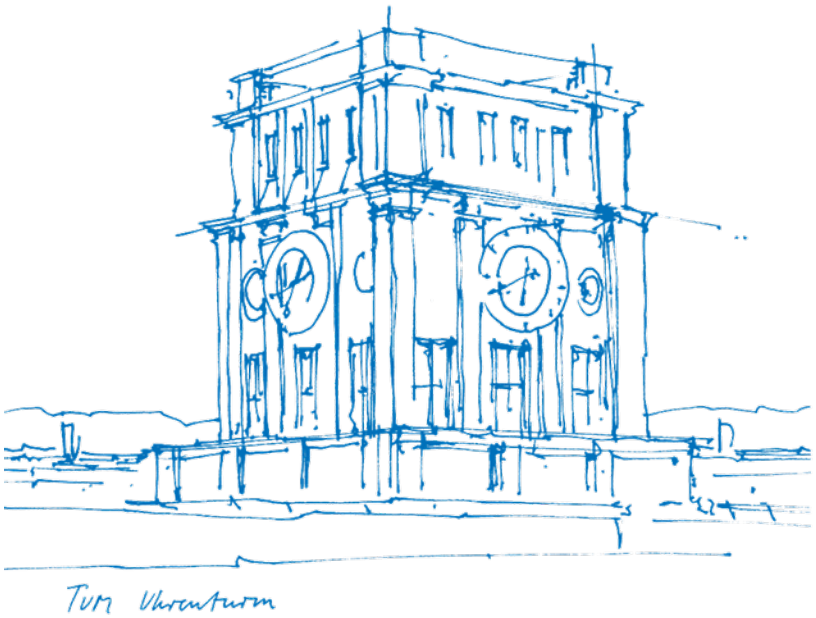
Grundlagen Datenbanken

Benjamin Wagner

15. Januar 2019



Allgemeines

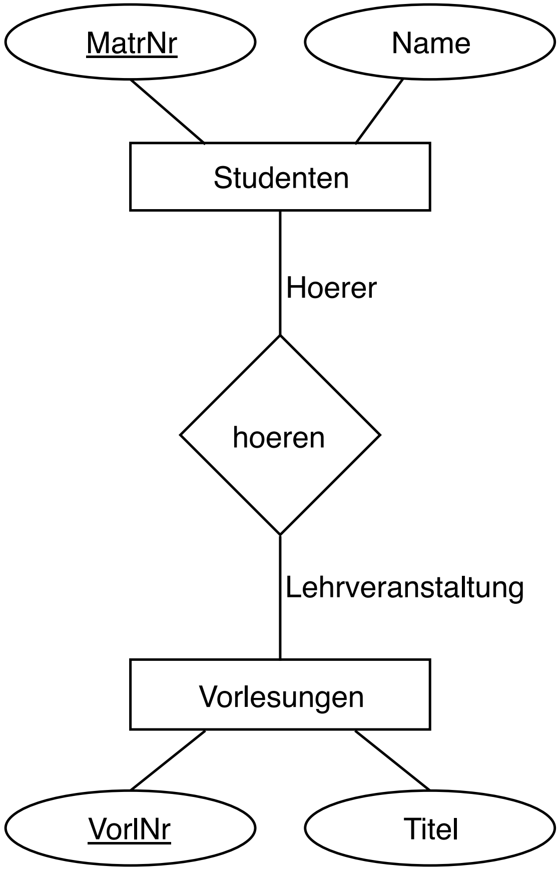
- Folien von mir sollen unterstützend dienen. Sie sind nicht von der Übungsleitung abgesegnet und haben keinen Anspruch auf Vollständigkeit (oder Richtigkeit).
- Bei Fragen oder Korrekturvorschlägen: wagnerbe@in.tum.de
- Vorlesungsbegleitendes Buch von Professor Kemper (Chemiebib)
- Mein Foliensatz ist online: `https://github.com/wagjamin/GDB2018`

Entity/Relationship-Modellierung

- **Entity:** Gegenstandstyp, welcher mit anderen Gegenständen in Beziehung steht
- **Relationship:** Modelliert die Beziehung zwischen Entities
- **Attribut:** Eine Eigenschaft einer Entity
- **Schlüssel:** Identifiziert eindeutig einen Datensatz
- **Rolle:** Welche Rolle nimmt eine Entity in einer Beziehung ein

