

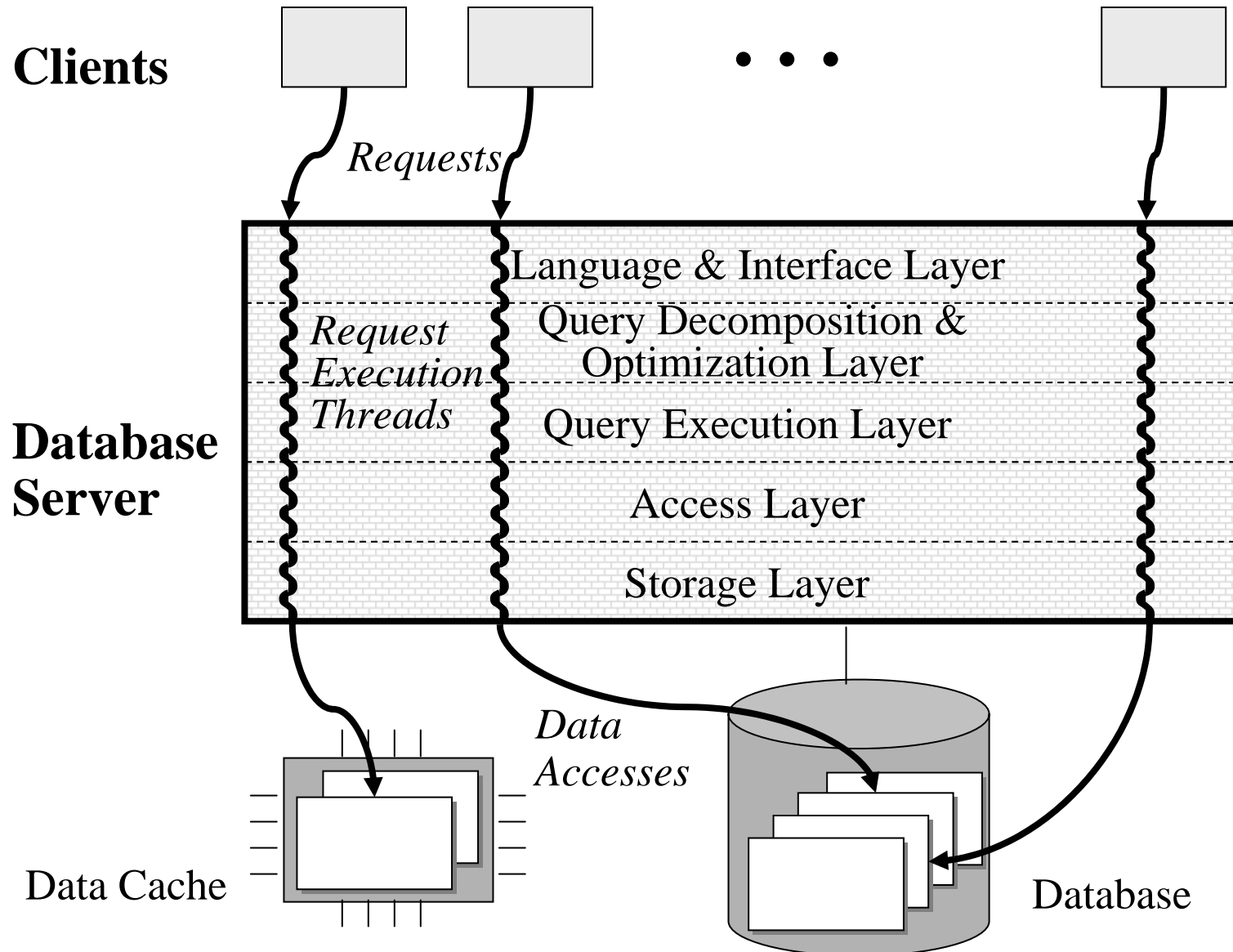
Kapitel 14: Datenspeicherung, Indexstrukturen und Anfrageausführung

