteranfragen nur syntaktische Varianten von Joins
- Rewriting: Joins explizit machen für Join-Order-Optimierung

Won Kim. On Optimizing an SQL-like Nested Query. ACM TODS, vol. 7, no. 3, **1982**

Zusätzliche Verbundprädikate

- Implizite Verbundprädikate wie in
  - ```
    select *
    from A, B, C
    where A.a = B.b and B.b = C.c;
    ```

  können explizit gemacht werden
  - ```
    select *
    from A, B, C
    where A.a = B.b and B.b = C.c and A.a = C.c
    ```

- Hierdurch werden Pläne möglich wie $(A \bowtie C) \bowtie B$

Optimiser: Kostenbasierte Optimierung

Anfragenoptimierer

Abschätzung der Ergebnisgröße

- Betrachte Anfrageblock für SELECT-FROM-WHERE-Anfrage Q
  - $R_1, \ldots, R_n$ Eingabetabellen im FROM
  - Mit einer Projektion $\pi_{projList}$ im SELECT
  - Mit einer Selektion $\sigma_{predicateList}$ im WHERE



- Abschätzung der Ergebnisgröße von Q durch

$$|Q| = |R_1| \cdot \cdots \cdot |R_n| \cdot \text{sel}(predicateList)$$

  - wobei
    - $|R_1|, \ldots, |R_n|$ Größe der Eingabetabellen
    - $\text{sel}(predicateList)$ **Selektivität** von σ

Tabellengrößen

- Größe einer Tabelle über den Systemkatalog verfügbar
  - Hier IBM DB2, Anzeige von
    - Tabellenname
    - Kardinalität
    - Anzahl der Seiten im Speicher
  - Vor Ausführung der Anfrage verfügbar
    - Bei DB-Änderungen wird Tabelle aktualisiert

```
db2 => select TABNAME, CARD, NPAGES
db2 (cont.) => from SYSCAT.TABLES
db2 (cont.) => where TABSCHEMA = 'TPCH';
TABNAME         CARD                  NPAGES
--------------- --------------------- --------------------
ORDERS          1500000               44331
CUSTOMER        150000                6747
NATION          25                    2
REGION          5                     1
PART            200000                7578
SUPPLIER        10000                 406
PARTSUPP        800000                31679
LINEITEM        6001215               207888
8 record(s) selected.
```

Selektivität

- Grobe Abschätzung durch Induktion über die Struktur des Anfrageblocks
  - $V(A,R)$ = Anzahl verschiedener Werte von Attribut (Spalte) A in Relation R · · · Woher?
  - $R.A = value$: $\text{sel}(\cdot) = \begin{cases} \frac{1}{V(A,R)} & \text{falls } \exists V(A,R) \\ \frac{1}{10} & \text{sonst} \end{cases}$ · · · Warum $\frac{1}{10}$?
  - $R.A = S.B$: $\text{sel}(\cdot) = \begin{cases} \frac{1}{\max\{V(A,R),V(B,S)\}} & \text{falls } \exists V(A,R) \wedge \exists V(B,S) \\ \frac{1}{V(Attr,Rel)} & \text{falls entweder } \exists V(A,R) \text{ oder } \exists V(B,S) \\ \frac{1}{10} & \text{sonst} \end{cases}$
  - $p_1 \wedge p_2$: $\text{sel}(\cdot) = \text{sel}(p_1) \cdot \text{sel}(p_2)$
  - $p_1 \vee p_2$: $\text{sel}(\cdot) = \text{sel}(p_1) + \text{sel}(p_2) - \text{sel}(p_1) \cdot \text{sel}(p_2)$

Verbesserung der Selektivitätsabschätzung

- Annahmen
  - Gleichverteilung der Datenwerte in einer Spalte
  - Unabhängigkeit zwischen einzelnen Prädikaten

- Annahmen nicht immer gerechtfertigt

- Sammlung von Datenstatistiken (offline)
  - Speicherung im Systemkatalog
    - IBM DB2: RUNSTATS ON TABLE
  - Meistverwendet: Histogramme

Histogramme

- Mit Histogrammen können echte Verteilungen von Werten einer Spalte A approximiert werden
  - Wenn Domäne von A endlich:
    - Aufteilung nach möglichen Werten x mit eingeschränkter Beziehung zwischen den Werten
  - Wenn Domäne von A gegeben durch Zahlen:
    - Aufteilung in angrenzende Intervalle mit Grenzwerten x_i
- Sammle statistische Parameter für jedes Intervall, z.B.
  1. Anzahl Zeilen t mit $x_i - 1 < t.A \leq x_i$ bzw. mit $t.A = x$
  2. Anzahl verschiedener Werte von A im Intervall $(x_i - 1, x_i]$, absolut oder relativ



Histogramme

- DB2: SYSCAT.COLDIST enthält Informationen wie
  - DB2: TYPE='Q' Quantile (cumulative), TYPE='F' Frequency
  - k-häufigste Werte (und deren Anzahl)
  - Auch Anzahl der verschiedenen Werte pro Histogramm-Rasterplatz anfragbar

- Tatsächlich können Histogramme auch absichtlich gesetzt werden, um den Optimierer zu beeinflussen

```
select SEQNO, COLVALUE, VALCOUNT
from SYSCAT.COLDIST
where TABNAME='LINEITEM'
        and COLNAME='L_EXTPRICE'
        and TYPE='Q';
SEQNO COLVALUE                 VALCOUNT
----- ---------------------- --------
1     +0000000000996.01 3001
2     +0000000004513.26 315064
3     +0000000007367.60 633128
4     +0000000011861.82 948192
5     +0000000015921.28 1263256
6     +0000000019922.76 1578320
7     +0000000024103.20 1896384
8     +0000000027733.58 2211448
9     +0000000031961.80 2526512
10    +0000000035584.72 2841576
11    +0000000039772.92 3159640
12    +0000000043395.75 3474704
13    +0000000047013.98 3789768
```

Bessere Abschätzung der Selektivität

- $MCV(A,R) = k$-häufigsten (top-k) Werte in Spalte A einer Tabelle R
- $MCF(A,R) =$ Häufigkeiten dieser Werte
  - $MCF(A,R)[value]$: Zugriff auf Häufigkeit von $value$, falls $value \in MCV(A,R)$
- Verbesserte Abschätzung für $R.A = value$
- $R.A = value$:

$$\text{sel}(\cdot) = \begin{cases} \frac{1}{MCF(A,R)[value]} & \text{falls } value \in MCV(A,R) \\ \frac{1}{V(A,R)} & \text{falls } value \notin MCV(A,R), \text{ aber } \exists V(A,R) \\ \frac{1}{10} & \text{sonst} \end{cases}$$

Kardinalitätsabschätzung für Projektion

- Anfrage $Q = \pi_L(R)$ mit $L = (A_1, \ldots, A_k)$ Liste von Spalten

- $$|Q| = \begin{cases} V(A,R) & \text{falls } L = (A) \text{ (mit Duplikatseliminierung)} \\ |R| & \text{falls Schlüsselattribut(e) von } R \text{ in } L \\ |R| & \text{ohne Duplikateneliminierung} \\ \min\{|R|, \prod_{A \in L} V(A,R)\} & \text{sonst} \end{cases}$$

Kardinalitätsabschätzungen für Mengenoperationen

- Abschätzung der Kardinalitäten für
  - Vereinigung (∪):

$$|R \cup S| \leq |R| + |S|$$

  - Differenz (−):

$$\max\{0, |R| - |S|\} \leq |R - S| \leq |R|$$

  - Kartesisches Produkt (×):

$$|R \times S| = |R| \cdot |S|$$

Kardinalitätsabschätzung für Join

- Im Allgemeinen nicht-trivial
- Bei Fremdschlüsselbeziehungen wie folgt abschätzbar:
  - Fremdschlüssel $S.A$ auf Primärschlüssel $R.A$
    - $|R \bowtie_{R.A=S.A} S| = |S|$
  - Bei Fremdschlüssel auf sonstiges Attribut
    - $|R \bowtie_{R.A=S.B} S| =$

$$\begin{cases} \frac{|R| \cdot |S|}{V(A,R)} & \text{falls } S.B \text{ Fremdschlüssel auf } R.A \\ \frac{|R| \cdot |S|}{V(B,S)} & \text{falls } R.A \text{ Fremdschlüssel auf } S.B \end{cases}$$

```
create table R (
   A int not null primary key, ...);
create table S (
   A int not null references R(A), ...);
```

```
create table R (
   A int, ...);
create table S (
   B int references R(A), ...);
```

```
create table R (
   A int references S(B), ...);
create table S (
   B int, ...);
```

Join-Optimierung: Es ist noch nicht alles gesagt...

- Problem: Welche Reihenfolge beim Join?
- Beispiel
  - Auflistung der möglichen Ausführungspläne, d.h.
    alle 3-Wege-Join-Kombinationen bei Join über Relationen R, S und T



Join-Optimierung: Suchraum

- Der sich ergebende Suchraum ist enorm groß:
  Schon bei 4 Relationen ergeben sich 120 Möglichkeiten

Anzahl an Relationen	C_{n-1}	Anzahl an Bäumen
2	1	2
3	5	12
4	14	120
5	42	1.680
6	132	30.240
7	429	665.280
8	1.430	17.297.280
10	16.796	17.643.225.600

Anzahl der Bäume für n Eingaberelationen:

$$n! \cdot C_{n-1} = \frac{\big(2(n-1)\big)!}{(n-1)!}$$

$C_{n-1} = \frac{(2(n-1))!}{n!(n-1)!}$ ($(n-1)$-te Catalanzahl)

Dynamische Programmierung

- Sammle gute Zugriffspläne für Einzelrelation (z.B. auch mit Indexscan und mit Ausnutzung von Ordnungen)
- Beschränke dich bei der Betrachtung der nächsten Kombination auf die guten Pläne der vorherigen Kombination

- Annahme: Optimalitätsprinzip

  Um den global optimalen Plan zu finden, reicht es aus, die optimalen Pläne bzgl. der Unteranfragen zu betrachten
  - Muss nicht gelten!

P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price: Access Path Selection in a Relational Database Management System. *In* Proc. ACM SIGMOD Conference on the Management of Data, pages 23-34, **1979**.

Kardinalitätsabschätzung: Beispiel

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Ausführungsplan A:



- Ausführungsplan B:



- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

Kardinalitätsabschätzung: Beispiel

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

- Abschätzung der Kardinalitäten der drei möglichen Operationen im ersten Schritt
  - $p \bowtie_{p.AuftragID=a.AuftragID} a$
    - $p.AuftragID$ Fremdschlüssel auf $a.AuftragID$
      $\rightarrow |p \bowtie_{\dots} a| = |p| = 6.000.000$
  - $a \bowtie_{a.KundenID=k.KundenID} k$
    - $a.KundenID$ Fremdschlüssel auf $k.KundenID$
      $\rightarrow |a \bowtie_{\dots} k| = |a| = 1.500.000$
  - $s = \sigma_{k.Name="IBM\ Corp."}(k)$
    - Annahme, dass der Name eindeutig ist
      $\rightarrow$ Kardinalität: $|s| = 1$

Kardinalitätsabschätzung: Beispiel

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$

$$150.000 \left|\, \sigma_{1} \right. k$$

- Abschätzung der Kardinalitäten der zwei möglichen nächsten Operationen
  - $p \bowtie_{p.AuftragID=a.AuftragID} a$
    - $|p \bowtie_{\dots} a| = |p| = 6.000.000$ (wie vorher)
  - $j = a \bowtie_{a.KundenID=s.KundenID} s$
    - $|j| = |a \bowtie_{\dots} s| = \frac{|a| \cdot |s|}{V(KundenID,s)} = \frac{1.500.000 \cdot 1}{1}$
      - Bzw. immer noch Fremd- auf Primärschlüssel
    - Als Selektion auffassen, da $|s| = 1$:

$$|a| \cdot \text{sel}(KundenID = id) = |a| \cdot \frac{1}{V(KundenID, a)}$$

    - Annahme $V(KundenID, a) = 150.000$, da $|k| = 150.000$, dann $|j| = \frac{1.500.000 \cdot 1}{150.000} = 10$

Kardinalitätsabschätzung: Beispiel

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$



- Abschätzung der Kardinalitäten der einen möglichen nächsten Operation
  - Nicht so wichtig, weil letzte Operation
  - $p \bowtie_{p.AuftragID=j.AuftragID} j$
    - $|p \bowtie_{\dots} j| = \frac{|p| \cdot |j|}{V(AuftragID,j)} = \frac{6.000.000 \cdot 10}{10}$
      - Bzw. immer noch Fremd- auf Primärschlüssel
    - Als zehnfache Selektion auffassbar ($|j| = 10$):

$$|j| \cdot \text{sel}(AuftragID = id) = |j| \cdot \frac{|p|}{V(AuftragID, p)}$$

    - Unter Annahme der Gleichverteilung:
      - $\frac{|p|}{V(AuftragID,p)} = \frac{6.000.000}{1.500.000} = 4$ Posten / Auftrag
      - Dann $|p \bowtie_{\dots} j| = 10 \cdot 4 = 40$

Kardinalitätsabschätzung: Beispiel

```
select p.Teile_ID, p.Anzahl, p.Preis
from Auftragsposten p, Auftraege a, Kunden k
where p.Auftrag_ID = a.Auftrag_ID
      and a.Kunden_ID = k.Kunden_ID
      and k.Name = 'IBM Corp.';
```

- Kardinalitäten der Tabellen
  - $|k| = 150.000$
  - $|a| = 1.500.000$
  - $|p| = 6.000.000$



- Tatsächliche Kardinalitäten:



Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
  - Operator-Evaluierer
  - Optimierer
    - Datenunabhängige Optimierung
      - Prädikatsvereinfachung
      - Anfrageentschachtelung
    - Datenabhängige Optimierung
      - Kardinalitätsabschätzung
      - Join-Reihenfolgen
- Transaktionsmanagement



Architektur eines DBMS

- Speicherung
- Anfragebeantwortung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 6. Anfrageverarbeitung

A. Speicherung
- Speichermedien
- Verwaltung
- Puffer
- Zugriff

B. Indexierung
- ISAM-Index
- B^+-Bäume (B*-Bäume)
- Hash-basierte Indexe

C. Anfragebeantwortung
- Sortieren
- Join-Verarbeitung
- Weitere Operationen
- Pipelining

D. Anfrageoptimierung
- Rewriting
- Datenabhängige Optimierung

→ Transaktionen

WWU
MÜNSTER

Transaktionen

Datenbanken



Transaktionszustandsdiagramm

wissen.leben

Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - Noch offen: verteilte DBs, deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs

Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 7. Transaktionen

A. Transaktionsverarbeitung
- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

B. Sperrverwaltung
- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

C. Wiederherstellungsverwaltung
- Fehlersituationen
- Logging
- Recovery

Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Viele gleichzeitige Nutzer*innen

- Jede*r Nutzer*in
  - Stellt Anfrage an den Kontostand
  - Verändert den Kontostand

  → Abfolge von DB-Befehlen = Transaktion
- Anforderungen:
  - **A**tomicity (Atomarität): Alles oder nichts
  - **C**onsistency (Konsistenz): Vorher ok, hinterher ok
  - **I**solation (Isolation): Jede*r denkt, sie seien alleine auf der DB
  - **D**urability (Dauerhaftigkeit): Transaktionen bestätigt? Dann sind die Daten jetzt sicher

ACID-Eigenschaften (später mehr)

Es ist später



DB-Transaktion - Definition

- (DB-)Transaktion: logische Verarbeitungseinheit auf einer DB
  - Enthält typischerweise mehrere DB-Operationen (Lesen, Einfügen, Aktualisieren, Löschen)
  - Können aus Anwendungsprogrammen heraus oder im Rahmen manueller Eingaben über eine SQL-Schnittstelle realisiert werden
  - Werden mit einem speziellen Schlüsselwort gestartet (BEGIN_TRANSACTION, BOT) und beendet (END_TRANSACTION, EOT)
    - Gelten nach Ausführung von END_TRANSACTION als teilweise (bzw. vorläufig) bestätigt
      - Sind dann aber erst „logisch“ und noch nicht physisch persistent abgeschlossen
      - Werden erst nach COMMIT endgültig physisch durchgeführt (Vgl. Logging, siehe später)
      - Müssen zurückgesetzt werden (ABORT), wenn die Persistenz nicht durch COMMIT erreicht wird

      → Transaktion muss mit COMMIT oder ABORT enden
- Lesende Transaktion: DB wird innerhalb einer Transaktion nicht verändert

Schüsselwörter einer Transaktion

- begin_transaction: Startet eine Transaktion
- end_transaction: Beendet eine Transaktion logisch
- commit (Bestätigung): Fügt die Ergebnisse der Transaktion persistent in den Datenbestand ein
- abort: Abbruch einer Transaktion, DB-Änderungen werden vollständig verworfen bzw. zurückgenommen



Transaktionszustandsdiagramm

Lese- und Schreib-Operationen

- Für diese Lerneinheit: Datenbank = Sammlung von Datenobjekten
  - Datenobjekt kann Feld (Attribut) eines Datensatzes (Tupels), ein gesamter Datensatz oder ein Plattenblock (Zusammenfassung vieler Datensätze) sein
    → Abstraktion von deren Größe (Granularität)
- Grundlegenden DB-Zugriffsoperationen in Transaktionen:
  - $\text{read\_item}(X)$:
    - Liest ein Datenobjekt X aus einer DB in eine Programmvariable
    - Zur Vereinfachung der Notation wird angenommen, dass die Programmvariable ebenfalls X heißt
  - $\text{write\_item}(X)$:
    - Schreibt den Wert einer Programmvariablen X in das entsprechende Datenobjekt X der DB
  - Beispiel: Transaktionen T_1, T_2

T_1	T_2
$\text{read\_item}(X)$	$\text{read\_item}(X)$
$X := X - N$	$X := X + M$
$\text{write\_item}(X)$	$\text{write\_item}(X)$
$\text{read\_item}(Y)$	
$Y := Y + N$	
$\text{write\_item}(Y)$	

Realisierung von Read und Write

- Aktionen zur Realisierung einer $\text{read\_item}(X)$-Anweisung:
  1. Suche Adresse des Plattenblocks, der Datenobjekt X enthält
  2. Kopiere den ermittelten Plattenblock in Arbeitsspeicher (sofern der Plattenblock noch extern ist, d.h. sich nicht im internen Arbeitsspeicher befindet)
  3. Kopiere den Wert des Datenobjekts X in die entsprechende Programm-variable X
- Aktionen zur Realisierung einer $\text{write\_item}(X)$-Anweisung:
  1. Suche Adresse des Plattenblocks, der Datenobjekt X enthält
  2. Kopiere den ermittelten Block in Arbeitsspeicher (sofern der Plattenblock noch extern ist, d.h. sich nicht im internen Arbeitsspeicher befindet)
  3. Kopiere den Wert der Programmvariablen X an die entsprechende Stelle im Arbeitsspeicher
  4. Schreibe den modifizierten Block sofort oder später in den Plattenblock

Fehlersituationen

Transaktionsverarbeitung

Fehlermöglichkeiten

- Eine (verschachtelt) nebenläufige Ausführung der in T_1 und T_2 enthaltenen Anweisungen ohne jegliche Kontrollmechanismen, die die korrekte Ausführung sicherstellen, kann in verschiedene Fehlersituationen münden:
  - Lost Update
  - Dirty Read
  - Ghost Update
  - Unrepeatable Read

  - *Für die vier grundsätzlichen Fehlersituationen folgt je ein Beispiel zur Erläuterung. Die Fehler entstehen je nach konkreter zeitlicher Abfolge der einzelnen Operationen in nebenläufigen Transaktionen.*
  - *Für alle Beispiele in diesem Foliensatz: Annahme, dass die Transaktionen mit ihrer letzten Operation logisch beendet werden (end_transaction)*

LOST UPDATE

- Operationen zweier Transaktionen greifen auf die gleichen Datenobjekte zu
- Dabei überschneiden sie sich zeitlich derart, dass einzelne durchgeführte Aktualisierungen der Datenobjekte verloren gehen
- Beispiel: Verschränkte Ausführung von T_1, T_2
  - Aktualisierung von X in T_1 in Zeile 5 geht durch das Schreiben in T_2 in Zeile 7 verloren
    - X hat einen falschen Wert in Zeile 3

Zeit

	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3		read_item(X)
4		$X := X + M$
5	write_item(X)	
6	read_item(Y)	
7		write_item(X)
8	$Y := Y + N$	
9	write_item(Y)	

DIRTY READ



- Durch eine Transaktion, die später abgebrochen wird, findet zunächst eine Aktualisierung eines Datenobjekts statt
- Eine andere Transaktion liest das modifizierte Datenobjekt, bevor die bereits durchgeführte Aktualisierung in T_1 verworfen wird
- Beispiel: Verschränkte Ausführung von T_1, T_2
  - T_1 schlägt fehl → Wert von X muss auf den alten Wert zurückgesetzt werden
  - Inzwischen hat aber T_2 den von T_1 geschriebenen Wert gelesen und verrechnet

Zeit

	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3	write_item(X)	
4		read_item(X)
5		$X := X + M$
6		write_item(X)
7	read_item(Y)	
8	abort	
9		

Wie spielt ACID hier rein?

GHOST UPDATE

- Zwei nebenläufige Transaktionen
  - Eine Transaktion berechnet eine Aggregation auf einer Menge von Datenobjekten
  - Eine andere Transaktion aktualisiert einige dieser Datenobjekte
- In die Aggregation gehen u.U. Datenobjekte ein, die bereits aktualisiert wurden, und andere mit ihrem Wert vor einer Aktualisierung
- Beispiel: Verschränkte Ausführung von T_1, T_3
  - T_3 liest X, nachdem N in T_1 subtrahiert wurde, und Y, bevor N in T_1 addiert wird
    - Ergibt ein um N falsches Resultat

	T_1	T_3
1		$sum := 0$
2		$\text{read\_item}(A)$
3		$sum := sum + A$
		$\vdots$
4	$\text{read\_item}(X)$	
5	$X := X - N$	
6	$\text{write\_item}(X)$	
7		$\text{read\_item}(X)$
8		$sum := sum + X$
9		$\text{read\_item}(Y)$
10		$sum := sum + Y$
11	$\text{read\_item}(Y)$	
12	$Y := Y + N$	
13	$\text{write\_item}(Y)$	

UNREPEATABLE READ

- Datenobjekt wird innerhalb einer Transaktion mehrfach gelesen, während eine andere Transaktion dieses Datenobjekt modifiziert
- Je nach zeitlicher Abfolge der Anweisungen in den beiden Transaktionen wird nicht der jeweils gleiche Wert wiederholt gelesen
- Ähnlicher Fehler: PHANTOM READ
  - Während eine Transaktion eine Tabelle (wiederholt) liest, fügt eine andere Transaktion neue Tupel ein oder löscht Tupel
- Beispiel: Verschränkte Ausführung von T', T''
  - In Zeile 9 hat X einen um N niedrigeren Wert, als wenn der Lesebefehl vor Start von T'' erfolgt wäre

	T'	T''
1	$sum := 0$	
2	$\text{read\_item}(A)$	
3	$sum := sum + A$	
	$\vdots$	
4	$\text{read\_item}(X)$	
5	$sum := sum + X$	
6		$\text{read\_item}(X)$
7	$\vdots$	$X := X - N$
8		$\text{write\_item}(X)$
9	$\text{read\_item}(X)$	

Zwischenzusammenfassung

- Verschiedene Operationen und Zustände während der Transaktionsverarbeitung
  - BEGIN_TRANSACTION, END_TRANSACTION, COMMIT, ABORT
  - READ_ITEM, WRITE_ITEM
- Probleme bei verschränkter Ausführung von Transaktionen
  - *Lost Update*: Überschreiben einer Aktualisierung bzw. Arbeiten mit einem veralteten Wert
  - *Dirty Read*: Lesen eines Wertes, der durch eine Transaktion geändert wurde, die dannach abgebrochen wurde
  - *Ghost Update*: Zusammenhängende aktualisierte und nicht aktualisierte Werte werden verarbeitet
  - *Unrepeatable Read*: Werte ändern sich beim wiederholten Lesen (andere Reihenfolge führt zu anderem Ergebnis)

Schedules

Transaktionsverarbeitung

Nebenläufige Ausführung

- Große Menge von Transaktionen
  - Sequentielle Ausführung vermeidet Probleme, aber kostet viel Zeit
  - Nebenläufige Ausführung bei entsprechender Hardware-Unterstützung ist schneller, kann aber zu Anomalien führen
- Anforderung an DBMS:
  Nebenläufige Ausführung und keine Anomalien

- Steuerprogramm (Scheduler) entscheidet über die Ausführungsreihenfolge der nebenläufigen Datenbankzugriffe
  - Ziel: Korrekte verschachtelte *Schedules* (Definition folgt)



Schedules von Transaktionen

- Ausführungsplan / Schedule S
  - Integrierte Abfolge der Operationen überlappend ausgeführter Transaktionen
  - Schedule S von n Transaktionen $T_1, \ldots, T_n$ ist eine Sequenz von Transaktionsoperationen mit der Eigenschaft, dass die Operationen einer Transaktion T_i in S in der gleichen Reihenfolge wie in T_i erscheinen,
    - I.e., $T_i = \langle p_1, \ldots, p_{N'} \rangle$ und $S = \langle o_1, \ldots, o_N \rangle$ mit $T_i = \pi_i(S)$ (Reihenfolge erhaltende Abb. von S auf T_i)
- Abkürzungen für die folgenden Folien:
  - Operation (read oder write) auf Objekt X einer Transaktion T_i: $p_i(X)$
  - READ_ITEM auf Objekt X einer Transaktion T_i: $r_i(X)$
  - WRITE_ITEM von Objekt X einer Transaktion T_i: $w_i(X)$
  - COMMIT einer Transaktion T_i: c_i
  - ABORT einer Transaktion T_i: a_i

Beispiele für Schedules

- Transaktionen
  - Reduziert auf w_i, r_i
  - $T_1 = \langle r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
    - Bzw. $T_1 = \langle r_1(X), w_1(X), r_1(Y), a_1 \rangle$
  - $T_2 = \langle r_2(X), w_2(X) \rangle$
- Schedules
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$
  - $S_b = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), a_1 \rangle$

S_a	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3		read_item(X)
4		$X := X + M$
5	write_item(X)	
6	read_item(Y)	
7		write_item(X)
8	$Y := Y + N$	
9	write_item(Y)	

S_b	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3	write_item(X)	
4		read_item(X)
5		$X := X + M$
6		write_item(X)
7	read_item(Y)	
8	abort	
9		

Serielle Schedules

- Schedule S ist seriell, wenn die Operationen jeder in S enthaltenen Transaktion T vollständig hintereinander (streng sequenziell) ausgeführt werden
  - Für jede Transaktion T gilt, dass ihr Schritte direkt aufeinander folgen
  - Andernfalls ist ein Schedule nicht-seriell

Serielle Schedules: Beispiele

- Schedule S_d
  - $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$
  - $T_1 | T_2$
- Schedule S_e
  - $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
  - $T_2 | T_1$

S_d	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3	write_item(X)	
4	read_item(Y)	
5	$Y := Y + N$	
6	write_item(Y)	
7		read_item(X)
8		$X := X + M$
9		write_item(X)

S_e	T_1	T_2
1		read_item(X)
2		$X := X + M$
3		write_item(X)
4	read_item(X)	
5	$X := X - N$	
6	write_item(X)	
7	read_item(Y)	
8	$Y := Y + N$	
9	write_item(Y)	

Korrekte Schedules

- Anomalien können nur auftreten, wenn die Schritte mehrerer Transaktionen verschränkt ausgeführt werden
  - Wenn alle Transaktionen bis zum Ende ausgeführt werden (keine Nebenläufigkeit), treten keine Anomalien auf
- Jede serielle Ausführung ist korrekt
- Verzicht auf nebenläufige Ausführung nicht praktikabel
  - Wartezeiten auf Platten
- Jede verschränkte Ausführung, die einen gleichen Zustand, wie eine serielle Ausführung erzeugt, ist korrekt

Korrekte Schedules

- Manchmal kann man einfach Teilschritte aus verschiedenen Transaktionen in einem Plan umordnen
  - Nicht jedoch die Teilschritte innerhalb einer einzelnen Transaktion (da sonst eventuell anderes Ergebnis)
- Jeder Plan S', der durch legale Umordnung von S generiert werden kann, heißt äquivalent zu S
- Falls Umordnung nicht möglich (weil Ergebnis möglicherweise verfälscht) → Konflikt

Konflikte

- Zwei Operationen $p_i(X), p_j(X)$ in einem Schedule stehen in Konflikt (sind konfliktär), wenn alle folgenden Bedingungen erfüllt sind:
  - Sie gehören zu unterschiedlichen Transaktionen ($i \neq j$)
  - Sie greifen auf das gleiche Datenobjekt zu (X)
  - Mindestens eine Operation schreibt X ($p_i(X) = w_i(X) \vee p_j(X) = w_j(X)$)
    - Konfliktmatrix siehe rechts unten

  → Reihenfolge von $p_i(X), p_j(X)$ ist relevant
    - Nicht in Konflikt stehende Operationen können auch parallel ausgeführt werden
      - Solche Schedules heißen partiell geordnet

Konfliktmatrix

	$r_j(X)$	$w_j(X)$
$r_i(X)$		X
$w_i(X)$	X	X

Serialisierbarkeit

- Schedule S mit n Transaktionen ist serialisierbar, wenn er zu einem seriellen Schedule S' äquivalent ist, d.h., den gleichen DB-Zustand erreicht
  - Reihenfolge der konfliktären Operationen ist gleich
- Ausführung eines serialisierbaren Schedules S ist korrekt
  - S muss nicht seriell sein

Konflikte, Äquivalenz, Serialisierbarkeit, Korrektheit: Beispiele

- Schedule S_d
  - $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$
  - Konflikte: $r_1(X), w_2(Y), w_1(X), r_2(X), w_1(X), w_2(X)$
- Schedule S_c
  - $S_c = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), w_1(Y) \rangle$
  - Konflikte: Wie oben
    - Äquivalent
    - Serialisierbar
  - Damit: S_c korrekt





Konflikte, Äquivalenz, Serialisierbarkeit, Korrektheit: Beispiele

- Schedule S_a
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$
  - Nicht äquivalent mit S_d
    - Konfliktäre Operationen $r_2(X), w_1(X)$ in S_a in anderer Reihenfolge in S_d
  - Auch nicht äquivalent mit $S_e = T_2 | T_1$
    - $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$
  - Damit S_a nicht serialisierbar
  - Damit S_a nicht korrekt





Korrektheitsprüfung mittels Serialisierungsgraphen

- Serialisierungsgraph (oder Konfliktgraph) SG für ein Schedule S ist ein gerichteter Graph $G = (N, E)$, wobei
  - $N = \{T_1, \dots, T_n\}$ Menge von Knoten, ein Knoten für jede Transaktion in S
  - $E = \{e_1, \dots, e_m\}$ Menge gerichteter Kanten
    - Jede Kante e im Graphen hat die Form $(T_i \longrightarrow T_j), 1 \leq i \leq n, 1 \leq j \leq n$, wobei T_i der Start- und T_j der Endknoten von e ist
- Kante $T_i \longrightarrow T_j$ wird erzeugt, wenn in S eine Operation von T_i vor einer mit ihr in Konflikt stehenden Operation in T_j vorkommt
  - I.e., $p_i(X)$ und $p_j(X)$ stehen in Konflikt und $p_i(X)$ erscheint vor $p_j(X)$ in S

Serialisierbarkeitstheorem
Schedule S ist serialisierbar, wenn der zugehörige Serialisierungsgraph keine Zyklen aufweist.

Korrektheitsprüfung

- Eingabe: Schedule S
- Ausgabe: S serialisierbar ja/nein
- Vorgehen
  1. Baue einen Serialisierungsgraph SG für S
     a. Erzeuge für jede Transaktion T_i, die in S auftritt, einen Knoten T_i in SG
     b. Für jeden Fall in S, bei dem erst T_i eine Operation $p_i(X)$ ausführt und dann T_j eine Operation $p_j(X)$ ausführt, wobei mindestens ein $p(X)$ eine Schreiboperation ist, erzeuge eine Kante $(T_i \longrightarrow T_j)$ in SG
        - Drei Fälle: (i) $w_i(X), r_j(X)$, (ii) $r_i(X), w_j(X)$, (iii) $w_i(X), w_j(X)$
  2. Prüfe, ob SG keinen Zyklus enthält
     a. Wenn kein Zyklus vorhanden: S ist serialisierbar
     b. Sonst: S ist nicht serialisierbar

Serialisierungsgraphen: Beispiele

- Serieller Schedule S_d
  - $S_d = \langle r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X) \rangle$

- Serieller Schedule S_e
  - $S_e = \langle r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y) \rangle$

- Beide Schedules serialisierbar

S_d	T_1	T_2
1	read_item(X)	
2	$X := X - N$	
3	write_item(X)	
4	read_item(Y)	
5	$Y := Y + N$	
6	write_item(Y)	
7		read_item(X)
8		$X := X + M$
9		write_item(X)

S_e	T_1	T_2
1		read_item(X)
2		$X := X + M$
3		write_item(X)
4	read_item(X)	
5	$X := X - N$	
6	write_item(X)	
7	read_item(Y)	
8	$Y := Y + N$	
9	write_item(Y)	

Serialisierungsgraphen: Beispiele

- Schedule S_a
  - $S_a = \langle r_1(X), r_2(X), w_1(X), r_1(Y), w_2(X), w_1(Y) \rangle$



- Schedule S_c
  - $S_c = \langle r_1(X), w_1(X), r_2(X), w_2(X), r_1(Y), w_1(Y) \rangle$

- S_a nicht serialisierbar (Zyklus), S_c ja





Serialisierungsgraphen: Beispiele

- Transaktionen T_1, T_2, T_3

T_1	T_2	T_3
read_item(X)	read_item(Z)	read_item(Y)
write_item(X)	read_item(Y)	read_item(Z)
read_item(Y)	write_item(Y)	write_item(Y)
write_item(Y)	read_item(X)	write_item(Z)
	write_item(X)	

- Schedule S_g rechts
  - Serialisierungsgraph:
    - Zyklen im Graph

  → Nicht serialisierbar (kein äquivalenter serieller Schedule)



Transaktionen

S_g	T_1	T_2	T_3
1		read_item(Z)	
2		read_item(Y)	
3		write_item(Y)	
4			read_item(Y)
5			read_item(Z)
6	read_item(X)		
7	write_item(X)		
8			write_item(Y)
9			write_item(Z)
10		read_item(X)	
11	read_item(Y)		
12	write_item(Y)		
13		write_item(X)	

In Tabelle keine weiteren Konflikte hervorgehoben, die zu einer schon bestehenden Kante gehören.

