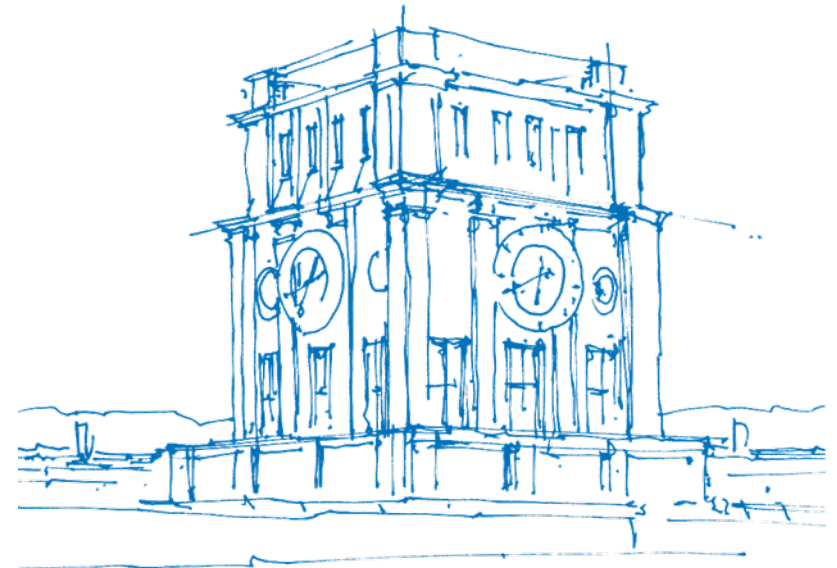


# Grundlagen Datenbanken

Benjamin Wagner

31. Januar 2019



*TUM Uhrenturm*

# Allgemeines

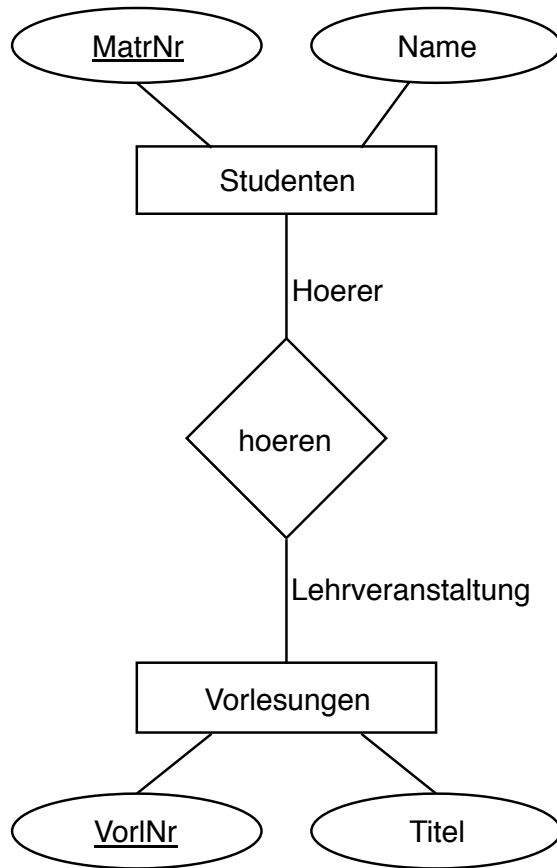
- Folien von mir sollen unterstützend dienen. Sie sind nicht von der Übungsleitung abgesegnet und haben keinen Anspruch auf Vollständigkeit (oder Richtigkeit).
- Bei Fragen oder Korrekturvorschlägen: [wagnerbe@in.tum.de](mailto:wagnerbe@in.tum.de)
- Vorlesungsbegleitendes Buch von Professor Kemper (Chemiebib)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