14.1 Wie Daten gespeichert werden

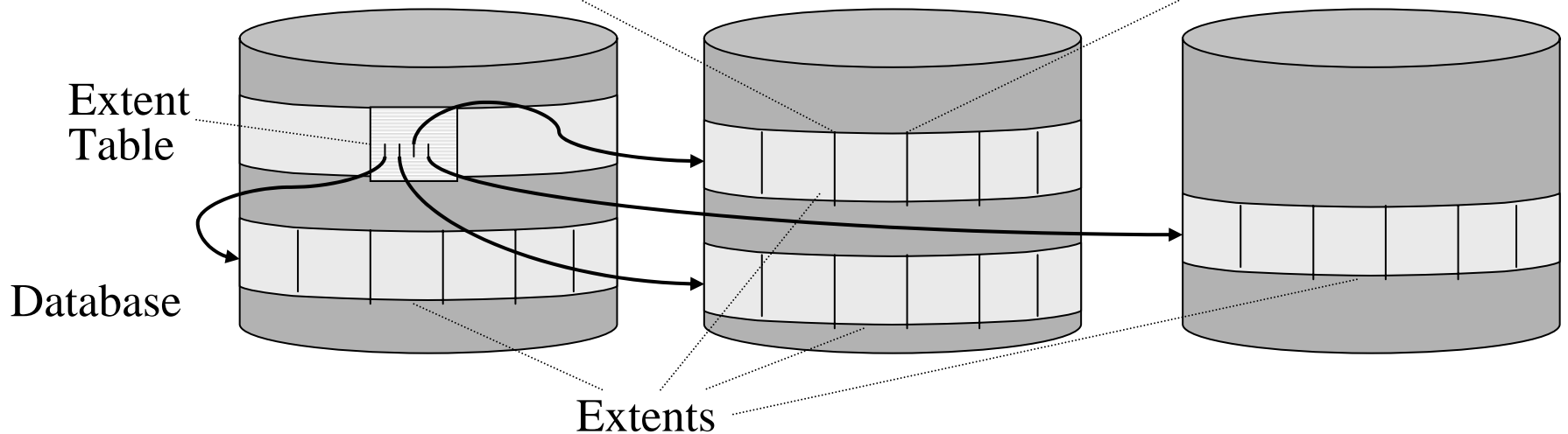
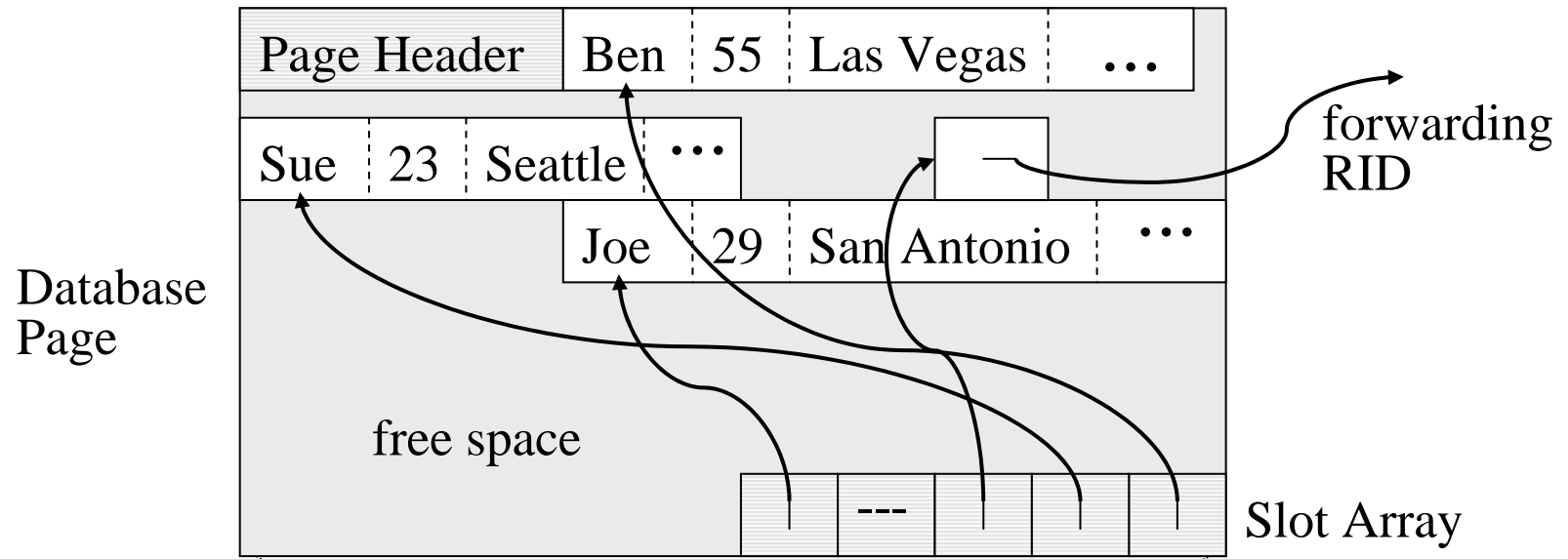
**14.2 Wie auf Daten effizient zugegriffen wird
(Indexstrukturen)**

