

Anfrageoptimierung (Teil 2)

VL Big Data Engineering
(vormals Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

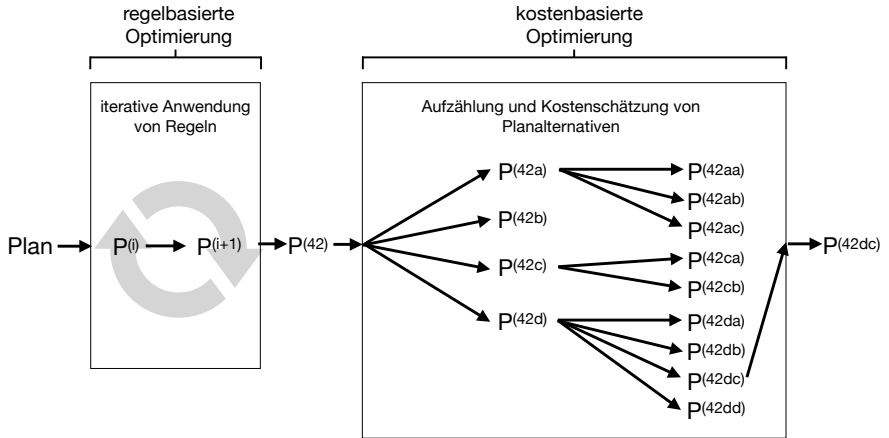
15. Juni 2022

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:

2. Welchen physischen Operator nutzen?

Beispiele: Hash-basierter Join, sortierbasierter Join oder XY Join?
(vgl. logische vs physische Operatoren-Diskussion)

3. 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](#))

4. Welche Ressourcen für welchen Teilplan nutzen?

Beispiele: Wieviele Threads wo einsetzen? Welcher Teil des Plans bekommt wie viel Rechenzeit/Hauptspeicher?

zu 3. Index benutzen oder nicht?

Um die Ergebnisse einer Anfrage der Form $\sigma_P(R)$ zu berechnen, haben wir im Grunde nur zwei Möglichkeiten:

Brute-Force (aka Scan)

Wir untersuchen jedes einzelne Tupel in R und prüfen, ob P zutrifft, und wenn das der Fall ist, fügen wir das Tupel zur Ergebnismenge hinzu.

VS

Index (aka Index Scan)

Wir organisieren den Inhalt von R so, dass wir zur Auswertung der Anfrage nicht jedes einzelne Tupel untersuchen müssen, um festzustellen, ob ein Tupel zum Ergebnis einer Anfrage gehört.

Induzierte rekursive horizontale Partitionierung

Induzierte rekursive horizontale Partitionierung

Seien p_0, p_1, \dots, p_k Partitionierungsfunktionen mit $p_i : [R] \rightarrow D_i \ \forall_{0 \leq i \leq k}$.

Start: Wir wenden p_0 auf R an. Dies ergibt eine Menge von horizontalen Partitionen $\{R_{i_0}\}$ für die $\forall_{t \in R} t \in R_{p_0(t)}$. (vgl. nicht-rekursive induzierte horizontale Partitionierung, 02 IMDb-Folien)

Rekursionsschritt: Wir wenden p_{i+1} iterativ auf alle horizontalen Partitionen an, die durch den vorherigen Partitionierungsschritt $p_{i \geq 0}$ erzeugt wurden.

Menge induzierter Partitionen: Dies ergibt eine Menge von horizontalen Partitionen $\{R_{i_0, \dots, i_k}\}, i_j \in D_{I_j}$.

Beispiel:

$R = \{(2, A), (7, B), (1, B), (6, C)\}$

$k = 1, p_0 : [R] \rightarrow \text{int}, p_0(t) := t.a \bmod 2, p_1 : [R] \rightarrow \text{char}, p_1(t) := t.b,$

Induzierte rekursive horizontale Partitionierung:

$R_{0,A} = \{(2, A)\}, R_{0,C} = \{(6, C)\}, R_{1,B} = \{(7, B), (1, B)\}$

Index

Index

Jede Menge von (rekursiven oder nicht rekursiven) horizontalen Partitionierungsfunktionen, die von einer Anfrage genutzt werden können, um den Suchraum der Anfrage zu reduzieren, wird als *Index* (oder Indexstruktur) bezeichnet.

