Anfrageoptimierung (Teil 2)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

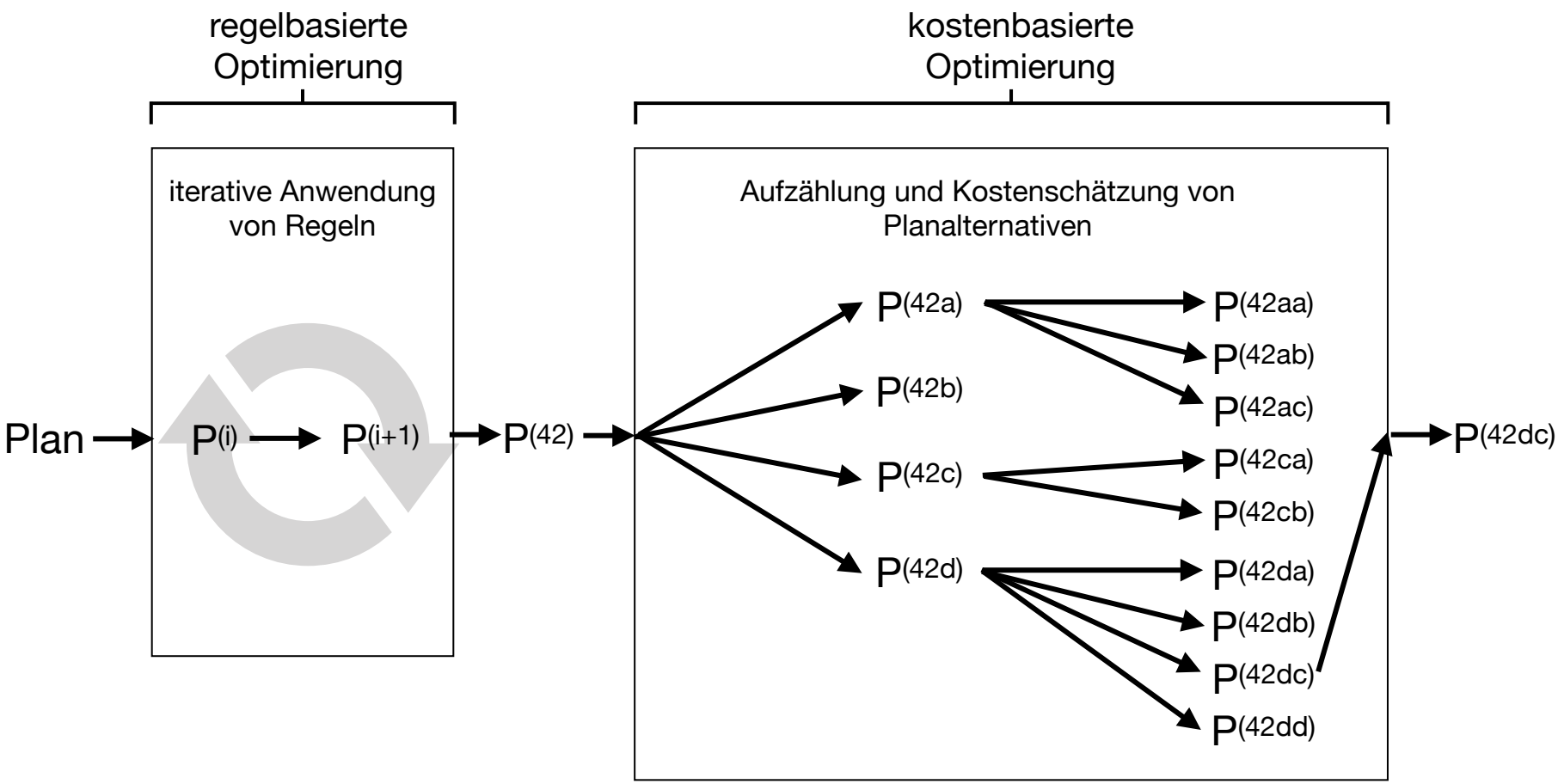
9. Oktober 2020

Kostenbasierte Optimierung (Fortsetzung)

3. transformierter logischer Plan
   ↓ kostenbasierte Optimierung
4. physischer Plan

- (gute) Anfrageoptimierer zählen eine große Zahl möglicher Pläne auf und versuchen, die Laufzeiten dieser Pläne mit Hilfe von Kostenmodellen zu schätzen
- nur der Plan mit der erwarteten kürzesten Laufzeit wird dann ausgeführt

Unterschied zu regelbasierter Optimierung (letzte Woche)



- in **beiden** Komponenten werden Pläne mit Regeln erstellt
- Unterschied: Anwendung von Regeln **ohne oder mit Kostenschätzung**

Verschiedene Dimensionen kostenbasierter Optimierung

Entscheidungen auf **logischer** Ebene:

1. Welche Joinreihenfolge nehmen?
   Beispiele: Welche Joinreihenfolge hat geringere Laufzeit: $R \bowtie (S \bowtie T)$ oder $(R \bowtie S) \bowtie T$ oder ...?

Entscheidungen auf **physischer** Ebene:

1. Welchen physischen Operator nutzen?
   Beispiele: Hash-basierter Join, sortierbasierter Join oder XY Join? (vgl. Logische vs physische Operatoren-Diskussion)
2. Index benutzen oder nicht? Falls ja, welchen Index benutzen?
   Beispiele: Relation Scannen oder Binäre Suche nutzen? Welche Art von Index nutzen? hash-basiert, baumbasiert oder ... ?
   (vgl. Picasso-Notebook)
3. Welche Ressourcen für welchen Teilplan nutzen?
   Beispiele: Wieviele Threads wo einsetzen? Welcher Teil des Plans bekommt wie viel Rechenzeit/Hauptspeicher?

zu 3. Joinreihenfolge: Baumstruktur des Plans

Die Dimension „Joinreihenfolge“ besteht aus unterschiedlichen Teilproblemen:

3.1 Baumstruktur des Plans

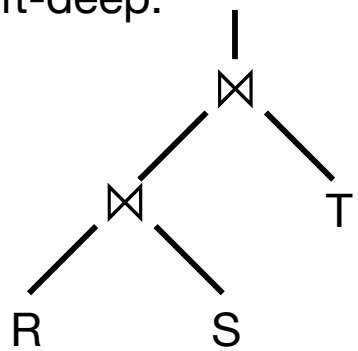
Für n Eingaberelationen gibt es C_n viel Binärbäume mit n Blattknoten.

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \sim \frac{4^n}{n^{1.5}\sqrt{\pi}}, n \geq 0 \qquad \text{(Catalan-Zahl)}$$

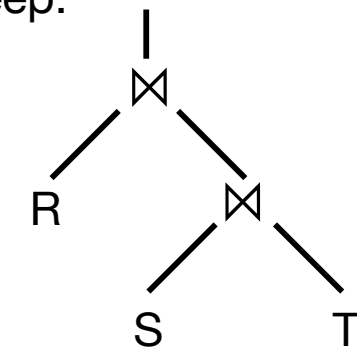
Die ersten (relevanten) Catalan-Zahlen für $C_3, C_4, \ldots$ sind 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, ...