⇒ Lässt sich als Graph darstellen, siehe Universitätsschema

# Beispiel: Schema

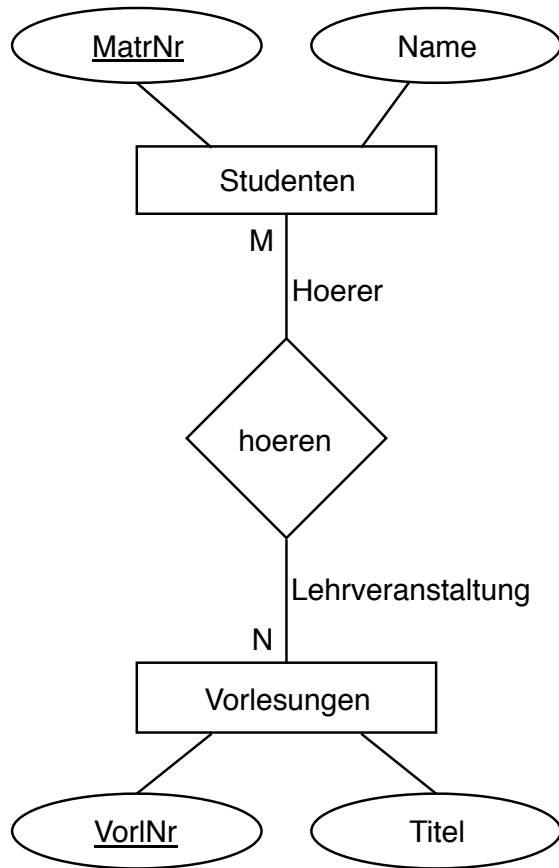


- Repräsentiert Studenten, die bestimmte Vorlesungen hören
- Schlüssel sind unterstrichen, ein Student ist eindeutig durch seine MatrNr bestimmt
- Hören modelliert eine Relationship zwischen Studenten und Vorlesungen
- Studenten treten hier in Rolle "Hörer" auf

# 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

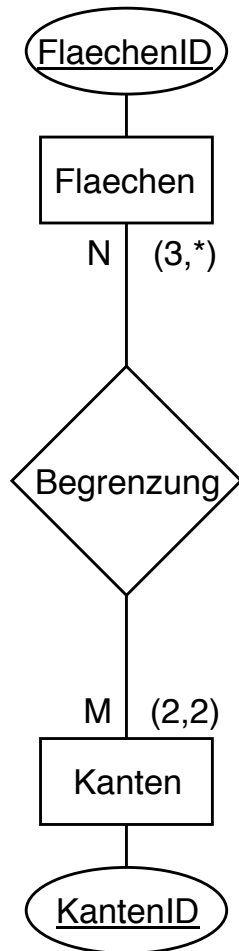


- Nun mit Funktionalitätsangaben
- Ein Student kann N Vorlesungen hören
- Eine Vorlesung kann von M Studenten gehört werden

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

# 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

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

# Das relationale Modell

- Es gibt Domänen  $D_1, D_2, \dots, D_n$ , das entspricht Wertebereichen  
z.B. Integer, Strings, Chars, Booleans
- Für eine Relation  $R$  gilt:  $R \subset D_1 \times D_2 \times \dots \times D_n$
- 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
$\sigma_{\text{Kondition}}$	Selektion
$\Pi_{\text{Attribute}}$	Projektion
$\times$	Kreuzprodukt
$\rho_{\text{neu} \leftarrow \text{alt}}$	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

÷

a3
1
2
3

=

a1	a2
3	5
5	5

# Kalküle

- **Tupelkalkül:** Schreibweise (hoffentlich) aus Mathe-Vorlesungen bekannt:  $\{t \mid P(t)\}$ , mit  $P(t)$  aussagenlogischer Formel
- **Domänenkalkül:** Domänenvariablen:  $\{[v_1, \dots, v_n] \mid P(v_1, \dots, v_n)\}$
- **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:

1 **SELECT** . . .

2 **FROM** . . .