Serialisierungsgraphen: Beispiele

- Transaktionen T_1, T_2, T_3

T_1	T_2	T_3
read_item(X)	read_item(Z)	read_item(Y)
write_item(X)	read_item(Y)	read_item(Z)
read_item(Y)	write_item(Y)	write_item(Y)
write_item(Y)	read_item(X)	write_item(Z)
	write_item(X)	

- Schedule S_h rechts
  - Serialisierungsgraph:
    - Graph zyklenfrei

    → Serialisierbar



Transaktionen

S_h	T_1	T_2	T_3
1			read_item(Y)
2			read_item(Z)
3	read_item(X)		
4	write_item(X)		
5			write_item(Y)
6			write_item(Z)
7		read_item(Z)	
8	read_item(Y)		
9	write_item(Y)		
10		read_item(Y)	
11		write_item(Y)	
12		read_item(X)	
13		write_item(X)	

In Tabelle keine weiteren Konflikte hervorgehoben, die zu einer schon bestehenden Kante gehören.

Äquivalente serielle Schedules

- Gegeben ein zyklenfreier Serialisierungsgraphen SG für ein Schedule S
- Äquivalenter serieller Schedule durch topologische Sortierung von SG
  - *Topologische Sortierung*: Sequenz aller Knoten im Graph, wobei Elternknoten vor Kindknoten in der Sequenz erscheinen müssen
  - Beispiel vorherige Folie
    - Serialisierungsgraph:
      - Graph zyklenfrei

      → Serialisierbar

      → Äquivalenter serieller Schedule:

      $T_3|T_1|T_2$



Transaktionen

S_h	T_1	T_2	T_3
1			read_item(Y)
2			read_item(Z)
3	read_item(X)		
4	write_item(X)		
5			write_item(Y)
6			write_item(Z)
7		read_item(Z)	
8	read_item(Y)		
9	write_item(Y)		
10		read_item(Y)	
11		write_item(Y)	
12		read_item(X)	
13		write_item(X)	

Zwischenzusammenfassung

- Schedules: Sequenz von Operationen aus einer Menge von Transaktionen
  - Ursprüngliche Reihenfolge der Operationen aus den Transaktionen beizubehalten
- Serielle Schedules: strikt sequentielle Anordnung der Transaktionen
  - Serielle Schedules sind korrekt
- Konflikte: Wenn Schreiboperationen auf demselben Datenobjekt involviert sind
- Serialisierbarkeit
  - Serialisierbarer Schedule äquivalent zu seriellem Schedule → Korrekter Schedule
- Serialisierungsgraphen
  - Knoten für Transaktionen; Kanten bei konfliktären Operationen
  - Test auf Serialisierbarkeit / Korrektheit: Zyklenfreier Serialisierungsgraph
  - Äquivalenter serieller Schedule mittels topologischer Sortierung

Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
    - Serielle, serialisierbare Schedules
    - Konflikte, Serialisierungsgraph
  - Sperrverwaltung
  - Wiederherstellungsverwaltung



Überblick: 7. Transaktionen

A. Transaktionsverarbeitung
- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

B. Sperrverwaltung
- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

C. Wiederherstellungsverwaltung
- Fehlersituationen
- Logging
- Recovery

Sperr-Verwaltung

- Transaktionen müssen Sperren anfragen für Datenobjekte, auf die sie zugreifen wollen
- Falls eine Sperre nicht zugeteilt wird (z.B. weil eine andere Transaktion T' die Sperre schon hält), wird die anfragende Transaktion T blockiert
  - Sperrverwalter setzt die Ausführung von Aktionen einer blockierten Transaktion T aus
- Sobald T' die Sperre freigibt, kann sie an T vergeben werden (oder an eine andere Transaktion, die darauf wartet)
  - Transaktion, die eine Sperre erhält, wird fortgesetzt
- Sperren regeln die relative Ordnung der Einzeloperationen verschiedener Transaktionen
- Ziel: Automatisierte Generierung von serialisierbaren Schedules

Implementierung eines Sperrverwalters

- Ein Sperrverwalter muss drei Aufgaben effektiv erledigen:
  1. Prüfen, welche Sperren für eine Ressource gehalten werden (um eine Sperranforderung zu behandeln)
  2. Bei Sperr-Rückgabe müssen die Transaktionen, die die Sperre haben wollen, schnell identifizierbar sein
  3. Wenn eine Transaktion beendet wird, müssen alle von der Transaktion angeforderten und gehaltenen Sperren zurückgegeben werden
- Sperrtabelle: Datenstruktur zur Speicherung der Lock-Information zu jedem Objekt
  - Meist als Hash-Tabelle organisiert, um effizient auf Lock-Informationen zugreifen zu können

Sperren und Sperrprotokolle

Sperrverwaltung

Protokolle und Sperren

- Protokolle mit Sperren:
  - Binäre Sperren (einfach aber restriktiv)
  - Mehrfachmodus- bzw. gemeinsame/exklusive Sperren (praxisrelevant)
  - Zwei-Phasen-Sperrprotokoll (praxisrelevant)
  - Multiversionsprotokolle (Verbesserung der Performanz)
  - Multiversionsprotokolle mit Zeitstempelung (Verbesserung der Performanz)
  - Zertifizierungssperren (Verbesserung der Performanz)
- Protokolle ohne Sperren:
  - Zeitstempelbasierte Transaktionsverarbeitung (praxisrelevant)

Binäre Sperren

- Mit jedem Datenobjekt X ist eine Sperre assoziiert
  - Sperre kann zwei Zustände annehmen: „gesperrt“ oder „entsperrt“
    - 🔒 $Lock(X) = 1$: Objekt ist gesperrt
    - 🔓 $Lock(X) = 0$: Objekt ist entsperrt
  - Operationen:
    - $\text{lock\_item}(X)$: sperrt das Objekt
    - $\text{unlock\_item}(X)$: entsperrt das Objekt
- Wenn eine Transaktion T_1 ein Objekt X gesperrt hat, kann eine Operation T_2 auf X nicht zugreifen (bzw. selber sperren)
  - T_2 muss warten, bis X durch T_1 wieder entsperrt wurde
- Implementierung
  - Eintrag in Sperrtabelle: $(DatenobjektID, LOCKZustand, TransaktionsID)$

Binäre Sperren

- Protokoll mit binären Sperren nach folgenden Regeln für jede Transaktion T:
  1. Vor $\text{read\_item}(X)$ und $\text{write\_item}(X)$: $\text{lock\_item}(X)$
  2. Nach allen $\text{read\_item}(X)$ und $\text{write\_item}(X)$: $\text{unlock\_item}(X)$
  3. Kein $\text{lock\_item}(X)$ durch T, wenn T schon eine Sperre auf X hat
  4. Nur dann $\text{unlock\_item}(X)$, wenn T auch eine Sperre auf X hat

- Problem:
  - Sehr restriktiv
    - Z.B. keine parallelen Leseoperationen möglich

Sperren mit Mehrfachmodus

- Mit jedem Datenobjekt X ist eine Sperre assoziiert mit drei möglichen Zuständen:
  - $Lock(X) = \mathrm{S}$ (Lesesperre)
    - Auf X wird lesend zugegriffen (*shared*) → Als Sperrmodus S bezeichnet
    - Weiterer Lesezugriff problemlos möglich, Schreibzugriff potenziell problematisch
  - $Lock(X) = \mathrm{X}$ (Schreibsperre)
    - Auf X wird schreibend zugegriffen (*exklusive*) → Als Sperrmodus X bezeichnet
    - Auf X kann durch andere Transaktion nicht zugegriffen werden
  - $Lock(X) = 0$ (entsperrt) : Auf X wird nicht zugegriffen
  - Kompatibilitätsmatrix (rechts)
    - Zustand: Gibt es eine Sperre?
    - Anforderung: Was für eine Sperre soll gesetzt werden?
- Operationen: $\mathrm{read\_lock}(X)$, $\mathrm{write\_lock}(X)$ und $\mathrm{unlock}(X)$

Anforderung \ Zustand	0	S	X
S	✓	✓	–
X	✓	–	–

Sperren mit Mehrfachmodus

- Mögliche Implementierung:
  - Einträge in der Sperrtabelle:
    $(DatenobjektID, LOCKZustand, \#Zugriffsoperationen, [TransaktionsID, \ldots])$
  - Idee: Zählen der lesenden Transaktionen ($\#Zugriffsoperationen$)
  - Bei schreibenden Transaktionen ($\text{lock}(X) = \text{X}$) ist $\#Zugriffsoperationen = 1$
- Regeln für Transaktion T:
  1. $\text{read\_lock}(X)$ oder $\text{write\_lock}(X)$ anstoßen vor irgendwelchen $\text{read\_item}(X)$
  2. $\text{write\_lock}(X)$ vor irgendwelchen $\text{write\_item}(X)$
  3. $\text{unlock}(X)$ nach allen $\text{read\_item}(X)$ und $\text{write\_item}(X)$
  4. Kein $\text{read\_lock}(X)$, falls irgendeine Sperre auf X durch T besteht
  5. Kein $\text{write\_lock}(X)$, falls irgendeine Sperre auf X durch T besteht
  6. Nur dann $\text{unlock}(X)$, wenn eine Sperre auf X durch T besteht

Anforderung \ Zustand	0	S	X
S	✓	✓	–
X	✓	–	–

Sperren mit Mehrfachmodus - Sperrenänderung

- Für weniger restriktive Lock-Mechanismen Änderung von Regeln 4 und 5 → Sperrenänderung
- Damit kann eine Schreibsperre zur Lesesperre gelockert oder eine Lesesperre zur Schreibsperre verschärft werden:
  1. $\text{read\_lock}(X)$ oder $\text{write\_lock}(X)$ anstoßen vor irgendwelchen $\text{read\_item}(X)$
  2. $\text{write\_lock}(X)$ vor irgendwelchen $\text{write\_item}(X)$
  3. $\text{unlock}(X)$ nach allen $\text{read\_item}(X)$ oder $\text{write\_item}(X)$
  4. Kein $\text{read\_lock}(X)$, falls eine Lesesperre auf X durch T besteht
  5. Kein $\text{write\_lock}(X)$, falls eine Schreibsperre auf X durch T besteht
  6. Nur dann $\text{unlock}(X)$, wenn eine Sperre auf X durch T besteht

Sperren: Bewertung

- Binäre und Mehrfach-Sperren garantieren selbst noch keine Serialisierbarkeit
- Beispiel: Schedule S
  - Anfangswerte:
    - $X = 20, Y = 30$
  - Resultat von S:
    - $X = 50, Y = 50$
  - Resultat des seriellen Plans $T_1 | T_2$:
    - $X = 50, Y = 80$
  - Resultat des seriellen Plans $T_2 | T_1$:
    - $X = 70, Y = 50$

→ Besseres Protokoll gesucht!

S	T_1	T_2
1	read_lock(Y)	
2	read_item(Y)	
3	unlock(Y)	
4		read_lock(X)
5		read_item(X)
6		unlock(X)
7		write_lock(Y)
8		read_item(Y)
9		$Y := X + Y$
10		write_item(Y)
11		unlock(Y)
12	write_lock(X)	
13	read_item(X)	
14	$X := X + Y$	
15	write_item(X)	
16	unlock(X)	

Zwei-Phasen-Sperrprotokoll (*Two-Phase Locking*, 2PL)

- Idee: Sperroperationen aller Objekte werden vor der ersten **Ent**sperroperation ausgeführt
- Damit ergeben sich zwei Phasen für eine Transaktion T:
  1. Wachstumsphasen:
     - T sammelt immer mehr Sperren auf Objekte
  2. Schrumpfungsphase:
     - T gibt immer mehr Sperren auf Objekte frei
  - Bei Sperrenänderungen:
    - Verschärfungen nur in Wachstumsphase (Lesesperre zu Schreibsperre)
    - Lockerungen nur in Schrumpfungsphase (Schreibsperre zu Lesesperre)



- Schedules, die dem Zwei-Phasen-Sperrprotokoll folgen, sind serialisierbar
  - Schreiboperationen sind durch exklusive Sperren abgesichert

Beispiel

- Nicht Zwei-Phasen-Sperrprotokoll
  - T_1: write_lock(X) nach unlock(Y)
  - T_2: write_lock(Y) nach unlock(X)

T_1	T_2
read_lock(Y)	read_lock(X)
read_item(Y)	read_item(X)
unlock(Y)	unlock(X)
write_lock(X)	write_lock(Y)
read_item(X)	read_item(Y)
$X := X + Y$	$Y := X + Y$
write_item(X)	write_item(Y)
unlock(X)	unlock(Y)

- Zwei-Phasen-Sperrprotokoll
  - Problem: Kann zu Deadlock führen
    - Beide Transaktionen aktiv + warten auf Sperre

T_2' wartet auf unlock(Y) von T_1'
und
T_1' wartet auf unlock(X) von T_2'

T_1'	T_2'
read_lock(Y)	read_lock(X)
read_item(Y)	read_item(X)
write_lock(X)	write_lock(Y)
unlock(Y)	unlock(X)
read_item(X)	read_item(Y)
$X := X + Y$	$Y := X + Y$
write_item(X)	write_item(Y)
unlock(X)	unlock(Y)

Varianten des Zwei-Phasen-Sperrprotokolls

Was sind Beweggründe für die Varianten?

- Konservatives 2PL: Transaktion T sperrt erst alle Objekte, bevor sie mit der Verarbeitung beginnt
  - Keine Deadlocks, aber alle benötigten Sperren müssen vorab bekannt und deklariert sein
- Striktes 2PL: Transaktion T hebt *Schreibsperren* erst auf, wenn T beendet wurde
  - In der Praxis recht verbreitet, garantiert aber nicht Deadlock-Freiheit
- Rigoroses 2PL: T hebt *Lese- und Schreibsperren* erst auf, wenn T beendet wurde
  - Einfacher umzusetzen als die strikte Variante, verhindert aber auch nicht Deadlocks





Beispiel

- Transaktionen T_1, T_2

T_1	T_2
read_lock(X)	write_lock(Y)
read_item(X)	read_item(Y)
read_lock(Y)	$Y := Y - 100$
read_item(Y)	write_lock(Z)
read_lock(Z)	read_item(Z)
read_item(Z)	$Z := Z + 100$
$S := X + Y + Z$	write_item(Y)
unlock(X)	write_item(Z)
unlock(Y)	unlock(Y)
unlock(Z)	unlock(Z)

S	T_1	T_2	X	Y	Z
1	begin_transaction		$free$	$free$	$free$
2	read_lock(X)		1: $read$		
3	read_item(X)				
4		begin_transaction			
5		write_lock(Y)		2: $write$	
6		read_item(Y)			
7	read_lock(Y)			1: $wait$	
8		$Y := Y - 100$			
9		write_lock(Z)			2: $write$
10		read_item(Z)			
11		$Z := Z + 100$			
12		write_item(Y)			
13		write_item(Z)			
14		end_transaction			
15		unlock(Y)		1: $read$	
16	read_item(Y)				
17	read_lock(Z)				1: $wait$
18		unlock(Z)			1: $read$
19	read_item(Z)				
20		commit			
21	$S := X + Y + Z$				
22	end_transaction				
23	unlock(X)		$free$		
24	unlock(Y)			$free$	
25	unlock(Z)				$free$
26	commit				

Probleme bei der Verwendung von Sperren

- Deadlocks können bei der Verwendung von Sperren entstehen
- Deadlock (Verklemmung):
  - Transaktion wartet auf ein Objekt, das eine andere Transaktion gesperrt hat – und umgekehrt
  - Kann auch zwischen mehr als zwei Transaktionen auftreten
  - Lösungsansätze folgen, können aber führen zu:
- Starvation (Verhungern):
  - Eine Transaktion wird über längere Zeit nicht abgearbeitet, da andere Transaktionen vorgezogen werden
  - Kompensationsmechanismen folgen

Deadlock

- Deadlock liegt vor, wenn jede Transaktion T einer Menge von zwei oder mehr Transaktionen $\boldsymbol{T}$ auf ein Objekt wartet, das von einer anderen Transaktion $T' \in \boldsymbol{T}, T' \neq T$, gesperrt wurde
- Beispiel:
  - T_1 und T_2 sperren wechselseitig Objekte X und Y
- Behandlung von Deadlocks
  - Vermeidung
  - Erkennung und Auflösung

S	T_1	T_2
1	read_lock(Y)	
2	read_item(Y)	
3		read_lock(X)
4		read_item(X)
5	write_lock(X)	
6		write_lock(Y)

Vermeidung von Deadlocks

- Konservatives 2PL
  - Eine Transaktion muss alle Objekte sperren, bevor sie ausgeführt wird
  - Ansonsten wartet sie, bis die gewünschten Objekte zugreifbar sind
  - Ist in der Praxis jedoch meist nicht umsetzbar
- Transaktionszeitstempel $TS(T)$
  - Eindeutiger Identifikator für eine Transaktion, der als Zeitstempel (*timestamp*) bezeichnet wird
    - In der Regel der Start-Zeitpunkt (Ablesen der internen Systemuhr) von T
  - Wird Transaktion T' vor T'' gestartet, so gilt $TS(T') < TS(T'')$
    - T' ist die ältere, T'' die jüngere Transaktion
  - Darauf basierende Verfahren:
    - Wait/Die
    - Wound/Wait

Vermeidung von Deadlocks

- Transaktion T' versucht Objekt X zu sperren, X durch andere Transaktion T schon gesperrt
  - Ältere Transaktionen werden bevorzugt

- Wait/Die ⏳|🗡️
  - Wenn $TS(T') < TS(T)$: T' ist älter als T und wartet
  - Wenn $TS(T') \geq TS(T)$: T' ist jünger als T und stirbt
    - T' bricht sich selbst ab (abort)
    - T' startet später mit gleichem Zeitstempel $TS(T')$ wieder (wird also älter sein, weniger häufig sterben)
  - T behält in beiden Fällen seine Sperre

- Wound/Wait: 🗡️|⏳
  - Wenn $TS(T') < TS(T)$: T' ist älter als T und verwundet/tötet T
    - T wird abgebrochen (abort)
    - T startet später mit dem gleichen Zeitstempel $TS(T)$ wieder (wird also älter sein, weniger häufig verwundet)
  - Wenn $TS(T') \geq TS(T)$: T' ist jünger als T und wartet
  - T behält seine Sperre nur, wenn T älter ist

Vermeidung von Deadlocks

- Weitere Ansätze:
  - No Waiting (NW)
    - Wenn die Transaktion keine Sperre bekommt, wird sie sofort abgebrochen
    - Viele unnötige Abbrüche und Neustarts
  - Cautious Waiting (CW)
    - T' versucht Sperre zu bekommen, T hat sie
      - Wenn T nicht blockiert ist, wird T' blockiert und wartet auf T
      - Sonst (T ist blockiert): T' wird abgebrochen (könnte länger dauern)
- Problem aller bisherigen Ansätze:
  - Sind zwar Deadlock frei
  - Erzeugen aber u.U. unnötige Abbrüche und Neustarts von Transaktionen, die nie einen Deadlock verursacht hätten → schlechtere Performanz

Erkennung von Deadlocks

- Optimistisches Verfahren: statt Vermeidung nachträgliche Erkennung und Auflösung
  - Gut, wenn wenig Deadlocks zu erwarten sind
- Grundideen:
  - (Physische) Zeitbeschränkungen
    - Wartet eine Transaktion T_i länger als eine Zeitbeschränkung t vorgibt:
      - Dann ist die Annahme: Transaktion ist in Deadlock-Situation
      - T_i wird abgebrochen
    - Vorteil: kann einfach geprüft werden
    - Nachteil: evtl. unnötige Transaktionsabbrüche
      - Schwierig Zeitbeschränkung richtig zu wählen:
        zu kurz → zu viele möglicherweise vermeidbare Abbrüche; zu lang → zu lange warten
  - (Logischer) Wartegraph
    - Nächste Folie…

Erkennung von Deadlocks: Wartegraph

- Wartegraph: Gerichteter Graph
  - Enthält für jede aktive Transaktion T_i einen Knoten
  - Wenn T_i auf eine Sperre von T_j wartet:
    - Füge Kante $(T_i \longrightarrow T_j)$ in den Graphen ein (Kante bei Freigabe löschen)
  - Weist der Wartegraph Zyklen auf, so liegt ein Deadlock vor
- Beispiel
  - T_1, T_2 starten (aktiv) → jeweils Knoten einfügen
  - Zeile 1 + 3: Objekte nicht gesperrt, alles ok
  - Zeile 5: T_1 fordert Sperre für X an
    - X gesperrt von T_2: Kante $(T_1 \longrightarrow T_2)$ → Kein Zyklus, alles ok
  - Zeile 6: T_2 fordert Sperre für Y an
    - Y gesperrt von T_1: Kante $(T_2 \longrightarrow T_1)$ → Zyklus



S	T_1	T_2
1	read_lock(Y)	
2	read_item(Y)	
3		read_lock(X)
4		read_item(X)
5	write_lock(X)	
6		write_lock(Y)

Behandlung von Deadlock - Opferauswahl

- Deadlock erkannt:
  - Transaktion T_i bestimmen, die abgebrochen werden soll, um Zyklus aufzulösen
- Folgende Heuristiken denkbar:
  - Wähle möglichst keine Transaktion, die bereits lange läuft
    - Bricht vor allem junge Transaktionen ab
    - Idee: Alte Transaktionen haben eine Chance endlich durchzulaufen
  - Wähle möglichst keine Transaktion, die bereits viele Aktualisierungen der DB durchgeführt hat
    - Idee: Nicht die ganze Arbeit verwerfen und alle Änderungen wieder rückgängig machen müssen
- Problem bei Wartegraphen-Ansatz:
  - Wann soll auf Existenz von Zyklen geprüft werden?
    - Zu oft → Kostet Zeit
    - Zu selten → Deadlocks können lange bestehen

Starvation

- Starvation: Transaktion wird über längere Zeit nicht abgearbeitet, da andere Transaktionen vorgezogen werden
  - Als Folge der Deadlock-Vermeidung oder -auflösung
- Kompensationsmechanismen:
  - First-Come-First-Served
    - Abarbeitung in Reihenfolge
  - Prioritäten (vergeben/anpassen)
    - Durch Transaktionsverwaltung abgebrochene Transaktion erhält höhere Priorität, wird deshalb bei der Neuausführung mit geringerer Wahrscheinlichkeit als Opfer gewählt
- Wait/Die und Wound/Wait vermeiden Verhungern
  - Ältere Transaktionen werden bevorzugt
  - Abgebrochene Transaktionen behalten ihre ID → Irgendwann ist jede Transaktion alt genug

Weitere Verfahren zur Mehrbenutzerkontrolle

Sperrverwaltung

Verfahren mit Zeitstempelung

- Zeitstempel $TS(T)$ der Transaktion T
  - Wie vorher definiert: Eindeutiger Identifikator, der vom DBMS generiert wird, um eine Transaktion zu identifizieren
    - In der Regel der Start-Zeitpunkt (Ablesen der internen Systemuhr) von T
- Zeitstempelbasierte Ansätze verwenden *keine Sperren* → *keine Deadlocks*
- Idee: Algorithmus stellt sicher, dass der Zugriff von konfliktären Operatoren nicht die Reihenfolge (mindestens) eines äquivalenten seriellen Schedules verletzt
  - $TS_r(X)$: Lesezeitstempel von X
    - Größter (aktuellster) $TS(T)$ aller Transaktionen, die X erfolgreich gelesen haben
  - $TS_w(X)$: Schreibzeitstempel von X
    - $TS(T)$ der Transaktion T, die X erfolgreich geschrieben hat

Basis-Zeitstempelordnung

- T möchte write_item(X) durchführen
  - Wenn $TS_r(X) > TS(T)$ oder $TS_w(X) > TS(T)$:



    - Jüngere Transaktionen haben X schon gelesen oder geschrieben
    - Operation wird abgewiesen, T wird abgebrochen („zu spät“)
      - T neu starten mit neuem Zeitstempel
  - Sonst:
    - write_item(X) wird durchgeführt
    - $TS_w(X) \coloneqq TS(T)$



- T möchte read_item(X) durchführen:
  - Wenn $TS_w(X) > TS(T)$:



    - Jüngere Transaktionen haben X schon geschrieben
    - Operation wird abgewiesen, T wird abgebrochen („zu spät“)
      - T neu starten mit neuem Zeitstempel
  - Sonst:
    - read_item(X) wird durchgeführt
    - $TS_r(X) \coloneqq \max\{TS_r(X), TS(T)\}$



Zeitstempelung: Beispiel

- Schedule: $r_8(X), r_6(X), r_9(X), w_8(X), w_{11}(X), r_{10}(X)$
  - Notation: Operation $p_i(X)$ der Transaktion T_i mit Zeitstempel $i = TS(T)$
- Initial: $TS_r(X) := 0, TS_w(X) := 0$

Anfrage	Antwort	Änderungen
$r_8(X)$	OK	$TS_r(X) := 8$
$r_6(X)$	OK	$TS_r(X) := 8$
$r_9(X)$	OK	$TS_r(X) := 9$
$w_8(X)$	abgelehnt	a_8
$w_{11}(X)$	OK	$TS_w(X) := 11$
$r_{10}(X)$	abgelehnt	a_{10}

Multiversionsprotokolle

- Multiversionsprotokolle verwalten unterschiedliche Versionen eines Datenobjekts, also auch die mit alten Werten
- Bei Zugriff wird die jeweils passende Version des Datenobjekts an die Transaktion geliefert
- Umsetzungen mit
  - Zeitstempelordnung
  - Zertifizierungssperren

Multiversionsprotokoll mit Zeitstempelordnung

- Mehrere Versionen $X_1, \ldots, X_k$ von einem Datenobjekt X
- Für jedes X_i werden Zeitstempel gespeichert:
  - $TS_r(X_i)$: der größte (aktuellste) aller Zeitstempel von Transaktionen, die diese Version gelesen haben
  - $TS_w(X_i)$: Zeitstempel der Transaktion T, die den Wert dieser Version geschrieben hat

Multiversionsprotokoll mit Zeitstempelordnung

- Zwei Regeln für die Serialisierung:

1. Schreiben:
   - Wenn Transaktion T eine write_item(X)-Operation ausführen will und
     - Für die Version i von X mit dem größten $TS_w(X_i)$ aller Versionen von X gilt, dass
       - $TS_w(X_i) < TS(T)$ und (ältere Transaktion als T hat X_i geschrieben)
       - $TS_r(X_i) > TS(T)$, (jüngere Transaktion als T hat X_i gelesen)

     dann wird T abgebrochen / zurückgesetzt
   - Sonst
     - Erstelle eine neue Version X_{k+1} von X
     - Setze $TS_w(X_{k+1})$ und $TS_r(X_{k+1})$ auf $TS(T)$

Multiversionsprotokoll mit Zeitstempelordnung

- Zwei Regeln für die Serialisierung

2. Lesen:
   - Wenn Transaktion T eine $\text{read\_item}(X)$-Operation ausführen will,
     - Gibt es eine Version i von X mit dem größten $TS_w(X_i)$ aller Versionen von X, für die gilt
       - $TS_w(X_i) < TS(T)$,

     dann wird T der Wert von X_i geliefert und $TS_r(X_i)$ wird auf den größeren Wert von $TS(T)$ und $TS_r(X_i)$ gesetzt.

     $\rightarrow \text{read\_item}(X)$-Operationen können immer ausgeführt werden

Multiversionsprotokoll mit Zertifizierungssperren

- Standardfall: Wenn Transaktion T' eine Schreibsperre auf Objekt X hat, kann kein andere Transaktion T auf X zugreifen
- Beim Multiversionsprotokoll mit Zertifizierungssperren ist es T gestattet, X zu lesen, während eine Transaktion T' eine Schreibsperre auf X hält
  - Für jedes Objekt X sind zwei Versionen X', X'' zugelassen:
    - Version X' muss dabei immer von einer bestätigten Transaktion geschrieben worden sein
    - Version X'' wird erzeugt, wenn eine Transaktion T' eine Schreibsperre auf X anfordert
- Andere Transaktionen können die bestätigte Version von X, nämlich X', weiterhin lesen, während T' eine Schreibsperre erhält und mit X'' arbeitet
- Wenn T' bereit ist ein COMMIT durchzuführen, muss T' Zertifizierungssperren für *jedes* Objekt X anfordern, für das es eine *Schreibsperre* hat
  - Evtl. warten, bis andere Transaktionen ihre Lesesperren auf X freigeben

Multiversionsprotokoll mit Zertifizierungssperren

- Erweitertes Sperrkonzept mit Mehrfachmodus-Sperren
  - Statt Zeitstempelordnung
- Einführung eines neuen Zustands von Sperren, damit vier:
  - Lesegesperrt (S)
  - Schreibgesperrt (X)
  - Zertifizierungsgesperrt (C)
  - Entsperrt (0)
- Idee: Auch bei Lesesperre noch Schreiben und bei Schreibsperre noch Lesen zu erlauben, wenn dies geordnet geschieht
  - Kompatibilitätsmatrix rechts

Anforderung \ Zustand	0	S	X	C
S	✓	✓	✓, aber…	–
X	✓	✓, aber…	–	–
C	✓	–	–	–

Optimistische Verfahren

- Bisherige Verfahren: pessimistisch, Kontrolle vor Ausführung
- Optimistische Verfahren: während der Ausführung keine Kontrolle
  1. Lesephase
     - Bestätigte Datenobjekte der DB lesen
     - Aktualisierungen werden nur in lokale Kopien geschrieben
  2. Validierungsphase
     - Kontrollmechanismen werden angestoßen, die Serialisierbarkeit/Korrektheit (nachträglich) prüfen
  3. Schreibphase
     - Wenn Validierungsphase erfolgreich: Aktualisierungen werden auf der DB ausgeführt
     - Sonst: Transaktion wird abgebrochen und neu angestoßen
- Lohnt sich, wenn es nur selten Konflikte gibt
  - Z.B. bei großer DB → verteiltes Arbeiten (an unterschiedlichen Stellen der DB)

Granularität des Sperrens

- Datenobjekt kann alles sein, von Datenbank bis hin zu Datensatz (Zeile einer Tabelle)
- Granularität des Sperrens unterliegt Abwägung
- Sperren mit multipler Granularität

Sperren mit multipler Granularität

- Granularität von Sperren für jede Transaktion entscheiden (abhängig von Charakteristik)
  - Zeilenweise Sperre z.B. für

```
select *                    Q1
from Kunden
where Kunden_ID = 42
```

  - und eine Tabellen-Sperre für

```
select *                    Q2
from Kunden
```

- Wie können die Sperren für die Transaktionen koordiniert werden?
  - Für Q_2 sollen nicht für alle Tupel umständlich Sperrkonflikte analysiert werden

Vorhabens-Sperren

- DBs setzen Vorhabens-Sperren (*intention locks*) für verschiedene Sperrgranularitäten ein
  - Sperrmodus Intention Share (IS)
  - Sperrmodus Intention Exclusive (IX)
  - Kompatabilitätsmatrix rechts
- Eine Sperre I_ auf einer gröberen Ebene bedeutet, dass es eine Sperre _ auf einer niederen Ebene gibt

Anforderung \ Zustand	0	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Vorhabens-Sperren

- Protokoll für Sperren auf mehreren Ebenen:
  1. Eine Transaktion kann jede Ebene g in Modus _ $\in \{S, X\}$ sperren
  2. Bevor Ebene g in Modus _ gesperrt werden kann, muss eine Sperre I_ für alle gröberen Ebenen gewonnen werden
- Beispiel
  - Anfrage Q_1 würde anfordern
    - IS-Sperre für Tabelle Kunden (plus Tablespace + DB)
    - S-Sperre auf dem Tupel mit Kunden_ID=42
  - Anfrage Q_2 würde anfordern
    - (IS-Sperren für Tablespace + DB)
    - S-Sperre für die Tabelle Kunden

Q_1
```
select *
from Kunden
where Kunden_ID = 42
```

Q_2
```
select *
from Kunden
```

Anforderung \ Zustand	0	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Entdeckung von Konflikten

- Beispiel
  - Momentane Sperren auf Tabelle Kunden oder tiefer
    - Tabelle Kunden: IS von Q_1, S von Q_2
    - Tupel mit Kunden_ID=42: S von Q_1
  - Nehmen wir an, Anfrage Q_3 ist auch noch zu bearbeiten
  - Benötigte Sperren
    - IX-Sperre auf Tabelle Kunden (plus Tablespace + DB)
    - X-Sperre auf dem Tupel mit Kunden_ID=17

    → Kompatibel mit Q_1 (kein Konflikt zwischen IX und IS auf Tabellenebene)

    → Inkompatibel mit Q_2 (S-Sperre auf Tabellenebene von Q_2 steht in Konflikt mit der IX-Sperre bzgl. Q_3)

Q_1
```
select *
from Kunden
where Kunden_ID = 42
```

Q_2
```
select *
from Kunden
```

Q_3
```
update Kunden
set Name= 'John Doe'
where Kunden_ID = 17
```

Anforderung \ Zustand	0	S	X	IS	IX
S	✓	✓	–	✓	–
X	✓	–	–	–	–
IS	✓	✓	–	✓	✓
IX	✓	–	–	✓	✓

Konsistenzgarantien

- In einigen Fällen kann man mit einigen kleinen Fehlern im Anfrageergebnis leben
  - Fehler bezüglich einzelner Tupel machen sich in Aggregatfunktionen evtl. kaum bemerkbar
    - Lesen inkonsistenter Werte (inconsistent read Anomalie)
- Ab SQL-92 kann man Isolations-Modi spezifizieren:
  SET ISOLATION <MODE>
  - Verfügbare Modi: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE
    - Je weiter durch in der Liste, desto strikter (weniger Fehler)
    - Je weiter durch in der Liste, umso mehr Verwaltungsaufwand, z.B. für Sperren (weniger Durchsatz)

```
set isolation <mode>;
```

Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)

Es muss kein read lock erworben werden

S	T_1	T_2 (read uncommitted)	DB: $acct$
1	read_item($acct$)		1200
2	$acct := acct - 100$		1200
3	write_item($acct$)		1200
4		read_item($acct$)	1100
5		$acct := acct - 200$	1100
6	abort		1200
7		~~write_item($acct$)~~	1200