Beispiele:

$$[R] = \{[a : \text{int}, b : \text{char}]\}$$

$$R_{0,A} = \{(2, A)\}, R_{0,C} = \{(6, C)\}, R_{1,B} = \{(7, B), (1, B)\}$$

$$p_0(t) := t.a \bmod 2, p_1(t) := t.b$$

$$\Rightarrow \sigma_{b='C'}(R) = \sigma_{b='C'}(R_{0,C}), \text{ d.h. statt 4 nur 1 Tupel filtern}$$

$$\Rightarrow \sigma_{a=7}(R) = \sigma_{a=7}(R_{1,B}), \text{ d.h. statt 4 nur 2 Tupel filtern}$$

$$\Rightarrow \sigma_{a=8}(R) = \sigma_{a=8}(R_{0,A} \cup R_{0,C}), \text{ d.h. statt 4 nur 2 Tupel filtern}$$

Typische Indexstrukturen in SQL-Engines

Hash-Tabellen, B-Bäume (ggf. optimiert für Hauptspeicher), Bitmaps

zu 1. Joinreihenfolge: Baumstruktur des Plans

Die Dimension „Joinreihenfolge“ besteht aus unterschiedlichen Teilproblemen:

3.1 Baumstruktur des Plans

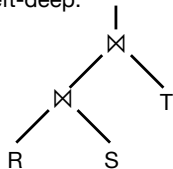
Für n Eingaberelationen gibt es C_{n-1} viele Binärbäume mit n Blattknoten.

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

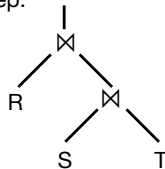
Die Catalan-Zahlen beginnen mit C_0, C_1, \dots sind 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, ...

Drei Eingaberelationen: $C_2 = 2$ Pläne

left-deep:

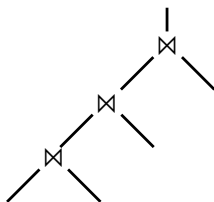


right-deep:

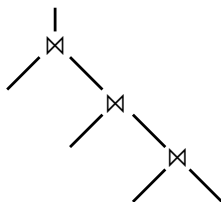


Vier Eingaberelationen: $C_3 = 5$ Pläne

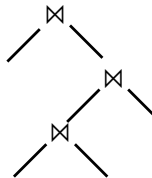
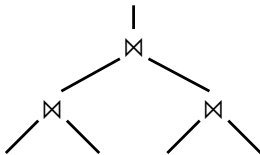
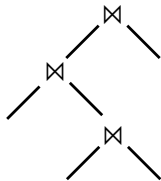
left-deep:



right-deep:



bushy:



zu 1. Joinreihenfolge: Reihenfolge der Eingaberelationen

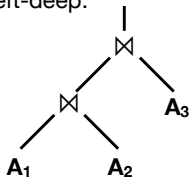
3.2 Reihenfolge der Eingaberelationen

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

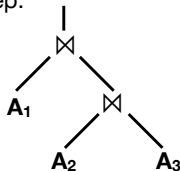
Berücksichtigen wir also die Planstruktur **und** die Reihenfolge der Eingaberelationen, sind wir bereits bei $C_{n-1} \cdot n!$ vielen Plänen.

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

$C_2 \cdot 3! = 2 \cdot 6 = 12$ 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

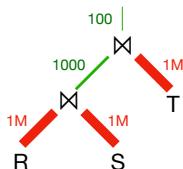
Mit Join-Selektivität bezeichnen wir das Verhältnis der Größe des Join-Ergebnisses zur Größe des Kreuzproduktes der Eingaberelationen:

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

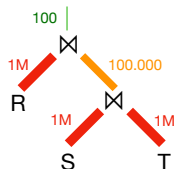
Falls für R und S kein Join-Prädikat existiert, folgt $sel_{R \bowtie S} = sel_{R \times S} = 1$.

