

184.686 VU Database Systems

Anfrageoptimierung

Katja Hose

Institute for Logic and Computation

Summer Term 2024



Informatics

Lernziele

Lernziele

- Verstehen, wie SELECT-Anfragen ausgeführt werden
- Kenntnis von grundlegenden Join-Algorithmen
- Grundlagen der heuristischen (logischen) Anfrageoptimierung verstehen
- Grundlagen der physischen Anfrageoptimierung verstehen

Motivation

- Die Kenntnis von Grundlagen der Anfragenbearbeitung und -optimierung sind Voraussetzungen für Database-Tuning

- 1 Motivation und Einführung
 - Anfragebearbeitung
 - Anfrageoptimierung
- 2 Logische (heuristische) Anfrageoptimierung
 - Äquivalenzerhaltende Transformationsregeln
 - Phasen der logischen Anfrageoptimierung
- 3 Implementierung von Operationen
 - Selektion (Access Paths)
 - Join-Strategien
- 4 Kostenbasierte (physische) Anfrageoptimierung
 - Selektivität und Kardinalität
 - Kostenschätzung
 - PostgreSQL

Ausführen einer SQL-Anfrage

Die Klauseln werden in folgender Reihenfolge angegeben.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

Aber die Anfrage wird in einer anderen Reihenfolge ausgeführt.

- Kartesisches Produkt der Tabellen in der FROM-Klausel
- Prädikate in der WHERE-Klausel
- GROUP-BY-Klausel
- Prädikate in der HAVING-Klausel (um Gruppen zu eliminieren)
- Aggregatfunktionen für jede verbleibende Gruppe berechnen
- Projektion auf Spalten der SELECT-Klausel

Ausführen einer SQL-Anfrage

Die Klauseln werden in folgender Reihenfolge angegeben.

- SELECT *column(s)*
- FROM *table list*
- WHERE *condition*
- GROUP BY *grouping column(s)*
- HAVING *group condition*
- ORDER BY *sort list*

SQL ist deklarativ!

Schritte der Anfragebearbeitung

Anfrage in einer abstrakten Anfragesprache

↓
Scanning, Parsing, and Semantic Analysis

↓ Intermediate Query Plan

Query Optimizer

↓ Execution Plan

Code Generator

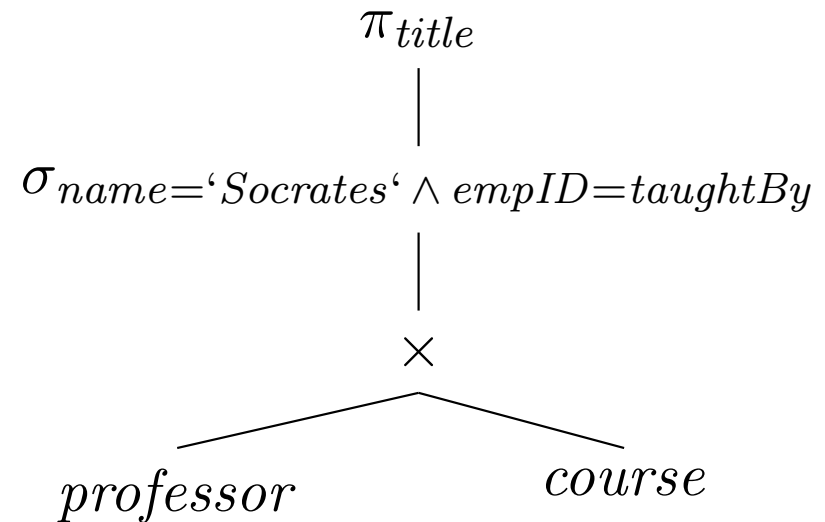
↓ Code for Query Execution

Runtime Database Processor

↓
Anfrageergebnis

Parsen einer Anfrage in einen initialen Anfrageplan

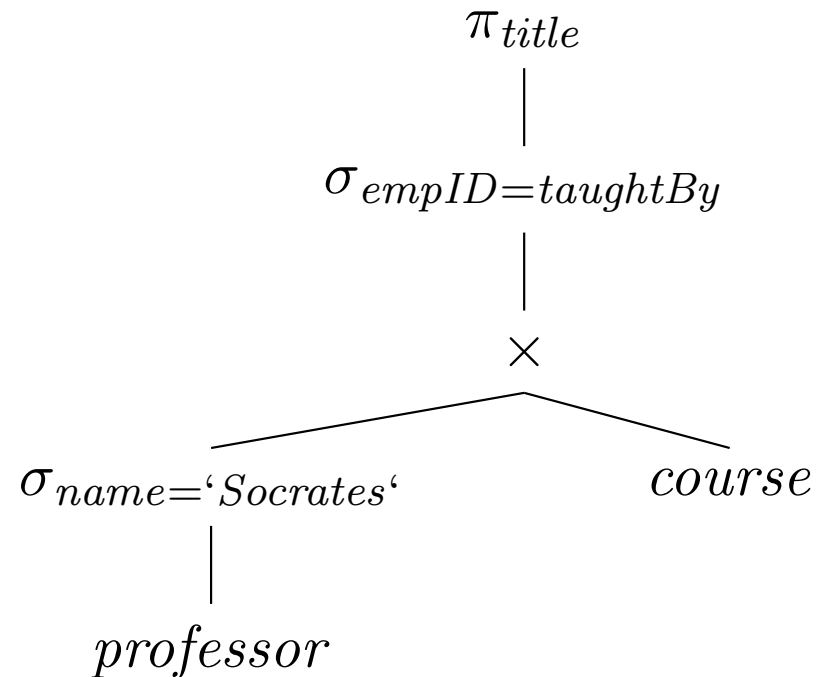
SELECT title
FROM professor, course
WHERE name='Socrates' AND \Rightarrow
 empID = taughtBy;



$\pi_{title}(\sigma_{name='Socrates' \wedge empID=taughtBy}(professor \times course))$

Alternativer Anfrageplan

SELECT title
FROM professor, course
WHERE name='Socrates' AND \Rightarrow
 empID = taughtBy;



$\pi_{title}(\sigma_{empID=taughtBy}(\sigma_{name='Socrates'} professor \times course))$

Anfrageoptimierung

Alternativen

- Äquivalente Ausführungspläne
- Algorithmen zur Ausführung von Algebraoperatoren
- Methoden, um auf Relationen zuzugreifen (Indexe)

Bei gleichem Ergebnis können Ausführungskosten sehr unterschiedlich sein.

Theorie vs. Realität

Es ist nicht die Aufgabe des Benutzers „effiziente“ Anfragen zu schreiben, sondern die der Anfrageoptimierung effiziente Ausführungspläne zu finden!

Aber in der Realität... Optimierer sind nicht perfekt.

Anfrageausführungskosten

Kostenmodell

- Gesamte Zeit bis das Anfrageergebnis vorliegt (**Response Time**)
- Viele Faktoren tragen bei
 - Festplattenzugriff
 - CPU-Kosten
 - Netzwerkkommunikation
 - Aktueller Query-Load
 - Parallelisierung
- Festplattenzugriff dominiert
 - Block Access Time: Seek Time, Rotation Time

Anfrageoptimierung

Logische Anfrageoptimierung

- Relationale Algebra
- Äquivalenzerhaltende Transformationsregeln
- Heuristische Optimierung

Physische Anfrageoptimierung

- Algorithmen and Operatorimplementationen
- Kostenmodell

- 2 Logische (heuristische) Anfrageoptimierung
 - Äquivalenzerhaltende Transformationsregeln
 - Phasen der logischen Anfrageoptimierung

Logische Anfrageoptimierung

Logische Anfrageoptimierung

- Grundlage: Äquivalenzerhaltende Transformationsregeln
- Algebraische Transformationen bilden den Suchraum
- Gegen sei ein initialer algebraischer Ausdruck:
verwende äquivalenzerhaltende Transformationsregeln, um neue Ausdrücke abzuleiten

Was ist ein guter Plan?

- Genaue Entscheidung ohne Kostenfunktion nicht möglich

⇒ logische Anfrageoptimierung basiert auf Heuristiken

Hauptziel der logischen Anfrageoptimierung

Größe von Zwischenergebnissen reduzieren

Äquivalenzerhaltende Transformationsregeln

Aufbrechen von Konjunktionen im Selektionsprädikaten

$$\sigma_{c_1 \wedge c_2 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$$

σ is kommutativ

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

π -Kaskaden

If $L_1 \subseteq L_2 \subseteq \dots \subseteq L_n$ dann gilt

$$\pi_{L_1}(\pi_{L_2}(\dots(\pi_{L_n}(R))\dots)) \equiv \pi_{L_1}(R)$$

Äquivalenzerhaltende Transformationsregeln

Vertauschen der Reihenfolge von σ und π

Falls die Selektion sich nur auf Attribute A_1, \dots, A_n der Projektionsliste bezieht, können die beiden Operationen vertauscht werden:

$$\pi_{A_1, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, \dots, A_n}(R))$$

\cup, \cap und \bowtie sind kommutativ

$$R \bowtie_c S \equiv S \bowtie_c R$$

Äquivalenzerhaltende Transformationsregeln

Vertauschen von σ und \bowtie

Falls das Selektionsprädikat c nur auf Attribute der Relation R zugreift, kann man die beiden Operationen vertauschen:

$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j S$$

Falls das Selektionsprädikat c eine Konjunktion der Form $c_1 \wedge c_2$ ist und c_1 sich nur auf Attribute aus R und c_2 sich nur auf Attribute aus S bezieht, gilt folgende Äquivalenz:

$$\sigma_c(R \bowtie_j S) \equiv \sigma_{c_1}(R) \bowtie_j \sigma_{c_2}(S)$$

Äquivalenzerhaltende Transformationsregeln

Vertauschen von π und \bowtie

Gegeben sei Projektionsliste $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, wobei A_i Attribute aus R und B_i Attribute aus S sind. Falls sich das Joinprädikat c nur auf Attribute aus L bezieht, gilt folgende Umformung:

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \dots, A_n}(R)) \bowtie_c (\pi_{B_1, \dots, B_m}(S))$$

Äquivalenzerhaltende Transformationsregeln

\bowtie, \cap, \cup sind (jeweils einzeln betrachtet) assoziativ

Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

$$(R \Phi S) \Phi T \equiv R \Phi (S \Phi T)$$

σ ist distributiv mit $\cap, \cup, -$

Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

$$\sigma_c(R \Phi S) \equiv (\sigma_c(R)) \Phi (\sigma_c(S))$$

π ist distributiv mit \cup

$$\pi_c(R \cup S) \equiv (\pi_c(R)) \cup (\pi_c(S))$$

Äquivalenzerhaltende Transformationsregeln

Join und/oder Selektionsprädikate können mittels de Morgan's Regeln umgeformt werden

$$\neg(c_1 \wedge c_2) \equiv (\neg c_1) \vee (\neg c_2)$$

$$\neg(c_1 \vee c_2) \equiv (\neg c_1) \wedge (\neg c_2)$$

Kombination von Kartesischem Produkt und Selektion

Ein kartesisches Produkt, das von einer Selektionsoperation gefolgt wird, deren Selektionsprädikat Attribute aus beiden Operanden des kartesischen Produktes enthält, kann in eine Joinoperation umgeformt werden.

$$\sigma_{\theta}(R \times S) \equiv R \bowtie_{\theta} S$$

Ebenfalls relevant: die alternativen Ausdrücke for Operatoren in der relationalen Algebra.