Read uncommited nur für lesende Transaktion

Isolations-Modi

- Read committed (auch ‘cursor stability‘)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
    - Transaktion sieht Daten, wie sie zum Zeitpunkt der aktuellen Operation comitted sind
  - Schreibsperren nach 2PL

S	T_1	T_2 (read committed)
1		$\text{read\_item}(A)$
2	$\text{write\_item}(A)$	
3	$\text{write\_item}(B)$	
4	commit	
5		$\text{read\_item}(B)$
6		$\text{read\_item}(A)$

Nur commited gelesen

Aber:
Unrepeatable Read
nicht ausgeschlossen

Isolations-Modi

- Repeatable read (auch ‘read stability‘)
  - Lesen und Schreiben mittels Lese- und Schreibsperren eines konsistenten Zustandes zu Beginn
    - Transaktion sieht Daten, wie sie zu Beginn der Transaktion (Sperrzuteilung) committed sind
  - Problem: Serialisierung nicht garantiert
    - Anfrage Q_1
      - ```
        select sum(Gehalt)
        from Mitarbeiter
        where AbtNr=5
        ```
    - Anfrage Q_2
      - ```
        insert into Mitarbeiter
            (SVN, NName, VName, Adresse, Gehalt, GebDatum, AbtNr, VorgesSVN)
        values ('90123456G789 ', 'Marini', 'Richard',
            '98 Oak Forest, Katy, TX', 37000, '30.12.1962‘, 5, '67890123D456‘)
        ```
    - Neue Daten erscheinen während der Ausführung von Q_1 → Phantom Read

Isolations-Modi

- Read uncommitted (auch: ‘dirty read‘ oder ‘browse‘)
  - Nur Schreibsperren akquiriert (nach 2PL)
- Read committed (auch ‘cursor stability‘)
  - Lesesperren nur halten, sofern Zeiger auf betreffendes Tupel zeigt
  - Schreibsperren nach 2PL
- Repeatable read (auch ‘read stability‘)
  - Lese- und Schreibsperren nach 2PL
- Serializable
  - Zusätzliche Sperranforderungen $I\_$, um Phantomproblem zu begegnen

Resultierende Konsistenzgarantien

Isolationsmodus	dirty read	non-repeatable read	phantom read
read uncommitted	möglich	möglich	möglich
read committed	–	möglich	möglich
repeatable read	–	–	möglich
serializable	–	–	–

- Einige Implementierungen unterstützen mehr, weniger oder andere Isolationsmodi
  - PostgreSQL: *predicate locks*, z.B. Bedingungen auf Spalten
    - Blockieren nicht, sondern markieren mögliche Abhängigkeiten zwischen Transaktionen, die bei Verdacht auf Anomalie abgebrochen werden
- Nur wenige Anwendungen benötigen (volle) Serialisierbarkeit



Sperren in Index-Strukturen

Sperrverwaltung

Nebenläufigkeit beim Indexzugriff

- Transaktion T_w führt zu Splitoperation
  - Beispiel: Einfügen eines Eintrags mit Schlüssel 6330
- Angenommen, Aufspaltung ist erfolgt, aber neue Verzeigerung hat noch nicht stattgefunden
- Weiterhin: nebenläufiges Lesen in Transaktion T_r, sucht nach 8050
  - Verzeigerung zeigt auf zweiten Blattknoten mit $5012{,}6330$ mit Zeigern auf dritten Blattknoten mit $8280{,}8404$
- Datensatz wird nicht gefunden → Sperren nötig





Sperren und B⁺-Baum-Indexe

- B⁺-Baum-Operationen
  - Für die Suche erfolgt ein Top-Down-Zugriff
  - Für Aktualisierungen
    - Suche
    - Daten in Blatt eintragen / löschen
      - Ggf. Aufspaltungen / Merge von Knoten nach oben
- Nach 2PL
  - Müssen Lese/Schreib-Sperren auf dem Weg nach unten akquiriert werden (Konversion provoziert ggf. Deadlocks)
  - Müssen alle Sperren bis zum Ende gehalten werden



→ Reduziert die Nebenläufigkeit drastisch

- Während des Indexzugriffs einer Transaktion müssen alle anderen Transaktionen warten, um die Sperre für die Wurzel des Index zu erhalten
- Wurzel wird zum Flaschenhals und serialisiert alle (Schreib-)Transaktionen

- 2PL nicht angemessen für B⁺-Bäume

Sperrprotokoll für B⁺-Bäume

- Protokoll Write-Only-Tree-Locking (WTL)
  1. Für alle Baumknoten n außer der Wurzel kann eine Sperre nur akquiriert werden, wenn die Sperre für den Elternknoten akquiriert wurde
     - Sperrkopplung
  2. Sobald ein Knoten entsperrt wurde, kann für ihn nicht erneut eine Sperre angefordert werden durch dieselbe Transaktion (2PL)

  → Garantiert Serialisierbarkeit
- Und damit gilt:
  - Alle Transaktionen folgen Top-Down-Zugriffsmuster
  - Keine Transaktion kann dabei andere überholen
  - WTL-Protokoll ist deadlockfrei

Aufspaltungssicherheit

- Wir müssen auf dem Weg nach unten in den B+-Baum Schreibsperren wegen möglicher Aufspaltungen halten
- Allerdings kann man leicht prüfen, ob eine Spaltung von Knoten n die Vorgänger überhaupt erreichen kann
  - Wenn n weniger als $2d$ Einträge enthält, kommt es nicht zu einer Weiterreichung der Aufspaltung nach oben
- Ein Knoten, der diese Bedingung erfüllt, heißt aufspaltungssicher (split safe)
- Ausnutzung zur frühen Sperrrückgabe
  - Wenn ein Knoten auf dem Weg nach unten als aufspaltungssicher gilt, können alle Sperren der Vorgänger zurückgegeben werden
  - Sperren werden weniger lang gehalten
  - Aufweichung des Zwei-Phasen-Sperrprotokolls

Sperrkopplungsprotokoll (Variante 1)

procedure read_lock(key)
    Place S lock on $root$
    $current \leftarrow root$
    while $current$ is not a leaf node **do**
        Find $child$ for key
        Place S lock on $child$
        Release S lock on $current$
        $current \leftarrow child$

procedure write_lock(key)
    Place X lock on $root$
    $current \leftarrow root$
    while $current$ is not a leaf node **do**
        Find $child$ for key
        Place X lock on $child$
        $current \leftarrow child$
        if $current$ is safe **then**
            Release all locks on ancestors of $current$

Erhöhung der Nebenläufigkeit

- Auch mit Sperrkopplung werden eine beträchtliche Anzahl von Sperren für innere Knoten benötigt
  - Mindert Nebenläufigkeit
- Innere Knoten selten durch Aktualisierungen betroffen
  - Wenn $d = 50$, dann Aufspaltung bei jeder 50. Einfügung (2% relative Auftretenshäufigkeit)
- Eine Einfügetransaktion könnte optimistisch annehmen, dass keine Aufspaltung nötig ist
  - Bei inneren Knoten werden während der Baumtraversierung nur Lesesperren akquiriert
    - Schreibsperre dann nur für das betreffende Blatt
  - Wenn die Annahme falsch ist, traversiere Indexbaum erneut unter Verwendung korrekter Schreibsperren

Sperrkopplungsprotokoll (Variante 2)

- Modifikationen nur für Schreibvorgänge

procedure optimistic_write_lock(key)
    Place S lock on $root$
    $current \leftarrow root$
    while $current$ is not a leaf node **do**
        Find $child$ for key
        if $child$ is a leaf **then**
            Place X lock on $child$
        else
            Place S lock on $child$
        Release lock on $current$
        $current \leftarrow child$
    if $current$ is unsafe **then**
        Release all locks and repeat with write_lock

procedure write_lock(key)
    Place X lock on $root$
    $current \leftarrow root$
    while $current$ is not a leaf node **do**
        Find $child$ for key
        Place X lock on $child$
        $current \leftarrow child$
        if $current$ is safe **then**
            Release all locks on ancestors of $current$

Diskussion

- Wenn eine Aufspaltung nötig ist, wird der Vorgang abgebrochen und erneut aufgesetzt
- Die resultierende Verarbeitung ist korrekt, obwohl es nach einem erneuten Sperren aussieht (was für WTL nicht erlaubt ist)
- Der Nachteil von Variante 2 ist, dass im Falle einer Blattaufspaltung Arbeit verloren ist
- Es gibt viele Varianten dieser Sperrprotokolle

Zwischenzusammenfassung

- Binäre Sperren: Datenobjekt ist gesperrt oder nicht; sehr restriktiv
- Zwei-Phasen-Sperrprotokoll
  - Grundlegendes Modell für viele Protokolle, die auf Sperren basieren
  - Keine Sperre mehr anfordern, nachdem eine erste Sperre freigegeben worden ist
- Zeitstempelbasierte Protokolle
- Multiversionsprotokolle: Verschiedene Versionen eines Datenobjekts
  - Umsetzung: zeitstempelbasiert oder mit Sperren
- Optimistische Verfahren: Änderungen nur lokal durchführen; Validierung um Änderungen persistent zu machen
- Granularitäten, Vorhaben-Sperren, Isolationsmodi
- Sperren in Index-Strukturen

Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
    - Sperren, Sperrprotokolle
    - Isolationsmodi
  - Wiederherstellungsverwaltung



Überblick: 7. Transaktionen

A. Transaktionsverarbeitung
- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

B. Sperrverwaltung
- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

C. Wiederherstellungsverwaltung
- Fehlersituationen
- Logging
- Recovery

TRANSAKTIONSFEHLER (1/3)

- Verschiedene Gründe, weshalb eine DB-Transaktion fehlschlagen kann
- Sieben typische Fehlerfälle
  - Mit in der Literatur uneinheitliche Bezeichnung
  - Meist in Kategorien untergliedert (z.B. System-, Medien-, Transaktionsfehler)

1. Systemabsturz [System- und Medienfehler]
   - Hardware-, Software- oder Netzwerkproblem tritt während der Transaktionsverarbeitung auf
   - Keine adäquate Handhabung durch die Software

   → „Systemabsturz“
2. Transaktionsabsturz [Transaktions-, System- und Medienfehler]
   - Einzelne Operationen schlagen fehl, z.B. aufgrund einer Division durch den Wert Null, aufgrund fehlerhafter Parameterwerte oder aus Programmfehlern
   - Benutzer bricht Transaktion (willkürlich) ab

WEITERE TRANSAKTIONSFEHLER (2/3)

3. Lokale Fehler: [Transaktionsfehler]
   - Während der Transaktionsausführung können Zustände auftreten, die einen Transaktionsabbruch notwendig machen
     - Beispiel: zur Transaktionsausführung benötigte Daten können nicht ermittelt werden
4. Speicherfehler: [Medienfehler]
   - Partieller oder vollständiger Ausfall eines Speichersystems

   → Lese-/Schreibfehler bei der Transaktionsausführung
5. „Katastrophen“: [Medienfehler]
   - Subsumierung verschiedener Ausnahmesituationen, z.B. physischer Verlust eines Datenträgers durch Brand oder Diebstahl
   - Ausnahmesituationen, die sich aus einer fehlerhaften Bedienung des Systems ergeben, wobei z.B. ein Datenträger versehentlich gelöscht wurde

WEITERE TRANSAKTIONSFEHLER (3/3)

6. Transaktionsannullierung: [kein klassischer Fehlerfall]
   - Ein bestimmter DB-Zustand kann dazu führen, dass eine Transaktion „aus inhaltlichen Gründen“ nicht weiter ausgeführt wird, falls z.B. kein genügend hohes Guthaben für eine Abbuchung vorliegt
7. Nebenläufigkeitskontrolle: [kein klassischer Fehlerfall]
   - Komponente zur Überwachung der nebenläufigen Transaktionsausführung veranlasst Unterbrechung (nicht Abbruch) einer Transaktion, um Interferenzen mit anderen Transaktionen zu vermeiden
     - Gestoppte Transaktion wird später wieder gestartet

- Die aufgezählten und weitere Fehlerfälle kann man durch nachfolgend beschriebene, einfache Basis-Mechanismen zu „reparieren“ versuchen

Beispiel: System- oder Medienfehler

- Transaktionen T_1, T_2, T_5 wurden vor dem Ausfall erfolgreich beendet
  → Dauerhaftigkeit: Es muss sicher-gestellt werden, dass die Effekte beibehalten werden oder wiederhergestellt werden können (*redo*)
- Transaktionen T_3, T_4 wurden noch nicht beendet
  → Atomarität: Alle Effekte müssen rückgängig gemacht werden (*undo*)



Wiederherstellungsverwaltung: Aufgabe

- Zentrale Komponente der Transaktionsverarbeitung
- Sichert Atomarität und Dauerhaftigkeit
- Koordiniert transaktionsübergreifend die Abarbeitung der DB-Operationen:
  - BEGIN_TRANSACTION und END_TRANSACTION
  - COMMIT und ABORT
- Bietet eigene Primitive für Recovery nach Fehlersituationen:
  - Warm restart
  - Cold restart
- Grundlage zur praktischen Umsetzung dieser Funktionalität sind Aufbau und Verwaltung eines Log-File



Logging

Wiederherstellungsverwaltung

Organisation des Log-File

- Sequentielle Struktur, die DB-Änderungen chronologisch speichert
  - Liegt auf persistentem Speicher
  - Verwaltung: Wiederherstellungsverwalter
- EOT und Commit fallen zusammen
  - Commit heißt jetzt erstmal nur logisches Ende (vorher EOT), aber nicht zwangsweise persistente Speicherung
    - Der Dateiverwaltung soll überlassen werden, wann was geschrieben wird
    - Transaktionen, die einen Commit Eintrag haben, sollen bei der Wiederherstellung wiederholt werden, wenn nicht im Speicher
- Zwei Typen von Log-Einträgen:
  1. Transaktionsbezogene Einträge:
     - BS / AS: Wert von X vor / nach T (*before / after state*)
     - BEGIN_TRANSACTION : $B(T)$
     - INSERT_ITEM : $I(T, X, AS)$
     - DELETE_ITEM : $D(T, X, BS)$
     - UPDATE_ITEM : $U(T, X, BS, AS)$
     - COMMIT : $C(T)$
     - ABORT : $A(T)$
  2. Systembezogene Einträge:
     - DUMP (vollständige DB-Kopie; eher selten)
     - CHECKPOINT (partielle Aktualisierung; häufig)

Organisation des Log-File

- Transaction Record:
  - Zusammenfassung der INSERT_ITEM-, UPDATE_ITEM- und DELETE_ITEM-Operationen einer einzelnen Transaktion
  - Beginnt mit (B)egin, endet mit (C)ommit oder (A)bort
  - Beispiel:



Log-Information COMMIT und Fehler

- COMMIT-Point einer Transaktion T bezeichnet den Zeitpunkt, an dem alle DB-Zugriffsoperationen von T erfolgreich ausgeführt und im Log erfasst wurden
  - Zu diesem Zeitpunkt gilt Transaktion als bestätigt, ihre Wirkung soll persistent in der DB gespeichert werden

  → [COMMIT, T]-Eintrag im Log wird generiert, als Zeichen der logischen, aber noch nicht physischen Beendigung der Transaktion
- Im Fehlerfall
  - Transaktionen T mit BEGIN_TRANSACTION und ohne COMMIT werden – soweit nötig – rückgängig gemacht → UNDO
  - Transaktionen T mit BEGIN_TRANSACTION und mit COMMIT können bzgl. ihrer Änderungsoperationen vollständig wiederholt werden, falls der DB-Zustand noch nicht persistent auf Platte gesichert wurde → REDO

UNDO

- Abbruch einer Transaktion T
- DB muss in Zustand vor Transaktionsausführung gebracht werden
- Log rückwärts bis zum Beginn von T durchsuchen, um Operationen von T zu identifizieren und betroffene Datenobjekte X zurückzusetzen
- Beispiel: UNDO für ein Datenobjekt X
  - UPDATE, DELETE: kopiere den Wert BS in X
  - INSERT: lösche das Objekt X



REDO

- Systemfehler in der DB
- Alle bestätigten Transaktionen T, die noch nicht im Hintergrundspeicher sind, müssen wiederholt werden
- Log vom Beginn von T vorwärts durchsucht und alle auftretenden Operationen erneut realisiert
- Beispiel: REDO für ein Datenobjekt X
  - INSERT, UPDATE:
    - Kopiere den Wert AS in X
  - DELETE:
    - Lösche das Objekt X



Log-Information: Sicherer Speicher

- Um während des DBMS-Betriebs eine verlässliche Grundlage für UNDO/REDO im Fehlerfall zu liefern, muss das Log – neben der Protokollierung im Hauptspeicher (intern) – auch verlässlich auf Platte (extern) gespeichert werden
  - Nach Systemabsturz können nur solche Log-Einträge bei der Fehlerbehandlung berücksichtigt werden, die bereits auf Platte gespeichert sind – Hauptspeicherinhalte stehen u.U. nicht mehr zur Verfügung
  - Nicht jeder Log-Eintrag wird direkt auch auf Platte gespeichert, um hohe Verzögerungszeiten beim Log-Zugriff zu vermeiden
- Forcewriting
  - Transaktion T kann erst dann ihren COMMIT-Point erreichen, wenn der T zugehörige Log-Inhalt verlässlich auf Platte gespeichert ist

Checkpoints

- Zeichnet alle aktiven Transaktionen auf
- Aktualisiert Hintergrundspeicher partiell aufgrund abgeschlossener Transaktionen
- Wird periodisch angestoßen
  - Währenddessen keine COMMIT-Anweisungen für aktive Transaktionen
- Dienst-Ende: es wird synchron (*force*) ein CHECKPOINT-Eintrag im Log-File generiert wird
  - DB-Veränderungen abgeschlossener Transaktionen dauerhaft in die DB einfügen
- Checkpoint-Eintrag:
  - $CK(T_1, T_2, T_3, \ldots, T_n)$, enthält die Identifier T_i der aktiven Transaktionen
- Bei einem Recovery-Vorgang muss dann nur bis zum letzten Checkpoint-Eintrag zurückgegangen werden und von dort an die aktiven Transaktionen abhandeln (UNDO/REDO)

DUMP

- Vollständige Kopie der DB
  - Backup
- Wird in der Regel erzeugt, wenn DB nicht operativ ist
  - (z.B. einmal pro Nacht)
- Wird im Allgemeinen im persistenten Speicher abgelegt
- Nach Erzeugung des DUMP entsprechender Eintrag im Log-File

Grundregeln für Log-Einträge

- Zwei Grundregeln:
  - Write-Ahead Log (WAL)
    BS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die entsprechende Operation auf der DB ausgeführt wird
  - Commit-Precedence (CP)
    AS-Teile der Log-Einträge müssen im Log-File gespeichert sein, bevor die Transaktion abgeschlossen wird
- Damit führt Verlust des Hauptspeichers nicht zu Datenverlust
- Mögliche Situationen
  - Aktualisierung auf Hintergrundspeicher vor dem COMMIT → Kein REDO bei Recovery nötig
  - COMMIT vor Aktualisierung auf Hintergrundspeicher → Kein UNDO bei Recovery nötig

Beispiele

Was geschieht bei Abbrüchen?



Recovery

Wiederherstellungsverwaltung



(Ideales) Fail-Stop-Model eines DBMS



Boot: Warm / Cold Restart

- Fehlersituationen beim Datenmanagement …
  - Systemfehler
    - Softwarefehler (z.B. Betriebssystemfehler) oder
    - Fehler von Geräten (z.B. aufgrund von Fehlern in der Stromversorgung).
    - Hauptspeicher geht verloren, Hintergrundspeicher bleibt erhalten
  - Gerätefehler
    - Fehler des Hintergrundspeichers (z.B. aufgrund eines „Platten-Crashes“)
    - Hauptspeicher und Hintergrundspeicher gehen verloren
    - Stabiler Speicher bleibt erhalten (Band, RAID-System)
- Protokolle zum Restart des DBMS bzw. der DB
  - Warm Restart ist im Fall von Systemfehlern geeignet
  - Cold Restart ist im Fall von Gerätefehlern geeignet

Restart (Boot)

- Zustand von Transaktionen vor Stop
  - Abgeschlossen
    - Resultat abgeschlossener Transaktionen ist im stabilen Speicher gespeichert
  - COMMIT vorhanden, aber u.U. nicht abgeschlossen:
    - DB-Veränderungen der Transaktion müssen wiederholt werden
  - Kein COMMIT vorhanden
    - DB-Veränderungen der Transaktion müssen verworfen werden

Warm Restart

- Phase 1: Suche im Log-File die Position des jüngsten CHECKPOINT-Eintrags
- Phase 2: Bilde UNDO- und REDO-Menge
  - UNDO-Menge: Transaktionen, die verworfen werden müssen
  - REDO-Menge: Transaktionen, die reproduziert werden können
- Phase 3: Behandle Transaktionen der UNDO-Menge
  - Durchlaufe das Log-File zurück zur Position der ersten Operation der ältesten Transaktion aus der UNDO- *und* der REDO-Menge
  - Führe UNDO für alle Operationen der Transaktionen der UNDO-Menge aus
- Phase 4: Behandle Transaktionen der REDO Menge
  - Durchlaufe das Log-File vorwärts und führe REDO für alle Operationen der Transaktionen in der REDO-Menge aus
- Vorgehen sichert Atomarität und Dauerhaftigkeit

Cold Restart

- Phase 1:
  - Der aktuellste DUMP wird genutzt, um die verlorenen bzw. zerstörten Teile der DB wieder neu im Hintergrundspeicher anzulegen
- Phase 2:
  - Das Log-File wird vorwärts durchlaufen und die dort vermerkten Operationen werden auf der Rekonstruktion der DB neu ausgeführt
- Phase 3:
  - Ein Warm Restart wird initiiert
- Auch dieses Vorgehen sichert Atomarität und Dauerhaftigkeit

Zusammenfassung

- Fehlersituationen
  - System-, Medien-, Transaktionsfehler
  - Dauerhaftigkeit, Atomarität möglicherweise verletzt
- Logging
  - Log-Information
  - Konzept und Organisation von Log-Files
  - Inhalt und Nutzung bei Fehlern
- Recovery
  - Wiederherstellung eines DBMS
    - Warm restart (Systemfehler) unter Nutzung des letzten Checkpoints
    - Cold restart (Gerätefehler) unter Nutzung des letzten Dumps

Architektur eines DBMS

- Speicherung
- Anfrageverarbeitung
- Transaktionsmanagement
  - Transaktionsverwaltung
  - Sperrverwaltung
  - Wiederherstellungsverwaltung
    - Logging
      - Undo, Redo
      - Checkpoints, Dump
    - Recovery:
      - Warm / Cold Restart



Überblick: 7. Transaktionen

A. Transaktionsverarbeitung
- Fehlersituationen
- Schedules: Korrektheit, Serialisierbarkeit, Äquivalenzen

B. Sperrverwaltung
- Sperren, Sperrprotokolle
- Deadlocks
- Weitere Methoden zur Mehrbenutzerkontrolle

C. Wiederherstellungsverwaltung
- Fehlersituationen
- Logging
- Recovery

→ Verteilte Datenbanken

WWU
MÜNSTER



Verteilte Datenbanken

Datenbanken

wissen.leben

Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Erweiterung**
   - ~~Noch offen:~~ verteilte DBs, ~~deduktive DBs (DataLog → Logik-Verbindung), XML, Graph-DBs~~

Motivation für verteilte Datenbanken

- Verteilung / Dezentralisierung
  - Vor allem bei Neuentwicklungen
  - (Verteilte) Realisierung von Anwendungen auf Basis verteilter (DB-)Systeme
  - (Kontrolliert) verteilte Datenspeicherung zur Lastverteilung, Nutzung von Parallelität Verkürzung von Antwortzeiten, Erhöhung der Ausfallsicherheit



- Integration
  - Bei bestehenden Systemen
  - Verteilt gespeicherte und unabhängig voneinander verwaltete, aber inhaltlich zusammengehörige Daten logisch zusammenbringen
  - Einheitlichen Zugriff auf Gesamtdatenbestand ermöglichen



Überblick: 8. Verteilte Datenbanken

A. Verteilte DBMS

- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

B. Anfragenbeantwortung in verteilten Systemen

- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

C. Föderierte DBS

- Integration
- Migration

Verteiltes DBMS (VDBMS)

- Sammlung von Informationseinheiten, verteilt auf mehreren Rechnern, verbunden mittels Kommunikationsnetz (nach Ceri & Pelagatti, 1984)
- Kooperation zwischen autonom arbeitenden Stationen, zur Durchführung einer globalen Aufgabe



Aufbau eines VDBMS

- Globales Schema
  - Relationales Schema wie zuvor
- Fragmentierungsschema
  - Verteilung der Daten in Fragmente
- Zuordnungsschema
  - Physische Zuordnung (Allokation) der Fragmente auf lokale DBs
  - Möglicherweise mit Replikation
- Lokales Schema / DBMS / DB
  - DB: Menge der lokal gespeicherten Daten
  - DBMS: Lokales System zur Verwaltung und Anfragebeantwortung
  - Schema: Lokale Sicht auf die Daten in lokaler DB



Fragmentierung

- Ziel: Verteilung von Daten
- Arten der Fragmentierung
  - Horizontal: Entlang der Tabellenzeilen
    - Aufteilung durch Selektion
  - Vertikal: Entlang der Tabellenspalten
    - Aufteilung durch Projektion
  - Gemischt: Verschachtelung von Fragmentierung
    - Aufteilung durch verschachtelte Selektion und Projektion
  - Abgeleitet: Anhand von Bedingungen
    - Z.B. über Fremdschlüsselrelation



Horizontale Fragmentierung



Vertikale Fragmentierung



Gemischte Fragmentierung

Beispiel für horizontale Fragmentierung

$$Profs = PhilProfs \cup PhysProfs \cup TheoProfs$$

$$PhilProfs = \sigma_{Fakultaet='Philosophie'}(Profs)$$

$$TheoProfs = \sigma_{Fakultaet='Theologie'}(Profs)$$

$$PhysProfs = \sigma_{Fakultaet='Physik'}(Profs)$$

Profs

PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
2125	Sokrates	W3	226	Philosophie	85000	1
2126	Russel	W3	232	Philosophie	80000	3
2127	Kopernikus	W2	310	Physik	65000	5
2133	Popper	W2	52	Philosophie	68000	1
2134	Augustinus	W2	309	Theologie	55000	5
2136	Curie	W3	36	Physik	95000	3
2137	Kant	W3	7	Philosophie	98000	1

PhilProfs

PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
2125	Sokrates	W3	226	Philosophie	85000	1
2126	Russel	W3	232	Philosophie	80000	3
2133	Popper	W2	52	Philosophie	68000	1
2137	Kant	W3	7	Philosophie	98000	1

TheoProfs

PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
2134	Augustinus	W2	309	Theologie	55000	5

PhysProfs

PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
2127	Kopernikus	W2	310	Physik	65000	5
2136	Curie	W3	36	Physik	95000	3

Beispiel für vertikale Fragmentierung

$$Profs = ProfVerw \bowtie Pfs$$

Profs

PersNr	Name	Rang	Raum	Fakultaet	Gehalt	St.Klasse
2125	Sokrates	W3	226	Philosophie	85000	1
2126	Russel	W3	232	Philosophie	80000	3
2127	Kopernikus	W2	310	Physik	65000	5
2133	Popper	W2	52	Philosophie	68000	1
2134	Augustinus	W2	309	Theologie	55000	5
2136	Curie	W3	36	Physik	95000	3
2137	Kant	W3	7	Philosophie	98000	1

$$ProfVerw = \pi_{PersNr,Name,Gehalt,Steuerklasse}(Profs)$$

ProfVerw

PersNr	Name	Gehalt	St.Klasse
2125	Sokrates	85000	1
2126	Russel	80000	3
2127	Kopernikus	65000	5
2133	Popper	68000	1
2134	Augustinus	55000	5
2136	Curie	95000	3
2137	Kant	98000	1

$$Pfs = \pi_{PersNr,Name,Rang,Raum,,Fakultaet}(Profs)$$

Pfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2127	Kopernikus	W2	310	Physik
2133	Popper	W2	52	Philosophie
2134	Augustinus	W2	309	Theologie
2136	Curie	W3	36	Physik
2137	Kant	W3	7	Philosophie

Beispiel für gemischte Fragmentierung

$$Profs = ProfVerw \bowtie (PhilPfs \cup PhysPfs \cup TheoPfs)$$

$$ProfVerw = \pi_{PersNr,Name,Gehalt,Steuerklasse}(Profs)$$

$$Pfs = \pi_{PersNr,Name,Rang,Raum,,Fakultaet}(Profs)$$

$$PhilPrs = \sigma_{Fakultaet='Philosophie'}(Pfs)$$

$$TheoPfs = \sigma_{Fakultaet='Theologie'}(Pfs)$$

$$PhysPfs = \sigma_{Fakultaet='Physik'}(Pfs)$$

Pfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2127	Kopernikus	W2	310	Physik
2133	Popper	W2	52	Philosophie
2134	Augustinus	W2	309	Theologie
2136	Curie	W3	36	Physik
2137	Kant	W3	7	Philosophie

PhilPfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

TheoPfs

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

PhysPfs

PersNr	Name	Rang	Raum	Fakultaet
2127	Kopernikus	W2	310	Physik
2136	Curie	W3	36	Physik

Abgeleitete Fragmentierung über Fremdschlüsselbeziehung

- Relation S wird auf Basis der Fragmentierung der Relation R abgeleitet (fragmentiert)
  - Fremdschlüssel B in S auf Primärschlüssel A in R
- R durch Selektion horizontal in Fragmente R_i zerlegt
  - Referenzpartitionierung für S
- S aufgrund Fremdschlüsselbeziehungen partitioniert
  - $S_i = S \ltimes_{S.B=A} (\pi_A(R_i))$
- $R_i.A$ bilden disjunkte Menge
  $\rightarrow$ auch S_i disjunkt





Beispiel für abgeleitete Fragmentierung

- Horizontale Fragmentierung der Relation Pfs
  - Entspricht der Relation R
  - Disjunkte Mengen der Primärschlüssel $PersNr$ von $PhilPfs, TheoPfs, PhysPfs$

Pfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2127	Kopernikus	W2	310	Physik
2133	Popper	W2	52	Philosophie
2134	Augustinus	W2	309	Theologie
2136	Curie	W3	36	Physik
2137	Kant	W3	7	Philosophie

$\pi_{PersNr}(PhilPfs)$

PersNr
2125
2126
2133
2137

PhilPfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

$\pi_{PersNr}(TheoPfs)$

PersNr
2134

TheoPfs

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

$\pi_{PersNr}(PhysPfs)$

PersNr
2127
2136

PhysPfs

PersNr	Name	Rang	Raum	Fakultaet
2127	Kopernikus	W2	310	Physik
2136	Curie	W3	36	Physik

Beispiel für abgeleitete Fragmentierung

- Horizontale Fragmentierung der Relation Pfs
- Ableitung der Fragmentierung der Relation $Vorlesungen$
  - $Vorlesungen$ entspricht S, $Dozierende$ Fremdschlüssel auf $PersNr$

$\pi_{PersNr}(PhilPfs)$

PersNr
2125
2126
2133
2137

$\pi_{PersNr}(TheoPfs)$

PersNr
2134

$\pi_{PersNr}(PhysPfs)$

PersNr
2127
2136

Vorlesungen

VorlNr	Titel	SWS	Dozierende
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5022	Glaube und Wissen	2	2134
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
6001	Grundlagen	3	2127
6003	Thermodynamik	4	2136
6010	Statik	4	2127

Beispiel für abgeleitete Fragmentierung

- Abgeleitete Fragmentierung:

Vorlesungen

VorlNr	Titel	SWS	Dozierende
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5022	Glaube und Wissen	2	2134
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
6001	Grundlagen	3	2127
6003	Thermodynamik	4	2136
6010	Statik	4	2127

PhilVorl

VorlNr	Titel	SWS	Doziernde
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133

TheoVorl

VorlNr	Titel	SWS	Doziernde
5022	Glaube und Wissen	2	2134

PhysVorl

VorlNr	Titel	SWS	Doziernde
6001	Grundlagen	3	2127
6003	Thermodynamik	4	2136
6010	Statik	4	2127

PhilPfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

TheoPfs

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

PhysPfs

PersNr	Name	Rang	Raum	Fakultaet
2127	Kopernikus	W2	310	Physik
2136	Curie	W3	36	Physik

Beispiele Fragmentierungen: Baumdarstellung



$$Profs = ProfVerw \bowtie (PhilPfs \cup PhysPfs \cup TheoPfs)$$
$$Vorlesungen = PhilVorl \cup PhysVorl \cup TheoVorl$$



Fragmentierung: Korrektheitsanforderungen

- Korrektheit der Fragmentierung
  - *Rekonstruierbarkeit*
    - Original-Relation lässt sich aus den Fragmenten wiederherstellen
  - *Vollständigkeit*
    - Jedes Tupel ist einem Fragment zugeordnet
  - *Disjunktheit*
    - Die Fragmente überlappen sich nicht

- Bisher betrachtete Fragmentierungen
  - Horizontal
    - Selektion darf keine Tupel auslassen
    - Dann rekonstruierbar, vollständig, disjunkt
  - Vertikal
    - Projektion darf keine Attribute auslassen
    - Zur Rekonstruktion der Originalrelation, Übernahme der Primärschlüssel in die Fragmente notwendig
    - Dann rekonstruierbar, vollständig, aber nicht mehr disjunkt (Primärschlüssel vielfach)
  - Abgeleitet und gemischt
    - Selbe Überlegungen wie oben

Fragmentierung

- Motivation
  - Lastverteilung
  - Nutzung von Parallelität zur Verkürzung von Antwortzeiten
- Art der Fragmentierung abhängig von häufigen Anfragen und Zugriffsmustern
- Herausforderungen
  - DBS muss Verteilung der Daten kennen und bei Anfragen berücksichtigen
    - An welche DB muss welche Teilanfrage?
    - Wie müssen Ergebnismengen zusammengefügt werden?
    - Was passiert, wenn eine DB ausfällt?



