Anfrageoptimierung (Teil 1)

VL Big Data Engineering
(aka Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

9. Oktober 2020

CC BY-NC-SA

Die Vergangenen Wochen: Einfache Datenanalyse

1. Konkrete Anwendung: IMDB, NSA oder ähnliche Anwendungen

Was wir gelernt haben?
Fundamentale Konzepte der Anfrageverarbeitung:

- Relationales Modell
- Entity Relationship-Modellierung
- Relationale Algebra
- SQL

Frage

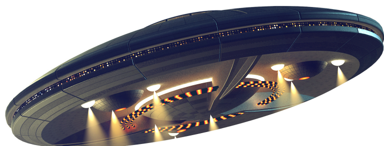
... und was ist, wenn die Daten größer werden? Wie kommen wir eigentlich von SQL zu einem effizienten Programm?

Übersicht über SQL-verarbeitende Systeme (SQL-Engines)

Sqlite

MySQL

PostgreSQL



Moderne Datenbanksysteme

CC BY-SA 2.0 Johann-Nikolaus Andreae, (c) istock.com Laspi/phive2015/3000ad

Was mache ich, wenn die Anfragen zu langsam sind?

Jetzt:

Frage 1

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

Automatische Anfrageoptimierung

1. SQL
   ↓ kanonische Übersetzung
2. annotierte relationale Algebra/logischer Plan
   ↓ heuristische Optimierung
3. transformierter logischer Plan
   ↓ kostenbasierte Optimierung
4. physischer Plan
   ↓ Code-Erzeugung
5. ausführbarer Code

Alle diese Optimierungen laufen intern ab: der Nutzer einer SQL-Engine muss sich **nicht** darum kümmern.

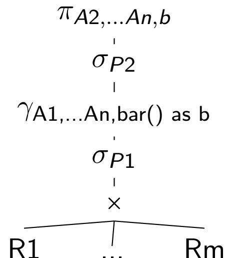
Je nach Anfrageoptimierer können einige dieser Übersetzungsschritte sehr simpel ausfallen, z.B. mit sehr einfachen Heuristiken oder auch ohne kostenbasierte Optimierung.

Kanonische Übersetzung

1. SQL
   ↓ kanonische Übersetzung
2. annotierte relationale Algebra/logischer Plan

Bei der kanonischen Übersetzung wird SQL in relationale Algebra übersetzt. Das Grundmuster ist das Folgende:

```
SELECT     A2,..,An, bar() AS b
FROM       R1,...,Rm
WHERE      P1
GROUP BY   A1,...,An     ↦
HAVING     P2
```



Kanonische Übersetzung

Weitere Schritte bei der kanonischen Übersetzung:

- Auflösung von Sichten, d.h. Textersetzung der dynamischen Sichtdefinitionen in Unteranfragen
- ggf. Ersetzen von Teilausdrücken durch vorberechnete bzw. vormaterialisierte (Teil-)ergebnisse (und materialisierte Sichten)

Beispiel:
exakt dieselbe Anfrage wurde schon berechnet UND die Datenbasis hat sich nicht geändert: gespeichertes Ergebnis zurückgeben ohne die Anfrage neu auszuführen

WAS vs WIE-Verwirrung

Achtung: WAS vs WIE-Verwirrung

Die relationale Algebra können wir sowohl deklarativ als auch prozedural lesen!

1. **deklarativ:** WAS ist das Ergebnis

   Die Reihenfolge der Operatoren in einem Ausdruck und wie die Operatoren implementiert sind, spielt keine Rolle.

   So haben wir es bisher gelesen.

2. **prozedural:** WIE berechnen wir das Ergebnis

   Wir können mindestens zwei verschiedene Sachen festlegen: Die Operatoren der relationalen Algebra werden in der Reihenfolge ausgeführt wie sie im Ausdruck stehen. Außerdem können wir entscheiden, wie welcher Operator implementiert wird.

   Diese prozedurale Leseweise gilt im Folgenden.