$\Rightarrow$ Lässt sich als Graph darstellen, siehe Universitätsschema

Beispiel: Schema

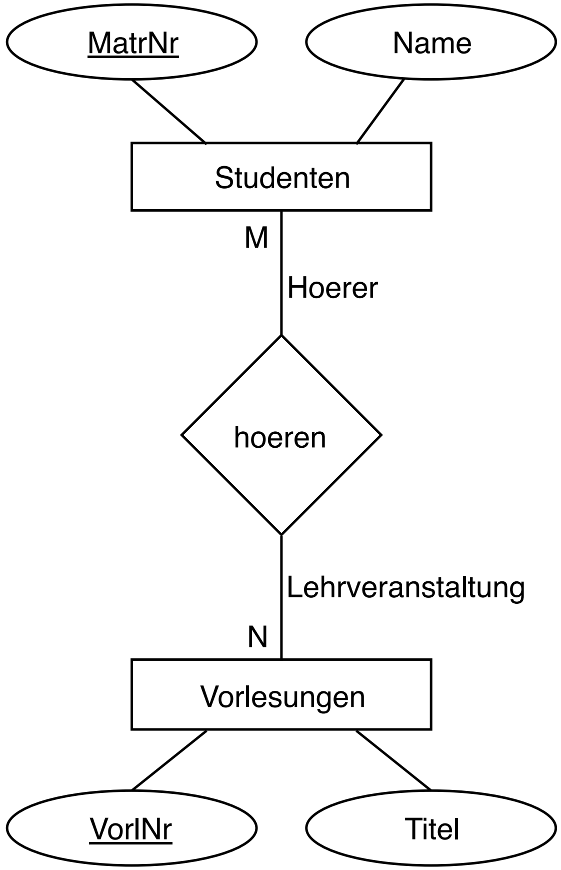


- Repräsentiert Studenten, die bestimmte Vorlesungen hören
- Schlüssel sind unterstrichen, ein Student ist eindeutig durch seine MatrNr bestimmt
- Hören modelliert eine Relationship zwischen Studenten und Vorlesungen
- Studenten treten hier in Rolle "Hörer" auf

Funktionalitäten

- Für eine Relationship R zwischen zwei Entities E_1 und E_2 gilt:
  $R \subset E_1 \times E_2$
- Funktionalitäten charakterisieren die Relationship
- Mögliche Funktionalitäten: 1:1, 1:N, N:1, N:M
- Das kann auf Relationships mit vielen Entities ausgedehnt werden
- **Beispiel?**

Beispiel: Funktionalitäten



- Nun mit Funktionalitätsangaben
- Ein Student kann N Vorlesungen hören
- Eine Vorlesung kann von M Studenten gehört werden

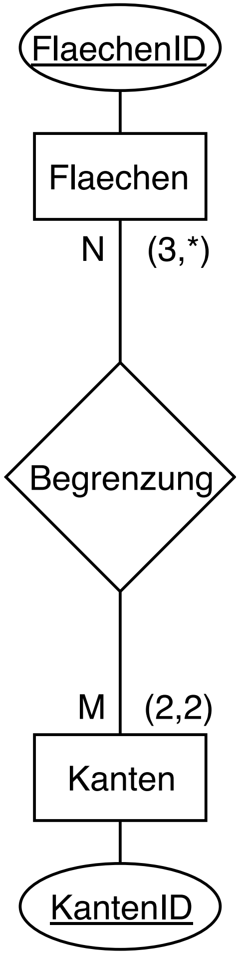
(min, max)-Notation

- Ergänzt Funktionalitätsangaben
- **Achtung:** Eines ersetzt nicht das Andere!
- Betrachte Relationship $R \subset E_1 \times E_2$
- (min_1, max_1) bei E_1 bedeutet:

  Für alle $e \in E_1$: mindestens min_1 Tupel $(e, \ldots) \in R$

  Für alle $e \in E_1$: maximal max_1 Tupel $(e, \ldots) \in R$

Beispiel: (min, max)-Notation



- **Funktionalitäten sagen aus:**
  Eine Fläche kann M Kanten haben
  Eine Kante kann N Flächen begrenzen
- **(min, max) sagt aus:**
  Eine Fläche muss von mehr als drei Kanten begrenzt werden
  Eine Kante begrenzt genau zwei Flächen
- Volles Beispiel in den Folien

Sonstige Konzepte

- Exitenzabhängige Entities: Funktionalität immer 1:N oder 1:1
- Generalisierung: ”is-a”-Relationship
- Aggregation: ”teil-von”-Relationship
- Das kann alles mit UML modelliert werden

Das relationale Modell

- Es gibt Domänen $D_1, D_2, ..., D_n$, das entspricht Wertebereichen
  z.B. Integer, Strings, Chars, Booleans
- Für eine Relation R gilt: $R \subset D_1 \times D_2 \times ... \times D_n$
- Ein Tupel ist ein Element einer Relation
- Das Schema gibt die Struktur der Relationen vor

Das relationale Modell

- Es gibt Domänen $D_1, D_2, ..., D_n$, das entspricht Wertebereichen
  z.B. Integer, Strings, Chars, Booleans
- Für eine Relation R gilt: $R \subset D_1 \times D_2 \times ... \times D_n$
- Ein Tupel ist ein Element einer Relation
- Das Schema gibt die Struktur der Relationen vor
- Sonstige Begriffe:
  Ausprägung: der aktuelle Zustand einer Relation
  Schlüssel: minimale Teilmenge von Attributen, welche Tupel eindeutig identifiziert
  Primärschlüssel: Einer der Schlüsselkandidaten