Replikation

- (Logisches) Duplizieren von Daten nach strategischen Gesichtspunkten
- Motivation: Erhöhung der
  - Effizienz (schnellerer Zugriff „vor Ort statt entfernt“)
  - Ausfallsicherheit und Verfügbarkeit (Replikat als „Sicherungskopie“)
  - Autonomie (Unabhängigkeit von ansonsten nur einmal verfügbaren Daten)
- Zielkonflikte bei der Replikation
  - Deutliche Erhöhung der Zugriffseffizienz, Verfügbarkeit und Autonomie
    → Große Anzahl von Replikaten auf möglichst vielen Knoten
  - Erhaltung der Datenkonsistenz
    → Alle Replikate möglichst synchron aktualisieren
  - Erhöhung der Effizienz bei Änderungsoperationen
    → Wenige Replikate wünschenswert; schneller synchron aktualisierbar

Allokation

- (Physische) gezielte Zuordnung von Daten auf Rechner
- Die (physisch orientierte) Allokation legt fest,
  - Auf welchem Rechner ein Fragment gehalten wird
  - Welche Fragmente auf welchen Rechnern repliziert gespeichert werden
- Physische Umsetzung der Überlegung aus Fragmentierung und Replikation
- Bei der Allokation zu berücksichtigende Aspekte:
  - Effizienz
    - Minimierung von Zugriffskosten für Remote-Zugriffe
    - Vermeidung von Flaschenhälsen (Übertragungskapazität im Netz, Leistung einzelner Rechenknoten, ...)
  - Datensicherheit
    - Auswahl von Knoten hinsichtlich Verlässlichkeit
    - Redundante Speicherung von Daten

Fortsetzung Beispiel

- Allokation ohne Replikation

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Philosophie	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Physik	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}

- Allokation mit Replikation, Beispiele:
  - Erhöhung der Ausfallsicherheit: Zweite St_{Verw}-Station
  - Prüfungsamt benötigt Vorlesungsinformation: Weitere Station St_{Vorl} mit den wiedervereinigten Vorlesungsfragmenten

Transparenz in verteilten Datenbanken

- Grad der Unabhängigkeit, den ein VDBMS dem Benutzer beim Zugriff auf verteilte Daten vermittelt
  - Verschiedene Stufen:
    - Fragmentierungstransparenz: Nutzer stellen Anfragen an das globale Schema, haben kein Wissen über Fragmentierung / Allokation
      - VDBMS muss Anfragen / Änderungen korrekt auf Fragmenten und in Stationen umsetzen
    - Allokationstransparenz: Nutzer müssen die Fragmentierung kennen, aber nicht die Allokation
      - Anfragen an Fragmente/lokale Schemata
    - Lokale Schema-Transparenz: Nutzer müssen Station kennen, auf dem Fragment liegt
      - Transparenz bezieht sich darauf, dass zumindest alle Stationen dasselbe Datenmodell und dieselbe Anfragesprache verwenden



Fragmentierungstranparenz: Beispiel

- Anfrage, die Fragmentierungstransparenz voraussetzt:
  - `select Titel, Name from Vorlesungen, Profs where Dozierende = PersNr;`
  - Jeweils ein Join von *Vorl mit *Pfs auf der jeweiligen lokalen Station
  - Vereinigung der Join-Ergebnisse auf der Station, auf der die Anfrage liegt

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Physik	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Philosophie	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}

PhilPfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

PhilVorl

VorlNr	Titel	SWS	Doziernde
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133

Fragmentierungstranparenz: Beispiel

- Änderungsoperation, die Fragmentierungstransparenz voraussetzt:
  - `update Professoren set Fakultät='Theologie' where Name='Sokrates';`
  - Transferieren des Tupels aus Fragment PhilPfs in das Fragment TheoPfs
    - Löschen aus PhilPfs, Einfügen in TheoPfs
  - Ändern der abgeleiteten Fragmentierung von Vorlesungen
    - Einfügen der von Sokrates gehaltenen Vorlesungen in TheoVorl, Löschen aus PhilVorl

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Physik	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Philosophie	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}

PhilPfs

PersNr	Name	Rang	Raum	Fakultaet
2125	Sokrates	W3	226	Philosophie
2126	Russel	W3	232	Philosophie
2133	Popper	W2	52	Philosophie
2137	Kant	W3	7	Philosophie

PhilVorl

VorlNr	Titel	SWS	Doziernde
4052	Logik	4	2125
4630	Die 3 Kriterien	4	2137
5049	Mäeutik	2	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133

Allokationstransparenz: Beispiel

- Nutzer*innen müssen Fragmentierung kennen, aber nicht Aufenthaltsort von Fragmenten
  - Beispielanfrage:
    - ```
      select Gehalt
      from ProfVerw
      where Name='Sokrates';
      ```
  - Unter Umständen muss Originalrelation rekonstruiert werden
    - Beispiel: Verwaltung möchte wissen, wieviel die W3-Professoren der Theologie insgesamt verdienen
      - ```
        select sum(Gehalt)
        from ProfVerw, TheoPfs
        where ProfVerw.PersNr=TheoPfs.PersNr and Rang='W3';
        ```

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Physik	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Philosophie	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}

ProfVerw

PersNr	Name	Gehalt	St.Klasse
2125	Sokrates	85000	1
2126	Russel	80000	3
2127	Kopernikus	65000	5
2133	Popper	68000	1
2134	Augustinus	55000	5
2136	Curie	95000	3
2137	Kant	98000	1

TheoPfs

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

Lokale Schema-Transparenz

- Der Benutzer muss auch noch den Rechner kennen, auf dem ein Fragment liegt
  - Beispielanfrage:
    - ```
      select Name
      from TheoPfs at St_Theo
      where Rang='W2';
      ```
  - Ist überhaupt Transparenz gegeben?
    - Lokale Schema-Transparenz setzt voraus, dass alle Rechner dasselbe Datenmodell und dieselbe Anfragesprache verwenden.
      - Anfrage kann somit analog auch an Station St_{Phil} ausgeführt werden
      - Nicht möglich bei Kopplung unterschiedlicher DBMS

Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Physik	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Philosophie	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}

ProfVerw

PersNr	Name	Gehalt	St.Klasse
2125	Sokrates	85000	1
2126	Russel	80000	3
2127	Kopernikus	65000	5
2133	Popper	68000	1
2134	Augustinus	55000	5
2136	Curie	95000	3
2137	Kant	98000	1

TheoPfs

PersNr	Name	Rang	Raum	Fakultaet
2134	Augustinus	W2	309	Theologie

Verwendung grundsätzlich verschiedener Datenmodelle auf lokalen DBMS nennt man Multi-Database-Systems (oft unumgänglich in realer Welt)
      ```

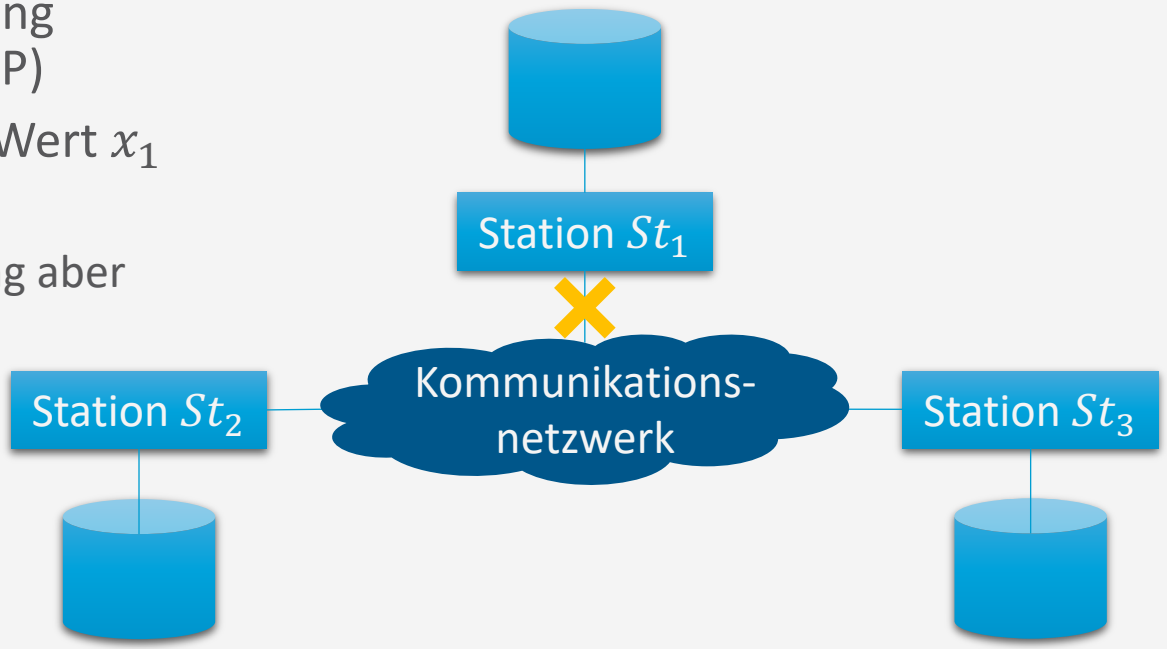
Konsistenzmodell: CAP

- Neben Transparenz, weitere Anforderungen an verteilte DBs:
  Konsistenz, Verfügbarkeit, Ausfalltoleranz
  - Consistency, Availability, Partition tolerance = CAP
  - *Konsistenz*: Alle Knoten liefern identische Daten
  - *Verfügbarkeit*: Akzeptable Reaktionszeit (Station muss antworten)
  - *Ausfalltoleranz*: Knoten / Kommunikationsverbindung kann ausfallen
- CAP-Theorem:
  Nur zwei der drei Anforderungen in verteilten Systemen erfüllbar
  - Vermutung von Brewer (2000)
  - Beweis von Gilbert und Lynch (2002)



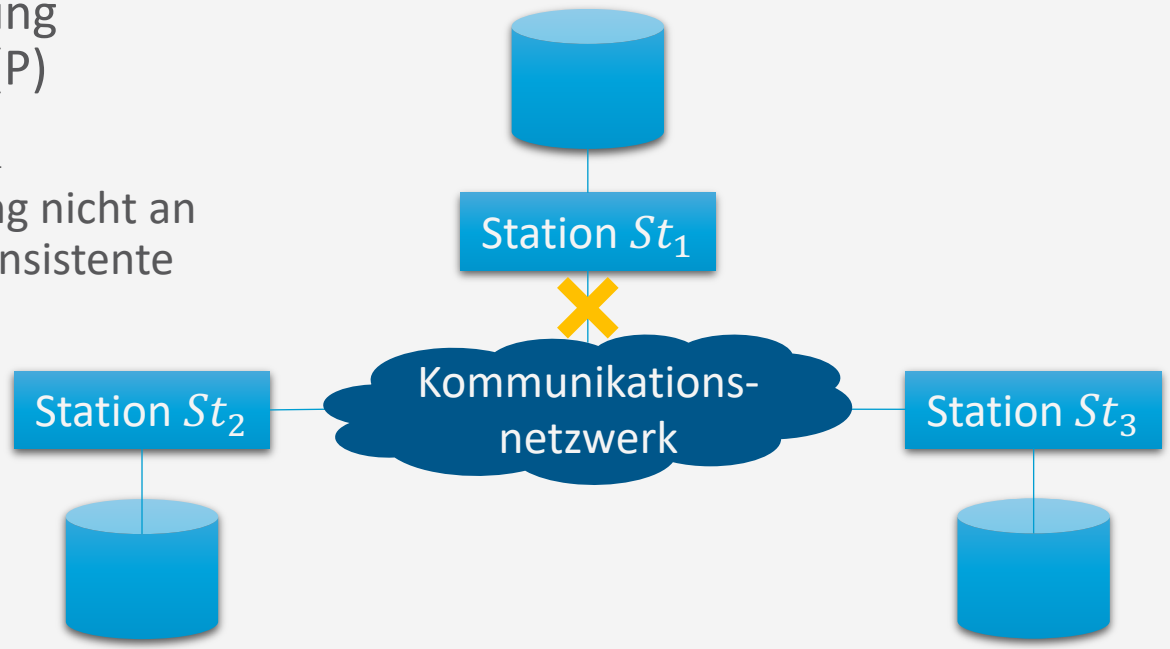
CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
  1. AP geht nicht mit C
     - Annahme: System ist partitioniert (keine Verbindung zwischen St_1 und St_2), was nach zu tolerieren ist (P)
     - Anfrage an Station St_1 Datum X zu schreiben mit Wert x_1
       - Da das System verfügbar ist (A), muss St_1 antworten
       - Da das System partitioniert ist, kann St_1 die Änderung aber nicht an die anderen Stationen weitergeben
     - Dann Anfrage an Station St_2 Datum X zu lesen
       - Da das System verfügbar ist (A), muss St_2 antworten
       - Da das System partitioniert ist, kann St_2 nur den veralteten Wert x_0 zurückgeben
         → Inkonsistenz (kein C)



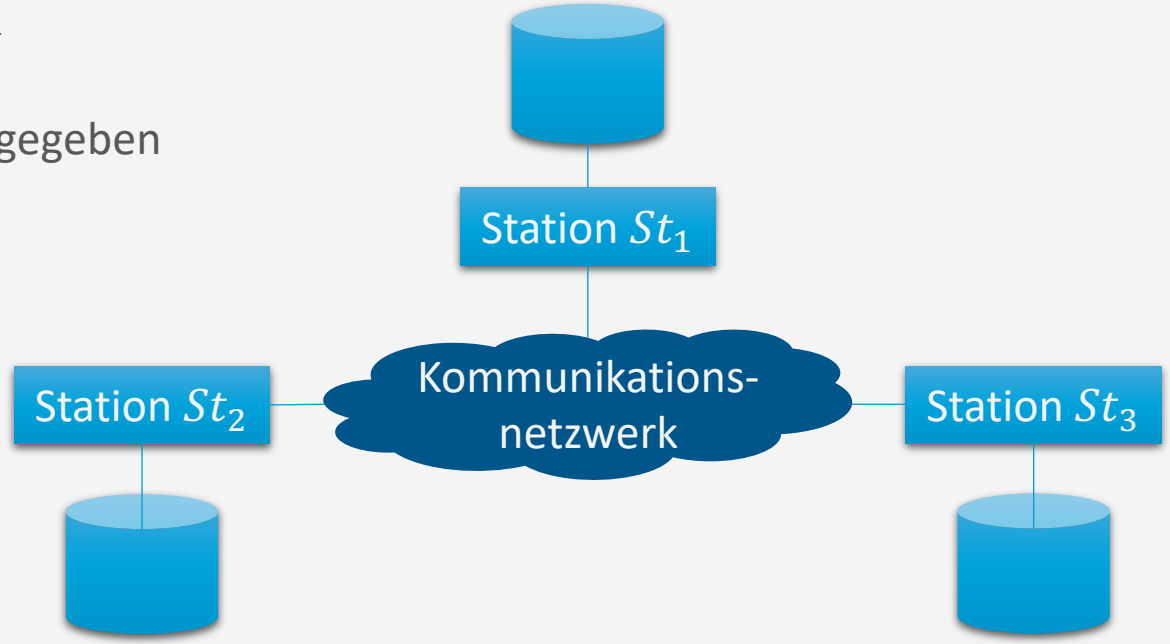
CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
  2. CP geht nicht mit A
     - Annahme: System ist partitioniert (keine Verbindung zwischen St_1 und St_2), was nach zu tolerieren ist (P)
     - Anfrage an Station St_1 X zu schreiben mit Wert x_1
       - Da das System partitioniert ist, kann St_1 die Änderung nicht an die anderen Stationen weitergeben, da es sonst inkonsistente Daten hätte (C erhalten)
       - St_1 muss die Anfrage ablehnen bzw. kann nicht antworten → Keine Verfügbarkeit (kein A)



CAP-Theorem

- Beweis-Idee (über Widerspruch)
  - Annahme: Es gibt ein CAP System
  3. CA geht nicht mit P
     - Anfrage an Station St_1 X zu schreiben mit Wert x_1
       - Da das System verfügbar ist (A), muss St_1 antworten
       - Die Änderung muss an die anderen Stationen weitergegeben werden (C), darf also nicht partitioniert sein
         → Ausfalltoleranz nicht gegeben (kein P)



CAP-Theorem

- Welche der Eigenschaften nicht berücksichtigt wird, gilt nicht unbedingt systemweit
  - CAP Eigenschaften in der Regel nicht binär, sondern graduell
- CA-Systeme (Konsistenz und Verfügbarkeit)
  - Beispiel: relationale Datenbank mit verteilten Transaktionen
  - Voraussetzung: Verbindung zwischen den Knoten besteht; sonst Schreiben ggf. blockieren
- CP-Systeme (Konsistenz und Ausfalltoleranz)
  - Beispiel: Bank-Anwendungen
  - Hohe Verfügbarkeit bei lesenden Operationen, bei Unterbrechung der Netzwerkverbindung geringere Verfügbarkeit
- AP-Systeme (Verfügbarkeit und Ausfalltoleranz)
  - Beispiel: Domain Name System (DNS)
  - Hohe Verfügbarkeit bei schreibenden und lesenden Operationen, eingeschränkte Konsistenz

Konsistenzmodell: ~~C~~AP

Not-only SQL (NoSQL)-Datenbanken verzichten zeit- bzw. teilweise auf Konsistenz und betonen Verfügbarkeit und Ausfalltoleranz

- Relaxiertes Konsistenzmodell:
  - Basically Available, Soft State, Eventually Consistent (BASE)
    - Ordnet Konsistenz der Verfügbarkeit unter
      - Konsistenz kein fester Zustand nach einer Transaktion, wird später erreicht
    - Replizierte Daten haben nicht alle den neuesten Zustand
      - Vermeidung des (teuren) Zwei-Phasen-Commit-Protokolls (später)
      - Transaktionen könnten veraltete Daten zu lesen bekommen
    - Eventual Consistency
      - Würde man das System anhalten, würden alle Kopien irgendwann in denselben Zustand übergehen
  - Read your Writes-Garantie
    - T_i leistet auf jeden Fall ihre eigenen Änderungen
  - Monotonic Read-Garantie
    - T_i würde beim wiederholten Lesen keinen älteren Zustand als den vorher mal sichtbaren lesen

Zwischenzusammenfassung

- Fragmentierung: Verteilung der Daten in Fragmente
  - Horizontal, vertikal, gemischt (hor. + vert.), abgeleitet (i.d.R. über Fremdschlüssel)
  - Korrektheit: Rekonstruierbarkeit, Vollständigkeit, Disjunktheit
- Replikation: Erhöhung der Verfügbarkeit / Effizienz
  - Nachteil: Daten müssen konsistenz gehalten werden
- Allokation: Physische Zuordnung auf Stationen (lokale DBs)
- Transparenz
  - Fragmentierungstransparenz, Allokationstransparenz, lokale Schema-Transparenz
  - Von nur globales Schema genutzt bis hin zu lokale Stationen müssen bekannt sein
- CAP: Konsistenz, Verfügbarkeit, Ausfalltoleranz
  - CAP-Theorem: Nur zwei von drei Eigenschaften zur Zeit erreichbar

Überblick: 8. Verteilte Datenbanken

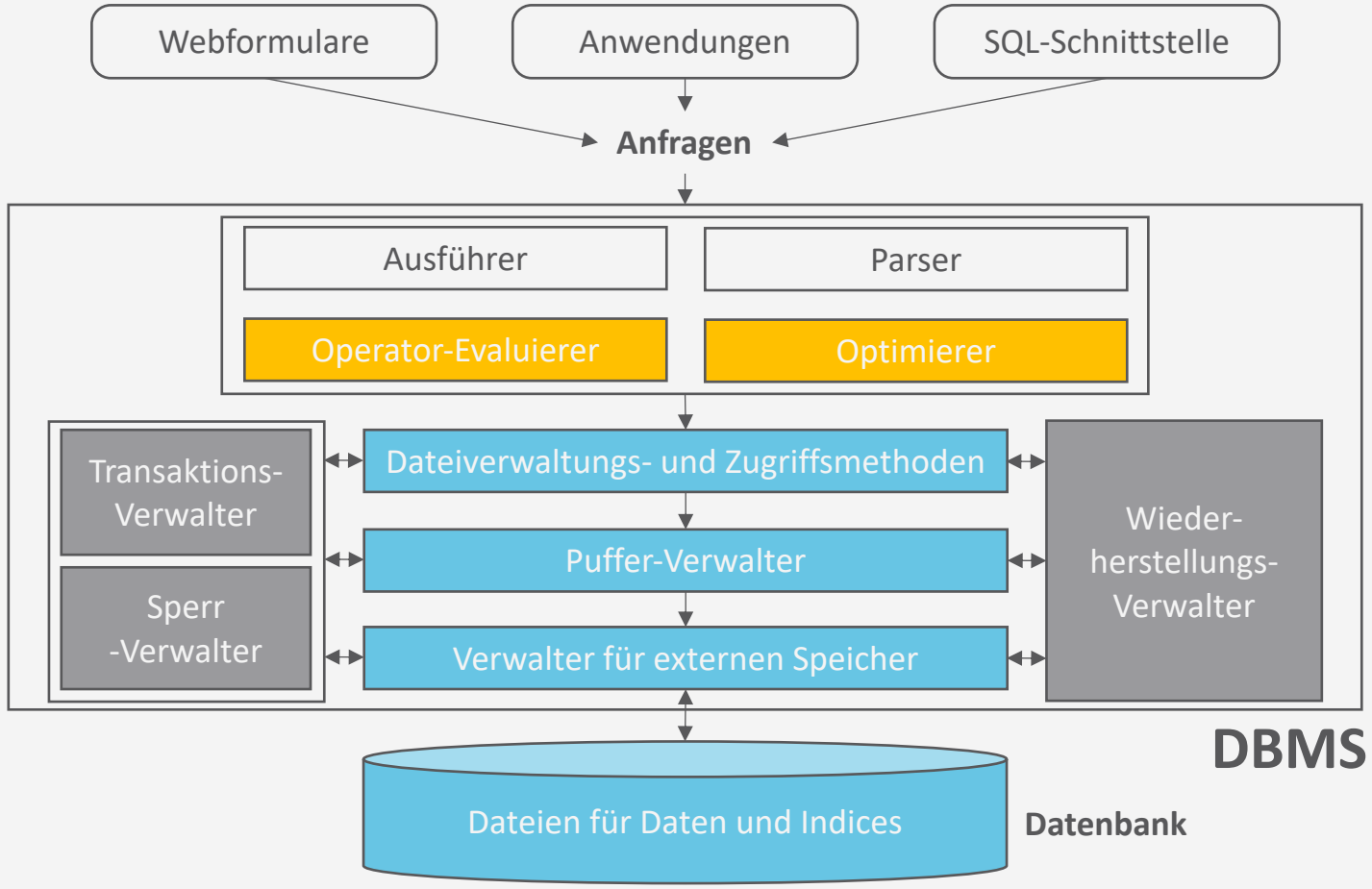
A. Verteilte DBMS
- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

B. Anfragenbeantwortung in verteilten Systemen
- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

C. Föderierte DBS
- Integration
- Migration

Anfrageübersetzung und Anfrageoptimierung

- Voraussetzung: Fragmentierungstransparenz
- Aufgabe des Operator-Evaluierers: Generierung eines Anfrageauswertungsplans auf den Fragmenten
- Aufgabe des Anfrageoptimierers: Generierung eines möglichst effizienten Auswertungsplanes
  - Abhängig von der Allokation der Fragmente auf den verschiedenen Stationen des Rechnernetzes

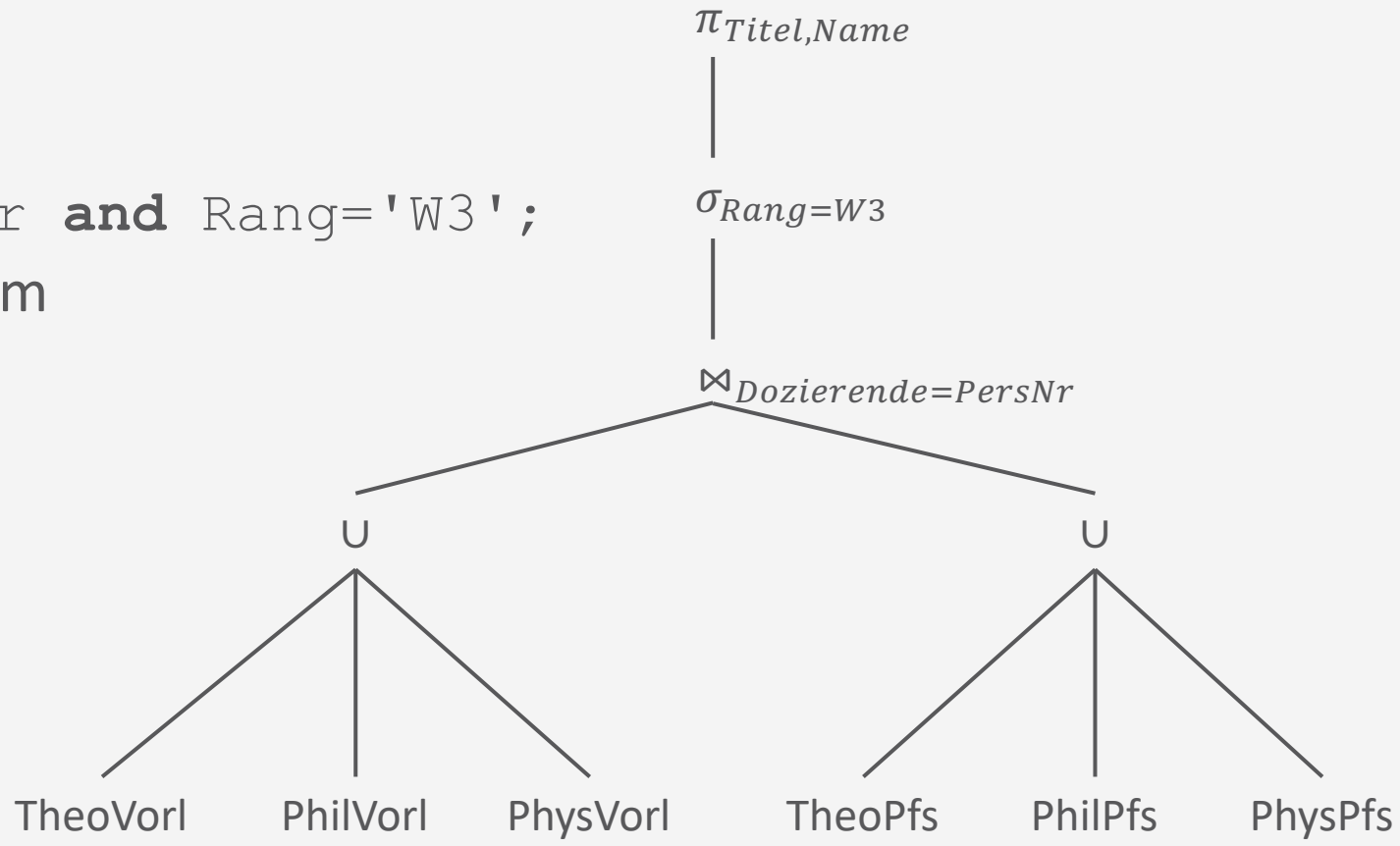
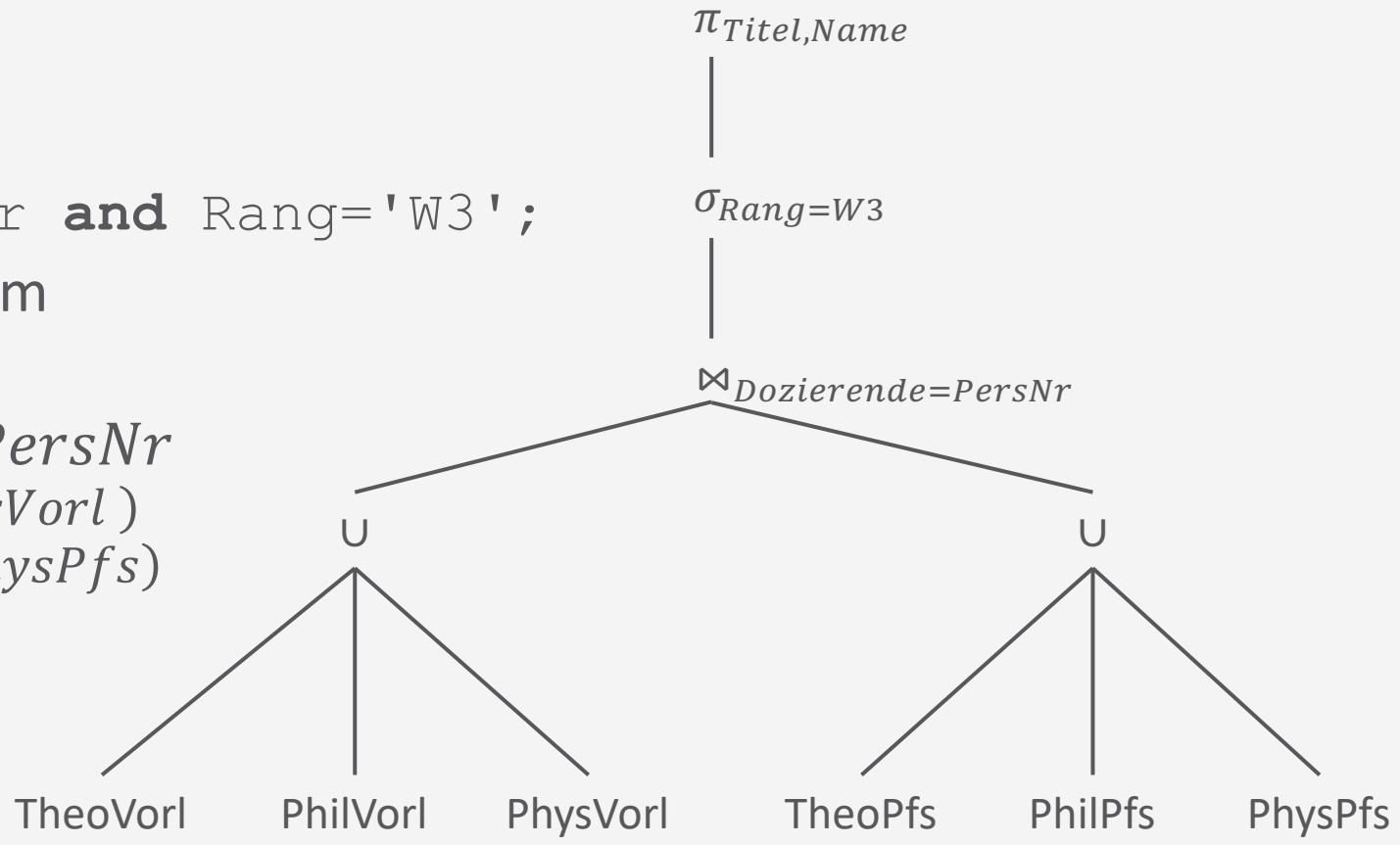


Anfragebearbeitung bei horizontaler Fragmentierung

- Übersetzung einer SQL-Anfrage auf dem globalen Schema in eine äquivalente Anfrage auf den Fragmenten benötigt zwei Schritte:
  1. Rekonstruktion aller in der Anfrage vorkommenden globalen Relationen aus den Fragmenten, in die sie während der Fragmentierungsphase zerlegt wurden
     - Hierfür erhält man einen algebraischen Ausdruck = kanonische Form der Anfrage
  2. Kombination des Rekonstruktionsausdrucks mit dem algebraischen Anfrageausdruck, der sich aus der Übersetzung der SQL-Anfrage ergibt

Algebraischer Ausdruck: Beispiel

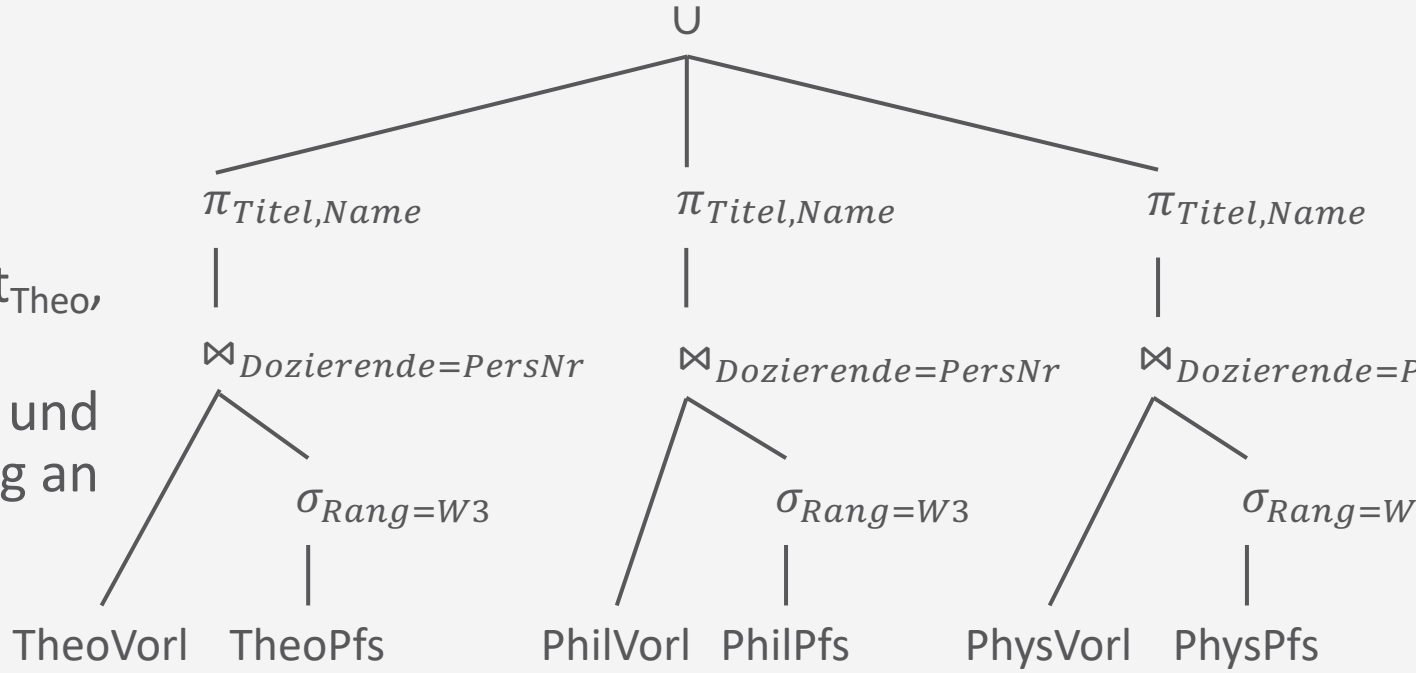
- Anfrage
  - ```
    select Titel, Name
    from Vorlesungen, Profs
    where Dozierende = PersNr and Rang='W3';
    ```
- Algebraischer Ausdruck in Baumform
  - I.e., kanonische Form der Anfrage


    ```

Algebraische Äquivalenzen

- Für eine effizientere Abarbeitung der Anfrage benutzt der Anfrageoptimierer die folgende Eigenschaft:

$$(R_1 \cup R_2) \bowtie_p (S_1 \cup S_2) = (R_1 \bowtie_p S_1) \cup (R_1 \bowtie_p S_2) \cup (R_2 \bowtie_p S_1) \cup (R_2 \bowtie_p S_2)$$

  - Verallgemeinerung auf n horizontale Fragmente $R_1, \dots, R_n$ von R und m Fragmente $S_1, \dots, S_m$ von S ergibt:

$$(R_1 \cup \cdots \cup R_n) \bowtie_p (S_1 \cup \cdots \cup S_m) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (R_i \bowtie_p S_j)$$

- Falls gilt $S_i = S \ltimes_p R_i$ mit $S = S_1 \cup \cdots \cup S_m$, dann gilt immer: $R \bowtie_p S = \bigcup_{i=1}^{m} (R_i \bowtie_p S_i)$
  - *Semi-Join* $\ltimes_p$: $T \ltimes_p U$ wählt diejenigen Tupel aus T, die ein Join-Tupel in U haben
    - Gegeben Attribute $A_1, \dots, A_k$ in T, $\ltimes_p$ ausdrückbar als: $T \ltimes_p U = \pi_{A_1,\dots,A_k}(T \bowtie_p U)$
  - Bei derart abgeleiteten horizontalen Fragmentierungen gilt somit immer:

$$(R_1 \cup \cdots \cup R_n) \bowtie_p (S_1 \cup \cdots \cup S_m) = (R_1 \bowtie_p S_1) \cup \cdots \cup (R_m \bowtie_p S_m)$$

Algebraischer Ausdruck: Beispiel

- Anfrage
  - ```
    select Titel, Name
    from Vorlesungen, Profs
    where Dozierende = PersNr and Rang='W3';
    ```
- Algebraischer Ausdruck in Baumform
  - I.e., kanonische Form der Anfrage
- Umformung: $p \triangleq Dozierende = PersNr$

$$(TheoVorl \cup PhilVorl \cup PhysVorl) \bowtie_p (TheoPfs \cup PhilPfs \cup PhysPfs)$$
$$= (TheoVorl \bowtie_p TheoPfs) \cup (PhilVorl \bowtie_p PhilPfs) \cup (PhysVorl \bowtie_p PhysPfs)$$


