

7. Anfrageoptimierung

Ausführen einer SQL-Anfrage

Klauseln und ihre Reihenfolge

Die Klauseln einer SQL-Anfrage 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`

Ausführungsreihenfolge einer SQL-Anfrage

Die Anfrage wird jedoch in einer *anderen Reihenfolge* ausgeführt, als sie angegeben wird:

1. **Kartesisches Produkt** der Tabellen in der `FROM` -Klausel.
 - *Erklärung:* Es werden alle möglichen Kombinationen von Zeilen aus den angegebenen Tabellen gebildet. Wenn z.B. Tabelle A 3 Zeilen und Tabelle B 4 Zeilen hat, entstehen 12 Zeilen im kartesischen Produkt.
2. Anwendung der **Prädikate** in der `WHERE` -Klausel.
 - *Erklärung:* Hier werden Zeilen herausgefiltert, die die angegebene Bedingung nicht erfüllen.
3. Anwendung der `GROUP-BY` -Klausel.
 - *Erklärung:* Die verbleibenden Zeilen werden zu Gruppen zusammengefasst, basierend auf den Werten in den `grouping column(s)`.
4. Anwendung der **Prädikate** in der `HAVING` -Klausel (um Gruppen zu eliminieren).
 - *Erklärung:* Dies filtert Gruppen basierend auf einer Bedingung, die sich oft auf Aggregatfunktionen bezieht.
5. Berechnung der **Aggregatfunktionen** für jede verbleibende Gruppe.
 - *Erklärung:* Funktionen wie `SUM()`, `AVG()`, `COUNT()`, `MAX()`, `MIN()` werden auf die jeweiligen Gruppen angewendet.
6. **Projektion** auf Spalten der `SELECT` -Klausel.
 - *Erklärung:* Es werden nur die Spalten ausgewählt und angezeigt, die in der `SELECT` -Klausel spezifiziert sind.
7. Anwendung der `ORDER-BY` -Klausel (nicht auf dem Slide explizit genannt, aber impliziert als letzter Schritt zur Sortierung des Ergebnisses).

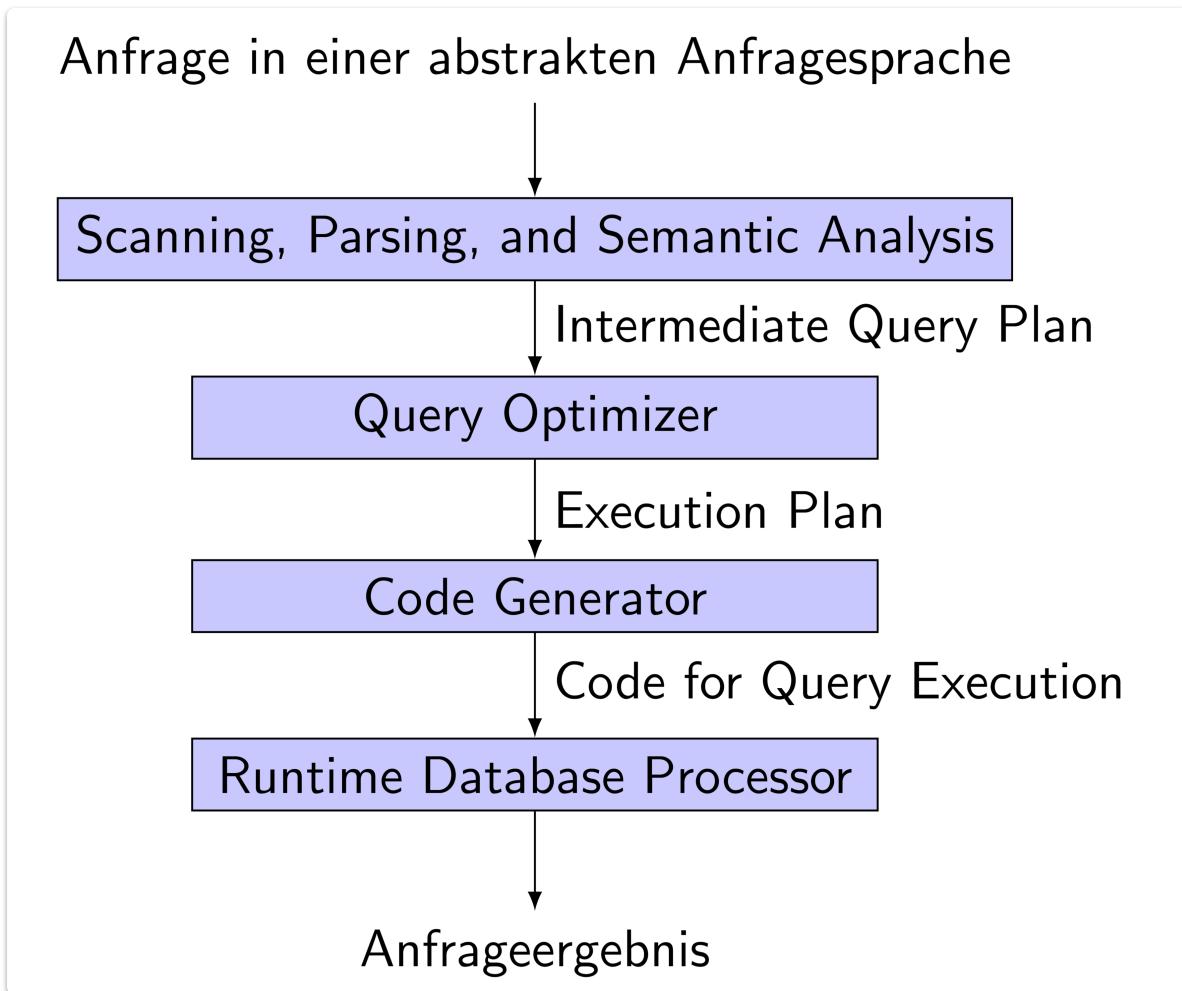
- **Erklärung:** Das endgültige Ergebnis wird nach den in `sort list` angegebenen Spalten sortiert.

SQL ist deklarativ!

- SQL ist eine **deklarative Sprache**.
 - Das bedeutet, man beschreibt *was* man möchte (das Ergebnis), aber nicht *wie* das Datenbankmanagementsystem (DBMS) es erreichen soll.
 - Das DBMS ist dafür verantwortlich, den effizientesten Weg zur Ausführung der Anfrage zu finden.

Schritte der Anfragebearbeitung

Die Verarbeitung einer Anfrage in einem Datenbanksystem durchläuft mehrere Schritte:



1. Anfrage in einer abstrakten Anfragesprache (z.B. SQL):

- Dies ist die vom Benutzer eingegebene Anfrage.

2. Scanning, Parsing, und Semantic Analysis:

- **Scanning (Lexikalische Analyse):** Der Anfragetext wird in Tokens (kleinste syntaktische Einheiten, z.B. Schlüsselwörter, Operatoren, Bezeichner) zerlegt.
- **Parsing (Syntaktische Analyse):** Die Tokens werden gemäß der Grammatik der Anfragesprache in einen **Parse Tree** (Syntaxbaum) oder eine **Anfragespezifikation** (Query Specification) übersetzt.

umgewandelt. Dabei wird geprüft, ob die Syntax korrekt ist.

- **Semantic Analysis (Semantische Analyse):** Hier wird die Bedeutung der Anfrage überprüft. Dazu gehören:
 - Prüfung, ob alle angegebenen Tabellen und Spalten existieren.
 - Typenprüfung (z.B. ob Vergleiche zwischen kompatiblen Datentypen stattfinden).
 - Auflösung von Namen und Aliassen.
 - Berechtigungsprüfung (ob der Benutzer die nötigen Rechte hat).
- Das Ergebnis dieses Schrittes ist ein **Intermediate Query Plan** (ein Zwischen-Anfrageplan, z.B. in Form eines Operatorbaums der relationalen Algebra).

