Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Mehrwertige Abhängigkeiten

Bislang:

Funktionale Abhängigkeiten der Form

- Ausprägung R
- Seien $\alpha \subseteq \mathcal{R}$ und $\beta \subseteq \mathcal{R}$
- $\alpha \rightarrow \beta$ genau dann, wenn $\forall r, s \in R$ gilt: $r.\alpha = s.\alpha \Rightarrow r.\beta = s.\beta$
- D.h. die α-Werte bestimmen die β-Werte funktional (=eindeutig)

Nun: Mehrwertige Abhängigkeiten (multivalued dependencies)

Notation: $\alpha \twoheadrightarrow \beta$
Falls einem Attributwert α eine **Menge** von β-Werten zugeordnet werden.

Genaue Defintion folgt (gleich).

Mehrwertige Abhängigkeiten: Beispiel

Fähigkeiten		
PersNr	Sprache	ProgSprache
3002	griechisch	C
3002	lateinisch	Pascal
3002	griechisch	Pascal
3002	lateinisch	C
3005	deutsch	Ada

Mehrwertige Abhängigkeiten dieser Relation:

- $\{\text{PersNr}\} \twoheadrightarrow \{\text{Sprache}\}$ und
- $\{\text{PersNr}\} \twoheadrightarrow \{\text{ProgSprache}\}$

MVDs führen zu Redundanz und Anomalien

Mehrwertige Abhängigkeiten

R		
A	B	C
a	b	c
a	bb	cc
a	b	cc
a	bb	c

- $A \twoheadrightarrow B$
- $A \twoheadrightarrow C$

Bei zwei Tupeln mit gleichen α-Werten kann man die β-Werte vertauschen und die resultierenden Tupel müssen auch in der Relation sein.

Mehrwertige Abhängigkeiten: Definition 1

	R		
	$\overbrace{A_1, ..., A_i}^{\alpha}$	$\overbrace{A_{i+1}, ..., A_j}^{\beta}$	$\overbrace{A_{j+1}, ..., A_n}^{\gamma}$
t_1	$a_1, ..., a_i$	$a_{i+1}, ..., a_j$	$a_{j+1}, ..., a_n$
t_2	$a_1, ..., a_i$	$b_{i+1}, ..., b_j$	$b_{j+1}, ..., b_n$
t_3	$a_1, ..., a_i$	$a_{i+1}, ..., a_j$	$b_{j+1}, ..., b_n$
t_4	$a_1, ..., a_i$	$b_{i+1}, ..., b_j$	$a_{j+1}, ..., a_n$

$\alpha \twoheadrightarrow \beta$ gilt genau dann, wenn für jede Ausprägung von R gilt:

- wenn es zwei Tupel t_1 und t_2 mit gleichen α-Werten gibt, dann muss es auch zwei Tupel t_3 und t_4 geben mit
  - $t_1.\alpha = t_2.\alpha = t_3.\alpha = t_4.\alpha$ (alle α-Werte gleich)
  - $t_3.\beta = t_1.\beta$,$t_4.\beta = t_2.\beta$ (β-Paare gleich)
  - $t_3.\gamma = t_2.\gamma$, $t_4.\gamma = t_1.\gamma$ (γ-Paare **vertauscht**)

Veranschaulichung: Spezialfall

Veranschaulichung für MVD $\alpha \twoheadrightarrow \beta$:

Wenn α, β, γ jeweils nur aus einem Attribut A, B, und C bestehen:

Wenn $\{b_1, ..., b_i\}$ und $\{c_1, ..., c_j\}$ die B bzw. C-Werte für einen bestimmen A-Wert a sind, dann muss die Relation auch die folgenden $(i * j)$ Tupel enhalten:

$$\{a\} \times \{b_1, ..., b_i\} \times \{c_1, ..., c_j\}$$

Mehrwertige Abhängigkeiten: Definition 2

- Eine mehrwertige Abhängigkeit (multivalued dependency, MVD) $\alpha \twoheadrightarrow \beta$ besagt, dass einem Attribut α in $\mathcal{R}$ eine **Menge** von β-Werten zugeordnet werden.
- Wenn die MVD $\alpha \twoheadrightarrow \beta$ in R erfüllt, dann kann es als Erweiterung zu FDs α, β, c geben mit

$$|\pi_\beta(\sigma_{\alpha=c}(R))| > 1$$

- Diese Zuordnung ist **unabhängig** von den restlichen Attributen in $\mathcal{R}$

Mit anderen Worten:

- α bestimmt **nicht nur** einen **einzelnen** Wert (ein singleton)
- genau das ist ja bei einer normalen FD $\alpha \rightarrow \beta$ der Fall!
- **sondern** eine Menge von Werten
- diese Wertemenge ist unabhängig von den anderen Attributen in $\gamma = \mathcal{R} - \alpha - \beta$

Natürlich: Jede FD ist auch eine MVD (aber nicht umgekehrt)

Beispiel

Fähigkeiten		
PersNr	Sprache	ProgSprache
3002	griechisch	C
3002	lateinisch	Pascal
3002	griechisch	Pascal
3002	lateinisch	C
3005	deutsch	Ada

$\pi_{PersNr,Sprache}$

Sprachen	
PersNr	Sprache
3002	griechisch
3002	lateinisch
3005	deutsch

$\pi_{PersNr,ProgSprache}$

ProgSprachen	
PersNr	ProgSprache
3002	C
3002	Pascal
3005	Ada

Verlustlose Zerlegung bei MVDs: hinreichende + notwendige Bedingung

Die Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1$ und $\mathcal{R}_2$ ist verlustlos **genau dann, wenn**

- $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$
- **und** mindestens eine von zwei MVDs gilt:
  - $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$ **oder**
  - $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$

Für unser Beispiel gilt:

- {PersNr, Sprache, ProgrSprache} = {PersNr, Sprache} ∪ {PersNr, ProgSprache}
- {PersNr} $\twoheadrightarrow$ {Sprache}
- {PersNr} $\twoheadrightarrow$ {ProgSprache}

D.h. es gelten sogar beide MVDs!

MVDs in Paaren

- es gilt zusätzlich: wenn $\alpha \twoheadrightarrow \beta$, dann gilt immer
  - $\alpha \twoheadrightarrow \gamma$
  - mit $\gamma = \mathcal{R} - \alpha - \beta$
- D.h. MVDs treten immer als Paare auf
- wir könnten MVDs deshalb auch so notieren: $\alpha \twoheadrightarrow \beta | \gamma$

Triviale MVDs ...

- .... sind solche, die von jeder Relationenausprägung R von $\mathcal{R}$ erfüllt werden.
- $\alpha \cup \beta \subseteq \mathcal{R}$
- Eine MVD $\alpha \twoheadrightarrow \beta$ ist trivial genau dann, wenn
  1. $\beta \subseteq \alpha$ oder
  2. $\beta = \mathcal{R} - \alpha$
- **Nur** die Bedingung 1 galt auch für normale FDs.
- Beispiel für Bedingung 2:
  - $\mathcal{R} = \{PersNr, Sprache\}$
  - $\alpha = \{PersNr\}$
  - $\beta = \{Sprache\}$
  - $\mathcal{R} - \alpha = \{PersNr, Sprache\} - \{PersNr\} = \{Sprache\} = \beta \Rightarrow$ MVD ist trivial!

Vierte Normalform

Eine Relation $\mathcal{R}$ ist in 4NF, wenn für jede MVD $\alpha \twoheadrightarrow \beta$ eine der folgenden Bedingungen gilt:

- Die MVD ist trivial **oder**
- α ist Superschlüssel von $\mathcal{R}$

- D.h. 4NF ist sehr ähnlich zu BCNF!
- Unterschied:
  - MVDs statt FDs
  - Definition von "trivial" wurde erweitert.
- 4NF erfüllt $\Rightarrow$ BCNF erfüllt, da jede FD eine MVD ist.

Dekomposition in 4NF

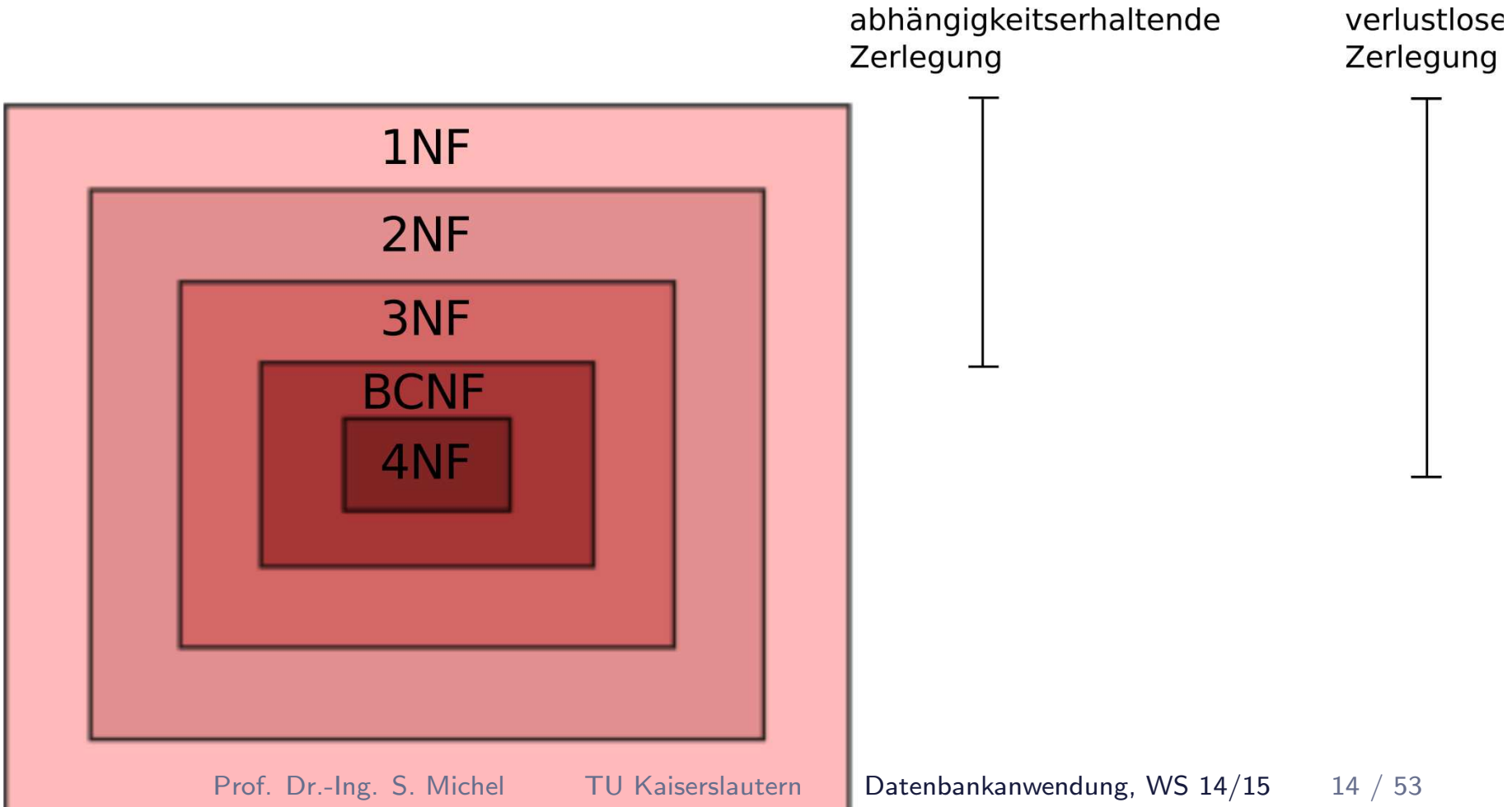
Starte mit der Menge $Z := \{\mathcal{R}\}$

Solange es noch ein Relationenschema $\mathcal{R}_i$ in Z gibt, dass nicht in 4NF ist, mache folgendes:

- Es gibt also eine für $\mathcal{R}_i$ geltende nicht-triviale MVD $(\alpha \twoheadrightarrow \beta)$, für die gilt:
  - $\alpha \cap \beta = \emptyset$
  - $\neg(\alpha \rightarrow \mathcal{R}_i)$
- Finde eine solche MVD
- Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i1} := \alpha \cup \beta$ und $\mathcal{R}_{i2} := \mathcal{R}_i - \beta$
- Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i1}$ und $\mathcal{R}_{i2}$ ein, also $Z := (Z - \mathcal{R}_i) \cup \{\mathcal{R}_{i1}\} \cup \{\mathcal{R}_{i2}\}$

Alle Normalformen auf einen Blick

- Die Verlustlosigkeit ist für alle Zerlegungsalgorithmen in alle Normalformen garantiert.
- Die Abhängigkeitserhaltung kann nur bis zur dritten Normalform garantiert werden.