- 2 Logische (heuristische) Anfrageoptimierung
 - Äquivalenzerhaltende Transformationsregeln
 - Phasen der logischen Anfrageoptimierung

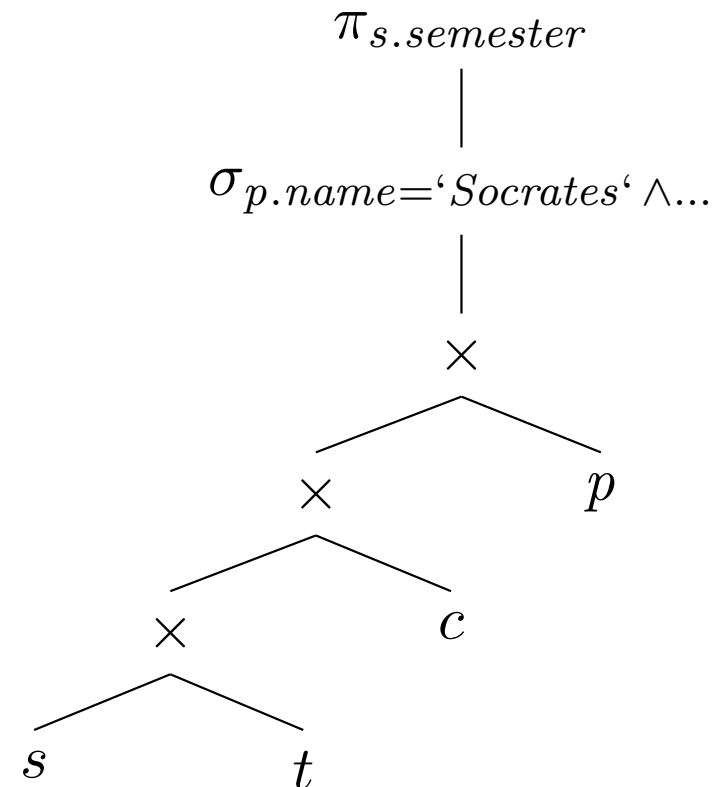
Phasen der logischen Anfrageoptimierung

- ① Aufbrechen von Selektionen
- ② Verschieben der Selektionen so weit wie möglich nach unten (pushing selections)
- ③ Joins einführen (Zusammenfassen von Selektionen und Kreuzprodukten)
- ④ Join-Reihenfolge bestimmen, so dass möglichst kleine Zwischenergebnisse entstehen
Heuristik: Joins mit Input von Selektionen vor anderen Joins auswerten
- ⑤ ggf. Einführen von Projektionen
- ⑥ Verschieben der Projektionen so weit wie möglich nach unten
Nicht immer sinnvoll

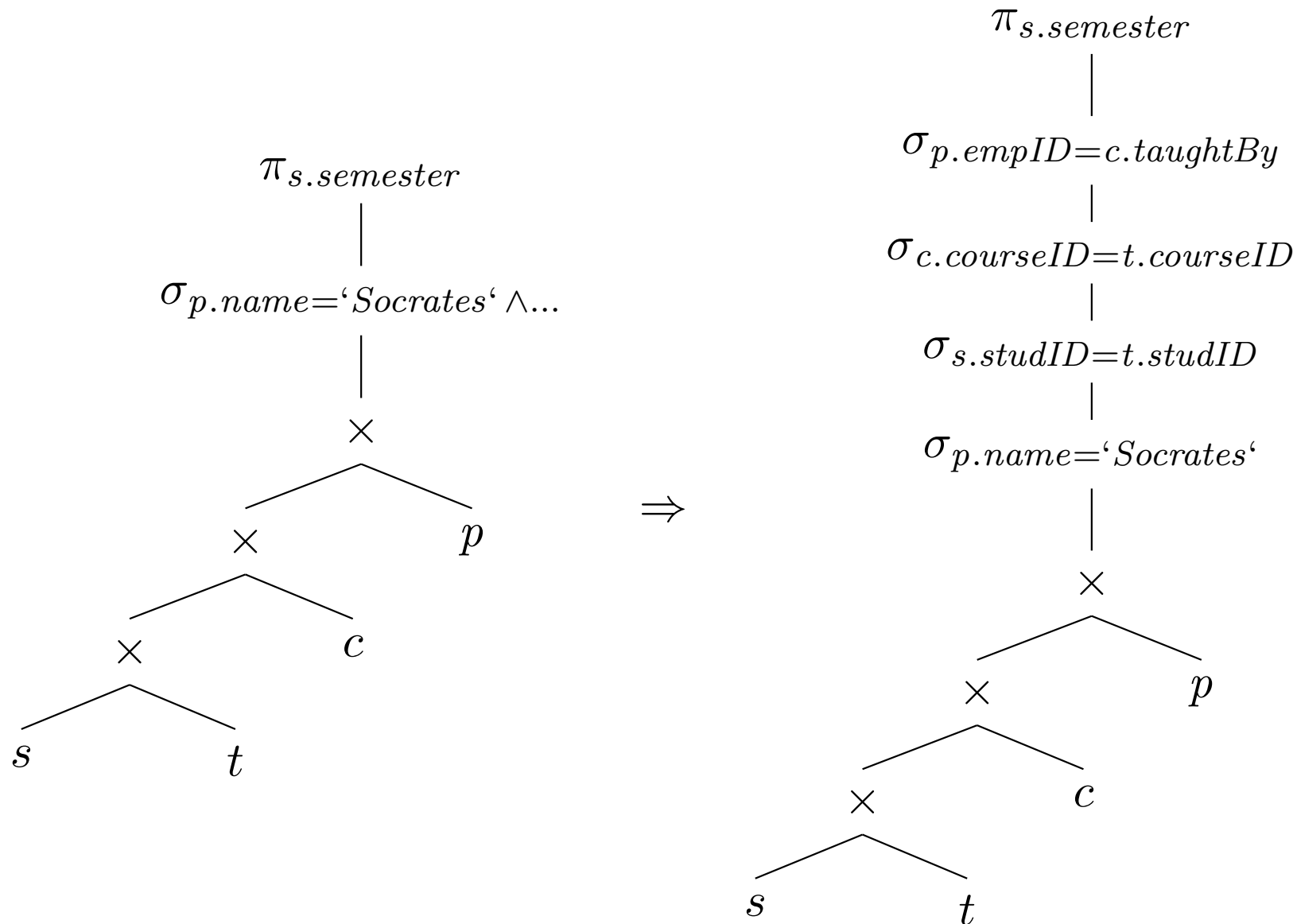
Beispiel

SELECT DISTINCT s.semester
FROM student s, takes t,
 course c, professor p
WHERE p.name='Socrates' AND
 c.taughtBy = p.empID AND
 c.courseID = t.courseID AND
 t.studID = s.studID;