Drei Eingaberelationen: $C_3 = 2$ Pläne

left-deep:

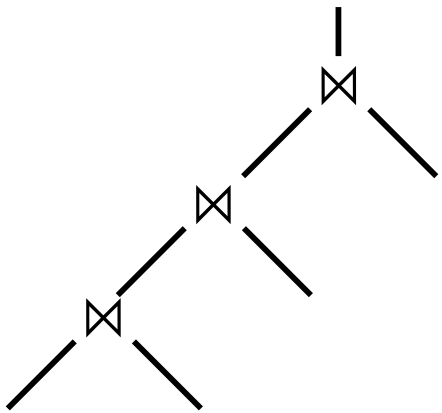


right-deep:

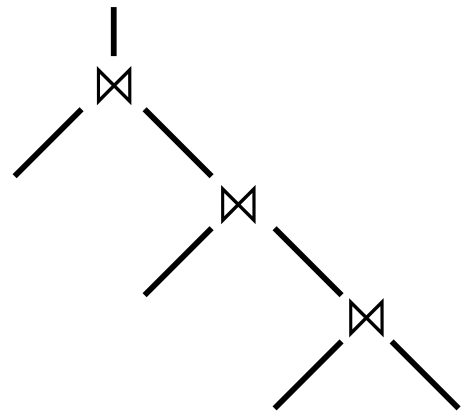


Vier Eingaberelationen: $C_4 = 5$ Pläne

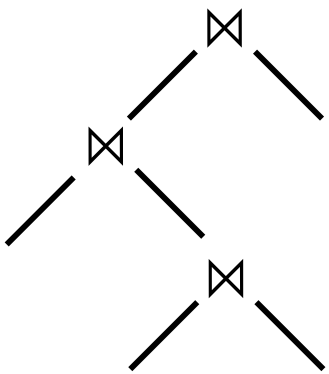
left-deep:

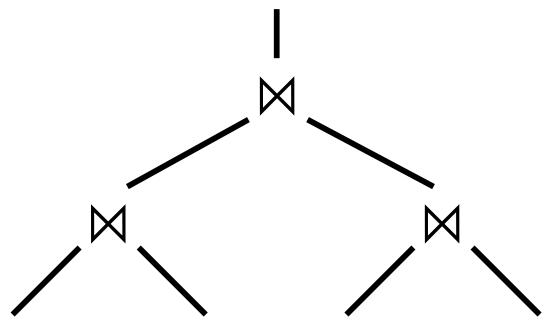


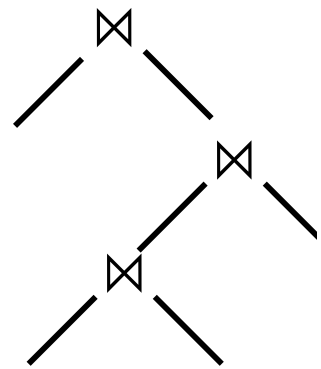
right-deep:



bushy:







zu 3. Joinreihenfolge: Reihenfolge der Eingaberelationen

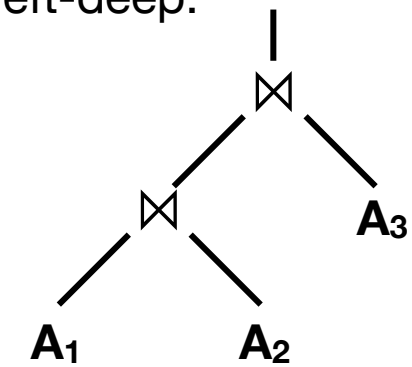
3.2 Reihenfolge der Eingaberelationen

Für n Eingaberelationen gibt es $n!$ viele Möglichkeiten diese anzuordnen und dann zuzuordnen zu den verschiedenen Baumstrukturen des Plans.

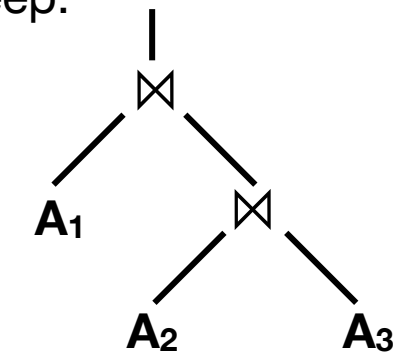
Berücksichtigen wir also die Planstruktur **und** die Reihenfolge der Eingaberelationen, sind wir bereits bei $C_n \cdot n!$ vielen Plänen.

Planstruktur und Reihenfolge der Eingaberelationen

left-deep:



right-deep:



jeweils 6 Eingabereihenfolgen:

A_1	A_2	A_3
R	S	T
R	T	S
S	R	T
S	T	R
T	R	S
T	S	R

$C_3 \cdot 3! = 2 \cdot 6 = 12$ verschiedene Pläne (theoretisch...)

Kurze Zwischenfrage

Realitäts-Check: Ergibt es überhaupt Sinn, alle möglichen Joinreihenfolgen aufzuzählen?

Wenn zwischen zwei Relationen keine Join-Bedingung definiert ist, müssen diese Relationen durch ein Kreuzprodukt verbunden werden! Und das wird mit hoher Wahrscheinlichkeit (zu) teuer. Wieso sollten wir diese Möglichkeit bei der Aufzählung überhaupt berücksichtigen?

Um diese genauer zu fassen, benötigen wir noch zwei weitere Begriffe: Join-Selektivität und Join-Graph.

Join-Selektivität

Join-Selektivität

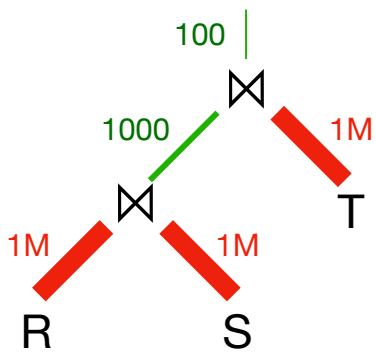
Mit Join-Selektivität bezeichnen wir das Verhältnis der Größe des Join-Ergebnisses zur Größe des Kreuzproduktes der Eingaberelationen:

$$sel_{R \bowtie S} = \frac{|R \bowtie S|}{|R \times S|} \leq 1$$

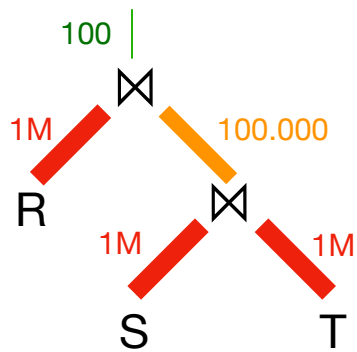
Falls für R uns S kein Join-Prädikat existiert, folgt $sel_{R \bowtie S} = sel_{R \times S} = 1$.

Beispiel:

$|R| = |S| = |T| = 10^6$, $sel_{R \bowtie S} = 10^{-9}$, $sel_{S \bowtie T} = 10^{-7}$,
$sel_{(R \bowtie S) \bowtie T} = sel_{R \bowtie (S \bowtie T)} = 10^{-16}$.



$\text{Kosten}_{\text{Zwischenergebnisse}} = 1100$



$\text{Kosten}_{\text{Zwischenergebnisse}} = 100.100$

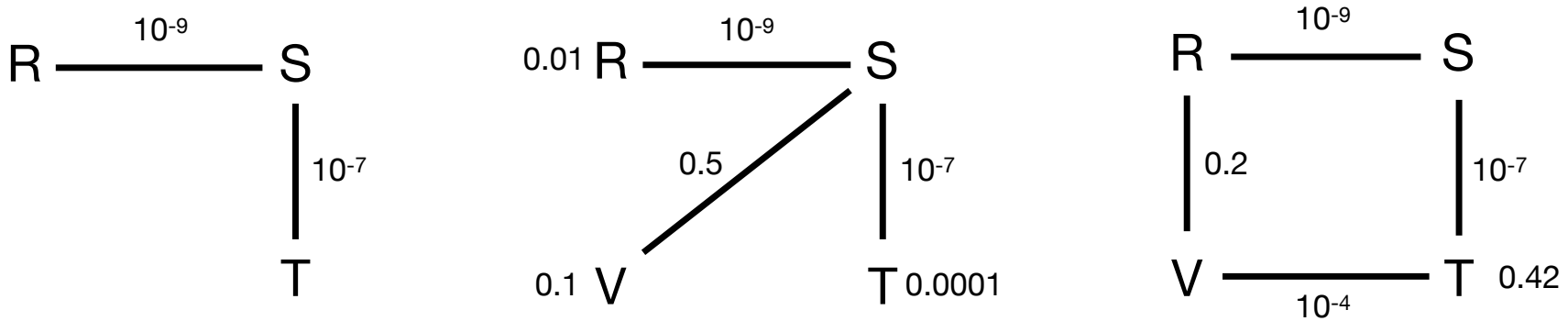
Diese beiden Joinreihenfolgen haben unterschiedliche Kosten.