Logische vs Physische Operatoren

Bei der Einführung des relationalen Modells und von Operatoren hatten wir gesagt:

„

Achtung

Die Relationale Algebra beschreibt nur abstrakt **WAS** berechnet werden soll, aber nicht **WIE** diese Berechnung algorithmisch umgesetzt wird!

“

In der Implementierung im Notebook „Relational Algebra“ hatten wir das bereits unterschieden in logische und physische Operatoren.
In dem Notebook werden bereits beide Aspekte (Reihenfolge und Implementirung der Operatoren) berücksichtig (sonst lässt sich das auch nur schwer implementieren...).

Logische vs Physische Operatoren

Logisch	Physisch
$\bowtie$	nested-loops (Kreuzprodukt) plus Nachfiltern[1]
	index-basiert[2]
	... (siehe Stammvorlesung)
$\cap$	nested-loops (Kreuzprodukt) plus Nachfiltern
	index-basiert[3]
	...
σ $\mid$ R	Iteration/Scan plus Nachfiltern
	index-basiert?
	...

[1] vgl. Implementierung von Theta_Join im Notebook „Relational Algebra“

[2] vgl. Übungszettel 2, Aufgabe 3

[3] vgl. Implementierung von Intersection im Notebook „Relational Algebra“ durch Python set

Heuristische Optimierung

2. annotierte relationale Algebra/logischer Plan
   $\downarrow$ heuristische Optimierung
3. transformierter logischer Plan

Grundidee

- Die heuristische Optimierung transformiert den kanonischen Plan mithilfe von Regeln iterativ in einen (hoffentlich) besseren logischen Plan.
- Eine solche regelbasierte Transformation verändert **niemals** das Ergebnis des Plans.
- Es wird solange versucht, den Plan zu transformieren, bis keine der heuristischen Regeln mehr anwendbar ist.

Grundidee einer Regel (siehe rule.py)

Regel

match (OperatorTree op) ↦ bool
: Suchmethode, die überprüft, ob der Eingabebaum op eine bestimmte Teilstruktur, ein bestimmtes Muster hat.

modify (OperatorTree op) ↦ OperatorTree
: Transformationsmethode, die den Eingabebaum op verändert und veränderten Eingabebaum op' zurückgibt.

Beispiele:

match: gibt es eine Selektion mit einer Konjunktion im Prädikat?
modify: dann breche das Prädikat auf ersetzte die Selektion durch zwei separate Selektionen

match: befindet sich eine Selektion mit Joinprädikat direkt über einem Kreuzprodukt?
modify: dann schreibe Selektion und Kreuzprodukt zu einem Join um

Heuristische Optimierung: Grundalgorithmus

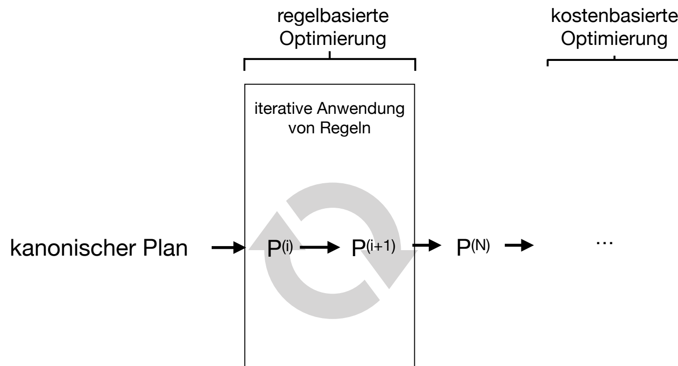
RuleOpt (Plan P = kanonischer Plan, Menge von Regeln R)

1. Solange irgendeine Regel r aus R auf P matched:
2. P = r.modify(P)

Erweiterungen:

- Abbruchbedingungen der Iteration:
  - maximale Anzahl an Iterationen deckeln
  - maximalen Aufwand für die Optimierung deckeln
  - Oszillation verhindern
- Priorität von Regeln einbauen (erst Regel 42, dann Regel 4)