14.3 Wie Anfragen ausgeführt werden

Interne (idealisierte) Schichtenarchitektur eines DBS



14.1 Wie Daten gespeichert werden

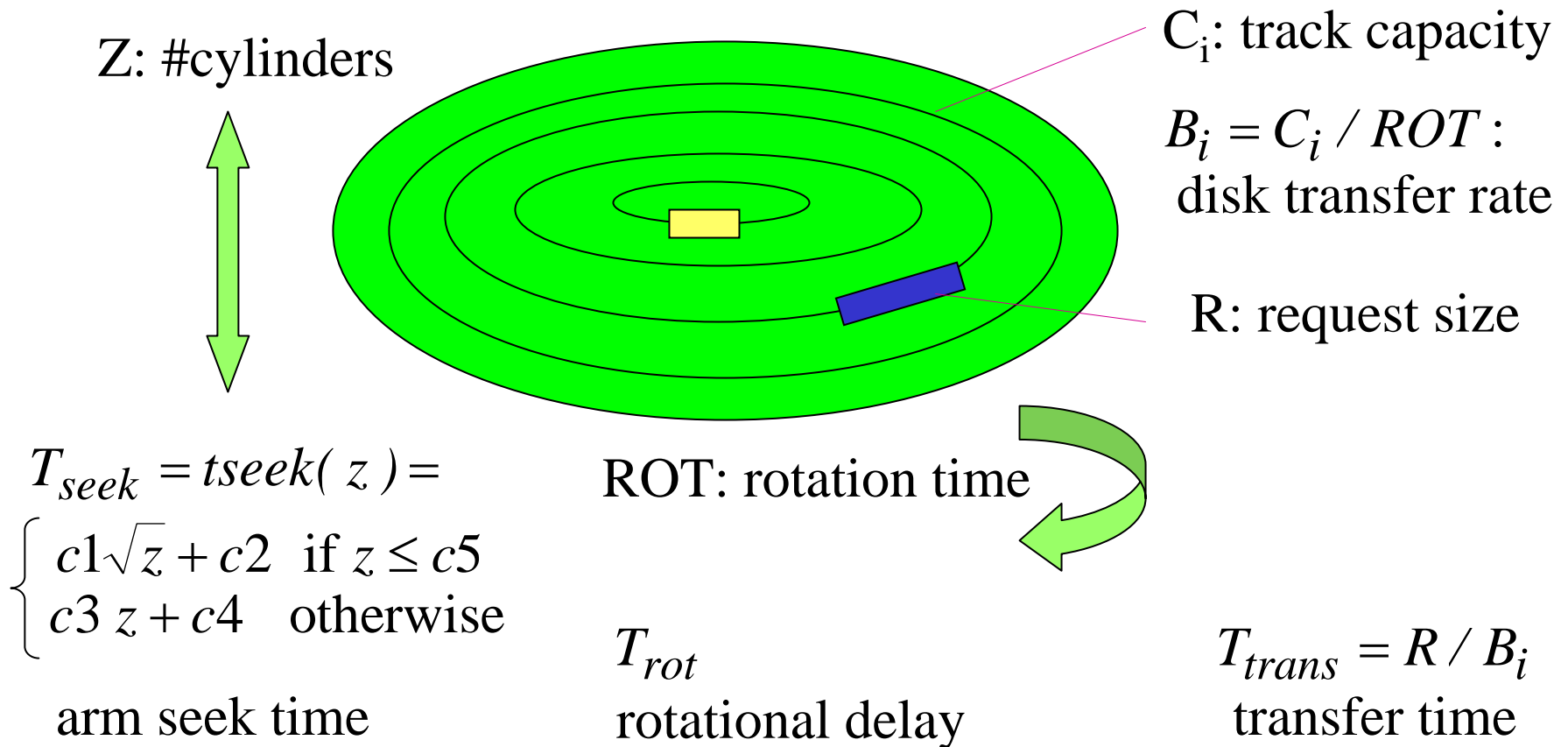


