

Datenbanken und Informationssysteme

5. Relationale Anfragebearbeitung

Prof. Dr. Stefan Decker

📍 RWTH Aachen

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung

- ▶ **Motivation**
- ▶ Einführung in die Query Processing Schritte & Optimierungstypen
- ▶ Grundlagen: Regelbasierte Anfrageoptimierung
- ▶ Grundlagen: Kostenbasierte Anfrageoptimierung
- ▶ Join-Reihenfolgen

2. Algorithmen für Basisoperationen

3. Indexstrukturen

- ▶ Indexstrukturen für eindimensionale Daten
- ▶ Indexstrukturen für mehrdimensionale Daten

Das Problem: SQL vs. Ausführung

SQL: deklarativ – „WAS wollen wir?“

Gesucht sind die Namen aller Studierenden ab dem 7. Semester, die eine von 2137 gelesene Vorlesung hören.

Beispielanfrage:

```
\ttfamily\footnotesize
SELECT DISTINCT s.Name\\
FROM Student s CROSS JOIN Hört h
CROSS JOIN Vorlesung v\\
WHERE s.MatrNr = h.MatrNr\\
\hspace*{1.5em}AND h.VorlNr =
v.VorlNr\\
\hspace*{1.5em}AND v.gelesenVon =
2137\\
\hspace*{1.5em}AND s.Semester >= 7
```

Basisrelationen

► Studenten (S): 8 Tupel

► Hört (H): 10 Tupel

⚙️ DBMS: prozedural

WIE bekommen wir es?



? **Frage:** Wie kann man diese Anfrage naiv ausführen?

Ausschnitt aus unserer Datenbank

Student		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

8 Tupel

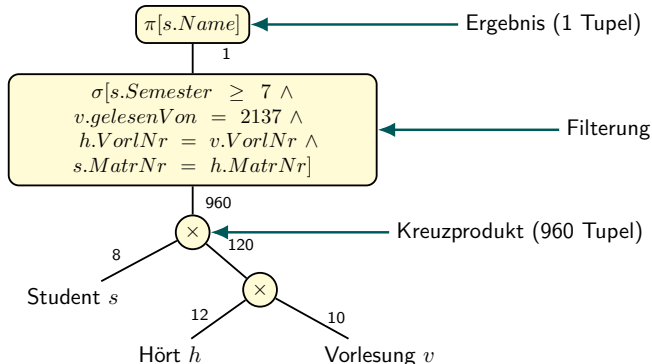
Hört	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

12 Tupel

Vorlesung			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

10 Tupel

Motivation - Der naive Ausführungsplan



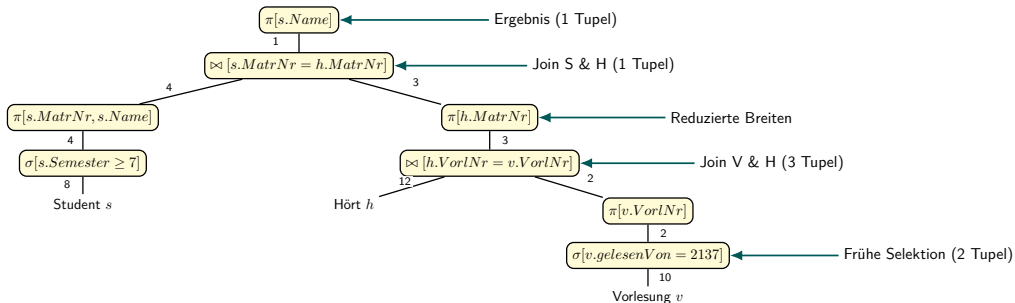
Kosten (vereinfacht: Summe Zwischenergebnisse):

- ▶ $8 + 12 + 10$ (Basis) $+ 120$ (1. Kreuzprod.) $+ 960$ (2. Kreuzprod.) $+ 1$ (Ergebnis) ≈ 1111 verarbeitete Tupel

→ **Problem:** Extrem große Zwischenergebnisse!

Motivation - Ein besserer Plan & Vergleich

Es gibt äquivalente Pläne mit gleichem Ergebnis!



Kosten (vereinfacht): ≈ 49 verarbeitete Tupel

Vergleich: Naiver Plan ~ 1111 Tupel, optimierter Plan ~ 49 Tupel

$\Rightarrow$ **Faktor 22 besser!**

$\Rightarrow$ Kann Unterschied zwischen Sekunden und Stunden bedeuten!

? Frage: Wie erhält man diesen besseren Plan?

Motivation - Warum das DBMS optimieren *muss*

1. Physische Datenunabhängigkeit

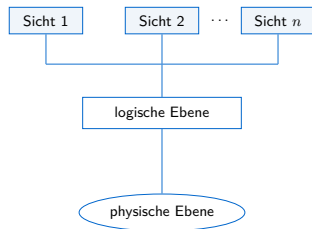
- ▶ Nutzer formulieren Anfragen unabhängig von der physischen Speicherung (Tabellen, Indizes etc.).
- ▶ DB-Admin kann Speicherung ändern, ohne dass Anwendungen angepasst werden müssen (z. B. Index hinzufügen/löschen).

2. Transparenz

- ▶ Das DBMS passt den Ausführungsplan automatisch an die aktuelle physische Struktur an.
- ▶ Es wählt idealerweise den besten Plan für die *momentane* Konfiguration.

⇒ **Weniger Aufwand** für Entwickler/Nutzer.

⇒ **Mehr Flexibilität** für Administration und Wartung.



3-Ebenen-Modell nach
ANSI/SPARC (1975)

Zusammenfassung

- ▶ Gleiche SQL-Anfragen können **sehr unterschiedlich schnell** ausgeführt werden.
- ▶ Optimierung durch das DBMS ist deshalb **kritisch**.

Nächste Frage: Wie transformiert das DBMS einen naiven Plan in einen besseren Plan?

Ein wichtiger Baustein:

Regelbasierte Anfrageoptimierung
(Anwendung logischer Äquivalenzregeln)

Zunächst: Einordnung der Verarbeitungsschritte und Optimierungstypen

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung

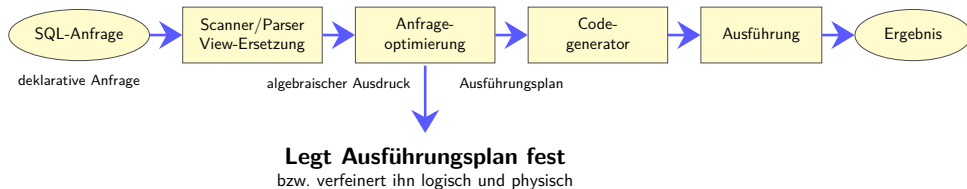
- ▶ Motivation
- ▶ **Einführung in die Query Processing Schritte & Optimierungstypen**
- ▶ Grundlagen: Regelbasierte Anfrageoptimierung
- ▶ Grundlagen: Kostenbasierte Anfrageoptimierung
- ▶ Join-Reihenfolgen

2. Algorithmen für Basisoperationen

3. Indexstrukturen

- ▶ Indexstrukturen für eindimensionale Daten
- ▶ Indexstrukturen für mehrdimensionale Daten

Wo passt Optimierung rein? Der Weg der Anfrage



Unser Fokus jetzt: der **Anfrageoptimierer**

Wichtige Einsicht

Der **Anfrageoptimierer** ist keine einzelne, simple Aktion.

Ziel des Anfrageoptimierers

Logische Anfrage des Nutzers in einen **effizienten physischen Ausführungsplan** übersetzen, also in die Form, in der das DBMS die Daten tatsächlich verarbeitet.

Im Inneren kombiniert der Anfrageoptimierer mehrere **Phasen** und **Strategien**. **Leitfrage:**
Welche grundsätzlichen Strategien gibt es?

Strategien im Anfrageoptimierer: Drei Blickwinkel

1. Regelbasiert

- ▶ Nutzt logische Äquivalenzregeln der Relationalen Algebra.
- ▶ Verwendet Heuristiken wie „Filter früh anwenden“.
- ▶ Arbeitet vor allem auf Schema- und Ausdrucksebene.
- ▶ Erzeugt logisch äquivalente, oft bessere Pläne.

2. Kostenbasiert

- ▶ Bewertet Kandidatenpläne mit einem Kostenmodell.
- ▶ Nutzt Statistiken über Größe und Verteilung von Daten.
- ▶ Ist zentral für Entscheidungen wie Join-Reihenfolgen.
- ▶ Wählt den Plan mit den niedrigsten geschätzten Kosten.

3. Physisch

- ▶ Trifft konkrete physische Entscheidungen.
- ▶ Wählt Algorithmen wie Hash- oder Merge-Join.
- ▶ Entscheidet über Indexnutzung.
- ▶ Legt den Datenfluss fest: Pipelining oder Materialisierung.

In realen Systemen sind diese drei Blickwinkel nicht strikt getrennt, sondern greifen ineinander.

Unser Fahrplan und Übergang

Diese Blickwinkel sind **nicht strikt getrennt**. Oft interagieren sie oder bauen aufeinander auf.

1. **Regelbasierte Optimierung** (JETZT)

Logische Grundlagen und Äquivalenzregeln verstehen.

2. **Kostenbasierte Optimierung** (SPÄTER)

Kostenmodelle, Statistiken und Join-Reihenfolgen.

3. **Physische Aspekte** (SPÄTER)

Algorithmen, Indexe und Datenfluss.

Als nächstes: Regelbasierte Optimierung - Äquivalenzregeln als Werkzeugkasten

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung

- ▶ Motivation
- ▶ Einführung in die Query Processing Schritte & Optimierungstypen
- ▶ **Grundlagen: Regelbasierte Anfrageoptimierung**
- ▶ Grundlagen: Kostenbasierte Anfrageoptimierung
- ▶ Join-Reihenfolgen

2. Algorithmen für Basisoperationen

3. Indexstrukturen

- ▶ Indexstrukturen für eindimensionale Daten
- ▶ Indexstrukturen für mehrdimensionale Daten

Ziel der Regelbasierten Optimierung:

Einen Algebra-Ausdruck (Plan) in einen logisch äquivalenten, aber hoffentlich effizienteren Ausdruck umwandeln.

Wie?

Ausnutzung von mathematischen Eigenschaften der Relationalen Algebra (Äquivalenzregeln).

Denkweise:

- ▶ Algebra-Ausdruck als Baumstruktur darstellbar (Query Plan)
- ▶ Äquivalenzregeln = erlaubte Baum-Transformationen

Grundlage: Diese Regeln ändern das Ergebnis der Anfrage **NICHT**, nur die Berechnungsweise.

Heuristiken helfen bei der Auswahl „guter“ Regeln.

Äquivalenzregeln formen einen logischen Plan in einen **äquivalenten, effizienteren** Plan um.

1. Push Selection

- ▶ Filter früh anwenden
- ▶ Zwischenergebnisse verkleinern
- ▶ Vor Joins besonders wichtig

2. Push Projection

- ▶ Nur benötigte Attribute
- ▶ Tupelbreite reduzieren
- ▶ Spart Speicher und Transfer

3. Combine Operations

- ▶ Operationen zusammenfassen
- ▶ Kreuzprodukte vermeiden
- ▶ Bessere Operatoren ermöglichen

Werkzeugkasten für die nächsten Beispielschritte.

Zusatzmaterial: Handout nach Elmasri/Navathe (2016), S. 697–701.

Regelbasierte Optimierung - Regeln für Selektionen

Ablauf: Konjunktion zerlegen, Filter umsortieren, Filter nach unten schieben.

Regel 1: Split

$$\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R))$$

Warum? Zerlegt eine grosse Konjunktion in einzelne Filter und ermöglicht getrenntes Verschieben.

Regel 2: Commute

$$\sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$$

Warum? Macht die Reihenfolge von Filtern flexibel.

Regel 6: Push

$$\sigma_B(R \times S) = \sigma_B(R) \times S$$

falls $\text{attr}(B) \subseteq \text{attr}(R)$

Warum? Reduziert Tupelanzahlen vor teuren Operationen.

Im Beispiel gilt also: **erst zerlegen, dann umsortieren, dann nach unten schieben.**

Zusatzmaterial: Handout nach Elmasri/Navathe (2016), S. 697–701.

Regelbasierte Optimierung - Projektionen und Joins

Zwei weitere Regeln reduzieren Datenvolumen und vermeiden unnötige Zwischenschritte:

Regel 7: Push Projection

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A_1}(R) \bowtie_B \pi_{A_2}(S))$$

$$A_1 = \text{attr}(R) \cap (A \cup \text{attr}(B))$$

$$A_2 = \text{attr}(S) \cap (A \cup \text{attr}(B))$$

Warum? Es werden nur noch die für spätere Joins und das Endergebnis benötigten Attribute weitergegeben.

Regel 12: Combine Operations

$$\sigma_C(R \times S) = R \bowtie_C S$$

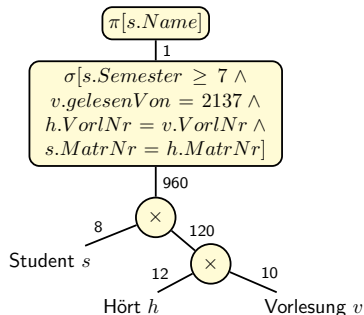
Warum? Ein explizites Kreuzprodukt mit anschließender Selektion wird als Join erkennbar und erlaubt bessere Join-Operatoren.

Im Beispiel führen diese Regeln anschliessend zu **expliziten Joins** und **kleineren Tupelbreiten**.

Zusatzmaterial: Handout nach Elmasri/Navathe (2016), S. 697–701.

Anwendung am Beispiel - Schritt 1: Split Selection

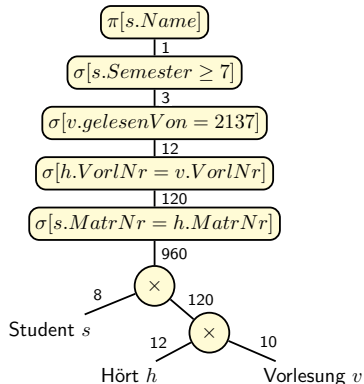
Ziel: Den naiven Plan für die Beispielanfrage schrittweise verbessern.



Regel 1 Split Selection

$$\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R))$$

Die große Konjunktion
wird in einzelne Filter
zerlegt.



Ergebnis: Struktur vorbereitet; echte Reduktion folgt durch Push Selection.

Anwendung am Beispiel - Schritt 2: Selektionen verschieben

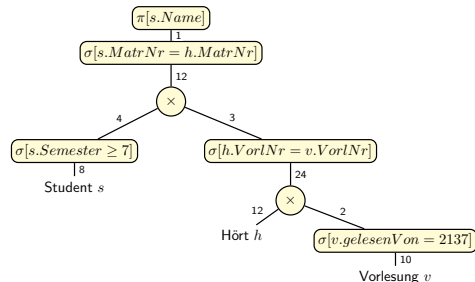
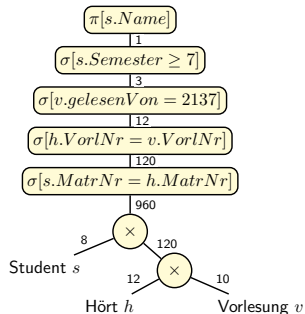
Regel 2 + 6

Commute +
Push
Selection

$$\sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$$

$$\sigma_B(R \times S) = \sigma_B(R) \times S$$

Filter werden
vertauscht und
an die frühest-
mögliche Stelle
verschoben.



Nächster Schritt: Selektion über Kreuzprodukt zu echten Joins zusammenfassen.

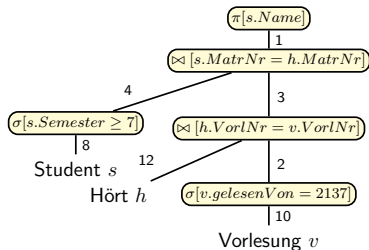
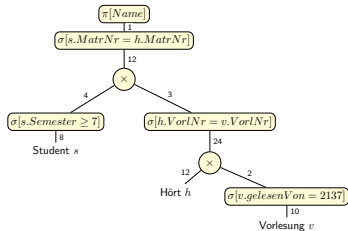
Anwendung am Beispiel - Schritt 3: Zu Joins zusammenfassen

Regel 12

Combine
Operations

$$\sigma_C(R \times S) = R \bowtie_C S$$

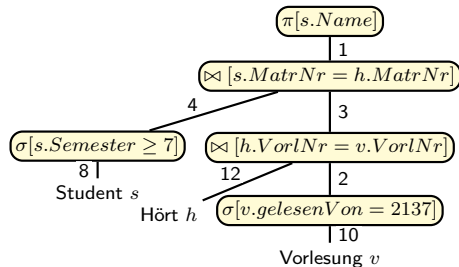
Selektion auf
Kreuzprodukt
wird zu einem
Join und erlaubt
bessere
Operatoren.



Nächster Schritt: Überflüssige Attribute früh entfernen - Push Projection.

