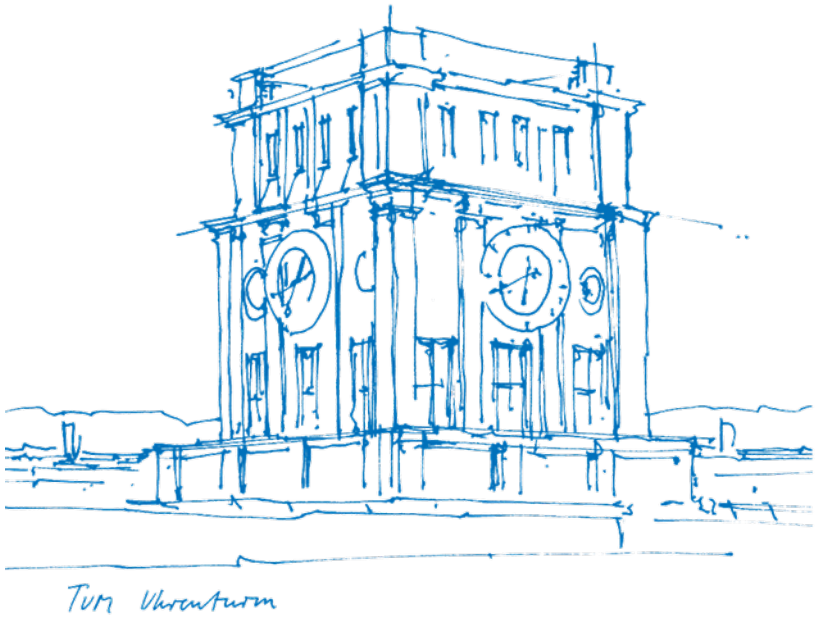
Grundlagen Datenbanken

Benjamin Wagner

31. Januar 2019



Allgemeines

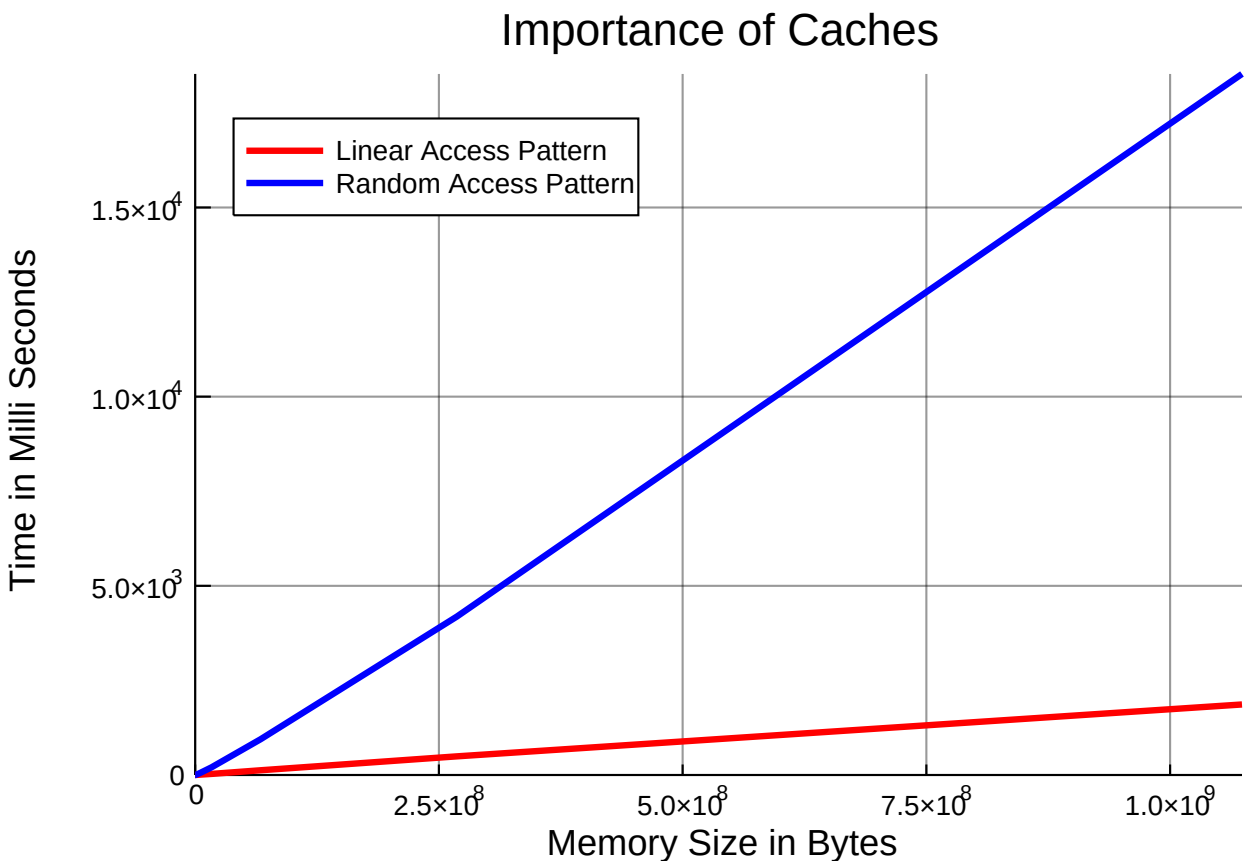
- Folien von mir sollen unterstützend dienen. Sie sind nicht von der Übungsleitung abgesegnet und haben keinen Anspruch auf Vollständigkeit (oder Richtigkeit).
- Bei Fragen oder Korrekturvorschlägen: wagnerbe@in.tum.de
- Vorlesungsbegleitendes Buch von Professor Kemper (Chemiebib)
- Mein Foliensatz ist online: `https://github.com/wagjamin/GDB2018`