Charakteristika moderner Magnetplatten

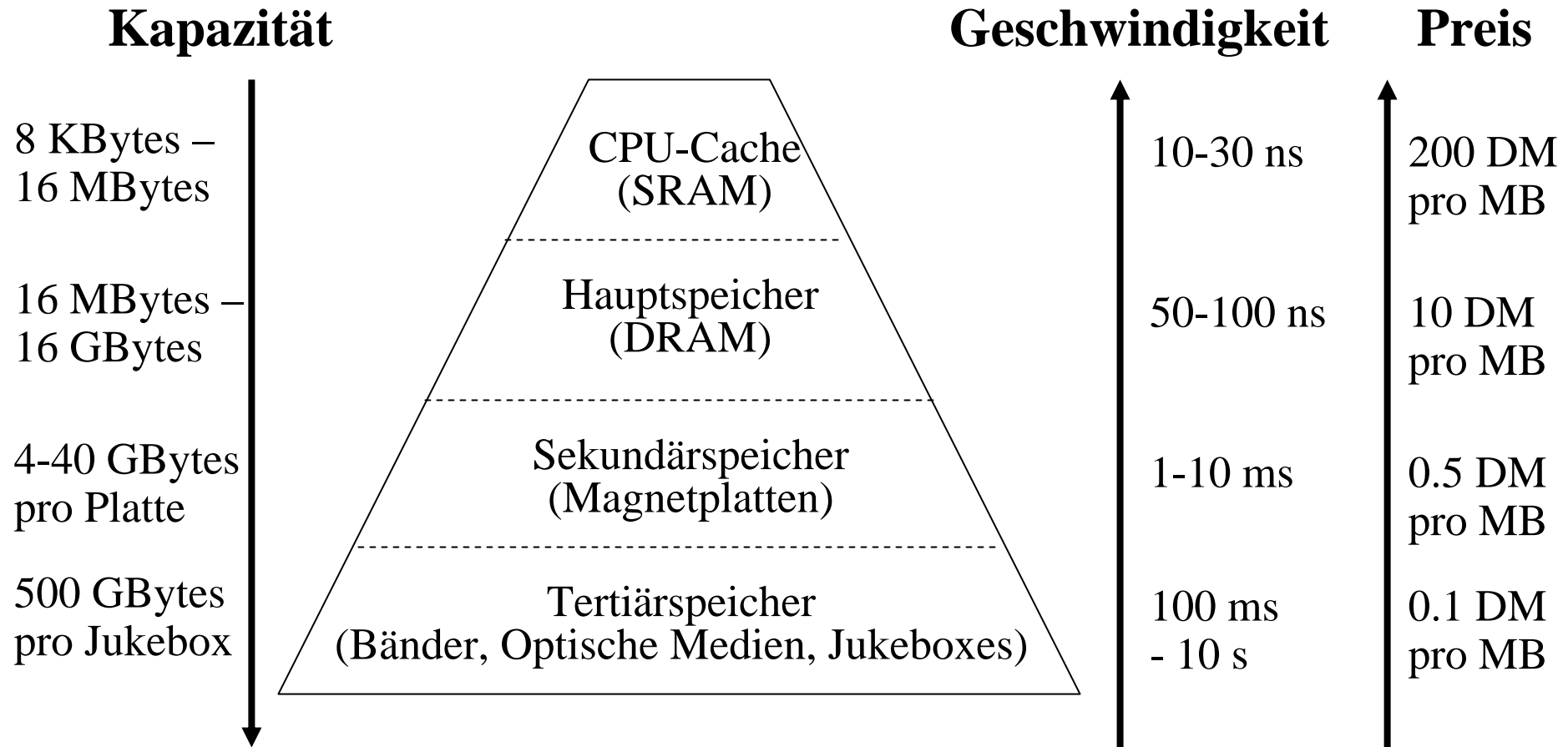
(z.B. Seagate Cheetah 18 mit Ultra-SCSI- oder Fibre-Channel-Schnittstelle)