Relationale Modellierung

- Wir können eine Relation nun aufschreiben:
  User:{[Cust_Id, Name, Bday, Credit_Card]}
- Es können Datentypen ergänzt werden:
  User:{[Cust_Id: Integer, Name: String, Bday: Date ...}
- Falls partielle Funktionen gelten kann das Schema verfeinert werden
- Das darf aber nur bei gleichem Schlüssel passieren
- **Achtung:** NULL-Werte sind zu vermeiden

Relationale Algebra

- Beschreibt auf abstrakte Art und Weise Anfragen an die Datenbank
- Trotzdem in der Realität wichtig ($\rightarrow$ später)
- Beachte: Es gibt eine ganze Reihe verschiedener Joins

Symbol	Bedeutung
$\sigma_{Kondition}$	Selektion
$\prod_{Attribute}$	Projektion
$\times$	Kreuzprodukt
$\rho_{neu \leftarrow alt}$	Umbenennung
$\bowtie$	Join
$-, +, \div, \cup, \cap$	Mengenoperationen

Wichtigste Operatoren, **nicht vollständig**

Relationale Division

- Divisionsoperator sorgt oft für Verwirrung
- Kann bei Aussagen mit Allquantoren verwendet werden
- Bei $R \div S$ muss immer gelten: $Schema(S) \subset Schema(R)$
- Das Schema des Ergebnisses ist dann: $Schema(R) / Schema(S)$
- Unpräzise: es werden Tupel in R gesucht, welche für **jedes** Tupel in S einen Match haben

Relationale Division - $R \div S$

a1	a2	a3
1	2	1
1	2	2
2	1	5
3	5	1
3	5	2
3	5	3
4	8	1
4	8	2
4	6	3
5	5	1
5	5	2
5	5	3
5	5	4

$\div$

a3
1
2
3

$=$

a1	a2
3	5
5	5

Kalküle

- **Tupelkalkül**: Schreibweise (hoffentlich) aus Mathe-Vorlesungen bekannt: $\{t|P(t)\}$, mit $P(t)$ aussagenlogischer Formel
- **Domänenkalkül:** Domänenvariablen: $\{[v_1, ..., v_n]|P(v_1, ..., v_n)\}$
- **Achtung**: "Sicherheit" muss in Tupel- und Domänenkalkül sichergestellt sein. D.h. keine unendlichen Ergebnisse.
- **Mächtigkeit**: Relationale Algebra, Tupel- und Domänenkalkül gleich mächtig

Wiederholung: Relationen

Professoren:

PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
...	...	...	...

- Jede Tabelle hat Spalten: **Attribute**
- Die einzelnen Zeilen nennt man: **Tupel**
- Jede Spalte hat einen: **Typ**
- **Schlüssel** markieren ein Tupel eindeutig

SQL

- Standard Anfragesprache für relationale Datenbanken
- Web-Interface: `http://hyper-db.de/interface.html`
- Möglichkeit, Anfragen auf Uni-Schema zu realisieren
- Läuft auf Hyper (Datenbank des Lehrstuhls)
- Grundstruktur einer SQL-Anfrage:

```
SELECT ...
FROM ...
WHERE ...
```

SQL - Datentypen

- Es gibt eine Reihe von Datentypen in SQL
- Z.B: char(n), varchar(n), integer, blob, date ...
- Damit können Tabellen erstellt werden:

```sql
CREATE TABLE Customers(
        CustId     integer not null,
        Name       varchar(30) not null,
        Birthday   date
);
```

SQL - Einfache Anfrage

- Suche Namen aller Professor*innen, deren Rang C4 ist

SQL - Einfache Anfrage

- Suche Namen aller Professor*innen, deren Rang C4 ist

```sql
SELECT Name
FROM Professoren
WHERE Rang = 'C4'
```

- Suche alle Studierenden, die seit mehr als vier Semestern studieren

SQL - Einfache Anfrage

- Suche Namen aller Professor*innen, deren Rang C4 ist

```sql
SELECT Name
FROM Professoren
WHERE Rang = 'C4'
```

- Suche alle Studierenden, die seit mehr als vier Semestern studieren

```sql
SELECT *
FROM Studenten
WHERE Semester > 4
```

SQL - Sprachkonstrukte

- **Kreuzprodukt von Relationen** - *from R1, R2*
- **Aufgabe:** Was ist der Name, des Professors, der ’Ethik’ liest

SQL - Sprachkonstrukte

- **Kreuzprodukt von Relationen** - *from R1, R2*
- **Aufgabe:** Was ist der Name, des Professors, der 'Ethik' liest

```sql
SELECT Professoren.Name
FROM Professoren, Vorlesungen
WHERE Professoren.persNr =
        Vorlesungen.gelesenVon
         AND Vorlesungen.Titel = 'Ethik'
```

SQL - Sprachkonstrukte

- **Duplikateliminierung** - *select distinct*
- **Aufgabe:** Suche das Semester aller Studierenden, die Logik hören