Zusammenfassung Entwurfstheorie

- Ziel: Bewertung der Güte einer Relation; Vermeidung von Anomalien durch Zerlegung in "bessere" Relationen
- Betrachtung von funktionalen Abhängigkeiten zur Identifikation von Redundanz
- Korrektheit von Zerlegung definiert als verlustlos und abhängigkeitsbewahrend.
- 1NF, 2NF, 3NF, BCNF und 4NF betrachtet.
- Algorithmen zur Zerlegung/Synthese

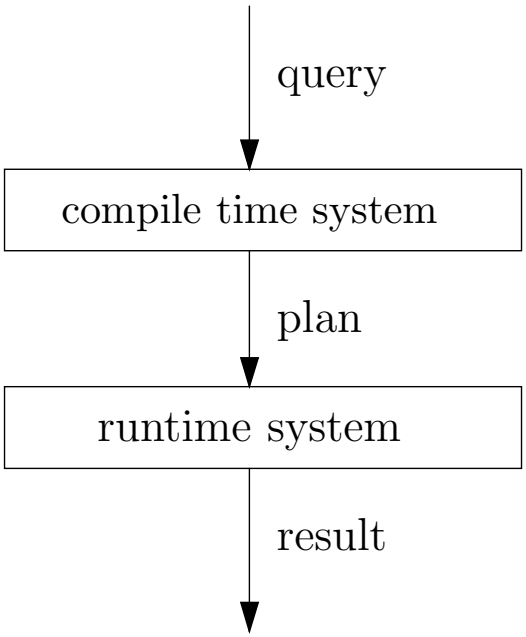
Übersicht und Ausblick

- Bislang bekannt, aus VL Informationssysteme: Optimierung anhand von Regeln auf logischer Ebene
- D.h. gegeben ein Anfrageplan, wie kann dieser optimiert werden (durch "Verschieben" von Operatoren)

Ausblick auf kommende Vorlesungen

- Physische Datenorganisation
- Zugriffskosten
- Implementierung der Operatoren
- Kostenschätzung
- Kostenbasierte Anfrageoptimierung

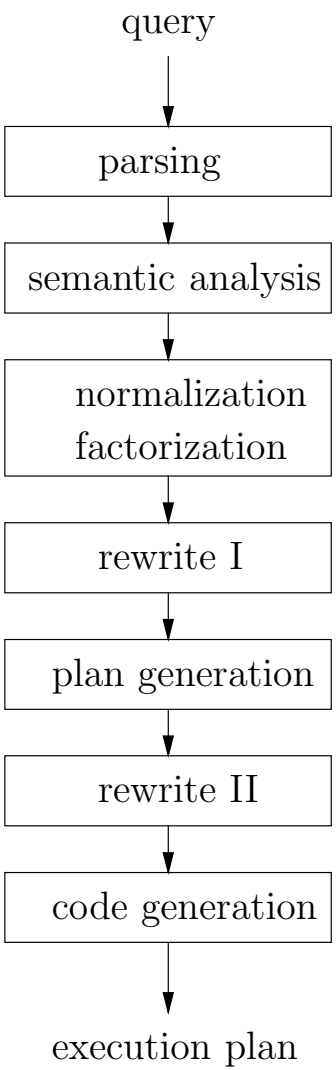
Übersicht Anfrageverarbeitung



- Eingabe: Anfrage als Text (String)
- Kompilierzeitsystem (compile time system) übersetzt und optimiert die Anfrage
- Zwischenprodukt: Anfrage als Anfrageplan (query plan)
- Laufzeitsystem (runtime system) führt die Anfrage aus
- Ausgabe: Ergebnisse der Anfrage

Übersicht, Buch "Working Draft" von Guido Moerkotte (Uni Mannheim):
`http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`
Vielen Dank an Thomas Neumann (TU München) für die Bereitstellung einiger im Folgenden benutzten Folien zu Anfrageverarbeitung/Anfrageoptimierung.

Übersicht Compile Time System



1. Parsen: Erzeugung Abstract Syntax Tree (AST)
2. Schema lookup, Typinferenz
3. Normalisierung, Faktorisierung etc.
4. Auflösen von Sichten, Entschachteln etc.
5. Erzeugung des Ausführungsplans (execution plan)
6. Weitere Optimierung des Plans
7. Erzeugung von ausführbarem Codes

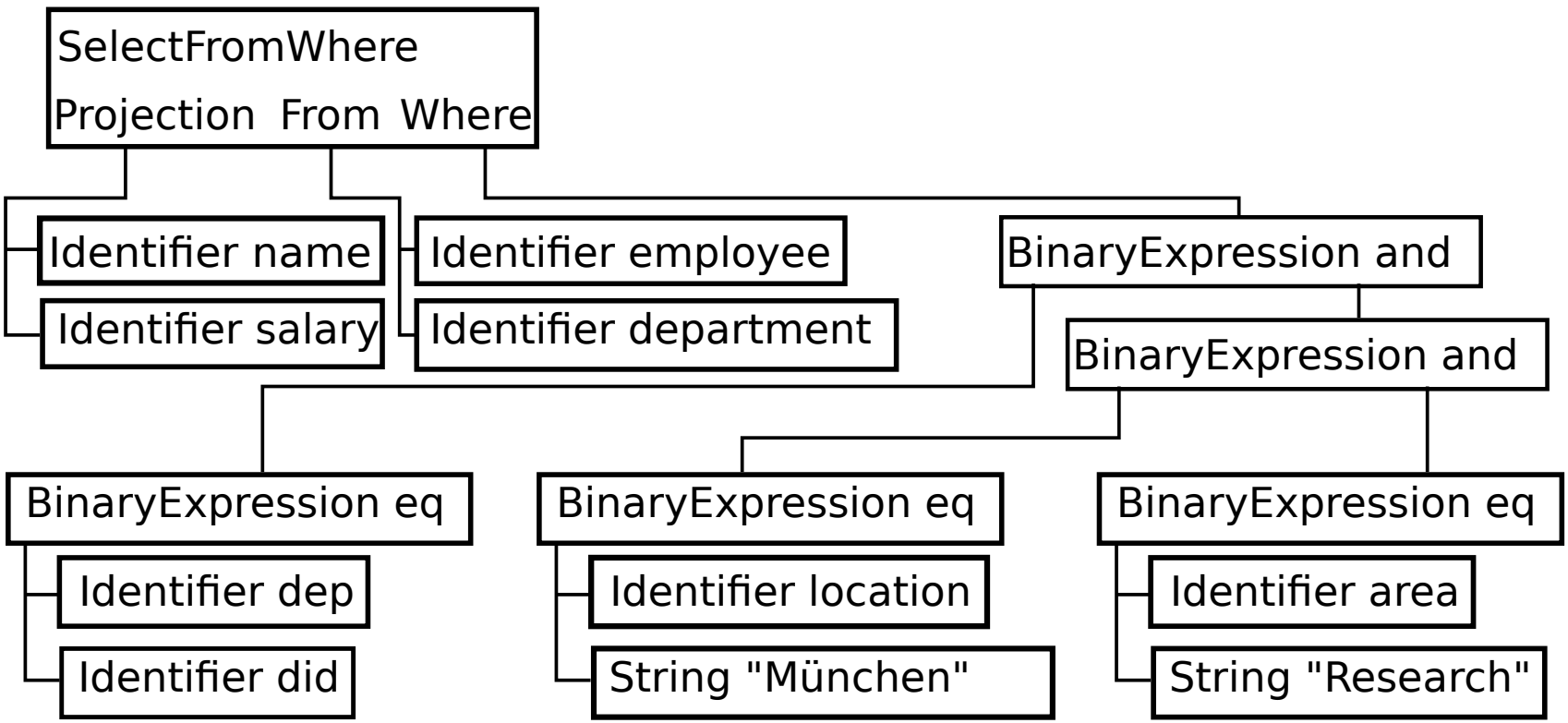
rewrite I, plan generation und rewrite II sind Teile des Anfrageoptimierers

Einfaches Beispiel: Eingabe

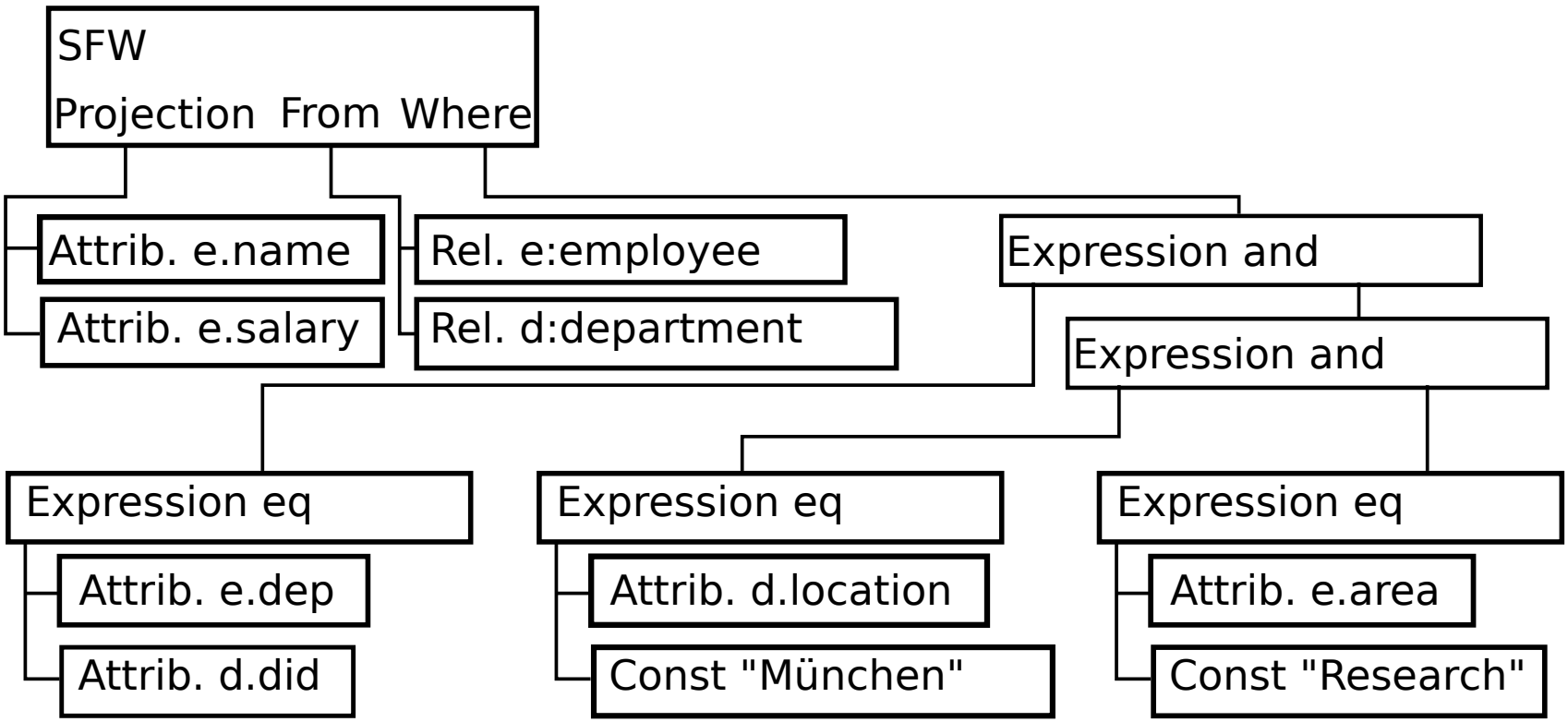
```
select name, salary
from employee, department
where dep=did
      and location="München"
      and area="Research"
```