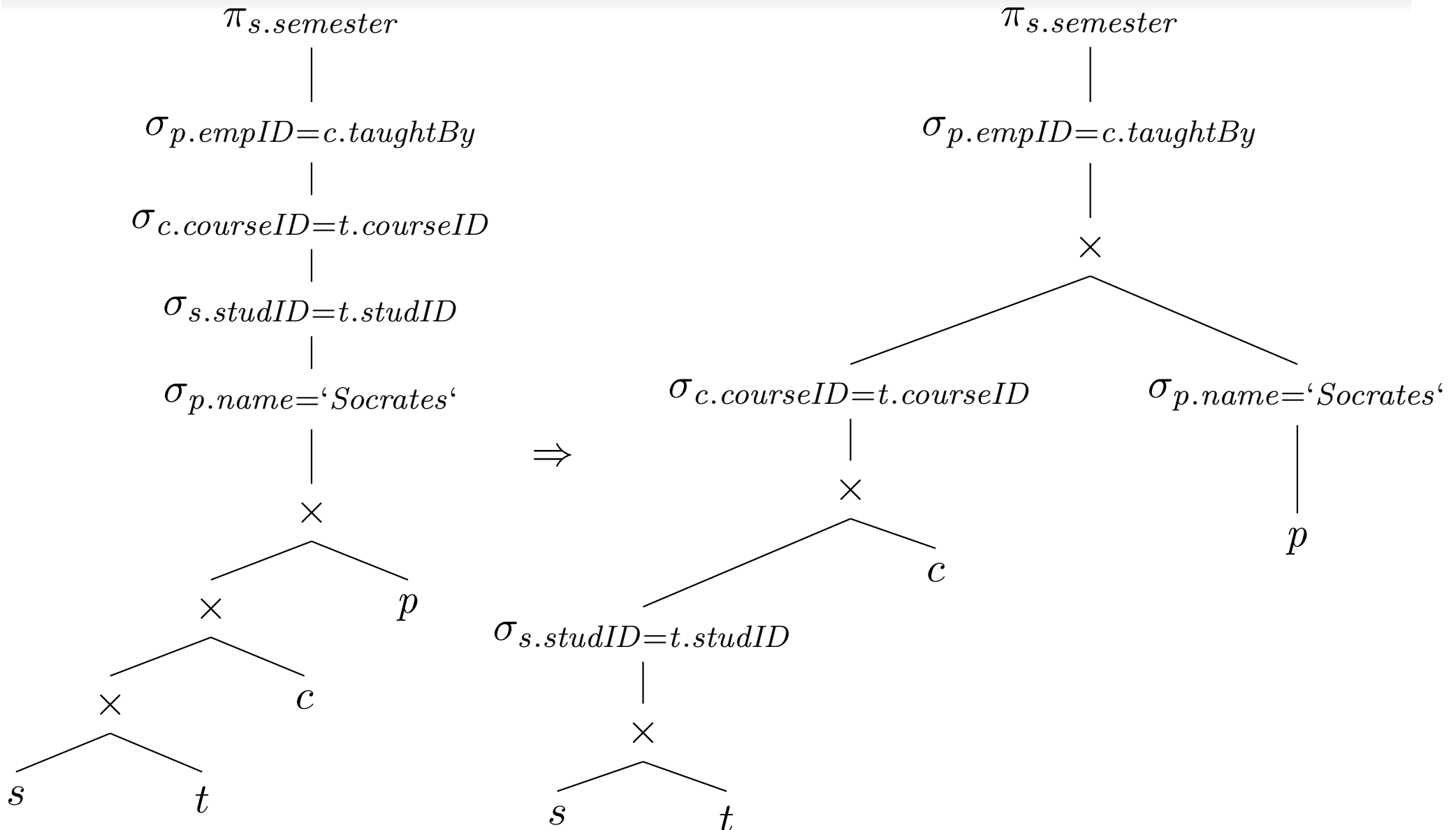
⇒



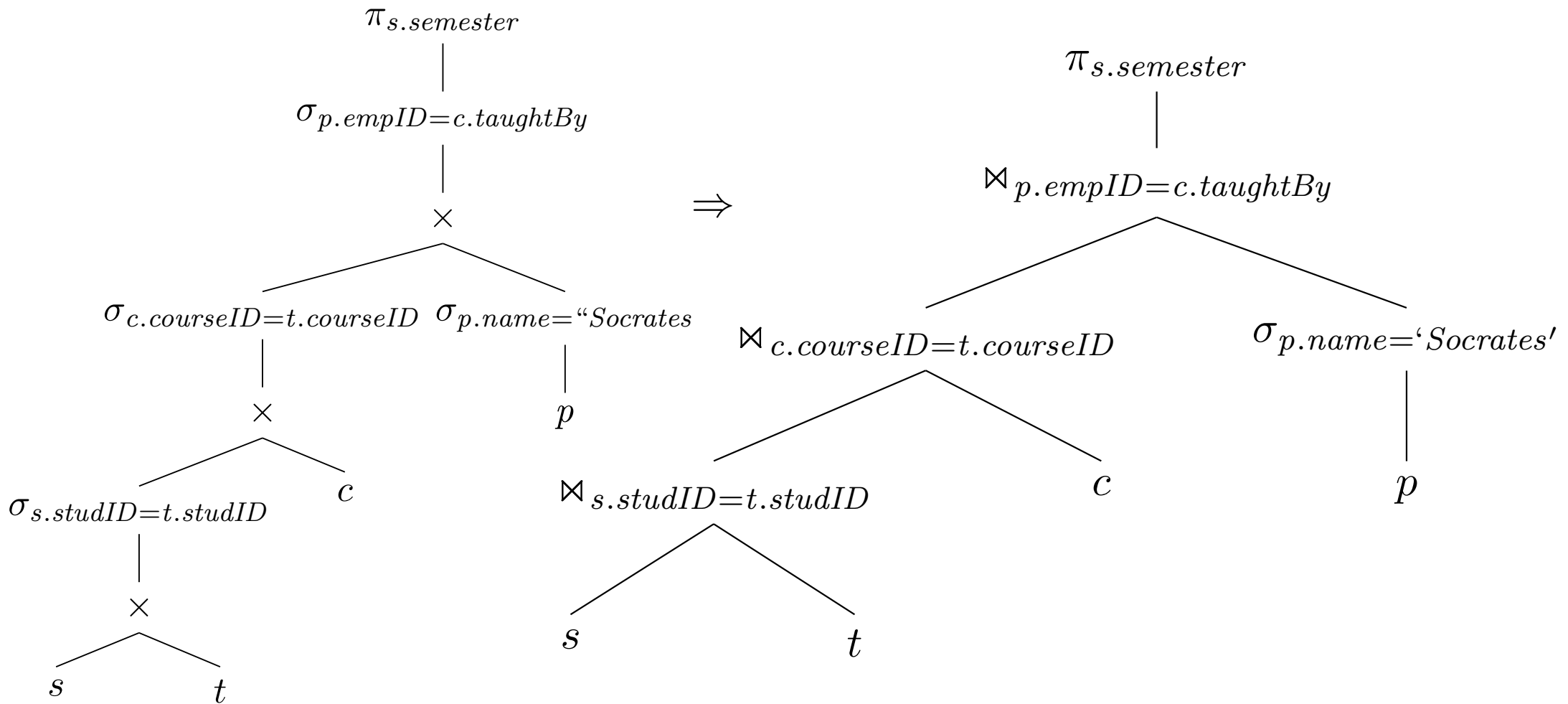
Aufbrechen von Selektionen



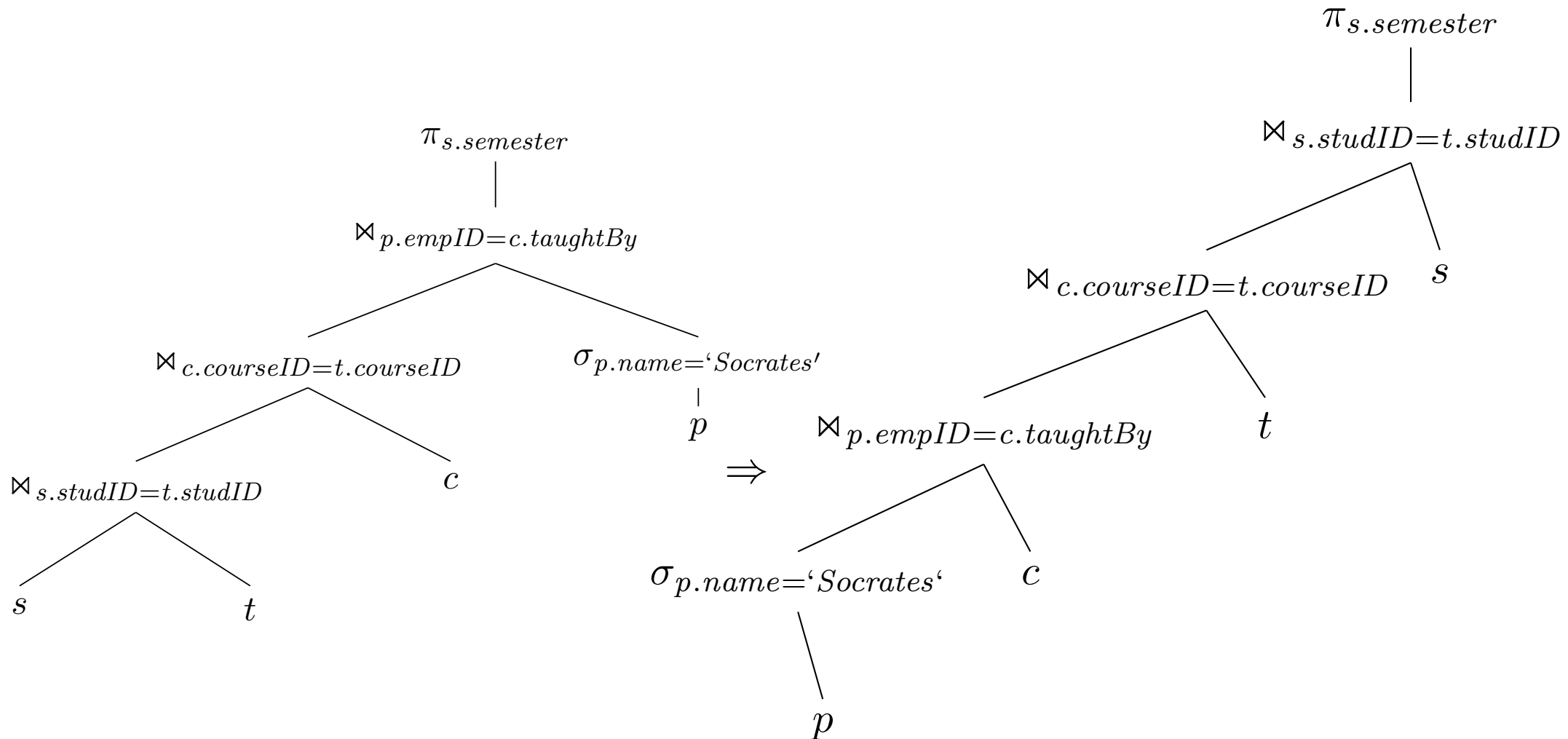
Verschieben von Selektionen



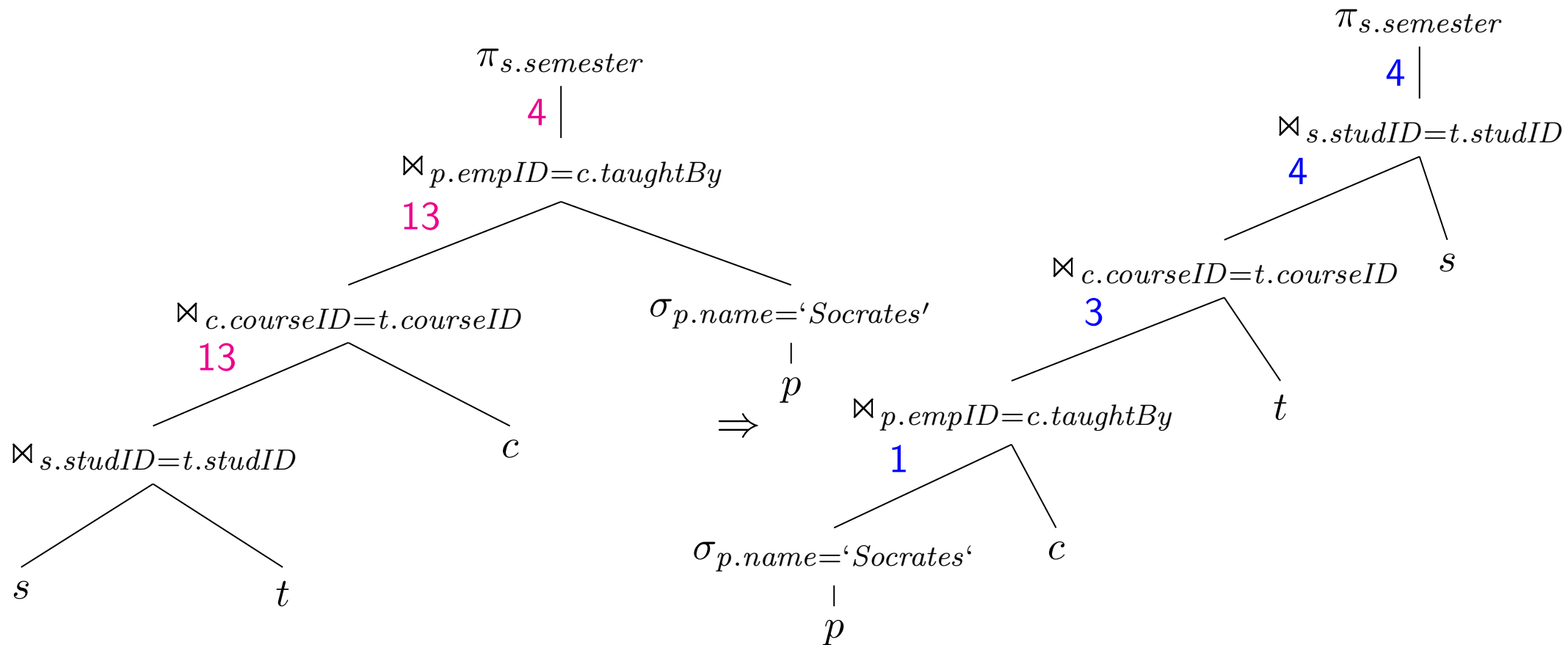
Joins einführen



Joinreihenfolge bestimmen

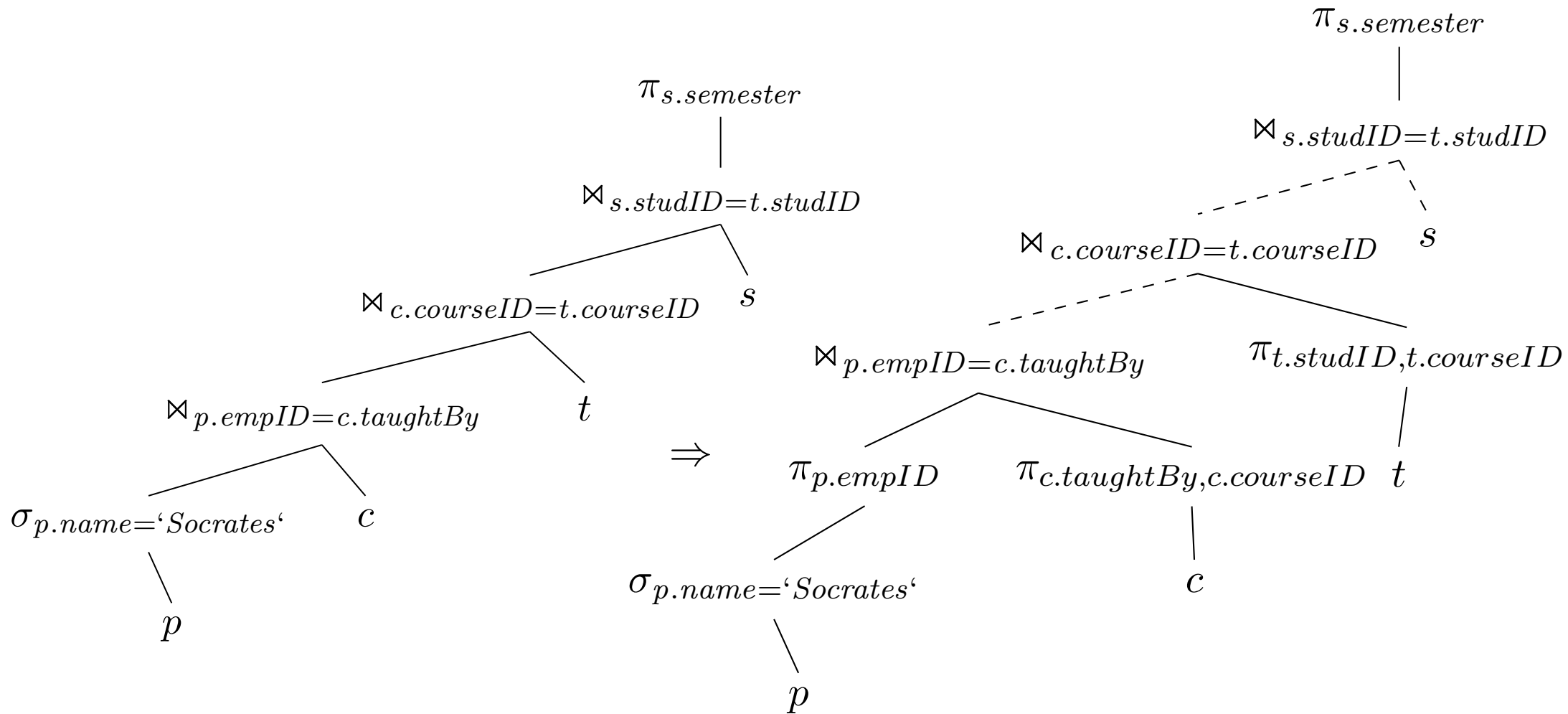


Effekt: Verkleinerung der Zwischenergebnisse



Schätzung der Zwischenergebnisgröße nur mit Statistiken möglich
 → Kostenmodelle

Einführen und verschieben von Projektionen



Vorsicht

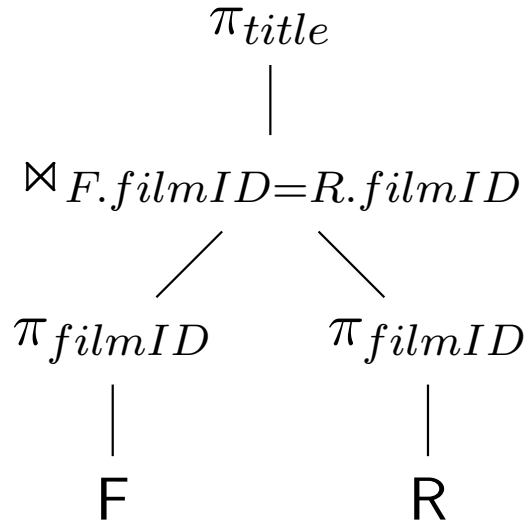
Finde die Titel von reservierten Filmen

```
SELECT DISTINCT title  
FROM film F, reserved R  
WHERE F.filmID = R.filmID
```