3 **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:

```
1 CREATE TABLE Customers (  
2     CustId      integer not null ,  
3     Name        varchar(30) not null ,  
4     Birthday    date  
5 );
```

# SQL - Einfache Anfrage

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

# SQL - Einfache Anfrage

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

```
1 SELECT Name
2 FROM Professoren
3 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

```
1 SELECT Name
2 FROM Professoren
3 WHERE Rang = 'C4'
```

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

```
1 SELECT *
2 FROM Studenten
3 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

```
1 SELECT Professoren.Name
2 FROM Professoren, Vorlesungen
3 WHERE Professoren.persNr =
4         Vorlesungen.gelesenVon
5         AND Vorlesungen.Titel = 'Ethik'
```



# SQL - Sprachkonstrukte

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

# SQL - Sprachkonstrukte

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

```
1 SELECT DISTINCT studenten.semester
2 FROM studenten, hoeren, voerlesungen
3 WHERE studenten.matrnr = hoeren.matrnr
4         AND hoeren.vorlNr = vorlesungen.vorlNr
5         AND vorlesungen.titel = 'Logik'
```

# SQL - Sprachkonstrukte

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

```
1 (SELECT p.Name
2 FROM Professoren p
3 WHERE NOT EXISTS (
4     SELECT *
5     FROM Vorlesungen v
6     WHERE v.gelesenVon = p.persNr);)
7 INTERSECT
8 (...)
```

# 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?

```
1 SELECT semester , count (*)  
2 FROM Studenten  
3 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?

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

# SQL - Sprachkonstrukte

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

```
1 WITH h AS (SELECT VorlNr ,  
2             count(*) AS AnzProVorl  
3             FROM hoeren  
4             GROUP BY VorlNr),  
5  
6 ( . . . )
```



# SQL - Sprachkonstrukte

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

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

# SQL - Sprachkonstrukte

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

```
1 SELECT MatrNr, (CASE
2     WHEN Note < 1.5 THEN 'sehr_gut'
3     WHEN Note < 2.5 THEN 'gut'
4     WHEN Note < 3.5 THEN 'befriedigend'
5     WHEN Note < 4.0 THEN 'ausreichend'
6     ELSE 'nicht_bestanden'
7     END)
8 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

```
1 WITH RECURSIVE TransVorl(Vorg, Nachf) AS
2   (SELECT Vorgaenger, Nachfolger
3    FROM voraussetzen
4   UNION ALL
5    SELECT t.Vorg, v.Nachfolger
6    FROM TransVorl t, Voraussetzen v
7   WHERE t.Nachf = v.Vorgaenger)
```

# SQL - Rekursion

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

```
1 SELECT Titel FROM Vorlesungen
2 WHERE VorlNr IN
3   (SELECT Vorg
4     FROM TransVorl where Nachf IN
5     (SELECT VorlNr FROM Vorlesungen
6       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*

```
1 CREATE TABLE Studenten(  
2   matrNr INTEGER PRIMARY KEY, (...)  
3 );  
4 CREATE TABLE Studentenausweis(  
5   besitzer INTEGER REFERENCES Studenten,  
6   (...))
```

# Datenintegrität

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

```
1 CREATE TABLE Studentenausweis(  
2   besitzer INTEGER REFERENCES Studenten  
3           ON DELETE SET NULL,  
4   (...))
```



# Datenintegrität

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