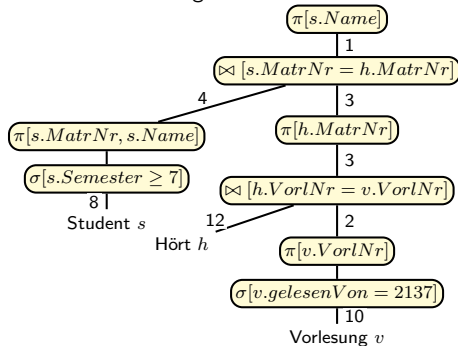
Anwendung am Beispiel - Schritt 4: Zusätzliche Projektionen

Ausgangspunkt: Explizite Joins und frühe Selektionen sind bereits erreicht. Jetzt entfernen wir auch **überflüssige Attribute** so früh wie möglich; dadurch transportiert der Plan früh nur noch benötigte Attribute.



Regel 7
Push
Projection

Weniger
Attribute



Fazit: Frühe Projektionen reduzieren Tupelbreite und unnötigen Datenfluss.

Der Werkzeugkasten - Regeln für Selektion (σ)

Regel 1: Konjunktionen aufbrechen

$$\sigma_{A \wedge B}(R) = \sigma_A(\sigma_B(R))$$

Ermöglicht, dass einzelne Bedingungen später getrennt verschoben werden.

Regel 2: Selektionen vertauschen

$$\sigma_A(\sigma_B(R)) = \sigma_B(\sigma_A(R))$$

Die Reihenfolge mehrerer Filter ist flexibel, solange ihre Bedeutung gleich bleibt (Kommutativität der Selektion)

Regel 6: Selektion nach unten schieben

$$\sigma_B(R \bowtie S) = \sigma_B(R) \bowtie S$$

$$\sigma_B(R \times S) = \sigma_B(R) \times S$$

jeweils falls $\text{attr}(B) \subseteq \text{attr}(R)$

Selektiere möglichst nahe an der Relation, auf die sich das Prädikat bezieht.

Regeln 10, 13, 14: Durch Mengenoperationen schieben

$$\sigma_B(R \cup S) = \sigma_B(R) \cup \sigma_B(S)$$

$$\sigma_B(R \cap S) = \sigma_B(R) \cap S = R \cap \sigma_B(S)$$

$$\sigma_B(R - S) = \sigma_B(R) - S$$

Auch bei stets wahren B : $\sigma_B(R) = R$.

Kernaussage: *Push Selection* verkleinert Zwischenergebnisse früh durch logisch äquivalente Umformungen.

Der Werkzeugkasten - Regeln für Projektion (π)

Regel 3: Redundante Projektionen

$$\pi_X(\pi_Y(R)) = \pi_X(R) \\ \text{falls } X \subseteq Y$$

Entfernt unnötige Schritte in Projektionsketten.

Regel 4: Projektion und Selektion

$$\pi_A(\sigma_B(R)) = \sigma_B(\pi_A(R)) \\ \text{falls } attr(B) \subseteq A$$

Hilft, Filter und Attributreduktion passend zu kombinieren.

Regel 7: Projektionen in Joins schieben

$$\pi_A(R \bowtie_B S) = \pi_A(\pi_{A_1}(R) \bowtie_B \pi_{A_2}(S)) \\ A_1 = attr(R) \cap (A \cup attr(B)) \\ A_2 = attr(S) \cap (A \cup attr(B))$$

Kernregel für *Push Projection*: Join-Partner behalten nur benötigte Attribute.

Regel 11: Projektion durch Vereinigung

$$\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$$

Gilt im Allgemeinen nicht entsprechend für $\cap$ (Schnittmenge) oder $-$ (Mengendifferenz).

Kernaussage: *Push Projection* reduziert früh die Tupelbreite, ohne benötigte Join- oder Ergebnisattribute zu verlieren.

Der Werkzeugkasten - Joins und weitere Umformungen

Regel 5: Kommutativität

$$R \times S = S \times R, \quad R \bowtie S = S \bowtie R$$

Eröffnet alternative Join-Reihenfolgen.

Regel 9: Assoziativität

$$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$

$$R \times (S \times T) = (R \times S) \times T$$

Alternative Join-Bäume sind logisch gleich, aber physisch oft sehr unterschiedlich teuer.

Regel 12: Kreuzprodukt plus Selektion wird Join

$$\sigma_C(R \times S) = R \bowtie_C S$$

Macht den Plan bereit für echte Join-Operatoren.

Regeln 8 und 9: Mengenoperationen

$$R \cup S = S \cup R, \quad R \cup (S \cap T) = (R \cup S) \cap T$$

$$R \cap S = S \cap R, \quad R \cap (S \cup T) = (R \cap S) \cup T$$

Hilft bei algebraischen Umformungen außerhalb klassischer Joins.

Regel 15: Triviale Vereinfachungen

$$R \cup \emptyset = R$$

$$\sigma_{\text{wahr}}(R) = R$$

Erfasst einfache algebraische Vereinfachungen.

Kernaussage

Diese Regeln erzeugen viele logisch äquivalente Join-Bäume, bewerten sie aber noch nicht.

Ein heuristischer Algorithmus für die Anfrageoptimierung

1. Split Selection

Zerlege komplexe Selektionen in eine Kaskade einfacher Filter.

Regelbezug: Regel 1.

2. Push Selection

Verschiebe Selektionen so weit wie möglich Richtung Blattknoten.

Regelbezug: Regeln 2, 4, 6, 10, 13, 14.

3. Restriktive Blätter zuerst

Ordne Blattknoten nach möglichst starken Filtern und vermeide unnötige Kreuzprodukte.

Regelbezug: Regeln 5 und 9.

4. Kreuzprodukte zu Joins machen

Kombiniere Kreuzprodukte mit passenden Selektionen zu echten Join-Knoten.

Regelbezug: Regel 12.

5. Push Projection

Teile und verschiebe Projektionen so, dass nur benötigte Attribute erhalten bleiben.

Regelbezug: Regeln 3, 4, 7, 11.

6. Teilbäume für Ausführungsalgorithmen finden

Markiere Teilpläne, die später von einem konkreten physischen Operator übernommen werden können.

Heuristische Idee: Erst den logischen Plan systematisch vereinfachen und erst danach konkrete physische Algorithmen wählen.

Nach Elmasri & Navathe, *Fundamentals of Database Systems*, 7th Edition, S.700–701.

Zusammenfassung der regelbasierten Optimierung

Was wir erreicht haben

- ▶ Algebraische Umformungen liefern logisch äquivalente Pläne.
- ▶ Heuristiken verbessern oft die Struktur des Anfragebaums.
- ▶ Frühe Selektionen und Projektionen verkleinern Zwischenrelationen.

Das Problem / die Grenze

- ▶ Besonders bei Join-Reihenfolgen gibt es sehr viele Alternativen.
- ▶ Regelbasierte Optimierung weiß nicht, welcher Plan *tatsächlich* am schnellsten ist.
- ▶ Es fehlt Wissen über Datenverteilungen und konkrete Algorithmen.

Was fehlt? Eine Methode zur **Kostenabschätzung** von Plänen.

Nächstes Thema: Kostenbasierte Optimierung

5. Relationale Anfragebearbeitung

1. Anfragebearbeitung und -optimierung

- ▶ Motivation
- ▶ Einführung in die Query Processing Schritte & Optimierungstypen
- ▶ Grundlagen: Regelbasierte Anfrageoptimierung
- ▶ **Grundlagen: Kostenbasierte Anfrageoptimierung**
- ▶ Join-Reihenfolgen

2. Algorithmen für Basisoperationen

3. Indexstrukturen

- ▶ Indexstrukturen für eindimensionale Daten
- ▶ Indexstrukturen für mehrdimensionale Daten

Kostenbasierte Optimierung (CBO) - Warum?

Rückblick

- ▶ Regelbasierte Optimierung erzeugt viele logisch äquivalente Pläne.
- ▶ Sie weiß aber nicht, welcher dieser Pläne am schnellsten ist.

Problem

Gerade bei Joins gibt es viele Alternativen mit potenziell sehr unterschiedlicher Performance.

Ziel der CBO

Wähle den Ausführungsplan mit den **geringsten geschätzten Ressourcenkosten** für Zeit, I/O und CPU.

Kernfokus

- ▶ Auswahl einer guten Join-Reihenfolge
- ▶ Auswahl der besten physischen Operatoren, also konkreter Algorithmen

Wie vergleicht man Pläne? → Wir brauchen Kosten!

Definition: Kostenmodell

Formeln und Parameter, die den erwarteten Ressourcenverbrauch von Operatoren und Gesamtplänen abschätzen.

- ▶ Typische Kostenfaktoren: I/O-Operationen (IOPS, Lese/Schreibzugriffe), CPU-Zeit (Rechenoperationen, Vergleiche etc.) und Netzwerk-Transfer bei verteilten DBs.

Kosten eines Operators hängen stark ab von:

- ▶ **Größe der Eingabedaten (Kardinalität!).**
- ▶ Implementierungsalgorithmus (z. B. Hash vs. Nested Loop-Join).
- ▶ Physischen Parametern (Puffergröße, Plattengeschw. etc.).

Folgerung: Um Kosten zu schätzen, müssen wir die Größe von Zwischenergebnissen schätzen können.

CBO - Kardinalitätsschätzung und Selektivität

Kosten hängen von Eingabegrößen ab. Deshalb brauchen wir Schätzungen für die **Kardinalität** von Zwischenergebnissen.

Schlüsselkonzept: Selektivität

Die Selektivität *sel* ist der geschätzte Anteil der betrachteten Eingabeobjekte, die eine Bedingung erfüllen oder zu einem Ergebnis beitragen.

Selektion: Anteil passender Tupel

$$sel_B = \frac{|\sigma_B(R)|}{|R|}$$
$$|\sigma_B(R)| \approx |R| \cdot sel_B$$

Join: Anteil passender Tupelpaare

$$sel_{RS} = \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$
$$|R \bowtie S| \approx |R| \cdot |S| \cdot sel_{RS}$$

Wichtig: Selektivität ist **kein** Kostenmaß. Sie ist Input für die Kardinalitätsschätzung, und diese ist wiederum Input für das Kostenmodell.

CBO - Einfache Selektivitätsschätzung

Wie schätzt man sel ? Typische Spezialfälle mit Vereinfachungen:

1. Gleichheit auf Schlüssel

Wenn A Schlüssel von R ist:

$$sel_{A=c} = \frac{|\sigma_{A=c}(R)|}{|R|} = \frac{1}{|R|}$$

Maximal ein Tupel passt.

2. Gleichheit auf Nicht-Schlüssel

Unter Annahme gleichverteilter Werte:

$$sel_{A=c} = \frac{1}{V(A, R)}$$

$V(A, R)$ ist die Anzahl verschiedener Werte von A in R .

3. Schlüssel/Fremdschlüssel-Join

Wenn A Schlüssel von R ist und B in S auf A verweist:

$$|R \bowtie_{A=B} S| \approx |S|, \quad sel_{A=B} \approx \frac{1}{|R|}$$

Sonst nur obere Schranke: $|R \bowtie_{A=B} S| \leq |S|$.

4. Boolesche Verknüpfungen

Unter Unabhängigkeitsannahme:

$$sel_{B_1 \wedge B_2} = sel_{B_1} \cdot sel_{B_2}, \quad sel_{\neg B_1} = 1 - sel_{B_1}$$

$$sel_{B_1 \vee B_2} = sel_{B_1} + sel_{B_2} - sel_{B_1} \cdot sel_{B_2}$$

Achtung: Alle Formeln beruhen auf Annahmen und liefern deshalb nur **Schätzungen**.

CBO - Methoden zur Selektivitätsschätzung

Wie schätzt das DBMS Selektivität genauer? Vor allem über Statistiken und Stichproben zu den Daten.

1. Histogramme

- ▶ Zerlegen den Wertebereich in Buckets.
- ▶ Speichern pro Bucket Tupelzahl oder häufige Werte.
- ▶ Typisch: *Equal Width* und *Equal Depth*.

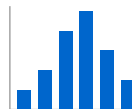
2. Parametrische Verteilungen

- ▶ Approximiere die Daten durch eine bekannte Funktion, z.B. Normalverteilung.
- ▶ Speichere nur wenige Parameter wie μ und σ^2 .
- ▶ Kompakt, aber nur gut, wenn die Funktion wirklich passt.

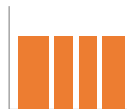
3. Stichproben (Sampling)

- ▶ Analysiere eine kleine, zufällige Teilmenge.
- ▶ Schließe von der Probe auf die Gesamtheit.
- ▶ Einfach, aber abhängig von Probengröße und Datenstruktur.

Histogramm-Idee



Equal Width



Equal Depth

Parametrische Approximation



? Frage: Wie genau kann man die Statistiken zur Schätzung der Selektivität verwenden?

CBO - Herausforderungen bei der Schätzung

Selektivitäts- und Kardinalitätsschätzung ist schwierig, vor allem aus drei Gründen:

1. Korrelationen zwischen Attributen

- ▶ Die Unabhängigkeitsannahme ist oft falsch.
- ▶ Beispiel:

$$sel(marke='VW' \wedge modell='Golf') \neq sel(marke='VW') \cdot sel(modell='Golf')$$

2. Mehrdimensionale Anfragen

- ▶ Prädikate betreffen oft mehrere Attribute gleichzeitig.
- ▶ Einfache 1D-Methoden ignorieren Abhängigkeiten.
- ▶ Mehrdimensionale Modelle sind komplex und leiden unter dem **Fluch der Dimensionen**.

3. Parameter in Anfragen

- ▶ Bei vorbereiteten Anfragen ist der konkrete Wert zur Optimierungszeit oft unbekannt, z.B. bei `WHERE datum > ?`.

Fazit: Schätzungen sind oft ungenau, aber notwendig. Moderne DBMS nutzen deshalb komplexe Statistiken und Modelle.



Frage: Was meint der Begriff „Fluch der Dimensionen“?

CBO - Nächster Schritt: Join-Reihenfolgen

Wir haben jetzt die Bausteine der kostenbasierten Optimierung:

1. Kostenmodell

Pläne werden über geschätzte Ressourcenverbräuche vergleichbar.

2.

Kardinalitätsschätzung

Ohne Zwischengrößen kann die CBO keine Kosten bestimmen.

3.

Selektivitätsmethoden

Histogramme, Modelle und Sampling liefern die nötigen Näherungen.

Damit können wir das Kernproblem der CBO angehen:
Die **im Kostenmodell günstigste Join-Reihenfolge** aus einer potenziell riesigen Menge möglicher Pläne auswählen.

Nächstes Thema: Join-Reihenfolgen bestimmen

5. Relationale Anfragebearbeitung

Kapitelüberblick: Innerhalb der Anfrageoptimierung verschiebt sich der Fokus jetzt auf die Wahl einer guten Join-Reihenfolge.

1. Anfragebearbeitung und Optimierung

- ▶ Motivation
- ▶ Einführung in Query Processing und Optimierungstypen
- ▶ Grundlagen: Regelbasierte Optimierung
- ▶ Grundlagen: Kostenbasierte Optimierung
- ▶ **Join-Reihenfolgen**

2. Basisalgorithmen

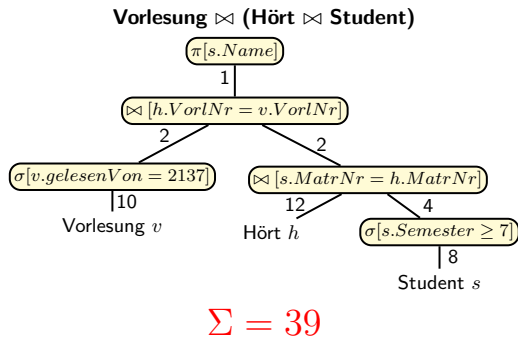
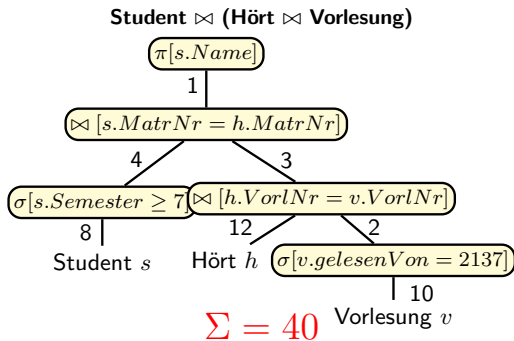
Physische Operatoren für Selektion, Projektion, Sortierung und Join.

3. Indexstrukturen

- ▶ Eindimensionale Indizes
- ▶ Mehrdimensionale Indizes

Jetzt im Fokus: Die CBO muss aus vielen korrekten Join-Plänen die günstigste Reihenfolge wählen.

Motivation: Wie wirkt sich die Join-Reihenfolge aus?



Beobachtung: Beide Pläne sind logisch äquivalent. Schon hier unterscheidet sich aber die Zahl der betrachteten Tupel leicht, und in großen Anfragen kann dieser Unterschied dramatisch werden.

? Frage: Anzahl der betrachteten Tupel für beide Planvarianten nachrechnen.

Join-Reihenfolgen (1) - Warum relevant?

CBO braucht

- ▶ gute Schätzungen für Kosten und Kardinalitäten
- ▶ eine Strategie, um unter vielen Plänen den im Kostenmodell günstigsten Plan zu finden

Ein Kernproblem: die Join-Reihenfolge

Warum gibt es viele mögliche Reihenfolgen?