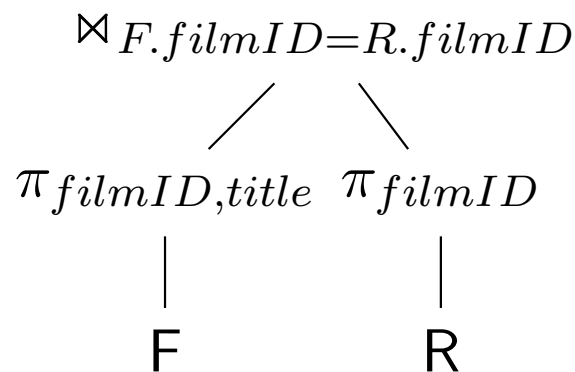
Vorsicht

Finde die Titel von reservierten Filmen

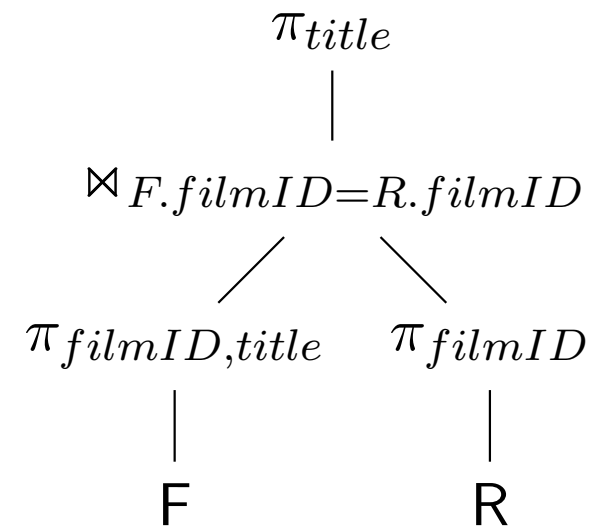
```
SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID
```



Zu viel Projektion



Zu wenig Projektion

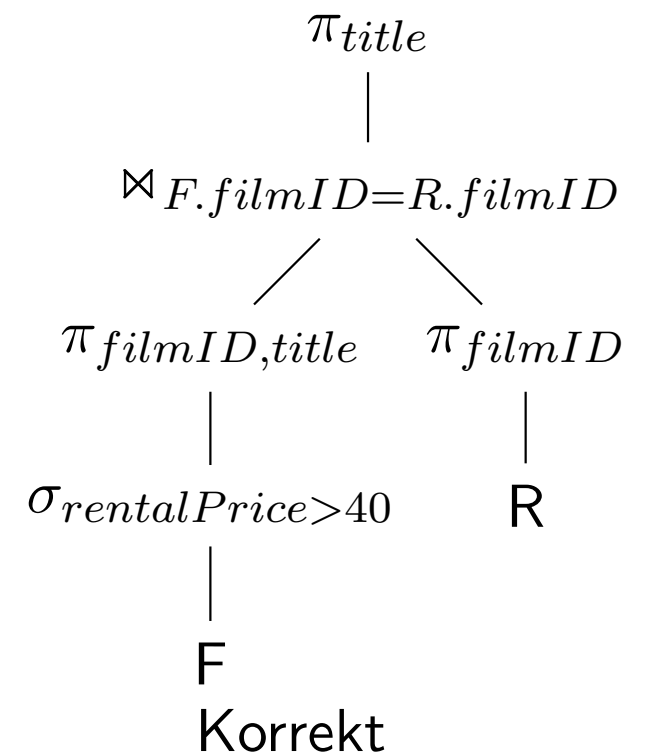
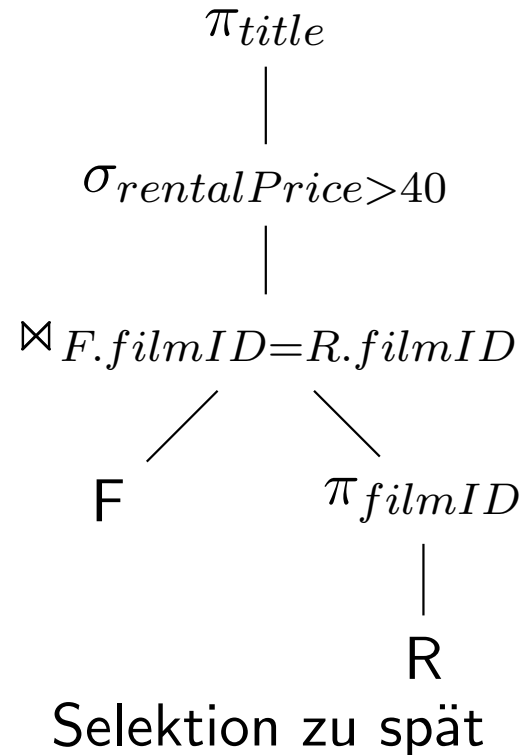
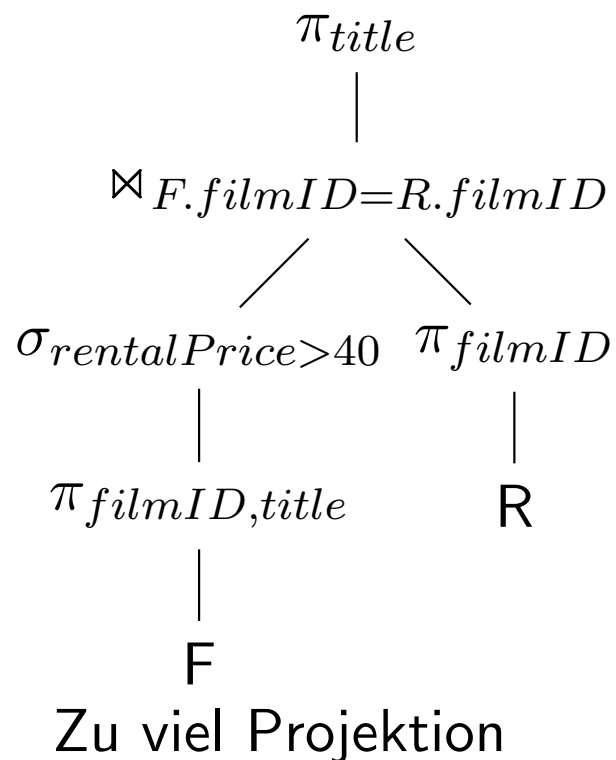


Korrekt

Vorsicht

Finde die Titel von teuren reservierten Filmen

```
SELECT DISTINCT title
FROM film F, reserved R
WHERE F.filmID = R.filmID AND F.rentalPrice > 40
```



Zusammenfassung: heuristische Anfrageoptimierung

Erfahrungsregeln

- Selektionen so früh wie möglich ausführen
- Projektionen so früh wie möglich ausführen

Optimierungsprozess

- Erstelle initialen Plan aus der SQL-Anfrage
- Modifiziere den Plan, um ihn einen effizienteren zu überführen

Hinweis

- Anfrageergebnisse werden durch einen einzigen Plan berechnet

- 3 Implementierung von Operationen
 - Selektion (Access Paths)
 - Join-Strategien

Beispieldatenbank

- customer (customerID, name, street, city, state)
- reserved (customerID, filmID, resDate)
- film (filmID, title, kind, rentalPrice)

Select-Anfragen

- Primary Key, Punkt

$$\sigma_{filmID=2}(film)$$

- Punkt

$$\sigma_{title='Terminator'}(film)$$

- Bereich

$$\sigma_{1 < rentalPrice < 4}(film)$$

- Konjunktion (logisches und)

$$\sigma_{kind='F' \wedge rentalPrice=4}(film)$$

- Disjunktion (logisches oder)

$$\sigma_{rentalPrice < 2 \vee kind='D'}(film)$$

Selektion

Hauptziel

Ersetze die Blattoperatoren im Anfrageplan durch konkrete Zugriffsmethoden.

Selektion – Punkt-/Bereichsanfragen

Linear Search

- Aufwändig, funktioniert aber immer

Binary Search

- Nur wenn die Datei entsprechend sortiert ist

Primary Hash Index

- Single Record Retrieval – funktioniert nicht für Bereichsanfragen

Primary/clustering Index

- Mehrere Records für jeden Wert
- Pointer zum Block mit dem ersten Record mit gesuchtem Wert

Secondary Index

- Jeder Record hat eigenen Pointer
- Kann teuer werden

Strategien für konjunktive Anfragen

```
SELECT *  
FROM customer  
WHERE name = 'Jensen' AND street = 'Elm'  
      AND state = 'Arizona'
```

- Kann ein Index auf (name) oder (street) benutzt werden? Ja
- Kann ein Index auf (name, street, state) benutzt werden? Ja
- Kann ein Index auf (name, street) benutzt werden? Ja
- Kann ein Index auf (name, street, city) benutzt werden? Ja
- Kann ein Index auf (city, name, street) benutzt werden? Nein

Optimierung von konjunktiven Anfragen

Indexe bieten gute Möglichkeiten, um die Performanz zu verbessern

Strategien für konjunktive Anfragen

- Existierende Indexe verwenden
 - Ideal: Es gibt einen passenden Composite Index
 - Falls es mehrere Indexe gibt
 - benutze den selektivsten Index, danach übrige Konditionen auswerten
- Überschneidung von Pointern ausnutzen (wenn mehrere Indexe anwendbar sind)
 - Index Lookups, um relevante Pointer zu erhalten
 - Überschneidung der Pointer ermitteln (Konjunktion der Bedingungen)
 - Die zugehörigen Records/Tupel lesen

Strategien für konjunktive Anfragen

- Existierende Indexe verwenden
 - Ideal: Es gibt einen passenden Composite Index
 - Falls es mehrere Indexe gibt
 - benutze den selektivsten Index, danach übrige Konditionen auswerten
- Überschneidung von Pointern ausnutzen (wenn mehrere Indexe anwendbar sind)
 - Index Lookups, um relevante Pointer zu erhalten
 - Überschneidung der Pointer ermitteln (Konjunktion der Bedingungen)
 - Die zugehörigen Records/Tupel lesen

Disjunktive Anfragen bieten wenig Gelegenheit zur Optimierung.

Tuning und das Anlegen von Indexen ist wichtig!