Beispiel:

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



Kosten_{Zwischenergebnisse} = 1100



Kosten_{Zwischenergebnisse} = 100.100

Diese beiden Joinreihenfolgen haben unterschiedliche Kosten.

Prädikate

Ausdrücke, die z.B. in WHERE oder ON definiert werden, werden zu CNF transformiert und wir bezeichnen die “Literale” im Folgenden als Prädikate einer Anfrage.

Zur Erinnerung

Eine Formel in CNF ist eine Konjunktion von Klauseln. Klauseln sind Disjunktionen von Literalen.

Beispiel: (Formel in CNF)

$$\underbrace{(e = 42 \vee c < 3)}_{\substack{\text{Klausel} \\ \text{Literal} \quad \text{Literal}}} \wedge \underbrace{a \cdot b = 42}_{\text{Literal}} \wedge \underbrace{a + b = c}_{\text{Literal}}$$

Stelligkeit eines Prädikats

Stelligkeit eines Prädikats

Sei R_p die Menge der Relationen, deren Attribute im Prädikat p referenziert werden. Dann ist die Stelligkeit von p wie folgt definiert:

$$||p|| = |R_p|$$

D.h. die Anzahl der im Prädikat referenzierten Attribute wird mit Stelligkeit bezeichnet.

Beispiele:

$$||e = 42|| = 1$$

$$||c = a|| = 2$$

$$||a \cdot b = 42|| = 2$$

$$||a + b = c|| = 3$$

alle Beispiele unter der Annahme, dass die Variablen aus unterschiedlichen Relationen kommen

Spezialfälle der Stelligkeit

Filterprädikat

Jedes Prädikat fp mit $||fp|| = 1$: fp heißt Filterprädikat.

Join-Prädikat

Jedes Prädikat jp mit $||jp|| = 2$: jp heißt Join-Prädikat.

Im Folgenden betrachten wir nur Prädikate mit Stelligkeit 1 oder 2.

Beispiele:

$||c < 3|| = 1$ (Filterprädikat)

$||c = a|| = 2$ (Join-Prädikat)

Join-Graph (aka Query-Graph)

Join-Graph (aka Query-Graph)

Der Join-Graph $G = (V, JP, FP)$ einer Anfrage A mit Prädikaten P besteht aus:

1. der Menge der Relationen $V = \{R_1, \dots, R_k\}$ (Knoten),
2. der Menge der Join-Prädikate JP (Annotation der Kanten)

$$JP = \left\{ (R, jp, R') \mid jp \in P \wedge \|jp\| = 2 \wedge attr(jp) \subseteq ([R] \circ [R']) \right\}, \text{ und}$$

3. der Menge der Filter-Prädikate FP (Annotation der Knoten)

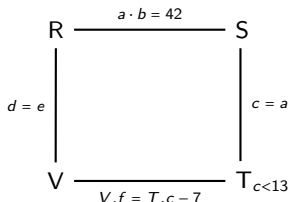
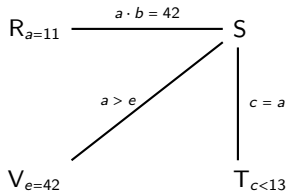
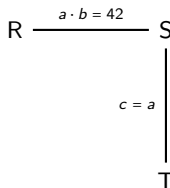
$$FP = \left\{ (R, fp) \mid fp \in P \wedge \|fp\| = 1 \wedge attr(fp) \subseteq [R] \right\}.$$

Für Prädikate mit Stelligkeit >2 kann die Definition des Join-Graphen erweitert werden zu einem Hyper-Graphen.

Beispiele für Join-Graphen

Beispiele:

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



Um Missverständnisse zu vermeiden, können Prädikate auch mit Punktnotation geschrieben werden, z.B. $V.f = T.c - 7$. Knoten mit Filterprädikaten können alternativ auch mit einem Filteroperator notiert werden: $\sigma_{c<13}(T)$ statt $T_{c<13}$.

