Datenbanksysteme II

Anfragebearbeitung

Wintersemester 2010/2011
Übung vom 12.01.2011

Clemens Schefels

Goethe-Universität Frankfurt am Main
Institut für Informatik

Lehrstuhl für Datenbanken und Informationssysteme (DBIS)

Anfragebearbeitung

- *Deklarative* Anfragesprachen wie z.B. SQL beschreiben das gewünschte Resultat, aber nicht, wie dieses erreicht werden soll („what, not how“).

- Physische Datenunabhängigkeit erlaubt es einem Datenbanksystem, konzeptuelle/logische Datenstrukturen physisch unterschiedlich zu implementieren.

- Das DBMS hat daher die Freiheit, zu einer Anfrage und physischen Datenstruktur einen entsprechenden, möglichst optimalen Ausführungsalgorithmus zu finden.

→ Mehrere mögliche Auswertungsstrategien.

→ **Ziel: schlechtesten Auswertungplan vermeiden (besten Auswertungsplan finden: NP-hart)**

Anfragebearbeitung

- DBMS kann die Kosten für die Ausführung eines Operators mit Hilfe von Kostenmodellen und Statistiken abschätzen.

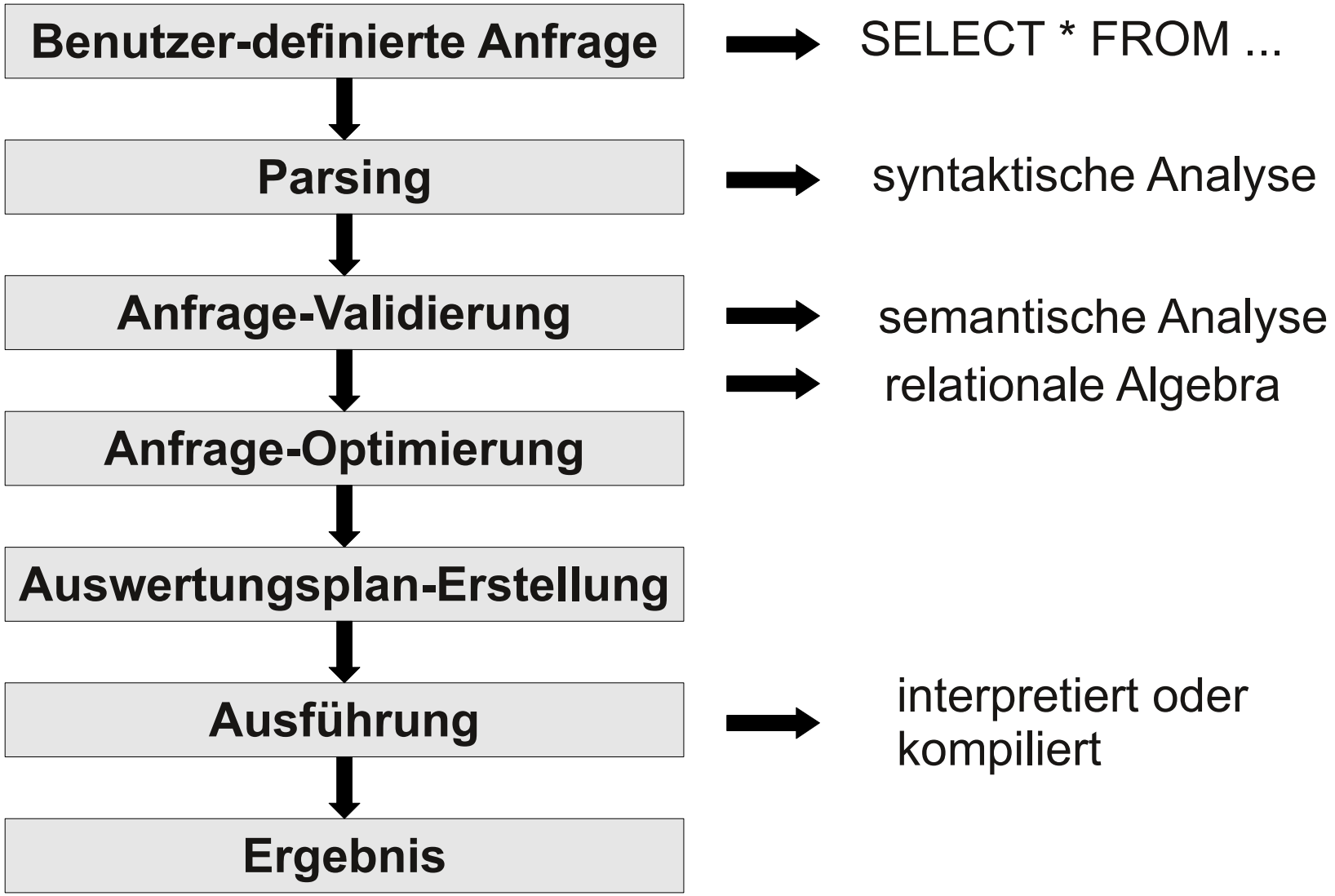
- Bei der Optimierung eines Plans werden Heuristiken angewandt, alle möglichen Pläne anzuschauen ist viel zu teuer.

- Optimierung kann auf verschiedenen Ebenen statt finden:

  → Logische Ebene (DB 1)

  → Physische Ebene (DB 2)

Ablauf der Anfrageverarbeitung



Logische Optimierung
(siehe DB 1)

- Ausgangspunkt ist relationaler algebraische Ausdruck, der nach kanonischer Übersetzung entstanden ist.

- Optimierung: Transformation relationaler algebraischer Ausdrücke in *äquivalente* Ausdrücke (die zu einem schnelleren/besseren Ausführungsplan führen).

- Umformungen sollten so gewählt werden, dass die Ausgaben der einzelnen Operatoren *möglichst* klein werden.