3 Implementierung von Operationen

- Selektion (Access Paths)
- Join-Strategien

Join-Algorithmen

Algorithmen

- Nested Loop Join
- Index-based Join
- Sort-Merge Join
- Hash Join

Strategien basieren auf Blöcken (nicht Tupel) als Basis

- Schätze I/Os (Block Retrievals)
- Benutze Buffer im Hauptspeicher

Tabellengröße und Join-Selektivität bestimmen die Kosten

- Selektivität einer Anfrage: $sel = \frac{\#tuples\ in\ result}{\#candidates}$
- Für einen Join, $\#candidates$ entspricht der Größe des Kartesischen Produkts

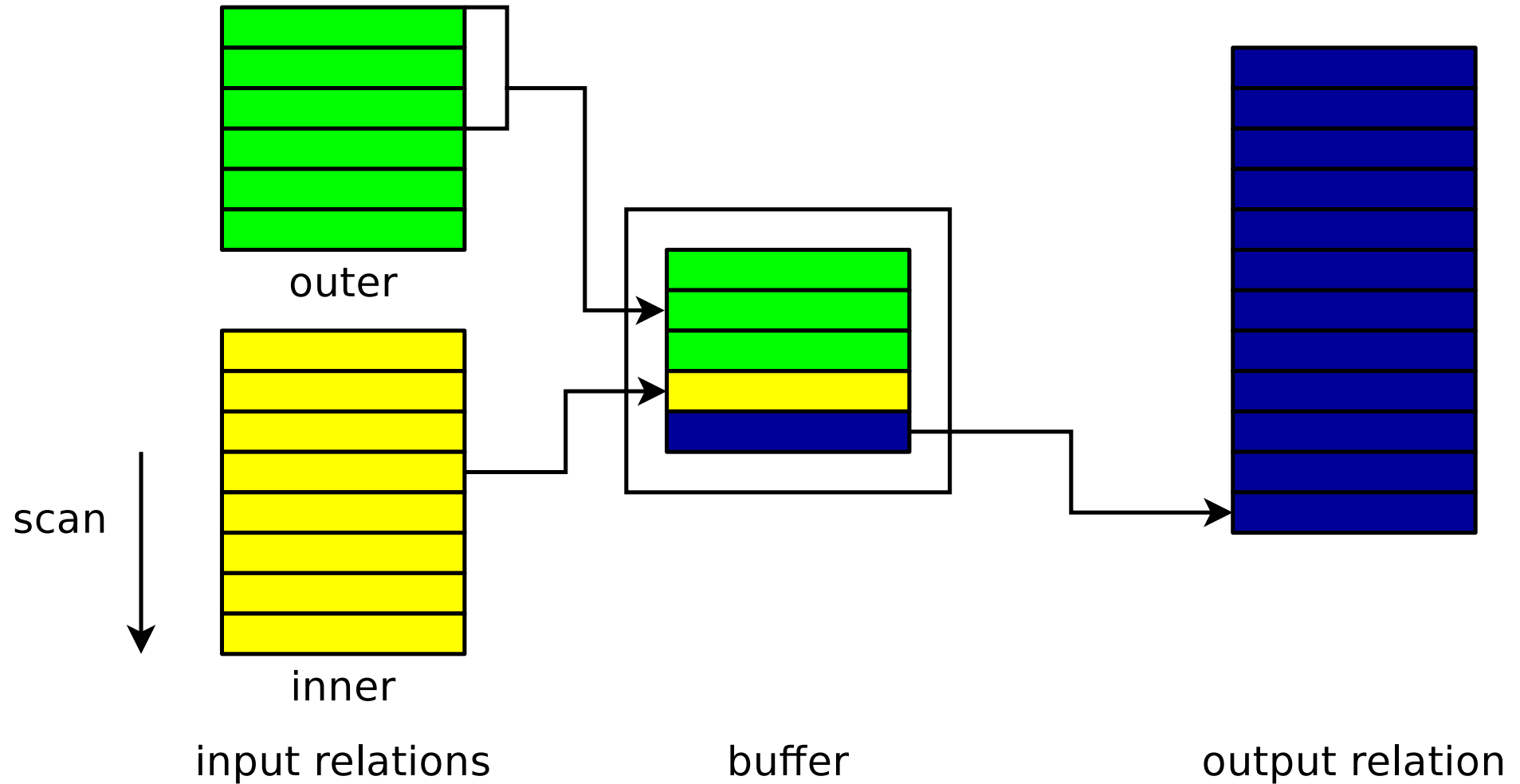
Nested Loop Join

ID	name		number	ID		ID	name	number
10	Jim		100	23		10	Jim	110
13	Joe		110	10		13	Joe	150
14	Sue		120	15		15	Pete	120
15	Pete		130	23		15	Pete	160
21	Dave		140	23		21	Dave	170
23	Anne		150	13		23	Anne	100
			160	15		23	Anne	130
			170	21		23	Anne	140

emp phone result

- Brute-force Strategie, teure Strategie, alle Tupel werden verglichen
- Kein Preprocessing der Input-Relationen
- Kein Index nötig, alle Joinbedingungen werden unterstützt

Nested Loop Join



Block Nested Loop Join

Nicht alle Blöcke beider Relationen passen gleichzeitig in den Hauptspeicher

repeat

lese $n_B - 2$ Blöcke der äußeren Relation

repeat

lese 1 Block der inneren Relation

Vergleiche enthaltene Tupel

until innere Relation ist vollständig durchlaufen

until äußere Relation ist vollständig durchlaufen

Parameter

- b_{inner}, b_{outer} : Anzahl von Blöcken
- n_B : Größe des Buffers im Hauptspeicher

Kostenschätzung (Block Transfers)

$$b_{outer} + (\lceil b_{outer} / (n_B - 2) \rceil) \cdot b_{inner}$$

Wenn wir weitere Systemparameter (Block-Transfer, Disk Seek, CPU Speed, ...) und die Größen der Relationen kennen, können wir die Berechnungszeit abschätzen.

Block Nested Loop Join

Beispiel (*reserved* ⋈ *customer*)

- Anzahl der Blöcke

$$b_{reserved} = 2.000, b_{customer} = 10$$

- Größe des Buffers im Hauptspeicher

$$n_B = 6$$

- Kostenschätzung (Block Transfers)

$$b_{outer} + (\lceil b_{outer} / (n_B - 2) \rceil) \cdot b_{inner}$$

Block Nested Loop Join

Example (*reserved* ⋈ *customer*)

- Anzahl von Blöcken
 $b_{reserved} = 2.000, b_{customer} = 10$
- Größe des Buffers im Hauptspeicher
 $n_B = 6$
- Kostenschätzung (Block Transfers)
 $b_{outer} + (\lceil b_{outer} / (n_B - 2) \rceil) \cdot b_{inner}$

Kosten

- reserved als äußere Relation
 $2.000 + \lceil (2.000/4) \rceil \cdot 10 = 7.000$
- customer als äußere Relation
 $10 + \lceil (10/4) \rceil \cdot 2.000 = 6.010$

Index-based Nested Loop Join

Gleiches Prinzip wie beim standard Nested Loop Join

- Äußere Relation
- Innere Relation
- Index Lookups können File Scan der inneren Relation ersetzen

Merge Join

Ausnutzen der sortierten Reihenfolge

R				S	
	A			B	
...	0	←	→	5	...
...	7			6	...
...	7			7	...
...	8			8	...
...	8			8	...
...	10			11	...
...

Annahme:

Beide Input-Relationen sind sortiert

Merge Join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
110	10
150	13
120	15
160	15
170	21
100	23
130	23
140	23

phone

=

ID	name	number
10	Jim	110
13	Joe	150
15	Pete	120
15	Pete	160
21	Dave	170
23	Anne	100
23	Anne	130
23	Anne	140

result

Merge Join – Kosten

Parameter

- b_1, b_2 : Anzahl der Blöcke

Kostenschätzung (Block Transfers)

$$b_1 + b_2$$

Erweiterungen

- Kombination mit Sortierung, wenn Input-Relationen nicht sortiert vorliegen
- Nicht genügend Hauptspeicher