Regelbasierter Optimierung



Die Mutter aller Transformationsregeln: Predicate Pushdown

Regel 1: Predicate Pushdown (Joins und Kreuzprodukte)

Seien R_1 und R_2 Relationen, p ein Prädikat, das nur Bedingungen an R_2 hat, und $OP = \{\bowtie, \times\}$ Dann gilt $\forall \oplus \in OP$:

$$\sigma_p(R_1 \oplus R_2) = R_1 \oplus \sigma_p(R_2)$$

Regel 2: Predicate/Projection Pushdown (allgemein)

„Selektionen und Projektionen sollten so nah wie möglich zu den Datenquellen, d.h. den Wurzeln des Baumes heruntergedrückt werden.“

Beispiele für Predicate Pushdown

- Sensordaten: direkt beim Messen der Daten Filterbedingungen auswerten
- Mail (IMAP): auf Server filtern vs Mails auf Client filtern
- Smart Disks: nicht-qualifizierende Tupel direkt beim Lesen vom Speichermedium verwerfen
- Verteilte Systeme: Filter auf entferntem System auswerten, dann statt aller Daten nur gefilterte Ergebnisse schicken
- Satellitenbilder: nur Bilder zur Erde schicken, die Filterprädikat erfüllen

Beispiel für Projection Pushdown

Dies bitte nicht verwechseln mit predicate Pushdown.

[R] : {[id: int, a:int, b:int]},
[S] : {[id: int, r_id: int, c:int, d:int]}.

Der ursprüngliche Ausdruck
$\pi_{a,c}\big(R \bowtie_{R.id=S.r_id} S\big)$

wird umgeschrieben zu:
$\pi_{a,c}\big(\pi_{id,a}(R) \bowtie_{R.id=S.r_id} \pi_{r_id,c}(S)\big)$

In dem Beispiel werden **zusätzliche** Projektionen eingeführt, die Attribute möglichst früh wegprojizieren. Dabei muss aufgepasst werden, dass Attribute, die für die Anfrageverarbeitung notwendig sind, nicht zu früh wegprojiziert werden.

In diesem Beispiel: $R.id$ und $S.r_id$.

Wichtige Transformationsregeln

Regel 3: Konjunktionen können aufgebrochen werden

Seien p und q Prädikate und R eine Relation. Dann gilt:
$\sigma_{p \wedge q}(R) = \sigma_p(\sigma_q(R))$.

Dies ermöglicht dann Pushdown individueller Selektionen (Regel 1 und 2).

Wichtige Transformationsregeln

Regel 4: Zusammenfassung von Selektion mit Kreuzprodukten&Joins

Seien R und S Relationen und P ein Prädikat. Für den Ausdruck $\sigma_P(R \times S)$ gilt: falls P ein Joinprädikat der Gestalt $P(x, y) := x$ op y ist, wobei x ein Attribut aus R und y ein Attribut aus S und op eine beliebige Vergleichsoperation ist, dann gilt: $\sigma_P(R \times S) = R \bowtie_P S$.

Ebenso gilt dann $\sigma_P(R \bowtie_{P_2} S) = R \bowtie_{P_2 \wedge P} S$.

Dies ist im Grunde eine Variante von Regel 2 (Predicate Pushdown): Anstatt erst das gesamte Kreuzprodukt zu erzeugen und nachzufiltern, wird der Filter direkt beim Erzeugen der Tupel berücksichtigt.

Regel 5: Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ und assoziativ.

Seien R_1, R_2, R_3 Relationen. Sei $OP = \{\bowtie, \cup, \cap, \times\}$. Dann gilt $\forall \oplus \in OP$

1. $\oplus$ ist kommutativ: $R_1 \oplus R_2 = R_2 \oplus R_1$, und
2. $\oplus$ ist assoziativ: $R_1 \oplus (R_2 \oplus R_3) = (R_1 \oplus R_2) \oplus R_3$.