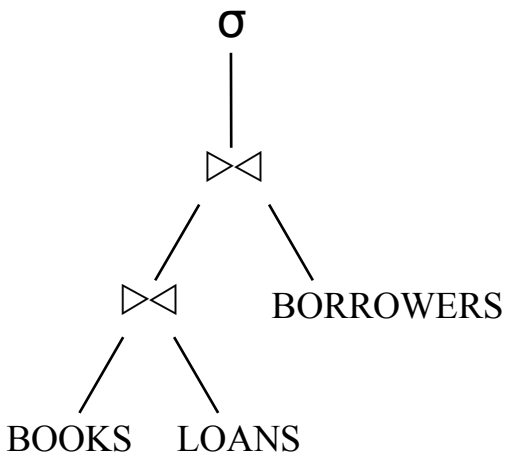
Logische Optimierung: Grundlegende Techniken

- Aufbrechen von **Selektionen**
- Verschieben von **Selektionen** nach ”unten” im Plan/Baum
- Zusammenfassen von **Selektionen** und **Kreuzprodukten** zu **Joins**
- Bestimmung der **Join**reihenfolge
- Einfügen von **Projektionen**
- Verschieben von **Projektionen** nach ”unten” im Plan/Baum

Beispiel aus DB 1

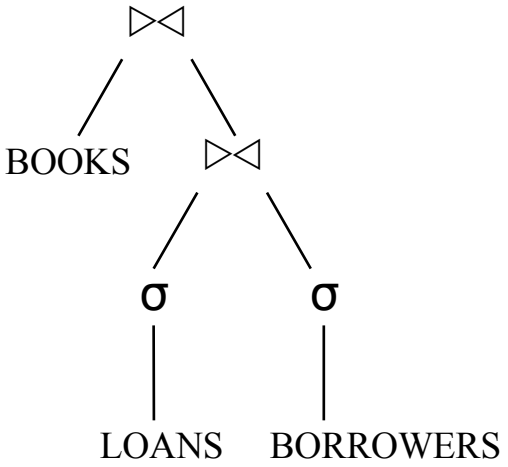
SELECT * FROM BOOKS, LOANS, BORROWERS
WHERE CITY='Frankfurt' **AND** Date='1.1.1997'

$$\sigma_{CITY='Frankfurt' \wedge DATE<1.1.1997}(BOOKS \bowtie LOANS \bowtie BORROWERS)$$



Beispiel aus DB 1 optimiert

$$\text{BOOKS} \bowtie (\sigma_{\text{DATE}<1.1.1997}(\text{LOANS}) \bowtie \sigma_{\text{CITY}='\text{Frankfurt}'}(\text{BORROWERS}))$$



Physische Optimierung

- Realisierung der **logischen** Algebraoperatoren durch **physische** Algebraoperatoren.

- Ein logischer Algebraoperator kann **mehrere** physische Realisierungen haben (siehe Joins).

- Bei der Auswahl der physischen Operatoren wird der **physische Aufbau** der Datenbank betrachtet (z.B. Indices).

Physische Realisierung der logischen Algebraoperatoren

- **Pipelining**:
  - schrittweise Realisierung
  - Algebraoperatoren werden komplett berechnet
  - Zwischenergebnisse werden gespeichert und weitergereicht

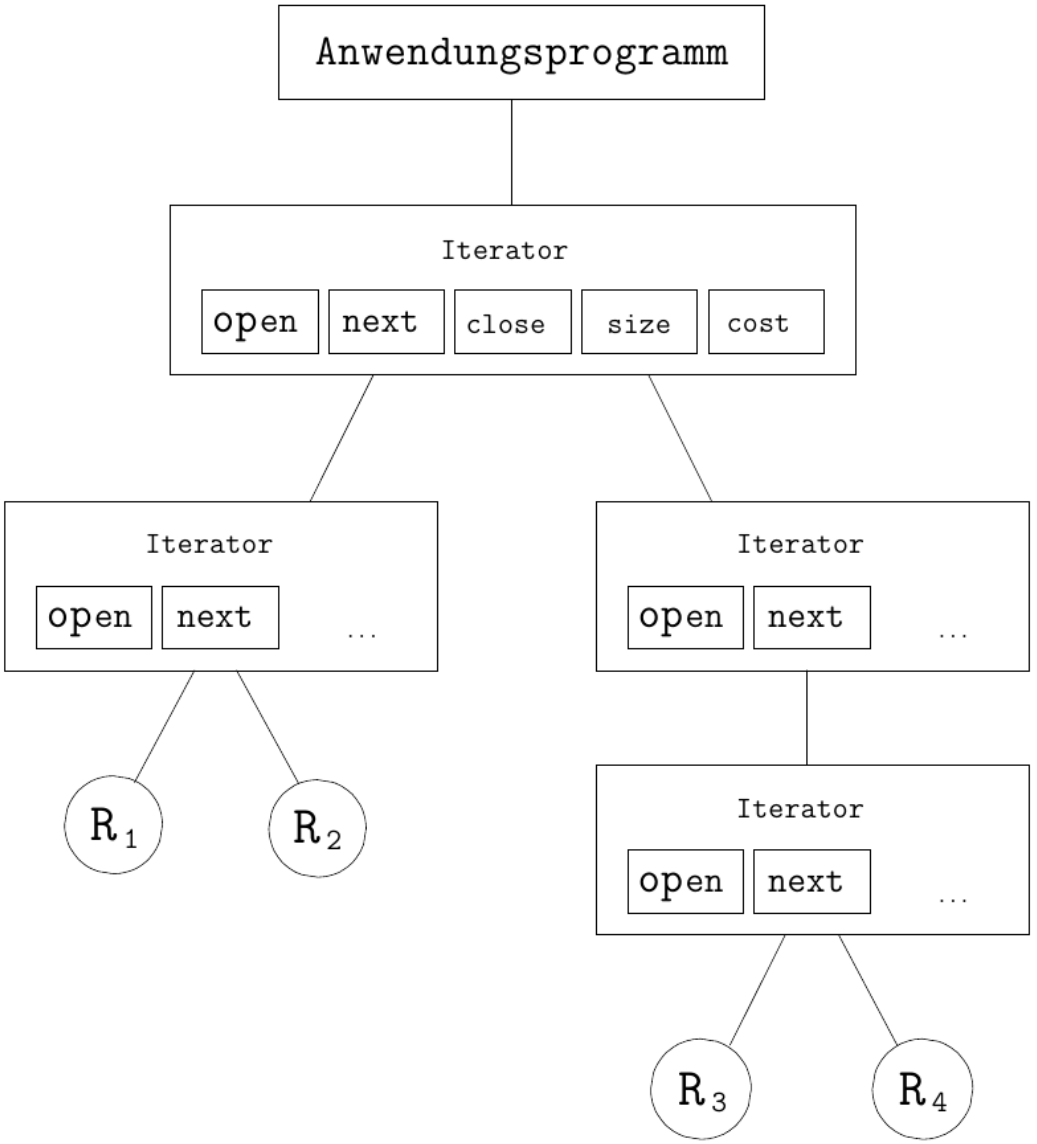
- **Iteratoren**:
  - baukastenartige Auswertungspläne
  - Iterator: abstrakter Datentyp
  - Teilergebnisse können ohne zwischen Speicherung weiterverarbeitet werden
  - baumartige Darstellung ähnlich der logischen Operatorbäumen