Hash Join

ID	name
10	Jim
13	Joe
14	Sue
15	Pete
21	Dave
23	Anne

emp

⋈

number	ID
100	23
110	10
120	15
130	23
140	23
150	13
160	15
170	21

phone

Wende Hashfunktion auf die Join-Attribute an
→ Partitioniere Tupel in Buckets

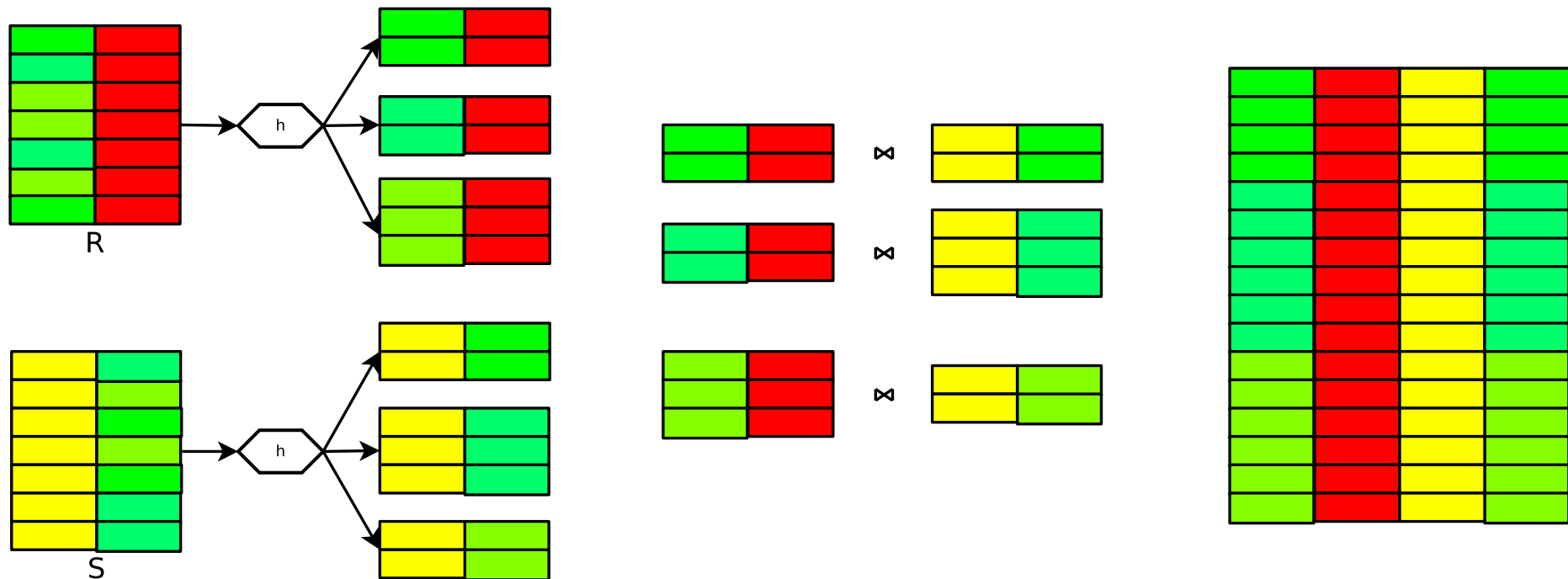
Hash Join

<table><tr><th>ID</th><th>name</th></tr><tr><td>15</td><td>Pete</td></tr><tr><td>21</td><td>Dave</td></tr></table> <p>emp₀</p>	ID	name	15	Pete	21	Dave	⋈	<table><tr><th>number</th><th>ID</th></tr><tr><td>120</td><td>15</td></tr><tr><td>160</td><td>15</td></tr><tr><td>170</td><td>21</td></tr></table> <p>phone₀</p>	number	ID	120	15	160	15	170	21	=	<table><tr><th>ID</th><th>name</th><th>number</th></tr><tr><td>15</td><td>Pete</td><td>120</td></tr><tr><td>15</td><td>Pete</td><td>160</td></tr><tr><td>21</td><td>Dave</td><td>170</td></tr></table> <p>result₀</p>	ID	name	number	15	Pete	120	15	Pete	160	21	Dave	170
ID	name																													
15	Pete																													
21	Dave																													
number	ID																													
120	15																													
160	15																													
170	21																													
ID	name	number																												
15	Pete	120																												
15	Pete	160																												
21	Dave	170																												
<table><tr><th>ID</th><th>name</th></tr><tr><td>10</td><td>Jim</td></tr><tr><td>13</td><td>Joe</td></tr></table> <p>emp₁</p>	ID	name	10	Jim	13	Joe	⋈	<table><tr><th>number</th><th>ID</th></tr><tr><td>110</td><td>10</td></tr><tr><td>150</td><td>13</td></tr></table> <p>phone₁</p>	number	ID	110	10	150	13	=	<table><tr><th>ID</th><th>name</th><th>number</th></tr><tr><td>10</td><td>Jim</td><td>110</td></tr><tr><td>13</td><td>Joe</td><td>150</td></tr></table> <p>result₁</p>	ID	name	number	10	Jim	110	13	Joe	150					
ID	name																													
10	Jim																													
13	Joe																													
number	ID																													
110	10																													
150	13																													
ID	name	number																												
10	Jim	110																												
13	Joe	150																												
<table><tr><th>ID</th><th>name</th></tr><tr><td>14</td><td>Sue</td></tr><tr><td>23</td><td>Anne</td></tr></table> <p>emp₂</p>	ID	name	14	Sue	23	Anne	⋈	<table><tr><th>number</th><th>ID</th></tr><tr><td>100</td><td>23</td></tr><tr><td>130</td><td>23</td></tr><tr><td>140</td><td>23</td></tr></table> <p>phone₂</p>	number	ID	100	23	130	23	140	23	=	<table><tr><th>ID</th><th>name</th><th>number</th></tr><tr><td>23</td><td>Anne</td><td>100</td></tr><tr><td>23</td><td>Anne</td><td>130</td></tr><tr><td>23</td><td>Anne</td><td>140</td></tr></table> <p>result₂</p>	ID	name	number	23	Anne	100	23	Anne	130	23	Anne	140
ID	name																													
14	Sue																													
23	Anne																													
number	ID																													
100	23																													
130	23																													
140	23																													
ID	name	number																												
23	Anne	100																												
23	Anne	130																												
23	Anne	140																												

$$\text{result} = \text{result}_0 \cup \text{result}_1 \cup \text{result}_2$$

Hash Join

- Jede Relation mit Hashfunktion partitionieren
- Jedes Bucket muss klein genug sein, um in den Hauptspeicher zu passen
- Joine die „passenden“ Buckets miteinander



Input Relations

Output Relation

Hash Join

Parameter

- b_1, b_2 : Anzahl Blöcke der Relationen R_1 und R_2

Schritte

- Partitioniere Relation R_1 mit h_1 in Buckets r_{1_i} (read all / write all)
 $2 \times b_1$
- Partitioniere Relation R_2 mit h_1 in Buckets r_{2_i} (read all / write all)
 $2 \times b_2$
- Build-Phase: benutze h_2 zur Erstellung eines in-memory Hash Indexes für Bucket r_{1_i} (read all)
 b_1
- Probe-Phase: für jedes r_{2_i} , benutze h_2 , um Joinpartner im in-memory Index zu finden (read all)
 b_2

Kostenschätzung (Block Transfers)

$$3 \times b_1 + 3 \times b_2 + \epsilon \quad (\text{unvollständig gefüllte Blöcke})$$

Kosten und Anwendung von Join-Algorithmen

Nested Loop Join

- Kann für alle Join-Typen verwendet werden
- Kann sehr teuer werden

Merge Join

- Dateien müssen auf Joinattributen sortiert sein
Sortierung kann für den Join vorgenommen werden
- Kann Indexe verwenden

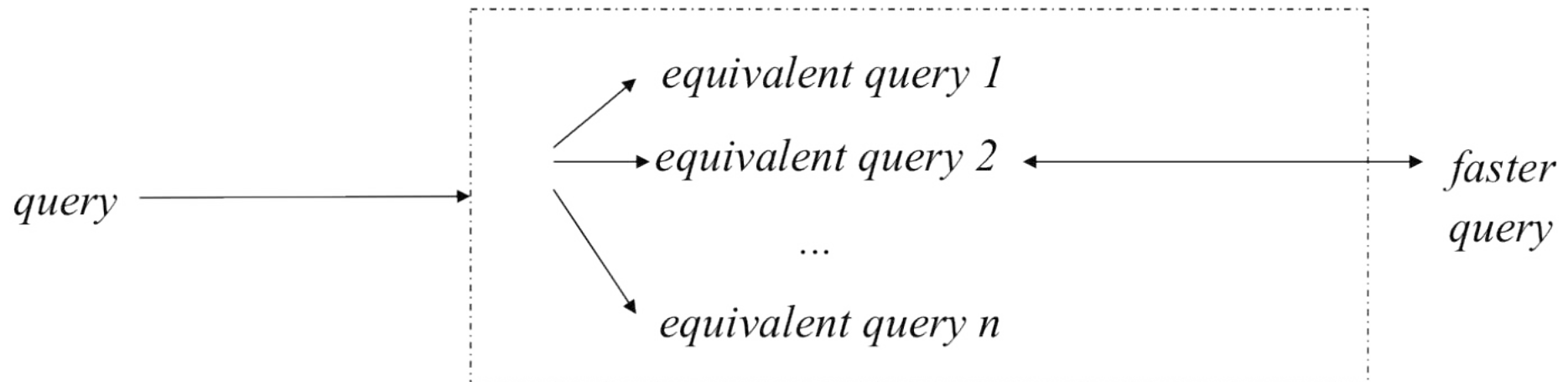
Hash Join

- Gute Hashfunktionen sind die Grundlage
- Beste Performanz, wenn die kleinere Relation in den Hauptspeicher passt

- 4 Kostenbasierte (physische) Anfrageoptimierung
 - Selektivität und Kardinalität
 - Kostenschätzung
 - PostgreSQL

Ziel

Für eine gegebene Anfrage, finde den besten Ausführungsplan



Optimierung

- Heuristische (logische) Optimierung
 - Anfrageplan (relationale Algebra)
- Kostenbasierte (physische) Optimierung
 - Anfrageausführungsplan (konkrete Algorithmen und Zugriffspfade)

Physische Anfrageoptimierung

Physische Anfrageoptimierung

- Erstelle alternative Ausführungspläne
- Wähle Algorithmen und Zugriffspfade (Access Paths)
- Berechne Kosten
- Wähle den günstigsten Ausführungsplan

Voraussetzung

- Kostenmodell
- Statistiken für die Input-Relationen
 - Statistiken für Blattrelationen: gespeichert im Systemkatalog (System Catalog)
 - Statistiken für Zwischenergebnisse müssen geschätzt werden (Kardinalität)

- 4 Kostenbasierte (physische) Anfrageoptimierung
 - Selektivität und Kardinalität
 - Kostenschätzung
 - PostgreSQL

Statistiken pro Relation

Für Relation r

- Anzahl Tupel (Records): n_r
- Größe der Tupel in Relation r : l_r
- Füllgrad (Load/Fill Factor), prozentuale Nutzung des Platzes je Block
- Blocking Factor (Anzahl Tupel pro Block)
- Größe der Relation in Blöcken: b_r
- Organisation der Relation
Heap, Hash, Indexes, Sortierung
- Anzahl Overflow-Buckets

Statistiken pro Attribut

Für Attribut A in Relation r

- Größe und Typ
- Anzahl „distinct“ Werte für Attribut A : $V(A, r)$
Entspricht $\pi_A(r)$
- Kardinalität einer Selektion $S(A, r)$
Entspricht $\sigma_{A=a}(r)$ für beliebigen Wert a
- Wahrscheinlichkeitsverteilung der Werte
Alternativ: Gleichverteilung annehmen

Statistiken müssen aktualisiert werden, wenn eine Tabelle aktualisiert wird!

Statistiken pro Index

- Baisrelation
- Indexierte Attribute
- Organisation, e.g., B⁺-Baum, Hash
- Clustering Index?
- Schlüsselattribut(e)?
- Sparse oder dense?
- Anzahl der Ebenen
- Anzahl Blätter

4 Kostenbasierte (physische) Anfrageoptimierung

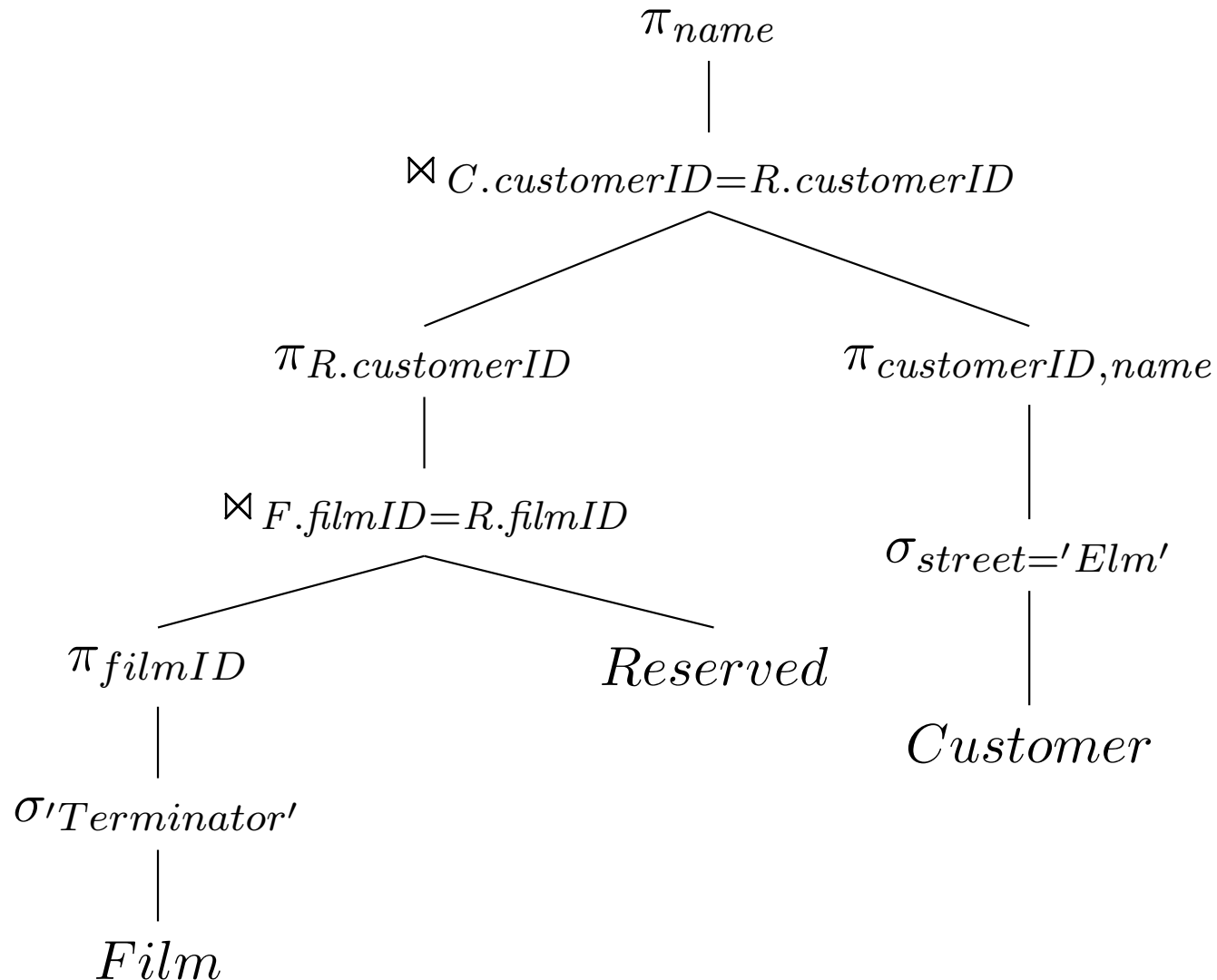
- Selektivität und Kardinalität
- Kostenschätzung
- PostgreSQL