Join-Graph (aka Query-Graph)

Join-Graph (aka Query-Graph)

Ein Join-Graph hat einen Knoten für jede Eingaberelation und eine Kante für jede Join-Bedingung. Die Kante wird annotiert mit der Join-Selektivität (optional mit dem Join-Prädikat).
Des Weiteren können Selektivitäten von Filterbedingungen auf einzelnen Knoten annotiert werden.

Beispiel:



Drei Join-Graphen auf demselben Schema generiert von drei unterschiedlichen Anfragen.

Antwort zur Zwischenfrage

Realitäts-Check: Macht es überhaupt Sinn, alle theoretisch möglichen Joinreihenfolgen aufzuzählen?

...

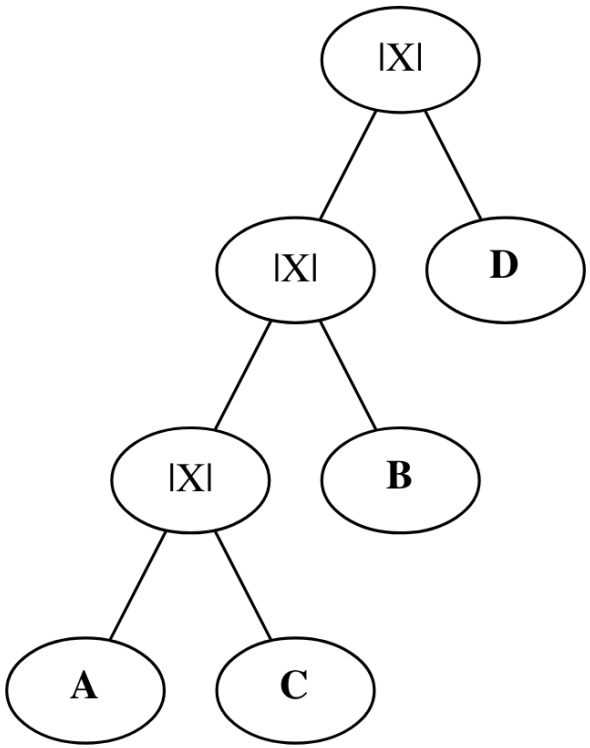
Antwort: Das ergibt keinen Sinn.

Die Aufzählung sollte nur entlang des Join-Graphen erfolgen und Kreuzprodukte ignorieren
(außer der Join-Graph ist nicht zusammenhängend).

Plan-Enumeration in Python

Find the cheapest plan for our join graph.

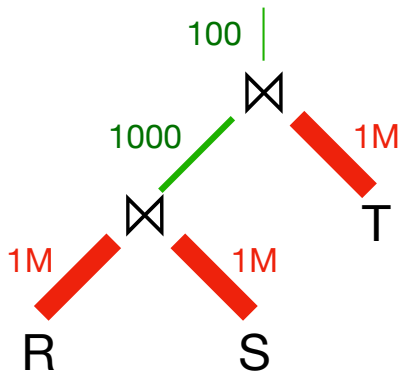
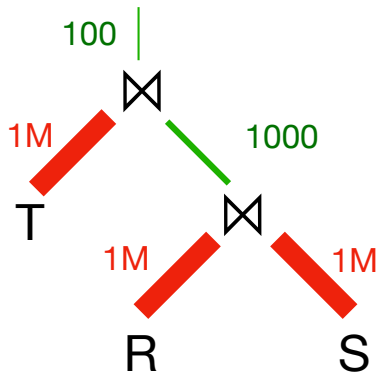
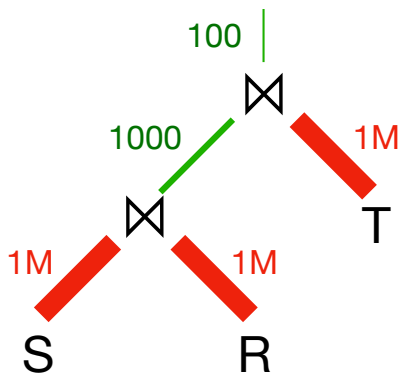
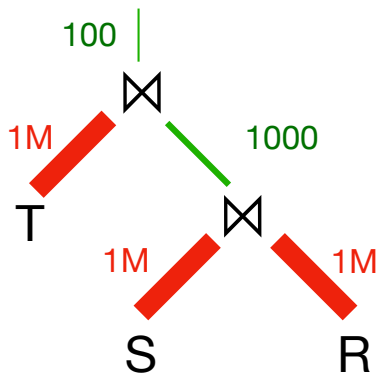
In [15]:
```
plan = find_cheapest_plan_exhaustive(the_join_graph, CostFunctions.size_of_inputs)
display(gv.Source(draw_query_plan(plan)))
cost = compute_cost(plan, the_join_graph, CostFunctions.size_of_inputs)
print(f'The cheapest plan is {plan} with cost {cost:,}.')
```




```
The cheapest plan is ('A', 'C', 'B', 'D') with cost 61,000.
```

github: Plan Enumeration.ipynb

Joinreihenfolge und Kommutativität

	T rechts	**T links**
R links		
S links		

Diese vier Pläne haben in diesem Kostenmodell dieselben Kosten:
$\text{Kosten}_{\text{Zwischenergebnisse}} = 1100$.

HashJoin-Algorithmus[1]

HashJoin (Relationen R und S, Joinprädikat JP:= r.x==s.y)

1. HashMap hm;
2. Relation ergebnis = {};

Füge alle Tupel aus der linken Eingabe R in die HashMap ein:

3. Für alle r aus R:
4. hm.insert(r.x, r);

Suche alle Tupel aus der rechten Eingabe S in der HashMap:

5. Für alle s aus S:
6. RES = hm.probe(s.y);

Falls die HashMap Einträge für den Schlüssel $s.y$ hat:

7. Falls RES $\neq \emptyset$:

Füge Teilergebnis *RES* zu Gesamtergebnis hinzu:

8. ergebnis = ergebnis $\cup$ RES $\times$ { s };
9. Return ergebnis;

[1] vgl. Übungszettel 2, Aufgabe 3

Na und?

Der HashJoin in verschiedenen Kostenmodellen (vgl. Teil 1, Folie 25):

Kostenmodell: Gesamtlaufzeit

$$\text{Kosten}_{\text{Gesamtlaufzeit}}(\mathit{HashJoin}) \approx c_1 \cdot |R| + c_2 \cdot |S|$$

Es entstehen lineare Kosten in der Größe der Eingaberelation für das Aufbauen der HashMap auf Eingaberelation R. Dann nochmal lineare Kosten in der Größe der Eingaberelation S für das Anfragen mit allen Tupeln.

- Die genauen Werte von c_1 und c_2 hängen von vielen Faktoren ab: Füllgrad der HashMap, Art der HashMap, Datenverteilung, etc.[2]
- Falls c_1 und c_2 ähnlich groß sind, macht HashJoin(R,S) oder HashJoin(S,R) keinen großen Unterschied. Dann sind die Kosten des Joins näherungsweise **symmetrisch**.
- Die Selektivität des Joins haben wir hier ignoriert. Eine Kombination von $\text{Kosten}_{\text{Gesamtlaufzeit}}$ mit $\text{Kosten}_{\text{Zwischenergebnisse}}$ ist sinnvoll.

[2] Weiterführende Literatur: Stefan Richter, Victor Alvarez, Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. PVLDB/VLDB 2016.