Iteratoren

abstrakter Datentyp mit vier Grundoperation:

- **open**: Eingabe öffnen und Initialisierung
- **next**: nächstes Tupel des Ergebnis der Berechnung weitergeben
- **close**: schließt Eingabe, gibt Ressourcen wieder frei
- **cost**: schätzt die Kosten der Berechnung
- **size**: schätzt die Größe des Ergebnisses

Iteratorbaum (Auswertungsplan)



Implementierung der Selektion mit Iteratoren

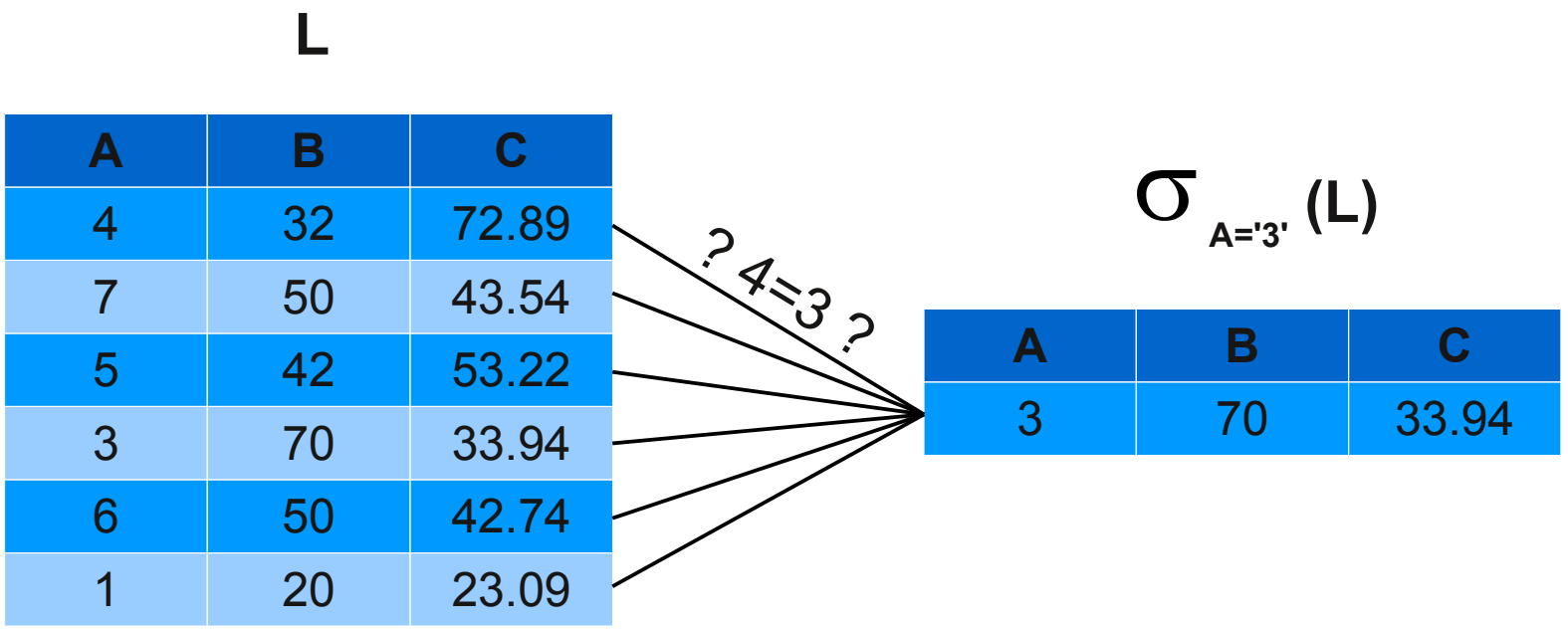
Iterator Standard-Select$_p$:

- **open**: öffne die Eingabe
- **next**: hole das jeweils nächste Tupel, bis eines die Bedingung *p* erfüllt (liefert die Eingabe kein Tupel mehr, melde fertig)
- **close**: schließe die Eingabe

Für den *Iterator Standard-Select*$_p$ lassen sich die Kosten exakt berechnen:

- Wird der *Iterator Standard-Select* auf eine Relation bzw. ein auf der Platte gespeichertes Zwischenergebnis angewendet, so greift er auf jeden Block davon genau einmal zu.
- Wird der *Iterator Standard-Select* auf einen anderen Iterator angewandt, wirkt er nur im Hauptspeicher als Filter, benötigt also keinen weiteren Plattenzugriff.