- ▶ Algebraische Gesetze erlauben Umordnungen:

$$(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C) \quad A \bowtie B = B \bowtie A$$

Warum ist das teuer?

- ▶ Die Reihenfolge beeinflusst die Größe der Zwischenergebnisse und damit Speicher- und CPU-Bedarf.
- ▶ Sie beeinflusst auch, welche physischen Algorithmen gut einsetzbar sind, z.B. mit Indexnutzung.

Kernaussage: Die CBO muss die geschätzt günstigste Join-Reihenfolge finden.

Join-Reihenfolgen (2) - Der Suchraum (Bushy)

Wie viele Reihenfolgen gibt es für $R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$?

Ansatz: Beliebige binäre Baumstrukturen

$$\# \text{Pläne} = \# \text{Baumformen} \cdot \# \text{Relationen-Anordnungen}$$

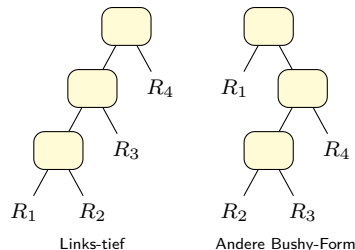
$$= C(m-1) \cdot m! = \frac{(2m-2)!}{(m-1)!}$$

mit $C(n)$ als Catalan-Zahl.

Wachstum ist explosiv

- ▶ $m = 3 \Rightarrow 12$ Pläne
- ▶ $m = 7 \Rightarrow 665,280$ Pläne
- ▶ $m = 10 \Rightarrow$

Beispiele für Planformen



Join-Reihenfolgen (2) - Der Suchraum (Bushy)

Wie viele Reihenfolgen gibt es für $R_1 \bowtie R_2 \bowtie \dots \bowtie R_m$?

Ansatz: Beliebige binäre Baumstrukturen

$\# \text{Pläne} = \# \text{Baumformen} \cdot \# \text{Relationen-Anordnungen}$

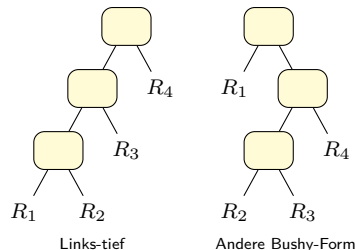
$$= C(m-1) \cdot m! = \frac{(2m-2)!}{(m-1)!}$$

mit $C(n)$ als Catalan-Zahl.

Wachstum ist explosiv

- ▶ $m = 3 \Rightarrow 12$ Pläne
- ▶ $m = 7 \Rightarrow 665,280$ Pläne
- ▶ $m = 10 \Rightarrow 17.6$ Milliarden Pläne

Beispiele für Planformen



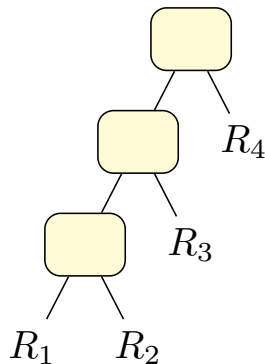
Folgerung: Vollständiges Durchsuchen aller Bushy-Pläne ist in realen Systemen meist viel zu teuer.

Join-Reihenfolgen (3) - Einschränkung: Left-Deep

Strategie: Reduziere den Suchraum, indem nur **Left-Deep Trees** zugelassen werden.

Struktur eines Left-Deep Trees

- ▶ Der rechte Operand eines Joins ist **immer** eine Basisrelation, niemals ein Zwischenergebnis.
- ▶ Visuell ergibt sich eine Kette nach links unten.
- ▶ Anzahl der Pläne sinkt auf $m!$, also nur auf die Permutationen der Relationen.
- ▶ Beispiel: $m = 10 \Rightarrow 3.6$ Millionen Pläne statt 17.6 Milliarden.



Vorteil: Einfachere Planstruktur, oft gut für Pipelining und Indexnutzung auf der rechten Seite.

Nachteil: Der global optimale Plan ist nicht garantiert left-deep; er könnte auch bushy sein.

Join-Reihenfolgen (4) - Dynamic Programming

Wie findet man den geschätzt günstigsten Plan systematisch, ohne alles vollständig zu enumerieren?

Dynamic Programming (DP) nach System-R-/Selinger-Idee:

Beste Teilpläne für kleine Mengen berechnen, speichern und zu grösseren Plänen kombinieren.

Grundidee

- ▶ Finde rekursiv die besten Pläne für kleine Teilmengen von Relationen.
- ▶ Kombiniere die besten Teilpläne zu Plänen für grössere Mengen.
- ▶ Speichere Kosten und Plan pro Teilmenge (*Memoization*).

Komplexität

Bushy Trees: $O(3^m)$ Left-Deep: $O(m2^m)$

Vergleich $m = 10$	Enumeration	DP
Bushy	> 17.6 Mrd.	ca. 59,000
Left-Deep	3.6 Mio.	ca. 10,000

? Frage: Warum kann DP unter den betrachteten Planformen eine optimale Lösung liefern?

? Frage: Wie kommt man intuitiv auf den Suchraumterm $O(3^m)$?

Fazit: Dynamic Programming ist deutlich besser als volle Enumeration, wird für sehr grosse Join-Mengen aber immer noch teuer.

Join-Reihenfolgen (5) - Heuristik: Greedy

Wenn exakte Suche per Dynamic Programming zu teuer wird, nutzen viele Systeme heuristische Suche.

Ziel: Schnell einen **guten**, aber nicht zwingend optimalen Plan finden.

Greedy-Idee: Wähle in jedem Schritt den lokal günstigsten Join gemäss den geschätzten Kosten; kleine Zwischenergebnisse (Selektivität) sind dabei oft nur ein nützlicher Indikator.

Greedy-Algorithmus für Left-Deep-Pläne auf $\{R_1, \dots, R_m\}$

- 1: Finde ein joinbares Paar (R_i, R_j) mit minimalen geschätzten Kosten für $R_i \bowtie R_j$.
- 2: $Plan \leftarrow R_i \bowtie R_j$, $Rest \leftarrow \{\text{alle Relationen}\} \setminus \{R_i, R_j\}$.
- 3: Solange $Rest \neq \emptyset$:
- 4: Wähle ein joinbares $R_k \in Rest$ mit minimalen geschätzten Kosten für $Plan \bowtie R_k$.
- 5: $Plan \leftarrow Plan \bowtie R_k$, $Rest \leftarrow Rest \setminus \{R_k\}$.
- 6: Gib $Plan$ zurück.

Vorteil: Laufzeit nur polynomial in m .

Nachteil: Keine Optimalitätsgarantie und anfällig für schlechte Kardinalitätsschätzungen.

Diskussionsfrage: Wann liefert Greedy dennoch den optimalen Plan? Wann nicht?

Join-Reihenfolgen - Fazit

Kernproblem der CBO: Finde die im Kostenmodell geschätzt günstigste Join-Reihenfolge.

Warum schwierig?

- ▶ Der Suchraum wächst exponentiell mit der Zahl der Relationen.
- ▶ Logisch äquivalente Pläne können physisch sehr unterschiedlich teuer sein.
- ▶ Schlechte Zwischenresultate treiben später Speicher- und I/O-Kosten.

Wie geht man damit um?

- ▶ **Suchraum reduzieren:** z. B. nur Left-Deep-Bäume zulassen.
- ▶ **Systematisch suchen:** Dynamic Programming speichert die besten Teilpläne.
- ▶ **Heuristisch suchen:** Greedy ist schnell, aber ohne Optimalitätsgarantie.

Kritisch für CBO

Gute **Kardinalitätsschätzungen** sind entscheidend. Typische Grundlage sind Statistiken wie Histogramme, Modelle oder Sampling.

Ausblick

Als nächstes geht es um **physische Algorithmen** für Selektion, Projektion und Join und danach um **Indexstrukturen**.

Merksatz: Ohne gute Schätzungen verliert jede Suchstrategie an Qualität.

5. Relationale Anfragebearbeitung

Kapitelüberblick: Nach der logischen Optimierung wechseln wir jetzt von der Planwahl zu den **physischen Operatoren**.

1. Anfragebearbeitung und Optimierung

- ▶ Motivation
- ▶ Query Processing und Optimierungstypen
- ▶ Grundlagen: Regelbasierte Anfrageoptimierung
- ▶ Grundlagen: Kostenbasierte Anfrageoptimierung
- ▶ Join-Reihenfolgen

2. Basialgorithmen

- ▶ Selektion
- ▶ Projektion
- ▶ Join
- ▶ Sortierung und Hilfsoperatoren

3. Indexstrukturen

- ▶ Eindimensionale Indizes
- ▶ Mehrdimensionale Indizes

Jetzt im Fokus: konkrete Operatoren für σ , π , Sortierung und $\bowtie$.

Physische Optimierung - Der nächste Schritt

Rückblick: Logischer Plan

- ▶ Reihenfolge der Operationen festgelegt
- ▶ Gute Join-Reihenfolge ausgewählt
- ▶ Selektionen und Projektionen früh angewandt

Nächste Stufe: Physische Optimierung

Physische Anfrageoptimierung wählt für den logischen Plan passende **Algorithmen und Zugriffspfade**, z. B.:

- ▶ Join: Hash Join oder Sort-Merge Join
- ▶ Selektion: Full Table Scan oder Index Scan
- ▶ Projektion: Sortieren oder Hashing für DISTINCT
- ▶ zusätzlich wichtig: Sortierung, Indizes und verfügbarer Speicher

Bei der Wahl zusammen betrachten: Kostenmodelle, Dateneigenschaften (Sortierung, Clusterung, Indizes) und Systemressourcen wie Hauptspeicher, CPU und I/O-Bandbreite.

Ziel: Finde den konkreten Ausführungsplan mit den geringsten erwarteten Gesamtkosten.

Kontext - Speicherhierarchie & I/O-Kosten

Warum ist die Wahl des Algorithmus so wichtig?

Zwischen Register, RAM, SSD und Festplatte liegen **große Größenordnungen** an Geschwindigkeitsunterschieden.

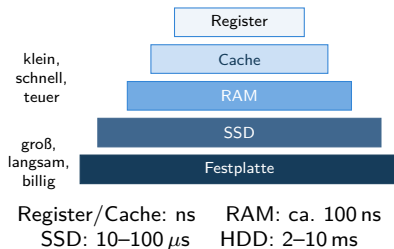
Beobachtung

- ▶ Daten liegen häufig auf SSD oder Festplatte; der Zugriff ist I/O-lastig.
- ▶ Bei HDDs dominieren bei kleinen, zufälligen Zugriffen **Seek** und **Rotationslatenz**.
- ▶ Bei SSDs dominieren eher **Latenz** und **Bandbreite/random I/O overhead**.
- ▶ Für HDD-Zugriffe kann man grob denken:
 $\text{Zugriffszeit} \approx \text{Suchzeit} + \text{Latenzzeit} + \text{Transferzeit}$.

Lösungsansatz: Blockorientierter Zugriff

- ▶ Daten werden in Seiten bzw. Blöcken gelesen und geschrieben (z. B. 4 KB oder 8 KB).
- ▶ Fixkosten eines I/Os werden so auf viele Tupel verteilt.
- ▶ Darum rechnen viele Algorithmen in **Block-I/Os** statt in Tupeln.

Speicherhierarchie (grob)



Ziel: Minimiere **Block-I/Os** zwischen RAM und SSD/HDD.

Algorithmen für Selektion (σ)

Ziel: Finde die Tupel, die eine WHERE-Bedingung erfüllen, z. B. `MatrNr = 12345` oder `Alter > 30`.

Option A1: Full Table Scan

- ▶ **Vorgehen:** Lies alle Blöcke von R und prüfe jedes Tupel.
- ▶ **Kosten:** $I/O \approx \# \text{Blöcke}(R) = b_R$
- ▶ **Gut wenn:** kein passender Index existiert oder die Bedingung viele Treffer liefert.

Option A2: Index Scan

- ▶ **Vorgehen:** Direkte Zugriffe über einen passenden Index.
- ▶ **Varianten:** Gleichheit oft via Hash oder B^+ -Baum; Bereiche typischerweise via B^+ -Baum.
- ▶ **Kostenidee:** Indexhöhe + Datenzugriffe; bei unclusterter Ablage können viele Treffer teuer werden.
- ▶ **Gut wenn:** passender Index vorhanden, die Bedingung selektiv ist und nur wenige Datenblöcke gelesen werden müssen.

Merksatz: CBO wählt zwischen Scan und Indexzugriff nach Selektivität und verfügbaren Indizes.

Algorithmen für Projektion (π mit DISTINCT)

Ziel: Führe `SELECT DISTINCT A, B, ... FROM R` effizient aus.

Zwei Teilaufgaben

1. Nur die benötigten Attribute behalten.
2. Duplikate aus den Ergebnis-Tupeln entfernen.

Option B1: Sortieren

1. Lies R und projiziere auf die Zielattribute.
2. Sortiere die Zwischenrelation.
3. Scanne das sortierte Ergebnis und behalte nur das erste Vorkommen.

Kostenidee: Dominiert durch den Sortieraufwand, besonders bei externem Sortieren.

Option B2: Hashing

1. Lies R und projiziere auf die Zielattribute.
2. Berechne für jedes Ergebnis-Tupel einen Hashwert.
3. Füge in eine Hashtabelle ein und verwirfe erkannte Duplikate.
4. Falls die Hashtabelle nicht in den Speicher passt: zunächst partitionieren (externes Hashing).

Kostenidee: Oft nahe linear, solange die Hashtabelle gut in den Speicher passt; teurer bei Kollisionen oder externem Hashing.

Ohne DISTINCT ist Projektion fast trivial; **teuer** wird erst die Duplikateliminierung.

Join-Algorithmen (1) - Einführung & Annahmen

Joins gehören oft zu den teuersten Operatoren. Fokus hier: **Equi-Join** $R \bowtie_{R.A=S.B} S$.

Notation für Beispiele

- ▶ R = äußere Relation, S = innere Relation
- ▶ b_R, b_S = Anzahl der Blöcke von R bzw. S
- ▶ $|R|, |S|$ = Anzahl der Tupel
- ▶ Im einfachen Modell: möglichst wenige **Block-I/Os**

Überblick über wichtige Verfahren

- ▶ **Nested Loop Join (NLJ)** und **Block Nested Loop Join (BNLJ)**
- ▶ **Sort-Merge Join (SMJ)**
- ▶ **Hash Join (HJ)**
- ▶ **Index Nested Loop Join (INLJ)**

Merksatz: Kein Join-Algorithmus ist universell am besten; die Wahl hängt von Datenverteilung, Indizes, Sortierung und Speicher ab.

Join-Algorithmen (2) - Nested Loop Join (simpel)

Algorithmus (tupelbasiert)

- 1: Für jedes Tupel $r \in R$:
- 2: Für jedes Tupel $s \in S$:
- 3: Falls $r.A = s.B$: gib (r, s) aus.

Intuition

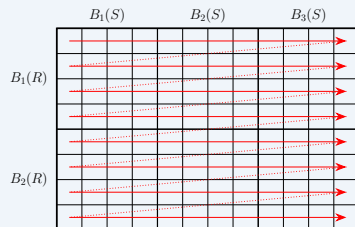
- ▶ Es wird effektiv das kartesische Produkt in Reihenfolge erzeugt.
- ▶ Die Join-Bedingung filtert passende Paare erst beim Vergleich heraus.
- ▶ Für jedes Tupel aus R wird S erneut komplett durchsucht.

Kosten (I/O)

- ▶ Lesen von R : b_R Blöcke
- ▶ Lesen von S : $|R| \cdot b_S$ Blöcke
- ▶ Total: $b_R + |R| \cdot b_S$ I/Os

Fazit: Für realistische Datenmengen meist unakzeptabel teuer und nur als Ausgangspunkt für bessere Varianten interessant.

Matrixnotation



Für jedes Tupel in einem Block von R wird ganz S durchlaufen.

Join-Algorithmen (3) - Block Nested Loop Join

Verbesserungsidee: Nicht tupelweise, sondern **blockweise** arbeiten.

Algorithmus (blockbasiert)

- 1: Für jeden Block $B_R \in R$:
- 2: Lade B_R in den Speicher.
- 3: Für jeden Block $B_S \in S$:
- 4: Lade B_S und vergleiche Tupelpaare aus B_R und B_S .

Wirkung

- ▶ Jeder Block aus R wird genau einmal gelesen.
- ▶ S wird nur noch pro **Block** von R erneut gescannt.
- ▶ Der Faktor $|R|$ schrumpft damit auf b_R .

Kosten (I/O)

- ▶ Lesen von R : b_R IOPS
- ▶ Lesen von S : $b_R \cdot b_S$ IOPS
- ▶ Total: $b_R + b_R \cdot b_S$ IOPS

Faustregel: Wähle möglichst die kleinere Relation als **äußere Relation**. Dann wird $b_R + b_R \cdot b_S$ kleiner.

Matrixnotation



Join-Algorithmen (4) - Block Nested Loop Join und Cache-Nutzung

Ziel: Reduziere den teuren Term $b_R \cdot b_S$ durch bessere Nutzung des Hauptspeicher-Puffers.