Speicherhierachie

- Moderne Rechner haben verschiedene Arten des Speichers
- Dieser ist hierarchisch angeordnet: größerer Speicher ist langsamer
- Bei Festplatten kann zusätzlicher Aufwand entstehen (z.B. disk seek)

Speicher	Größe	Latenz	Vergleich
Register	bytes	1ns	Schreibtisch
Cache	K-M bytes	<10ns	Zimmer
Hauptspeicher	G bytes	<100ns	Nachbarschaft
Externer Speicher	T bytes	1ms	Tokyo

Caches



Zugriff auf Daten in linearer oder zufälliger Reihenfolge; Quelle: selbstgeschrieben

Anfrageoptimierung

- Wenn wir eine Anfrage an die Datenbank stellen, muss diese abgearbeitet werden
- Die Datenbank muss einen Plan erstellen, wie sie die Anfrage abarbeiten soll
- Dies wird in relationaler Algebra ausgedrückt
- **Aber:** verschiedene Algebrabäume können logisch äquivalent sein

$\Rightarrow$ Die Datenbank muss einen möglichst effizienten Plan finden

Anfrageoptimierung - Erste Idee

- Beginn: übersetzen die Anfrage in einen beliebigen relationalen Plan
- Diesen Plan können wir dann mit Umformungsregeln modifizieren
- **Frage:** was sind Beispiele für solche Regeln?
- Suche Heuristiken, welche Regeln zu "billigeren"Plänen führen
- Wunsch: Zwischenergebnisse sollen klein sein

Anfrageoptimierung - Erste Idee

- Beginn: übersetzen die Anfrage in einen beliebigen relationalen Plan
- Diesen Plan können wir dann mit Umformungsregeln modifizieren
- **Frage:** was sind Beispiele für solche Regeln?
- Suche Heuristiken, welche Regeln zu "billigeren"Plänen führen
- Wunsch: Zwischenergebnisse sollen klein sein

1. Selektionen aufbrechen
2. Selektionen nach unten schieben
3. Kreuzprodukte & Selektionen zu Joins verschmelzen
4. Joinreihenfolge bestimmen (klein: links, groß: rechts)

Korrelierte Unteranfragen

- Warum ist folgende Anfrage bei der Ausführung potentiell langsam?

```
1 SELECT s.Name, p.VorlNr
2 FROM Studenten s , prüfen p
3 WHERE s.MatrNr = p.MatrNr AND p.Note = (
4   SELECT MIN(p2.Note)
5   FROM prüfen p2
6   WHERE s.MatrNr=p2.MatrNr )
```

- **Frage:** wie lösen wir dieses Problem?