Der HashJoin im Hauptspeicher (nur Speicherkosten)

Kostenmodell: HashJoin im Hauptspeicher

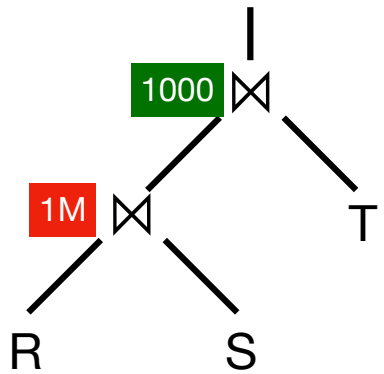
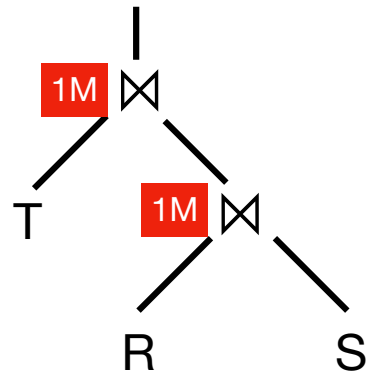
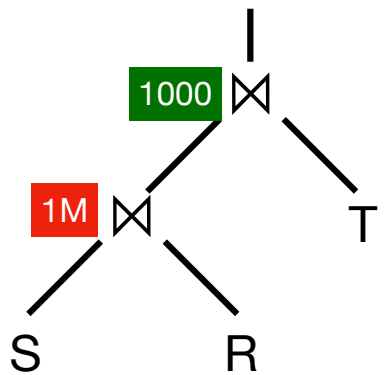
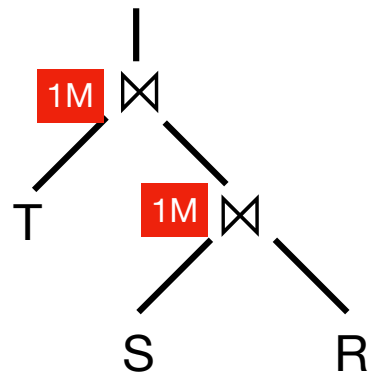
$\text{Kosten}_{\text{Hauptspeicher}}(\mathit{HashJoin}) \approx c_3 \cdot c_4 \cdot |R|$.

c_3 := Overhead der HashMap für das Speichern eines Eintrags.
c_4 := Größe eines Tupels aus R.

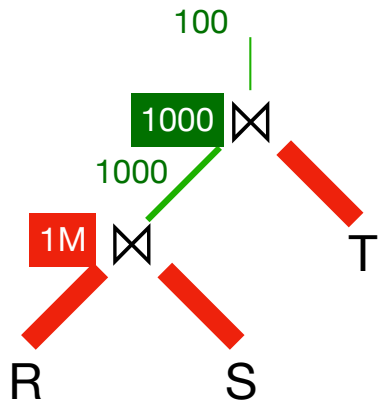
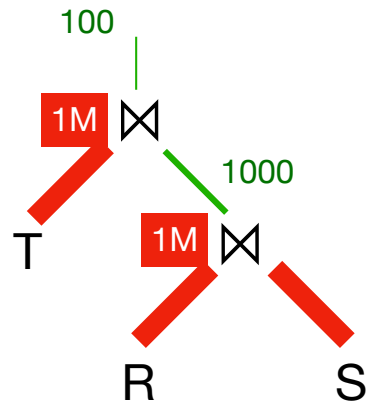
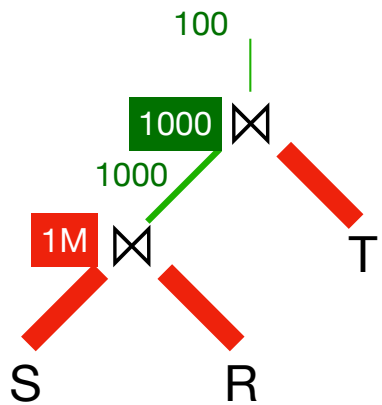
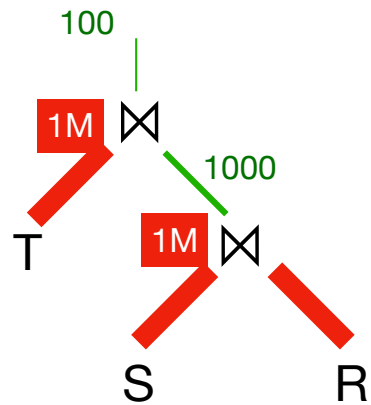
Es entstehen lineare Kosten für das Aufbauen der HashMap auf Eingaberelation R. Dann nochmal ..., nee das war's!

- die Größe von S spielt in diesem Kostenmodell **keine** Rolle.
- Es ist in diesem Kostenmodell von Vorteil, die kleinere Eingaberelation als linke Eingaberelation zu nehmen.
- Die Kosten des HashJoins sind in diesem Kostenmodell **nicht symmetrisch**.

Joinreihenfolge und Hauptspeicherkosten

	T rechts	**T links**
R links		
S links		
	$\text{Kosten}_{\text{Hauptspeicher}} = c_3 \cdot c_4 \cdot 1.001.000!$	$\text{Kosten}_{\text{Hauptspeicher}} = c_3 \cdot c_4 \cdot 2.000.000!$

Beide Kostenmodelle kombiniert

	T rechts	**T links**
R links		
S links		

Beide Kostenmodelle kombiniert

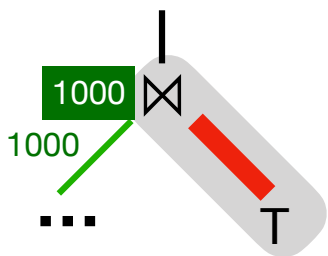
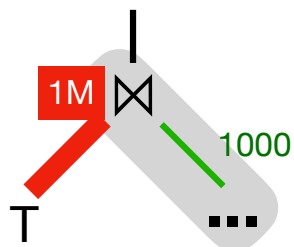
Kostenmodell: Kombination von Hauptspeicher und Zwischenergebnissen

$$\begin{aligned}\text{Kosten}_{\text{kombiniert}}(\mathit{HashJoin}) := \\ & c_5 \cdot \text{Kosten}_{\text{Hauptspeicher}}(\mathit{HashJoin}) \\ & + c_6 \cdot \text{Kosten}_{\text{Zwischenergebnisse}}(\mathit{HashJoin}).\end{aligned}$$

c_5, c_6: Gewichtungsfaktoren

Kostenmodell vs reale Kosten: Abgleich von Modell und Realität...

Falls T von rechts drangejoint wird, haben wir modelliert, dass dies keine zusätzliche **Speicherkosten** für T verursacht. Spiegelt das die Realität wieder?

	T rechts	**T links**
R links		

In beiden Fällen wird die rechte Eingabe des Joins, (die bei "T rechts" 1 Millionen Tupel enthält), als Parameter übergeben, also „irgendwie" durch den Hauptspeicher bewegt!

HashJoin (Relationen L und R, Joinprädikat JP:= l.x==r.y)

...

HashJoin-Algorithmus: Kosten im Hauptspeicher

Was bedeutet es, wenn Relationen **als Parameter** an eine Funktion übergeben werden wie hier L und R (linke und rechte Eingabe)?

- Sind dann die kompletten Relationen im Hauptspeicher (oder auf einem anderen Speichermedium) materialisiert?
- Was, wenn eine (oder beide) der Relationen sehr groß ist (sind)?