    ```

Optimale Form der Anfrage

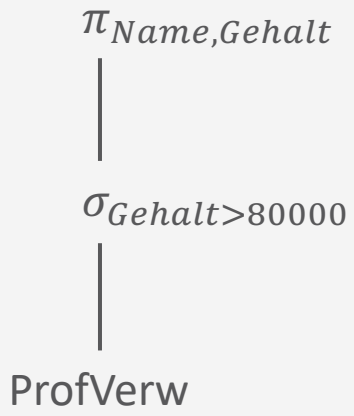
- Selektionen und Projektionen über den Vereinigungsoperator hinweg „nach unten drücken“:
  - $\sigma_p(R_1 \cup R_2) = \sigma_p(R_1) \cup \sigma_p(R_2)$
  - $\pi_{List}(R_1 \cup R_2) = \pi_{List}(R_1) \cup \pi_{List}(R_2)$
- Ergibt folgenden Auswertungsplan:
  - Auswertungen lokal auf den Stationen St_{Theo}, St_{Phys} und St_{Phil} ausführen
    → Stationen können parallel abarbeiten und lokales Ergebnis voneinander unabhängig an die Station, die die abschließende Vereinigung durchführt, übermitteln

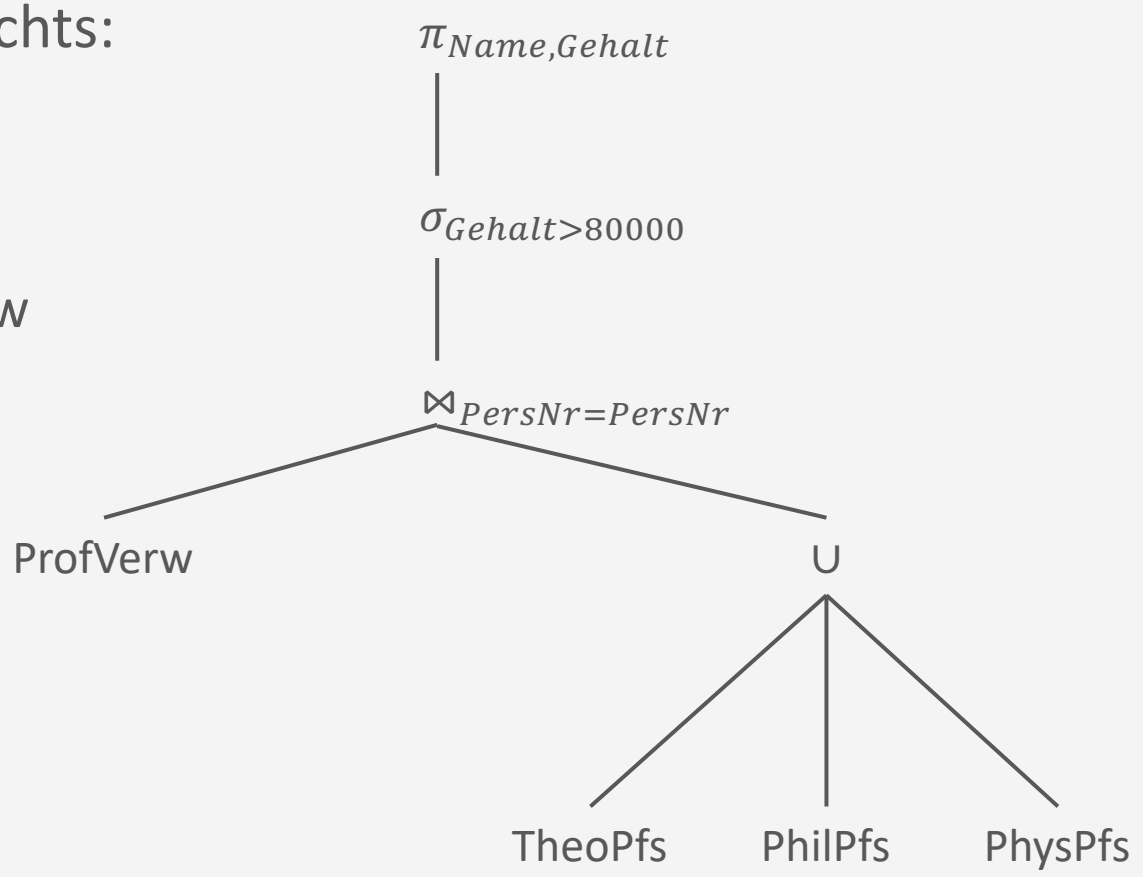
Lokale DB/Station	Bemerkung	Zugeordnete Fragmente
St_{Verw}	Verwaltungsrechner	{ProfVerw}
St_{Phil}	Dekanat Physik	{PhilVorl, PhilPfs}
St_{Phys}	Dekanat Philosophie	{PhysVorl, PhysPfs}
St_{Theo}	Dekanat Theologie	{TheoVorl, TheoPfs}



Anfragebearbeitung bei vertikaler Fragmentierung

- Beispiel mit kanonischem Auswertungsplan rechts:
  - **select** Name, Gehalt
    from Profs
    where Gehalt > 80000;
  - Alle notwendigen Informationen sind in ProfVerw enthalten → Teil mit Vereinigung und Join kann abgeschnitten werden
    - Das ergibt einen optimierten Auswertungsplan
    - Beispiel für schlecht zu optimierende Anfrage
      - Dazu noch Rang ausgeben wollen





Join-Auswertung in VDBMS

- Join-Auswertung spielt kritischere Rolle als in zentralisierten Datenbanken
  - Problem: Argumente eines Joins zweier Relationen können auf unterschiedlichen Stationen liegen
  - Allgemeinster Fall:
    - Äußere Argumentrelation R ist auf Station St_R gespeichert
    - Innere Argumentrelation S ist dem Knoten St_S zugeordnet
    - Ergebnis der Joinberechnung wird auf einem dritten Knoten St_{Result} benötigt
- Zwei Möglichkeiten: Join-Auswertung mit und ohne Filterung
  - Ohne Filterung: Nested-Loops, Transfer einer Argumentrelation, Transfer beider Argumentrelationen
  - Mit Filterung: Semi-Join zur Filterung

$R \bowtie S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Nested-Loops

- Iteration durch die äußere Relation R mittels Laufvariable r und Anforderung des/der zu jedem Tupel r passenden Tupel $s \in S$ mit $r.C = s.C$ (über Kommunikationsnetz bei St_S)
  - C die Join-Variable
- Diese Vorgehensweise benötigt pro Tupel aus R eine Anforderung und eine passende Tupelmenge aus S (welche bei vielen Anforderungen leer sein könnte)
  $\rightarrow$ Es werden $2 \cdot |R|$ Nachrichten benötigt

$R \bowtie S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Transfer einer Argumentrelation

1. Vollständiger Transfer einer Argumentrelation (z.B. R) zum Knoten der anderen Argumentrelation
2. Ausnutzung eines möglicherweise auf $S.C$ existierenden Indexes

$R \bowtie S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Transfer beider Argumentrelationen

1. Transfer beider Argumentrelationen zum Rechner St_{Result}
2. Berechnung des Ergebnisses auf dem Knoten St_{Result} mittels
   a. Merge-Join (bei vorliegender Sortierung)

   oder

   b. Hash-Join (bei fehlender Sortierung)

→ Evtl. Verlust der vorliegenden Indexe für die Join-Berechnung

→ Kein Verlust der Sortierung der Argumentrelation(en)

$R \bowtie S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Join-Auswertung mit Filterung

- Idee: Nur Transfer von Tupeln mit passendem Join-Partner über Semi-Join
  - Benutzung der folgenden algebraischen Eigenschaften:
    - Bei zwei Relationen R, S mit Filterung von S über Join-Attribut C
    - $R \bowtie S = R \bowtie (R \rtimes S)$
    - $R \rtimes S = \pi_C(R) \rtimes S$
  - Vorgehen
    1. Transfer der unterschiedlichen C-Werte von R $(= \pi_{List}(R))$ nach St_S
    2. Auswertung des Semi-Joins $R \rtimes S = \pi_C(R) \rtimes S$ auf St_S und Transfer nach St_R
    3. Auswertung des Joins auf St_R, der nur diese transferierten Ergebnistupel des Semi-Joins braucht
  - Transferkosten werden nur reduziert, wenn gilt: $\|\pi_C(R)\| + \|R \rtimes S\| < \|S\|$
    - Mit $\|R\| =$ Größe (in Byte) einer Relation

$R \bowtie S$

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

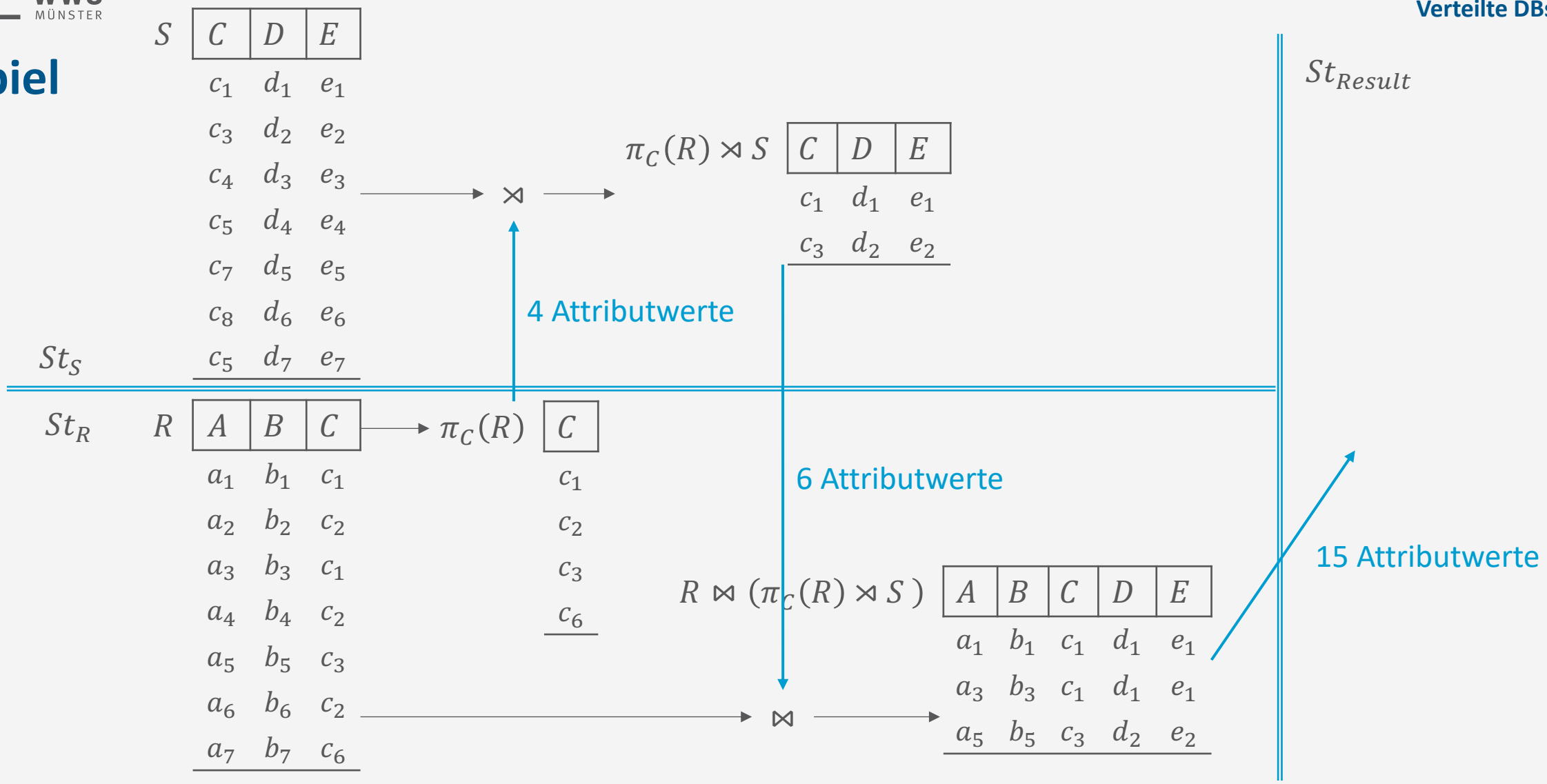
R

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

S

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

Beispiel


St_{Result}
St_S
St_R
S
C D E
c_1 d_1 e_1
c_3 d_2 e_2
c_4 d_3 e_3
c_5 d_4 e_4
c_7 d_5 e_5
c_8 d_6 e_6
c_5 d_7 e_7
$\pi_C(R) \rtimes S$
C D E
c_1 d_1 e_1
c_3 d_2 e_2
4 Attributwerte
R
A B C
a_1 b_1 c_1
a_2 b_2 c_2
a_3 b_3 c_1
a_4 b_4 c_2
a_5 b_5 c_3
a_6 b_6 c_2
a_7 b_7 c_6
$\pi_C(R)$
C
c_1
c_2
c_3
c_6
6 Attributwerte
$R \bowtie (\pi_C(R) \rtimes S)$
A B C D E
a_1 b_1 c_1 d_1 e_1
a_3 b_3 c_1 d_1 e_1
a_5 b_5 c_3 d_2 e_2
15 Attributwerte

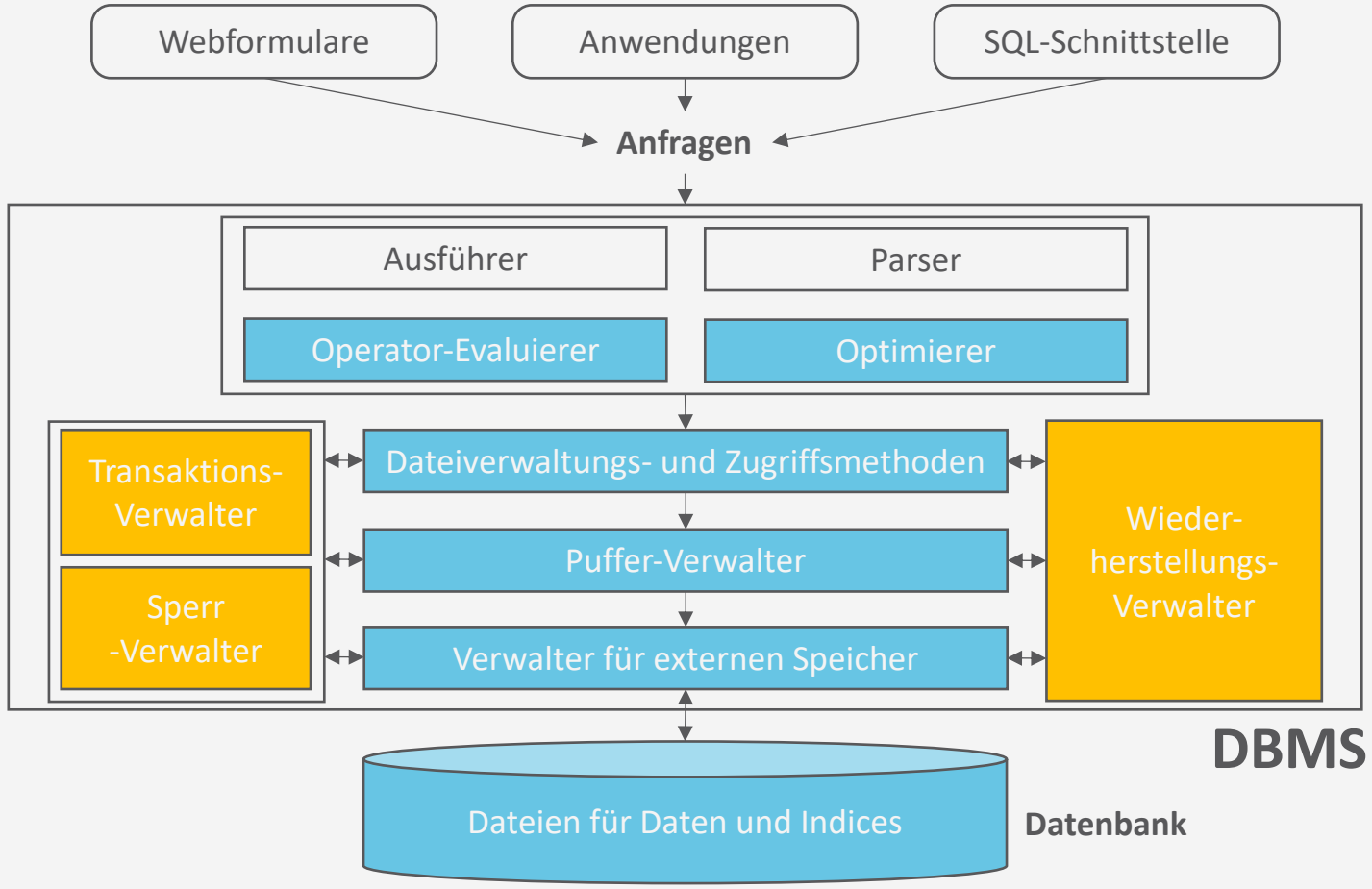
Parameter für die Kosten eines Auswertungsplan

- Kardinalitäten von Argumentrelationen
  - Wie bisher
- Selektivitäten von Joins und Selektionen
  - Wie bisher
- *Transferkosten für Datenkommunikation (Verbindungsaufbau + von Datenvolumen abhängiger Anteil für Transfer)*
- *Auslastung der einzelnen VDBMS-Stationen*

Effektive Anfrageoptimierung muss auf Basis eines Kostenmodells durchgeführt werden und soll mehrere Alternativen für unterschiedliche Auslastungen des VDBMS erzeugen.

Transaktionsmanagement in VDBMS

- Transaktionen können sich bei VDBMS über mehrere Rechnerknoten erstrecken



Widerherstellungsverwaltung in VDBMS

- Recovery:
  - Redo
    - Wenn eine Station nach einem Fehler wieder anläuft, müssen alle Änderungen einmal abgeschlossener Transaktionen - seien sie lokal auf dieser Station oder global über mehrere Stationen ausgeführt worden - auf den an dieser Station abgelegten Daten wiederhergestellt werden
  - Undo
    - Die Änderungen noch nicht abgeschlossener lokaler und globaler Transaktionen müssen auf den an der abgestürzten Station vorliegenden Daten rückgängig gemacht werden

EOT-Behandlung

- Die EOT (End-of-Transaction)-Behandlung von globalen Transaktionen stellt in VDBMS ein Problem dar
- Eine globale Transaktion muss atomar beendet werden, d.h.

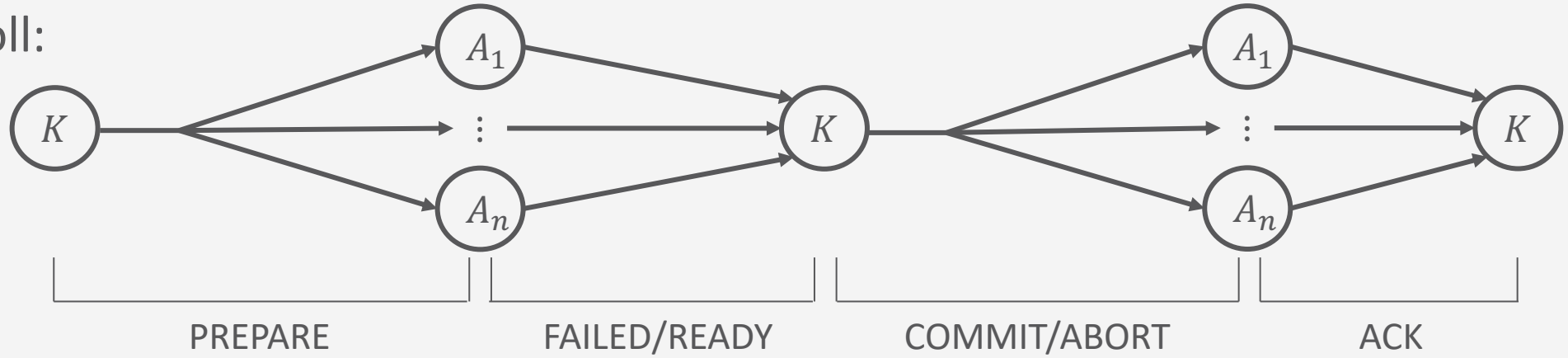
  entweder
  - commit: Globale Transaktion wird an allen (relevanten) lokalen Stationen festgeschrieben

  oder
  - abort: Globale Transaktion wird gar nicht festgeschrieben

→ Problem in verteilter Umgebung, da die Stationen eines VDBMS unabhängig voneinander abstürzen können

Problemlösung: Zwei-Phasen-Commit-Protokoll

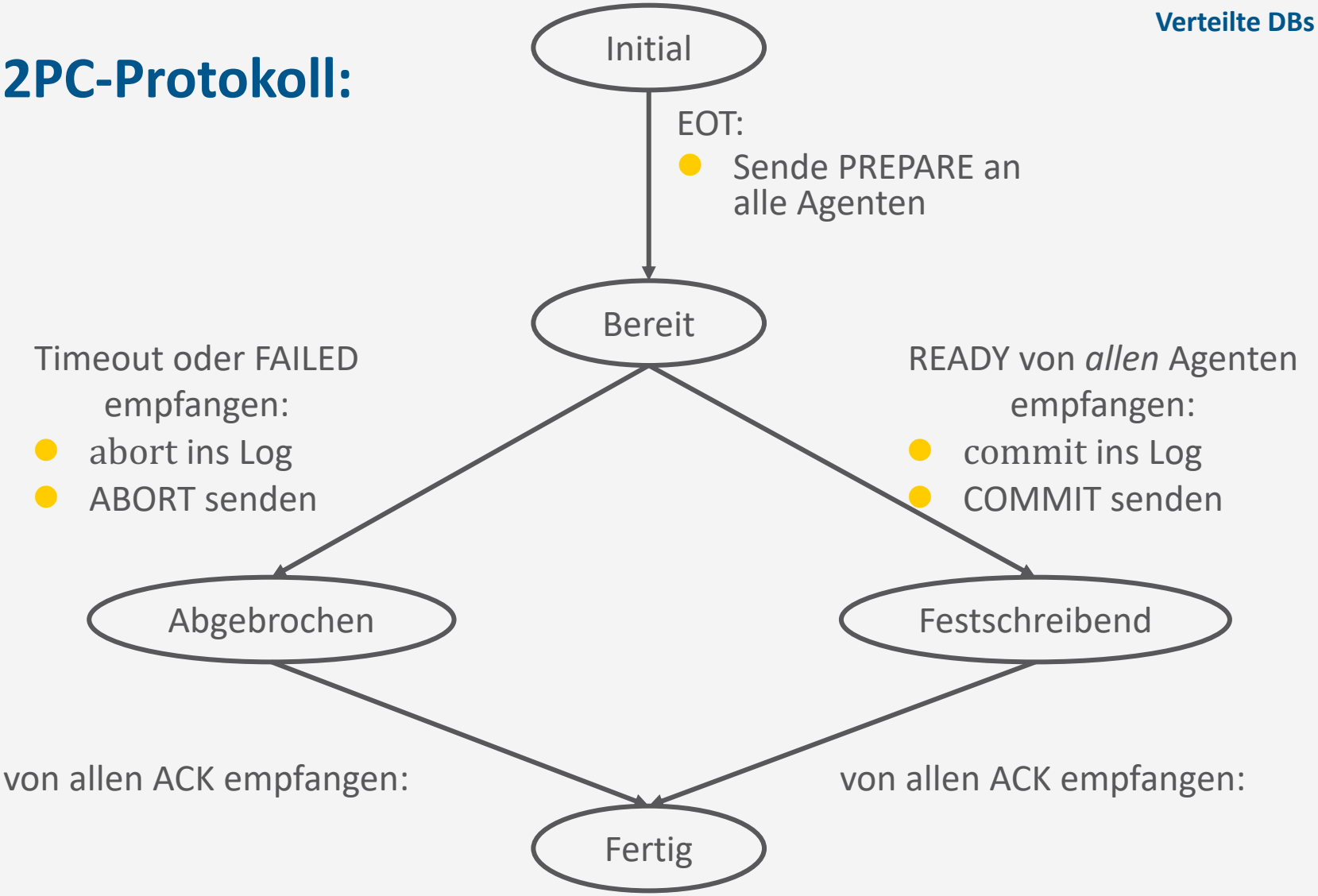
- Zwei-Phasen-Commit-Protokoll (2PC) gewährleistet Atomarität der EOT-Behandlung
- 2PC-Verfahren wird von sogenanntem Koordinator K überwacht
- Gewährleistet, dass die n Agenten (Stationen im VDBMS) $A_1, \ldots, A_n$, die an einer Transaktion beteiligt waren, entweder alle von Transaktion T geänderten Daten festschreiben oder alle Änderungen von T rückgängig machen
- Nachrichtenaustausch beim 2PC-Protokoll:



Ablauf der EOT-Behandlung beim 2PC-Protokoll

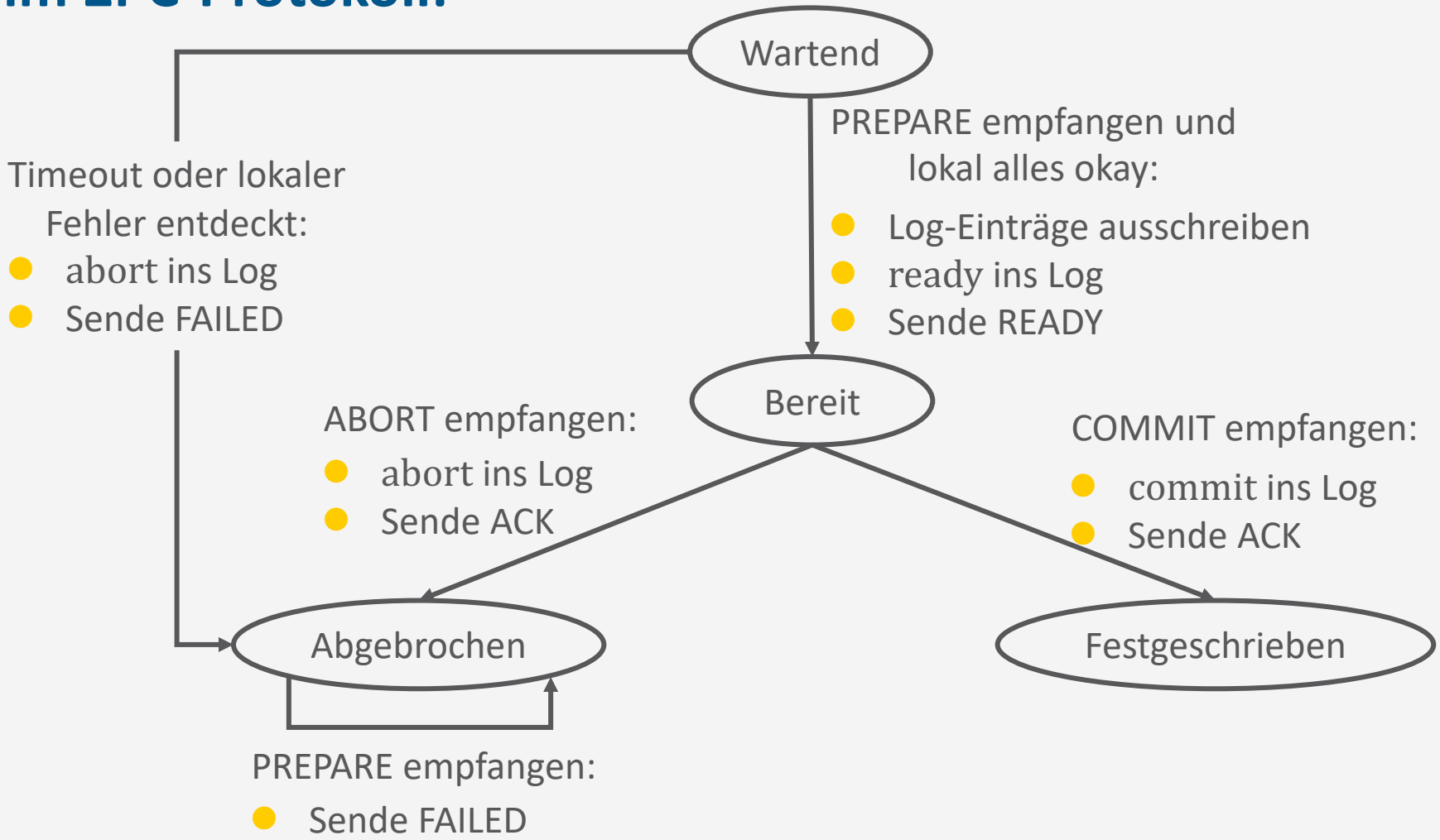
- K schickt allen Agenten eine PREPARE-Nachricht, um herauszufinden, ob sie Transaktionen festschreiben können
  - Jeder Agent A_i empfängt PREPARE-Nachricht und schickt eine von zwei möglichen Nachrichten an K:
    - READY, falls A_i in der Lage ist, die Transaktion T lokal festzuschreiben
    - FAILED, falls A_i kein Commit durchführen kann (wegen Fehler, Inkonsistenz etc.)
- Hat K von allen n Agenten $A_1, \ldots, A_n$ ein READY erhalten, kann K ein COMMIT an alle Agenten schicken mit der Aufforderung, die Änderungen von T lokal festzuschreiben
  - Antwortet einer der Agenten mit FAILED oder gar nicht innerhalb einer bestimmten Zeit ($timeout$), schickt K ein ABORT an alle Agenten und diese machen die Änderungen der Transaktion rückgängig
  - Haben die Agenten ihre lokale EOT-Behandlung abgeschlossen, schicken sie eine ACK-Nachricht (=acknowledgement, dt. Bestätigung) an den Koordinator

Zustandsübergang beim 2PC-Protokoll: Koordinator



„Bullet“ ●
= wichtigste Aktion(en)

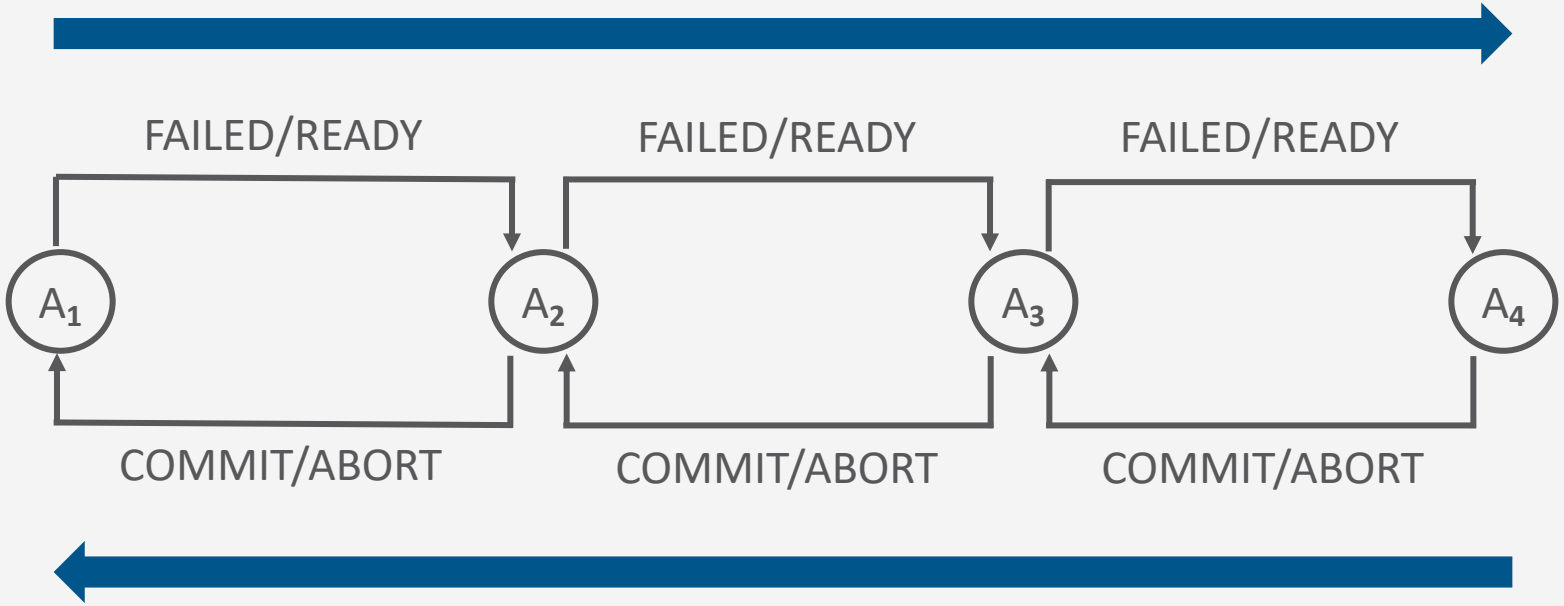
Zustandsübergang beim 2PC-Protokoll: Agent