Kostenschätzungsbeispiel

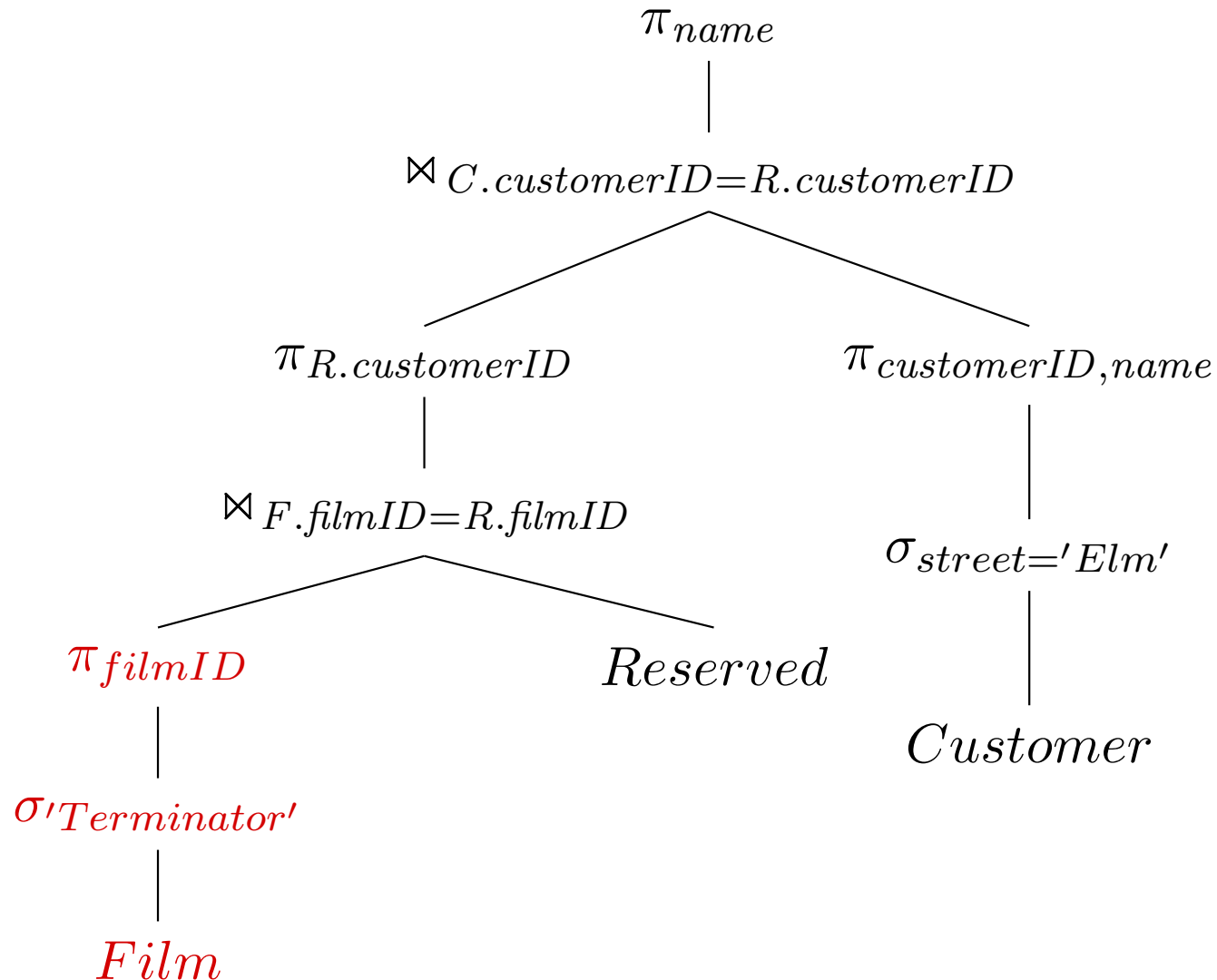
Welche Kunden wohnen in der Elm Street und haben den Film „Terminator“ reserviert?

```
SELECT name  
FROM customer C, reserved R, Film F  
WHERE title = 'Terminator' AND F.filmID = R.filmID  
AND C.customerID = R.customerID AND C.street = 'Elm';
```

Kostenschätzungsbeispiel



Kostenschätzungsbeispiel



Kostenschätzungsbeispiel

$$\pi_{filmID}(\sigma_{title='Terminator'}(Film))$$

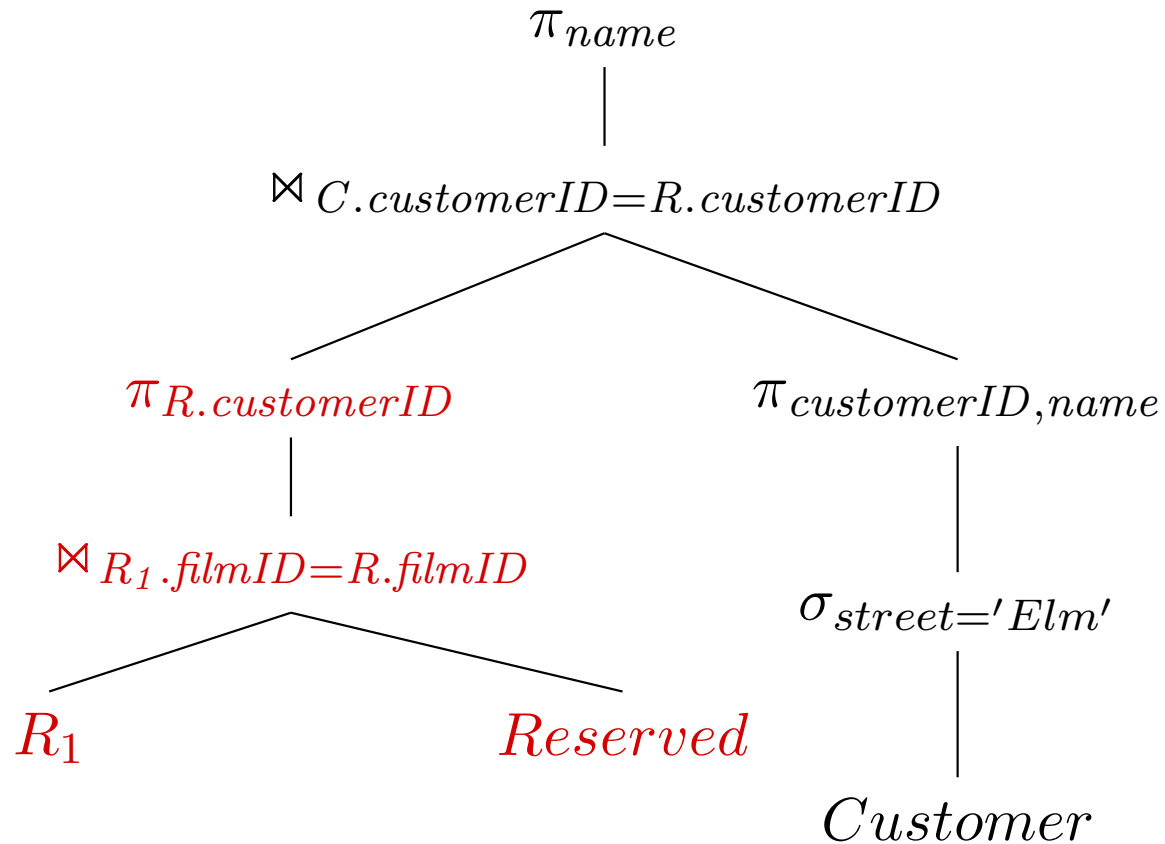
Statistiken

- Statistiken für Relation
 - Anzahl Tupel: $n_{Film} = 5000$
 - Größe der Relation in Blöcken: $b_{Film} = 50$
- Statistiken für Attribute
 - Kardinalität der Selection: $S(title, Film) = 1$
- Statistiken für Index
 - Hash Index für Attribut „title“

Ausführung

- Benutze Index mit „Terminator“ $costs_{disk\ access} = 1$
- Projektion auf filmID Ergebnisgröße: 1 Tupel
- Lasse Ergebnis im Hauptspeicher (1 Block)

Kostenschätzungsbeispiel



Kostenschätzungsbeispiel

$$\pi_{R.customerID}(R_1 \bowtie_{R_1.filmID=R.filmID} Reserved)$$

Statistics

- Statistiken für Relation
 - Anzahl Tupel: $n_{Reserved} = 40000$
 - Größe der Relation in Blöcken: $b_{Film} = 2000$
- Statistiken für Attribute
 - Kardinalität der Selection: $S(filmID, Reserved) = 8$
- Statistiken für Index
 - Primary B⁺-Baum Index für Reserved auf filmID mit 3 Ebenen

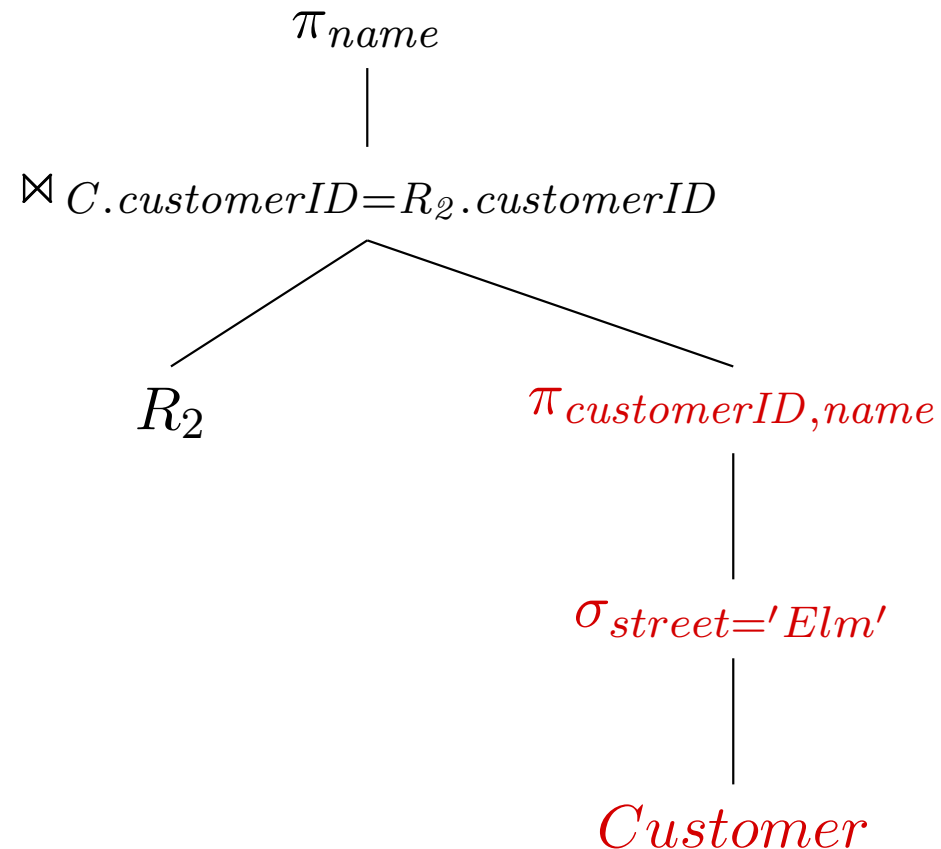
Ausführung

- Index Join unter Verwendung des B⁺-Baums
- Projektion auf customerID
- Lass Ergebnis im Hauptspeicher (1 Block)

$costs_{disk\ access} = 2$
(Letzte Indexebene auf Platte, 1 Tupel Lookup)