Was passiert im HashJoin-Algorithmus?

```
234 class Equi_Join_HashBased(Equi_Join):
235     def _dot(self, graph, prefix):
236         return super()._dot(graph, prefix, "⋈_HashBased{},{}")
237
238     def evaluate(self):
239         l_eval_input = self.l_input.evaluate()
240         r_eval_input = self.r_input.evaluate()
```

Das `evaluate` in Zeile 240 wäre Grund genug, den Join mit „T rechts“ auch rot zu zeichnen! Denn hier werden alle Zwischenergebnisse der rechten Eingabe zunächst aufgesammelt (im Hauptspeicher materialisiert)!
⇒ Das Kostenmodell von oben ist im Widerspruch zur Implementierung!

Abhilfe? Option 1: Wir ändern das Modell

altes Modell von oben:

Kostenmodell: HashJoin im Hauptspeicher

$\text{Kosten}_{\text{Hauptspeicher}}(\mathit{HashJoin}) \approx c_3 \cdot c_4 \cdot |R|$.

neues Modell:

Kostenmodell: HashJoin im Hauptspeicher inklusive Kosten für rechte Eingaberelation)

$$\text{Kosten}_{\text{Hauptspeicher komplett}}(\mathit{HashJoin}) = \text{Kosten}_{\text{Hauptspeicher}}(\mathit{HashJoin}) + c_5 \cdot |S|.$$

c_5 := Größe eines Tupels aus S.

Option 2: Wir ändern die Realität

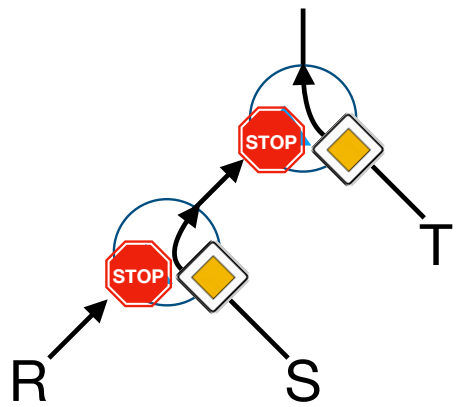
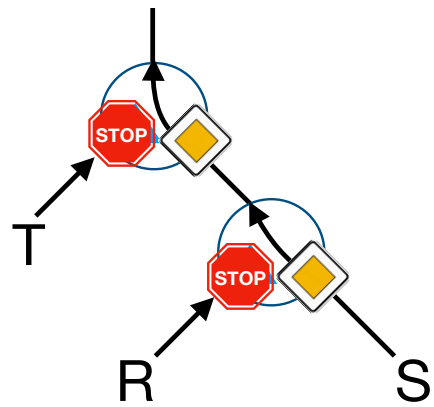
Beobachtung

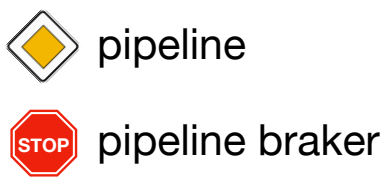
Aktuell werden für jeden Aufruf die Relationen materialisiert. Das ist aber eigentlich unnötig. Wann materialisiert werden muss, hängt vom Operator ab.

Beispiele: Die Eingabe wird komplett gelesen und das Ergebnis materialisiert, bevor ein Ergebnis an den nächsten Operator weitergereicht wird!

- Selektion und Projektion:
  Aber: eigentlich kann jedes Tupel unabhängig gefiltert oder projiziert werden, d.h. wir könnten es direkt weitergeben!
- Aggregation mit max():
  Aber: Wir müssen zwar alle Tupel anschauen, um entscheiden zu können, was das Maximum ist. Allerdings brauchen wir nicht alle Eingabetupel zwischenzuspeichern. Wir könnten jedes Tupel direkt mit dem aktuellen Maximum vergleichen.

Pipelines vs Pipeline-Breaker

	T rechts	**T links**
links		

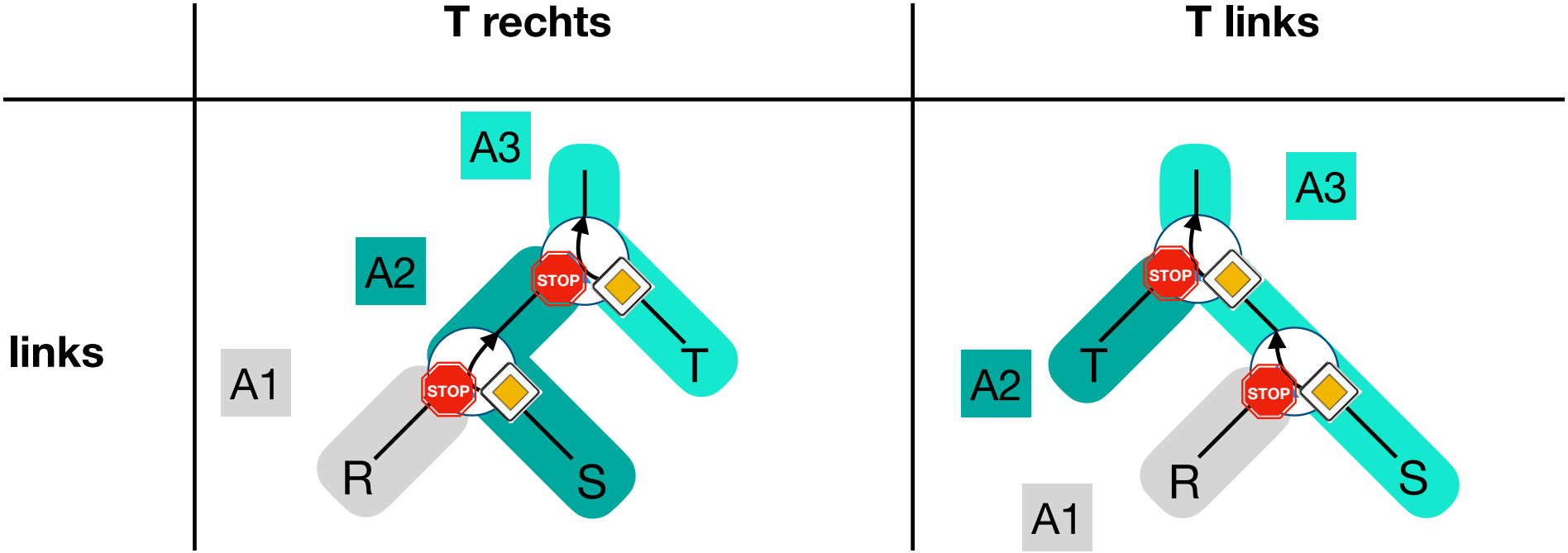


Pipeline-Breaker

An gewissen Punkten im physischen Plan **müssen** Daten materialisiert werden. Diese Stellen nennen wir Pipeline-Breaker. STOP