Beispiel: Parsen der Eingabe

Erstellt einen Abstract Syntax Tree (AST)



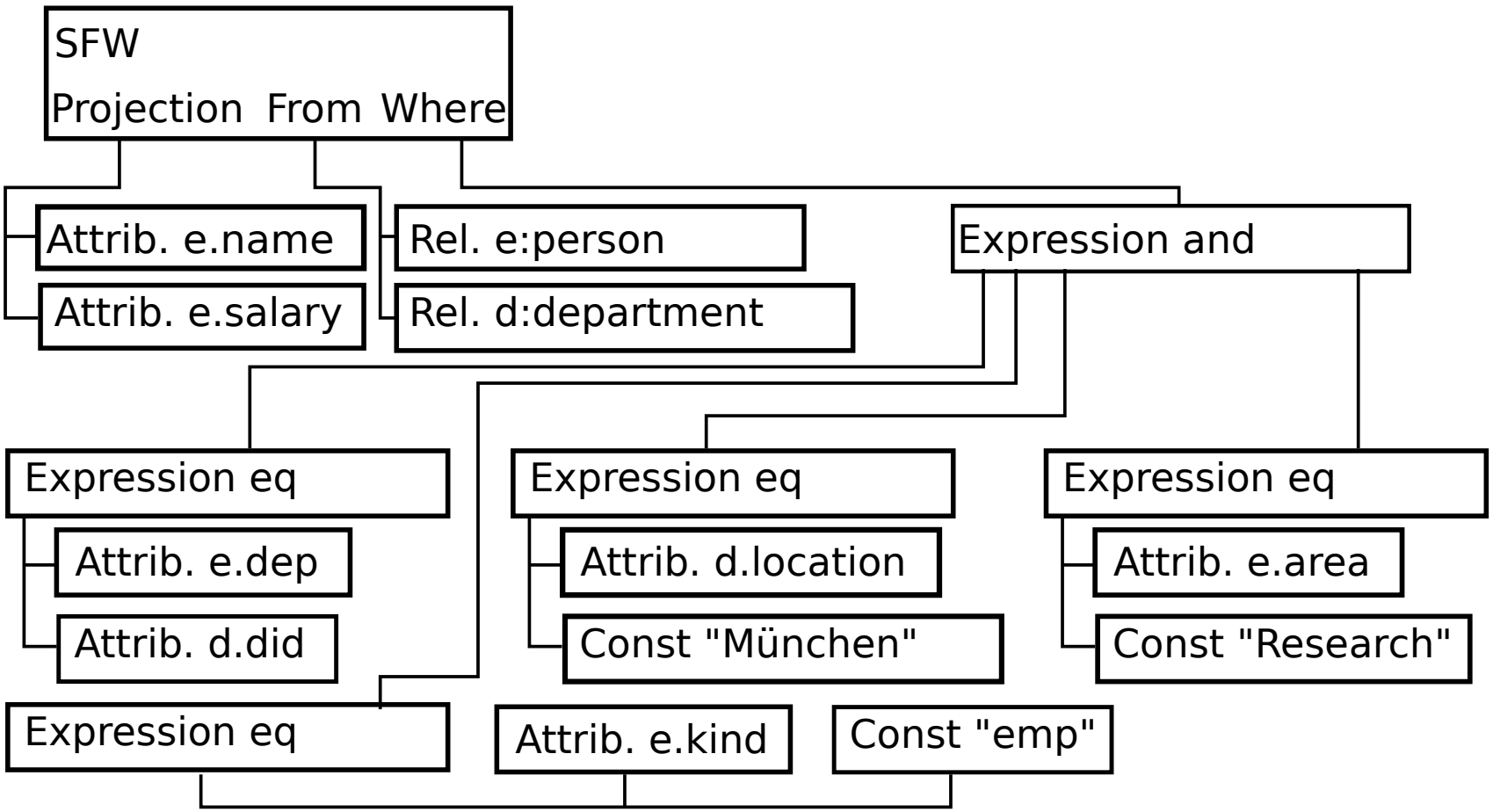
Beispiel: Semantische Analyse



Typen sind hier im Beispiel nicht erwähnt. Ergebnis ist vom Typ $bag < string, number >$

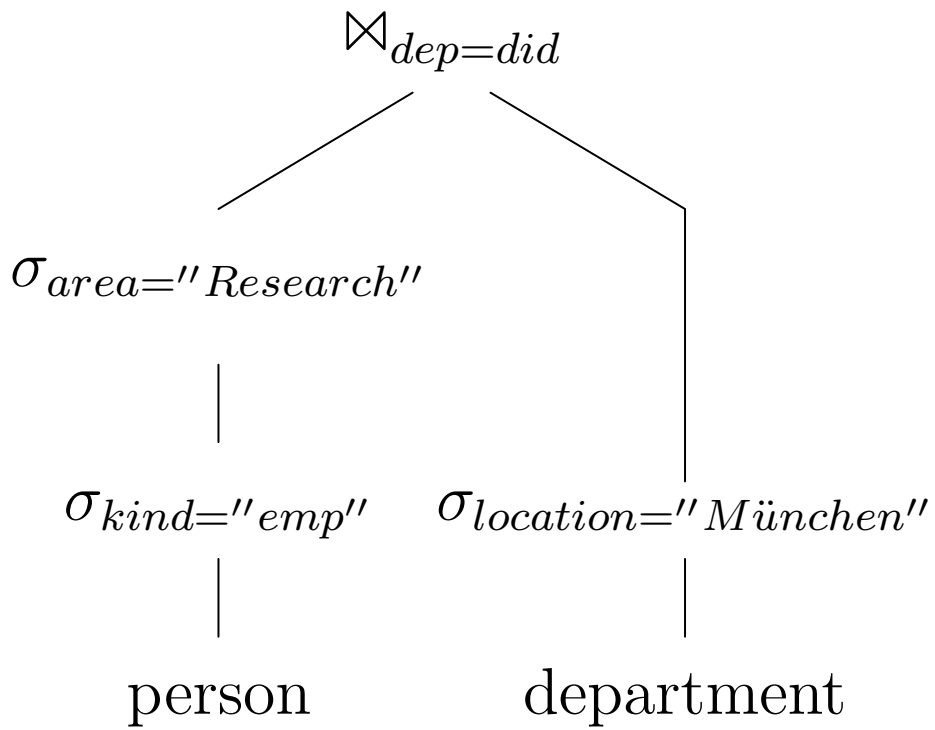
Beispiel: Rewrite I

Auflösen von Sichten, Entschachteln, Optimierung

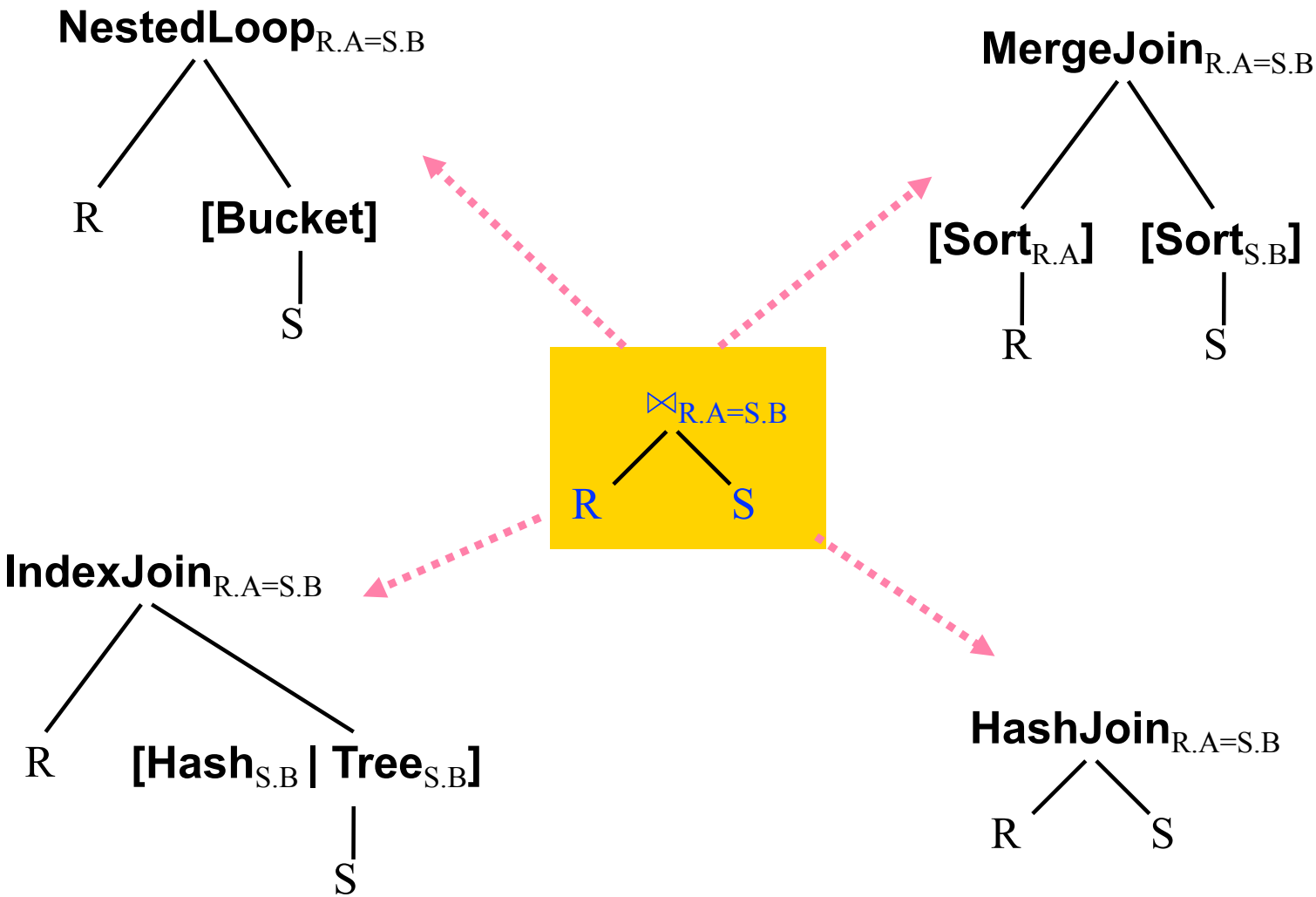


Beispiel: Plan Generierung

Findet die beste Strategie zur Ausführung. Erstellt einen physischen Plan.



z.B. Auswahl Join Implementierung



Beispiel: Code Generierung

Produziert einen ausführbaren Plan

```
<
   @c1 string 0
   @c2 string 0
   @c3 string 0
   @kind string 0
   @name string 0
   @salary float64
   @dep int32
   @area string 0
   @did int32
   @location string 0
   @t1 uint32 local
   @t2 string 0 local
   @t3 bool local
>
[main
   load_string "emp" @c1
   load_string "M\u00fcnchen" @c2
   load_string "Research" @c3
   first_notnull_bool
   <#1 BlockwiseNestedLoopJoin
      memSize 1048576
      [combiner
         unpack_int32 @dep
         eq_int32 @dep @did @t3
         return_if_ne_bool @t3
         unpack_string @name
         unpack_float64 @salary
      ]
[storer
   check_pack 4
   pack_int32 @dep
   pack_string @name
   check_pack 8
   pack_float64 @salary
   load_uint32 0 @t1
   hash_int32 @dep @t1 @t1
   return_uint32 @t1
]
[hasher
   load_uint32 0 @t1
   hash_int32 @did @t1 @t1
   return_uint32 @t1
]
<#2 Tablescan
   segment 1 0 4
   [loader
      unpack_string @kind
      unpack_string @name
      unpack_float64 @salary
      unpack_int32 @dep
      unpack_string @area
      eq_string @kind @c1 @t3
      return_if_ne_bool @t3
      eq_string @area @c3 @t3
      return_if_ne_bool @t3
   ]
>
      <#3 Tablescan
         segment 1 0 5
         [loader
            unpack_int32 @did
            unpack_string @location
            eq_string @location @c2 @t3
            return_if_ne_bool @t3
         ]
      >
   > @t3
   jf_bool 6 @t3
   print_string 0 @name
   cast_float64_string @salary @t2
   print_string 10 @t2
   println
   next_notnull_bool #1 @t3
   jt_bool -6 @t3
]
```