Antwort zur Zwischenfrage

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

...

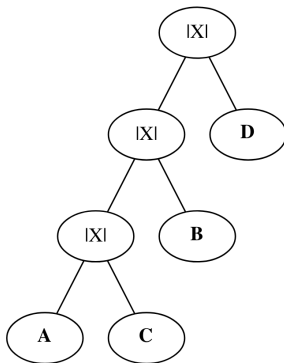
Antwort: Das ergibt keinen Sinn.

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.

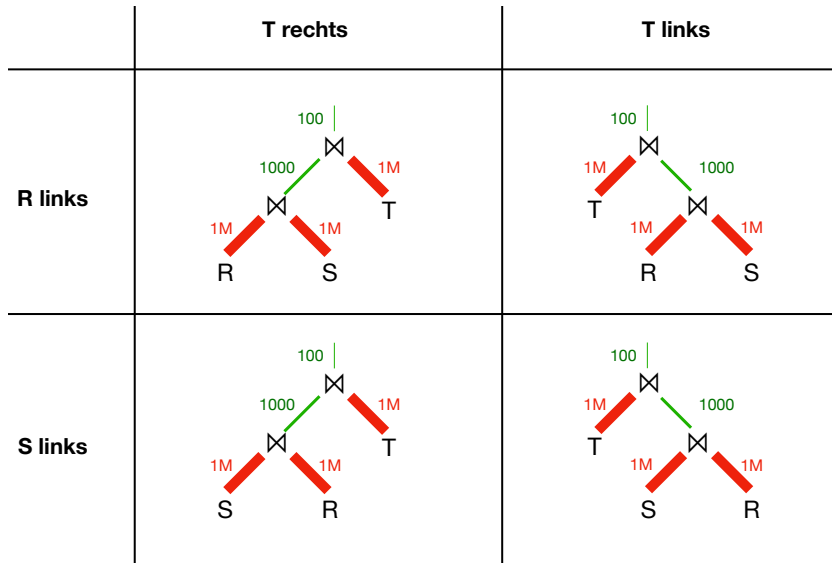
```
In [15]: plan = find_cheapest_plan_exhaustive(the_join_graph, CostFunctions.size_of_inputs)
display(gv.Source(draw_query_plan(plan)))
cost = compute_cost(plan, the_join_graph, CostFunctions.size_of_inputs)
print(f'The cheapest plan is {plan} with cost {cost:,}.')
```



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

github: [Plan Enumeration.ipynb](#)

Joinreihenfolge und Kommutativität



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

HashJoin-Algorithmus

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 $ergebnis$;

Na und?

Der HashJoin in verschiedenen Kostenmodellen:

Kostenmodell: Gesamtlaufzeit

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

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

- Die genauen Werte von c_1 und c_2 hängen von vielen Faktoren ab: Füllgrad der HashMap, Art der HashMap, Datenverteilung, etc.¹
- 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.

¹Weiterführende 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

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

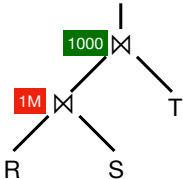
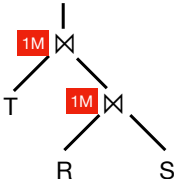
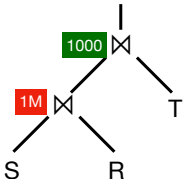
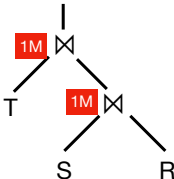
c_3 := Overhead der HashMap für das Speichern eines Eintrags.

c_4 := Größe eines Tupels aus R .