Ausführung der Anfrage

- Wir haben nun einen (hoffentlich) guten Query-Plan
- Dieser muss von der Datenbank ausgeführt werden
- Klassisch: jeder Operator wird implementiert
- Interface eines Operators: *open, close, next*
- Die Tupel werden dann bottom-up durch den Iteratorbaum geschleust
- **Ausblick:** das ist leider sehr ineffizient, Hyper kompiliert stattdessen den Operatorbaum in Maschinencode
- Wir müssen nun noch die einzelnen Operatoren implementieren

Joins - Algorithmen

- Der Join ist einer der wichtigsten Operatoren

$\Rightarrow$ Wir brauchen eine effiziente Implementierung!

- **Frage:** welche Join-Algorithmen gibt es?

Joins - Algorithmen

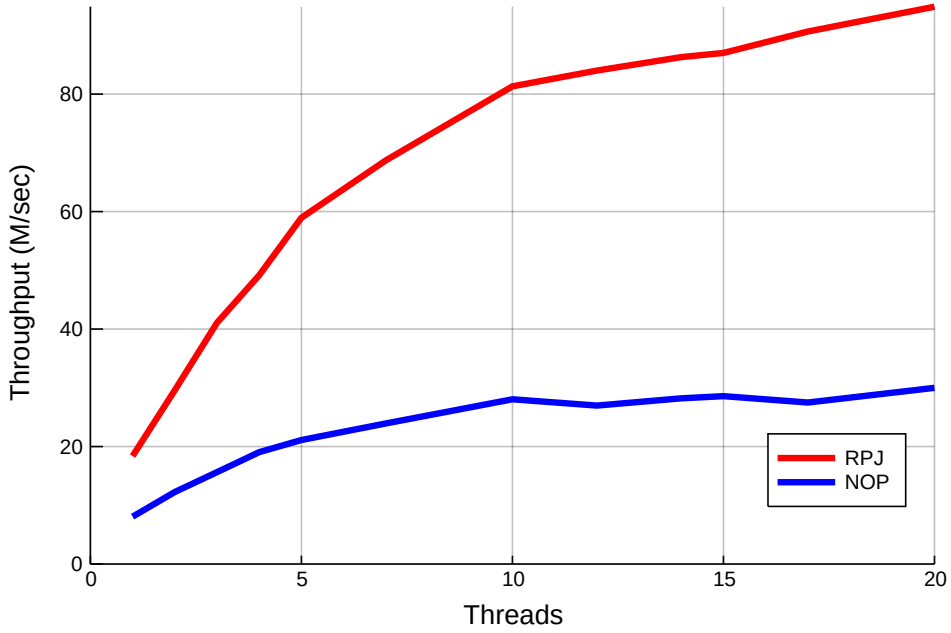
- Der Join ist einer der wichtigsten Operatoren

$\Rightarrow$ Wir brauchen eine effiziente Implementierung!

- **Frage:** welche Join-Algorithmen gibt es?
- Nested-Loop-Join, Hash-Join, Sort-Merge-Join
- In Hauptspeicherdatenbanken sind i.d.R. Hash-Joins am schnellsten
- **Aber:** selbst für Hash-Joins gibt es viele Implementierungen

Haupspeicher Hash Joins - Algorithmen

- Zwei konkurrierende Hash Join Implementierungen
- Gleiche Daten, alles im Hauptspeicher



S uniformly distributed, $|R| = |S| \approx$16.8M. Quelle.