Durchmesser	3.5 Zoll
Speicherkapazität	18.2 GBytes
Preis	ca. 4000 DM
Größe	ca. 14 x 10 x 4 cm
Gewicht	ca. 1.2 kg
Energieverbrauch	15 Watt
Zuverlässigkeit (MTTF)	800 000 Stunden (> 75 Jahre)
Anzahl Oberflächen	24
Anzahl Zylinder	6962
Spurkapazität	87 bis 128 KBytes
Rotationszeit	6 ms (10000 U/min)
Mittlere Armpositionierungszeit	5.7 ms
Minimale Armpositionierungszeit	0.6 ms
Übertragungsrate	14.5 bis 21.3 MBytes / s
Random I/O auf 1 Block a 4 KB	ca. 9 ms
Sequential I/O auf 30 Blöcke a 4 KB	ca. 12 ms

Leistungsparameter einer Magnetplatte



Charakteristika von Speicherhierarchien



14.2 Wie auf Daten effizient zugegriffen wird (Indexstrukturen)

CREATE [UNIQUE] INDEX index-name
ON table (column, {, column ...})

zur Beschleunigung von:

- *Exact-Match-Selektionen*: $A1=\text{wert1} \wedge A2=\text{wert2} \wedge \dots \wedge Ak=\text{wertk}$
(z.B. City = 'Miami' oder City = 'Paris' \wedge State = 'Texas')
- *Bereichs-Selektionen*: $u1 \leq A1 \leq o1 \wedge \dots \wedge uk \leq Ak \leq ok$
(z.B. $21 \leq \text{Age} \leq 30$ oder $21 \leq \text{Age} \leq 30 \wedge \text{Salary} \geq 100\,000$)
- *Prefix-Match-Selektionen*:
 $u1 \leq A1 \leq o1 \wedge u2 \leq A2 \leq o2 \wedge \dots \wedge uj \leq Aj \leq oj$ (mit $j < k$)
(z.B. $21 \leq \text{Age} \leq 30$ bei einem Index über Age, Salary).

Welche Indexstrukturen für welche Attributkombinationen?

→ Problem des physischen Datenbankentwurfs
für DBA oder „Index Wizard“ !

Binäre Suchbäume (für Hauptspeicher)

Schlüssel (mit den ihnen zugeordneten Daten)
bilden die Knoten eines binären Baums
mit der Invariante:

für jeden Knoten t mit Schlüssel $t.key$ und alle Knoten l im linken Teilbaum von t , $t.left$, und alle Knoten r im rechten Teilbaum von t gilt: $l.key \leq t.key \leq r.key$

Suchen eines Schlüssels k :

Traversieren des Pfades von der Wurzel bis zu k bzw. einem Blatt

Einfügen eines Schlüssels k :

Suchen von k und Anfügen eines neuen Blatts

Löschen eines Schlüssel k :