3. Query Optimizer (Anfrageoptimierer):

- Dies ist eine der wichtigsten Komponenten eines DBMS.
- Der Optimierer nimmt den Intermediate Query Plan und sucht nach dem **effizientesten Ausführungsplan**.
- Es gibt oft mehrere Wege, dieselbe Anfrage zu beantworten (z.B. verschiedene Reihenfolgen von Joins oder Filteroperationen).
- Der Optimierer bewertet die Kosten der verschiedenen Pläne (z.B. geschätzte Anzahl der zu lesenden Blöcke, CPU-Kosten) und wählt den Plan mit den niedrigsten geschätzten Kosten aus.
- Das Ergebnis ist ein **Execution Plan** (Ausführungsplan).

4. Code Generator:

- Der Code Generator nimmt den optimierten Execution Plan und generiert den eigentlichen Code, der vom Datenbankprozessor ausgeführt werden kann.
- Dies kann maschinennaher Code oder eine Sequenz von Aufrufen von Laufzeitroutinen sein.

5. Code for Query Execution:

- Der generierte Code, bereit zur Ausführung.

6. Runtime Database Processor (Laufzeit-Datenbankprozessor):

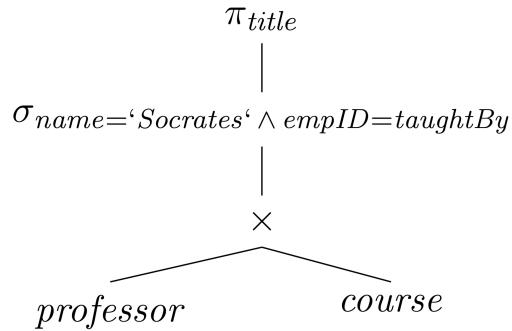
- Führt den generierten Code aus.
- Interagiert mit anderen Komponenten des DBMS (z.B. Puffermanagement, Speicherverwaltung, Transaktionsmanager), um die benötigten Daten zu lesen, zu verarbeiten und das Ergebnis zu liefern.

7. Anfrageergebnis:

- Das Endergebnis, das dem Benutzer präsentiert wird.

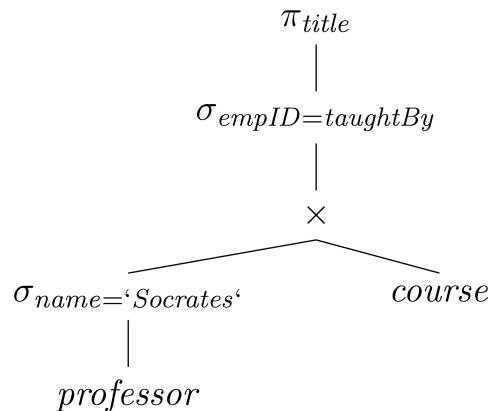
Alternativer Anfrageplan (Beispiel)

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



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

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



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

Anfrageoptimierung

Alternativen bei der Anfrageoptimierung

Der Anfrageoptimierer hat verschiedene Möglichkeiten, um einen optimalen Ausführungsplan zu finden:

- **Äquivalente Ausführungspläne:** Für dieselbe Anfrage gibt es oft mehrere Ausführungspläne, die dasselbe Ergebnis liefern. Der Optimierer wählt den kostengünstigsten.
- **Algorithmen zur Ausführung von Algebraoperatoren:** Für relationale Algebraoperatoren (z.B. Join, Selektion, Projektion) gibt es verschiedene Implementierungsalgorithmen (z.B. Nested-Loop Join, Hash Join, Sort-Merge Join). Der Optimierer wählt den passenden Algorithmus.

- **Methoden, um auf Relationen zuzugreifen (Indizes):** Es gibt verschiedene Möglichkeiten, auf Daten in Tabellen zuzugreifen. Ob ein Index verwendet wird oder ein vollständiger Tabellen-Scan, hängt von den Kosten ab.

Bei gleichem Ergebnis können Ausführungskosten sehr unterschiedlich sein.

Theorie vs. Realität

- **Es ist nicht die Aufgabe des Benutzers, "effiziente" Anfragen zu schreiben,** sondern die Aufgabe der Anfrageoptimierung, effiziente Ausführungspläne zu finden!
- Aber in der Realität... **Optimierer sind nicht perfekt.**
 - Sie basieren auf Heuristiken und Statistiken, die nicht immer 100% genau sind.
 - In komplexen Szenarien kann es vorkommen, dass der Optimierer nicht den absolut besten Plan findet.

Anfrageausführungskosten

Kostenmodell

Die **Gesamte Zeit bis das Anfrageergebnis vorliegt** wird als **Response Time** (Antwortzeit) bezeichnet. Viele Faktoren tragen zu dieser bei:

- **Festplattenzugriff (I/O-Kosten):**
 - Dies ist oft der **dominierende Faktor** bei den Kosten.
 - Daten müssen von der Festplatte in den Hauptspeicher geladen werden.
 - **Block Access Time:**
 - **Seek Time:** Die Zeit, die der Schreib-/Lesekopf benötigt, um zu der richtigen Spur auf der Festplatte zu gelangen.
 - **Rotation Time:** Die Zeit, die der benötigte Sektor benötigt, um unter dem Schreib-/Lesekopf vorbeizudrehen.
- **CPU-Kosten:**
 - Kosten für die Verarbeitung von Daten im Hauptspeicher (z.B. Sortieren, Hashen, Vergleichen).
- **Netzwerkkommunikation:**
 - Wenn Daten über ein Netzwerk abgerufen oder Ergebnisse an einen Client gesendet werden müssen.
- **Aktueller Query-Load:**
 - Die Anzahl und Art der gleichzeitig laufenden Anfragen kann die Performance beeinflussen (Konkurrenz um Ressourcen).
- **Parallelisierung:**
 - Wenn Operationen parallel ausgeführt werden können, kann dies die Antwortzeit verkürzen. Kosten für die Koordination.

Logische Anfrageoptimierung

Die logische Anfrageoptimierung befasst sich mit der Umformung des Anfrageplans auf einer höheren, **abstrakteren Ebene**, bevor physikalische Details berücksichtigt werden.

- **Relationale Algebra:**
 - Der initiale Anfrageplan wird oft in Operatoren der relationalen Algebra dargestellt.
 - Die logische Optimierung manipuliert diesen Operatorbaum.
- **Äquivalenzerhaltende Transformationsregeln:**
 - Dies sind Regeln, die es erlauben, einen relationalen Algebraausdruck in einen anderen umzuwandeln, ohne das Ergebnis der Anfrage zu verändern.
 - *Beispiel:* Das Vorziehen von Selektionen (Filtern) vor Joins ist oft effizienter, da die Datenmenge vor dem teuren Join reduziert wird.
- **Heuristische Optimierung:**
 - Dies sind "Faustregeln" oder Daumenregeln, die auf Erfahrungen basieren, um einen besseren Anfrageplan zu finden, ohne alle möglichen Pläne exakt zu kosten.
 - *Beispiel:* "Filter früh anwenden" ist eine typische Heuristik.

Physische Anfrageoptimierung

Die physische Anfrageoptimierung befasst sich mit der Auswahl der **konkreten Implementierungsstrategien** für die Operatoren des Anfrageplans und der Berücksichtigung der tatsächlichen Kosten.

- **Algorithmen und Operatorimplementationen:**
 - Hier werden die spezifischen Algorithmen für die relationalen Algebraoperatoren ausgewählt (z.B. welcher Join-Algorithmus verwendet wird: Nested-Loop Join, Hash Join, Sort-Merge Join).
 - Auch die Zugriffspfade auf die Daten werden festgelegt (z.B. Tabellen-Scan oder Index-Scan).
- **Kostenmodell:**
 - Ein Kostenmodell wird verwendet, um die geschätzten Ausführungskosten für verschiedene physische Pläne zu berechnen.
 - Dabei werden Faktoren wie CPU-Zeit, E/A-Operationen (Festplattenzugriffe) und Netzwerkkommunikation berücksichtigt.
 - Der Plan mit den geringsten geschätzten Kosten wird ausgewählt.