In dieser Vorlesung wird Postgresql verwendet/betrachtet

- "The world's most advanced open source database"
- Einfach zu Installieren
- Gute SQL Unterstützung
- Transaktionen
- Gute Dokumentation (es gibt auch Bücher dazu)

http://www.postgresql.org/

Beispieldatenbanken zum "Üben"

Installieren Sie eine aktuelle Version Postgresql. Folgende Daten stellen wir Ihnen als Postgresql-Dump bereit:

1. Die "Universitäts Datenbank"
2. Eine bereits generierte Version des TPC-H Benchmark-Datensatzes `www.tpc.org/tpch/`

können Sie hier herunterladen:

`http://wwwlgis.informatik.uni-kl.de/extra/vl/dbaw1415/`

Login wird benötigt:

- Benutzername: dbaw1415
- Passwort[1]:

[1] Wird/Wurde in der VL bekannt gegeben

PSQL: Command Line Interface

- Anfragen ausführen, Schema anlegen, Daten aus CVS Dateien laden, Skripte ausführen, ... siehe Befehlsübersicht via \?

```
postgres@      :~$ psql university
psql (9.3.5, server 9.1.14)
Type "help" for help.

university=# select *
university-# from studenten NATURAL JOIN hoeren
university-# where semester > 6;
 matrnr |  name  | semester | vorlnr
--------+--------+----------+--------
  26120 | Fichte |       10 |   5001
  25403 | Jonas  |       12 |   5022
(2 rows)

university=#
```

Beispiel: Datenbank erzeugen und Dump einspielen

Datenbank erzeugen

- In GUI oder via command line:

  `createdb university`

Daten einspielen

- Mit GUI
- oder via command line (und psql Kommando):

  `psql university < uni_db.dmp`
- Achtung: es gibt auch backup/restore. "Dumps" sind reine SQL Statements und Daten, werden erzeugt mit `pg_dump databasename`

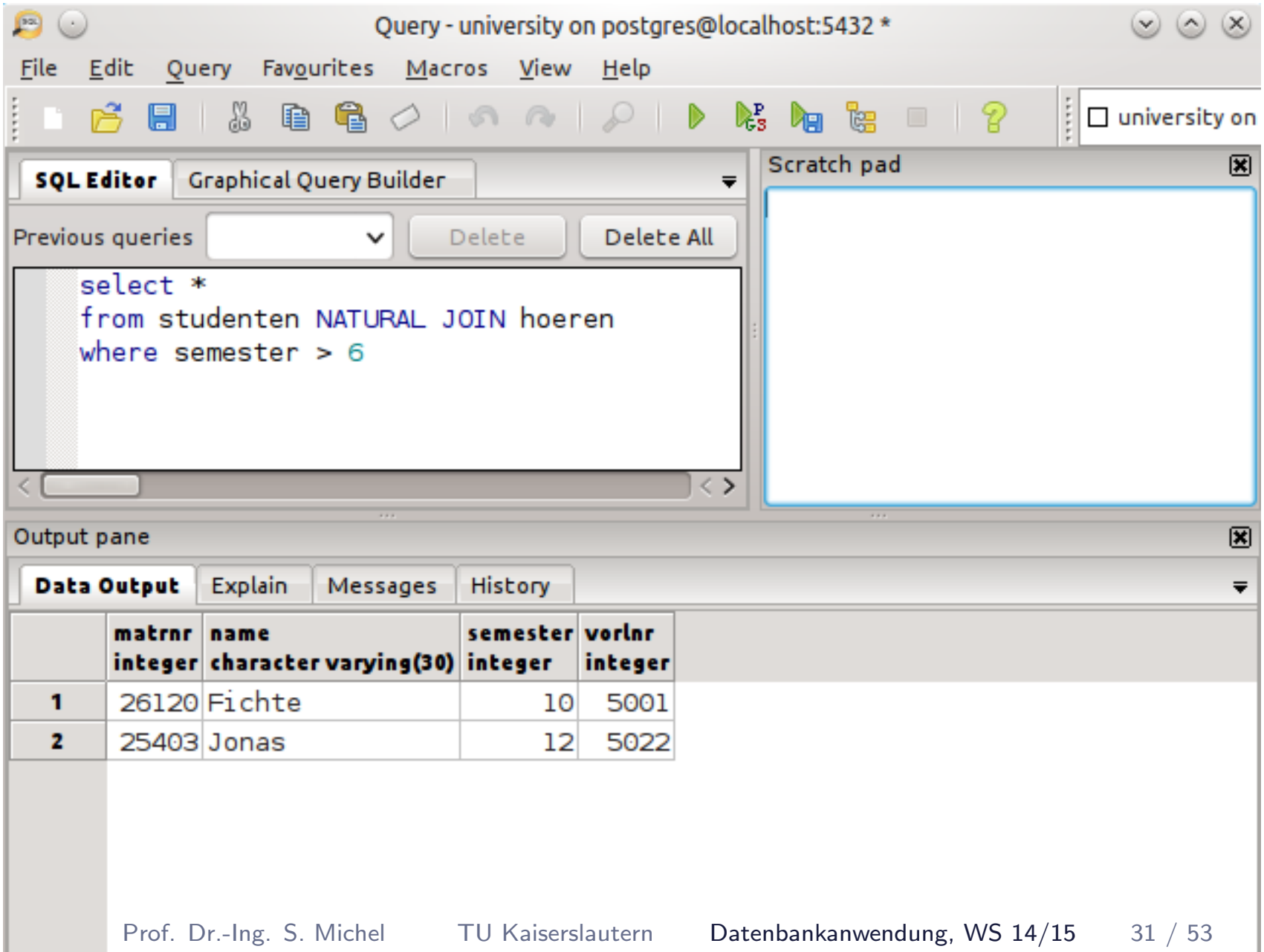
PGAdmin3

User Interface: Download unter `http://www.pgadmin.org/`

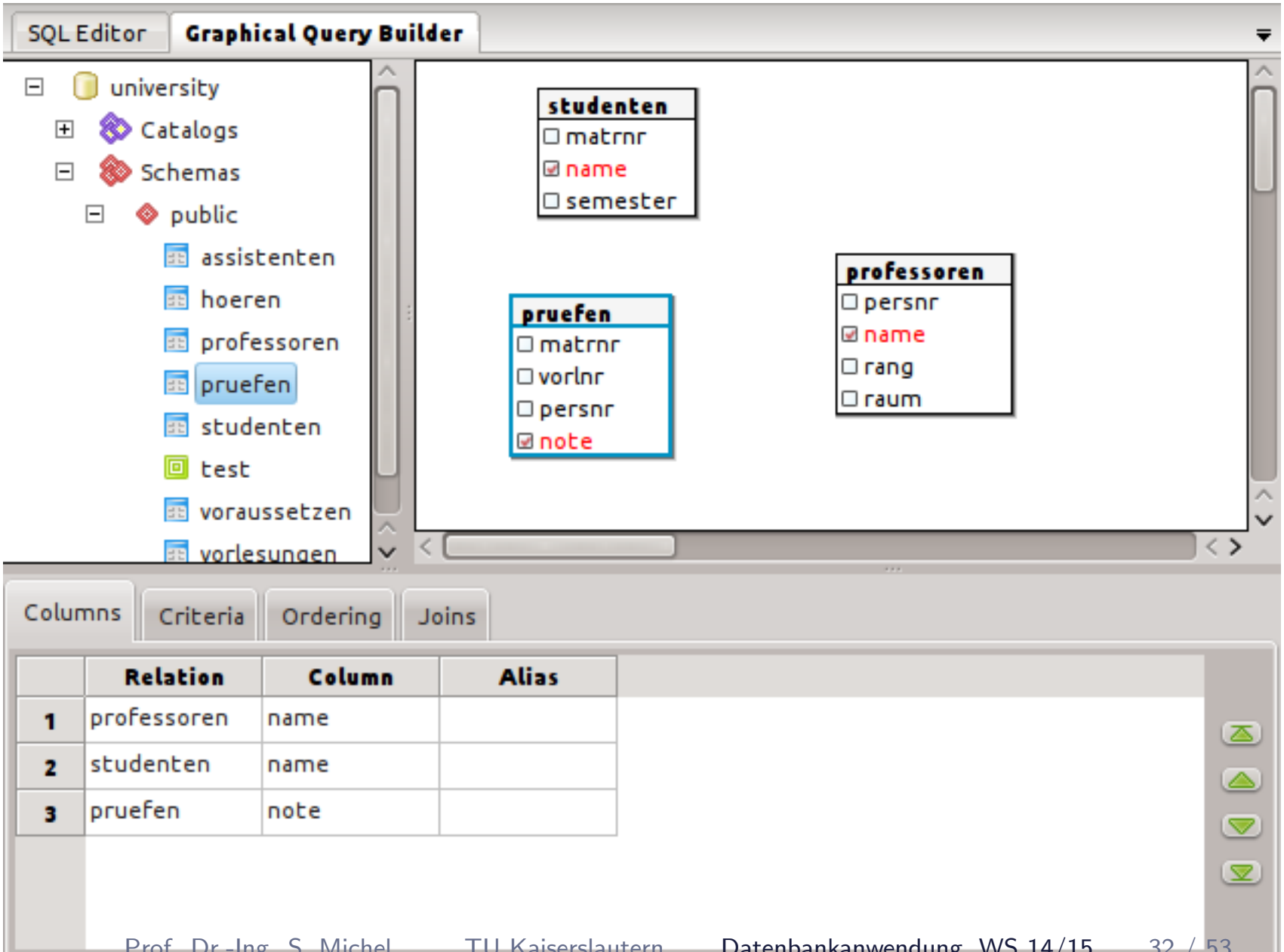
Features

- Query Builder
- Sehr anschauliche Darstellung von Ausführungplänen (explain plan)
- Komfortable Administration (Benutzer, Rechteverwaltung, ...)