SQL - Sprachkonstrukte

- **Duplikateliminerung** - *select distinct*
- **Aufgabe:** Suche das Semester aller Studierenden, die Logik hören

```sql
SELECT DISTINCT studenten.semester
FROM studenten, hoeren, voerlesungen
WHERE studenten.matrnr = hoeren.matrnr
      AND hoeren.vorlNr = vorlesungen.vorlNr
      AND vorlesungen.titel = 'Logik'
```

SQL - Sprachkonstrukte

- **Relation benennen** - *from Professoren p1, Professoren p2*
- **Mengenoperationen** - *union, intersects, minus*
- **Quantor** - *exists*

```
(SELECT p.Name
FROM Professoren p
WHERE NOT EXISTS (
        SELECT *
        FROM Vorlesungen v
        WHERE v.gelesenVon = p.persNr);)
INTERSECT
(...)
```

SQL - Sprachkonstrukte

- **Gruppierung** - *group by*
- Bildet Gruppen von Tupeln mit den selben Werten in den Attributen der "group by" Klausel
- Auf den anderen Attributen können dann Aggregatsfunktionen aufgerufen werden
- **Aufgabe:** Wie viele Studenten studieren in welchem Semester?

SQL - Sprachkonstrukte

- **Gruppierung** - *group by*
- Bildet Gruppen von Tupeln mit den selben Werten in den Attributen der ”group by” Klausel
- Auf den anderen Attributen können dann Aggregatsfunktionen aufgerufen werden
- **Aufgabe:** Wie viele Studenten studieren in welchem Semester?

```sql
SELECT semester, count(*)
FROM Studenten
GROUP BY semester
```

SQL - Sprachkonstrukte

- **Gruppierung** - *group by*
- Es gibt viele Aggregatsfunktionen: avg, max, min, count, sum
- Für Selektion auf Aggregaten: *having*
- **Aufgabe:** Welche Professoren halten mehr als 2 Vorlesungen?

SQL - Sprachkonstrukte

- **Gruppierung** - *group by*
- Es gibt viele Aggregatsfunktionen: avg, max, min, count, sum
- Für Selektion auf Aggregaten: *having*
- **Aufgabe:** Welche Professoren halten mehr als 2 Vorlesungen?

```sql
SELECT p.Name, count(*)
FROM Professoren p, Vorlesungen v
WHERE p.persNr = v.gelesenVon
GROUP BY v.gelesenVon
HAVING count(*) > 2
```

SQL - Sprachkonstrukte

- **Temporäre Relation** - *with ... as()*
- Komplexe Anfragen können u.U. modularisiert werden

```sql
WITH h AS (SELECT VorlNr,
                  count(*) AS AnzProVorl
               FROM hoeren
               GROUP BY VorlNr),

(...)
```

SQL - Sprachkonstrukte

- **String Vergleiche** - *like ...*
- '_' dient als Placeholder für ein Zeichen
- '%' dient als Placeholder für beliebig viele Zeichen

```
SELECT *
FROM Studenten
WHERE name like 'T%eophrastos';
```

SQL - Sprachkonstrukte

- **Fallunterscheidung** - *case when ...*
- Die erste passende Bedingung wird ausgewertet

```sql
SELECT MatrNr, (CASE
            WHEN Note < 1.5 THEN 'sehr gut'
            WHEN Note < 2.5 THEN 'gut'
            WHEN Note < 3.5 THEN 'befriedigend'
            WHEN Note < 4.0 THEN 'ausreichend'
            ELSE 'nicht bestanden'
            END)
FROM prüfen;
```

SQL - Rekursion

- Unsere bisherigen Mittel reichen nicht ganz aus
- Beispiel: finde alle direkten und indirekten Vorgänger einer Vorlesung
- Hier hilft Rekursion
- **Idee:** definiere rekursiv eine Tabelle mit *with ... as*
- Nutze diese dann ganz normal weiter

SQL - Rekursion

- Rekursive Vorgänger-Nachfolger Relation
- Wir sehen: die Relation darf im *SELECT...* Teil verwendet werden

```
WITH RECURSIVE TransVorl(Vorg, Nachf) AS
   (SELECT Vorgaenger, Nachfolger
   FROM voraussetzen
   UNION ALL
   SELECT t.Vorg, v.Nachfolger
   FROM TransVorl t, Voraussetzen v
   WHERE t.Nachf = v.Vorgaenger)
```

SQL - Rekursion

- Und dann? Wir benutzen TransVorl ganz normal weiter...

```
SELECT Titel FROM Vorlesungen
WHERE VorlNr IN
 (SELECT Vorg
  FROM TransVorl where Nachf IN
  (SELECT VorlNr FROM Vorlesungen
   WHERE Titel= 'Der Wiener Kreis'))
```

Datenintegrität

- Wissen schon:
  Wie kann ich Schemata modellieren?
  Wie kann ich Anfragen an meine Datenbank formulieren?
- **Jetzt:** Wie stelle ich Korrektheit der Daten sicher?
- **Beispiel:** in einer Relationship soll immer auf einen existierenden Schlüssel verwiesen werden