Grundidee

- ▶ Halte möglichst viele Blöcke der **kleineren Relation** (z. B. R) im Cache.
- ▶ Falls R nicht komplett passt: Lade mehrere Blöcke von R auf einmal (*Chunks*).
- ▶ Scanne S nur einmal pro Chunk statt einmal pro Einzelblock von R .

Idealfall

- ▶ Die kleinere Relation R passt komplett in den Puffer:
 $b_R \leq \text{CacheSize} - 1$
- ▶ Lade R einmal, scanne S blockweise und joine im Speicher.
- ▶ Dann sinken die I/O-Kosten auf ungefähr $b_R + b_S$.

Warum CacheSize - 1?

Mindestens **ein** Pufferblock wird noch für die aktuell gelesenen Blöcke von S oder für Ausgabeseiten benötigt. Deshalb kann man von R nicht alle Cache-Rahmen belegen (im hier gezeigten vereinfachten Modell ohne separaten Ausgabepuffer)

Realität

Mehr Hauptspeicher kann Block Nested Loop Join deutlich verbessern; die tatsächliche Ersparnis hängt aber von Puffergröße, Datenanordnung und Caching-Strategie ab.

Join-Algorithmen (5) - Sort-Merge Join (SMJ)

Idee: Sind beide Relationen nach dem Join-Attribut sortiert, kann man sie beim Join effizient *mergen*.

Merge-Algorithmus

- 1: Falls nötig: sortiere R nach A und S nach B
- 2: Setze **cursorR** und **cursorS** auf das erste Tupel
- 3: **while** **cursorR** und **cursorS** gültig sind **do**
- 4: **if** $R.A < S.B$:
- 5: bewege **cursorR**
- 6: **else: if** $R.A > S.B$:
- 7: bewege **cursorS**
- 8: **else:**
- 9: kombiniere passende Tupel(gruppen) und gib Join-Tupel aus
- 10: bewege die Cursor auf die nächsten noch nicht verarbeiteten Werte

Kostenintuition

Sind beide Relationen schon sortiert, ist die Merge-Phase fast linear in $b_R + b_S$. Sonst dominieren oft die Sortierkosten.

Geeignet: wenn Sortierung schon vorliegt oder ohnehin benötigt wird.

Sort-Merge Join: Beispiel 1 - Annahme R und S sind sortiert

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Status

cursorR zeigt auf (1, 10.000)

cursorS zeigt auf (1, Müller)

Start: Beide Relationen sind sortiert und die Cursor stehen auf den ersten Tupeln.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

noch leer

Sort-Merge Join: Beispiel 2

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (1, 10.000)

cursorS auf (1, Müller)

Aktion: 1 = 1, also Match.

Erzeuge (1, 10.000, Müller).

cursorR bleibt, da in *S* noch ein weiteres Tupel mit Wert 1 folgt.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller

Sort-Merge Join: Beispiel 3

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR bleibt auf (1, 10.000)
cursorS steht nun auf
(1, Schuster)
Aktion: **cursorS** wurde auf den
nächsten gleichen Join-Wert
bewegt; **cursorR** bleibt auf 1.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller

Sort-Merge Join: Beispiel 4

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (1, 10.000)

cursorS auf (1, Schuster)

Aktion: 1 = 1, also weiterer
Match.

Erzeuge (1, 10.000, Schuster).

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel 5

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (1, 10.000)
cursorS auf (2, Schneider)
Aktion: $1 < 2$.
Deshalb wird als nächstes
cursorR weiterbewegt.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel 6

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (2, 20.000)

cursorS auf (2, Schneider)

Aktion: Beide Cursor sind wieder auf demselben Join-Wert; im nächsten Schritt wird der Match erzeugt.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster

Sort-Merge Join: Beispiel 7

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (2, 20.000)
cursorS auf (2, Schneider)
Aktion: 2 = 2, also Match.
Erzeuge (2, 20.000, Schneider).

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider

Sort-Merge Join: Beispiel 8

Relation R

Gehalts- gruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR bleibt auf (2, 20.000)
cursorS steht nun auf
(2, Schmidt)
Aktion: **cursorS** wurde auf den
nächsten gleichen Join-Wert
bewegt; **cursorR** bleibt auf 2.

Relation S

Gehalts- gruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehalts- gruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider

Sort-Merge Join: Beispiel 9

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR auf (2, 20.000)

cursorS auf (2, Schmidt)

Aktion: 2 = 2, also weiterer Match.

Erzeuge (2, 20.000, Schmidt).

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

Bisherige Ausgabe

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider
2	20.000	Schmidt

Sort-Merge Join: Beispiel 10

Relation R

Gehaltsgruppe	Gehalt
1	10.000
2	20.000
3	30.000

Cursor / Aktion

cursorR steht nun auf (3, 30.000)

cursorS = Ende

Aktion: Die Duplikatgruppe zu 2 ist vollständig verarbeitet.

Da **cursorS** kein weiteres Tupel mehr hat, endet der Merge.

Relation S

Gehaltsgruppe	Angestellter
1	Müller
1	Schuster
2	Schneider
2	Schmidt

cursorS = Ende

Bisherige Ausgabe

Gehaltsgruppe	Gehalt	Angestellter
1	10.000	Müller
1	10.000	Schuster
2	20.000	Schneider
2	20.000	Schmidt

Join-Algorithmen (6) - SMJ Analyse

Kosten (I/O)

- ▶ Sortierphase nur nötig, wenn die Eingaben nicht vorsortiert sind
- ▶ Externes Sortieren ist dann oft der dominante Kostenanteil
- ▶ Die Merge-Phase selbst ist sehr effizient
- ▶ Im Idealfall reicht danach ein linearer Scan mit $b_R + b_S$
- ▶ Viele Duplikate können den Aufwand leicht erhöhen

Wann ist SMJ eine gute Wahl?

- ▶ wenn eine oder beide Relationen bereits sortiert vorliegen
- ▶ wenn das Join-Ergebnis später in sortierter Form benötigt wird
- ▶ Varianten von Merge Join können auch für bestimmte Bereichs- und Nicht-Equi-Joins genutzt werden

Merksatz

SMJ ist vor allem dann stark, wenn die Sortierung schon vorhanden ist. Teuer wird meist nicht das Mergen, sondern das Vorsortieren.

Join-Algorithmen (7) - Hash Join (HJ)

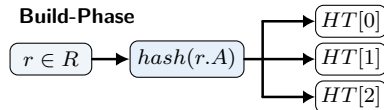
Grundidee: Nutze eine Hashtabelle für schnellen Zugriff auf die kleinere Relation. **Hash Join ist nur für Equi-Joins geeignet.**

Hash-Join-Algorithmus

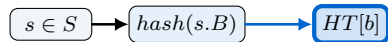
- 1: Wähle die kleinere Relation R als Build-Input
- 2: Erzeuge eine leere In-Memory-Hashtabelle HT
- 3: **for all** Tupel $r \in R$ **do**
- 4: $b \leftarrow \text{hash}(r.A)$
- 5: Füge r in Bucket $HT[b]$ ein
- 6: **for all** Tupel $s \in S$ **do**
- 7: $b \leftarrow \text{hash}(s.B)$
- 8: **for all** $r \in HT[b]$ **do**
- 9: **if** $r.A = s.B$:
- 10: gib (s, r) aus

Build- und Probe-Idee

Build-Phase



Probe-Phase



nur diesen
Bucket auf Gleichheit
prüfen.

Intuition: Build verteilt R auf Buckets;
Probe prüft für Tupel aus S
nur den **passenden** Bucket.

Join-Algorithmen (8) - HJ Analyse

Kosten im Idealfall

Falls die Build-Relation S in den Hauptspeicher passt:

$$\text{Build: } b_S \quad \text{Probe: } b_R \quad \text{Total } \approx b_R + b_S$$

Fazit: Hash Join kann in diesem Fall sehr schnell sein.

Voraussetzungen / Einschränkungen

- ▶ nur für Equi-Joins
- ▶ gute Hashfunktion wichtig, damit wenige Kollisionen entstehen
- ▶ die kleinere Relation sollte als Hashtabelle in den RAM passen
- ▶ die reale Laufzeit hängt stark von Speicher und Datenverteilung ab

Wenn die Hashtabelle zu groß ist

- ▶ **Grace Hash Join:** partitioniere R und S auf Platte nach Hashwerten und joine die Partitionen einzeln
- ▶ **Hybrid Hash Join:** kombiniert In-Memory-Buckets mit Partitionierung auf Platte
- ▶ **Mehrkosten:** zusätzliche I/Os für die Partitionierung

Join-Algorithmen (9) - Index Nested Loop Join (INLJ)

Idee: Falls für die innere Relation S ein Index auf dem Join-Attribut B existiert, nutze diesen für gezielte Nachschläge statt eines vollen Scans.

Algorithmus

```
1: for all Tupel  $r \in R$  do
2:    $lookup\_value \leftarrow r.A$ 
3:   nutze Index auf  $S.B$ 
4:   finde Tupel  $s$  mit  $s.B = lookup\_value$ 
5:   for all gefundene Tupel  $s$  do
6:     gib  $(r, s)$  aus
```

Kosten (I/O)

- ▶ Scan von R : b_R I/Os
- ▶ Index-Zugriffe auf S : $|R| \cdot Cost(Index\ Probe)$
- ▶ Total: $b_R + |R| \cdot Cost(Index\ Probe)$

Wann ist INLJ gut?

- ▶ wenn die äußere Relation R klein ist
- ▶ wenn auf $S.B$ ein effizienter, selektiver Index existiert
- ▶ wenn einzelne Treffer statt großer Vollscans benötigt werden

Risiko

Ist R groß, entstehen viele Index-Lookups. Bei unclusterter Ablage verursachen viele Treffer teure zufällige Datenzugriffe. Dann wird INLJ oft teurer als block- oder hashbasierte Verfahren.

Join-Algorithmen (10) - Vergleich & Wahl

Überblick & Vergleich (stark vereinfacht)

Algorithmus	Kosten-Typ (Ideal)	Index?	Memory?	Bemerkung
BNLJ	$\sim b_R + b_R \cdot b_S$	Nein	Min/Med	Basisverfahren
BNLJ+Cache	$\sim b_R + b_S$	Nein	Hoch	nur im Idealfall
SMJ	$Sort + b_R + b_S$	Hilft	Hoch	Ausgabe sortiert
Hash Join	$\sim b_R + b_S$	Nein	Hoch (B)	nur Equi-Join
INLJ	$\sim b_R + R \cdot Cost(Index\ Probe)$	Ja (S)	Gering	stark bei kleinem R

Risiko

Die Tabelle zeigt nur **Idealwerte**. Reale Kosten hängen von Speicher, Sortierung, Indizes und Datenverteilung ab.

Legende

(S) = Index auf innerer Relation S
(B) = Hauptspeicher für Build-Phase

- ▶ Der Optimierer schätzt Kosten für alle **anwendbaren** Join-Algorithmen.
- ▶ Gewählt wird das Verfahren mit den **minimalen geschätzten Kosten** im konkreten Gesamtplan.
- ▶ Nächstes Thema: Indexstrukturen, die Verfahren wie INLJ erst möglich machen.

5. Relationale Anfragebearbeitung

Kapitelüberblick: Nach den Join-Algorithmen wechseln wir jetzt von den physischen Operatoren zu den Indexstrukturen.

1. Anfragebearbeitung und Optimierung

- ▶ Motivation und Query Processing
- ▶ regelbasierte Optimierung
- ▶ kostenbasierte Optimierung
- ▶ Join-Reihenfolgen

2. Basisalgorithmen

- ▶ Selektion
- ▶ Projektion
- ▶ Join
- ▶ Sortierung und Hilfsoperatoren

3. Indexstrukturen

- ▶ **Eindimensionale Indizes**
- ▶ Mehrdimensionale Indizes

Jetzt im Fokus: Wie Indizes direkte Zugriffe ermöglichen und damit Verfahren wie INLJ unterstützen.

1D-Indexstrukturen - Warum & Was?

Rückblick

Datensätze liegen häufig auf Festplatten oder SSDs. Der Zugriff erfolgt blockweise und ist deutlich langsamer als RAM-Zugriff.

Motivation und Ziel

- ▶ Full Table Scans sind für selektive Anfragen zu langsam, etwa bei `WHERE ID = 123` oder `WHERE Alter > 60`
- ▶ Ziel ist ein schneller, direkter Zugriff auf relevante Daten über Attributwerte
- ▶ Gute Indexstrukturen minimieren die Zahl der benötigten I/Os

Zwei Einordnungsachsen

1. Nach indexiertem Attribut

- ▶ **Primärindex:** Index über dem Primärschlüssel; Werte sind eindeutig
- ▶ **Sekundärindex:** Index über andere Attribute; Duplikate sind möglich

2. Nach physischer Ablage

- ▶ **Clustered Index:** Daten liegen in der Reihenfolge des Indexschlüssels
- ▶ **Unclustered Index:** Der Index enthält nur Verweise auf unsortiert gespeicherte Datensätze

Prinzipien der blockweisen Speicherung von Daten

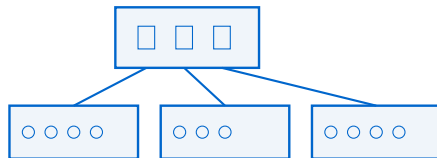
Block- oder seitenweise Speicherung

- ▶ viele Datensätze werden auf derselben Seite gespeichert
- ▶ für effiziente Suche werden Daten mit ähnlichen Werten in suchfreundlichen Organisationen möglichst nah beieinander abgelegt
- ▶ wird eine Seite zu voll, wird sie auf mehrere Seiten aufgespalten



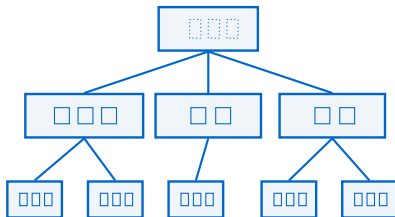
Hierarchische Organisation

- ▶ übergeordnete Knoten beschreiben und erschließen mehrere Datenblöcke



Von der blockweisen Speicherung zu mehrstufigen Hierarchien (Bäume)

Rekursive Aufspaltung liefert Baumstruktur



Suchprinzip

- ▶ Seiten werden hierarchisch statt linear erschlossen.
- ▶ Knoten enthalten Suchinformationen und Zeiger.
- ▶ Jeder Schritt zerlegt den Suchraum in kleinere Bereiche.

Kernaussage

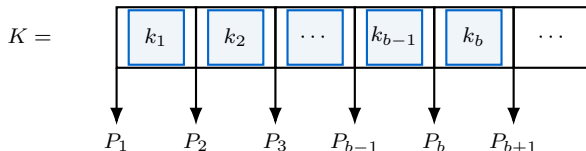
Knoten strukturieren den Suchraum; Blätter sind Endpunkte der Verzweigung.

Struktureigenschaften

- ▶ Blattknoten sind Knoten ohne weitere Nachfolger.
- ▶ Die Baumhöhe bestimmt die Anzahl gelesener Seiten.
- ▶ Jeder Knoten hat nur beschränkt viele Einträge.
- ▶ Balancierung und Belegungsregeln folgen beim B-Baum.

Charakterisierung

- ▶ Knoten mit M Einträgen haben bis zu $M + 1$ Nachfolger
- ▶ die Bäume sind blockorientiert: ganze Knoten liegen auf Plattenseiten
- ▶ damit ergibt sich M aus Seitengröße sowie Größe von Schlüsseln und Zeigern



P_1 zeigt auf Werte $< k_1$, P_i auf Werte zwischen k_{i-1} und k_i , P_{b+1} auf Werte $> k_b$.

Knoten K eines $(M + 1)$ -Wege-Suchbaums besteht aus:

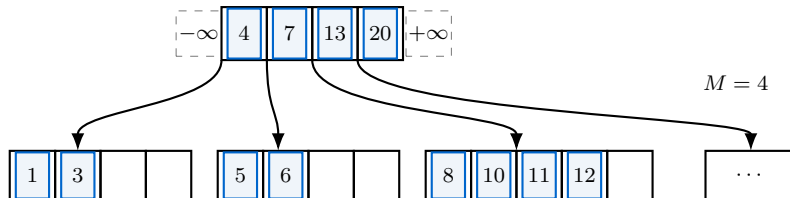
- ▶ Verzweigungsgrad $b + 1 = \text{Grad}(K) \leq M + 1$
- ▶ Datensätzen mit Schlüsseln k_i für $1 \leq i \leq b$
- ▶ Zeigern P_i auf die Unterbäume für $1 \leq i \leq b + 1$

Mehrweg-Suchbäume

Suchbaumeigenschaft für Mehrwegebäume

- ▶ Die Schlüssel $k_1, k_2, \dots, k_b$ in einem Knoten K sind geordnet, also $k_i \leq k_{i+1}$ für $i = 1, \dots, b - 1$.
- ▶ Für jeden Teilbaum gilt ein Intervall: $k_{p-1} < k' \leq k_p$ mit $k_0 = -\infty$ und $k_b = +\infty$.

Beispiel



Definition B-Baum

Definition

Ein **B-Baum** ist ein selbstbalancierender, geordneter $(M + 1)$ -Wege-Suchbaum.