Logische (heuristische) Anfrageoptimierung

Logische Anfrageoptimierung

Grundlagen der Logischen Anfrageoptimierung

- **Grundlage: Äquivalenzerhaltende Transformationsregeln**
 - Diese Regeln sind zentral für die logische Optimierung. Sie erlauben es, einen Ausdruck der relationalen Algebra in einen anderen umzuformen, ohne das Ergebnis der Anfrage zu verändern.
 - Sie bilden den **Suchraum** der möglichen Anfragepläne.
- **Algebraische Transformationen** bilden den Suchraum.
- **Gegeben sei ein initialer algebraischer Ausdruck:**
 - Verwende äquivalenzerhaltende Transformationsregeln, um neue Ausdrücke abzuleiten.
 - Der Optimierer erkundet diesen Suchraum, um alternative Pläne zu finden.

Was ist ein guter Plan?

- Eine **genaue Entscheidung ohne Kostenfunktion ist nicht möglich**. Die logische Optimierung trifft noch keine konkrete Kostenabschätzung.
- Logische Anfrageoptimierung basiert auf **Heuristiken**.
 - Diese "Faustregeln" helfen, den Suchraum einzuschränken und vielversprechende Pläne zu identifizieren.

Hauptziel der logischen Anfrageoptimierung

- **Größe von Zwischenergebnissen reduzieren!**
 - Dies ist entscheidend, da kleinere Zwischenergebnisse weniger E/A-Operationen und CPU-Ressourcen benötigen und somit die Gesamtausführungszeit erheblich reduzieren.

Äquivalenzerhaltende Transformationsregeln

Diese Regeln ermöglichen es, relationale Algebraausdrücke umzuformen und dabei ihre Semantik (das Ergebnis) zu bewahren.

Aufbrechen von Konjunktionen in 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 ist 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

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

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

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

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:

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

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:

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

\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

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

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 für Operatoren der relationalen Algebra.

Phasen der logischen Anfrageoptimierung

Die logische Anfrageoptimierung durchläuft typischerweise mehrere Phasen, um einen effizienten Anfrageplan zu generieren. Das Hauptziel ist dabei immer, die Größe der Zwischenergebnisse so früh wie möglich zu reduzieren.

1. Aufbrechen von Selektionen:

- Konjunktive Selektionsprädikate (Bedingungen, die mit AND verbunden sind) werden in einzelne Selektionsoperationen aufgeteilt.
- *Beispiel:* WHERE A > 10 AND B = 'x' wird zu zwei einzelnen Selektionen.

2. Verschieben der Selektionen so weit wie möglich nach unten (pushing selections):

- Dies ist eine der wichtigsten Heuristiken. Selektionen (Filteroperationen) sollten so früh wie möglich im Ausführungsplan angewendet werden.
- Je früher Daten gefiltert werden, desto kleiner werden die Zwischenergebnisse, was die Kosten für nachfolgende Operationen (wie Joins) erheblich reduziert.
- *Beispiel:* SELECT * FROM R JOIN S ON R.id = S.id WHERE R.value > 10 ist effizienter, wenn R.value > 10 vor dem Join auf Tabelle R angewendet wird.

3. Joins einführen (Zusammenfassen von Selektionen und Kreuzprodukten):

- Kreuzprodukte (CROSS JOIN) gefolgt von Selektionen mit Gleichheitsbedingungen (WHERE R.id = S.id) werden in explizite Join-Operationen umgewandelt.
- Dies macht den Plan übersichtlicher und ermöglicht es dem Optimierer, spezifische Join-Algorithmen zu berücksichtigen.

4. Join-Reihenfolge bestimmen, so dass möglichst kleine Zwischenergebnisse entstehen:

- Die Reihenfolge, in der Joins ausgeführt werden, hat einen enormen Einfluss auf die Größe der Zwischenergebnisse und damit auf die Leistung der Anfrage.
- **Heuristik:** Joins mit Input von Selektionen vor anderen Joins auswerten.
 - Das bedeutet: Tabellen, die bereits durch Selektionen stark reduziert wurden, sollten bevorzugt gejoint werden, um die Datenmenge für die nachfolgenden Joins klein zu halten.

5. ggf. Einführen von Projektionen:

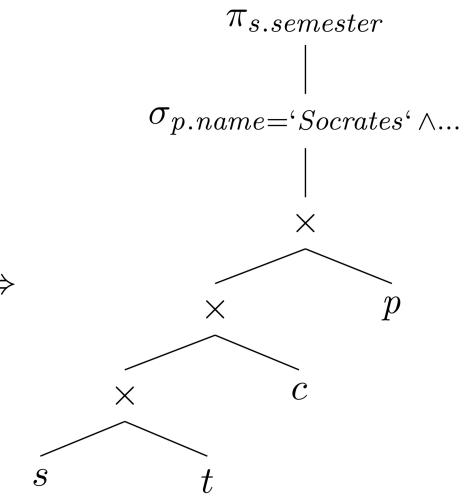
- Projektionen (SELECT-Klausel) wählen die benötigten Spalten aus. Manchmal müssen Projektionen eingefügt werden, um unnötige Spalten frühzeitig zu eliminieren.

6. Verschieben der Projektionen so weit wie möglich nach unten:

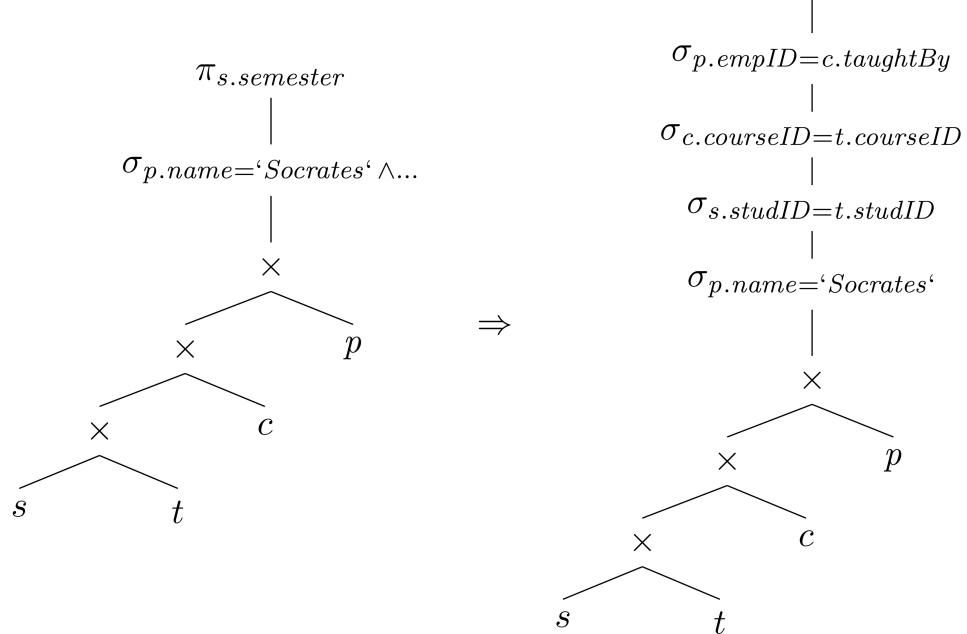
- Ähnlich wie bei Selektionen kann es vorteilhaft sein, Projektionen so früh wie möglich anzuwenden, um die Breite der Zwischenergebnisse zu reduzieren (d.h., weniger Spalten zu verarbeiten).
- **Nicht immer nötig:** Manchmal sind Projektionen nach Joins oder Aggregationen sinnvoller, insbesondere wenn die Projektionsspalten das Ergebnis von Aggregationen oder Join-Spalten sind, die erst später verfügbar werden. Die Regel aus der äquivalenzerhaltenden Transformation ($\pi_{A_1, \dots, A_n}(\sigma_C(R)) = \sigma_C(\pi_{A_1, \dots, A_n}(R))$) ist hier relevant.