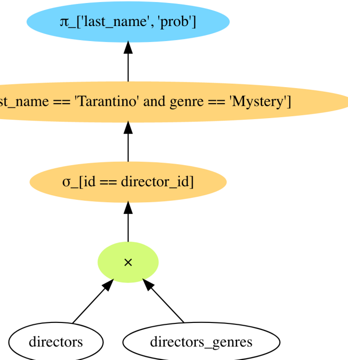
Heuristische Optimierung in Python

In [4]:

```
# Unoptimized plan
cp = Cartesian_Product(directors, directors_genres)
sel1 = Selection(cp, "id == director_id")
sel2 = Selection(sel1, "last_name == 'Tarantino' and genre == 'Mystery'")
proj = Projection(sel2, ['last_name', 'prob'])

graph = proj.get_graph()
Source(graph)
```

Out[4]:



github: Rule-based Optimization.ipynb

Kostenbasierte Optimierung

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

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

Grundbegriffe: Selektivität

Selektivität

Mit Selektivität bezeichnen wir für einen unären Operator das Verhältnis der Größe der Ausgaberelation zur Eingaberelation:
$sel = \frac{|\text{Ausgaberelation}|}{|\text{Eingaberelation}|} \leq 1$.

Beispiel:
$|R| = 1.000.000$, $|\sigma_p(R)| = 1.000$, $|\sigma_q(R)| = 500.000$

Hohe Selektivität („viele Tupel fallen weg“):
Selektivität von $\sigma_p(R)$: $\frac{|\sigma_p(R)|}{|R|} = \frac{1.000}{1.000.000} = 0.001 = 0.1\%$

Geringe Selektivität („wenige Tupel fallen weg“):
Selektivität von $\sigma_q(R)$: $\frac{|\sigma_q(R)|}{|R|} = \frac{500.000}{1.000.000} = 0.5 = 50\%$

Selektivität

Regel 6: Selektionen sind untereinander vertauschbar

Seien p und q Prädikate und R eine Relation. Dann gilt:
$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$.

Dies folgt direkt aus Regel 3 und dem Kommutativgesetz.
Es spielt eine Rolle, wenn ein Prädikat viel selektiver ist als das andere.

Beispiel:
$|R| = 1.000.000$, $|\sigma_p(R)| = 1.000$, $|\sigma_q(R)| = 500.000$

Geringe Selektivität $\sigma_p(\sigma_q(R))$:
Hier filtert σ_q erst 1.000.000 Eingabetupel zu 500.000 Ausgabetupeln.
Dann filtert σ_p 500.000 Eingabetupel.
In der Summe werden 1.500.000 Tupel überprüft (≈**Ausführungskosten**)

Hohe Selektivität $\sigma_q(\sigma_p(R))$:
Hier filtert σ_p erst 1.000.000 Eingabetupel zu 1.000 Ausgabetupeln.
Dann filtert σ_q 1.000 Eingabetupel.
In der Summe werden 1.001.000 Tupel überprüft (≈**Ausführungskosten**)

Kostenmodelle

Kostenmodell

Ein Kostenmodell schätzt für einen gegebenen (Teil-)Plan die Ausführungskosten mithilfe eines geeigneten Modells ab. Für beliebige Pläne P definieren hierfür eine Funktion: $\text{Kosten}(P) \mapsto \text{float}$.

Beispiel: Im Beispiel oben haben wir argumentiert, dass die Auswertungsreihenfolge $\sigma_q(\sigma_p(R))$ besser sei, weil bei der Ausführung weniger Tupel überprüft werden müssen:

Kostenmodell: Zwischenergebnisse

$\text{Kosten}_{\text{Zwischenergebnisse}}(P) :=$ Summe über die Kardinalitäten der Ausgaberelationen aller Operatoren in P.