Standard-Selektion



Implementierung der Selektion mit Iteratoren

Falls ein Index für das Attribut in der Selektionsbedingung angelegt ist:

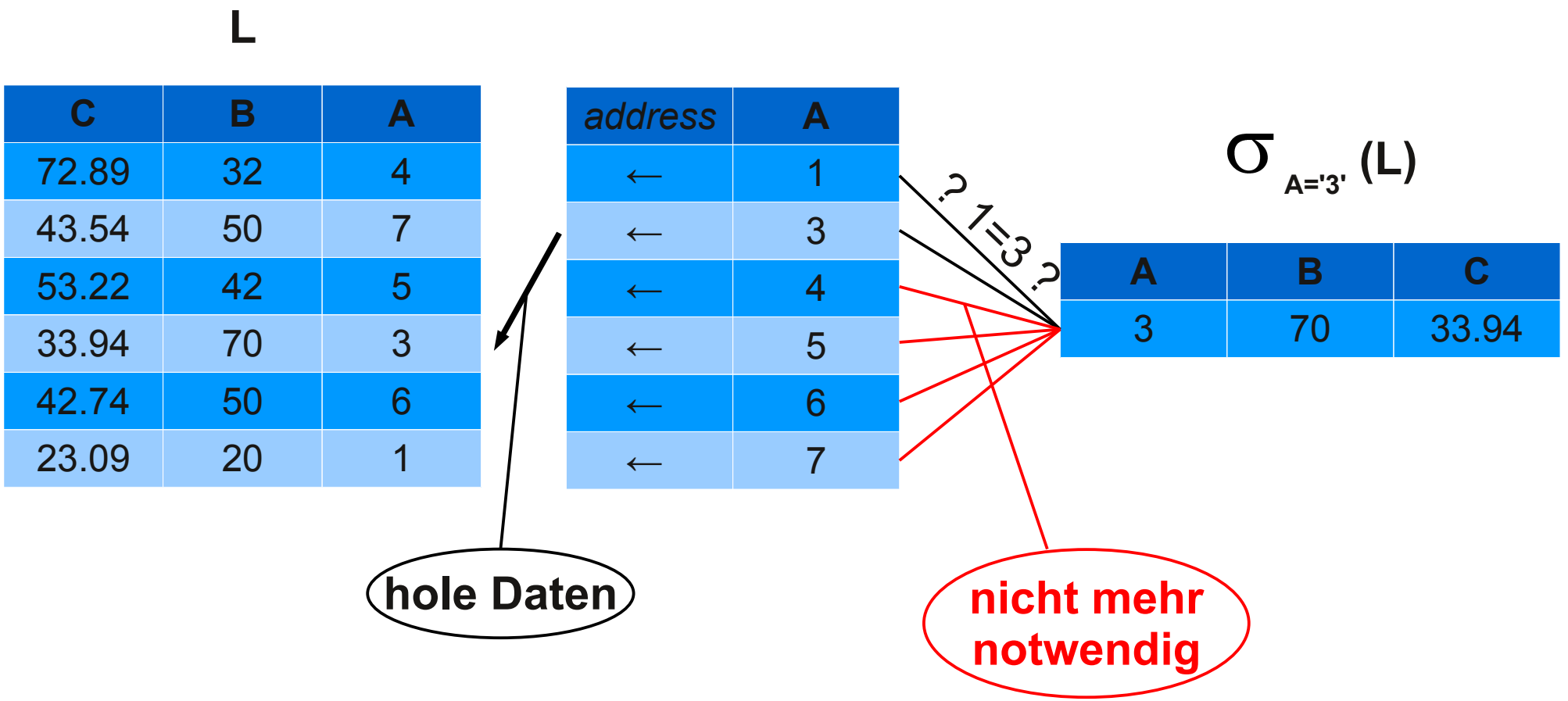
Iterator Index-Select$_p$:

- **open**: suche im Index die erste Stelle,
  an der die Bedingung *p* erfüllt ist.
- **next**: falls das jeweils nächste Tupel noch die Bedingung erfüllt,
  gib es zurück, sonst melde fertig.
- **close**: schließe die Eingabe.

Für den *Iterator Index-Select* auf einem Index, nach dem die Relation sortiert ist, lassen sich die Kosten nur abschätzen:

- Er erfordert den Abstieg im Index bis zu den Nutzdaten (in der Regel zwei Plattenzugriffe) und operiert anschließend sequentiell auf den Nutzdatenblöcken im Index.
  Dafür benötigt er Selektivität_des_Index * Blöcke_der_Relation weitere Plattenzugriffe.

Selektion mit Index



Implementierung binärer Zuordnungsoperatoren
Nested-Loop Join

Nested-Loop Join-Algorithmus: $L \bowtie_{L.A=R.B} R$

- **open**: öffne linke Eingabe L
- **next**: - rechte Eingabe R geschlossen?
  → öffne sie
  - fordere rechts solange Tupel an, bis Bedingung L.A=R.B erfüllt ist.
  - Sollte zwischendurch rechte Eingabe erschöpft sein
  → schließe rechte Eingabe R
  → fordere nächstes Tupel der linken Eingabe L an
  → starte **next** neu
  - Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück
- **close**: schließe Eingaben L und R

Weitere Join Implementierungen folgen in der DB2 Vorlesung!