„Bullet“ ●
= wichtigste Aktion(en)

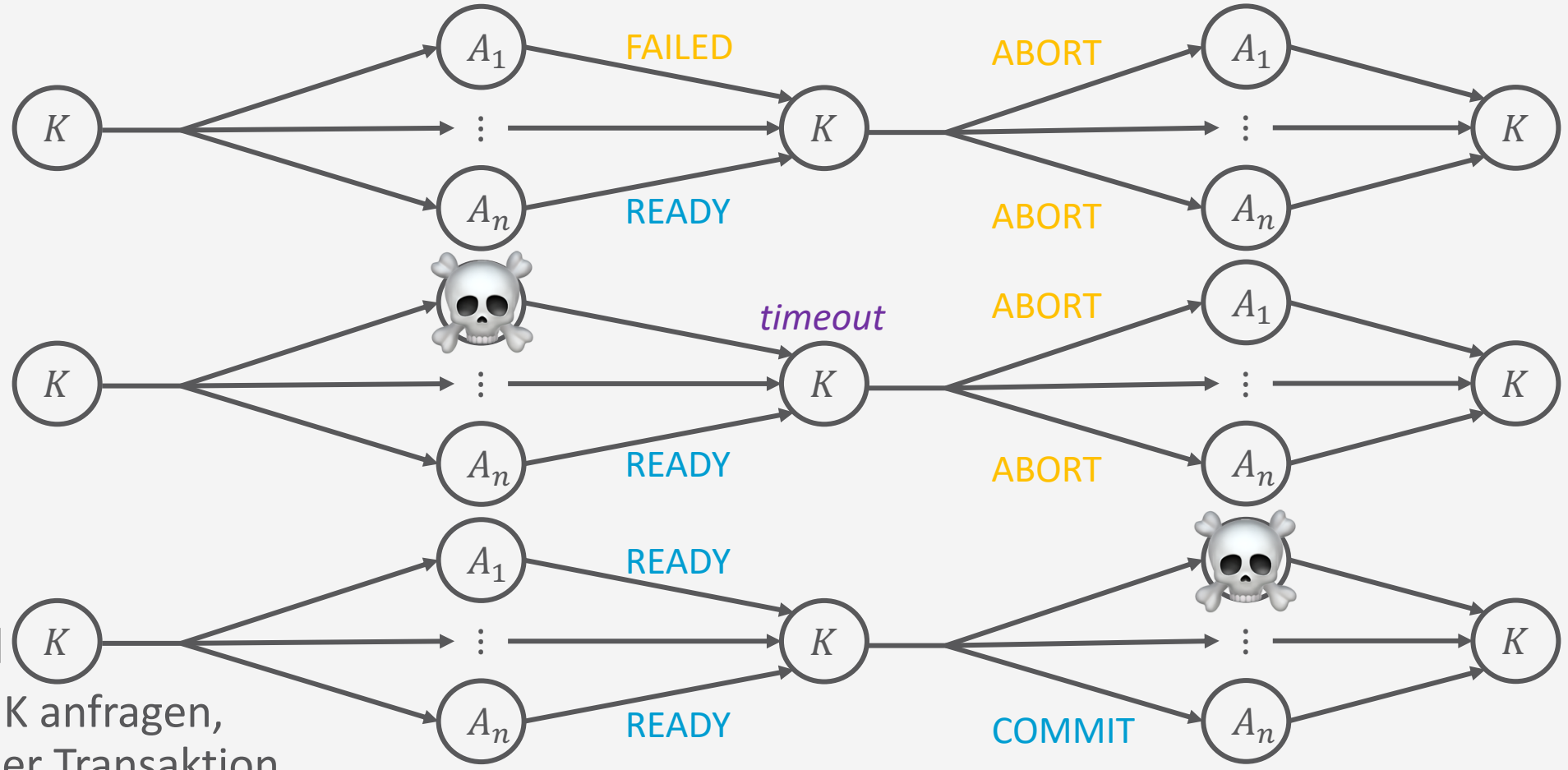
2PC-Protokoll ohne Koordinator

- Lineare Organisationsform: Reihenfolge der Agenten festlegen
- Nacheinander kommunizieren
  1. Agenten reichen ihren eigenen Status und den der vorherigen Nachbarn an den nächsten weiter, nachdem sie den entsprechenden Statusbericht vom Vorgänger bekommen haben
  2. Der letzte Agent trifft die Entscheidung und reicht sie zurück



Problemlösung: Zweiphasen-Commit-Protokoll

1. Transaktion kann nicht durchgeführt werden
2. Station crasht, bevor READY gesendet wird (timeout bei K)
3. Station crasht, bevor COMMIT empfangen wird
   - Nach Restart bei K anfragen, ob undo / redo der Transaktion



Absturz eines Agenten

- Antwortet ein Agent innerhalb eines Timeout-Intervalls nicht auf die PREPARE-Nachricht, gilt der Agent als abgestürzt; der Koordinator bricht die Transaktion ab und schickt eine ABORT-Nachricht an alle Agenten
- Abgestürzter Agent schaut beim Wiederanlauf in seine Log-Datei:
  - Kein ready-Eintrag bzgl. Transaktion T → Agent führt ein Abort durch und teilt dies dem Koordinator mit (FAILED-Nachricht)
  - ready-Eintrag, aber kein commit-Eintrag → Agent fragt Koordinator, was aus Transaktion T geworden ist; Koordinator teilt COMMIT oder ABORT mit, was beim Agenten zu einem Redo oder Undo der Transaktion führt
  - commit-Eintrag vorhanden → Agent weiß ohne Nachfragen, dass ein (lokales) Redo der Transaktion nötig ist (ACK senden)

Absturz eines Koordinators

- Absturz vor dem Senden einer COMMIT-Nachricht
  → Rückgängigmachen der Transaktion durch Versenden einer ABORT-Nachricht
- Absturz nachdem Agenten ein READY mitgeteilt haben
  → Blockierung der Agenten
  - Hauptproblem des 2PC-Protokolls beim Absturz des Koordinators, da dadurch die Verfügbarkeit des Agenten bezüglich andere globaler und lokaler Transaktionen drastisch eingeschränkt ist
  - Um Blockierung von Agenten zu verhindern, wurde ein Dreiphasen-Commit-Protokoll konzipiert, das aber in der Praxis zu aufwendig ist (VDBMS benutzen das 2PC-Protokoll)

Verlorengegangene Nachrichten

- PREPARE-Nachricht des Koordinators an einen Agenten geht verloren oder READY-(oder FAILED-)Nachricht eines Agenten geht verloren
  - Nach Timeout-Intervall geht Koordinator davon aus, dass betreffender Agent nicht funktionsfähig ist und sendet ABORT-Nachricht an alle Agenten (Transaktion gescheitert)
- Agent erhält im Zustand Bereit keine Nachricht vom Koordinator
  - Agent ist blockiert, bis COMMIT- oder ABORT-Nachricht vom Koordinator kommt, da Agent nicht selbst entscheiden kann (deshalb schickt Agent eine Erinnerung an den Koordinator)

Mehrbenutzersynchronisation in VDBMS

- Serialisierbarkeit
  - Lokale Serialisierbarkeit an jeder der an den Transaktionen beteiligten Stationen reicht nicht aus
  - Deshalb muss man bei der Mehrbenutzersynchronisation auf globaler Serialisierbarkeit bestehen
  - Beispiel
    - Lokal serialisierbare Historien
      - St_1: lokal ausgeführte Operationen seriell ausgeführt
      - St_2: lokal ausgeführte Operationen seriell ausgeführt
    - Global nicht serialisierbar
      - p_i^a: a identifiziert die Station, i die Transaktion
      - Schedule: $r_1^1(X), w_2^1(X), w_2^2(X), r_1^2(X)$
      - Nicht umformbar zu seriellen Schedule

St_1	T_1	T_2
1	$r(X)$	
2		$w(X)$

St_2	T_1	T_2
3		$w(Y)$
4	$r(Y)$	

Sperrverwaltung in VDBMS

- Lokale Sperrverwaltung: Globale Transaktion muss vor Zugriff / Modifikation eines Datums X, das auf Station S liegt, eine Sperre vom Sperrverwalter der Station S erwerben
  - Verträglichkeit der angeforderten Sperre mit bereits existierenden Sperren kann lokal entschieden werden → Favorisiert lokale Transaktionen, da diese nur mit ihrem lokalen Sperrverwalter kommunizieren müssen
- Globale Sperrverwaltung: Alle Transaktionen fordern alle Sperren an einer einzigen, ausgezeichneten Station an
  - Nachteile:
    - Zentraler Sperrverwalter kann zum Engpass des VDBMS werden, besonders bei einem Absturz der Sperrverwalter-Station
    - Verletzung der lokalen Autonomie der Stationen, da auch lokale Transaktionen ihre Sperren bei der zentralisierten Sperrverwaltung anfordern müssen

  → Zentrale Sperrverwaltung im Allgemeinen nicht akzeptabel

Deadlocks in VDBMS

- Erkennung von Deadlocks (Verklemmungen)
  - Beispiel: Verteilter Deadlock
    - T_2 bei St_2 wartet auf Freigabe von X bei St_1
    - T_1 bei St_1 wartet auf Freigabe von Y bei St_2
    - Lokale Wartegraphen: zyklenfrei

    - Globaler Wartegraph: mit Zyklus

- Vorgehen zur Erkennung:
  - Timout, zentralisiert, dezentral

Erkennung von Deadlocks: Wartegraph

- Wartegraph: Gerichteter Graph
  - Enthält für jede aktive Transaktion T_i einen Knoten
  - Wenn T_i auf eine Sperre von T_j wartet:
    - Füge Kante $(T_i \longrightarrow T_j)$ in den Graphen ein (Kante bei Freigabe löschen)
  - Weist der Wartegraph Zyklen auf, so liegt ein Deadlock vor

St_1	T_1	T_2
0	BOT	
1	$lockS(X)$	
2	$r(X)$	
3		BOT
4		$lockX(Y)$
5		$w(Y)$
6		$lockX(X)$
		...

St_2	T_1	T_2
3		BOT
4		$lockX(Y)$
5		$w(Y)$
7	$lockS(Y)$	
	...	

Erkennung von Deadlocks: Timeout

- Betreffende Transaktion wird zurückgesetzt und erneut gestartet
  - Einfach zu realisieren
  - Problem: Richtige Wahl des Timeout-Intervalls
    - Zu lang
      → Schlechte Ausnutzung der Systemressourcen
    - Zu kurz
      → Deadlock-Erkennung, wo gar keine Verklemmung vorliegt

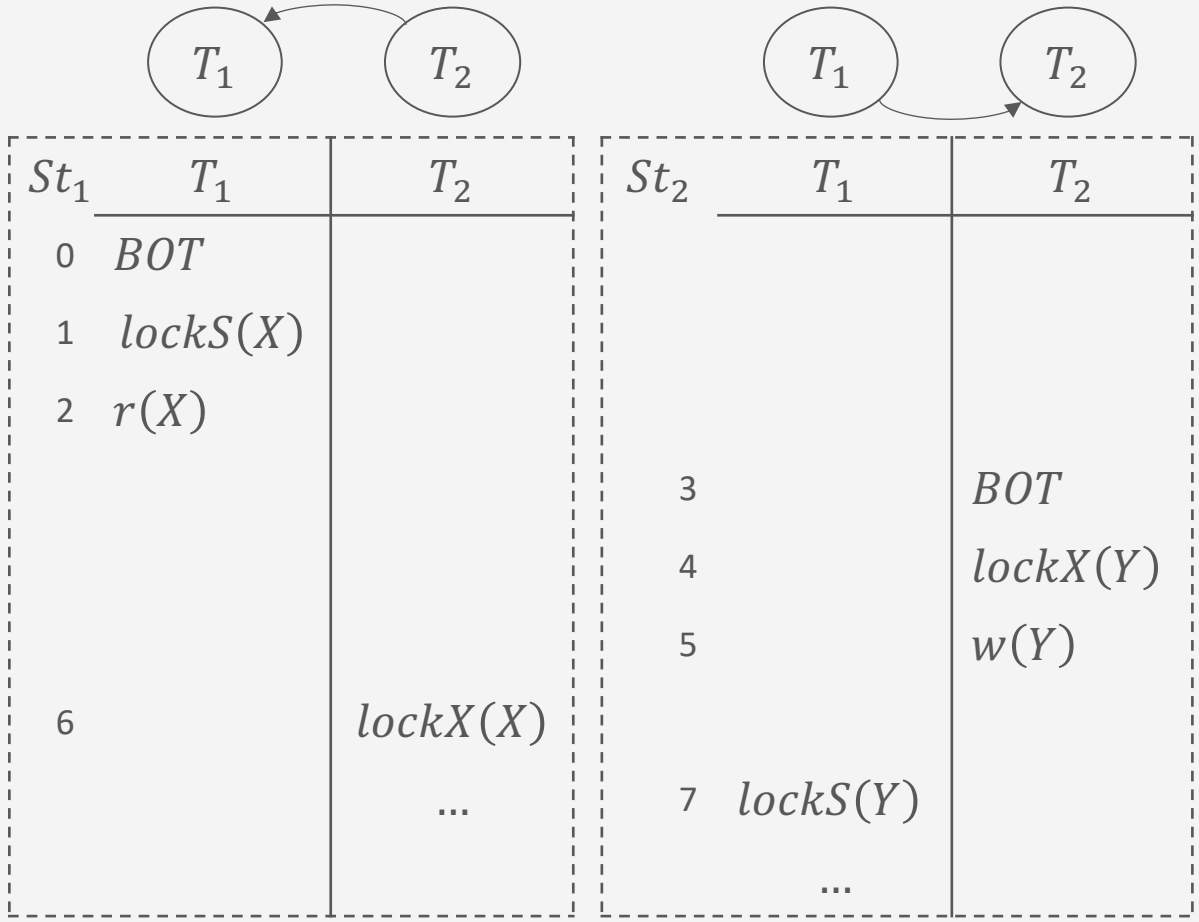
St_1	T_1	T_2
0	BOT	
1	$lockS(X)$	
2	$r(X)$	
6		$lockX(X)$
		...

St_2	T_1	T_2
3		BOT
4		$lockX(Y)$
5		$w(Y)$
7	$lockS(Y)$	
	...	

Zentralisierte Deadlock-Erkennung

- Stationen melden lokal vorliegende Wartebeziehungen an neutralen Knoten, der daraus globalen Wartegraphen aufbaut (Zyklus im Graphen → Deadlock)
  - Sichere Lösung
  - Beispiel: Globaler Wartegraph

- Nachteile:
  - Hoher Aufwand (viele Nachrichten)
  - Entstehung von Phantom-Deadlocks (= nicht-existierende Deadlocks) durch Überholen von Nachrichten im Kommunikationssystem



Dezentrale Deadlock-Erkennung: Obermarck‘s Path Pushing Algorithmus

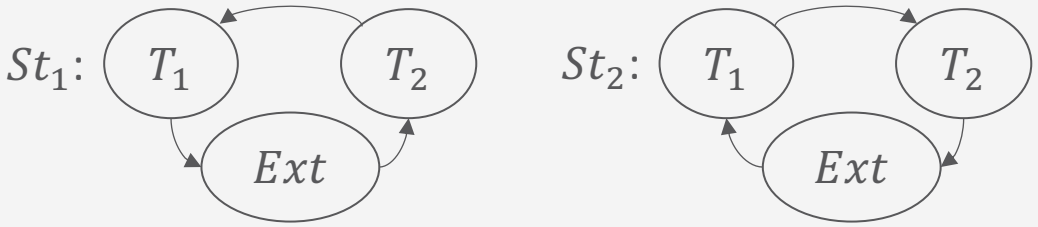
- Zur Erkennung von globalen Deadlocks
- Zuordnung jeder Transaktion zu einem Heimatknoten (Station), von wo aus externe Subtransaktionen auf anderen Stationen initiiert werden
  - Globale Ordnung der Transaktionen (z.B. eindeutiger String mit lexikographischer Ordnung)
- Jede Station hat einen lokalen Wartegraphen, erweitert um einen Knoten Ext (*external*)
  - Ext modelliert der stationenübergreifenden Wartebeziehungen zu externen Subtransaktionen
- Formal: Wartegraph G mit initial
  - Für jede Transaktion T_i ein Knoten T_i (beheimatete und fremde) mit Kanten wie bisher
    - Kante $T_i \rightarrow T_j$, wenn T_i auf T_j wartet
  - Dazu einen Knoten Ext mit Kanten wie folgt
    - Kante $Ext \rightarrow T_i$ wird für jede von außen kommende Transaktion T_i
    - Kante $T_j \rightarrow Ext$ wird für jede Transaktion T_j dieser Station, falls T_j nach außen geht

Dezentrale Deadlock-Erkennung: Obermarck‘s Path Pushing Algorithmus

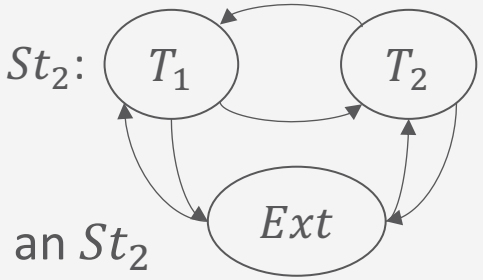
- Vorgehen an jeder Station (asynchron)
  1. Einkommende Informationen aus Schritt 3 von anderen Stationen einbauen
     - Gestoppte Transaktionen T_a und deren Kanten löschen
     - Knoten T_i für unbekannte T_i und Kanten einfügen aus Pfaden, die nicht T_a beinhalten
  2. Liste von elementaren Zyklen in Wartegraph erstellen
     - Elementare Zyklen: Zyklen, so dass keine Subzyklen enthalten sind
  3. Zyklen bearbeiten
     - Lokale Zyklen (ohne Ext Beteiligung) beheben durch Stoppen von Transaktionen
       - Entferne Knoten T_i sowie verbundene Kanten aus Wartegraph, lösche Zyklen mit T_i aus Liste
       - Falls T_i verteilt ausgeführte Transaktion, Information an andere Stationen senden
     - Für alle Zyklen $Ext \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow Ext$:
       - Pfad an Stationen schicken, die Subtransaktionen von T_n gestartet haben
       - Nachrichtenaufkommen reduzieren: Nur schicken, wenn $T_1 \prec T_n$

Dezentrale Deadlock-Erkennung: Beispiel

- St_1 Heimatknoten von T_1, St_2 Heimatknoten von T_2; Ordnung $T_1 \prec T_2$
  - Wartegraphen



  - St_1 Pfad $Ext \rightarrow T_2 \rightarrow T_1 \rightarrow Ext$: $T_1 \prec T_2$
  - St_2 sendet Pfad $Ext \rightarrow T_1 \rightarrow T_2 \rightarrow Ext$ an St_1
    - St_1: Lokaler Zyklus
      → T_1 stoppen
      → Zyklenfrei
      → Info über T_1 an St_2



  - St_1 sendet stop von T_1 an St_2
    - Nach Einbau Wartegraph zyklenfrei

St_1	T_1	T_2
0	BOT	
1	$lockS(X)$	
2	$r(X)$	
6		$lockX(X)$
		…

St_2	T_1	T_2
3		BOT
4		$lockX(Y)$
5		$w(Y)$
7	$lockS(Y)$	
	…	

Dezentrale Deadlock-Erkennung

- Path Pushing Algorithmus hat seine Nachteile
  - Viele Nachrichten, Phantom-Deadlocks
- Weitere Ansätze, aber weiterhin offenes Forschungsproblem
  - Edge Chasing (entlang virtueller Kanten des Wartegraphen)
    - Wartende Transaktion sendet eine Nachricht an die Transaktion, auf die sie wartet
    - Transaktion leitet solch eine Nachricht an andere Transaktionen weiter, auf die sie selber wartet
    - Kommt die Nachricht an die Initiatorin zurück → Deadlock / Zyklus
  - Deadlock Detection Agents (DDAs)
    - DDA zuständig für eine Zusammenhangskomponente (ZHK) in einem Wartegraphen
    - Wenn Kante dazukommt, die ZHKs verbindet, verschmilzt jüngerer DDA mit älterem DDA
    - Irgendwann wird die gesamte ZHK des Wartegraphen (in der sich der Zyklus befindet) in einem DDA landen, der den Zyklus dann entdeckt

Deadlock-Vermeidung

- Optimistische Mehrbenutzersynchronisation:
  Nach Abschluss der Transaktionsbearbeitung wird Validierung durchgeführt
- Zeitstempel-basierte Synchronisation:
  - Zuordnung eines Lese-/Schreib-Stempels zu jedem Datum entscheidet, ob beabsichtigte Operation durchgeführt werden kann ohne Serialisierbarkeit zu verletzen oder ob Transaktion abgebrochen wird (abort)
- Sperrbasierte Synchronisation
  - wound/wait: Nur jüngere Transaktionen warten auf ältere (wie vorher)
  - wait/die: Nur ältere Transaktionen warten auf jüngere (wie vorher)
- Voraussetzungen für Deadlockvermeidungsverfahren:
  Vergabe global eindeutiger Zeitstempel als Transaktionsidentifikatoren
  - Lokale Uhren müssen hinreichend genau aufeinander abgestimmt sein

Synchronisation bei replizierten Daten: Read One, Write All (ROWA)

- Zu einem Datum X gibt es Kopien $X_1, \ldots, X_n$, die auf unterschiedlichen Stationen liegen
- Eine logische Leseoperation auf einer beliebigen Kopie durchführen
  - Z. B. die am nächsten Netzwerk verfügbare Kopie
- Eine logische Schreiboperation wird zu physischen Schreiboperationen auf allen Kopien
  - Synchrones Schreiben aller Kopien in derselben Transaktion
  - 2PL sperrt alle Kopien von X
  - 2PC um ein Commit für die Schreiboperation atomar auf allen Kopien durchzuführen
- Vorteil: Sehr einfach zu implementieren, da es nahtlos mit den Protokollen zusammenpasst; effizientes Lesen; alle Kopien auf gleichem Stand
- Nachteil: Hohe Laufzeit, Verfügbarkeitsprobleme (alle kopienhaltenden Stationen stehen in Abhängigkeit; Änderungstransaktionen werden n mal so lang → hohe Deadlock-Rate)

Synchronisation bei replizierten Daten: Quorum-Consensus Verfahren

- Idee: Änderung auf einer Kopie wird nur dann ausgeführt, wenn die Transaktion in der Lage ist, die Mehrheit der Kopien dafür zu gewinnen (z.B. geeignet zu sperren)
  - Gewichtung der individuellen Stimmen: Entweder eine Stimme pro Kopie oder Gewichte pro Stimme (z.B. mehr Gewicht für hoch verfügbare Stationen)
  - Anzahl der Stimmen, die für das Erreichen der Mehrheit erforderlich sind: Statisch (fest vorgegeben), dynamisch (nur Stationen, die beim letzten Update dabei waren, haben Stimme)
- Für Lesezugriffe (Lesequorum Q_r) und Schreibzugriffe (Schreibquorum Q_w) können unterschiedliche Anzahlen von erforderlichen Stimmen festgelegt werden
  - Ein-Kopie-Serialisierbarkeit bei $Q_w + Q_w > Q$ und $Q_w + Q_r > Q$, Q Gesamtgewicht
    - Dazu Zeitstempel pro Kopie; beim Lesen: Zeitstempel prüfen, neueste Kopie lesen
      - Durch $Q_w + Q_r > Q$ müssen Lese- und Schreiboperationen überlappen, weswegen Lese-Operationen nicht nur veraltete Kopien lesen (mindestens eine ist aktuell)
- Auch ohne Sperren möglich (Stationen im Ring angeordnet); Erweiterung: Tree Quorum

Zwischenzusammenfassung

- Anfrageoptimierung unter Berücksichitung der Fragmentierung
- Verteilte Join-Auswertung aufwendig, da u.U. viele Nachrichten nötig
- Wiederherstellung: redo / undo
- EOT-Behandlung durch 2PC-Protokoll
  - Fehlersituationen: Absturz eines Agenten im Rahmen des Protokolls behandelbar, Absturz des Koordinators u.U. schwierig (Agenten blockiert), verlorengegangene Nachricht (Koordinator wartet behandelbar; Agent wartet → Agent blockiert)
- Transaktionsverwaltung
  - Lokale Serialisierbarkeit bedeutet keine globale Serialisierbarkeit
  - Sperrverwaltung: Lokale Sperren einfach, globale Sperren sicherer aber schwierig umzusetzen
  - Deadlockerkennung schwierig: dezentrale Lösung existiert, aber aufwendig
  - Synchronisation bei replizierten Daten schwierig

Überblick: 8. Verteilte Datenbanken

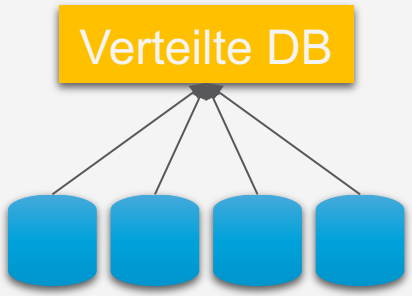
A. Verteilte DBMS
- Fragmentierung, Replikation, Allokation
- Transparenz
- CAP-Theorem

B. Anfragenbeantwortung in verteilten Systemen
- Anfrageverarbeitung
- Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

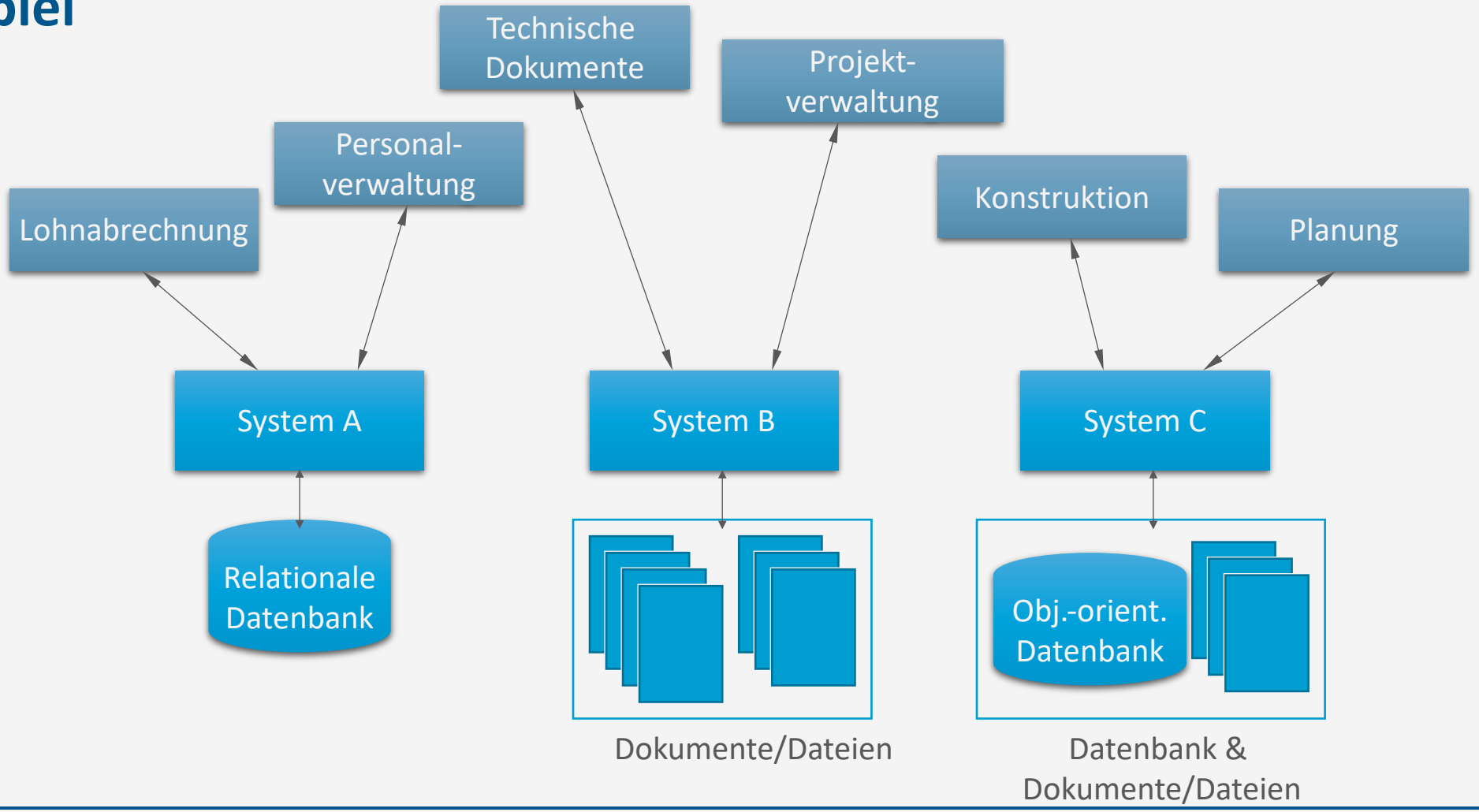
C. Föderierte DBS
- Integration
- Migration

Anwendungsbeispiel

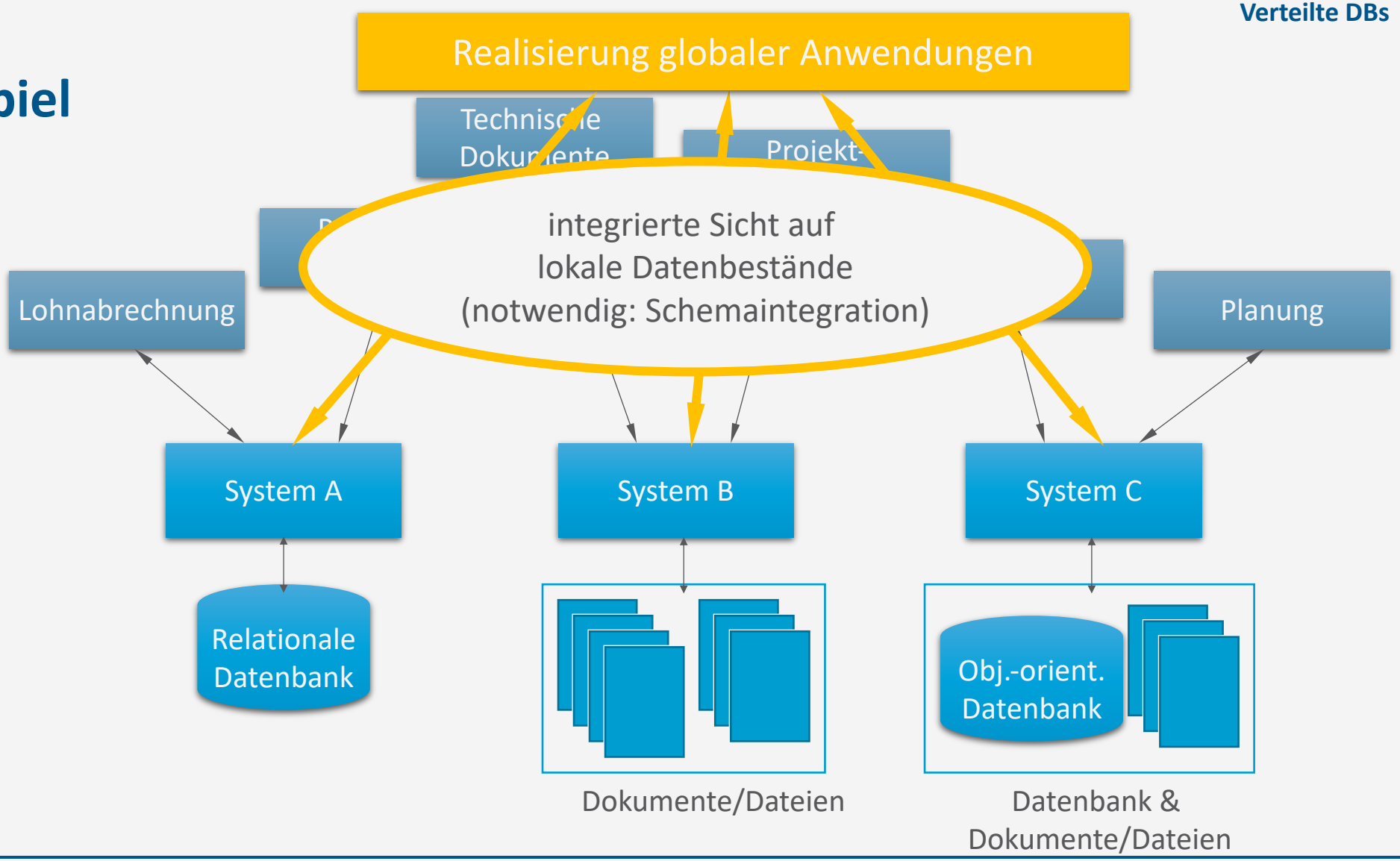
- Typische Situation in Unternehmen und Organisationen:
  - Unabhängig voneinander entstandene Datenbestände in verschiedenen Abteilungen / Arbeitsgruppen
    - Unterschiedliche Strukturierung gleich(artig)er Daten
    - Teilweise redundante Datenhaltung
    - Global inkonsistente Datensituation möglich
  - Autonome, nicht abgestimmte Entscheidungen über Anschaffung von Hardware und Software
- Entstehung von „Insellösungen“, die einen einheitlichen Zugriff auf verteilte Daten erschweren oder sogar unterbinden