Dieses Kostenmodell schätzt die Gesamtkosten eines Plans anhand der Größe der Zwischenergebnisse ab. Kosten für die einzelnen Operatoren (Was ist die Laufzeit des Joins?) oder den Zugriff auf bestimmte Relationen (muss ich das aus dem Netzwerk oder von SSD lesen?) werden in diesem Kostenmodell **nicht** berücksichtigt!

Weitere Kostenmodelle

Kostenmodell: I/O-Kosten

$\text{Kosten}_{\text{I/O}}(P)$:= Summe über alle Zugriffszeiten/Lesezeiten von Festplatte/SSD/Netzwerk über alle Eingaberelationen.

Kostenmodell: Gesamtlaufzeit

$\text{Kosten}_{\text{Gesamtlaufzeit}}(P)$:= Gesamtlaufzeit des Plans in Sekunden.

Kostenmodell: Energieverbrauch

$\text{Kosten}_{\text{Energieverbrauch}}(P)$:= Energie, die benötigt wird, um den Plan auszuführen.

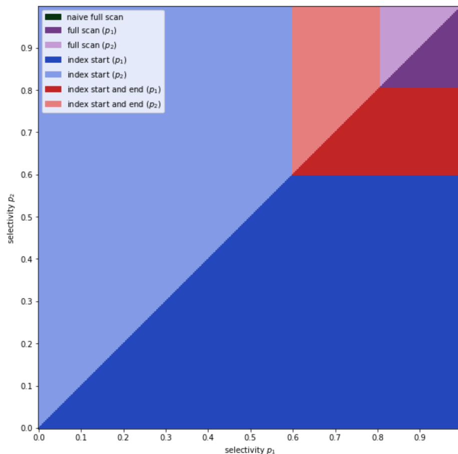
Kostenmodell: Hauptspeicher

$\text{Kosten}_{\text{Hauptspeicher}}(P)$:= Benötigter Hauptspeicher, um den Plan ausführen zu können.

Picasso Plan-Diagramme in Python

In [35]:
```
fig, ax = plt.subplots(figsize=(10, 10))

plot_plan_diagram(plans, plan_labels, 300, ax, color_list)
```



https://github.com/BigDataAnalyticsGroup/bigdataengineering/blob/master/Picasso.ipynb

Modell vs Realität

Falsche Modelle

All models are wrong, but some are useful.
[*George Box*]

- Verwechseln Sie **niemals** Modell und Realität!
- Gleichen Sie regelmäßig Modell und Realität ab!
- Ein Modell hat immer einen Zweck, einen bestimmten Nutzen.
- Bestimmte Dinge werden im Modell simplifiziert oder weggelassen, um andere herauszuarbeiten.
- Das ist das Wesen von:

Abstraktion

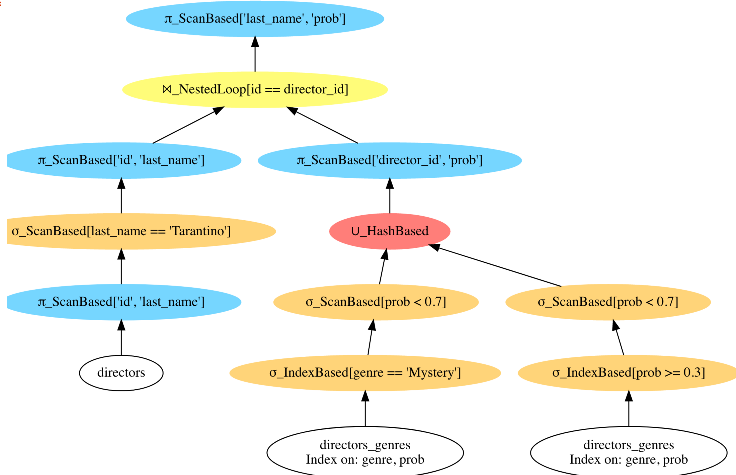
„Denkprozess des Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres."

- Wikipedia: All models are wrong-Historie
- Wikipedia: Abstraktion

Code-Erzeugung

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

Code-Erzeugung und Indextransformation in Python



github: Rule-based Optimization.ipynb