Datenintegrität

- **Kandidatenschlüssel:** *unique*
- **Primärschlüssel:** *primary key*
- **Attribut darf nicht NULL sein:** *NOT NULL*
- **Referenz:** *references*

```
CREATE TABLE Studenten(
matrNr INTEGER PRIMARY KEY, (...)
);
CREATE TABLE Studentenausweis(
besitzer INTEGER REFERENCES Studenten,
(...))
```

Datenintegrität

- Was, wenn Referenzen gelöscht/geändert werden?
- **Änderung übernehmen:** *on update/delete cascade*
- **Referenz NULL setzen**: *on update/delete set null*

```
CREATE TABLE Studentenausweis(
besitzer INTEGER REFERENCES Studenten
                 ON DELETE SET NULL,
(...))
```

Datenintegrität

- Es können kompliziertere Konsistenzbedingungen gefordert werden
- **Bedingung:** *check(...)*
- Wird vor Änderung am Datenbestand geprüft

```sql
CREATE TABLE Studentenausweis(
besitzer INTEGER REFERENCES Studenten
         ON DELETE SET NULL,
CHECK(besitzer != 0)
(...))
```

Funktionale Abhängigkeiten

- Betrachte Schema $\mathscr{R}$ bestehend aus Relationen $\mathscr{R}_1, \mathscr{R}_2, \ldots, \mathscr{R}_n$ mit Ausprägung R
- Betrachte **funktionale Abhängigkeit** $\alpha \rightarrow \beta$
- Das heißt: $r, t \in R : r.\alpha = t.\alpha \Rightarrow r.\beta = r.\beta$
- **Frage:** Was bedeutet das in Worten?
- Zu einer Menge funktionaler Abhängigkeiten F kann die Hülle F^+ bestimmt werden

Schlüssel

- Wir erinnern uns: Schlüssel identifizieren Tupel eindeutig
- In der Relation $\mathscr{R}$ ist $\alpha \subseteq \mathscr{R}$ ein **Superschlüssel**, falls: $\alpha \rightarrow \mathscr{R}$
- Volle funktionale Abhängigkeit: α kann nicht weiter verkleinert werden
- Dann heißt α **Kandidatenschlüssel**

Warum machen wir das alles?!

- Wir wollen quantifizieren, ob Schemata gut oder schlecht sind
- Dafür braucht es etwas Theorie
- **Ziel:** Schöne Schemata entwerfen können
- Ab jetzt: Zerlege Relationenschema $\mathscr{R}$ in Schemata $\mathscr{R}_1, \mathscr{R}_2, \ldots, \mathscr{R}_n$
- Invariante: Abhängigkeitserhaltung, Verlustlosigkeit
- Abhängigkeitserhaltung: $F_{\mathscr{R}}^{+} = (F_{\mathscr{R}_1} \cup \ldots \cup F_{\mathscr{R}_n})^{+}$

Verlustlosigkeit

- Eine Zerlegung von $\mathscr{R}$ in $\mathscr{R}_1, \mathscr{R}_2$ heißt verlustlos, wenn mindestens eine der folgenden funktionalen Abhängigkeiten herleitbar ist: $\mathscr{R}_1 \cap \mathscr{R}_2 \rightarrow \mathscr{R}_1 \in F_R^+$ oder $\mathscr{R}_1 \cap \mathscr{R}_2 \rightarrow \mathscr{R}_2 \in F_R^+$
- **Beispiel:** Pizaesser

Pizzaesser		
Restaurant	Gast	Pizza
Bella Italia	Ben	Funghi
Pizza Huber	Jonas	Salami
Bella Italia	Jonas	Tonno

- **Frage:** Kann man die Relation verlustlos in {[Restaurant, Gast]} und {[Gast, Pizza]} zerlegen?

Attributhülle

- Bestimme maximales $\beta \subseteq \mathscr{R}$, sodass $\alpha \rightarrow \beta$ gilt

Algorithmus 1: Attributhülle

```
Data: Funktionale Abhägigkeiten F, α ⊆ ℛ
Result: β ⊆ ℛ maximal, sodass α → β
abhängig = {α} ;
repeat
    abhängig_alt = abhängig ;
    for β → γ ∈ F do
        if β ⊆ abhängig then
            abhängig = abhängig ∪ γ ;
        end
    end
until abhängig_alt == abhängig;
return abhängig
```

Kanonische Überdeckung

- F_c heißt kanonische Überdeckung von F, wenn gilt:

* $F_c^+ = F^+$
* $(\alpha \to \beta) \in F : \forall A \in \alpha : (F_c \setminus \{\alpha \to \beta\} \cup \{(\alpha \setminus \{A\}) \to \beta\})^+ \neq F_c^+$
* $(\alpha \to \beta) \in F : \forall B \in \beta : (F_c \setminus \{\alpha \to \beta\} \cup \{\alpha \to (\beta \setminus \{B\})\})^+ \neq F_c^+$
* Jede linke Seite einer FD in F_c ist einzigartig

- **Frage:** was bedeuten Bedingung zwei & drei in Worten?