```
1 CREATE TABLE Studentenausweis (  
2   besitzer INTEGER REFERENCES Studenten  
3       ON DELETE SET NULL ,  
4   CHECK (besitzer != 0)  
5   (...))
```

# Funktionale Abhängigkeiten

- Betrachte Schema  $\mathcal{R}$  bestehend aus Relationen  $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$  mit Ausprägung  $R$
- Betrachte **funktionale Abhängigkeit**  $\alpha \rightarrow \beta$
- Das heißt:  $r, t \in R : r.\alpha = t.\alpha \Rightarrow r.\beta = t.\beta$
- **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  $\mathcal{R}$  ist  $\alpha \subseteq \mathcal{R}$  ein **Superschlüssel**, falls:  $\alpha \rightarrow \mathcal{R}$
- Volle funktionale Abhängigkeit:  $\alpha$  kann nicht weiter verkleinert werden
- Dann heißt  $\alpha$  **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  $\mathcal{R}$  in Schemata  $\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n$
- Invariante: Abhängigkeitserhaltung, Verlustlosigkeit
- Abhängigkeitserhaltung:  $F_{\mathcal{R}}^+ = (F_{\mathcal{R}_1} \cup \dots \cup F_{\mathcal{R}_n})^+$

# Verlustlosigkeit

- Eine Zerlegung von  $\mathcal{R}$  in  $\mathcal{R}_1, \mathcal{R}_2$  heißt verlustlos, wenn mindestens eine der folgenden funktionalen Abhängigkeiten herleitbar ist:

$$\mathcal{R}_1 \cap \mathcal{R}_2 \rightarrow \mathcal{R}_1 \in F_R^+ \text{ oder } \mathcal{R}_1 \cap \mathcal{R}_2 \rightarrow \mathcal{R}_2 \in F_R^+$$

- **Beispiel:** Pizaesser

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

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

# Attributhülle

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

---

## Algorithmus 1: Attributhülle

---

**Data:** Funktionale Abhängigkeiten  $F$ ,  $\alpha \subseteq \mathcal{R}$

**Result:**  $\beta \subseteq \mathcal{R}$  maximal, sodass  $\alpha \rightarrow \beta$

$abh\ddot{a}ngig = \{\alpha\}$  ;

**repeat**

$abh\ddot{a}ngig\_alt = abh\ddot{a}ngig$  ;

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

**if**  $\beta \subseteq abh\ddot{a}ngig$  **then**

$abh\ddot{a}ngig = abh\ddot{a}ngig \cup \gamma$  ;

**end**

**end**

**until**  $abh\ddot{a}ngig\_alt == abh\ddot{a}ngig$ ;

**return**  $abh\ddot{a}ngig$

---

# Kanonische Überdeckung

- $F_c$  heißt kanonische Überdeckung von  $F$ , wenn gilt:
  - \*  $F_c^+ = F^+$
  - \*  $(\alpha \rightarrow \beta) \in F : \forall A \in \alpha : (F_c \setminus \{\alpha \rightarrow \beta\} \cup \{(\alpha \setminus \{A\}) \rightarrow \beta\})^+ \neq F_c^+$
  - \*  $(\alpha \rightarrow \beta) \in F : \forall B \in \beta : (F_c \setminus \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta \setminus \{B\})\})^+ \neq F_c^+$
  - \* 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 Abhängigkeiten  $F$

**Result:** Linksreduktion  $F'$  von  $F$

$F' = F$  ;

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

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

**if**  $\beta \subseteq \text{Attribut\_Hülle}(F', \alpha \setminus \{A\})$  **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 Abhängigkeiten  $F$

**Result:** Rechtsreduktion  $F'$  von  $F$

$F' = F$  ;

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

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