1. **Balanciertheit:** Alle Blattknoten befinden sich auf derselben Ebene.
2. **Knotengröße (Schlüsselanzahl):**
 - ▶ Maximum: M Schlüssel pro Knoten
 - ▶ Minimum außer Wurzel: $\lfloor M/2 \rfloor$ Schlüssel
 - ▶ Wurzel: mindestens ein Schlüssel, sofern der Baum nicht leer ist
3. **Verzweigung:** Jeder innere Knoten mit k Schlüsseln besitzt genau $k + 1$ Kinder.

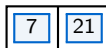
Konsequenz: B-Bäume ermöglichen Suche, Einfügen und Löschen in $O(\log_{m+1} N)$ I/Os, wobei $m = \lfloor M/2 \rfloor$ die minimale Schlüsselanzahl pro Nicht-Wurzelknoten bezeichnet.

Nur die Schlüssel werden betrachtet; reale Einträge enthalten Schlüssel plus Wert oder Zeiger.

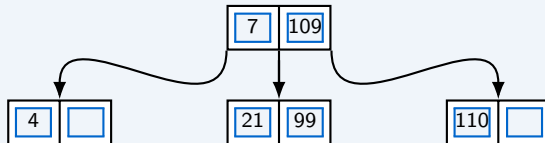
Beispiele für B-Bäume

Beispiele für B-Bäume mit $M = 2$ (also maximal drei Nachfolger)

Beispiel Höhe 1



Beispiel Höhe 2



Suche nach einem Datensatz

- ▶ beginne in der Wurzel
- ▶ suche auf jedem Knoten binär nach dem passenden Schlüssel oder Intervall
- ▶ falls nicht gefunden, steige in den passenden Teilbaum ab
- ▶ wiederhole dies rekursiv bis zum Blatt

Komplexität der Suche

- ▶ Höhe h : maximal h besuchte Knoten
- ▶ Baumhöhe bei N Objekten: typischerweise $h \approx \log_m N$
- ▶ binäre Suche auf einem Knoten: höchstens $\log_2 M$ Vergleiche
- ▶ insgesamt: etwa $\log_2 M \cdot \log_m N \approx \log_2 N$
- ▶ Plattenzugriffe: höchstens $\log_m N$

Beispiel: Für $N = 10^6$ und $M = 100$ ergeben sich ungefähr 20 Vergleiche und 3 Plattenzugriffe, wenn die Wurzel im Cache liegt.

Bereichsanfrage im B-Baum

Suche Objekte, die in einen Bereich $[min, max]$ fallen

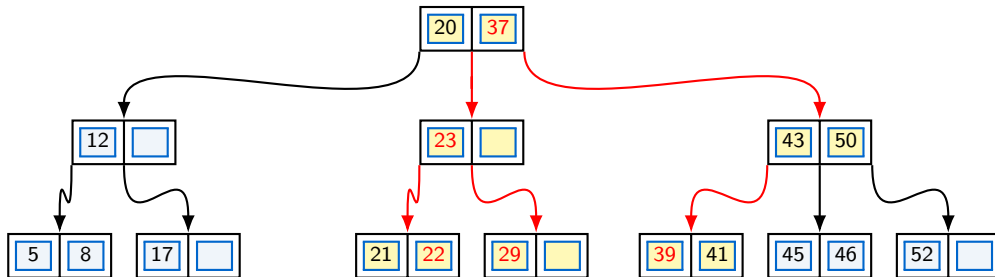
- ▶ finde rekursiv den kleinsten Eintrag $e \geq min$
- ▶ laufe von dort in Inorder-Reihenfolge bis zum grössten Eintrag $e' \leq max$
- ▶ sei r die Zahl der gefundenen Elemente

Kosten

Vergleiche: $O(r + \log_2 N)$

Plattenzugriffe: $O(r/m + \log_m N)$

Beispiel: $[22, 40]$



Einfügen im B-Baum

Ablauf als Algorithmus

- 1: Suche das zuständige Blatt wie bei einer normalen Suche.
- 2: Füge den neuen Wert im Blatt sortiert ein.
- 3: Bei Überlauf (Anzahl Schlüssel $> M$): splitte den Knoten in zwei Hälften.
- 4: Lasse die linke Hälfte im alten Knoten und verschiebe die rechte in einen neuen Knoten.
- 5: Schiebe den mittleren Schlüssel in den Elternknoten hoch.
- 6: Wiederhole den Split bei Bedarf bis zur Wurzel; überläuft die Wurzel, entsteht eine neue Wurzel.

Beispiel

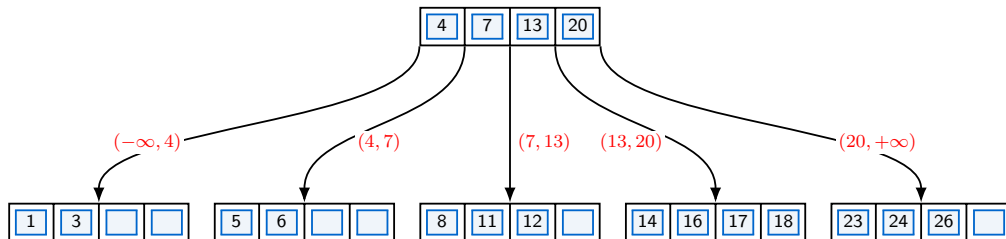
Beispiel:



Merke: Der mittlere Schlüssel steigt auf; ohne Elternknoten entsteht eine neue Wurzel. **Kosten:** $O(\log_m N)$.

B-Baum: Beispiel Einfügen

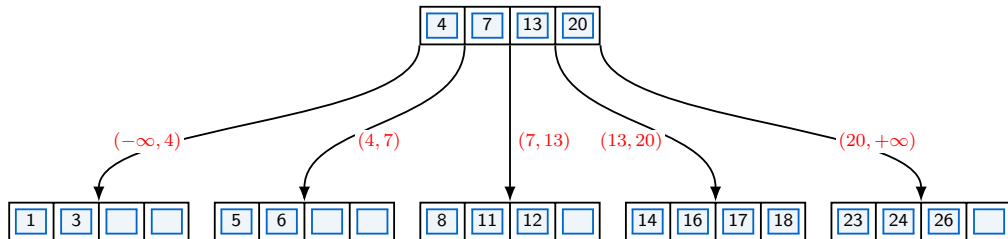
Suche das passende Blatt über die Intervallgrenzen der Wurzel.



B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

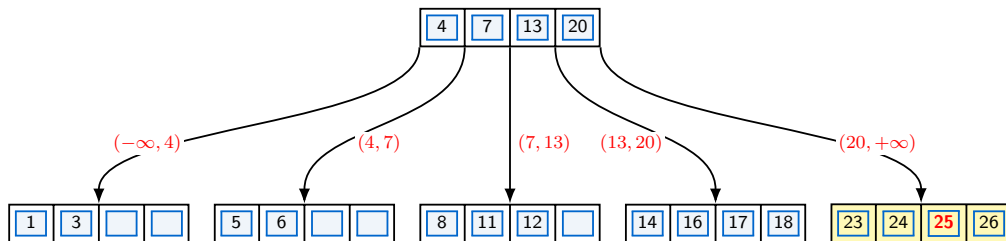
Einfügen: 25



B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

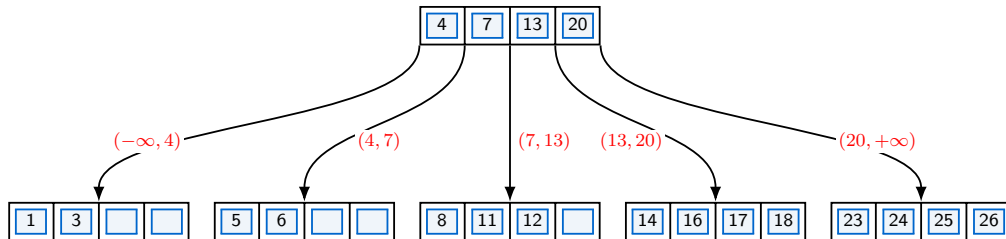


B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19

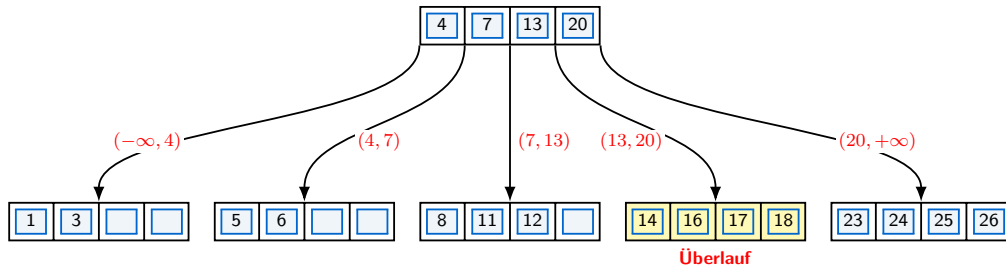


B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19

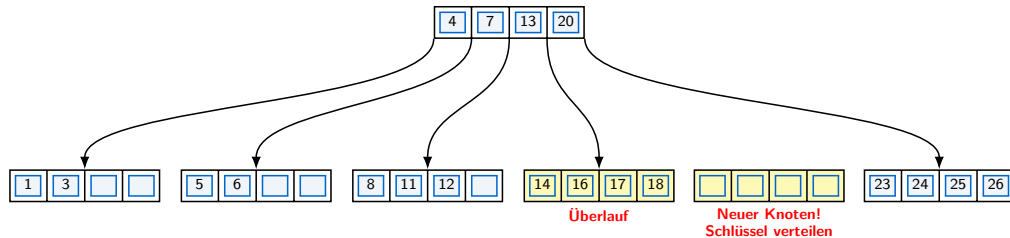


B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19

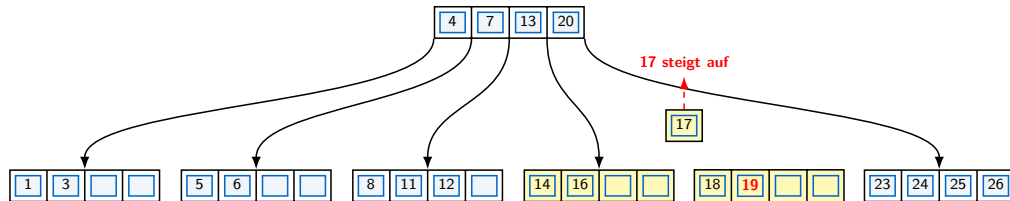


B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19

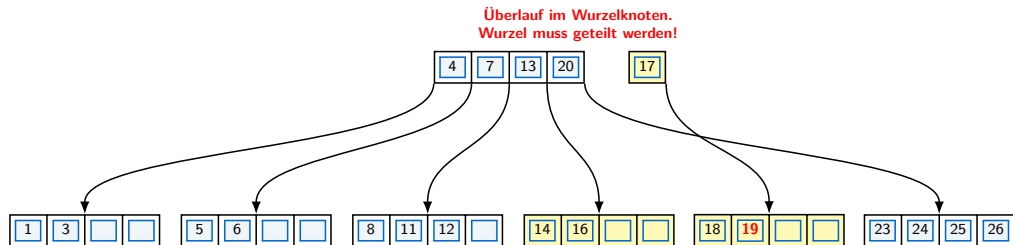


B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19



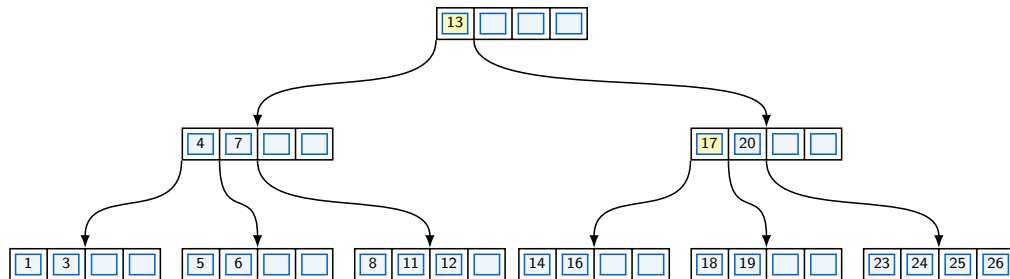
B-Baum: Beispiel Einfügen

Suche das passende Blatt über die Intervallgrenzen der Wurzel.

Einfügen: 25

Einfügen: 19

Das mittlere Element (13)
wird die neue Wurzel.

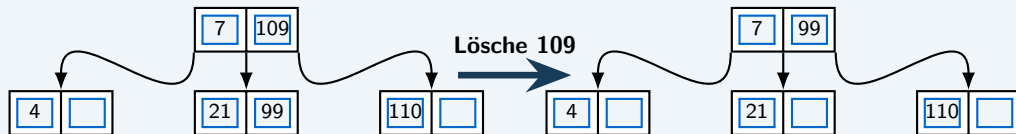


Löschen in B-Bäumen (Algorithmus Teil 1)

Phase 1: Schlüssel entfernen

- 1: Finde Knoten K , der Schlüssel k enthält.
- 2: **if** K ist innerer Knoten :
 - 3: Finde Vorgänger/Nachfolger k' im Blatt, ersetze k in K durch k' . $k \leftarrow k'$, $K \leftarrow \text{Blatt}$.
- 4: Entferne k aus Blattknoten K .
- 5: **if** $|K.\text{keys}| < \lfloor M/2 \rfloor$: REORGANISATION(K) # Unterlauf!

Visuelles Beispiel



Löschen in B-Bäumen (Algorithmus Teil 2)

Phase 2: Reorganisation bei Unterlauf

```
1: def REORGANISATION( $K$ ):
2:   if  $K$  ist die Wurzel :
3:     if  $|K.keys| == 0$  and  $K$  hat Kinder: Wurzel  $\leftarrow$  Kind von  $K$ 
4:     return
5:   Sei  $E$  der Elternknoten von  $K$ ,  $N$  ein Nachbarknoten.
6:   if  $|N.keys| > \lfloor M/2 \rfloor$  :                                # Nachbar ist „reich“  $\rightarrow$  Rotation
7:     Verschiebe Trennschlüssel aus  $E$  nach  $K$ .
8:     Verschiebe angrenzenden Schlüssel aus  $N$  nach  $E$ .
9:     Hänge das zugehörige Kind von  $N$  an  $K$  um.
10:  else:                                                         # Nachbar ist „arm“  $\rightarrow$  Verschmelzen (Merge)
11:    Verschmelze  $K$ ,  $N$  und den Trennschlüssel aus  $E$ .
12:    Entferne Trennschlüssel und Zeiger aus  $E$ .
13:    if  $|E.keys| < \lfloor M/2 \rfloor$ : REORGANISATION( $E$ )                # Kaskade!
```

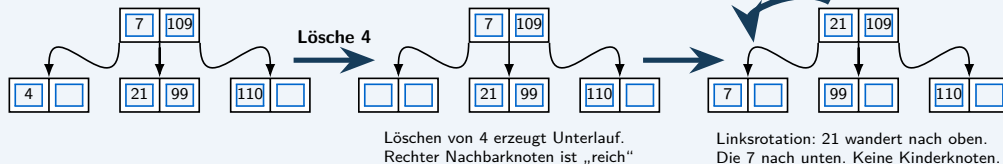
Zusammenfassung der Reorganisation:

- ▶ **Rotation (Ausleihen):** Wenn Nachbar „reich“ ist. Verhindert Reduzierung der Baumhöhe.
- ▶ **Verschmelzen (Merge):** Wenn Nachbar „arm“ ist. Baum verkleinert sich (ggf. reduzierte Höhe).

Löschen in B-Bäumen (Unterlauf & Rotation)

Beispiel: Löschen des Schlüssels 4. Dies erzeugt einen Unterlauf, der durch „Ausleihen“(Rotation) behoben wird.

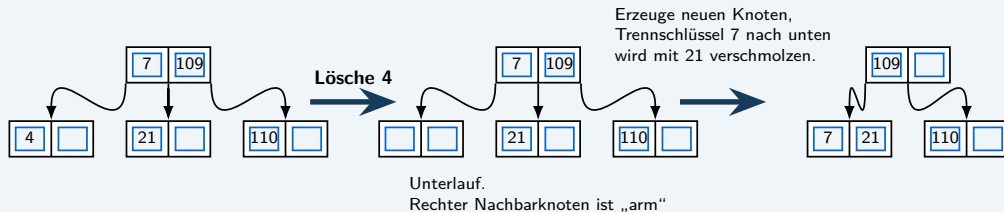
Ausgleich durch Ausleihen (Rotation)



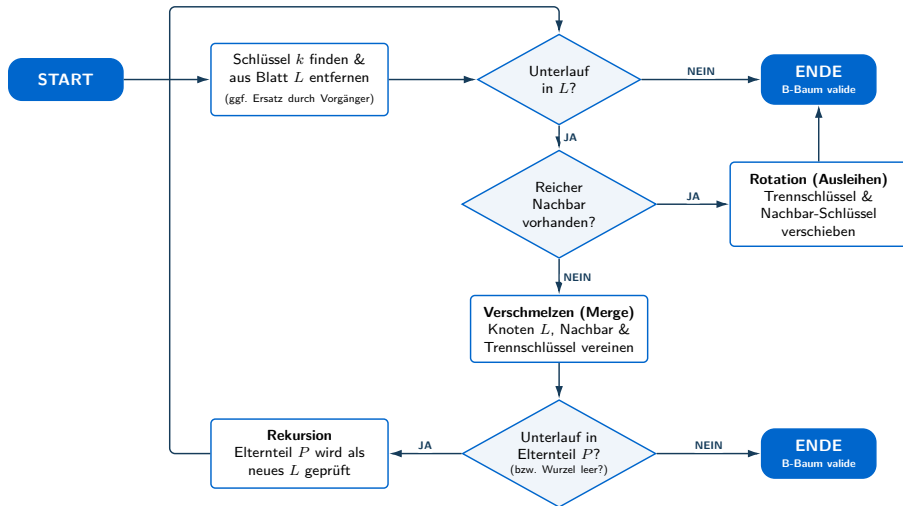
Löschen in B-Bäumen (Unterlauf & Verschmelzen)

Beispiel: Löschen des Schlüssels 4. Dies erzeugt einen Unterlauf. Der rechte Nachbarknoten ist „arm“ (nur 1 Schlüssel), daher wird verschmolzen.