Kanonische Überdeckung

- Linksreduktion macht linke Seiten der FDs so klein wie möglich

Algorithmus 2: Linksreduktion

Data: Funktionale Abhägigkeiten F

Result: Linksreduktion F' von F

$F' = F$;

for $\alpha \rightarrow \beta \in F$ **do**

  for $A \in \alpha$ **do**

    if $\beta \subseteq \mathit{Attribut\_H\ddot{u}lle}(F', \alpha \setminus \{A\})$ **then**

      $F' = F' \setminus \{\alpha \rightarrow \beta\} \cup \{(\alpha \setminus \{A\}) \rightarrow \beta\}$;

    end

  end

end

Kanonische Überdeckung

- Rechtsreduktion macht rechte Seiten der FDs so klein wie möglich

Algorithmus 3: Rechtsreduktion

Data: Funktionale Abhägigkeiten F

Result: Rechtsreduktion F' von F

$F' = F$;

for $\alpha \rightarrow \beta \in F$ **do**

  for $B \in \beta$ **do**

    if $B \in$ *Attribut_Hülle(*$F' \setminus \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta \setminus \{B\})\}, \alpha)$ **then**

      $F' = F' \setminus \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta \setminus \{B\})\}$;

    end

  end

end

Kanonische Überdeckung

- Nun können wir kanonische Überdeckung bestimmen

Algorithmus 4: Kanonische Überdeckung bestimmen

Data: Funktionale Abhägigkeiten F

Result: Kanonische Überdeckung F_c von F

$F_c = F$;

$F_c =$ Linksreduktion(F_c) ;

$F_c =$ Rechtsreduktion(F_c) ;

for $\alpha \to \emptyset \in F_c$ **do**

  $F_c = F_c \setminus \{\alpha \to \emptyset\}$;

end

$F_c =$ Gleiche_Linke_Seiten_Zusammenfassen(F_c);

Normalformen

- Quantifizieren Qualität der Relation
- **Erste Normalform:** bei uns immer eingehalten: Attribute müssen atomare Werte haben
- **Zweite Normalform:** Eine Relation $\mathscr{R}$ mit FDs F ist in 2NF, falls jedes Nichtschlüssel-Attribut $A \in \mathscr{R}$ von jedem Kandidatenschlüssel in $\mathscr{R}$ voll funktional abhängig ist
- **Dritte Normalform:** Nichtschlüssel-Attribute dürfen nur Fakten von Schlüsseln darstellen
- **Boyce-Codd Normalform:** Informationseinheiten werden nicht mehrmals gespeichert

Dritte Normalform

- Relationenschema $\mathcal{R}$ ist in 3NF, wenn für jede FD $\alpha \to B$ mit $\alpha \subseteq \mathcal{R}$ und $B \in \mathcal{R}$ gilt:
  - $B \in \alpha$, d.h. FD trivial, oder
  - α ist Superschlüssel von $\mathcal{R}$, oder
  - B in Kandidatenschlüssel von $\mathcal{R}$ enthalten
- **Synthesealgorithmus** berechnet verlustlose, abhängigkeitsbewahrende Zerlegung von $\mathcal{R}$ in 3NF
- **Kanonische Überdeckung:** möglichst redundanzfreie Darstellung der FDs einer Relation

Dritte Normalform

Algorithmus 5: Synthesealgorithmus

Data: Relationenschema $\mathscr{R}$, FDs F

Result: Zerlegung $\mathscr{R}_1, ..., \mathscr{R}_n$ in 3NF

F_c = kanonische_überdeckung(F) ;

for $(\alpha \to \beta) \in F_c$ **do**

  $\mathscr{R}_\alpha = \alpha \cup \beta$;

  $F_a = \{\alpha\prime \to \beta\prime | \alpha\prime \cup \beta\prime \in \mathscr{R}_\alpha\}$;

end

if *Kein* $\mathscr{R}_\alpha$ *enthält Kandidatenschlüssel* **then**

  κ = kandidatenschlüssel($\mathscr{R}$) ;

  $\mathscr{R}_\kappa = \kappa$;

  $F_\kappa = \emptyset$;

end

Teilschemata eliminieren;

Boyce-Codd Normalform

- Relationenschema $\mathscr{R}$ ist in BCNF, wenn für jede FD $\alpha \to \beta$ mit $\alpha, \beta \subseteq \mathscr{R}$ gilt:
  - $\beta \subseteq \alpha$, d.h. FD trivial, oder
  - α ist Superschlüssel von $\mathscr{R}$
- **Dekompositionsalgorithmus** berechnet verlustlose Zerlegung von $\mathscr{R}$ in BCNF
- **Achtung:** es kann nicht garantiert werden, dass die Zerlegung abhängigkeitsbewahrend ist