Optimierung Revisited

- Bisher: Heuristiken zur Optimierung von Anfragebäumen
- Haben gesehen: Join-Reihenfolge kann zu sehr unterschiedlichen Laufzeiten führen
- Optimierte Join-Algorithmen können schlechte Pläne nicht verstecken
- Heuristiken können auch zu sehr schlechten Plänen führen
- **Ziel:** suche Algorithmen, welche bessere Pläne generieren

Selektivität

- Müssen quantifizieren, ob Operatoren große oder kleine Zwischenergebnisse generieren
- Die Selektivität beschreibt, welcher Anteil von Tupeln die in den Operator gehen, auch im Ergebnis landen
- Selektivität von $\sigma_p := \frac{|\sigma_p(R)|}{|R|}$
- Selektivität von $\bowtie := \frac{|R \bowtie S|}{|R \times S|}$
- **Frage:** was sind die Selektivitäten von $\sigma_{true}, \sigma_{false}$
- **Frage:** in R.a sind Werte gleichverteilt in $\{1, 2, ..., n\}$. Was ist die erwartete Selektivität von $\sigma_{R.a=1}(R)$

Kostenabschätzung

- Können nun abschätzen, wie groß Zwischenergebnisse sind
- Müssen nun noch Kosten für eigentliche Ausführung schätzen
- Hier helfen Kostenmodelle für die einzelnen Operatoren
- Diese können verschiedene Parameter haben und verschieden Aufwändig sein
- **Frage:** was sind sinnvolle Parameter beim Abschätzen einer Selektion?

Optimierung - Algorithmen

- Für einzelne Operatorbäume können wir nun gut schätzen, wie teuer die Ausführung ist
- Ein Algorithmus zum Bestimmen der Joinreihenfolge kann nun also einen guten Plan wählen
- **Problem:** der Suchraum der möglichen Anfragepläne ist in der Regel zu groß

$\Rightarrow$ müssen Algorithmen finden, welche Suchraum einschränken

- **Ausblick:** Vorlesung „Query Optimization“

Dynamische Programmierung

- Klassischer Optimierungsalgorithmus
- Idee: größere optimale Lösungen lassen sich aus kleineren optimalen Lösungen generieren
- **Frage:** inwieweit ist das bei Joins sinnvoll?
- **Idee:** baue Pläne Bottom-Up aus möglichst guten Teilplänen zusammen
- Level =1: bestimme Zugriffsmethode auf Relationen
- Level >1: verbinde Ergebnisse des vorherigen Levels
- Entferne in jedem Schritt „schlechte“ Zwischenpläne (engl. pruning)

Algorithmus 1: Optimierung Dynamische Programmierung

Data: Relationen $R_1, R_2, \ldots, R_n$

Result: Guter Anfrageplan

tabelle = $\emptyset$;

for $i \in \{1, 2, \ldots, n\}$ **do**

  tabelle = tabelle $\bigcup$ zugriffe(R_i);

  prune($\{R_i\}$);

end

for $i \in \{2, 3, \ldots, n\}$ **do**

  for $S \subseteq \{R_1, \ldots, R_n\}$ *mit* $|S| = i$ **do**

    tabelle[S] = $\emptyset$;

    for $O \subset S$ **do**

      tabelle[S] = tabelle[S] $\bigcup$ plan(tabelle[O], tabelle[$S \setminus O$]);

    end

    prune(S);

  end

end

prune($\{R_1, R_2, \ldots, R_n\}$);

return *tabelle*[$\{R_1, R_2, \ldots, R_n\}$]

Transaktionen

- Transaktionen verpacken eine Folge von Operationen
- In einer Datenbank kann z.B. jede SQL-Anfrage als Transaktion modelliert werden
- Genereller Ablauf: **BOT** $\rightarrow$ Operationen $\rightarrow$ **C oder A**
- Datenbanken haben hohe Anforderungen an Datenintegrität
- Wir können Transaktionen nicht komplett frei arbeiten lassen
- Datenbank muss **ACID**-Forderungen erfüllen
- **Ausblick:** Vorlesung „Transaction Systems“