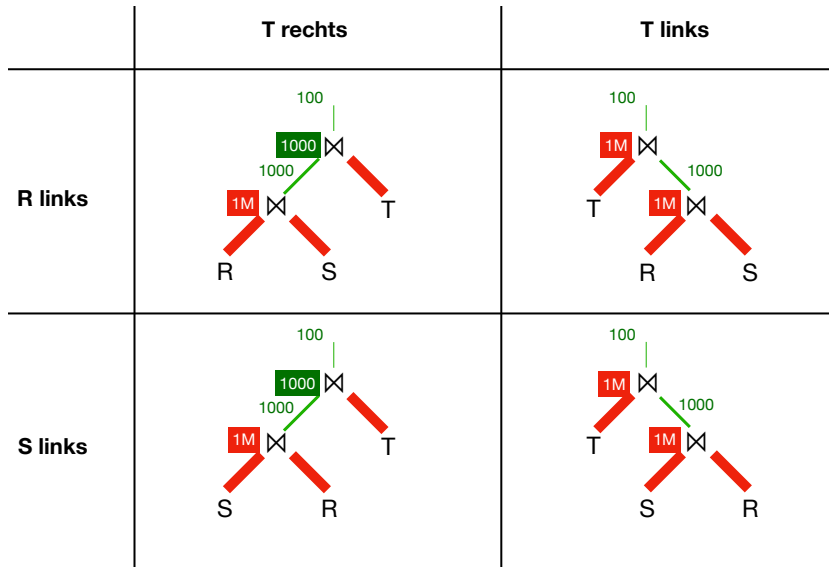
Es entstehen lineare Kosten für das Aufbauen der HashMap auf Eingaberelation R . Dann nochmal ..., nee, das war's!

- 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	 <p>Kosten_{Hauptspeicher} = $c_3 \cdot c_4 \cdot 1.001.000!$</p>	 <p>Kosten_{Hauptspeicher} = $c_3 \cdot c_4 \cdot 2.000.000!$</p>

Beide Kostenmodelle kombiniert



Beide Kostenmodelle kombiniert

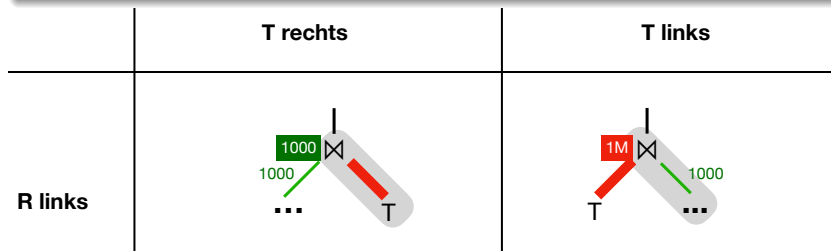
Kostenmodell: Kombination von Hauptspeicher und Zwischenergebnissen

$$\text{Kosten}_{\text{kombiniert}}(\textit{HashJoin}) := \\ c_5 \cdot \text{Kosten}_{\text{Hauptspeicher}}(\textit{HashJoin}) \\ + c_6 \cdot \text{Kosten}_{\text{Zwischenergebnisse}}(\textit{HashJoin}).$$

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?



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

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

neues Modell:

Kostenmodell: HashJoin im Hauptspeicher inklusive Kosten für rechte Eingaberelemente

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

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!

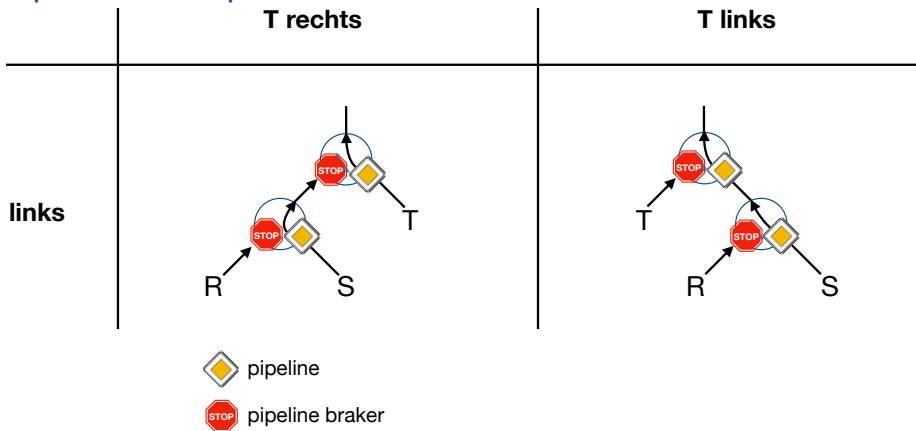
- Selektion und Projektion:

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 zwischenspeichern. Wir könnten jedes Tupel direkt mit dem aktuellen Maximum vergleichen.