Beispiel

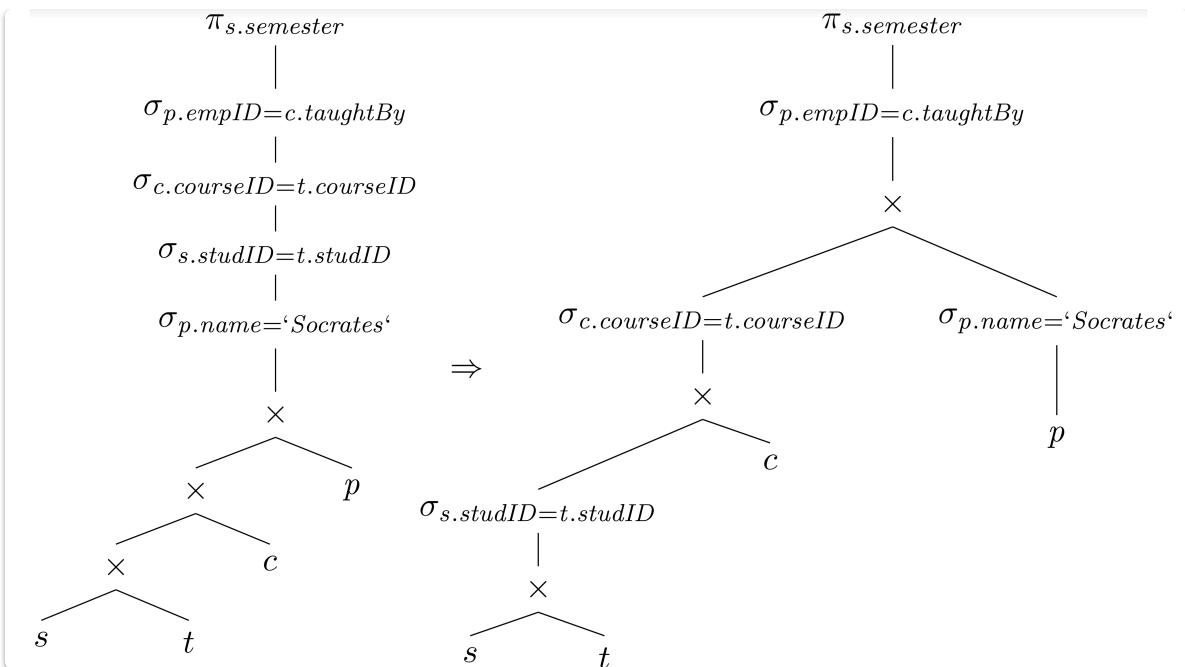
SELECT DISTINCT s.semester
 FROM student s, takes t,
 course c, professor p
 WHERE p.name='Socrates' AND
 c.taughtBy = p.empID AND
 c.courseID = t.courseID AND
 t.studID = s.studID;