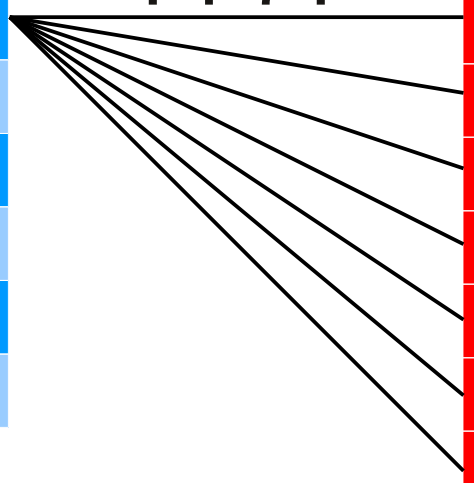
Nested-Loop Join (1)

L

A	B	C
4	32	72.89
7	50	43.54
5	42	53.22
3	70	33.94
6	50	42.74
1	20	23.09

L.A = R.A

? 4=7 ?



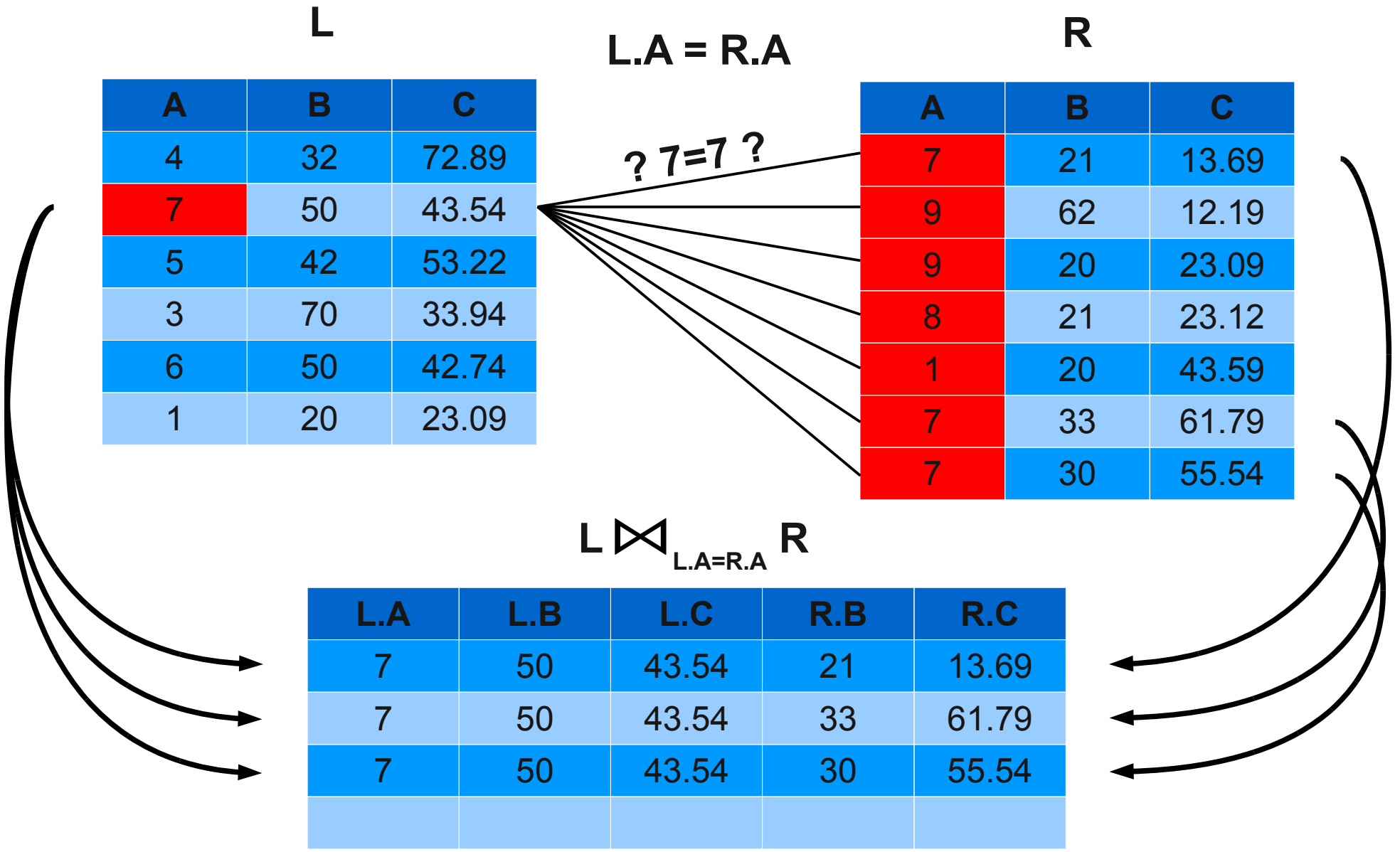
R

A	B	C
7	21	13.69
9	62	12.19
9	20	23.09
8	21	23.12
1	20	43.59
7	33	61.79
7	30	55.54

$L \bowtie_{L.A=R.A} R$

L.A	L.B	L.C	R.B	R.C

Nested-Loop Join (2)



Nested-Loop Join (3)