**if**  $B \in \text{Attribut\_Hülle}(F' \setminus \{\alpha \rightarrow \beta\} \cup \{\alpha \rightarrow (\beta \setminus \{B\})\}, \alpha)$  **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 Abhängigkeiten  $F$

**Result:** Kanonische Überdeckung  $F_c$  von  $F$

$F_c = F$  ;

$F_c = \text{Linksreduktion}(F_c)$  ;

$F_c = \text{Rechtsreduktion}(F_c)$  ;

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

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

**end**

$F_c = \text{Gleiche\_Linke\_Seiten\_Zusammenfassen}(F_c)$ ;

---

# Normalformen

- Quantifizieren Qualität der Relation
- **Erste Normalform:** bei uns immer eingehalten: Attribute müssen atomare Werte haben
- **Zweite Normalform:** Eine Relation  $\mathcal{R}$  mit FDs  $F$  ist in 2NF, falls jedes Nichtschlüssel-Attribut  $A \in \mathcal{R}$  von jedem Kandidatenschlüssel in  $\mathcal{R}$  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  $\alpha \rightarrow B$  mit  $\alpha \subseteq \mathcal{R}$  und  $B \in \mathcal{R}$  gilt:
  - \*  $B \in \alpha$ , d.h. FD trivial, oder
  - \*  $\alpha$  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: Syntheseargorithmus

---

**Data:** Relationenschema  $\mathcal{R}$ , FDs  $F$

**Result:** Zerlegung  $\mathcal{R}_1, \dots, \mathcal{R}_n$  in 3NF

$F_c = \text{kanonische\_überdeckung}(F)$  ;

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

$\mathcal{R}_\alpha = \alpha \cup \beta$  ;  
     $F_\alpha = \{ \alpha' \rightarrow \beta' \mid \alpha' \cup \beta' \in \mathcal{R}_\alpha \}$  ;

**end**

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

$\kappa = \text{kandidatenschlüssel}(\mathcal{R})$  ;  
     $\mathcal{R}_\kappa = \kappa$  ;  
     $F_\kappa = \emptyset$  ;

**end**

Teilschemata eliminieren;

---

# Boyce-Codd Normalform

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

# Boyce-Codd Normalform

---

## Algorithmus 6: Dekompositionsalgorithmus BCNF

---

**Data:** Relationenschema  $\mathcal{R}$ , FDs  $F$

**Result:** Zerlegung  $Z = \{\mathcal{R}_1, \dots, \mathcal{R}_n\}$  in BCNF

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

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

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

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

            break ;

**end**

**end**

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

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

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

$Z = Z \cup \mathcal{R}_{i_1} \cup \mathcal{R}_{i_2} ;$

**end**

---

# Mehrwertige Abhängigkeiten

- MVDs verallgemeinern funktionale Abhängigkeiten
- Für  $\alpha, \beta \subseteq \mathcal{R}$ , schreiben wir:  $\alpha \twoheadrightarrow \beta$
- Das heißt: für Tupel mit gleichem  $\alpha$  kann man  $\beta$  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  $\mathcal{R}$  in  $\mathcal{R}_1, \mathcal{R}_2$  ist verlustlos, genau dann wenn mindestens eine der folgenden MVDs herleitbar ist:  
 $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$  oder  $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$
- Gibt Regeln, mit denen man aus einer Menge  $D$  von MVDs die Hülle  $D^+$  berechnen kann
- MVD  $\alpha \twoheadrightarrow \beta$  heißt trivial, wenn  $\beta \subseteq \alpha$  oder  $\beta = \mathcal{R} \setminus \alpha$