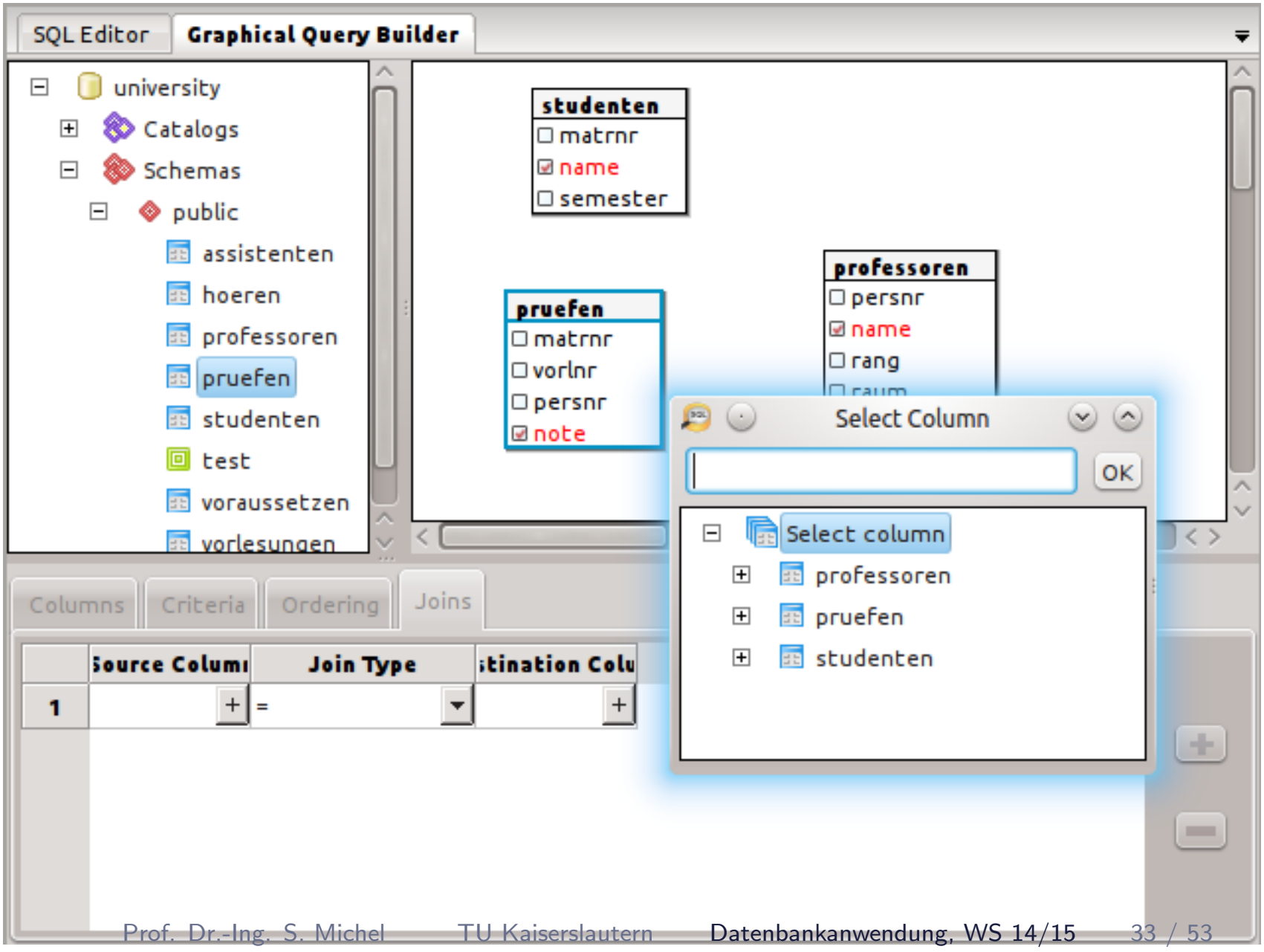
PGAdmin3 (2)


Query - university on postgres@localhost:5432 *
File Edit Query Favourites Macros View Help
university on
SQL Editor
Graphical Query Builder
Previous queries
Delete
Delete All
select *
from studenten NATURAL JOIN hoeren
where semester > 6
Scratch pad
Output pane
Data Output
Explain
Messages
History
matrnr integer
name character varying(30)
semester integer
vorlnr integer
1 26120 Fichte 10 5001
2 25403 Jonas 12 5022

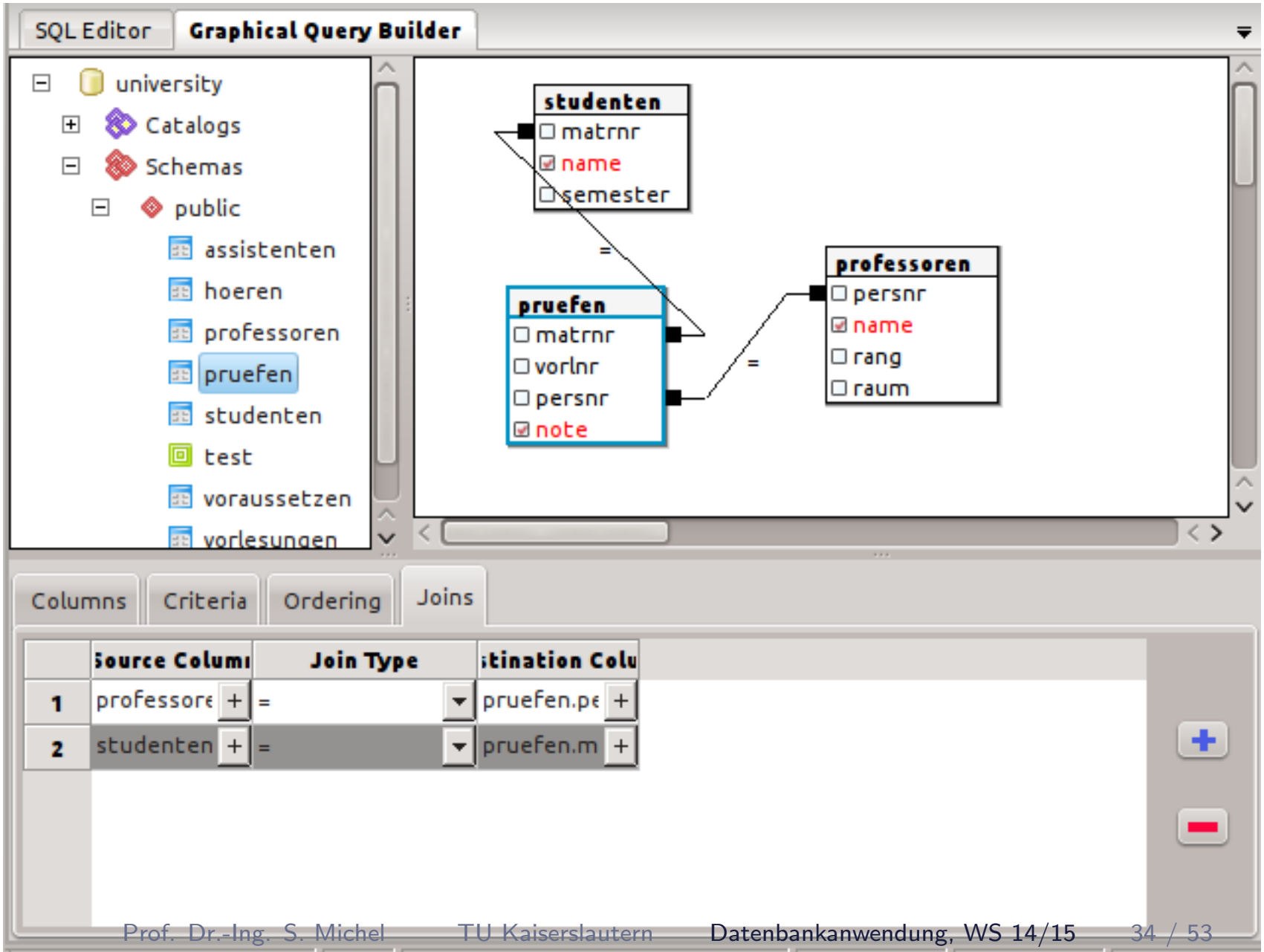
PGAdmin3: query builder (1)



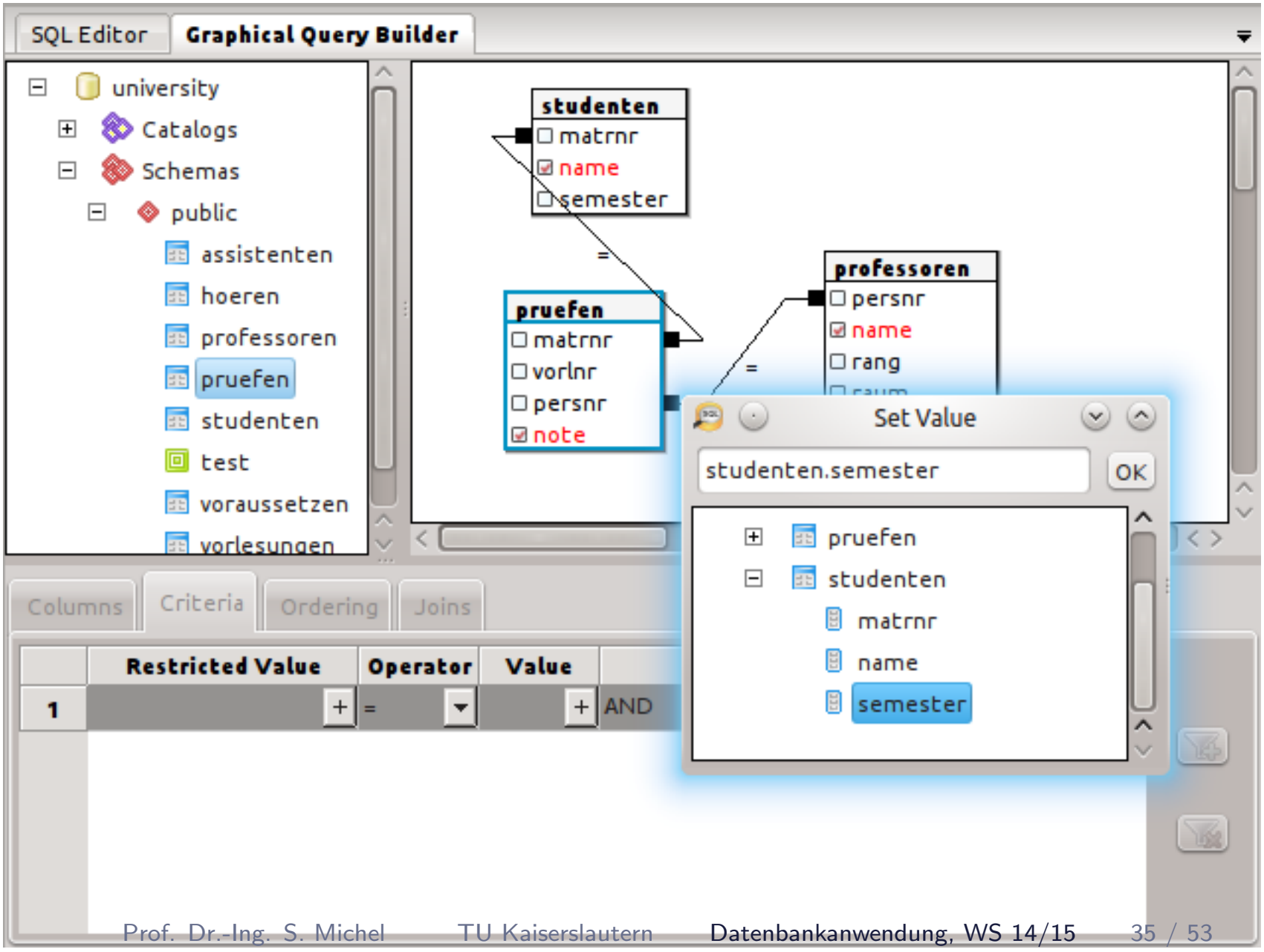
PGAdmin3: query builder (2)