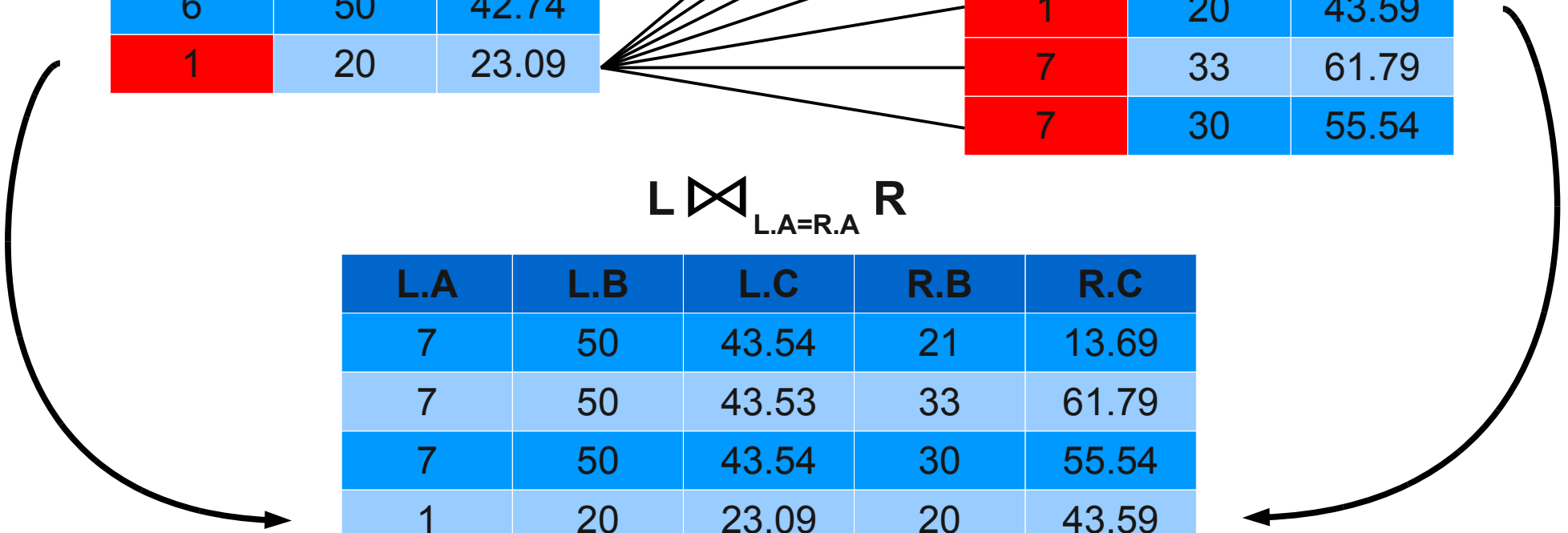
L

A	B	C
4	32	72.89
7	50	43.54
5	42	53.22
3	70	33.94
6	50	42.74
1	20	23.09

L.A = R.A

R

A	B	C
7	21	13.69
9	62	12.19
9	20	23.09
8	21	23.12
1	20	43.59
7	33	61.79
7	30	55.54



$L \bowtie_{L.A=R.A} R$

L.A	L.B	L.C	R.B	R.C
7	50	43.54	21	13.69
7	50	43.53	33	61.79
7	50	43.54	30	55.54
1	20	23.09	20	43.59

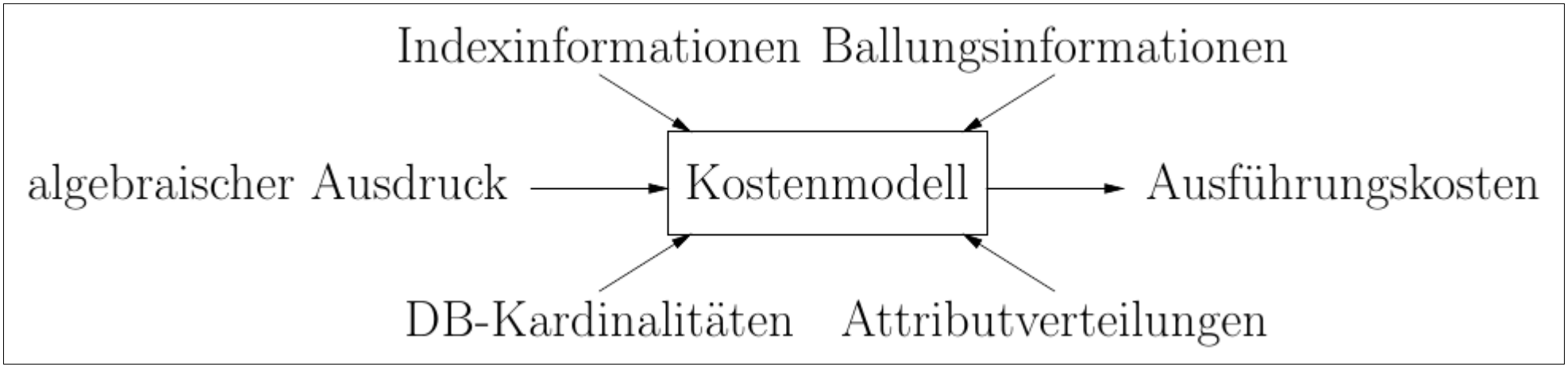
Übersetzung der logischen Algebra

- Auswahl der „optimalen“ physischen Operatoren durch:
  - Statische Optimierung:

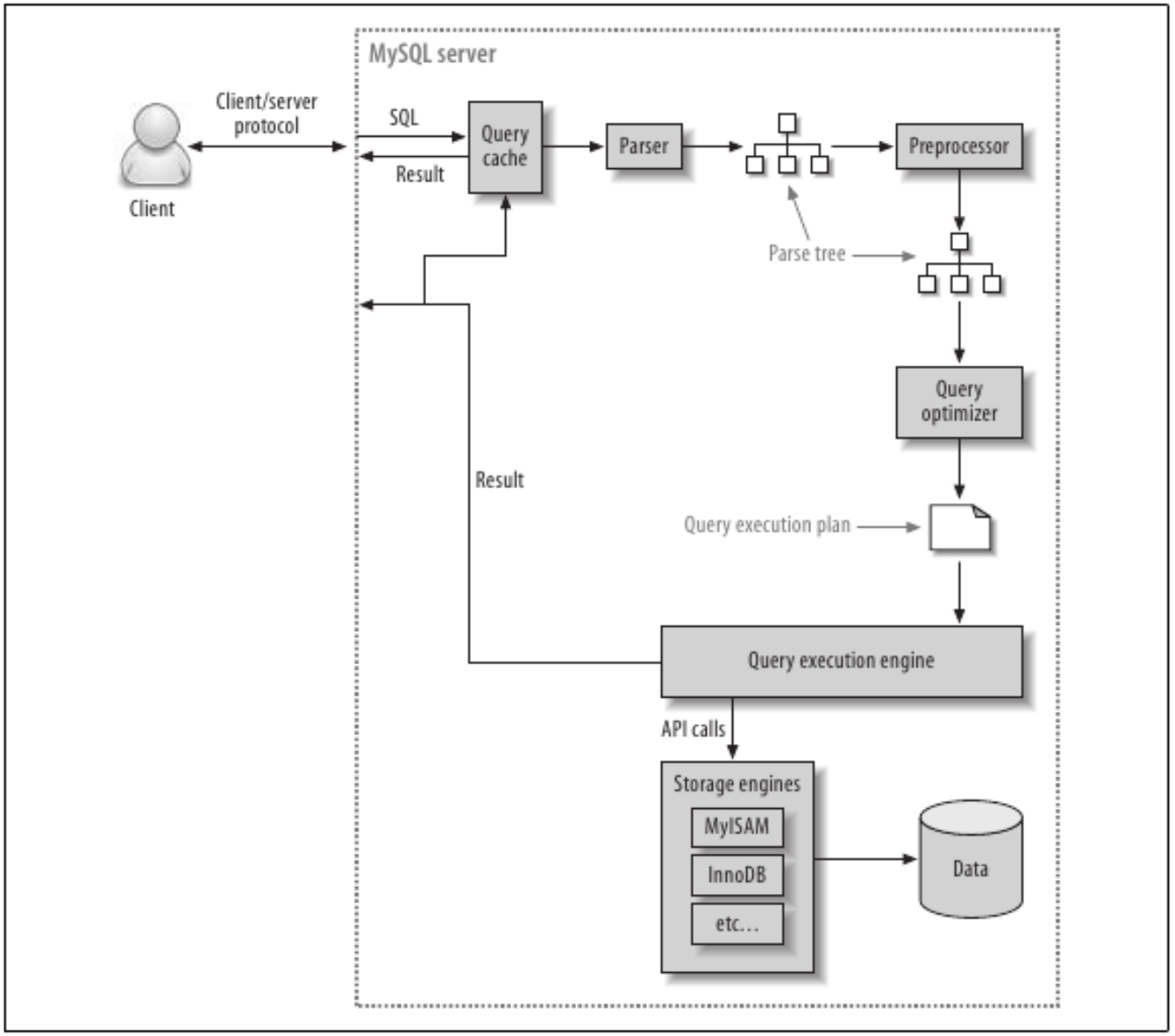
    Heuristiken/Regeln (also basierend auf Erfahrungswerten, Schätzungen etc.)

  - Dynamische Optimierung:

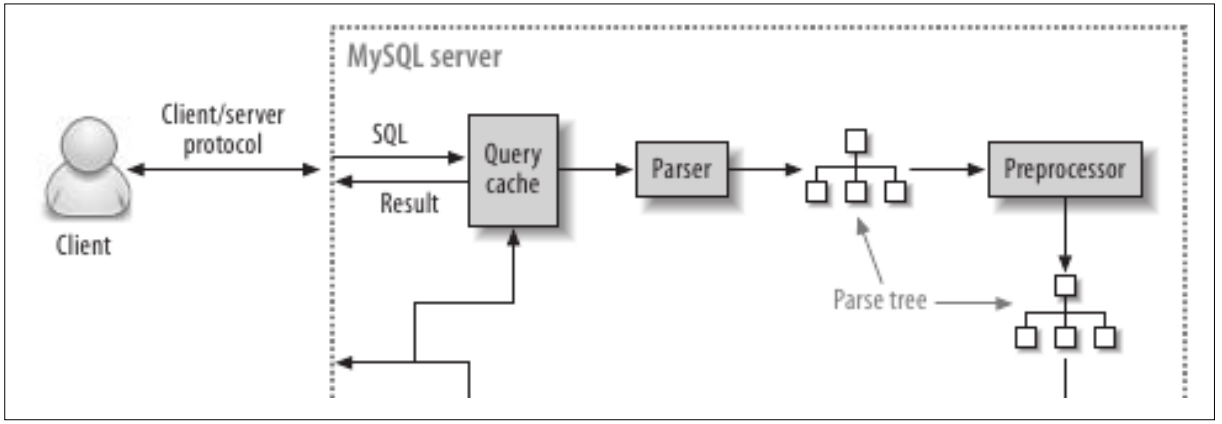
    Kostenmodell (siehe Grafik)



MySQL Ausführungspfad einer Abfrage (1)

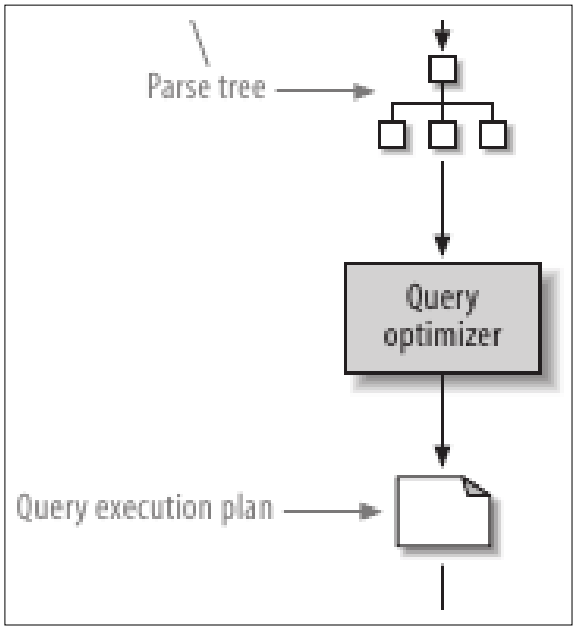


MySQL Ausführungspfad einer Abfrage (2)