Pipelines vs Pipeline-Breaker

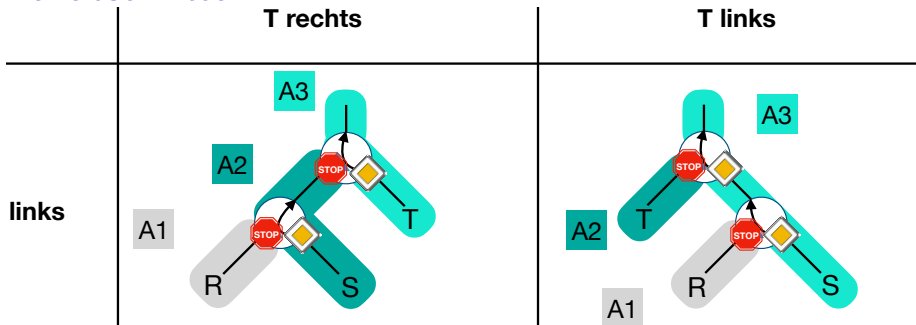


Pipeline-Breaker


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

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


Planabschnitte



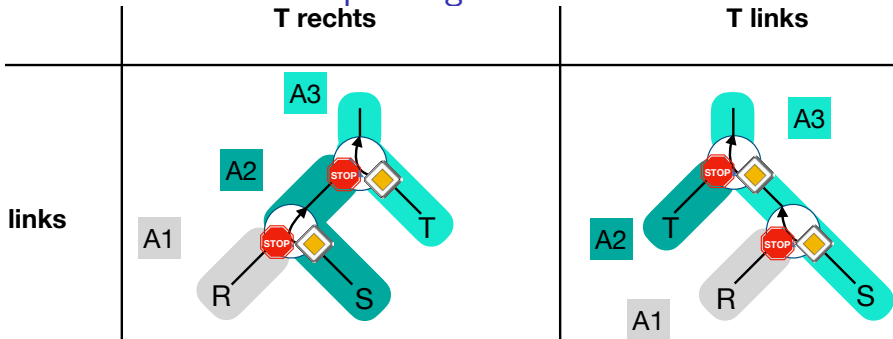
Pipeline-Breaker und Planabschnitte

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

Pipelining

 Daten können innerhalb eines Planabschnitts gestreamt (gepipelint²) werden, ohne den Datenfluss zu unterbrechen.

Planabschnitte und Pipelining



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)

falls $\text{sizeof}(R \rtimes S) < \text{sizeof}(T)$

Abschnitt A1: genau wie links

Abschnitt A2: Relation T wird komplett in der Hash-Tabelle materialisiert

Abschnitt A3: genau wie links

⇒ guter Plan!

(bezüglich Pipelining)

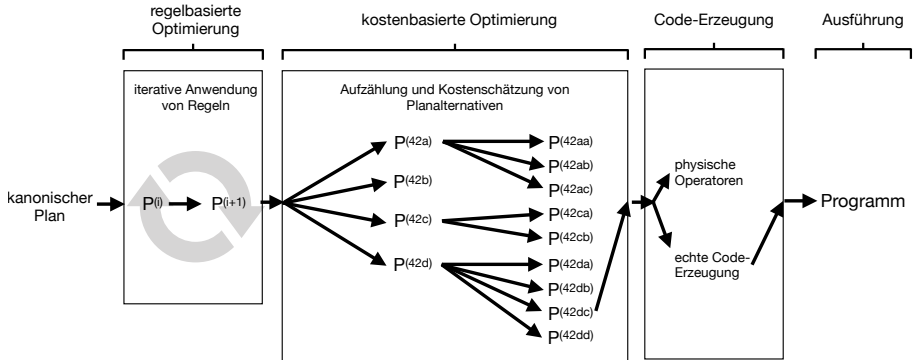
falls $\text{sizeof}(R \rtimes S) > \text{sizeof}(T)$

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 einer Bibliothek von physischen Operatoren. Dies nennen wir *Plan-Interpretation*.