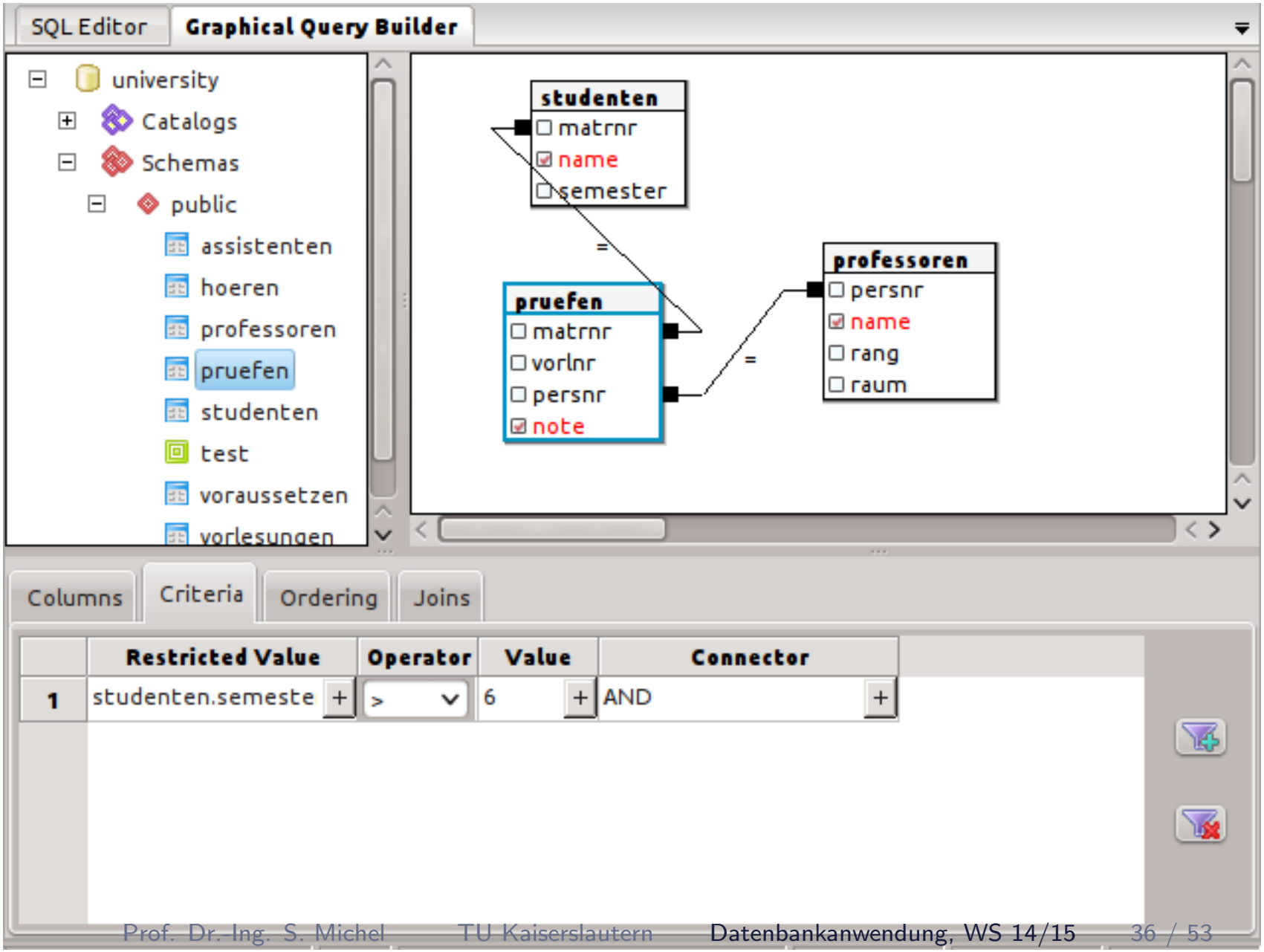
PGAdmin3: query builder (3)



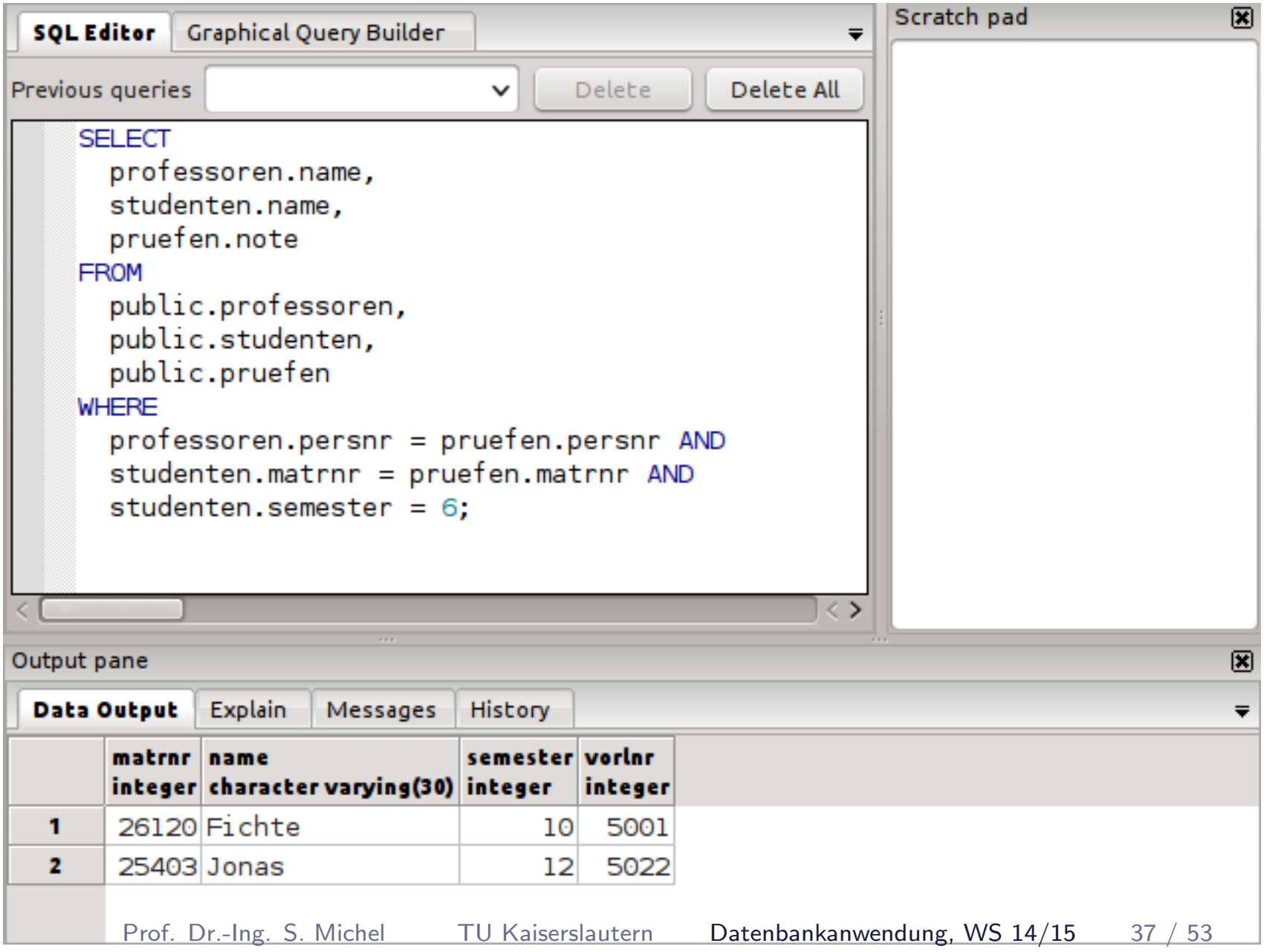
PGAdmin3: query builder (4)



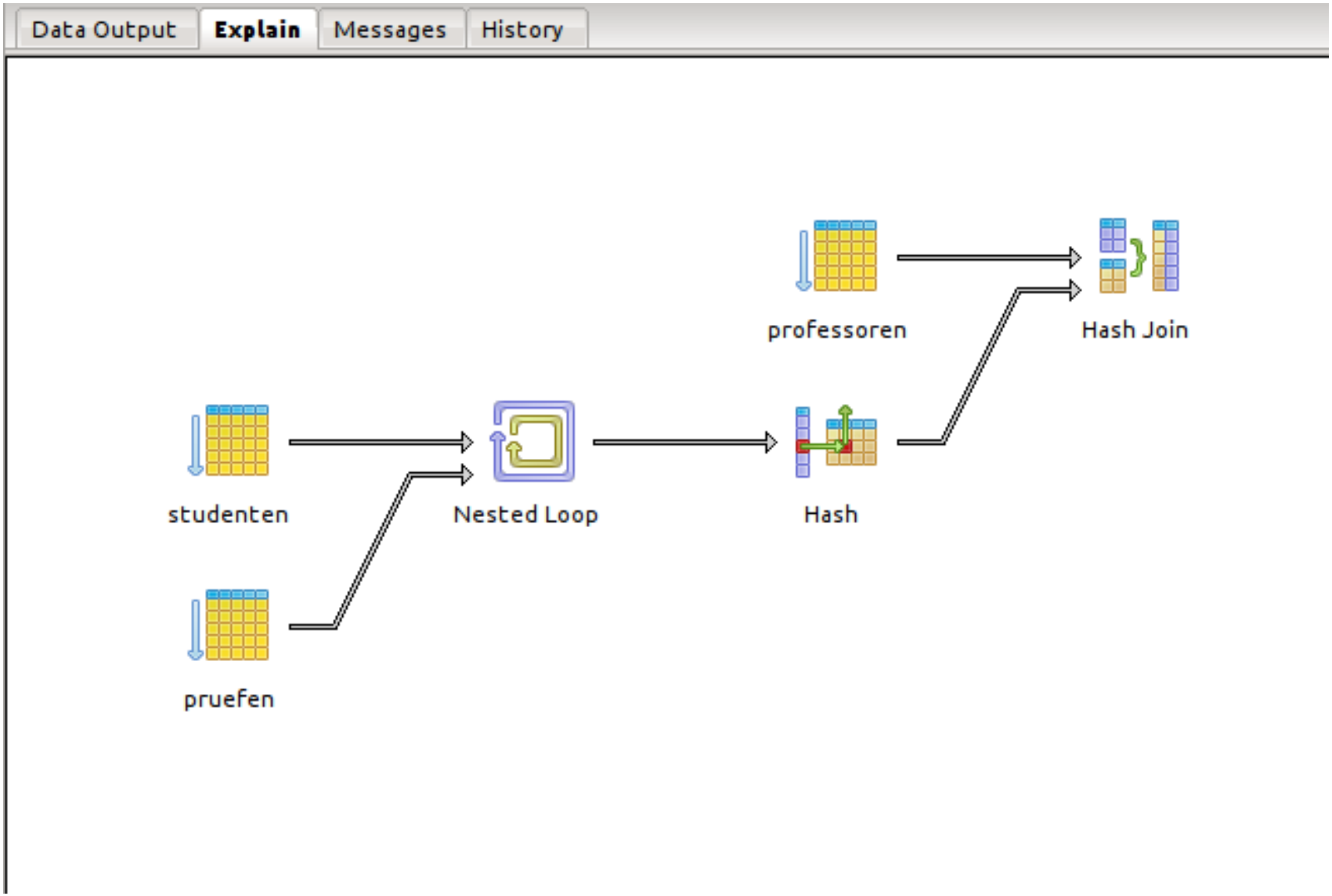
PGAdmin3: query builder (5)



PGAdmin3: query builder (6)



Postgresql/PGAdmin3: Explain Plan (1)



Postgresql/PGAdmin3: Explain Plan (2)

Data Output | Explain | Messages | History

	QUERY PLAN text
1	Hash Join (cost=2.18..3.29 rows=1 width=168) (actual time=21.784..21.785 rows=1 loops=1)
2	Hash Cond: (professoren.persnr = pruefen.persnr)
3	-> Seq Scan on professoren (cost=0.00..1.07 rows=7 width=82) (actual time=11.201..11.203 rows=7 loops=1)
4	-> Hash (cost=2.17..2.17 rows=1 width=94) (actual time=10.563..10.563 rows=1 loops=1)
5	Buckets: 1024 Batches: 1 Memory Usage: 1kB
6	-> Nested Loop (cost=0.00..2.17 rows=1 width=94) (actual time=10.556..10.559 rows=1 loops=1)
7	Join Filter: (studenten.matrnr = pruefen.matrnr)
8	-> Seq Scan on studenten (cost=0.00..1.10 rows=1 width=82) (actual time=0.009..0.011 rows=1 loops=1)
9	Filter: (semester = 6)
10	-> Seq Scan on pruefen (cost=0.00..1.03 rows=3 width=20) (actual time=10.533..10.535 rows=3 loops=1)
11	Total runtime: 21.841 ms