z.B. im HashJoin für das Befüllen der Hash-Tabelle

Planabschnitte



Pipeline-Breaker und Planabschnitte

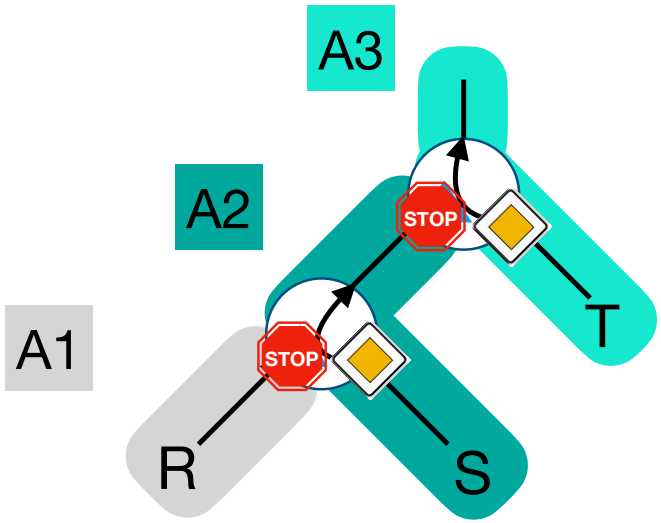
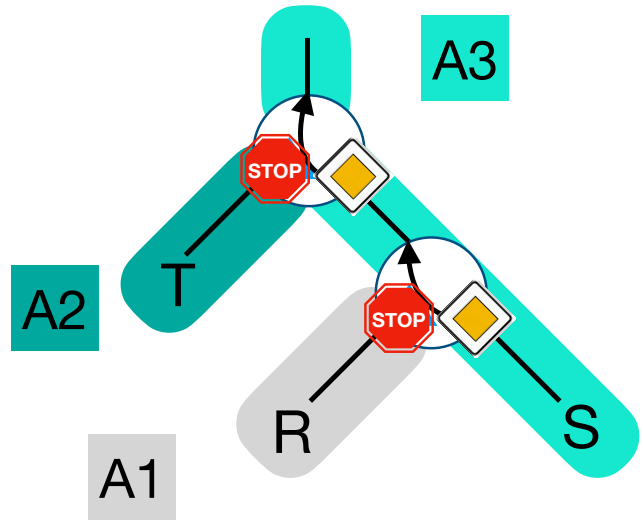
STOP Pipeline-Breaker partitionieren einen physischen Plan auf natürliche Weise in Planabschnitte.

Pipelining

◈ Daten können innerhalb eines Planabschnitts gestreamt (gepipelint[3]) werden, ohne den Datenfluss zu unterbrechen.

[3] Ein Fall für die Sprachpolizei. Diese Ausdrücke sind aber gebräuchlich.

Planabschnitte und Pipelining

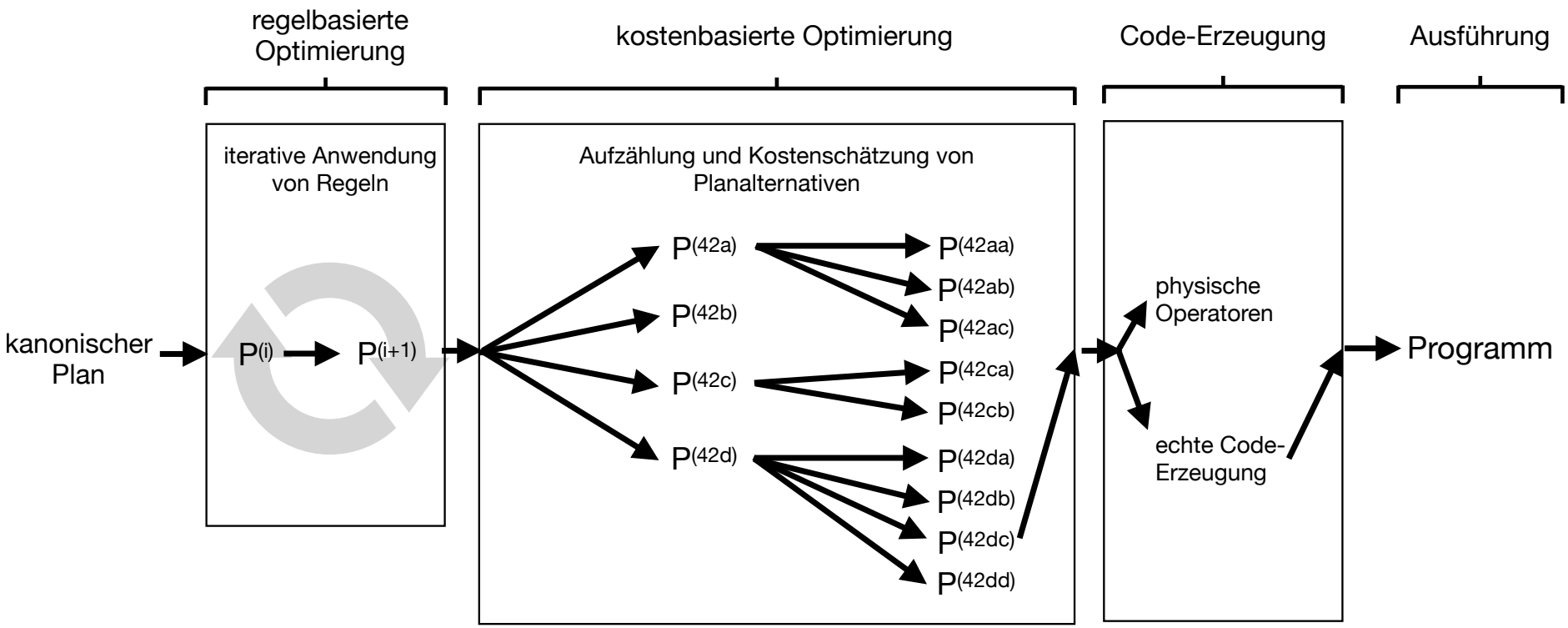
	T rechts	**T links**
links		
	Abschnitt A1: Relation R wird komplett in der Hash-Tabelle materialisiert Abschnitt A2: materialisiert nur R⋈S in der zweiten Hash-Tabelle Abschnitt A3: materialisiert überhaupt nichts ⇒ **guter Plan!** (bezüglich Pipelining)	Abschnitt A1: genau wie links Abschnitt A2: Relation T wird komplett in der Hash-Tabelle materialisiert Abschnitt A3: genau wie links ⇒ **nicht ganz so guter Plan!** (bezüglich Pipelining)

Code-Erzeugung

4. physischer Plan
   ↓ Code-Erzeugung
5. ausführbarer Code

- im letzten Schritt erzeugen wir aus dem physischen Plan ausführbaren Code
- typischerweise wird C/C++ oder direkt LLVM erzeugt

Optimierung, Code-Erzeugung und Ausführung: Übersicht



Plan-Interpretation vs Code-Erzeugung

Plan-Interpretation

Bei einigen Systemen fällt die Code-Erzeugung weg: dann werden die geplanten physischen Operatoren 1:1 in Programmiersprachenkonstrukte umgesetzt, z.B. durch Aufrufe in eine Bibliothek eine physischer Operatoren. Dies nennen wir *Plan-Interpretation*.

Genau dies passiert in unserer Implementierung im Notebook „Rule-based Optimization.ipynb“:

```
physical_plan = compile_plan(bup_or.root)
```

Code-Erzeugung

Andere Systeme generieren in diesem Schritt Programmcode, der die gedachten Grenzen der physischen Operatoren durchbricht. Die für die Planung verwendeten physischen Operatoren werden **nicht notwendigerweise** 1:1 in Programmiersprachenkonstrukte umgesetzt. Dies nennen wir *Code-Erzeugung*.

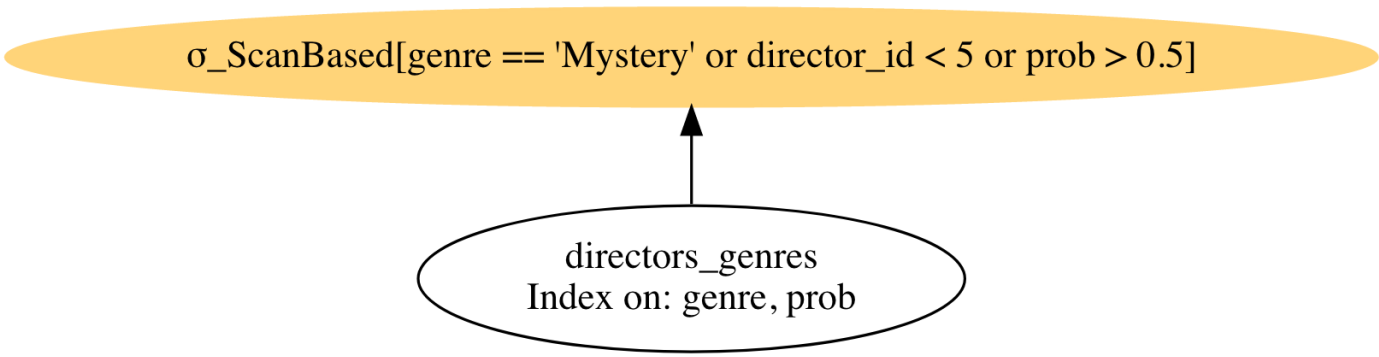
Plan-Interpretation in Python

Query Compilation

In [19]:
```
physical_plan = compile_plan(bup_or.root)
```

In [20]:
```
graph = physical_plan.get_graph()
Source(graph)
```

Out[20]:



In [21]:
```
physical_plan.evaluate().print_set()
```

```
[Result] : {[ director_id:int, genre:str, prob:float ]}
{
        (78273, Mystery, 0.125),
        (43095, Drama, 0.625),
        (78273, Drama, 0.75),
        (43095, Mystery, 0.0625)
}
```

github: Rule-based Optimization.ipynb

Echte Code-Erzeugung (nach Pseudo-Code)

1. HashMap hm1; HashMap hm2;

A1: Füge alle Tupel aus Eingabe R
in die HashMap hm1 ein:

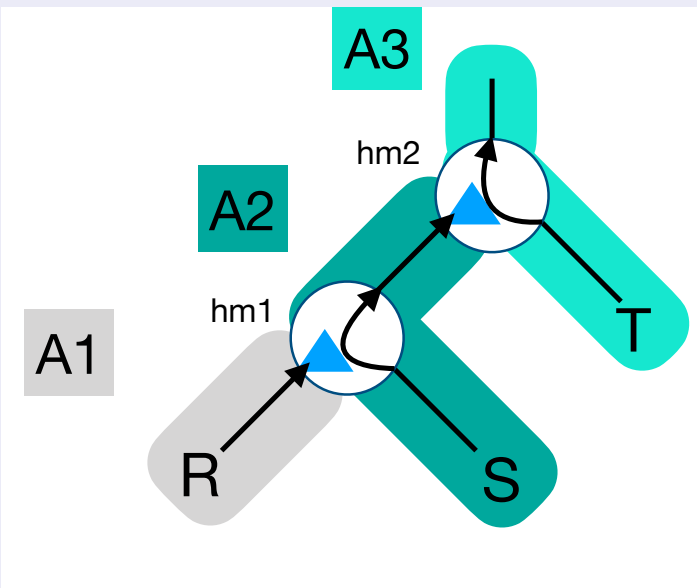
2. Für alle r aus R:
3. hm1.insert(r.x, r);

A2: Suche alle Tupel aus der rechten
Eingabe S in der HashMap hm1:

4. Für alle s aus S:
5. RES = hm1.probe(s.y);
6. Für alle z aus $(RES \times \{s\})$:
7. hm2.insert(z. ... , z);

A3: Suche alle Tupel aus der rechten Eingabe T in der HashMap hm2:

8. Für alle t aus T:
9. RES = hm2.probe(t...);
10. yield (RES $\times\{z\}$);



Horizontale Partitionierung einer Relation

Horizontale Partitionierung einer Relation

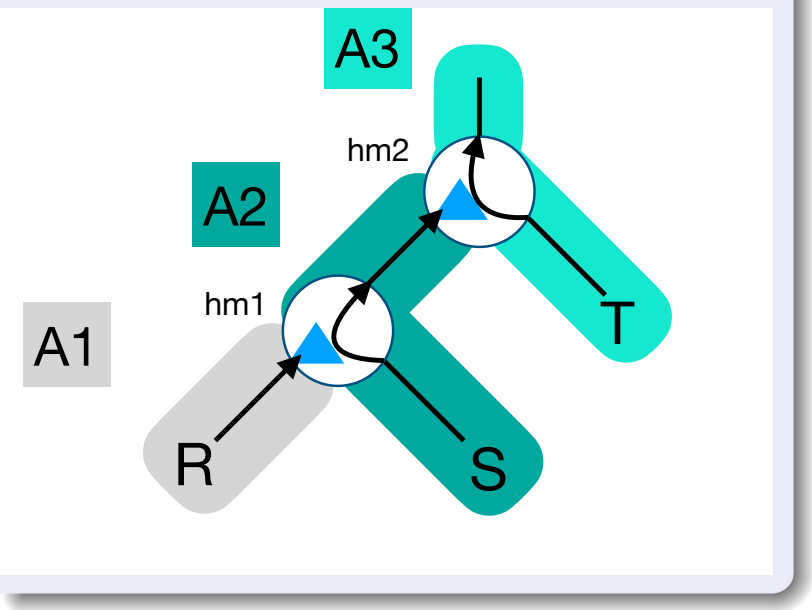
Eine Aufteilung einer Relation R in Teilrelationen $R_1, \ldots, R_n, n > 1$ heißt **Horizontale Partitionierung von R**, falls gilt:

1. $R = \bigcup_{i=1}^{n} R_i$ (vollständig),
2. $R_i \cap R_j = \varnothing \; \forall_{i \neq j, 1 \leq i,j \leq n}$ (paarweise disjunkt).

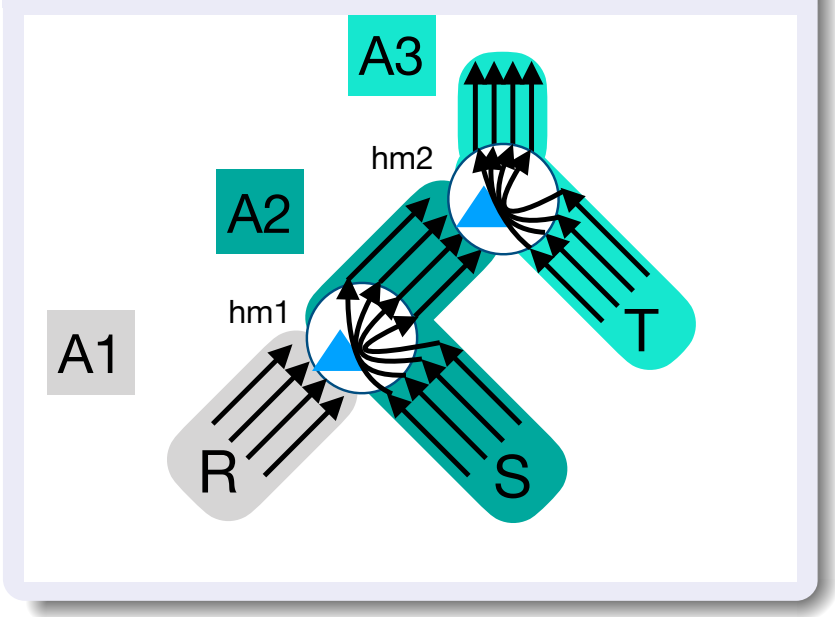
Beispiel:
die Gruppierung (ohne Aggregation!) in relationaler Algebra und SQL

Multi-Threading

Single-Threaded



Multi-Threaded (4 Threads)