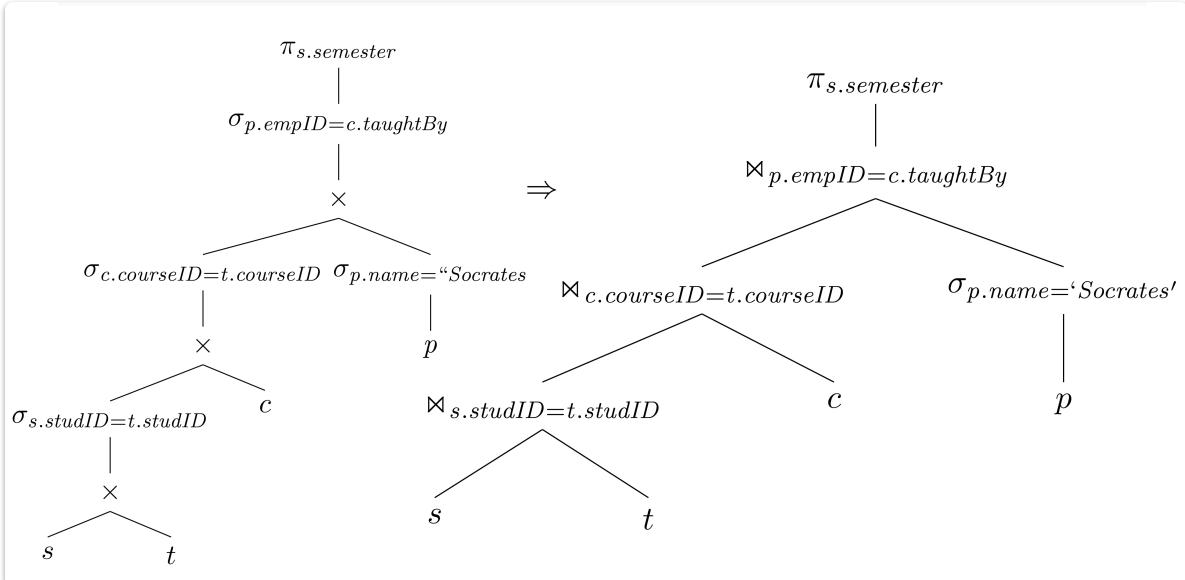
Aufbrechen von Selektionen



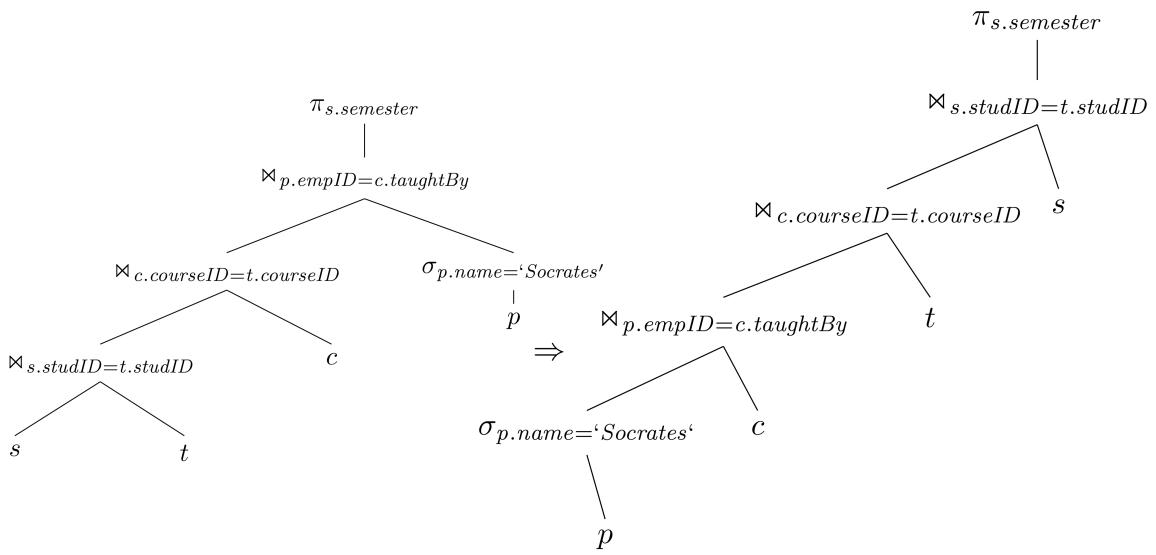
Verschieben von Selektionen



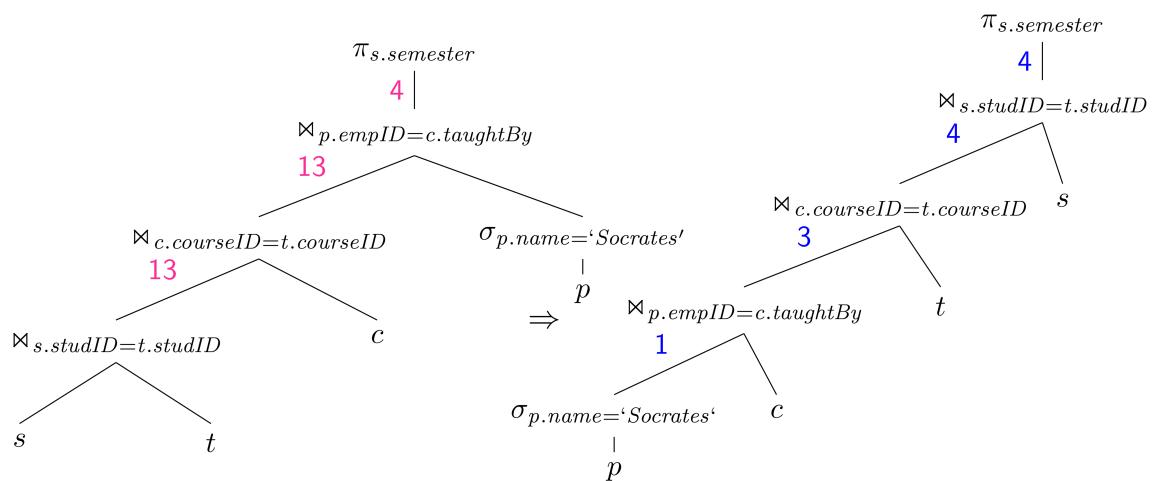
Joins einführen



Joinreihenfolge bestimmen

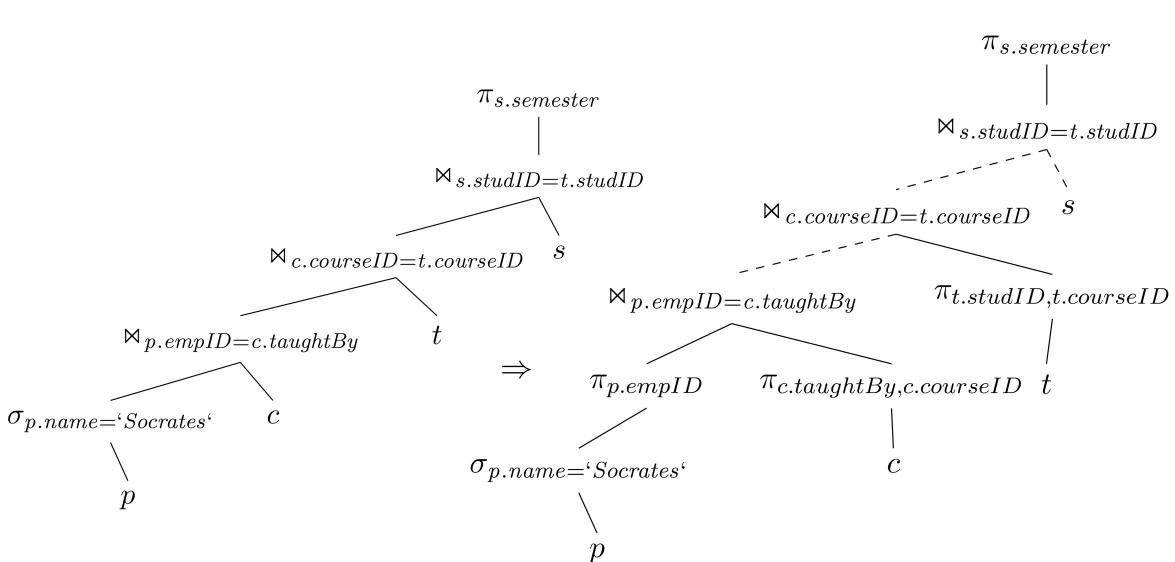


Effekt: Verkleinerung der Zwischenergebnisse



Schätzung der Zwischenergebnisgröße nur mit Statistiken möglich
→ Kostenmodelle

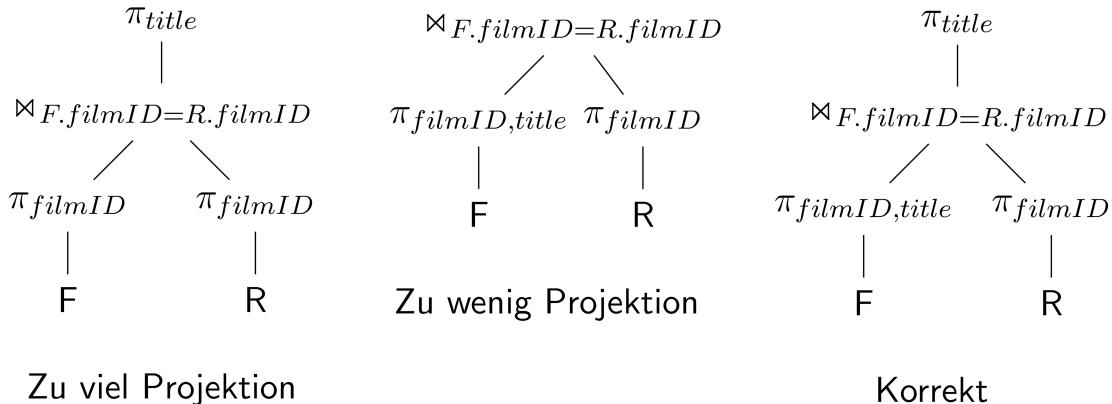
Einführen und verschieben von Projektionen



Vorsicht - Beispiele

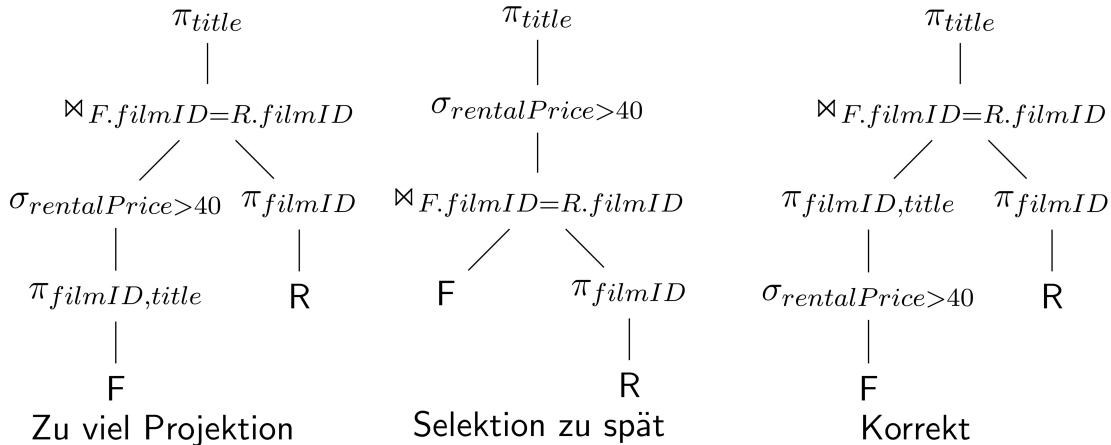
Finde die Titel von reservierten Filmen

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



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

Die heuristische Anfrageoptimierung konzentriert sich auf bewährte "Faustregeln", um effiziente Anfragepläne zu erstellen, ohne eine vollständige Kostenanalyse für alle möglichen Pläne durchzuführen.

Erfahrungsregeln

Die wichtigsten Heuristiken (Erfahrungsregeln) der logischen Anfrageoptimierung sind:

- Selektionen so früh wie möglich ausführen:

- Das Anwenden von Filtern (WHERE -Klausel) so früh wie möglich im Ausführungsplan reduziert die Anzahl der Zeilen in Zwischenergebnissen drastisch.
- Kleinere Zwischenergebnisse führen zu geringeren E/A- und CPU-Kosten für nachfolgende Operationen wie Joins.
- **Projektionen so früh wie möglich ausführen:**
 - Das Entfernen von unnötigen Spalten (SELECT -Klausel) so früh wie möglich im Ausführungsplan reduziert die Breite (Anzahl der Attribute) der Zwischenergebnisse.
 - Dies spart Speicherplatz und reduziert die Datenmenge, die zwischen den Operatoren bewegt werden muss.

Optimierungsprozess

Der allgemeine Prozess der heuristischen Anfrageoptimierung umfasst zwei Schritte:

1. **Erstelle initialen Plan aus der SQL-Anfrage:**
 - Die ursprüngliche SQL-Anfrage wird in einen ersten, meist naiven Operatorbaum der relationalen Algebra übersetzt (z.B. ein Parsen in einen Baum).
2. **Modifiziere den Plan, um ihn in einen effizienteren zu überführen:**
 - Äquivalenzerhaltende Transformationsregeln und die oben genannten Heuristiken werden angewendet, um den initialen Plan schrittweise in einen effizienteren Ausführungsplan umzuwandeln.

Hinweis

- **Anfrageergebnisse werden durch einen einzigen Plan berechnet:**
 - Obwohl es viele äquivalente Ausführungspläne geben kann, wählt der Optimierer letztendlich *einen* Plan aus, der dann zur Berechnung des Anfrageergebnisses verwendet wird.