- **Anfrage-Cache** (Query-Cache):
  - Ist selbe Abfrage schon gestellt worden?
    → wenn ja, sende gespeichertes Ergebnis an Client
- **Parser**:
  - Token und Parse-Baum Erzeugung
  - Validitätsprüfung
- **Preprozessor**:
  - existieren alle Tabellen/Spalten aus Abfrage
  - Namensauflösung und Rechteprüfung

MySQL Ausführungspfad einer Abfrage (3)



- **Abfrageoptimierer** (Query Opimizer):
  - Erstellung des Ausführungsplans
  - MySQL verwendet kostenbasierten und regelbasierten Optimierer

MySQL Ausführungspfad einer Abfrage (4)

```
mysql> SHOW STATUS LIKE 'last_query_cost';

+--------------------+-----------------+
| Variable_name      | Value           |
+--------------------+-----------------+
| Last_query_cost    | 1040.599000     |
+--------------------+-----------------+
```

→ Optimierer schätzt 1040 zufällige Lesezugriffe auf Datenseiten

MySQL Ausführungspfad einer Abfrage (5)
- Probleme des Optimierers -

- Statistiken können falsch sein:
  → Datenbestand kann sich geändert haben

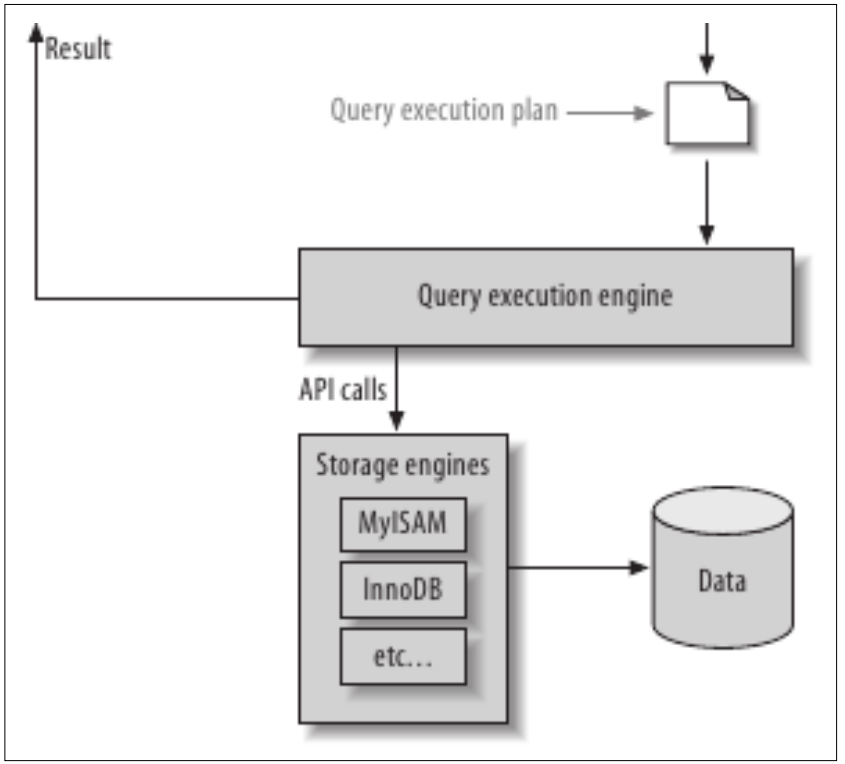
- Wahre Kosten nicht immer vorhersehbar:
  → wie liegen Daten auf dem Sekundärspeicher?

- Was ist optimal?
  → Schnelle Antwortzeit oder geringe Kosten

- Parallele Abfragen können sich gegenseitig beeinflussen:
  → z.B. Sperren

- Nicht alle Operationen können in der Kostenberechnung berücksichtigt werden:
  → z.B. benutzerdefinierte Funktionen

MySQL Ausführungspfad einer Abfrage (6)



- **Ausführungsplan** (Query execution plan)
- **Abfrageausführungs-Engine (Query execution engine)**:
  - MySQL erzeugt keinen Bytecode
    → Interpretation des Anweisungsbaums

Zusammenfassung

- Anfragebearbeitung
- Logische Optimierung (siehe DB 1)
- Physische Optimierung
  - Iteratoren Konzept
    - Implementierung algebraischer Operatoren mit Iteratoren
- MySQL Ausführungspfad einer Abfrage
  - Probleme des Optimierers

Literatur

- A. Kemper, A. Eickler: '**Datenbanksysteme - Eine Einführung** ', 7. Auflage, Oldenburg Verlag, 2009, ISBN 978-3-486-59018-0
  → Kapitel 8: Anfragebearbeitung
- Derek J. Balling, et al.: '**High Performance MySQL. Optimierung, Datensicherung, Replikation & Lastverteilung**', O'Reilly, 2009, ISBN: 3897218895
  → Kapitel 4: Optimierung der Abfrageleistung
- G. Vossen: '**Datenmodelle, Datenbanksprachen und Datenbank-managementsysteme**', 5. Auflage, Oldenburg Verlag, 2008, ISBN 3486275747
  → Kapitel 18: Verarbeitung und Optimierung von Anfragen
- Prof. Stefan Böttcher: '**Datenbanken SS 98**'
  → Kapitel 4: Physische Speicherorganisation und Anfrageoptimierung
  http://www2.cs.uni-paderborn.de/cs/ag-boettcher/lehre/SS98/daba/dbs98k4n.html