Ergebnisgröße: 8 Tupel

Kostenschätzungsbeispiel



Kostenschätzungsbeispiel

$$\pi_{customerID,name}(\sigma_{street='Elm'}(Customer))$$

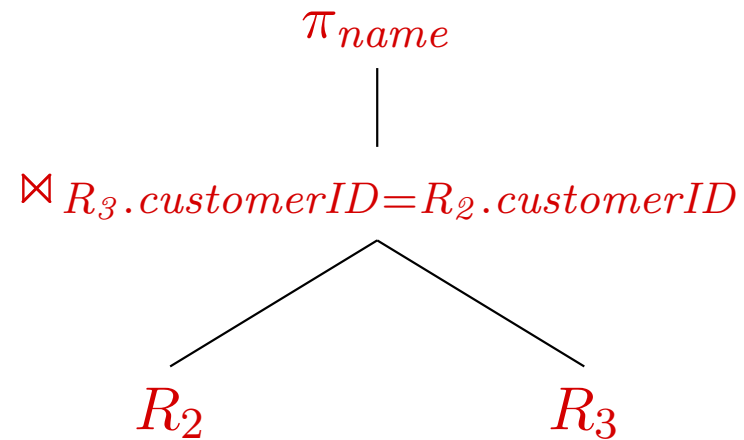
Statistiken

- Statistiken für Relation
 - Anzahl Tupel: $n_{Customer} = 200$
 - Größe der Relation in Blöcken: $b_{Customer} = 10$
- Statistiken für Attribute
 - Kardinalität der Selection: $S(street, Customer) = 10$
- Statistiken für Index
 - Kein Index auf „street“

Ausführung

- Lineare Suche in Customer $costs_{\text{disk access}} = 10$
- Projektion auf customerID, name Ergebnisgröße: 10 Tupel
- Lass Ergebnis im Hauptspeicher (1 Block)

Kostenschätzungsbeispiel



Kostenschätzungsbeispiel

$$\pi_{name}(R_2 \bowtie_{R_3.customerID=R_2.customerID} R_3)$$

Ausführung

- Join im Hauptspeicher

Total Costs

$$costs_{\text{disk access}} = 1 + 2 + 10 + 0 = 14$$

Kostenmodell

Kostenmodelle erfassen mehr als nur Festplattenzugriffe

- CPU Time
- Communication Time
- Main Memory Usage
- ...

Größen von Input/Output für jede Operation abschätzen

- Statistiken für Relationen: im System Catalog gespeichert
- Statistiken für Zwischenergebnisse müssen geschätzt werden (Kardinalitäten)

Weitere Aspekte

- Suchraum aufspannen (Dynamic Programming, Exhaustive Search,...)
- Bushy vs. Left-Deep Join-Trees (Parallelität vs. Pipelining)
- Multiquery-Optimization (Shared Scans,...)
- ...

Heuristische vs. kostenbasierte Anfrageoptimierung

Heuristisch

- Kann immer verwendet werden
- Anfragepläne werden sequenziell generiert
- Jeder Plan ist (wahrscheinlich) effizienter als der vorherige
- Suche ist linear

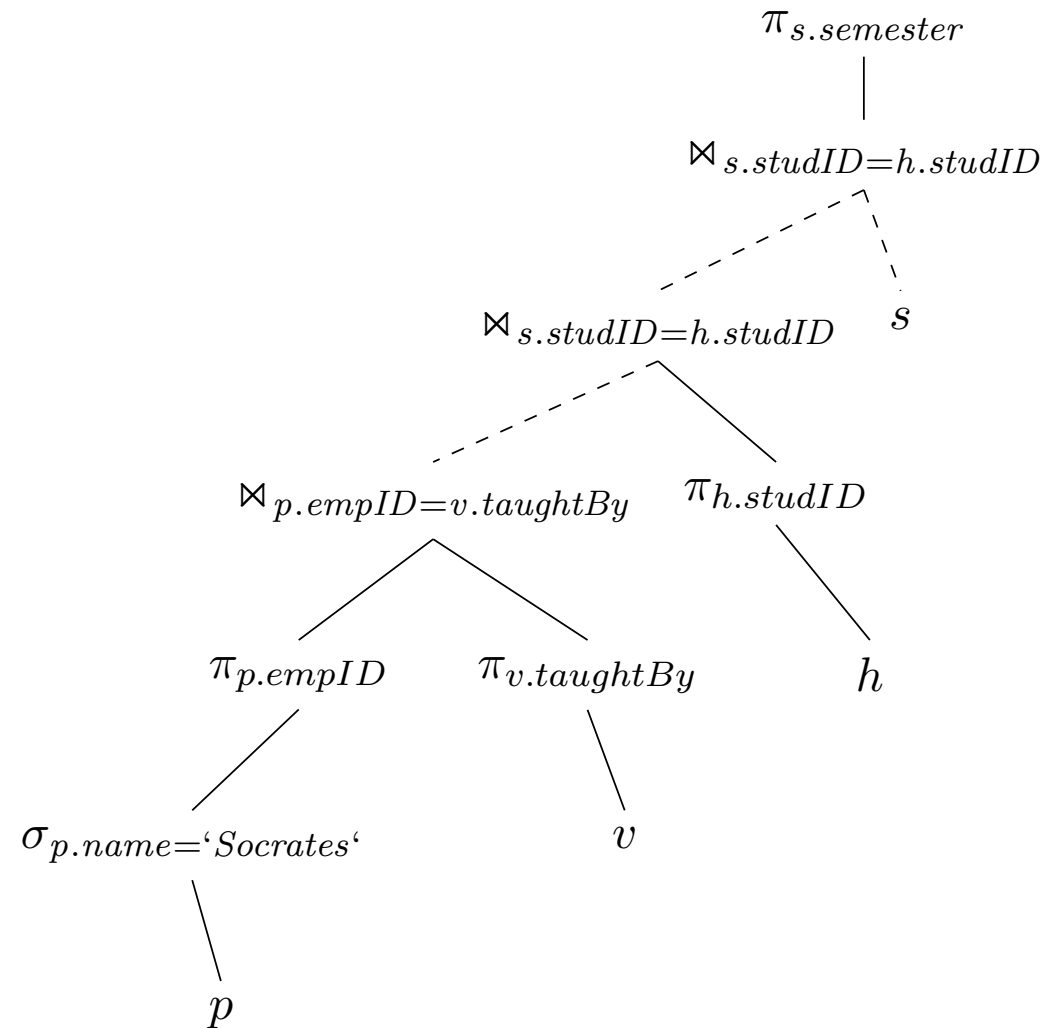
Kostenbasiert

- Kann nur verwendet werden, wenn Statistiken vorliegen
- Mehrere Pläne werden generiert
- Kosten für jeden Plan berechnen, den günstigsten wählen
- Suche ist mehrdimensional

- 4 Kostenbasierte (physische) Anfrageoptimierung
 - Selektivität und Kardinalität
 - Kostenschätzung
 - PostgreSQL

PostgreSQL

SELECT DISTINCT s.semester
 FROM student s, takes h,
 course v, professor p
 WHERE p.name='Socrates' AND
 v.taughtBy = p.empID AND
 v.courseID = h.courseID AND
 h.studID = s.studID;



PostgreSQL EXPLAIN

```
EXPLAIN SELECT DISTINCT s.semester  
FROM student s, takes h,  
      course v, professor p  
WHERE p.name='Socrates' AND  
      v.taughtBy = p.empID AND  
      v.courseID = h.courseID AND  
      h.studID = s.studID;
```

EXPLAIN

Zeige den Ausführungsplan, den PostgreSQL für das Statement generiert

PostgreSQL EXPLAIN

QUERY PLAN

text

Unique (cost=4.61..4.62 rows=2 width=4)
-> Sort (cost=4.61..4.62 rows=2 width=4)
Sort Key: s.semester
-> Hash Join (cost=3.47..4.60 rows=2 width=4)
Hash Cond: (s.studid = h.studid)
-> Seq Scan on student s (cost=0.00..1.08 rows=8 width=8)
-> Hash (cost=3.45..3.45 rows=2 width=4)
-> Hash Join (cost=2.26..3.45 rows=2 width=4)
Hash Cond: (h.courseid = v.courseid)
-> Seq Scan on takes h (cost=0.00..1.13 rows=13 width=8)
-> Hash (cost=2.25..2.25 rows=1 width=4)
-> Hash Join (cost=1.10..2.25 rows=1 width=4)
Hash Cond: (v.taughtby = p.empid)
-> Seq Scan on course v (cost=0.00..1.10 rows=10 width=8)
-> Hash (cost=1.09..1.09 rows=1 width=4)
-> Seq Scan on professor p (cost=0.00..1.09 rows=1 width=4)
Filter: ((name)::text = 'Socrates'::text)

PostgreSQL EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT DISTINCT s.semester  
FROM student s, takes h,  
      course v, professor p  
WHERE p.name='Socrates' AND  
      v.taughtBy = p.empID AND  
      v.courseID = h.courseID AND  
      h.studID = s.studID;
```

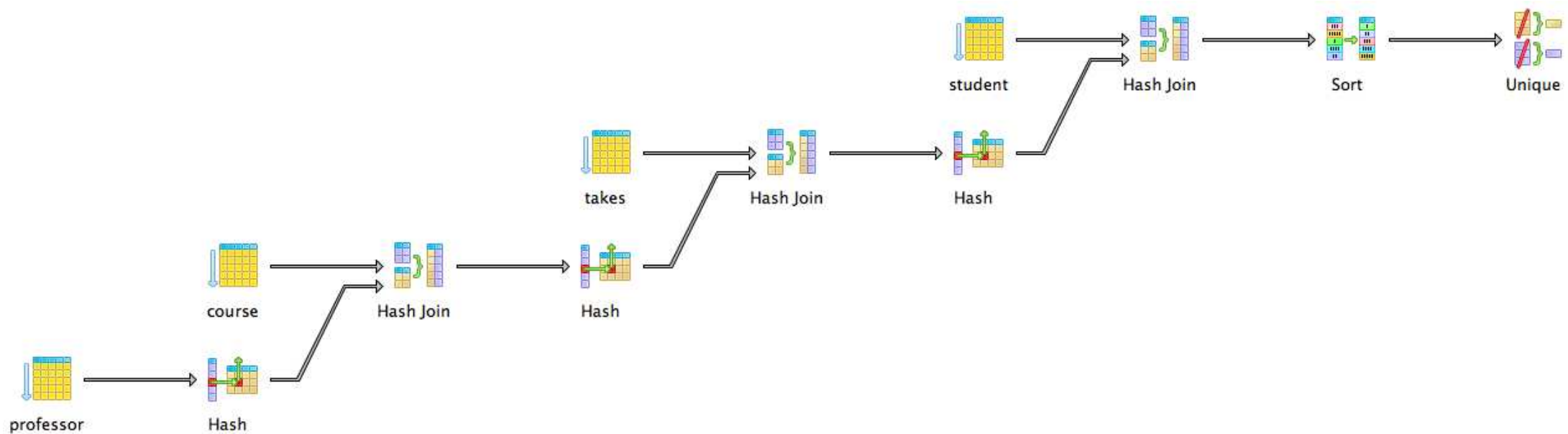
EXPLAIN ANALYZE

Die zusätzliche ANALYZE-Option sorgt dafür, dass die Anfrage zusätzlich ausgeführt wird.

ANALYZE

ANALYZE sammelt Statistiken zu den Inhalten der Tabellen.

PostgreSQL EXPLAIN ANALYZE



Sequential Scans vs. indexe

Ob ein Index „nützlich“ ist oder nicht hängt ab von

- Wie viele Daten sind für die Anfrage relevant
- Größe der Relation
- Eigenschaften des Index (clustered, multiple columns,...)
- Welche Algorithmen benötigen die Daten als Input
- ...

Bis die Anfrageoptimierung perfektioniert wird, ist Tuning die Hauptaufgabe eines Datenbankadministrators (Indexe erstellen, etc.).

Zusammenfassung

- Anfrageoptimierung ist eine Kernkomponente von relationalen DBMS
- Heuristische Optimierung kann immer verwendet werden, kann aber zu suboptimalen Plänen führen
- Kostenbasierte Optimierung ist auf Statistiken angewiesen
- Datenbanksysteme bieten Informationen zu Anfrageausführung an (EXPLAIN)
- Datenbankadministratoren müssen stetig über Verbesserungen nachdenken (z.B. Indexe)