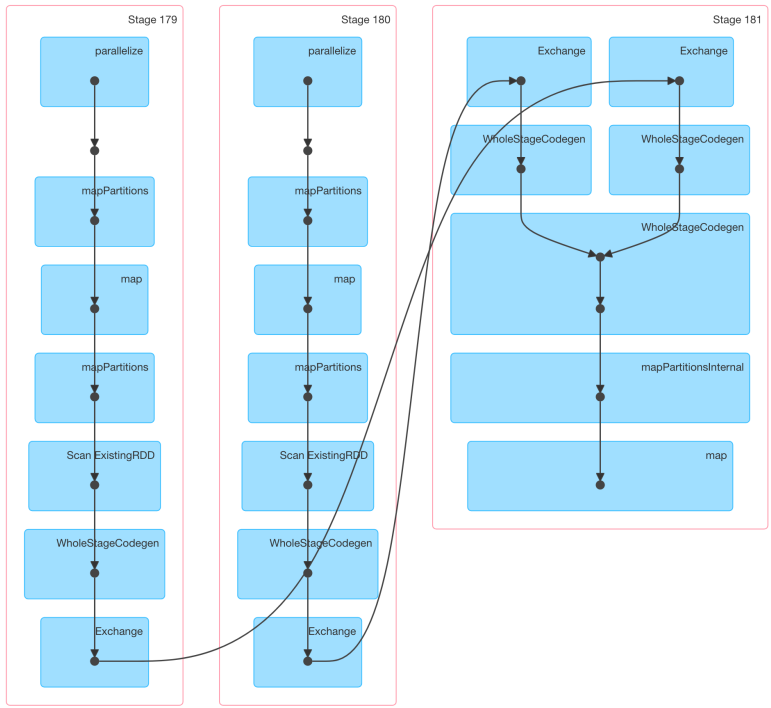
- Die Code-Erzeugung lässt sich in diesem Fall leicht erweitern, um Multi-Threading zu unterstützen (im Allgemeinen kann das beliebig komplex werden...).
- Hier werden die Eingaberelationen **horizontal partitioniert**.
- Jede Partition wird dann in einem separaten Thread abgearbeitet.
- Die einzigen Berührungspunkte der Threads sind die beiden (hoffentlich) Thread-sicheren Hash-Tabellen.

Multi-Threading und Planabschnitte in Spark

```
== Analyzed Logical Plan ==
last_name: string, prob: float
Project [last_name#14, prob#24]
+- Filter ((last_name#14 = Tarantino) && (genre#23 = Mystery))
   +- Filter (id#12 = director_id#22)
      +- Join Cross
         :- LogicalRDD [id#12, first_name#13, last_name#14], false
         +- LogicalRDD [director_id#22, genre#23, prob#24], false

== Optimized Logical Plan ==
Project [last_name#14, prob#24]
+- Join Cross, (id#12 = director_id#22)
   :- Project [id#12, last_name#14]
   :  +- Filter (last_name#14 = Tarantino)
   :     +- LogicalRDD [id#12, first_name#13, last_name#14], false
   +- Project [director_id#22, prob#24]
      +- Filter (genre#23 = Mystery)
         +- LogicalRDD [director_id#22, genre#23, prob#24], false

== Physical Plan ==
*(5) Project [last_name#14, prob#24]
+- *(5) SortMergeJoin [id#12], [director_id#22], Cross
   :- *(2) Sort [id#12 ASC NULLS FIRST], false, 0
   :  +- Exchange hashpartitioning(id#12, 200)
   :     +- *(1) Project [id#12, last_name#14]
   :        +- *(1) Filter (last_name#14 = Tarantino)
   :           +- Scan ExistingRDD[id#12,first_name#13,last_name#14]
   +- *(4) Sort [director_id#22 ASC NULLS FIRST], false, 0
      +- Exchange hashpartitioning(director_id#22, 200)
         +- *(3) Project [director_id#22, prob#24]
            +- *(3) Filter (genre#23 = Mystery)
               +- Scan ExistingRDD[director_id#22,genre#23,prob#24]
```



- github: Spark.ipynb
- In Spark heißen Planabschnitte *Stages*.
- Innerhalb einer Stage werden Daten durch horizontale Partitionierung aufgeteilt.
- Pro Partition wird ein Thread benutzt.

Was mache ich, wenn die Anfragen zu langsam sind?

Frage 1

Wie kommen wir eigentlich prinzipiell von SQL zu einem ausführbaren Programm?

Durch automatische Regel- und kostenbasierte Optimierung.

Zusammenfassung: Kostenbasierte Optimierung

3. transformierter logischer Plan
   ↓ kostenbasierte Optimierung
4. physischer Plan

Was sind die Hauptaufgaben der kostenbasierten Optimierung?

1. Aufzählung und Bewertung von Planalternativen (benötigt Statistiken und Kostenschätzungen)
2. Wahl der Joinreihenfolge
3. Wahl der physischen Operatoren
4. Planung des Pipelining (Blocking vs. Non-Blocking Operators)
5. Planung des Multi-Threading

Zusammenfassung: Code-Erzeugung

4. physischer Plan
   ↓ Code-Erzeugung
5. ausführbarer Code

Was sind die Hauptaufgaben der Code-Erzeugung?

Grundsätzliche Entscheidung treffen: wie den Plan ausführen?
Entweder:

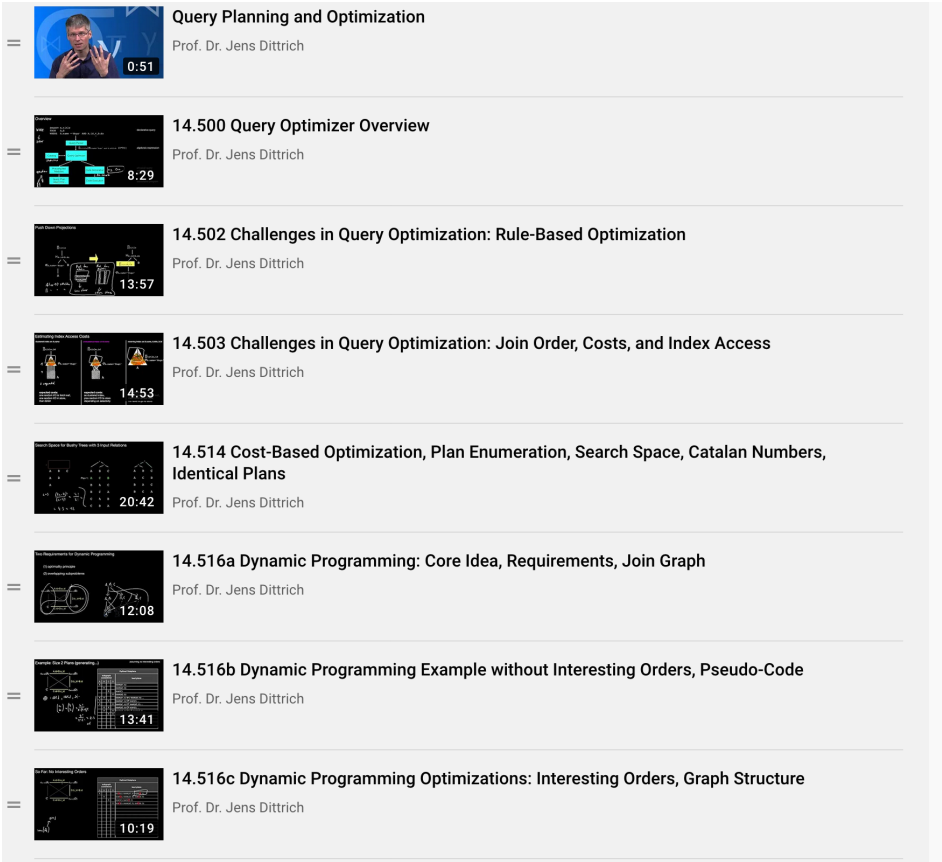
A Interpretation: Ein Baum aus physischen Operatoren wird geeignet traversiert und ausgeführt.

B (echte) Code-Erzeugung: Ein Programm, das das Ergebnis der Anfrage berechnet, wird erstellt, kompiliert und ausgeführt. Operatorengrenzen werden dabei durchbrochen.

Achtung

Wir haben für alle diesen Themen bisher nur ein wenig an der Oberfläche gekratzt... Es gibt sehr viele interessante Techniken in dem Bereich:

Weiterführendes Material



Youtube Videos von Prof. Dittrich zu Anfrageoptimierung (Englisch)

sowie:

- Jens Dittrich. Patterns in Data Management. https://bigdata.uni-saarland.de/datenbankenlernen/book.pdf
- Stammvorlesung Database Systems im WS 20/21