Genau dies passiert in unserer Implementierung im Notebook.

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*.

Echte Code-Erzeugung (nach Pseudo-Code)

1. HashMap hm1; HashMap hm2;

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

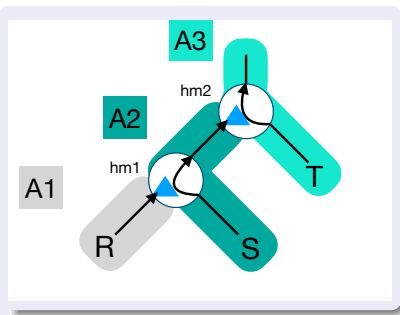
2. Für alle r aus R :
3. $\text{hm1.insert}(r.x, r);$

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

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

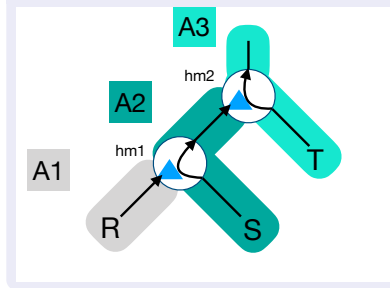
A3: Suche alle Tupel aus der rechten Eingabe T in der HashMap hm2:

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

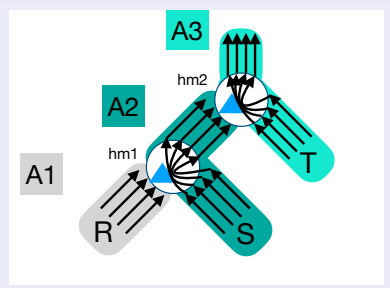


Multi-Threading

Single-Threaded



Multi-Threaded (4 Threads)



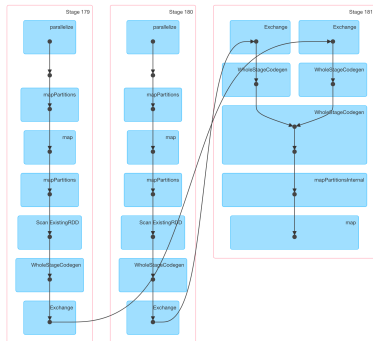
- Die Code-Erzeugung lässt sich in diesem Fall leicht erweitern, um Multi-Threading zu unterstützen (im Allgemeinen kann das beliebig komplex werden...).
- Hier werden die Eingaberelationen **horizontal partitioniert**.
- Jede Partition wird dann in einem separaten Thread abgearbeitet.
- Die einzigen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

Multi-Threading und Planabschnitte in Spark

```
== Analyzed Logical Plan ==
last_name: string, prob#24]
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. Operatorenengrenzen 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

	Query Planning and Optimization Prof. Dr. Jens Dittrich
	14.500 Query Optimizer Overview Prof. Dr. Jens Dittrich
	14.502 Challenges in Query Optimization: Rule-Based Optimization Prof. Dr. Jens Dittrich
	14.503 Challenges in Query Optimization: Join Order, Costs, and Index Access Prof. Dr. Jens Dittrich
	14.514 Cost-Based Optimization, Plan Enumeration, Search Space, Catalan Numbers, Identical Plans Prof. Dr. Jens Dittrich
	14.516a Dynamic Programming: Core Idea, Requirements, Join Graph Prof. Dr. Jens Dittrich
	14.516b Dynamic Programming Example without Interesting Orders, Pseudo-Code Prof. Dr. Jens Dittrich
	14.516c Dynamic Programming Optimizations: Interesting Orders, Graph Structure Prof. Dr. Jens Dittrich

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