- **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  $\alpha \twoheadrightarrow \beta \in D^+$  gilt:
  - \*  $\alpha \twoheadrightarrow \beta$  ist trivial, oder
  - \*  $\alpha$  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  $\mathcal{R}$ , MVDs  $D$

**Result:** Zerlegung  $Z = \{\mathcal{R}_1, \dots, \mathcal{R}_n\}$  in 4NF

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

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

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

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

**break** ;

**end**

**end**

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

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

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

$Z = Z \cup \mathcal{R}_{i_1} \cup \mathcal{R}_{i_2}$  ;

**end**

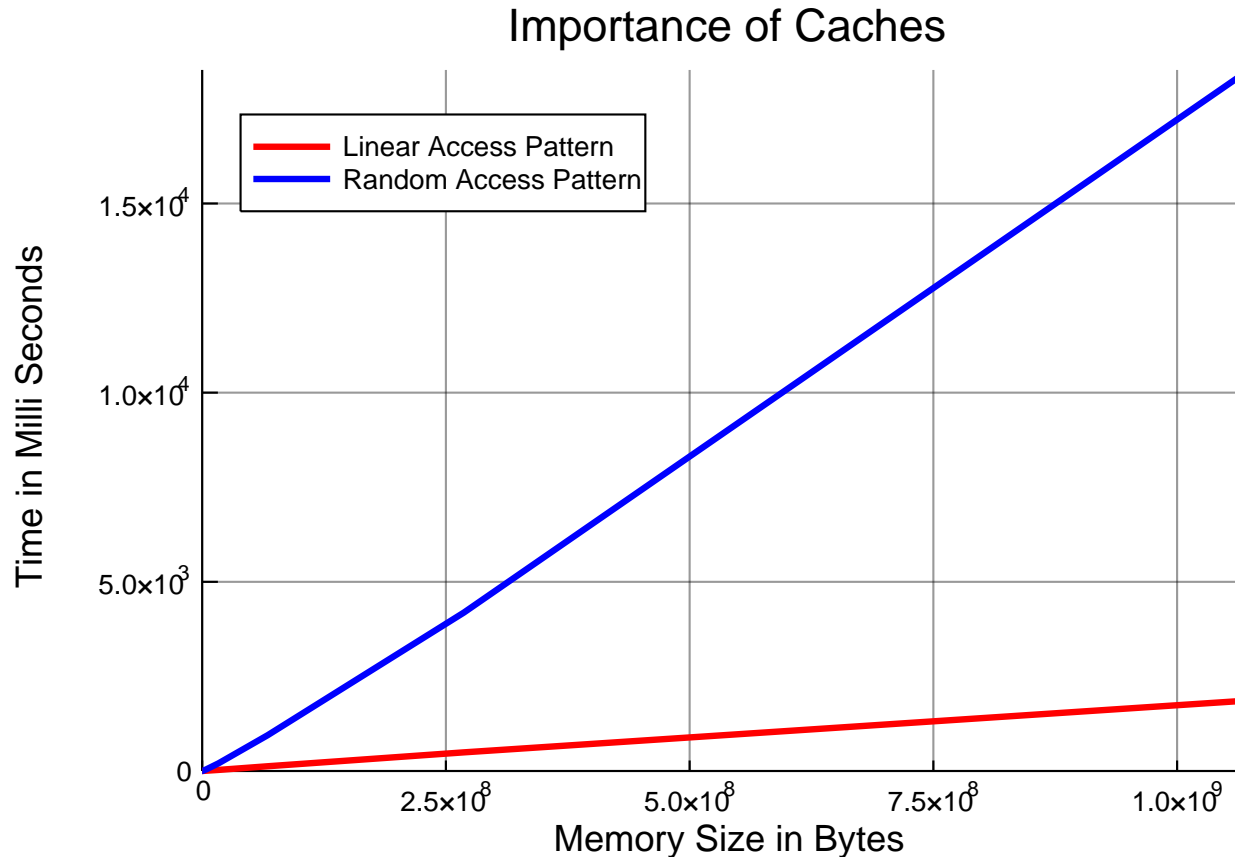
---

# Speicherhierarchie

- 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

# Caches



Zugriff auf Daten in linearer oder zufälliger Reihenfolge; Quelle: selbstgeschrieben

# 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

⇒ 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<sup>+</sup>-Baum speichert Daten nur in Blättern
- **Achtung:** Löschen und Einfügen kann Knotenstruktur ändern

# B<sup>+</sup>-Bäume

- Datenbanken benutzen anstatt B-Bäumen in der Regel B<sup>+</sup>-Bäume.

**Warum?**

# B<sup>+</sup>-Bäume

- Datenbanken benutzen anstatt B-Bäumen in der Regel B<sup>+</sup>-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<sup>+</sup>-Bäumen schwierig

# B<sup>+</sup>-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 Hashtabelle 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 Hashtabelle voll ist?
- I.d.R. nicht sinnvoll, alle Werte neu in größere 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 ist voll, aber 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

# Anfrageoptimierung

- Wenn wir eine Anfrage an die Datenbank stellen, muss diese abgearbeitet werden
  - Die Datenbank muss einen Plan erstellen, wie sie die Anfrage abarbeiten soll
  - Dies wird in relationaler Algebra ausgedrückt
  - **Aber:** verschiedene Algebrabäume können logisch äquivalent sein
- ⇒ Die Datenbank muss einen möglichst effizienten Plan finden

# Anfrageoptimierung - Erste Idee

- Beginn: übersetzen die Anfrage in einen beliebigen relationalen Plan
- Diesen Plan können wir dann mit Umformungsregeln modifizieren
- **Frage:** was sind Beispiele für solche Regeln?
- Suche Heuristiken, welche Regeln zu "billigeren" Plänen führen
- Wunsch: Zwischenergebnisse sollen klein sein

# Anfrageoptimierung - Erste Idee

- Beginn: übersetzen die Anfrage in einen beliebigen relationalen Plan
- Diesen Plan können wir dann mit Umformungsregeln modifizieren
- **Frage:** was sind Beispiele für solche Regeln?
- Suche Heuristiken, welche Regeln zu "billigeren" Plänen führen
- Wunsch: Zwischenergebnisse sollen klein sein

1. Selektionen aufbrechen
2. Selektionen nach unten schieben
3. Kreuzprodukte & Selektionen zu Joins verschmelzen
4. Joinreihenfolge bestimmen (klein: links, groß: rechts)

# Korrelierte Unteranfragen

- Warum ist folgende Anfrage bei der Ausführung potentiell langsam?