Ausgleich durch Verschmelzen (Merge)



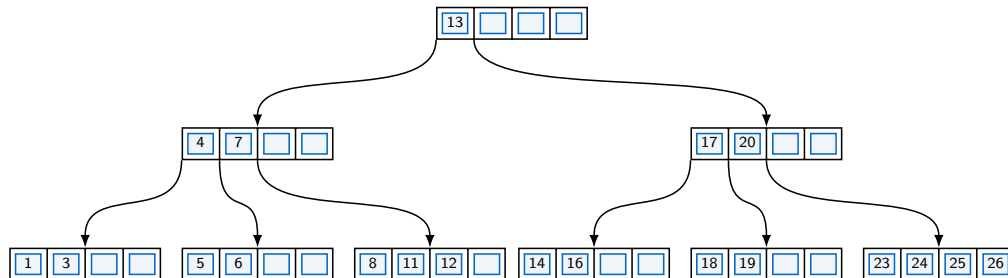
Flussdiagramm Löschen



B-Baum: Beispiel Löschen

Ausgangszustand nach den Einfüge-Operationen.

Wir löschen nun schrittweise Elemente aus diesem B-Baum.

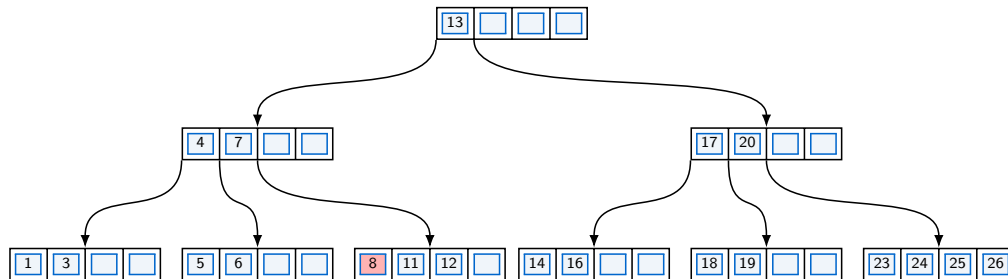


B-Baum: Beispiel Löschen

Fall 1: Einfaches Löschen aus einem Blatt ohne Unterlauf

Löschen: 8

Der Knoten hat 3 Elemente. Da $3 > k$ (mit Minimalbelegung $k = 2$), entsteht kein Unterlauf. Wir können direkt löschen.

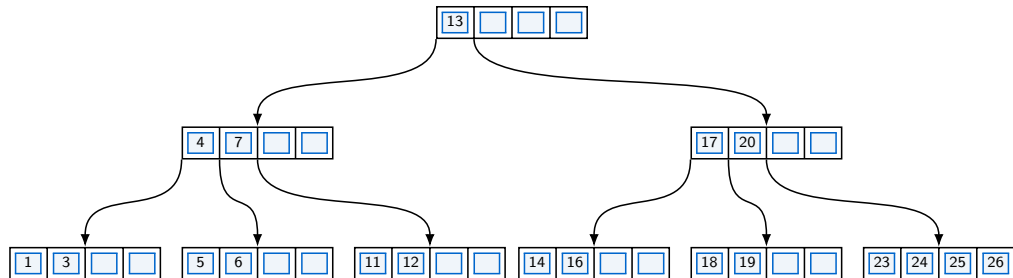


B-Baum: Beispiel Löschen

Fall 1: Einfaches Löschen aus einem Blatt ohne Unterlauf

Ergebnis nach Löschen von 8

Das Element wurde entfernt. Die verbleibenden Elemente rücken auf. Der Knoten erfüllt weiterhin die Baum-Eigenschaften.

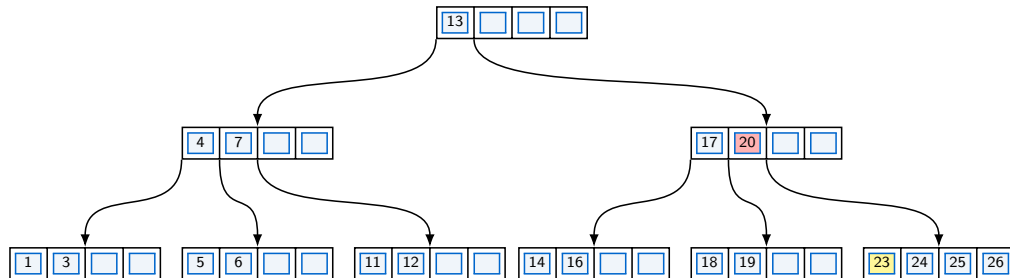


B-Baum: Beispiel Löschen

Fall 2: Löschen aus einem inneren Knoten

Löschen: 20

Die 20 ist ein innerer Knoten. Wir müssen sie durch ihren symmetrischen Vorgänger (19) oder Nachfolger (23) ersetzen. Wir wählen die 23, da ihr Knoten reichlich belegt ist (4 Elemente).

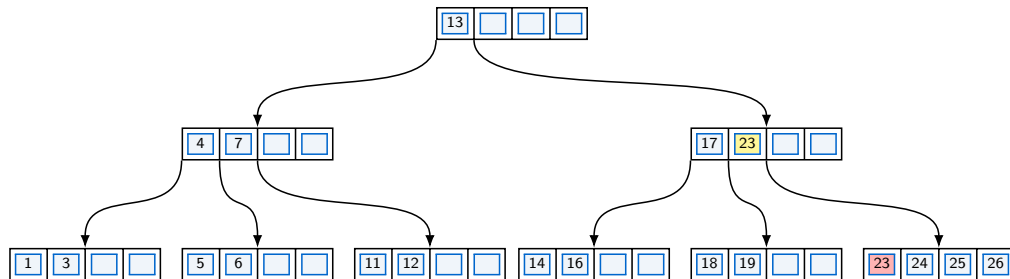


B-Baum: Beispiel Löschen

Fall 2: Löschen aus einem inneren Knoten

Schritt 1: Nachfolger kopieren

Die 23 ersetzt die 20 im inneren Knoten. Das Löschproblem verschiebt sich nun in den Blattknoten: Wir müssen die ursprüngliche 23 aus dem Blatt löschen.

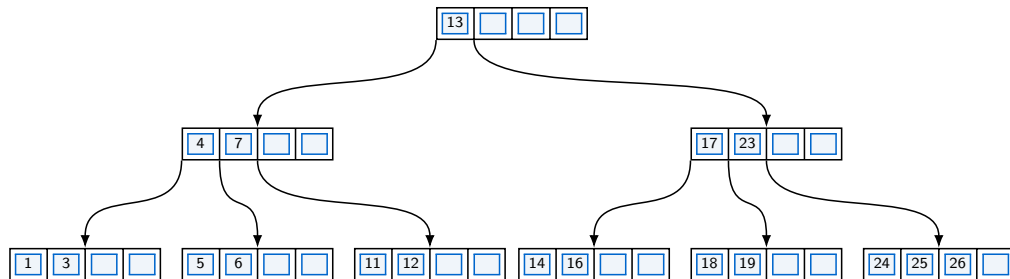


B-Baum: Beispiel Löschen

Fall 2: Löschen aus einem inneren Knoten

Schritt 2: Element im Blatt löschen

Die 23 wird aus dem Blatt gelöscht. Da der Knoten noch 3 Elemente hat ($3 \geq k$), entsteht kein Unterlauf und der Baum ist wieder vollkommen gültig.

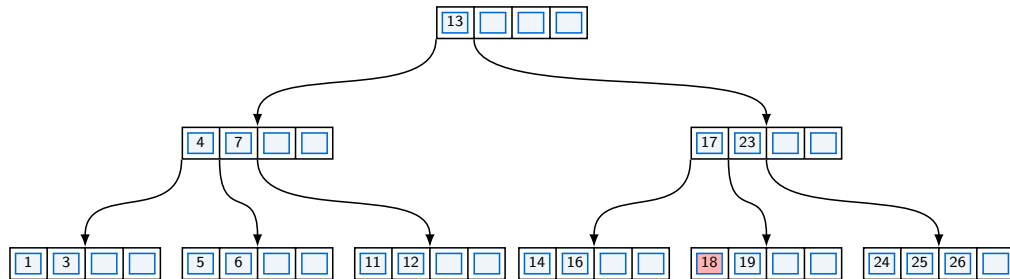


B-Baum: Beispiel Löschen

Fall 3: Löschen mit Unterlauf (Ausleihen vom Nachbarn)

Löschen: 18

Die 18 befindet sich in einem Blatt. Wenn wir sie löschen, hat der Knoten nur noch 1 Element (die 19). Da $1 < k$ (Minimalbelegung $k = 2$), entsteht ein **Unterlauf**.

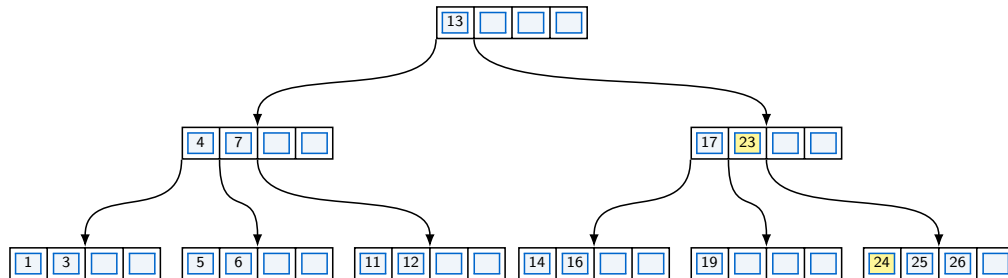


B-Baum: Beispiel Löschen

Fall 3: Löschen mit Unterlauf (Ausleihen vom Nachbarn)

Schritt 1: Ausgleich durch Rotation

Wir prüfen die Nachbarn: Der rechte Nachbar hat 3 Elemente ($> k$) und kann aushelfen. **Rotation:** Das Trenn-Element im Vaterknoten (23) wandert herab, das kleinste Element des rechten Nachbarn (24) steigt auf.

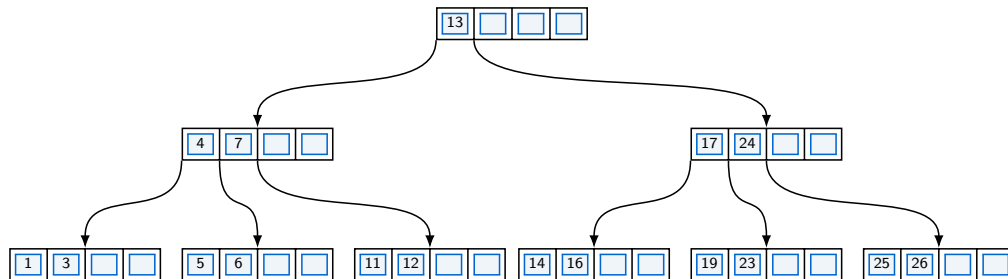


B-Baum: Beispiel Löschen

Fall 3: Löschen mit Unterlauf (Ausleihen vom Nachbarn)

Schritt 2: Ergebnis nach der Rotation

Der Unterlauf ist behoben. Alle Knoten (auch der Nachbarknoten, der nun 2 Elemente besitzt) erfüllen wieder die Bedingung für die Minimalbelegung ($k = 2$).

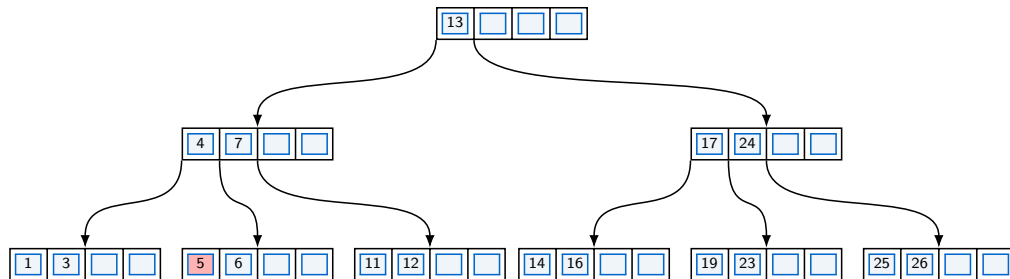


B-Baum: Beispiel Löschen

Fall 4: Löschen mit Verschmelzen (Merge)

Löschen: 5

Nach dem Löschen der 5 bleibt nur die 6 (Unterlauf). Beide Nachbarn haben genau $k = 2$ Elemente und können nichts abgeben. Es muss **verschmolzen** werden.



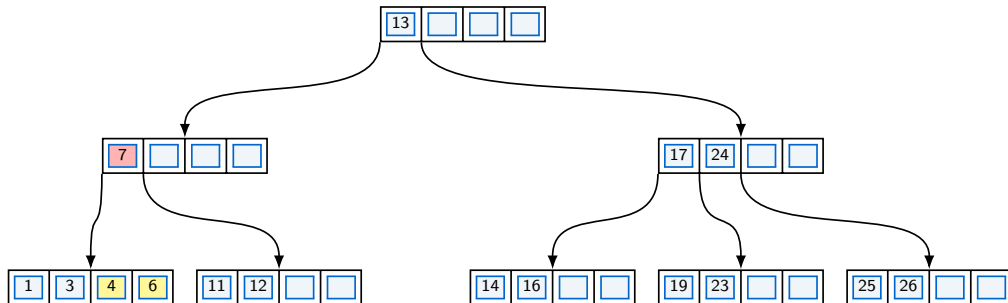
B-Baum: Beispiel Löschen

Fall 4: Löschen mit Verschmelzen (Merge)

Schritt 1: Verschmelzen im Blatt

Die verbleibende 6 verschmilzt mit dem Nachbarn [1, 3] und dem Trennelement 4 aus dem Vaterknoten.

Achtung: Der Vaterknoten hat nun einen propagierten Unterlauf (nur noch die 7)!

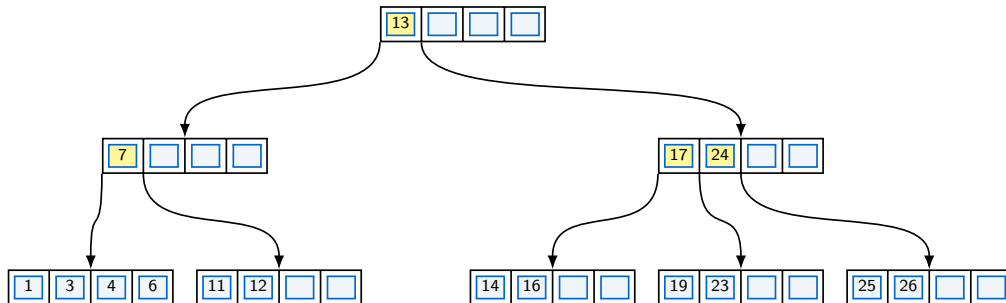


B-Baum: Beispiel Löschen

Fall 4: Löschen mit Verschmelzen (Merge)

Schritt 2: Kaskadierender Unterlauf im inneren Knoten

Der innere Knoten [7] kann nicht vom Nachbarn [17, 24] leihen. Also verschmelzen wir [7] mit dem Nachbarn [17, 24] und dem Trennelement 13 (der Wurzel).

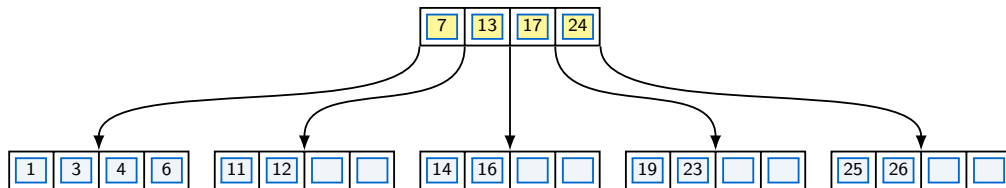


B-Baum: Beispiel Löschen

Fall 4: Löschen mit Verschmelzen (Merge)

Schritt 3: Ergebnis nach Verkleinerung der Baumhöhe

Die alte Wurzel wurde aufgelöst. Der neue Wurzelknoten fasst alle verbliebenen inneren Elemente zusammen. Der Baum ist nun flacher und erfüllt wieder alle Eigenschaften!