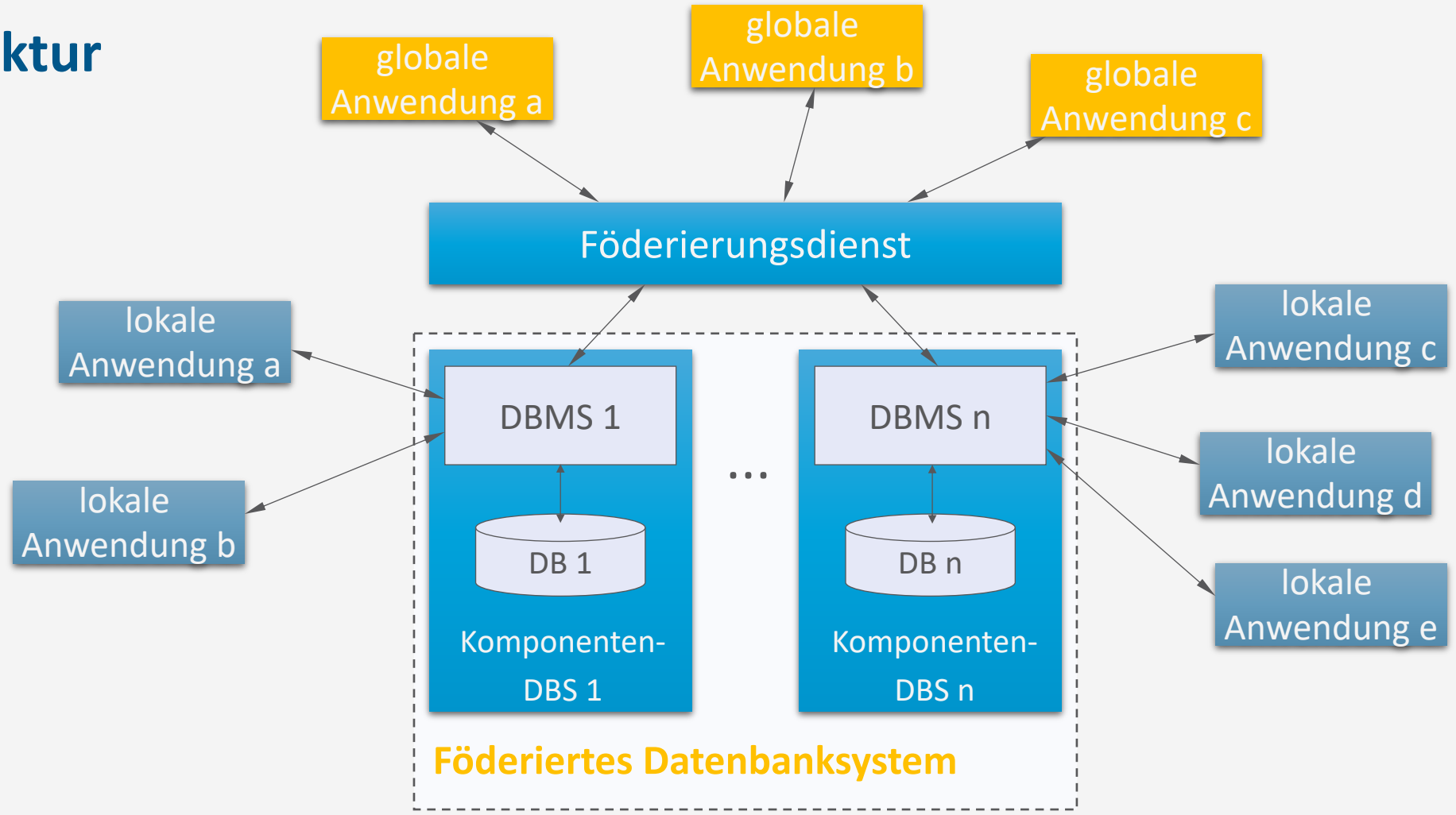
Anwendungsbeispiel



Anwendungsbeispiel



Föderierte DBS: Allgemeine Architektur

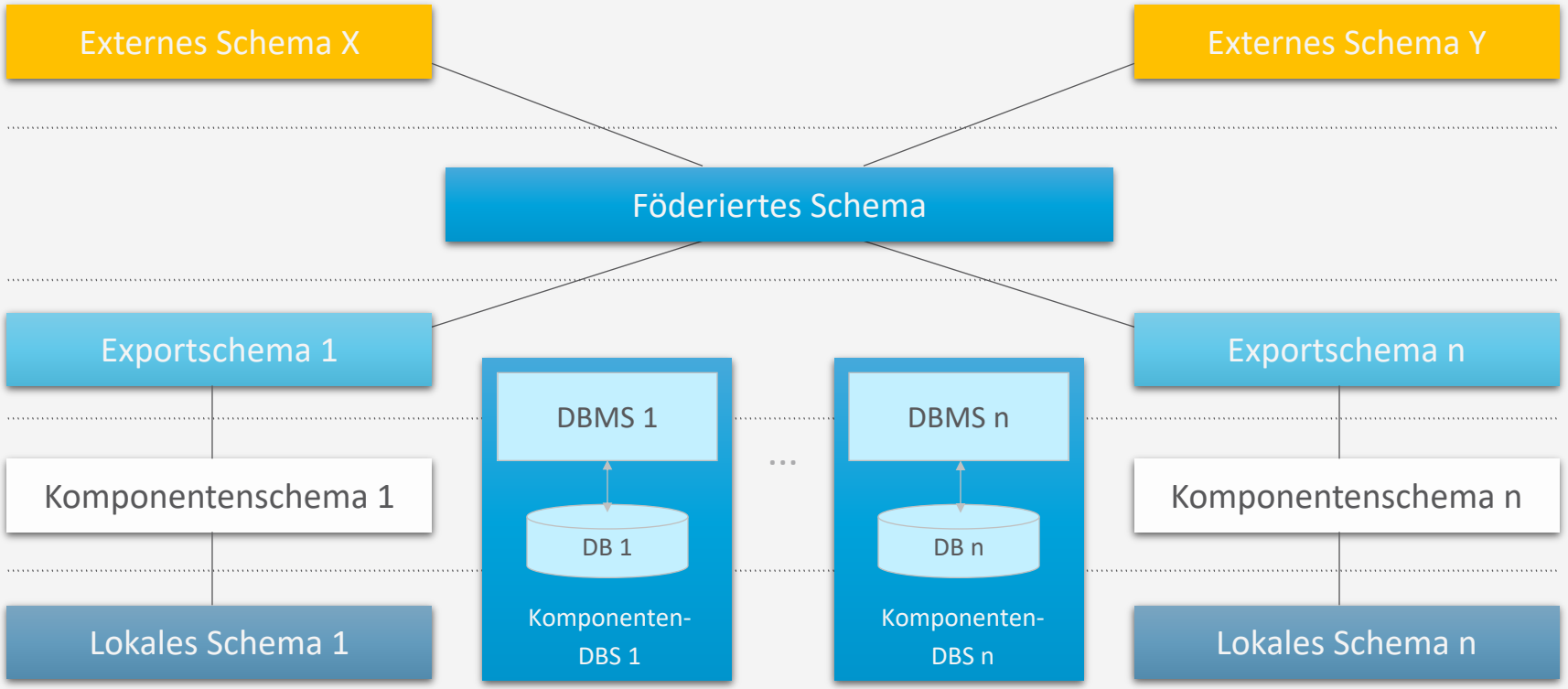


Was soll ein föderiertes DBS können?

- Im Idealfall volle Funktionalität eines DBS, dabei Verbergen der Verteilung bzgl.
  - Transaktionsverwaltung
  - Integritätskontrolle
  - Anfrageverarbeitung und -optimierung
  - Recovery

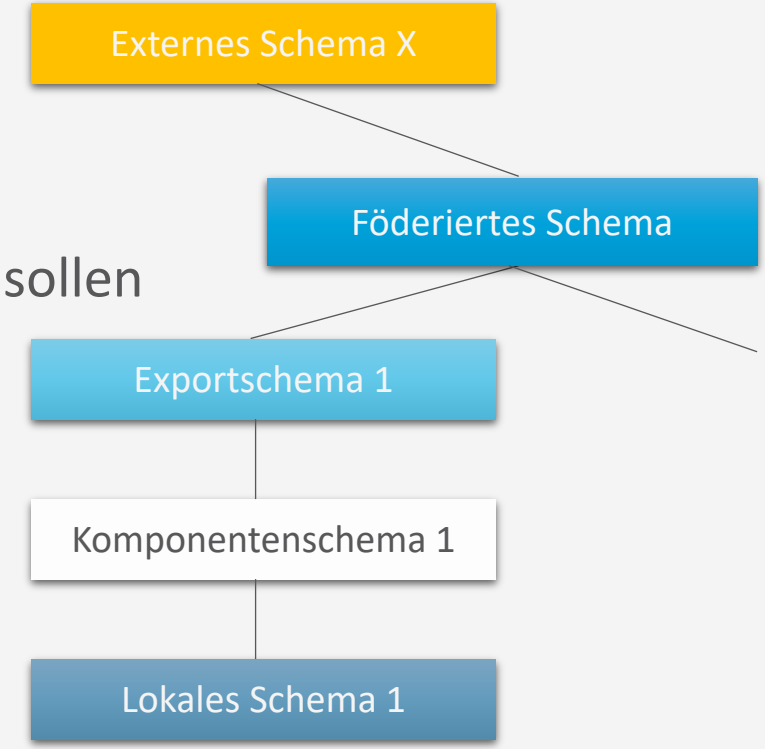
  auf globaler Ebene (im Föderierungsdienst) → Insgesamt hohen Aufwand
- Pragmatischer Ansatz:
  - Nur solche Funktionalität im föderierten System realisieren, die tatsächlich benötigt wird – eine stark anwendungsgetriebene Diskussion

5-Schichten-Architektur



Schemaintegration

- Lokales Schema
  - Datenschema des lokalen DBS
- Komponentenschema
  - Schema des lokalen DBS im globalen Datenmodell
- Exportschema
  - Teile des lokalen Schemas, die in Föderation eingebracht werden sollen
- Föderiertes Schema
  - Globales Schema des föderierten DBS
- Externes Schema
  - Schema für einzelne Anwendungen bzw. Benutzer



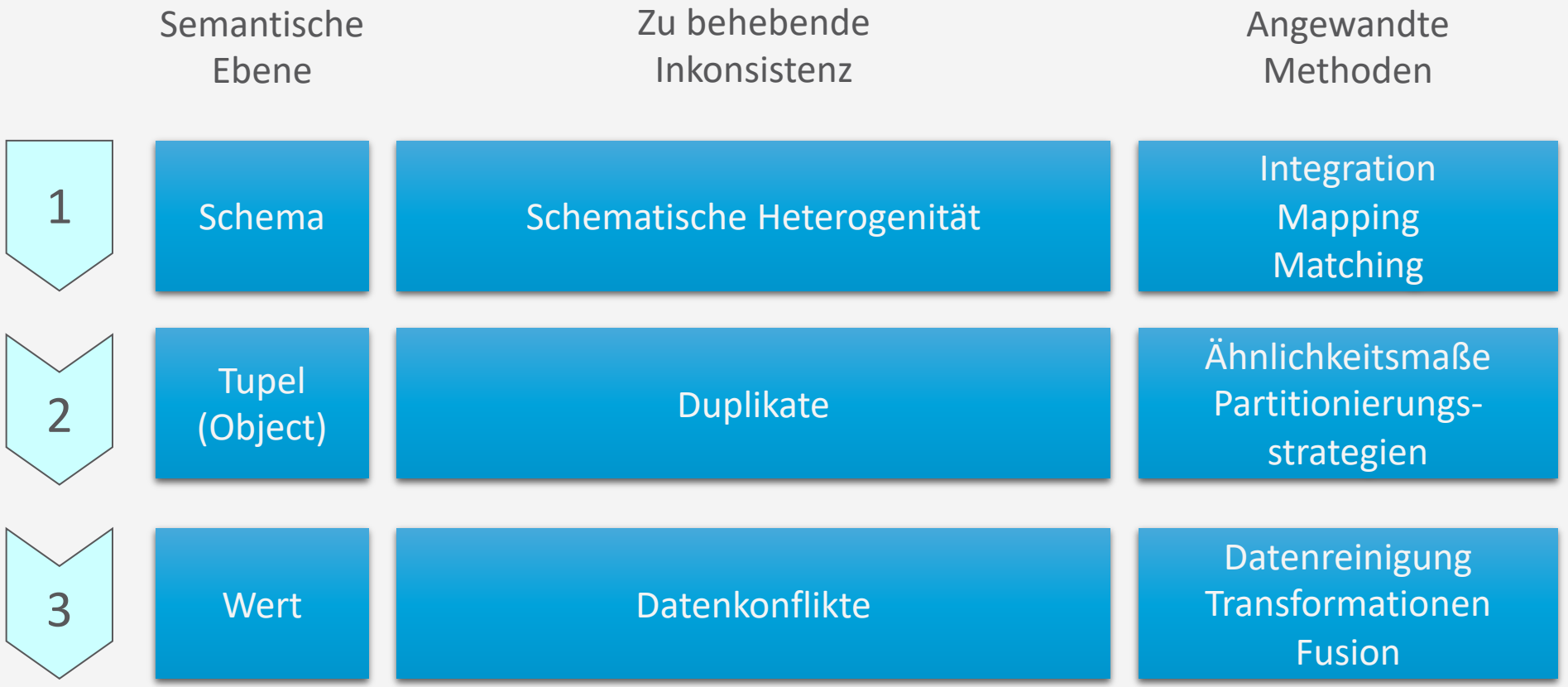
Schemaintegration

- Vorgehen
  - Schematransformation:
    - Übersetzung der lokalen Schemata in Komponentenschemata (semi-automatisch oder interaktiv anhand von Transformationsregeln)
  - Verhandlung (unter Domänenexperten und Datenbankdesignern):
    - Anwendungsorientierte Diskussion der angestrebten Exportschemata
  - Eigentliche Schemaintegration:
    - Vereinigung der Exportschemata zu föderiertem Schema

- Probleme
  - Synonyme: gleiche Bedeutung bei unterschiedlicher Benennung
  - Homonyme: gleiche Namen bei unterschiedlicher Bedeutung
  - Ähnlichkeit, Spezialisierung, Überschneidung

Forschungsgebiet u.a.: Ontology-based data access (OBDA)
Gemeinsame Ontologie definieren, für Übersetzung von globalen Anfragen in lokale Anfragen / Zusammenstellung der Einzelergebnisse in ein Gesamtergebnis
- Anwendung z.B. in der Zusammenführung von medizinischen Daten aus unterschiedlichen Krankenhausinformationssystemen

Schemaintegration → Datenintegration

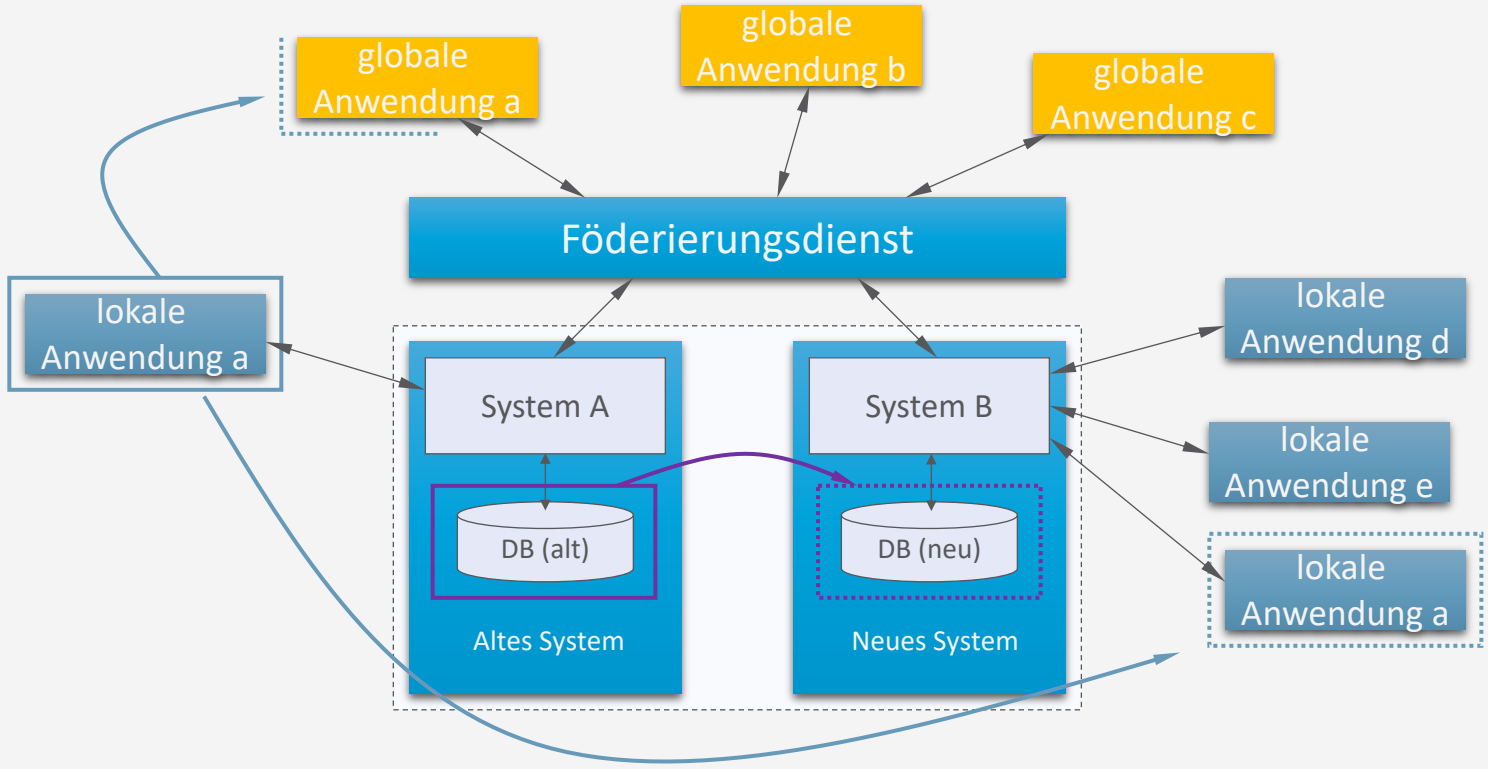


Beispiele für föderierte DBS

- Ablösung von Altsystemen (legacy systems) – z.B. wegen ...
  - Ansteigender Wartungskosten
  - Nachlassender Unterstützung durch Systemhersteller
  - Überlastung des Altsystems in vorliegender bzw. perspektivischer Anwendungssituation – keine ausreichenden Skalierungsmöglichkeiten
  - Reorganisation betrieblicher Strukturen – z.B. aufgrund einer Fusionen mit anderem Unternehmen, die letztlich die Vereinheitlichung der Systemplattformen notwendig macht
- Wichtige Anforderungen dabei
  - Investitionsschutz
    - Erhalt von Daten
    - Erhalt von Anwendungsprogrammen
  - Fortlaufender Betrieb während Migration

Migration in kleinen Schritten

- Für fortlaufenden Betrieb
  - System B aufsetzen, Föderierungsdienst definieren
  - Lokale Anwendung a in globale Anwendung a übersetzen
  - Daten aus System A übertragen
  - Lokale Anwendung a an System B überführen, dann alte lokale Anwendung a und globale Anwendung a löschen



Zwischenzusammenfassung

- Föderierte Datenbanken
  - Einheitliche Sicht auf unterschiedliche DBs
  - Integration
  - Systematische Migration
  - ... von Daten und Anwendungen

Überblick: 8. Verteilte Datenbanken

A. *Verteilte DBMS*
   - Fragmentierung, Replikation, Allokation
   - Transparenz
   - CAP-Theorem

B. *Anfragenbeantwortung in verteilten Systemen*
   - Anfrageverarbeitung
   - Transaktionskontrolle, Sperrverwaltung, Deadlockvermeidung

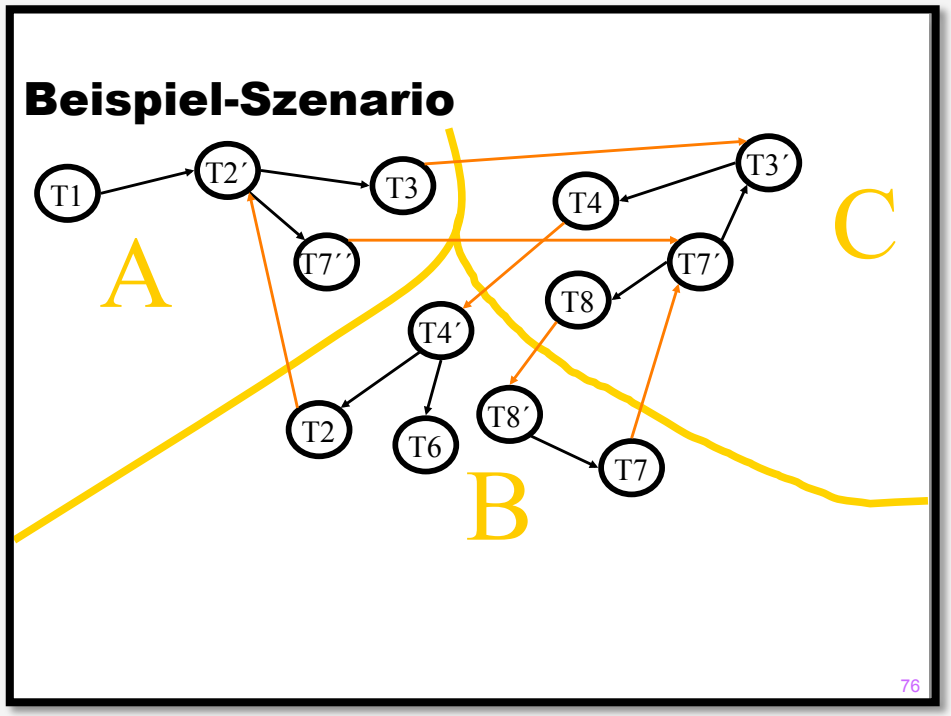
C. *Föderierte DBS*
   - Integration
   - Migration

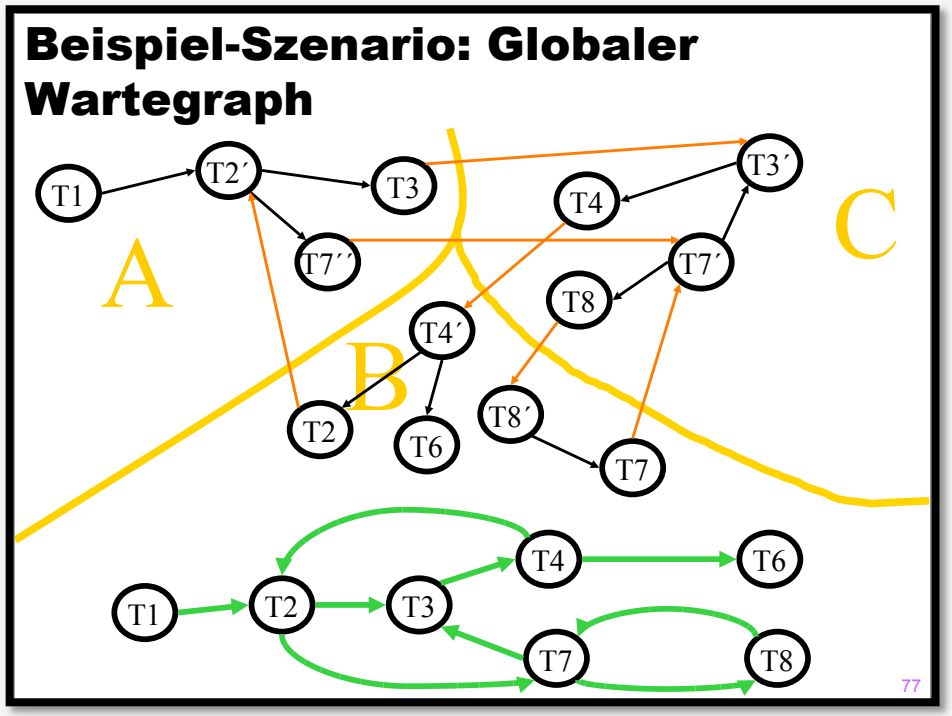
Ende

Anhang

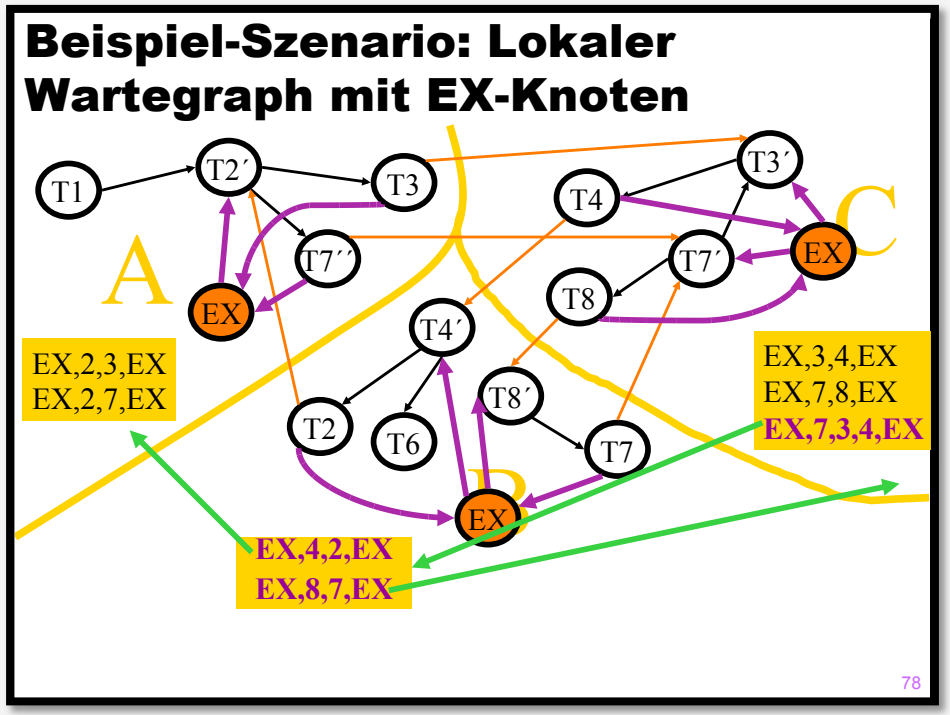
Dezentrale Deadlock-Erkennung: Größeres Beispiel

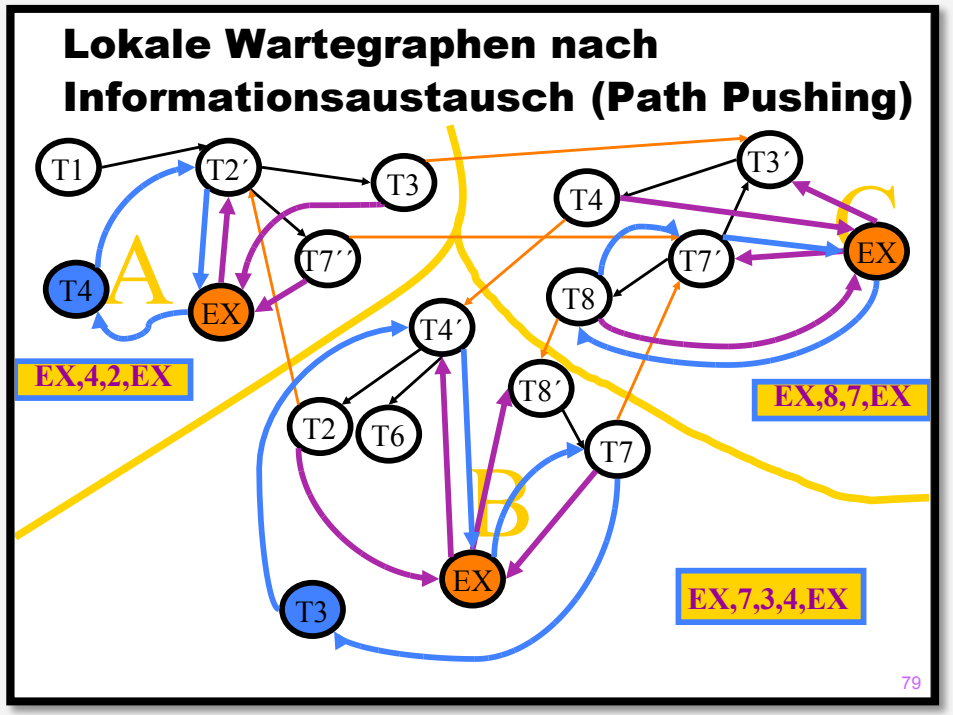
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)





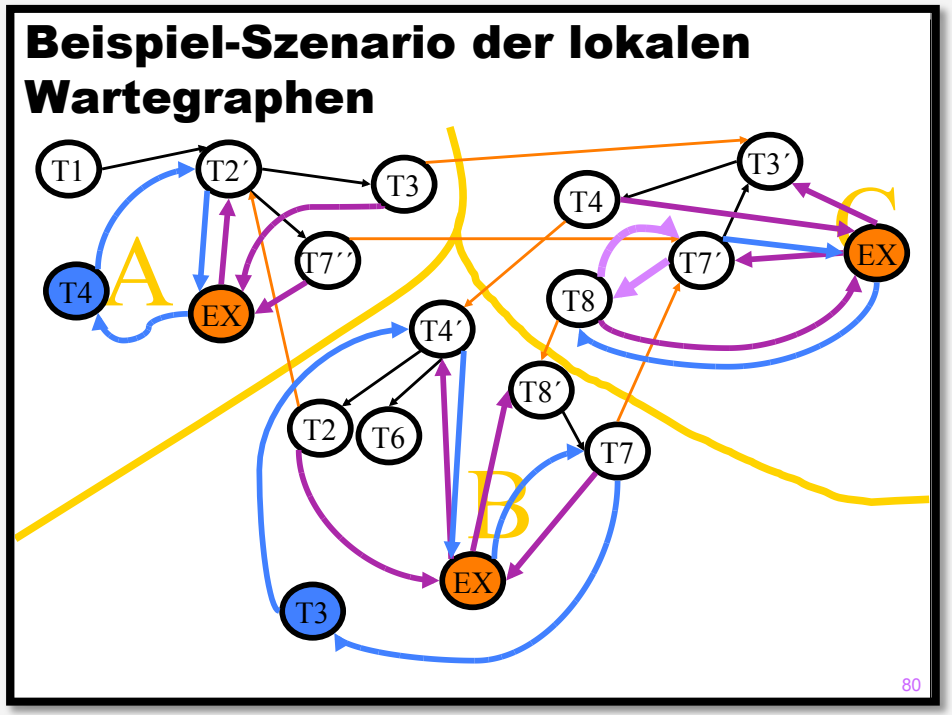
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)

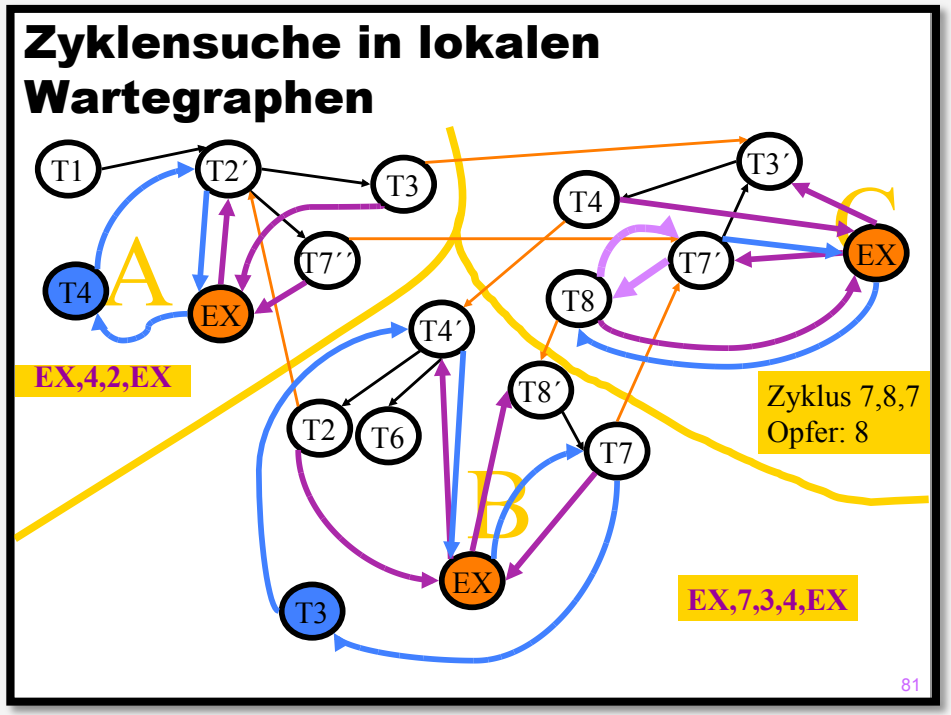




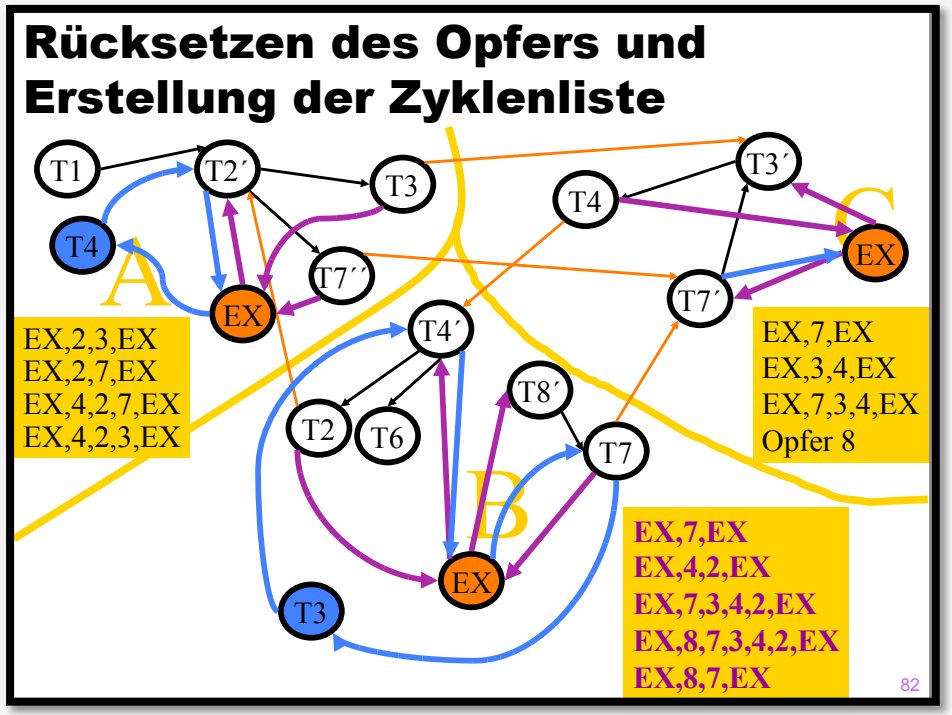
https://db.in.tum.de/teaching/ws1415/verteilteWebInf/slides/Teil4.pdf?lang=de