```
1 SELECT s.Name, p.VorlNr
2 FROM Studenten s, prüfen p
3 WHERE s.MatrNr = p.MatrNr AND p.Note = (
4     SELECT MIN(p2.Note)
5     FROM prüfen p2
6     WHERE s.MatrNr=p2.MatrNr )
```

- **Frage:** wie lösen wir dieses Problem?

# Ausführung der Anfrage

- Wir haben nun einen (hoffentlich) guten Query-Plan
- Dieser muss von der Datenbank ausgeführt werden
- Klassisch: jeder Operator wird implementiert
- Interface eines Operators: *open*, *close*, *next*
- Die Tupel werden dann bottom-up durch den Iteratorbaum geschleust
- **Ausblick:** das ist leider sehr ineffizient, Hyper kompiliert stattdessen den Operatorbaum in Maschinencode
- Wir müssen nun noch die einzelnen Operatoren implementieren

# Joins - Algorithmen

- Der Join ist einer der wichtigsten Operatoren

⇒ Wir brauchen eine effiziente Implementierung!

- **Frage:** welche Join-Algorithmen gibt es?



# Joins - Algorithmen

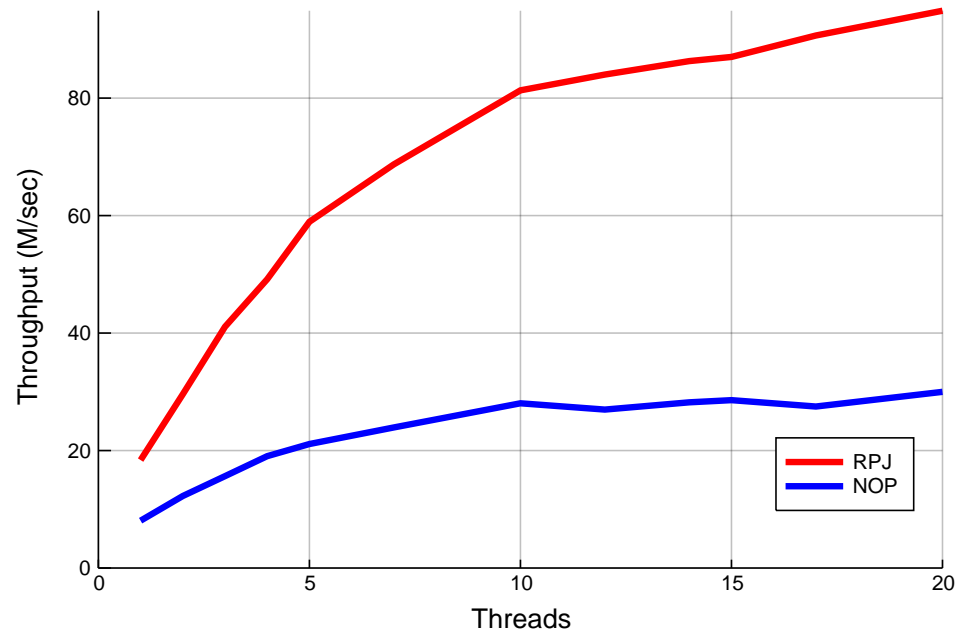
- Der Join ist einer der wichtigsten Operatoren

⇒ Wir brauchen eine effiziente Implementierung!

- **Frage:** welche Join-Algorithmen gibt es?
- Nested-Loop-Join, Hash-Join, Sort-Merge-Join
- In Hauptspeicherdatenbanken sind i.d.R. Hash-Joins am schnellsten
- **Aber:** selbst für Hash-Joins gibt es viele Implementierungen

# Hauptspeicher Hash Joins - Algorithmen

- Zwei konkurrierende Hash Join Implementierungen
- Gleiche Daten, alles im Hauptspeicher



S uniformly distributed,  $|R| = |S| \approx 16.8\text{M}$ . Quelle.

# Optimierung Revisited

- Bisher: Heuristiken zur Optimierung von Anfragebäumen
- Haben gesehen: Join-Reihenfolge kann zu sehr unterschiedlichen Laufzeiten führen
- Optimierte Join-Algorithmen können schlechte Pläne nicht verstecken
- Heuristiken können auch zu sehr schlechten Plänen führen
- **Ziel:** suche Algorithmen, welche bessere Pläne generieren

# Selektivität

- Müssen quantifizieren, ob Operatoren große oder kleine Zwischenergebnisse generieren
- Die Selektivität beschreibt, welcher Anteil von Tupeln die in den Operator gehen, auch im Ergebnis landen
- Selektivität von  $\sigma_p := \frac{|\sigma_p(R)|}{|R|}$
- Selektivität von  $\bowtie := \frac{|R \bowtie S|}{|R \times S|}$
- **Frage:** was sind die Selektivitäten von  $\sigma_{true}, \sigma_{false}$
- **Frage:** in R.a sind Werte gleichverteilt in  $\{1, 2, \dots, n\}$ . Was ist die erwartete Selektivität von  $\sigma_{R.a=1}(R)$