Maximum ($N_{\max}$) und Minimum ($N_{\min}$) der Schlüsselanzahl im B-Baum

Berechnung: $N_{\max}$ – Maximal gefüllter Baum

Annahme: Jeder Knoten (auch die Wurzel) hat M Schlüssel; jeder innere Knoten hat $M + 1$ Kinder.

Anzahl Knoten pro Ebene:

- ▶ Ebene 1 (Wurzel): 1 Knoten
- ▶ Ebene 2: $(M + 1)$ Knoten
- ▶ Ebene i : $(M + 1)^{i-1}$ Knoten

Einsetzen in Summenformel der Geometrischen Reihe:

$$\sum_{k=0}^n q^k = \frac{1 - q^{n+1}}{1 - q}$$

Gesamtzahl der Knoten:

$$Knoten_{\max} = \frac{(M + 1)^h - 1}{M}$$

Maximale Schlüsselanzahl:

$$\begin{aligned} N_{\max} &= Knoten_{\max} \times M \\ &= \left(\frac{(M + 1)^h - 1}{M} \right) \times M \\ \Rightarrow N_{\max} &= (M + 1)^h - 1 \end{aligned}$$

Berechnung: $N_{\min}$ – Minimal gefüllter Baum

Annahme: Jeder Knoten (außer Wurzel) ist minimal gefüllt.

- ▶ Wurzel (Ebene 1): Min. 1 Schlüssel $\rightarrow$ mind. 2 Kinder
- ▶ Andere Knoten: $m = \lfloor M/2 \rfloor$; mind. $m + 1$ Kinder

Anzahl Knoten pro Ebene:

- ▶ Ebene 1 (Wurzel): 1 Knoten
- ▶ Ebene 2: 2 Knoten
- ▶ Ebene 3: $2 \times (m + 1)$ Knoten
- ▶ Ebene i für $(i \geq 2)$: $2 \times (m + 1)^{i-2}$ Knoten

Minimale Schlüsselanzahl:

$$\begin{aligned} N_{\min} &= 1 + \sum_{i=2}^h \left(2 \times (m + 1)^{i-2} \right) \times m \\ &= 1 + 2m \sum_{j=0}^{h-2} (m + 1)^j \\ &= 1 + 2 \left((m + 1)^{h-1} - 1 \right) \\ \Rightarrow N_{\min} &= 2(m + 1)^{h-1} - 1 \end{aligned}$$

Höhenabschätzung – Auflösen nach h aus $N_{\max}$ und $N_{\min}$

Abschätzung der minimalen Höhe (bei gegebenem N und maximaler Anzahl Schlüssel pro Knoten):

- ▶ $N_{\max} = (M + 1)^h - 1$
- ▶ Umformen nach h und logarithmieren zur Basis $(M + 1)$: $h = \log_{M+1}(N + 1)$
- ▶ $\Rightarrow h_{\min} = \lceil \log_{M+1}(N + 1) \rceil$

Abschätzung der maximalen Höhe (bei gegebenem N und minimaler Anzahl Schlüssel pro Knoten):

- ▶ $N_{\min} = 2(m + 1)^{h-1} - 1$
- ▶ Umformen nach h und logarithmieren zur Basis $(m + 1)$
- ▶ $\Rightarrow h_{\max} = \left\lfloor 1 + \log_{m+1}\left(\frac{N+1}{2}\right) \right\rfloor$

Damit die vollständige Höhenabschätzung:

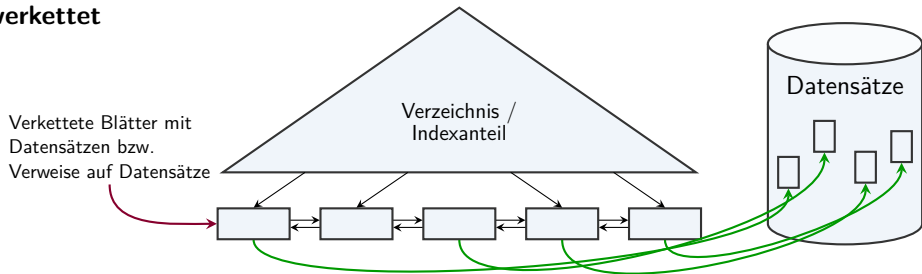
$$\lceil \log_{M+1}(N + 1) \rceil \leq h \leq \left\lfloor 1 + \log_{m+1}\left(\frac{N + 1}{2}\right) \right\rfloor$$

Betrachtung der Schranken: Die Höhe eines B-Baumes ist durch den Logarithmus zur Basis der maximalen bzw. minimalen Anzahl Nachfolger eines Knotens beschränkt.

Wichtige Variante B⁺-Baum

Ein B⁺-Baum ist eine B-Baum-Variante mit zwei Knotentypen

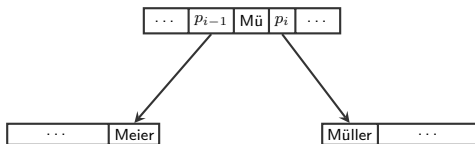
- ▶ **Blätter enthalten Schlüssel mit Datensätzen** oder Schlüssel mit Verweisen auf Datensätze
- ▶ Innere Knoten enthalten **keine Datensätze**, nur Trennschlüssel
- ▶ Als Trennschlüssel (Separatoren, Wegweiser) nutzt man z.B. die **Schlüssel selbst oder geeignete Präfixe** (bei Strings)
- ▶ Für ein effizientes Durchlaufen großer Bereiche der Daten sind die **Blätter miteinander verkettet**



Vergleich B⁺-Baum und B-Baum

- ▶ In inneren Knoten stehen im B⁺-Baum **nur Schlüssel ohne Datensätze**. Dadurch passen auf eine Seite deutlich mehr Einträge.
- ▶ Trennschlüssel dienen nur als Wegweiser und können oft als **kurze Präfixe** gespeichert werden.

Beispiel: Das Präfix „Mü“ reicht als Trennschlüssel zwischen „Meier“ und „Müller“ aus.



- ▶ Dadurch ist der B⁺-Baum in der Regel **breiter** (höherer Fanout) und **weniger hoch** (weniger Ebenen) als ein klassischer B-Baum.
- ▶ In der Praxis (DBMS) werden deshalb **fast ausschließlich Varianten von B⁺-Bäumen** eingesetzt.

1D-Indexstrukturen – Fazit zu B/B⁺-Bäumen

Anforderungen

- ▶ Effizient für große Datenmengen auf Platte/SSD.
- ▶ Minimierung der I/O-Kosten.
- ▶ Schnelle Punkt- und Bereichsanfragen.
- ▶ Effiziente Updates.

Konsequenzen der Speicherhierarchie

1. **Blockorientierung:** Indexknoten sollten auf Blöcke bzw. Seiten passen und viele Einträge enthalten. Ergebnis: hoher Verzweigungsgrad und wenige I/Os.
2. **Balanciertheit:** Alle Blätter liegen auf derselben Tiefe. Ergebnis: kurze, vorhersagbare Suchpfade und logarithmische Höhe in N .

Kernaussage und Übergang

B/B⁺-Bäume minimieren den **I/O-Flaschenhals**: Ihr hoher Verzweigungsgrad führt zu sehr flachen, blockorientierten Suchbäumen.

Diese Stärke gilt für ein **geordnetes Suchattribut**. Im nächsten Schritt betrachten wir Anfragen mit **mehreren Attributen gleichzeitig**.

5. Relationale Anfragebearbeitung

Kapitelüberblick: Nach den eindimensionalen Indizes wechseln wir nun zu Anfragen, die **mehrere Attribute gleichzeitig** einschränken.

1. Anfragebearbeitung und Optimierung

- ▶ Motivation
- ▶ Query Processing und Optimierungstypen
- ▶ Grundlagen: Regelbasierte Anfrageoptimierung
- ▶ Grundlagen: Kostenbasierte Anfrageoptimierung
- ▶ Join-Reihenfolgen

2. Algorithmen für Basisoperationen

- ▶ Selektion
- ▶ Projektion
- ▶ Join

3. Indexstrukturen

- ▶ Eindimensionale Daten
- ▶ **Indexstrukturen für mehrdimensionale Daten**

Jetzt im Fokus: Anfragen mit mehreren Filterattributen und Indexstrukturen, die diese Bedingungen gemeinsam berücksichtigen.

Mehrdimensionale Indexstrukturen (1) – Problemstellung

Bisher: B⁺-Bäume für Anfragen auf *einem* Attribut (z.B. WHERE Alter > 30)

Problem: Viele reale Anfragen filtern nach *mehreren* Attributen gleichzeitig!

Beispiele:

- ▶ „Finde alle Studenten mit Nebenfach BWL und im 7–9 Semester”
WHERE Nebenfach = 'BWL' AND Semester >= 7 AND Semester <= 9
- ▶ „Finde alle Sensormessungen im Rechteck (2.5,3.8)–(4.1,6.2)”
WHERE 2.5 <= x AND x <= 4.1 AND 3.8 <= y AND y <= 6.2

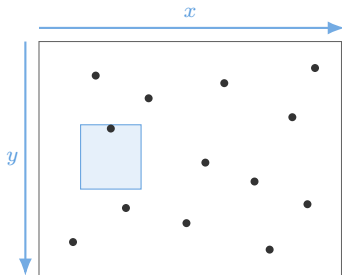
Herausforderung:

Wie beantwortet man solche Anfragen effizient?

Wie vermeidet man, Kandidaten erst eindimensional zu holen und *danach* weitere Bedingungen separat zu prüfen?

⇒ Bedarf an Indexstrukturen, die mehrere Dimensionen (Attribute) gleichzeitig berücksichtigen

Matrikelnummer	Nebenfach	Semester
171283	BWL	9
184238	Medizin	7
191373	Elektrotechnik	5
...	...	...



Ansatz 1: Mehrere 1D-Indexe – Die invertierte Liste

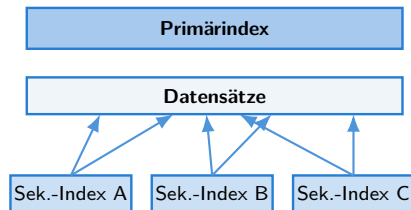
Intuition: Gute 1D-Indexe lassen sich pro Dimension getrennt nutzen.

Ziel: Unterstützung von Anfragen über mehrere Attribute.

- ▶ Multiattributsuche mit **invertierten Listen**
- ▶ Für jedes Attribut gibt es einen (Sekundär-)Index.
- ▶ Jede Liste wird unabhängig von den anderen durchsucht.
- ▶ Die Treffermengen werden per **Schnittmengenbildung** kombiniert.

Indexgefüge

- ▶ **Primärindex:** Index über den Primärschlüssel
- ▶ **Sekundärindex:** Index über ein Nichtschlüsselattribut und damit nur eine Sammlung von Verweisen



Invertierte Listen (2)

Prinzip der invertierten Listen:

- ▶ Für anfragerrelevante Attribute werden Sekundärindexe (*invertierte Listen*) angelegt.
- ▶ Jeder Sekundärindex liefert Verweise auf passende Antwortkandidaten.

Vorgehen bei einer Anfrage über $A_1, \dots, A_m$:

1. Für jedes spezifizierte Attribut A_i die passende Liste im zugehörigen Sekundärindex nachschlagen.
2. Dadurch entstehen m Listen mit Verweisen auf Antwortkandidaten in der Datei.
3. Bei AND-Bedingungen die **Schnittmenge** bilden.
4. Bei OR-Bedingungen die **Vereinigung** bilden.

Invertierte Listen – Beispiel

Primärdaten

Adr.	Name	Stadt	Alter
1	Adams	Athen	30
2	Blake	Paris	30
3	Clark	London	50
4	Hart	Chicago	40
5	James	Athen	30
6	Jones	Paris	40
7	Parker	New York	40
8	Smith	London	30

```
SELECT * FROM R
WHERE Stadt = 'Athen'
AND Alter = 30
```

Sekundärindex Stadt

Athen $\rightarrow \{1, 5\}$
Chicago $\rightarrow \{4\}$
London $\rightarrow \{3, 8\}$
New York $\rightarrow \{7\}$
Paris $\rightarrow \{2, 6\}$

Sekundärindex Alter

30 $\rightarrow \{1, 2, 5, 8\}$
40 $\rightarrow \{4, 6, 7\}$
50 $\rightarrow \{3\}$

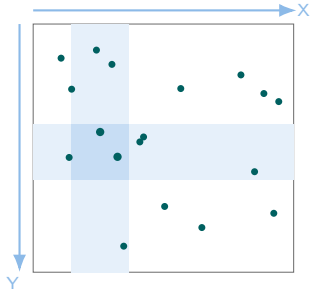
Ergebnis:

$$\{1, 5\} \cap \{1, 2, 5, 8\} = \{1, 5\}$$

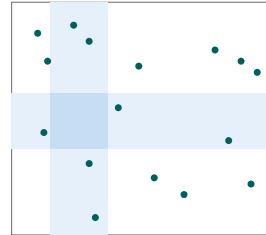
Invertierte Listen: Punktdaten

Bei 2-dimensionalen Punktdaten:

- ▶ Speicherung der X- und Y-Werte in jeweils einem Sekundärindex
- ▶ bei Suchen von Punkten in Bereichen (Rechtecken):
 - ▶ Bilde Punktmenge, deren X-Koordinaten im Anfragebereich liegen
 - ▶ Bilde Punktmenge, deren Y-Koordinaten im Anfragebereich liegen
 - ▶ Bilde die Schnittmenge beider Mengen



Antwort: zwei Punkte



„worst-case“

Invertierte Listen – Eigenschaften

Beobachtungen

- ▶ Die Antwortzeit ist **nicht proportional** zur Zahl der Treffer.
- ▶ Je mehr Attribute spezifiziert sind, desto mehr Listen müssen kombiniert werden.
- ▶ Updates sind teuer, weil viele Sekundärindexte gepflegt werden müssen.

Ursachen und Folgen

- ▶ Die Attributwerte eines Tupels sind **nicht in einer gemeinsamen Struktur** verbunden.
- ▶ Sekundärindexte verändern die physische Speicherung der Datensätze nicht.
- ▶ Eine Ordnung über dem Sekundärschlüssel wird **nicht erhalten**.

Fazit: Invertierte Listen sind brauchbar, wenn die Trefferlisten sehr klein sind. Für größere Kandidatenmengen zeigen sie ein **schlechtes Leistungsverhalten**.

Übergang: Für Attribute mit wenigen verschiedenen Werten gibt es einen effizienteren Spezialfall: den **Bitmap-Index**.

Bitmap-Indizes: Motivation & Grundidee

Das Problem: „Low Cardinality“

- ▶ B-Bäume sind ineffizient bei Attributen mit **sehr wenigen Ausprägungen** (z. B. Geschlecht, Status).
- ▶ Die Prädikate sind oft wenig selektiv → viele Datensätze müssten geladen werden.

Die Lösung: Bitvektoren

Für jeden möglichen Attributwert wird eine eigene **Bitmap** angelegt.

1 Bit pro Datensatz (Tupel):

1 = Wert ist vorhanden

0 = Wert ist nicht vorhanden

Ausgangsdaten (Beispiel)

Tupel	Name	Geschlecht	Einkommen
0	Klaus	m	L1
1	Diana	w	L2
2	Maria	w	L1
3	Peter	m	L4
4	Johannes	d	L3

← Wir wollen schnelle Suchen über die Attribute

„Geschlecht“ und „Einkommen“ durchführen, ohne diese Tabelle laden zu müssen.

Aufbau des Bitmap-Index

Jede Zeile repräsentiert eine Bitmap für einen Attributwert.
Die Position im Vektor entspricht der relativen **Tupel-ID** aus der Basistabelle.

Bitmaps für Geschlecht

m:

1	0	0	1	0
---	---	---	---	---

w:

0	1	1	0	0
---	---	---	---	---

d:

0	0	0	0	1
---	---	---	---	---

Bitmaps für Einkommensstufe

L1:

1	0	1	0	0
---	---	---	---	---

L2:

0	1	0	0	0
---	---	---	---	---

L3:

0	0	0	0	1
---	---	---	---	---

L4:

0	0	0	1	0
---	---	---	---	---

Anfrageverarbeitung (Bit-Operationen)

Logische Operationen

UND (Schnittmenge)

$100110 \text{ AND } 110011 = 100010$

ODER (Vereinigung)

$100110 \text{ OR } 110011 = 110111$

NICHT (Komplement)

$\text{NOT } 100110 = 011001$

Konkretes Beispiel

Frage: Finde alle Männer mit Einkommenstufe L1.

Bitmap(m) = 10010

Bitmap(L1) = 10100

Ergebnis = 10000

 Nur Tupel 0 erfüllt beide Bedingungen.

Der enorme Vorteil:

Das „Zählen“ (COUNT) von Ergebnissen erfolgt durch das superschnelle Zählen der Einsen im Vektor (*Popcount*) auf der CPU – ohne echte Datensätze laden zu müssen!

1 Bit

Speicherbedarf pro Tupel und Attributwert. Für ein Attribut mit d verschiedenen Werten entstehen also d Bitmaps, also unkomprimiert etwa d Bits pro Tupel.