Implementierung von Operatoren

Selektion (Access Paths)

Verschiedene Verwendungen von Select

- 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 - Punkt-/Bereichsanfragen

Bei der Selektion (dem Filtern von Daten) gibt es verschiedene grundlegende Suchstrategien, die je nach Datenstruktur und Art der Anfrage angewendet werden können:

- **Linear Search (Sequenzielle Suche):**
 - **Aufwändig:** Jedes Tupel (jede Zeile) der Relation muss gelesen und geprüft werden.
 - **Funktioniert aber immer:** Unabhängig davon, ob die Daten sortiert sind oder Indizes existieren.
- **Binary Search (Binäre Suche):**
 - **Nur wenn die Datei entsprechend sortiert ist:** Setzt voraus, dass die Daten physisch nach dem Suchkriterium sortiert sind.
- **Primary Hash Index:**
 - **Single Record Retrieval** – Funktioniert sehr effizient für den direkten Zugriff auf einzelne Datensätze über einen exakten Schlüsselwert (Gleichheitsanfragen).
 - **Funktioniert nicht für Bereichsanfragen:** Da Hash-Funktionen keine Ordnung herstellen.
- **Primary/Clustering Index:**
 - Mehrere Records für jeden Wert.

- Pointer zum Block mit dem ersten Record (Daten sind physisch nach Index sortiert).
- **Vorteil:** Ideal für Bereichsanfragen, da zugehörige Datenblöcke sequenziell gelesen werden können.
- **Secondary Index:**
 - Jeder Record hat einen eigenen Pointer.
 - **Kann teuer werden:** Insbesondere bei Bereichsanfragen, die viele Tupel zurückgeben, da dies zu vielen *zufälligen* E/A-Zugriffen auf der Festplatte führen kann.

Strategien für konjunktive Anfragen

Selektion (Access Paths)

- Beispiel einer SQL-Anfrage mit konjunktiven Bedingungen:

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

- Nutzung von Indizes bei konjunktiven Bedingungen:
 - Kann ein Index auf (*name*) oder (*street*) benutzt werden? **Ja** (für die jeweilige Bedingung)
 - Kann ein Index auf (*name, street, state*) benutzt werden? **Ja** (präfix-basiert, also wenn die ersten Spalten im Index vorkommen)
 - Kann ein Index auf (*name, street*) benutzt werden? **Ja** (präfix-basiert)
 - Kann ein Index auf (*name, street, city*) benutzt werden? **Ja** (hier wird der Index bis (*name, street*) genutzt, die *city*-Spalte im Index ist für diese Anfrage nicht relevant)
 - Kann ein Index auf (*city, name, street*) benutzt werden? **Nein** (da die Abfragebedingungen *name, street, state* sind und keine Präfix-Übereinstimmung mit *city* vorliegt, kann dieser Index nicht *effektiv* genutzt werden)

Optimierung von konjunktiven Anfragen

- Indizes bieten gute Möglichkeiten, die Performance von Anfragen zu verbessern. (Insbesondere bei konjunktiven Bedingungen, da sie den Zugriff auf die Daten beschleunigen können).

Strategien für konjunktive Anfragen (Fortsetzung)

- Existierende Indizes verwenden:
 - **Idealfall:** Es existiert ein Index, der alle Attribute abdeckt, die in der Anfrage benötigt werden.
 - **Falls es mehrere Indizes gibt:**

- Der Datenbankoptimierer wählt den Index, der am selektivsten ist (d.h. der die Anzahl der potenziellen Ergebnisse am stärksten reduziert).
- Die verbleibenden Bedingungen werden anschließend auf die durch den Index gefilterten Ergebnisse angewendet (ausgewertet).
- **Überschneidung von Pointern ausnutzen (Index Merge / Intersection):**
 - Diese Strategie wird angewendet, wenn mehrere Indizes auf verschiedene Attribute in einer konjunktiven Anfrage anwendbar sind.
 - **Schritte:**
 1. **Index Lookups:** Für jede Bedingung, für die ein Index existiert, werden die relevanten Pointer (Verweise auf die Datenblöcke/Tupel) ermittelt.
 2. **Überschneidung der Pointer:** Die Schnittmenge (Konjunktion) dieser Pointer wird gebildet. Das bedeutet, es werden nur die Pointer behalten, die in *allen* relevanten Index-Lookups vorkommen. (Also nur die Tupel, die *alle* Bedingungen erfüllen.)
 3. **Records/Tupel lesen:** Basierend auf den ermittelten, überschneidenden Pointern werden die zugehörigen Records/Tupel aus der Tabelle gelesen.

Disjunktive Anfragen (mit OR-Verknüpfungen)

- Disjunktive Anfragen bieten **wenig Gelegenheit zur Optimierung** durch Indizes im Vergleich zu konjunktiven Anfragen. (Oft muss hier ein Full Table Scan durchgeführt werden, es sei denn, die Indizes decken sehr spezifische Fälle ab oder es gibt eine Möglichkeit, die OR-Bedingungen in eine Reihe von UNION-Operationen umzuwandeln, die Indizes nutzen können.)

Wichtigkeit von Tuning

- **Tuning und das Anlegen von Indizes ist wichtig!** (Eine gute Indexstrategie ist entscheidend für die Performance von Datenbankabfragen.)

Join Algorithmen

Algorithmen

- **Nested Loop Join (Geschachtelter Schleifen-Join):** Ein grundlegender Join-Algorithmus, bei dem für jedes Tupel der äußeren Relation die innere Relation durchlaufen wird.
- **Index-based Join (Index-basierter Join):** Nutzt Indizes auf einer der Relationen, um den Join-Prozess zu beschleunigen.
- **Sort-Merge Join (Sortier-Misch-Join):** Beide Relationen werden nach dem Join-Attribut sortiert und anschließend in einem Merge-Schritt zusammengeführt.
- **Hash Join (Hash-Join):** Erstellt Hash-Tabellen für eine oder beide Relationen, um Tupel mit übereinstimmenden Join-Attributen schnell zu finden.

Strategien basieren auf Blöcken (nicht Tupeln) als Basis

- Bei der Kostenabschätzung von Join-Algorithmen geht man davon aus, dass Daten in Blöcken (Seiten) und nicht in einzelnen Tupeln gelesen werden.
- Schätze I/Os (Block Retrievals):** Die Kosten eines Join-Algorithmus werden primär durch die Anzahl der benötigten Ein-/Ausgabeoperationen (I/Os), d.h. das Lesen von Blöcken vom Speicher (z.B. Festplatte) in den Hauptspeicher, bestimmt.
- Benutze Puffer (Buffer) im Hauptspeicher:** Ein Puffer im Hauptspeicher wird genutzt, um die gelesenen Blöcke temporär zu speichern und die Anzahl der teuren Plattenzugriffe zu minimieren.

Tabellengröße und Join-Selektivität bestimmen die Kosten

- Selektivität einer Anfrage (*sel*):**

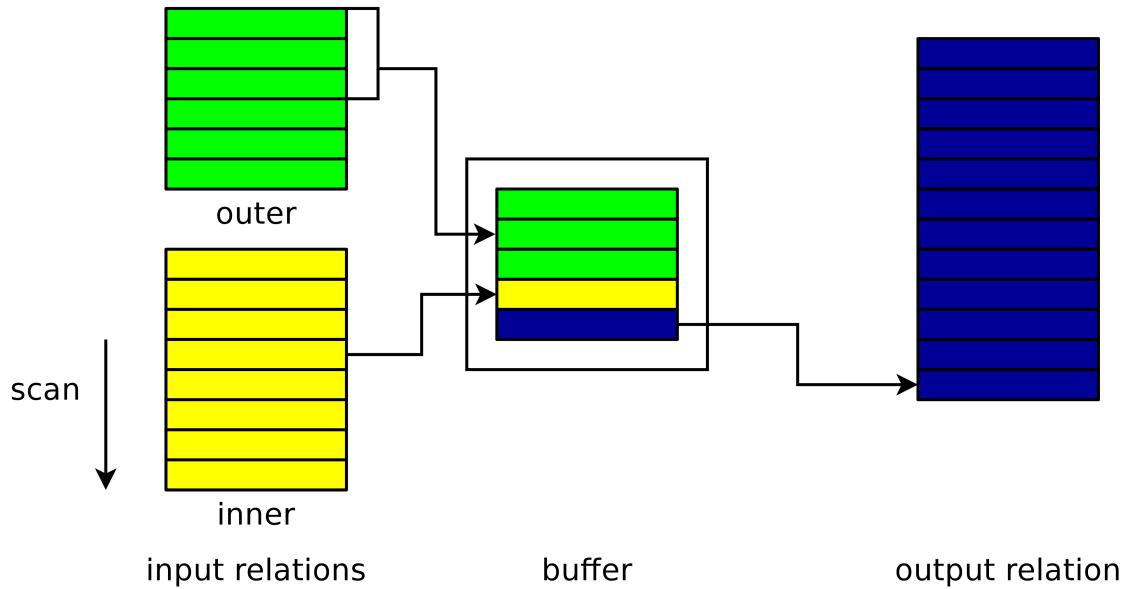
$$sel = \frac{\#\text{tuples in result}}{\#\text{candidates}}$$