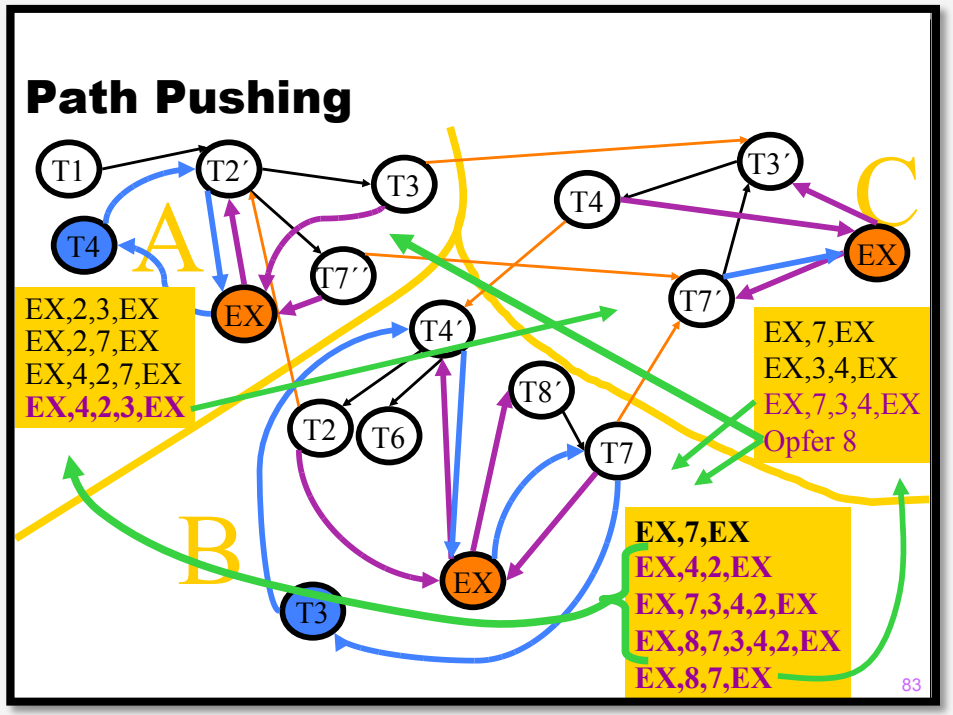
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)



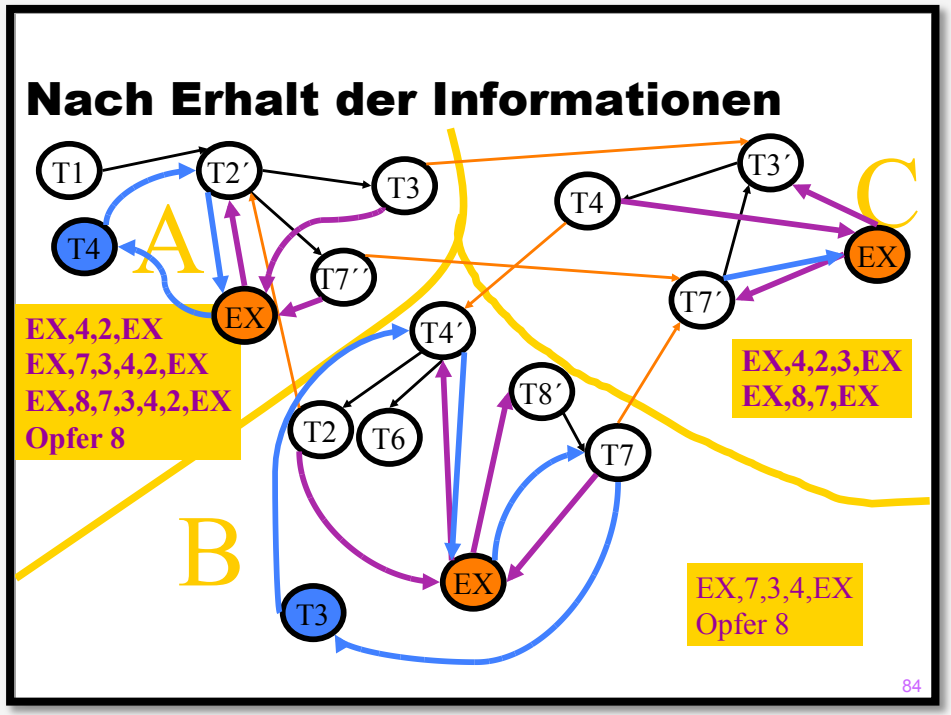


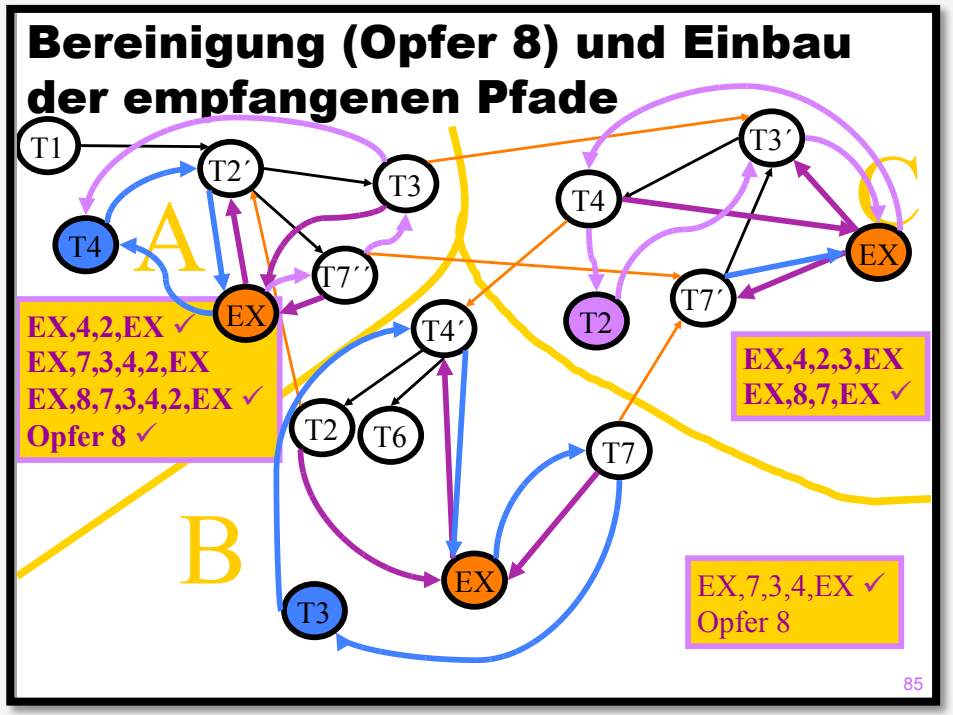
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)



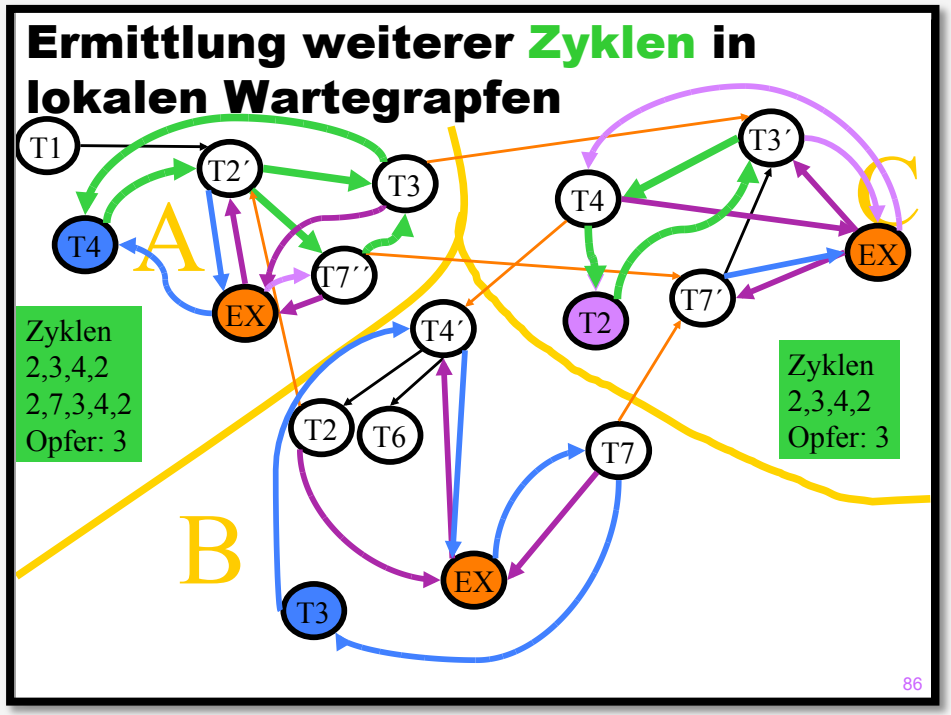


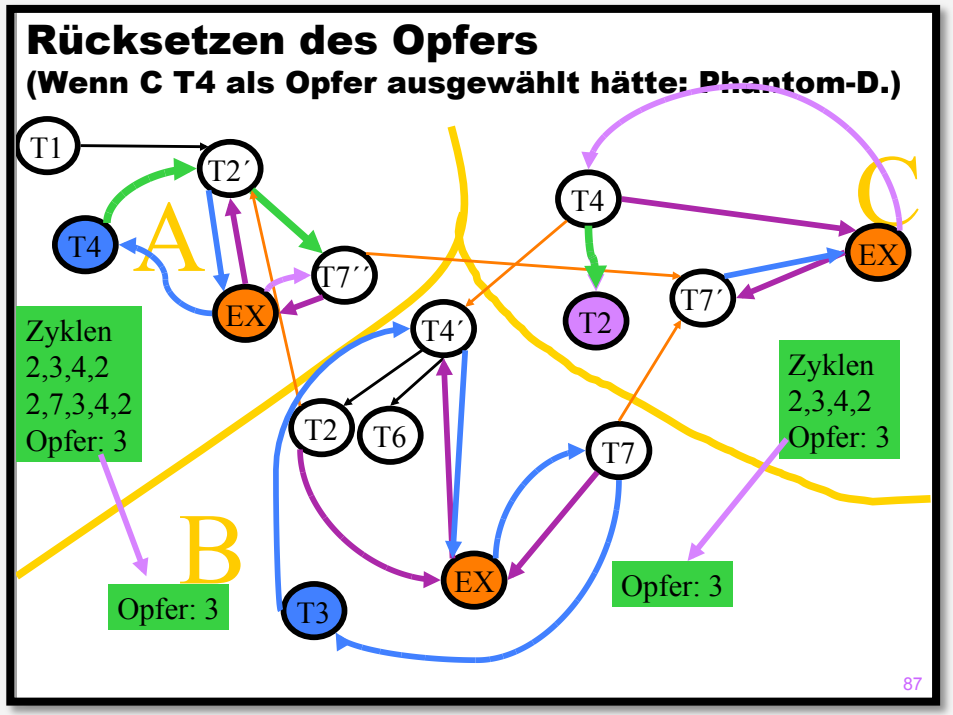
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)



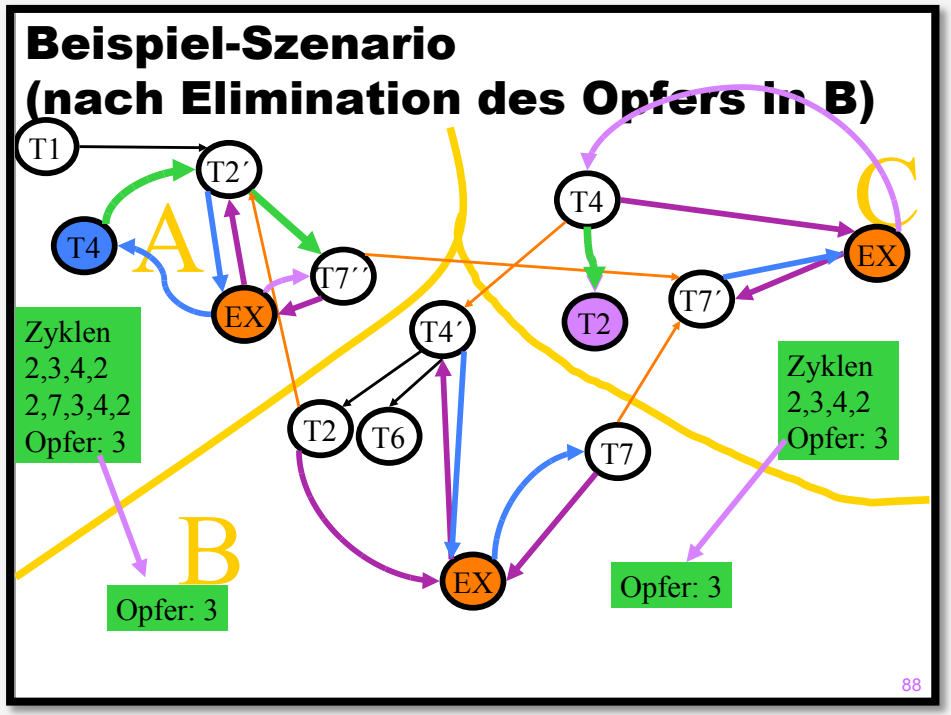


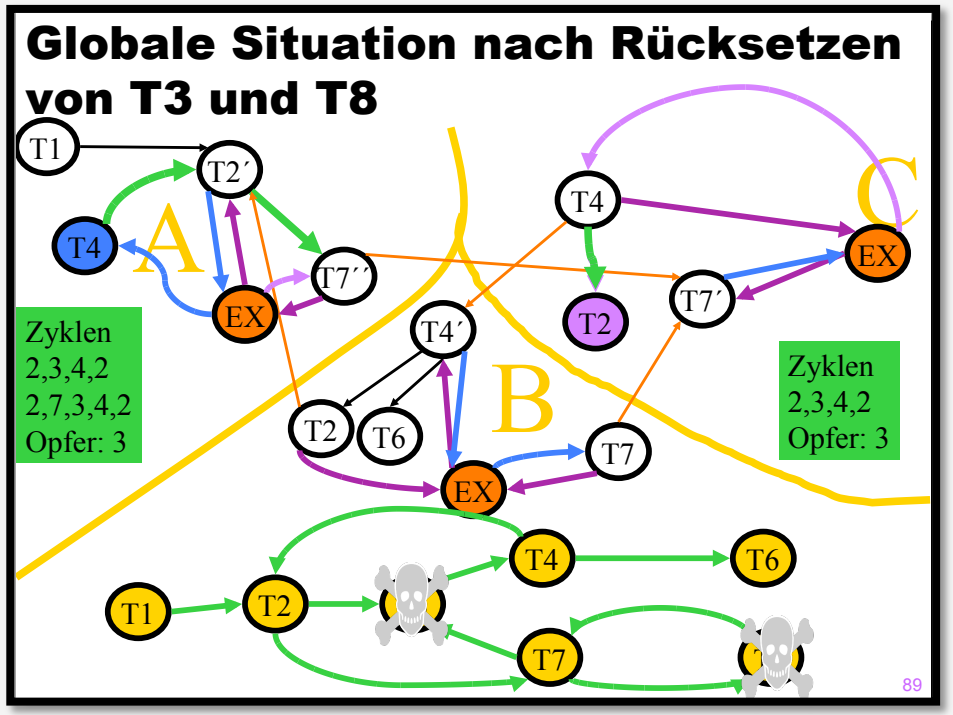
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)



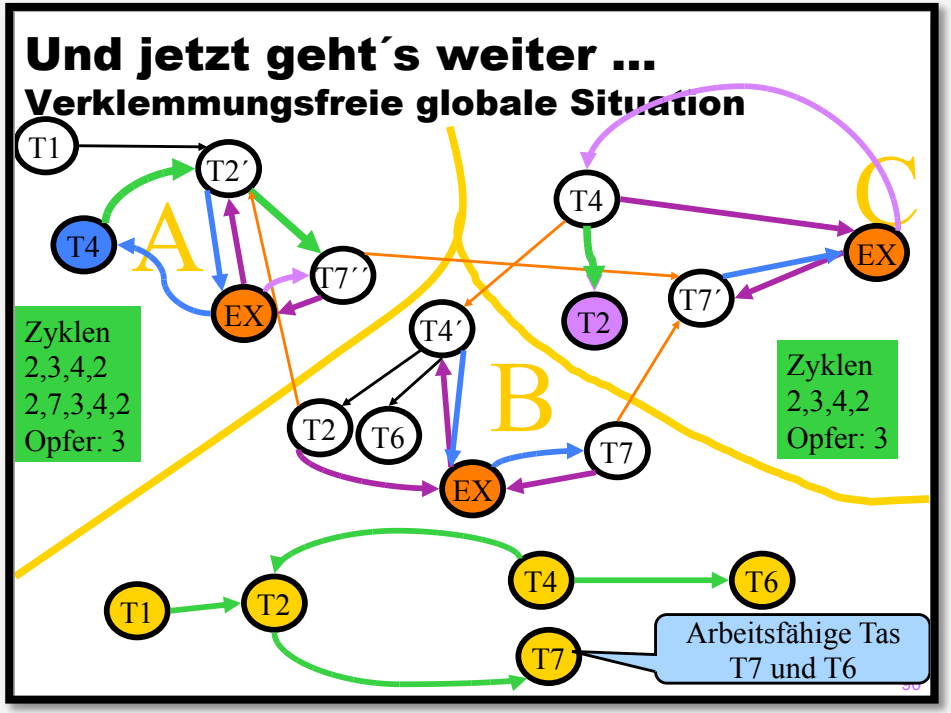


Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)





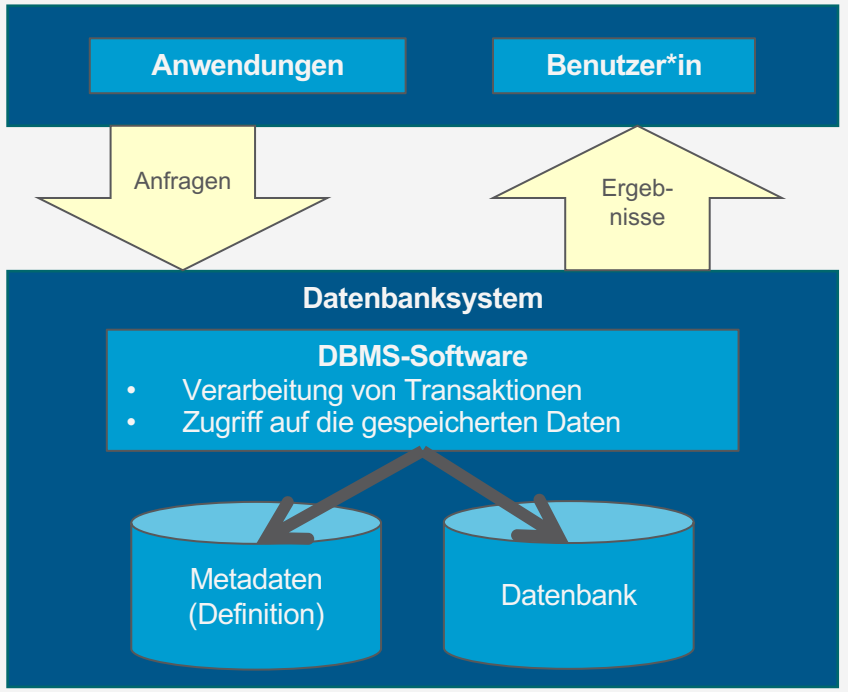
Beispiel zu Path Pushing nach Kemper (Ordnung $i > j$)



WWU MÜNSTER

Ende

Datenbanken



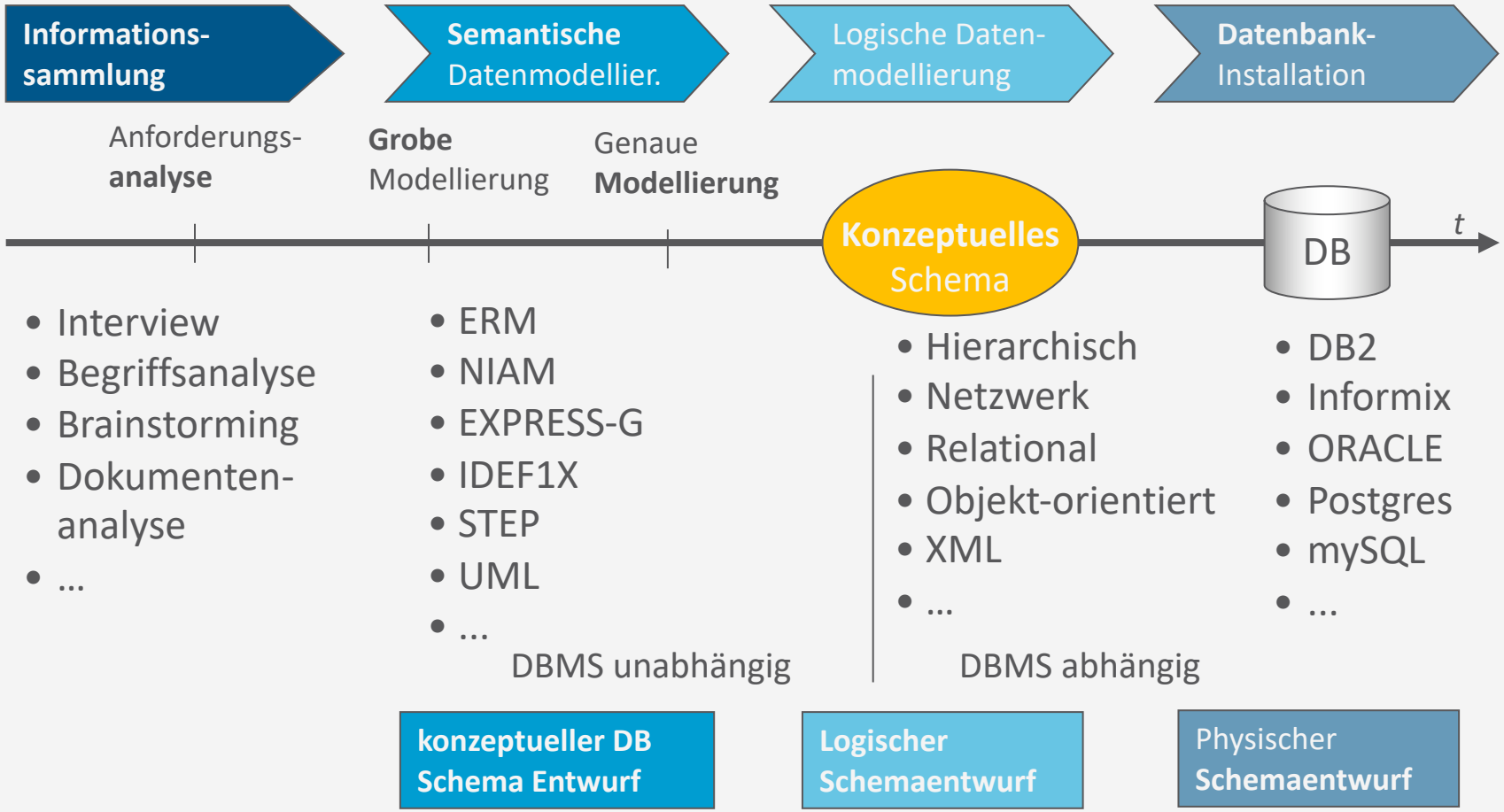
Tanya Braun
Arbeitsgruppe Data Science, Institut für Informatik

Inhalte: Datenbanken (DBs)

1. **Einführung**
   - Anwendungen
   - Datenbankmanagementsysteme
2. **Datenbank-Modellierung**
   - Entity-Relationship-Modell (ER-Modell)
   - Beziehung zwischen ER und UML
3. **Das relationale Modell**
   - Relationales Datenmodell (RM)
   - Vom ER-Modell zum RM
   - Relationale Algebra als Anfragesprache
4. **Datenbank-Entwurf**
   - Funktionale Abhängigkeiten
   - Normalformen
5. **Structured Query Language (SQL)**
   - Datendefinition
   - Datenmanipulation
6. **Anfrageverarbeitung**
   - Architektur
   - Indexierung
   - Anfragepläne, Optimierung
7. **Transaktionen**
   - Transaktionsverarbeitung, Schedules, Sperren
   - Wiederherstellung
8. **Verteilte Datenbanken**
   - Fragmentierung, Replikation, Allokation; Anfrageverarbeitung; Föderierte DBs

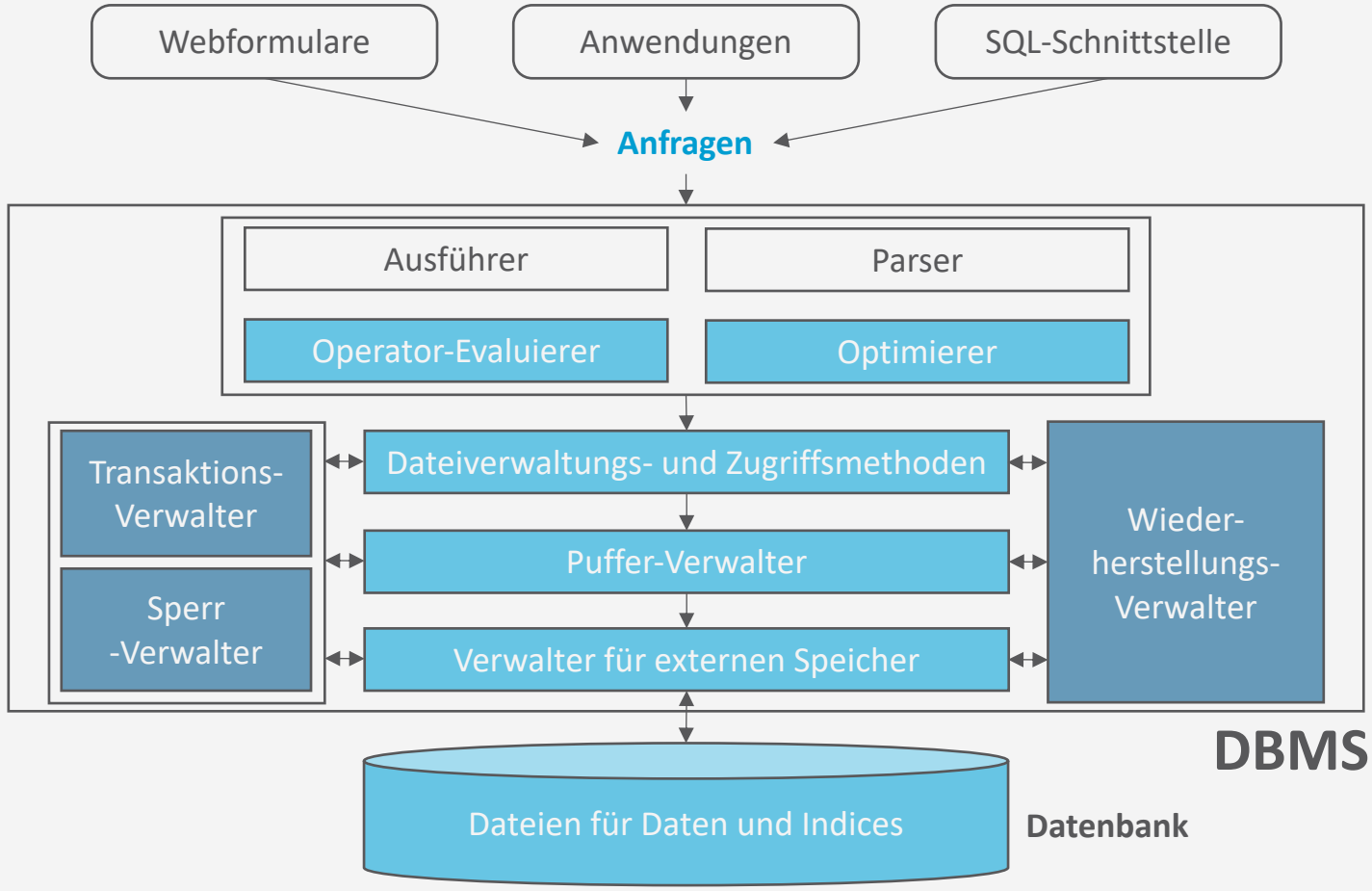
Phasen des DB-Entwurfs

- Ausblick: Von der Anwendung her
  - Teil von 2. DB-Modellierung
    - Methode: ERM
  - Teil von 3. Das relationale Datenmodell
    - Methode: relationale Modellierung
  - Teil von 4. DB-Entwurf
  - Teil von 5. SQL & Übergang zu „Hinter den Kulissen“



Architektur eines DBMS

- Ausblick: Hinter den Kulissen
  - Teil von 4. Relational Datenmodell – Relationale Algebra und 5. SQL
    - Anfragen stellen
    - Daten ändern
  - Teil von 6. Anfrageverarbeitung
    - Speicherung und Verwaltung der DB
    - Effiziente Umsetzung der SQL Befehle im DBMS
  - Teil von 7. Transaktionen
    - ACID-Umsetzung



Skript, Literatur, Quellen

- Skript: Folien und Vorlesung vor Ort

  In Klausuraufgaben sind nur Konstrukte und Notationen zu nutzen, die auch in der Vorlesung eingeführt worden sind.

- Literatur
  - **A. Kemper, A. Eickler:** ***Datenbanksysteme: Eine Einführung*****.**
  - R. Elmasri, S.B. Navathe: *Grundlagen von Datenbanksystemen*.
  - A. Silberschatz, H. F. Korth, S. Sudarshan: *Database System Concepts*.

- Folien (wenn nicht anders angegeben; fast immer angepasst)
  - Vorlesungsfolien der Vorlesung „Informationssysteme I“ Prof. Dr. Daniela Nicklas
  - Vorlesungsfolien der Vorlesung „Datenbanken“ Prof. Dr. Ralf Möller, Dr. Özgür Öczep

Klausur

- 1. Termin: 20.7.2023, 12.00-14.xx Uhr
- 2. Termin: 19.9.2023, 9.00-11.xx Uhr
- Ort: *folgt im Learnweb*
- Dauer: 120 Minuten

- Zugelassene Hilfsmittel: KEINE

- Klausur
  - 100 Punkte insgesamt
  - 50 Punkte zum Bestehen

Klausur: Aufbau

- Klausurthemen: Kapitel 2 bis 7
- 5 Aufgaben mit insgesamt 100 Punkten
  1. ER-Modellierung / relationales Modell (Kapitel 2 + 3)
  2. Entwurfstheorie (Kapitel 4)
  3. Relationale Algebra / SQL (In Kapitel 3 + 5)
  4. Anfrageverarbeitung (Kapitel 6)
  5. Transaktionen (Kapitel 7)

Klausur: Beispielaufgabentypen

- *Verständnisaufgaben*: zu Zusammenhängen zwischen den Inhalten
- *Anwendungsaufgaben*: Anwenden von Algorithmen, Durchführen von Modellierungen / Transformationen
- Beispiele (nicht abschließend)
  - Gegeben ein Text, erstellen Sie das ER-Diagramm
  - Gegeben ein ER-Diagramm, stellen Sie das relationale Modell dazu auf
  - Gegeben ein relationales Modell, definieren Sie die DB mittels SQL

  - Formulieren Sie eine gegebene Frage in relationaler Algebra / SQL
  - Übersetzen Sie eine gegebene Anfrage in relationaler Algebra / SQL in das jeweils andere
  - Gegeben Tabellen, bestimmen Sie das Ergebnis einer gegebenen SQL-Anfrage / Anfrage in relationaler Algebra

Klausur: Beispielaufgabentypen

- Fortsetzung Beispiele (nicht abschließend)
  - Gegeben eine Menge von FDs, zeigen Sie, ob eine Zerlegung verlustlos / abhängigkeitserhaltend ist
  - Gegeben eine Menge von FDs, berechnen Sie die kanonische Überdeckung / Schlüssel / etc.
  - Gegeben eine Menge von FDs, bestimmen Sie die höchste NF, die sie erfüllt
  - Gegeben eine Menge von FDs, überführen Sie das Schema in eine bestimmte NF
    - Zum Beispiel Synthesealgorithmus, Zerlegungsalgorithmus

  - Gegeben ein Index, fügen Sie ein Datum ein / löschen Sie ein Datum
  - Vergleichen Sie zwei Anfragepläne hinsichtlicht ihres Aufwandes

Klausur: Beispielaufgabentypen

- Fortsetzung Beispiele (nicht abschließend)
  - Testen Sie einen Schedule auf Serialisierbarkeit
  - Gegeben ein Schedule, erklären Sie, welche Anomalie vorliegt
  - Gegeben ein Schedule, bestimmen Sie, ob ein Deadlock vorliegt
  - Gegeben ein Ausschnitt aus einer Log-Datei, bestimmen Sie die redo / undo Mengen

  - Zusammenhänge zwischen ER / relationales Datenmodell, relationales Datenmodell + relationale Algebra / SQL, ER / relationales Datenmodell / Entwurfstheorie, DB / Transaktionsmanagement, …; warum sind manche Dinge einfach / manche Dinge schwer / welche Probleme gibt es …

Bei Fragen

- Q&A am 5.7.2023
  - Kein neuer Inhalt
  - Das Treffen geht, so lange Sie Fragen haben, aber maximal 90 Minuten

- Learnweb-Forum
  - Dann hat auch jede*r was davon
  - Jede*r ist aufgerufen, sich an den Fragen / Diskussionen zu beteiligen
    - Nicht immer auf uns warten (müssen)
    - Sollte etwas Falsches dabei sein, melden wir uns