Warum Bitmap-Indizes dominieren

- ▶ **Minimaler Overhead:**
Bei Datensätzen mit 100 Byte kostet eine Bitmap nur ca. $\frac{1}{800}$ der Tabellengröße.
- ▶ **Extreme Kompression:**
Bitmaps mit langen Ketten von Nullen (sehr seltene Werte) lassen sich massiv komprimieren (z. B. *Run-Length Encoding*).
- ▶ **Hardware-Nähe:**
Komprimierte Index-Vektoren passen häufig komplett in den extrem schnellen **CPU-Cache**.

Umgang mit Deletes: Die Existenz-Bitmap

Das Problem bei Löschungen

- ▶ Wenn ein Datensatz physisch gelöscht wird, rücken nachfolgende Tupel auf.
- ▶ **Folge:** Alle relativen Positionen sind kaputt. Sämtliche Bitmaps müssten neu geschrieben werden!

Lösung: Existenz-Bitmap (B_{exist})

Gelöschte Tupel bleiben physisch erhalten, werden aber als „gelöscht“ markiert.

1 = Datensatz ist gültig 0 = **gelöscht**

Jede Suchanfrage wird am Ende bitweise mit der Existenz-Bitmap geschnitten:

AND B_{exist}

Beispiel: Tupel 2 wird gelöscht

ID	Name	Sex
0	Klaus	m
1	Diana	w
2	Maria	w
3	Peter	m
4	Johannes	d

Existenz (B_{exist}): 11**0**11

Anfrage: Finde alle Frauen (w)

Bitmap(w) 01100

AND B_{exist} 11**0**11

Ergebnis 01**0**00

Die NULL-Problematik

Die NOT-Logik wird komplex. Ein Tupel darf bei $\text{NOT}(A = v)$ nicht im Ergebnis auftauchen, wenn Attribut A NULL ist.

$$\neg(A = v) = (\neg B_{A=v}) \wedge B_{\text{exist}} \wedge (\neg B_{A=\text{NULL}})$$

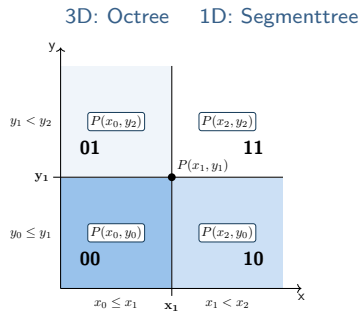
Dies erfordert das Vorhalten von **separaten Bit-maps für NULL-Werte**, um eine exakte Filterung nach SQL-Semantik zu garantieren.

OLAP vs. OLTP

- ▶ **Data Warehouses (OLAP):**
Perfekt geeignet! Es gibt massenhaft Lesezugriffe und komplexe Filterungen. Updates erfolgen meist nur im Batch in der Nacht.
- ▶ **Transaktionale Systeme (OLTP):**
Ungeeignet! Da bei jedem neuen Datensatz (INSERT) *alle* Bitvektoren wachsen müssten, würden permanente Schreibvorgänge das System massiv blockieren.

Punkt-Quadtree (2D)

- ▶ Nicht-balancierte Datenstruktur für mehrdimensionale Punkte (hier 2D)
- ▶ **Ansatz:** Raumteilung an jedem Knoten für x- und y-Koordinate:
 $x (\leq, >)$ und $y (\leq, >)$ $\rightarrow$ 4 Kindknoten („Quad“-Tree)
- ▶ **Binäre Codierung** (Kind-Reihenfolge: **11, 10, 01, 00**):
 - ▶ **1. Bit (x-Achse):** $0 \hat{=}\leq$ | $1 \hat{=}>$
 - ▶ **2. Bit (y-Achse):** $0 \hat{=}\leq$ | $1 \hat{=}>$
- ▶ **Aufbau des Baums:**
 - ▶ Erster Punkt bildet die Wurzel
 - ▶ Weitere Punkte: Quadrant bestimmen $\rightarrow$ absteigen $\rightarrow$ falls leer, als neues Kind einfügen
- ▶ *Einsatz:* Meist Hauptspeicherstruktur; in DB-Systemen werden für Sekundärspeicher eher *Bucket-Quadrees* (Daten in den Blättern) genutzt.

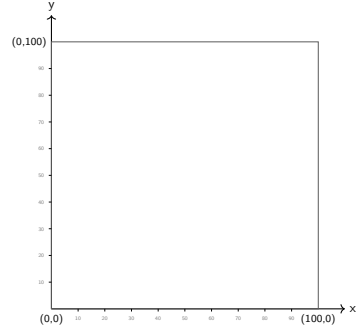


Beispiel: Schrittweiser Aufbau (Baum & Raum)

Einfügen:

Baumstruktur:

2D-Datenraum:



Beispiel: Schrittweiser Aufbau (Baum & Raum)

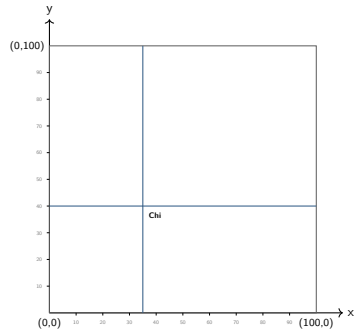
Einfügen:

- Chicago
(35,40)

Baumstruktur:

35|40
Chicago

2D-Datenraum:

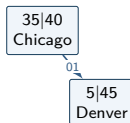


Beispiel: Schrittweiser Aufbau (Baum & Raum)

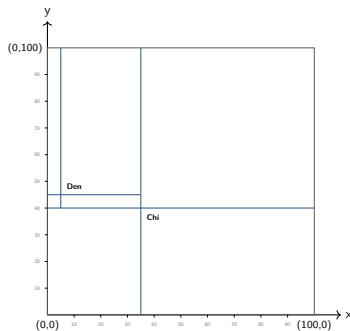
Einfügen:

- ▶ Chicago
(35,40)
- ▶ Denver (5,45)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: $0(\leq), 1(>)$

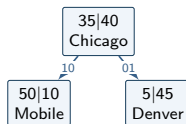
y: $0(\leq), 1(>)$

Beispiel: Schrittweiser Aufbau (Baum & Raum)

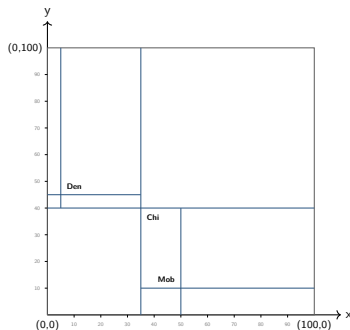
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: $0(\leq), 1(>)$

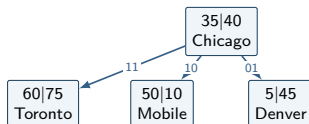
y: $0(\leq), 1(>)$

Beispiel: Schrittweiser Aufbau (Baum & Raum)

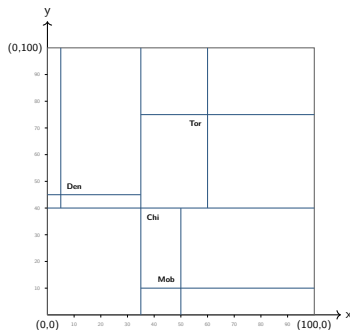
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)
- ▶ Toronto (60,75)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: 0($\leq$), 1($>$)

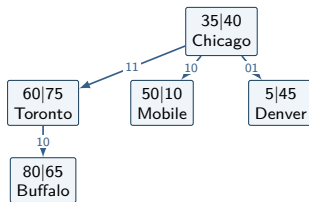
y: 0($\leq$), 1($>$)

Beispiel: Schrittweiser Aufbau (Baum & Raum)

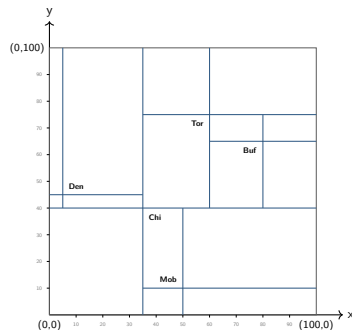
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)
- ▶ Toronto (60,75)
- ▶ Buffalo (80,65)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: $0(\leq), 1(>)$

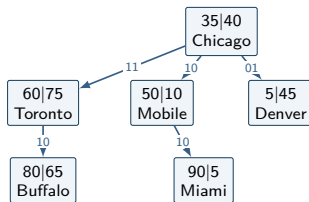
y: $0(\leq), 1(>)$

Beispiel: Schrittweiser Aufbau (Baum & Raum)

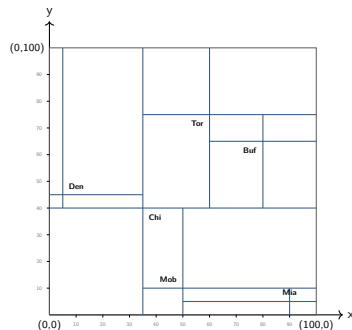
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)
- ▶ Toronto (60,75)
- ▶ Buffalo (80,65)
- ▶ Miami (90,5)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: $0(\leq), 1(>)$

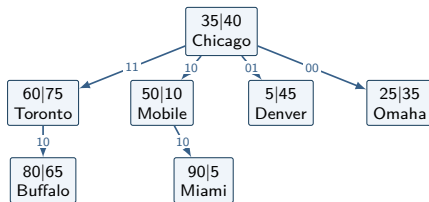
y: $0(\leq), 1(>)$

Beispiel: Schrittweiser Aufbau (Baum & Raum)

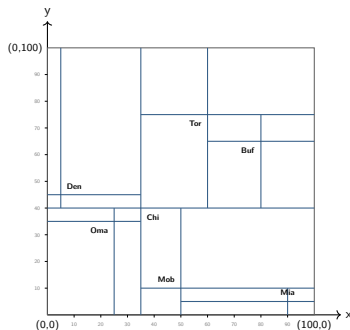
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)
- ▶ Toronto (60,75)
- ▶ Buffalo (80,65)
- ▶ Miami (90,5)
- ▶ Omaha (25,35)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: 0($\leq$), 1($>$)

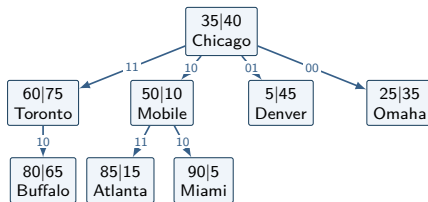
y: 0($\leq$), 1($>$)

Beispiel: Schrittweiser Aufbau (Baum & Raum)

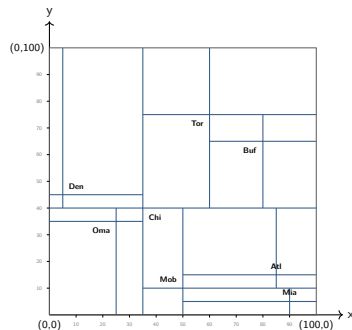
Einfügen:

- ▶ Chicago (35,40)
- ▶ Denver (5,45)
- ▶ Mobile (50,10)
- ▶ Toronto (60,75)
- ▶ Buffalo (80,65)
- ▶ Miami (90,5)
- ▶ Omaha (25,35)
- ▶ Atlanta (85,15)

Baumstruktur:



2D-Datenraum:



Erinnerung Code:

x: $0(\leq), 1(>)$

y: $0(\leq), 1(>)$

Ein alltägliches Problem

‘Finde alle Pizzerien in 2 km Umkreis um den Aachener Dom.’

- ▶ **Naiver Ansatz:** 2D-Koordinaten (x, y) in einen 1D-Index (**B⁺-Baum**) zwingen (z. B. verkettet).
- ▶ **Das Problem:** Räumliche Nähe geht bei der 1D-Abbildung verloren. B-Bäume scheitern an mehrdimensionalen Bereichsanfragen.

Ziel: Wir brauchen eine Datenstruktur, die *räumliche Ausdehnung* nativ abbildet.

Konzept: R-Baum (Rectangle-Tree)

Grundlegende Idee:

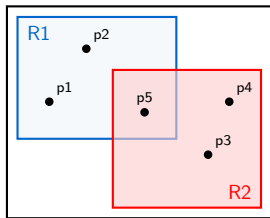
- ▶ Verallgemeinerung der B^+ -Baum-Idee auf den mehrdimensionalen Raum.
- ▶ Basiert auf der Technik **überlappender Seitenregionen**.
- ▶ Approximation der Objekte durch **minimale umgebende Rechtecke** (Abk. MUR, engl. MBR “Minimum Bounding Rectangle”).

Aufbau einer Seite (Knoten)

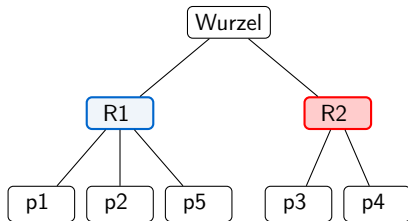
- ▶ **Verzeichnis-Seiten (innere Knoten):**
Einträge bestehen aus MBRs und Verweisen auf andere Kind-Seiten.
- ▶ **Datenseiten (Blattknoten):**
Einträge bestehen aus MBRs und Verweisen auf die exakte Objekt-Repräsentation (oder direkt Datenpunkte).

Visuelle Verknüpfung: Datenraum vs. Indexbaum

Datenraum (Partitionierung)



Resultierender R-Baum



- ▶ Parameter: M (max. Anzahl Einträge), $m \leq M/2$ (Mindestbelegung).
- ▶ Baum ist **höhenbalanciert** (alle Blätter auf derselben Höhe).

Der Paradigmenwechsel: "Die Überlappung"

B-Baum (1D)

- ▶ Partitioniert den Schlüsselraum **disjunkt** (strikt getrennt).
- ▶ Suchpfad ist **immer eindeutig**.

R-Baum (n-D)

- ▶ Regionen (MBRs) auf gleicher Ebene können sich **überlappen**.
- ▶ Suchpfad ist **nicht eindeutig**!

 **Frage:** Warum müssen wir Überlappungen zulassen?

Objekte im echten Raum sind selten uniform verteilt. Um die minimal umgebenden Rechtecke (MBRs) klein zu halten und keinen "leeren Raum" mitzuindizieren, muss der R-Baum Überlappungen tolerieren.



Achtung: Weitreichende Konsequenz: Im Worst-Case degeneriert die Suche zu einer Tiefensuche über mehrere Teilbäume.

1. Punktanfrage (Point Query) für Punkt p

1. Starte bei der Wurzel.
2. Untersuche jeden Kindknoten: **Enthält** sein MBR den Punkt p ?
3. Falls ja $\rightarrow$ steige in alle entsprechenden Teilbäume ab.
4. Überprüfe in den Blattknoten, ob ein Rechteck den Punkt p exakt enthält.

2. Rechteckanfrage (Window Query) für Rechteck R

1. Starte bei der Wurzel.
2. Untersuche jeden Kindknoten: **Schneidet** sein MBR das Anfrage-Rechteck R ?
3. Falls ja $\rightarrow$ steige in alle betroffenen Teilbäume ab.
4. Überprüfe in den Blattknoten, ob das Objekt das Rechteck R schneidet.

Baum-Modifikationen: Einfügen & Löschen

Einfügen eines neuen Objekts O :

- ▶ Einfügungen erfolgen stets in den Blattknoten.
- ▶ **Dilemma:** Objekt könnte in mehrere MBRs passen (wegen Überlappung) oder in gar keins.
- ▶ **Heuristik (*Minimal Enlargement*):** Wähle den Teilbaum, dessen MBR um die *geringste Fläche vergrößert* werden muss.
- ▶ Bei Überlauf (Split): Verschiedene Strategien (linear, quadratisch), um Einträge so auf zwei Knoten zu verteilen, dass die Flächenvergrößerung minimal bleibt.

Löschen:

- ▶ Suche das Objekt und entferne es. Passe MBRs auf dem Pfad zur Wurzel an.
- ▶ **Underflow-Behandlung:** Oft pragmatischer Ansatz – statt komplexer Verschmelzungen werden "verwaiste" Knoten gelöscht und ihre Elemente von oben neu in den Baum eingefügt (*Re-Insertion*).

- ▶ **Zweck:** Nativer Index für mehrdimensionale und räumliche Daten.
- ▶ **Prinzip:** Hierarchische Anordnung von **minimal umgebenden Rechtecken** (MBRs).
- ▶ **Trade-off: Überlappungen** sind zwingend nötig (zur Vermeidung von Leerraum), erfordern bei Suchanfragen aber oft das Absteigen in mehrere Teilbäume.

Ausblick: Varianten wie der **R*-Baum** nutzen verbesserte Heuristiken, um Überlappungen schon beim Einfügen proaktiv zu minimieren.

Fazit

- ▶ Es gibt **keinen One-Size-Fits-All-Index**.
- ▶ Die Wahl hängt stark von Daten, Anfragen und Systemanforderungen ab.
- ▶ Moderne Systeme kombinieren oft mehrere Techniken.
- ▶ Forschung geht weiter: approximative Indexe, KI-gestützte Indexselektion und Vektor-Indizes.
- ▶ Wer die Grundprinzipien versteht, kann informierte Architekturentscheidungen treffen.

Ausblick

Nächstes Thema: Zurück zum Schema-Design

Wie müssen gute relationale Schemata aussehen?