- Beschreibt das Verhältnis der Anzahl der Tupel im Ergebnis zur Anzahl der potenziellen Tupel.
- Eine höhere Selektivität (näher an 1) bedeutet, dass ein größerer Anteil der Kandidaten in das Ergebnis aufgenommen wird. Eine geringere Selektivität (näher an 0) bedeutet, dass ein kleinerer Anteil der Kandidaten die Bedingung erfüllt.
- Für einen Join entspricht *#candidates* der Größe des Kartesischen Produkts:**
 - Das Kartesische Produkt zweier Relationen R und S ($R \times S$) ist die Menge aller möglichen Tupelpaare, die durch die Kombination jedes Tupels aus R mit jedem Tupel aus S entstehen.
 - Die Anzahl der Kandidaten für einen Join ist somit $|R| \times |S|$, also die Produkt der Anzahlen der Tupel der beiden Relationen.
 - Die Join-Operation filtert dann dieses Kartesische Produkt basierend auf den Join-Bedingungen.

Beispiel: Nested Loop Join

Sehr umfangreiches Beispiel von [DBS-9, p.37](#) bis [DBS-9, p.37](#)

Block Nested Loop Join (BNLJ)



- **Problemstellung:** Nicht alle Blöcke beider Relationen passen gleichzeitig in den Hauptspeicher. (Deshalb muss blockweise gelesen und verglichen werden.)
- **Algorithmus (Pseudocode):**

```

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 der Blöcke der inneren bzw. äußeren Relation.
 - n_B : Größe des Puffers im Hauptspeicher (Anzahl der Blöcke, die gleichzeitig in den Hauptspeicher geladen werden können).
- **Kostenschätzung (Anzahl der Block-Transfers / I/Os):**

Die geschätzten Kosten in Form von Block-Transfers für den Block Nested Loop Join sind:

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

- **Erklärung der Formel:**
 - b_{outer} : Dies sind die Kosten für das *einmalige* Lesen der äußeren Relation.
 - $\left\lceil \frac{b_{outer}}{(n_B - 2)} \right\rceil$: Dies ist die Anzahl der Iterationen der äußeren Schleife. In jeder Iteration werden $n_B - 2$ Blöcke der äußeren Relation gelesen. Wir ziehen 2 ab, da ein Block für die innere Relation und ein weiterer Block für die Ausgabe des Join-Ergebnisses benötigt werden. Die Ceil-Funktion ($\lceil \dots \rceil$) stellt sicher, dass auch der letzte, möglicherweise unvollständige Block-Batch berücksichtigt wird.
 - b_{inner} : In jeder Iteration der äußeren Schleife wird die innere Relation *vollständig* gelesen.

- **Berechnung der Berechnungszeit:**

Wenn weitere Systemparameter wie Block-Transfer-Zeiten, Disk Seek-Zeiten, CPU-Geschwindigkeit, etc. sowie die Größen der Relationen bekannt sind, können wir die gesamte Berechnungszeit detaillierter abschätzen.

Beispiel

- **Beispiel:** Join von `reserved` \bowtie `customer`

- **Anzahl der Blöcke:**

- $b_{\text{reserved}} = 2.000$ (Anzahl der Blöcke der Relation `reserved`)
- $b_{\text{customer}} = 10$ (Anzahl der Blöcke der Relation `customer`)

- **Größe des Buffers im Hauptspeicher:**

- $n_B = 6$ (Anzahl der Blöcke, die gleichzeitig im Hauptspeicher gehalten werden können)

- **Kostenschätzung (Block Transfers) - Wiederholung der Formel:**

$$b_{\text{outer}} + \left(\left\lceil \frac{b_{\text{outer}}}{(n_B - 2)} \right\rceil \cdot b_{\text{inner}} \right)$$

- **Kostenberechnung für das Beispiel:**

- **1. Fall:** `reserved` als äußere Relation ($b_{\text{outer}} = b_{\text{reserved}}$, $b_{\text{inner}} = b_{\text{customer}}$)

- $n_B - 2 = 6 - 2 = 4$
- Kosten = $2.000 + (\lceil 2.000/4 \rceil) \cdot 10$
- Kosten = $2.000 + (500) \cdot 10$
- Kosten = $2.000 + 5.000 = 7.000$ Block-Transfers

- **2. Fall:** `customer` als äußere Relation ($b_{\text{outer}} = b_{\text{customer}}$, $b_{\text{inner}} = b_{\text{reserved}}$)

- $n_B - 2 = 6 - 2 = 4$
- Kosten = $10 + (\lceil 10/4 \rceil) \cdot 2.000$
- Kosten = $10 + (2.5) \cdot 2.000$
- Kosten = $10 + (3) \cdot 2.000$
- Kosten = $10 + 6.000 = 6.010$ Block-Transfers

- **Ergebnis des Beispiels:** Es ist effizienter, die kleinere Relation (`customer`) als äußere Relation zu wählen, um die Anzahl der Block-Transfers zu minimieren.

Index-based Nested Loop Join

- **Gleiches Prinzip wie beim Standard Nested Loop Join:** Es gibt eine äußere und eine innere Relation.
- **Äußere Relation:** Wird sequenziell oder blockweise durchlaufen.
- **Innere Relation:** Der **File Scan** (komplettes Durchsuchen der Datei) der inneren Relation kann durch **Index Lookups** ersetzt werden. Das bedeutet: Für jedes Tupel der äußeren Relation wird der Index der inneren Relation verwendet, um passende Tupel schnell zu

finden, anstatt die gesamte innere Relation erneut zu scannen. Dies ist besonders effizient, wenn die innere Relation auf dem Join-Attribut indiziert ist.

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

Umfangreiches Beispiel

In den Slides von [DBS-9, p.43|Seite 126](#) bis [DBS-9, p.43|Seite 136](#)

Kosten von Merge Join

Parameter

- b_1, b_2 : Anzahl der Blöcke der beiden zu joinenden Relationen.

Kostenschätzung (Block Transfers)

- Wenn beide Relationen bereits nach dem Join-Attribut sortiert vorliegen, sind die Kosten für den Merge Join sehr gering:

$$b_1 + b_2$$

- Diese Kosten entstehen, weil jede Relation einmal sequenziell gelesen werden muss, um die Tupel zu mergen.

Erweiterungen (Zusätzliche Überlegungen)

- **Kombination mit Sortierung, wenn Input-Relationen nicht sortiert vorliegen:**
 - Der Merge Join setzt voraus, dass die Input-Relationen nach dem Join-Attribut sortiert sind.

- Falls sie es nicht sind, müssen sie zuerst sortiert werden. Die Kosten für diese Sortierung (oft externer Sortieralgorithmus) müssen dann zu den $b_1 + b_2$ Kosten hinzugaddiert werden. Diese Sortierkosten können erheblich sein.
 - **Nicht genügend Hauptspeicher:**
 - Wenn die Relationen zu groß sind, um vollständig in den Hauptspeicher geladen zu werden (was oft der Fall ist), müssen externe Sortier- und Merge-Verfahren angewendet werden. Dies erhöht die Komplexität und die I/O-Kosten des Algorithmus.

Hash Join

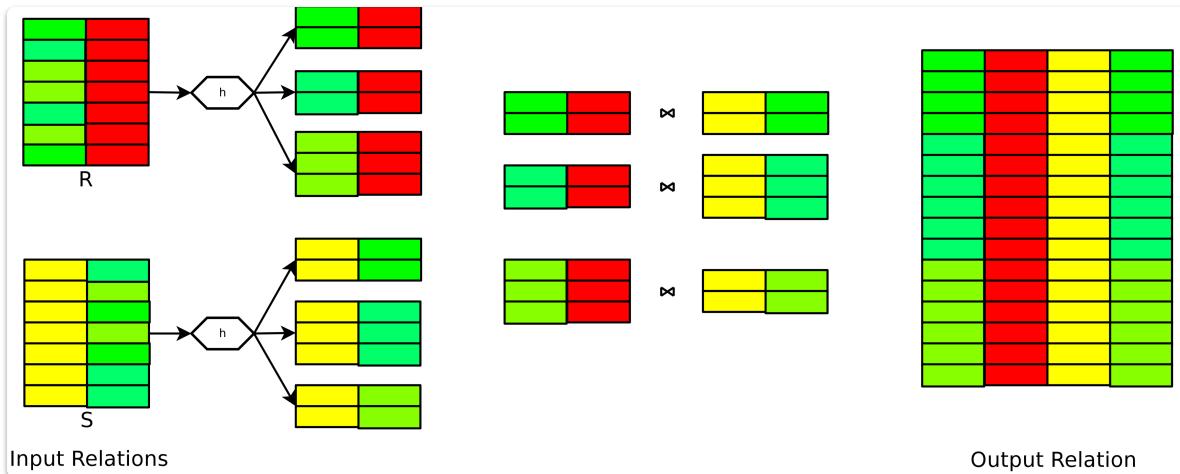