Boyce-Codd Normalform

Algorithmus 6: Dekompositionsalgorithmus BCNF

Data: Relationenschema $\mathscr{R}$, FDs F

Result: Zerlegung $Z = \{\mathscr{R}_1, ..., \mathscr{R}_n\}$ in BCNF

$Z = \{\mathscr{R}\}$;

while $\exists \mathscr{R}_i \in Z : \mathscr{R}_i$ *nicht in BCNF* **do**

    for $(\alpha \rightarrow \beta) \in F_{\mathscr{R}_i}$ *nicht trivial* **do**

        if $(\alpha \cap \beta = \emptyset) \wedge !(\alpha \rightarrow \mathscr{R}_i)$ **then**

            break ;

        end

    end

    $\mathscr{R}_{i_1} = \alpha \cup \beta$;

    $\mathscr{R}_{i_2} = \mathscr{R}_i \setminus \beta$;

    $Z = Z \setminus \mathscr{R}_i$;

    $Z = Z \cup \mathscr{R}_{i_1} \cup \mathscr{R}_{i_1}$;

end

Mehrwertige Abhängigkeiten

- MVDs verallgemeinern funktionale Abhängigkeiten
- Für $\alpha, \beta \subseteq \mathscr{R}$, schreiben wir: $\alpha \rightarrow\rightarrow \beta$
- Das heißt: für Tupel mit gleichem α kann man β vertauschen und die Tupel bleiben in R
- **Frage:** warum ist das nicht immer erfüllt?

Fähigkeiten		
Name	Sprache	ProgSprache
Benjamin	Deutsch	Java
Benjamin	Englisch	C++
Benjamin	Deutsch	C++
Benjamin	Englisch	Java
Kanye	Englisch	LOLCAT

Mehrwertige Abhängigkeiten

- Eine Zerlegung von $\mathscr{R}$ in $\mathscr{R}_1, \mathscr{R}_2$ ist verlustlos, genau dann wenn mindestens eine der folgenden MVDs herleitbar ist:
  $\mathscr{R}_1 \cap \mathscr{R}_2 \rightarrow\rightarrow \mathscr{R}_1$ oder $\mathscr{R}_1 \cap \mathscr{R}_2 \rightarrow\rightarrow \mathscr{R}_2$
- Gibt Regeln, mit denen man aus einer Menge D von MVDs die Hülle D^+ berechnen kann
- MVD $\alpha \rightarrow\rightarrow \beta$ heißt trivial, wenn $\beta \subseteq \alpha$ oder $\beta = \mathscr{R} \setminus \alpha$
- **Frage:** welche MVDs gelten in der Relation zuvor?

Vierte Normalform

- Hier wird zusätzlich zur BCNF noch die Redundanz durch MVDs ausgeschlossen
- Schema $\mathcal{R}$ ist in 4NF, wenn für jede MVD $\alpha \rightarrow\rightarrow \beta \in D^+$ gilt:
  - $\alpha \rightarrow\rightarrow \beta$ ist trivial, oder
  - α ist Superschlüssel von $\mathcal{R}$
- **Dekompositionsalgorithmus** kann wieder verwendet werden, um verlustlose Zerlegung in 4NF zu berechnen
- **Frage:** warum ist eine Relation in 4NF immer auch in BCNF?

Vierte Normalform

Algorithmus 7: Dekompositionsalgorithmus 4NF

Data: Relationenschema $\mathscr{R}$, MVDs D

Result: Zerlegung $Z = \{\mathscr{R}_1, ..., \mathscr{R}_n\}$ in 4NF

$Z = \{\mathscr{R}\}$;

while $\exists \mathscr{R}_i \in Z : \mathscr{R}_i$ *nicht in 4NF* **do**

  for $(\alpha \twoheadrightarrow \beta) \in D_{\mathscr{R}_i}$ *nicht trivial* **do**

    if $(\alpha \cap \beta = \emptyset) \wedge !(\alpha \twoheadrightarrow \mathscr{R}_i)$ **then**

      break ;

    end

  end

  $\mathscr{R}_{i_1} = \alpha \cup \beta$;

  $\mathscr{R}_{i_2} = \mathscr{R}_i \setminus \beta$;

  $Z = Z \setminus \mathscr{R}_i$;

  $Z = Z \cup \mathscr{R}_{i_1} \cup \mathscr{R}_{i_1}$;

end

Speicherhierachie

- Moderne Rechner haben verschiedene Arten des Speichers
- Dieser ist hierarchisch angeordnet: größerer Speicher ist langsamer
- Bei Festplatten kann zusätzlicher Aufwand entstehen (z.B. disk seek)

Speicher	Größe	Latenz	Vergleich
Register	bytes	1ns	Schreibtisch
Cache	K-M bytes	<10ns	Zimmer
Hauptspeicher	G bytes	<100ns	Nachbarschaft
Externer Speicher	T bytes	1ms	Tokyo

RAID

- Ziel: **Hoher Durchsatz**, **Fehlertoleranz** in Festplattenverbund
- **RAID0:** Block-Striping
- **RAID1:** Block-Mirroring
- **RAID3:** Bit-Striping & Paritätsplatte
- **RAID4:** Block-Striping & Paritätsplatte
- **RAID5:** Block-Striping, verteilte Paritätsblöcke
- RAID5 bietet gutes Verhältnis zwischen Overhead & Leistung

Buffer Manager

- Historisch waren Datenbestände größer als der Hauptspeicher
- Es mussten zusätzlich Daten auf Festplatten gespeichert werden
- Hierfür werden Tupel in ”Slotted Pages” angeordnet
- Buffer Manager verwaltet, welche Slotted Pages im Hauptspeicher sind

Datenstrukturen

- Datenstrukturen sollen diesen Aufbau berücksichtigen
- Wenige Speicherzugriffe, große und zusammenhängende Speicherbereiche
- Trotzdem noch schnelle Zugriffszeiten

$\Rightarrow$ Klassische Binärbäume reichen nicht aus

- **Frage:** Warum nicht? Was verschafft Abhilfe?