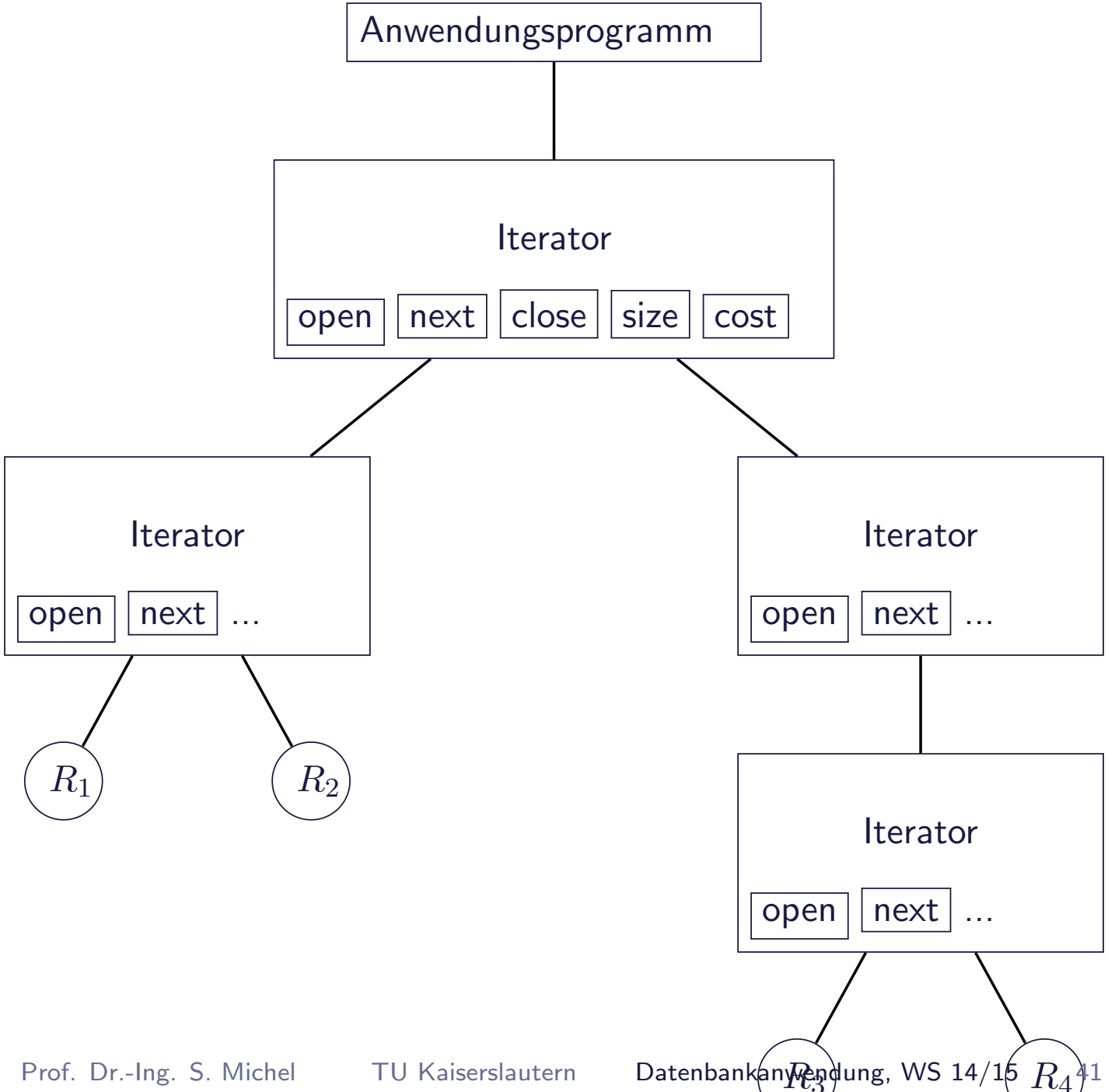
Realisierung von Operatoren und Physische Optimierung

Bislang

- Logische Algebraoperatoren
- Beschreiben was berechnet wird (z.B. Join), aber nicht wie (Algorithmus).

Jetzt: Physische Optimierung

- Welche Implementierungen gibt es für verschiedene Operatoren?
- Im obigen Beispiel (explain plan) haben wir bereits zwei **verschiedene Implementierungen eines Joins** gesehen.
- Iteratorkonzept (Open, Next, Close, ...)


Anwendungsprogramm
Iterator
open
next
close
size
cost
Iterator
open
next
...
Iterator
open
next
...
R_1
R_2
Iterator
open
next
...
R_3
R_4

Iterator

- Open: Konstruktor, initialisiert, öffnet die Eingabe
- Next: Liefert das nächste Ergebnis
- Close: Schließt die Eingabe
- Cost und Size: Geben Informationen über die geschätzten (!) Kosten

Empfehlenswert: Übersichtsartikel von G. Graefe: Query Evaluation Techniques for Large Databases, ACM Computing Surveys, 1993, volume 25, number 2, pages 73-170.

Pull-basierte Verarbeitung

- Operatoren werden in Form von Iteratoren implementiert
- Datenfluss hierbei ist "von unten nach oben"
- Konsument-Produzent Verhältnis
- Der konsumierende Iterator bezieht Tupel von seinen Eingaben, die ebenfalls Iteratoren sind, durch deren `next()` Schnittstelle

- Im Allgemeinen werden Zwischenergebnisse nicht explizit materialisiert.

Anmerkung: Push-basierte Verarbeitung

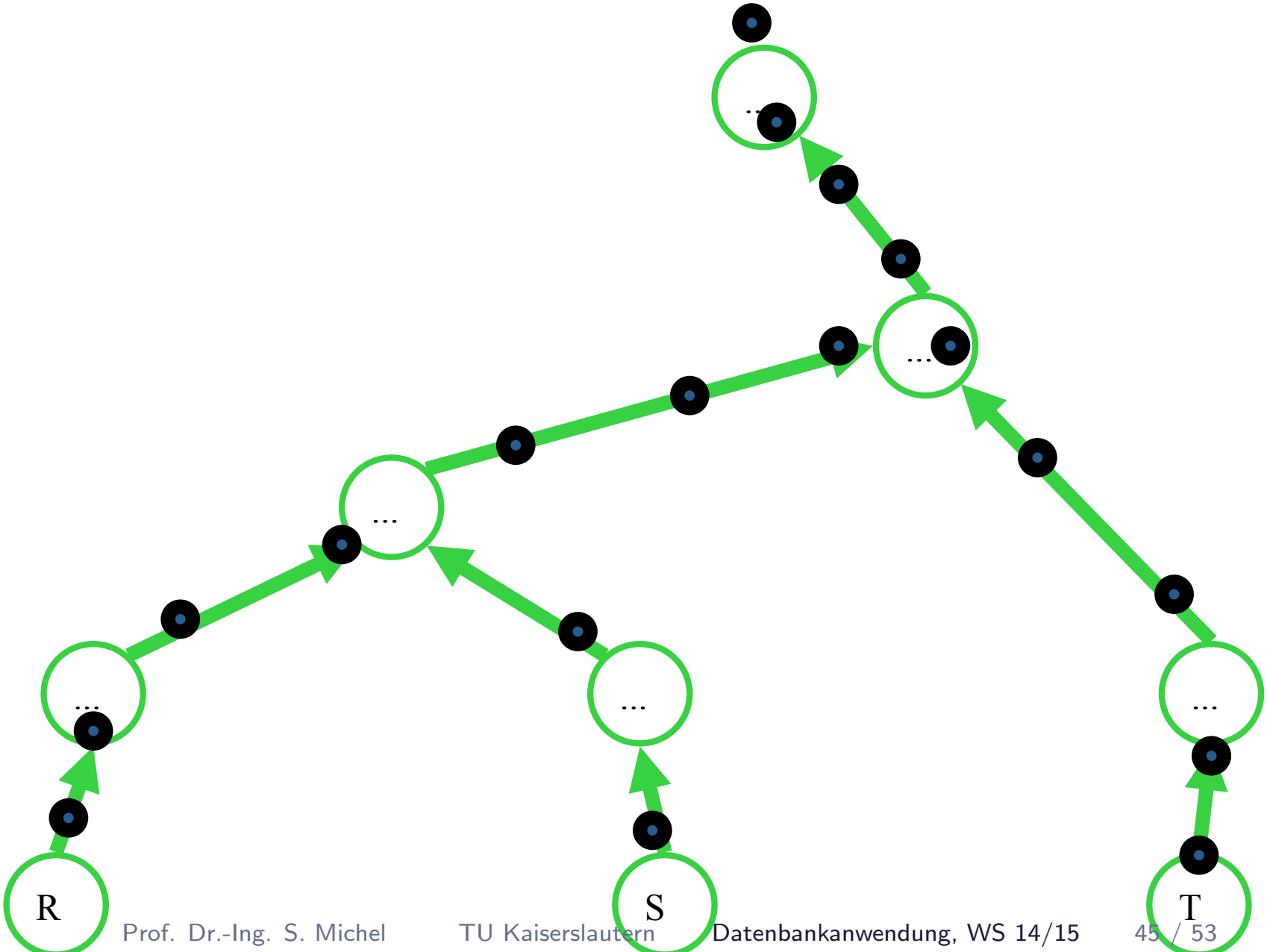
Es gibt insbesondere bei Systemen zur Datenstromverarbeitung die sogenannte Push-basierte Verarbeitung. Dort registrieren sich "Konsumenten" an Datenquellen oder anderen Operatoren. Falls diese neue Daten haben, werden die Daten an die Konsumenten weiter gereicht (aktiv).

Blockierende Operatoren

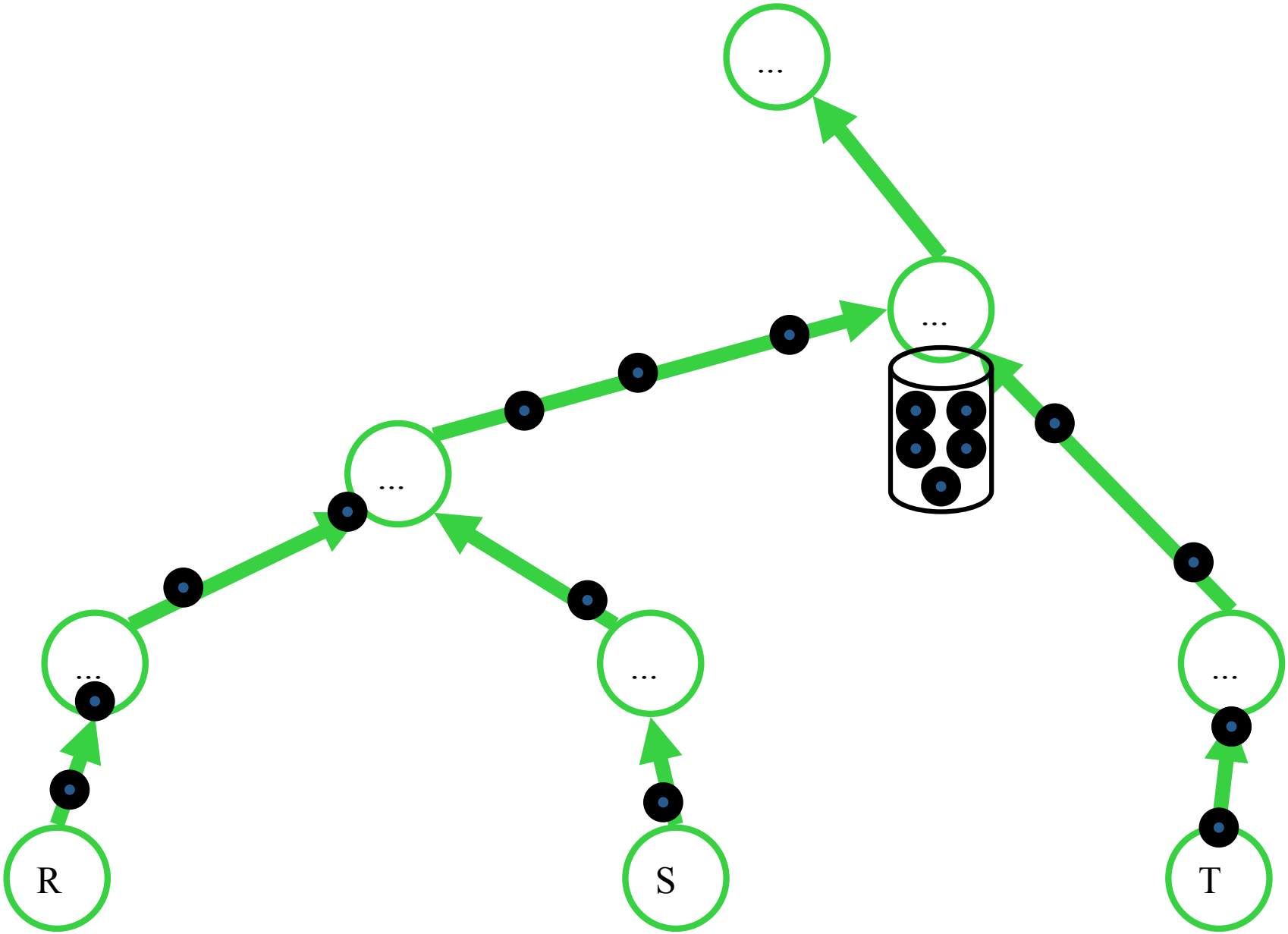
Idealerweise

- Operatoren blockieren den Datenfluss nicht
- D.h. beim Aufruf von `next()` werden darunter liegende Operatoren angefragt via next und das Ergebnis direkt weiter geleitet. Nur wenige Tupel werden dabei gelesen.
- Erlaubt Pipelining
- Im Gegensatz dazu: Operatoren, die den Datenfluss "blockieren", sogenannte Pipeline-Breaker