Wende Hashfunktion auf die Join-Attribute an
→ Partitioniere Tupel in Buckets

<table border="1"><tr><td>ID</td><td>name</td></tr><tr><td>15</td><td>Pete</td></tr><tr><td>21</td><td>Dave</td></tr></table>	ID	name	15	Pete	21	Dave	\bowtie	<table border="1"><tr><td>number</td><td>ID</td></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>	number	ID	120	15	160	15	170	21	$=$	<table border="1"><tr><td>ID</td><td>name</td><td>number</td></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>	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																												
emp ₀		phone ₀		result ₀																										
<table border="1"><tr><td>ID</td><td>name</td></tr><tr><td>10</td><td>Jim</td></tr><tr><td>13</td><td>Joe</td></tr></table>	ID	name	10	Jim	13	Joe	\bowtie	<table border="1"><tr><td>number</td><td>ID</td></tr><tr><td>110</td><td>10</td></tr><tr><td>150</td><td>13</td></tr></table>	number	ID	110	10	150	13	$=$	<table border="1"><tr><td>ID</td><td>name</td><td>number</td></tr><tr><td>10</td><td>Jim</td><td>110</td></tr><tr><td>13</td><td>Joe</td><td>150</td></tr></table>	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																												
emp ₁		phone ₁		result ₁																										
<table border="1"><tr><td>ID</td><td>name</td></tr><tr><td>14</td><td>Sue</td></tr><tr><td>23</td><td>Anne</td></tr></table>	ID	name	14	Sue	23	Anne	\bowtie	<table border="1"><tr><td>number</td><td>ID</td></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>	number	ID	100	23	130	23	140	23	$=$	<table border="1"><tr><td>ID</td><td>name</td><td>number</td></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>	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																												
emp ₂		phone ₂		result ₂																										

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

- Jede Relation mit Hashfunktion partitionieren:

- Die Input-Relationen R und S werden mithilfe einer Hashfunktion h (auf das Join-Attribut angewendet) in mehrere Buckets aufgeteilt.
- **Jedes Bucket muss klein genug sein, um in den Hauptspeicher zu passen:**
 - Die erzeugten Buckets sollen idealerweise in den Hauptspeicher passen, um I/O-Operationen zu minimieren.
- **Join die "passenden" Buckets miteinander:**
 - Nur Buckets, die denselben Hash-Wert aufweisen (d.h. von derselben Partition stammen), müssen miteinander gejoined werden. (Z.B. Bucket 1 von R mit Bucket 1 von S , Bucket 2 von R mit Bucket 2 von S , etc.)



Parameter

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

Schritte

1. Partitioniere Relation R_1 mit h_1 in Buckets r_1 :

- Lese R_1 komplett ein (`read all`).
- Schreibe R_1 nach Hashing in Buckets auf die Platte (`write all`).
- Kosten: $2 \times b_1$ (einmal lesen, einmal schreiben)

2. Partitioniere Relation R_2 mit h_1 in Buckets r_2 :

- Lese R_2 komplett ein (`read all`).
- Schreibe R_2 nach Hashing in Buckets auf die Platte (`write all`).
- Kosten: $2 \times b_2$ (einmal lesen, einmal schreiben)

3. Build-Phase:

- Für jedes Bucket von r_1 (aus der kleineren Relation):
 - Benutze eine Hashfunktion h_2 (oft eine einfache interne Hash-Tabelle) zur Erstellung eines In-Memory Hash Index.
 - Lese das Bucket r_1 (`read all`).
- Kosten: b_1 (einmaliges Lesen der partitionierten R_1 Buckets).

4. Probe-Phase:

- Für jedes Bucket von r_2 :

- Benutze den In-Memory Hash Index, der in der Build-Phase erstellt wurde, um Joinpartner zu finden.
- Lese das Bucket r_2 (read all).
- Kosten: b_2 (einmaliges Lesen der partitionierten R_2 Buckets).

Kostenschätzung (Block Transfers)

- Die Gesamtkosten für den Hash Join sind:

$$2 \cdot b_1 + 2 \cdot b_2 + b_1 + b_2 = 3 \cdot b_1 + 3 \cdot b_2$$

- Hier sind die Kosten für (unvollständig gefüllte Blöcke) mit ϵ nicht explizit aufgeführt, aber die Formel $3 \cdot b_1 + 3 \cdot b_2$ repräsentiert die Summe der Lese- und Schreiboperationen während der Partitionierungs- und Join-Phasen.
-

Kosten und Anwendung von Join-Algorithmen

Nested Loop Join

- Kann für alle Join-Typen verwendet werden. (Sehr flexibel, aber oft ineffizient).
- Kann sehr teuer werden. (Besonders bei großen Relationen, da viele I/O-Operationen anfallen können).

Merge Join

- Daten müssen auf Joinattributen sortiert sein. (Wenn nicht, fallen zusätzliche Sortierkosten an).
- Sortierung kann für den Join vorgelagert vorgenommen werden. (D.h., wenn Daten bereits für andere Operationen sortiert wurden, können diese Sortierkosten für den Join wiederverwendet werden).
- Kann Indexe verwenden. (Wenn ein Index auf dem Join-Attribut existiert, kann dieser für die Sortierung genutzt werden oder um eine bereits sortierte Reihenfolge zu gewährleisten).

Hash Join

- Gute Hashfunktionen sind die Grundlage. (Eine gleichmäßige Verteilung der Tupel auf die Buckets ist entscheidend für die Performance).
 - Beste Performance, wenn die kleinere Relation in den Hauptspeicher passt. (Idealfall: Die kleinere Relation kann komplett in den Hauptspeicher geladen und dort eine Hash-Tabelle aufgebaut werden, was die I/O-Kosten stark reduziert).
-

Zusammenfassung

- **Anfrageoptimierung ist eine Kernkomponente von relationalen DBMS.** (Entscheidend für die Leistungsfähigkeit einer Datenbank).
- **Heuristische Optimierung kann immer verwendet werden, kann aber zu suboptimalen Plänen führen.** (Schnelle, aber nicht immer die beste Lösung).
- **Kostenbasierte Optimierung ist auf Statistiken angewiesen.** (Genauere Optimierung durch Berücksichtigung von Datenverteilung und -größe).
- **Datenbanksysteme bieten Informationen zu Anfrageausführung an (EXPLAIN).** (Werkzeuge wie EXPLAIN ermöglichen es, den Ausführungsplan einer Abfrage zu analysieren und Engpässe zu identifizieren).
- **Datenbankadministratoren müssen stetig über Verbesserungen nachdenken (z.B. Indizes).** (Kontinuierliches Monitoring und Tuning sind notwendig, um die Performance aufrechtzuerhalten).