B-Bäume

- Knoten speichern (Tupel, Pointer) Paare
- Ein Knoten kann viele hundert Einträge haben
- Binäre Suche im Knoten
- Knoten können ebenfalls auf Festplatte ausgelagert werden
- B-Baum garantiert Auslastung von $\geq 50\%$
- B^+-Baum speichert Daten nur in Blättern
- **Achtung:** Löschen und Einfügen kann Knotenstruktur ändern

B^+-Bäume

- Datenbanken benutzen anstatt B-Bäumen in der Regel B^+-Bäume.
  Warum?

B^+-Bäume

- Datenbanken benutzen anstatt B-Bäumen in der Regel B^+-Bäume.
  Warum?
- Mehr innere Knoten pro Blatt, einfache Bereichsabfragen
- In der Regel haben Blätter *next*-Zeiger zum (rechten) Nachbarblatt
- Oft werden nur *inserts*, aber keine *deletes* unterstützt
- Mehrbenutzersynchronisation von B/B^+-Bäumen schwierig

B^+-Bäume - inserts

1. Suche Blatt, in welches Schlüssel eingefügt werden soll
2. Falls noch Kapazität:
   – Füge Schlüssel ein
3. Falls keine Kapazität mehr:
   – Teile vollen Knoten in zwei neue Knoten auf
   – Füge den Schlüssel in einen der neuen Knoten ein
   – Füge mittleren Eintrag in Vaterknoten → Schritt 2 *
   – Falls kein Vaterknoten: lege neue Wurzel an

- ***Achtung**, bei inneren Knoten müssen inserts wie in klassischen B-Bäumen ablaufen
- **Achtung:** splits können sich bis zur Wurzel fortpflanzen
- **Frage:** wie verhalten sich *insert* und *lookup* asymptotisch?

Hashing

- Hashtabellen eignen sich für Punktanfragen
- **Frage:** wie verhalten sich *insert* und *lookup* asymptotisch?

Hashing

- Hashtabellen eignen sich für Punktanfragen
- **Frage:** wie verhalten sich *insert* und *lookup* asymptotisch?
- Es wäre interessant, Hashtabellen als Index zu nutzen
- **Aber:** Wie kann ich eine Hashtabelle auf Disk auslagern? Was passiert, wenn die Hashtablle voll ist?

Hashing

- Hashtabellen eignen sich für Punktanfragen
- **Frage:** wie verhalten sich *insert* und *lookup* asymptotisch?
- Es wäre interessant, Hashtabellen als Index zu nutzen
- **Aber:** Wie kann ich eine Hashtabelle auf Disk auslagern? Was passiert, wenn die Hashtablle voll ist?
- I.d.R. nicht sinnvoll, alle Werte neu in gößere Tabelle einzufügen
- Erweiterbares Hashing (*engl. extendible hashing*) löst diese Probleme
- Hashing auch in Anfragebearbeitung interessant (Hash-Joins)

Erweiterbares Hashing

- Größe der Hashtabelle ist immer Zweierpotenz
- Daten werden in Buckets gespeichert
- Mehrere Indizes der Hashtabelle können auf gleiche Buckets zeigen
- Inserts können in drei Fälle unterteilt werden:
  - Bucket hat noch Platz $\Rightarrow$ kein Problem
  - Bucket is voll, abere mehrere Indizes zeigen auf gleichen Bucket $\Rightarrow$ splitte Buckets
  - Es zeigt nur ein Index auf den vollen Bucket $\Rightarrow$ verdopple Größe der Hashtabelle
- Wachstum ist nun weniger invasiv, es muss nicht jeder Bucket neu angepasst werden
- **Aber:** Verdoppelung der Tabellengröße trotzdem sehr invasiv
- **Ausblick:** Linear Hashing, Multi Level Extendible Hashing

Mehrdimensionale Indexstrukturen

- Was passiert, wenn wir unsere Daten mehrdimensional sind?
- Z.b. für Events mit Ortsangaben
- Wir brauchen Indexstrukturen, die Anfragen auf solchen Daten effizient möglich machen
- Hier gibt es viele Möglichkeiten: Grid Files, R-Bäume, R^+-Bäume

R-Baum

- **Idee:** Teile Raum iterativ in disjunkte Rechtecke auf
- Wie beim B-Baum kann es mehrere Ebenen geben
- Splitten von Knoten ist oft aufwendig und nicht immer eindeutig