Pipelining vs. Pipeline-Breaker



Pipelining vs. Pipeline-Breaker



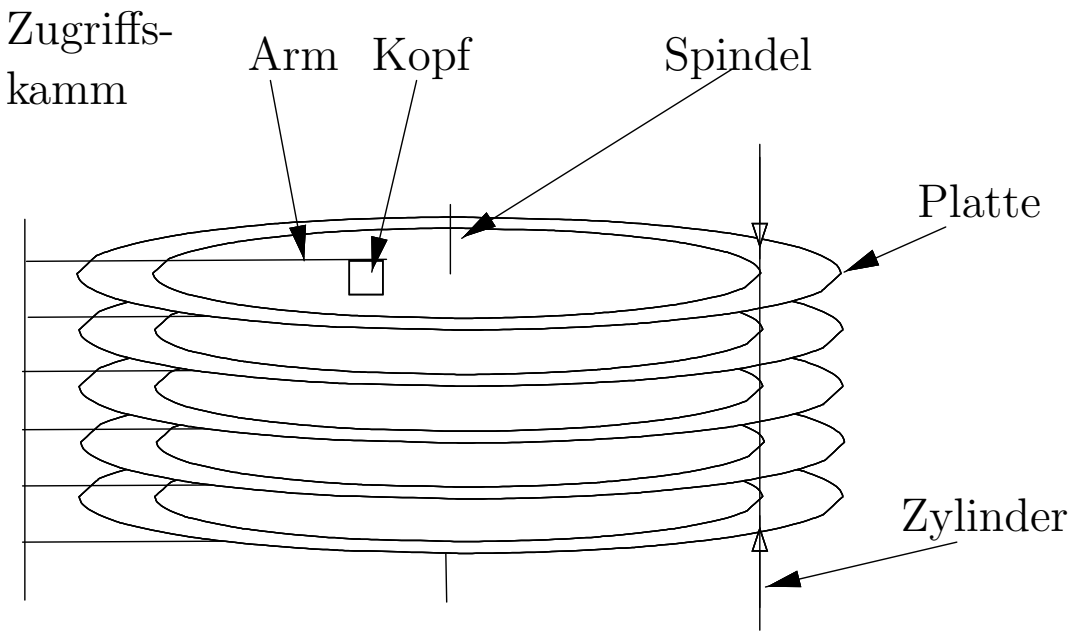
Pipeline-Breaker

- Sortieren
- Duplikate eliminieren (unique, distinct)
- Aggregation: min, max, avg, ...
- Joins (je nach Implementierung)
- Union (je nach Implementierung)

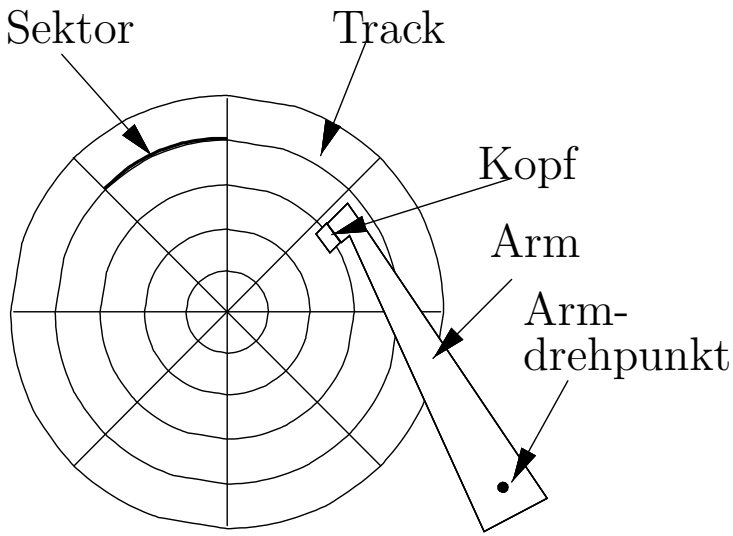
Implementierung von Operatoren

- Bei der Erzeugung eines physischen Anfrageplans muss entschieden werden wie genau die Anfrage ausgeführt werden soll
- Welche Implementierungen gibt es für die diversen Operatoren?
- Welche Implementierung ist effizienter?
- Und wie kann dies überhaupt berechnet/vorhergesagt werden? (Kostenmodelle)
- Gibt es Indexstrukturen, die ausgenutzt werden können?
- ...

Aufbau einer (klassischen) Festplatte



a. Seitenansicht

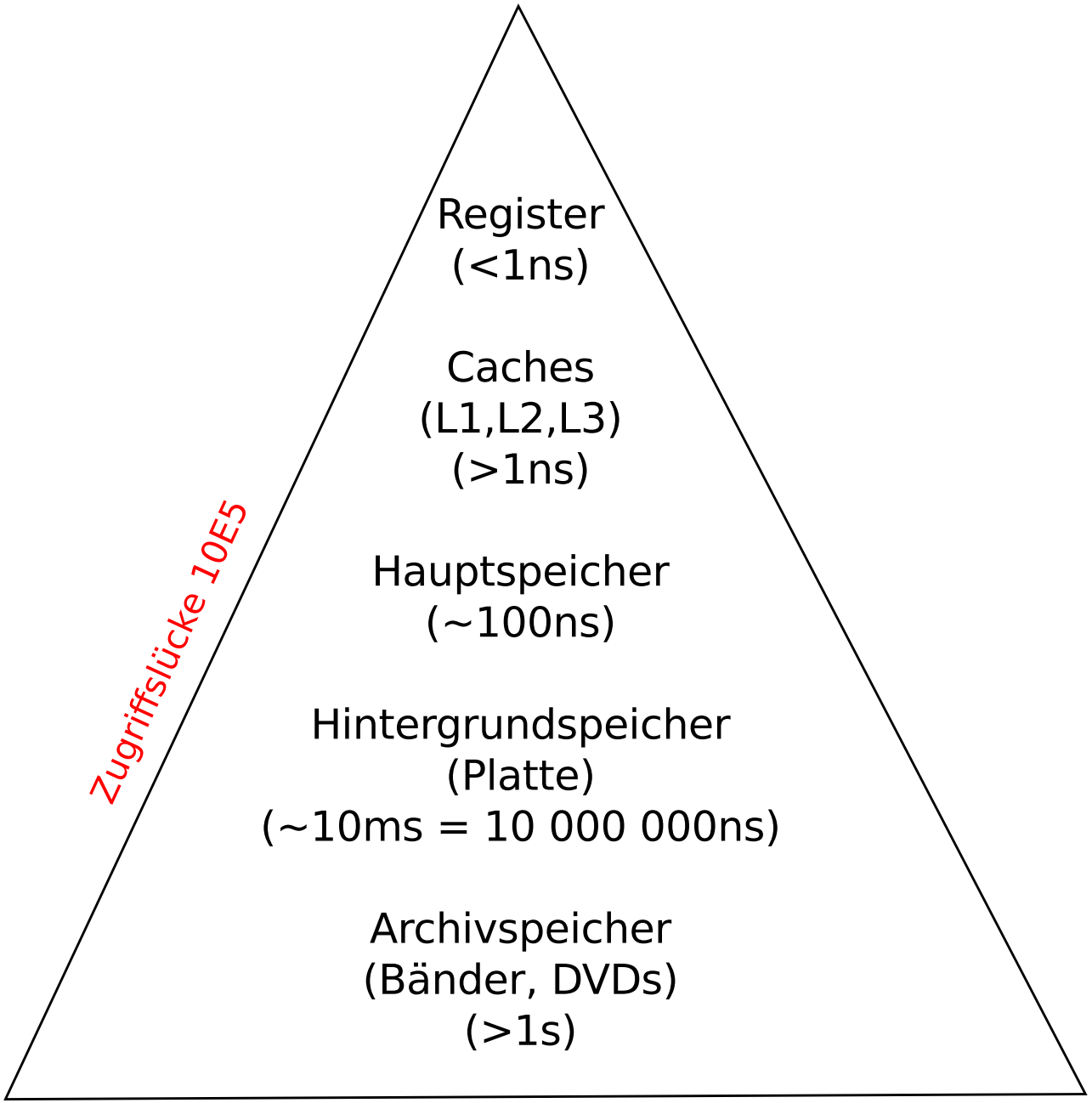


b. Draufsicht

Was kostet wieviel?

- L1 cache reference 0.5 ns
- L2 cache reference 7 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10,000 ns
- Send 2K bytes over 1 Gbps network 20,000 ns
- Read 1 MB sequentially from memory 250,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30,000,000 ns
- Send packet CA $\rightarrow$ Netherlands $\rightarrow$ CA 150,000,000 ns

Numbers by Jeff Dean (Google)



Analogie

Kopf
(1 Min)

Raum
(10 Min)

Stadt
(1 Std)

Pluto
(2 Jahre)

Andromeda
(2000 Jahre)

Aufbau Festplatte (Tracks, Sektoren, Zonen)

- Track: Teil eines Zylinders auf einer Platte
- Sektor: Teil eines Tracks. Anzahl Sektoren pro Track ursprünglich gleich für alle Tracks
- Aber, auf Zylinder/Tracks weiter "außen" passen mehr Sektoren. Daher, aktuelle Hardware, variable Anzahl von Tracks: äußere Tracks haben mehr Sektoren als innere Tracks; Zylinder sind in Zonen unterteilt.

Blöcke

- Aka. Sektoren, Aka. physical Record. Kleinste Transfereinheit bei Block-Storage-Devices. Seit wenigen Jahren typischerweise 4KB oder sonst (historisch) 512 Byte groß.
- Achtung, es gibt auch Unterschiede zu Blöcken bzw. Seiten des Dateisystems.

Beispiel

Die Festplatte SAMSUNG HD103SJ, die in meinem Desktop im Büro eingebaut ist hat laut (Linux tool) `hdparm -i` (oder `hdparm -I` für mehr Details):

- 1953525168 Sektoren
- a 512 Bytes.

Das macht: 1 TB

Ausgabe (Auszug) von hdparm -I /dev/sda

```
CHS current addressable sectors:   16514064
LBA    user addressable sectors:  268435455
LBA48  user addressable sectors: 1953525168
Logical  Sector size:                   512 bytes
Physical Sector size:                   512 bytes
device size with M = 1024*1024:      953869 MBytes
device size with M = 1000*1000:     1000204 MBytes (1000 GB)
```