# Kostenabschätzung

- Können nun abschätzen, wie groß Zwischenergebnisse sind
- Müssen nun noch Kosten für eigentliche Ausführung schätzen
- Hier helfen Kostenmodelle für die einzelnen Operatoren
- Diese können verschiedene Parameter haben und verschieden Aufwändig sein
- **Frage:** was sind sinnvolle Parameter beim Abschätzen einer Selektion?

# Optimierung - Algorithmen

- Für einzelne Operatorbäume können wir nun gut schätzen, wie teuer die Ausführung ist
- Ein Algorithmus zum Bestimmen der Joinreihenfolge kann nun also einen guten Plan wählen
- **Problem:** der Suchraum der möglichen Anfragepläne ist in der Regel zu groß

⇒ müssen Algorithmen finden, welche Suchraum einschränken

- **Ausblick:** Vorlesung „Query Optimization“

# Dynamische Programmierung

- Klassischer Optimierungsalgorithmus
- Idee: größere optimale Lösungen lassen sich aus kleineren optimalen Lösungen generieren
- **Frage:** inwieweit ist das bei Joins sinnvoll?
- **Idee:** baue Pläne Bottom-Up aus möglichst guten Teilplänen zusammen
- Level =1: bestimme Zugriffsmethode auf Relationen
- Level >1: verbinde Ergebnisse des vorherigen Levels
- Entferne in jedem Schritt „schlechte“ Zwischenpläne (engl. pruning)

---

**Algorithmus 8:** Optimierung Dynamische Programmierung

---

**Data:** Relationen  $R_1, R_2, \dots, R_n$

**Result:** Guter Anfrageplan

tabelle =  $\emptyset$ ;

**for**  $i \in \{1, 2, \dots, n\}$  **do**

    tabelle = tabelle  $\cup$  zugriffe( $R_i$ );

    prune( $\{R_i\}$ );

**end**

**for**  $i \in \{2, 3, \dots, n\}$  **do**

**for**  $S \subseteq \{R_1, \dots, R_n\}$  mit  $|S| = i$  **do**

        tabelle[S] =  $\emptyset$ ;

**for**  $O \subset S$  **do**

            tabelle[S] = tabelle[S]  $\cup$  plan(tabelle[O], tabelle[ $S \setminus O$ ]);

**end**

        prune(S);

**end**

**end**

prune( $\{R_1, R_2, \dots, R_n\}$ );

**return** tabelle[ $\{R_1, R_2, \dots, R_n\}$ ]

---



# Transaktionen

- Transaktionen verpacken eine Folge von Operationen
- In einer Datenbank kann z.B. jede SQL-Anfrage als Transaktion modelliert werden
- Genereller Ablauf: **BOT** → Operationen → **C oder A**
- Datenbanken haben hohe Anforderungen an Datenintegrität
- Wir können Transaktionen nicht komplett frei arbeiten lassen
- Datenbank muss **ACID**-Forderungen erfüllen
- **Ausblick:** Vorlesung „Transaction Systems“