Ersetzen von k durch das „rechtteste“ Blatt links von k

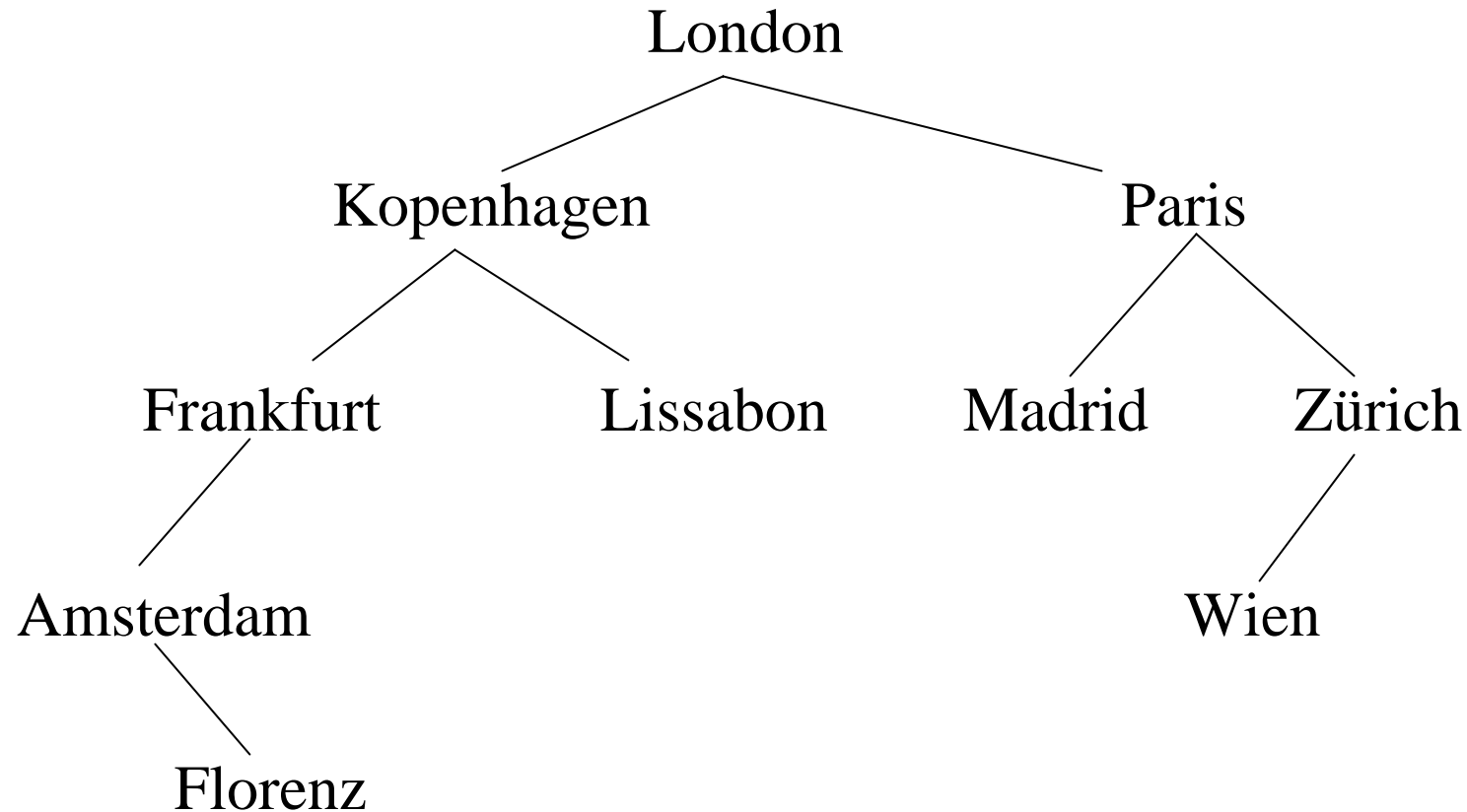
Worst-Case-Suchzeit für n Schlüssel: $O(n)$

bei geeigneten Rebalancierungsalgorithmen

(AVL-Bäume, Rot-Schwarz-Bäume, usw.): $O(\log n)$

Beispiel für einen binären Suchbaum

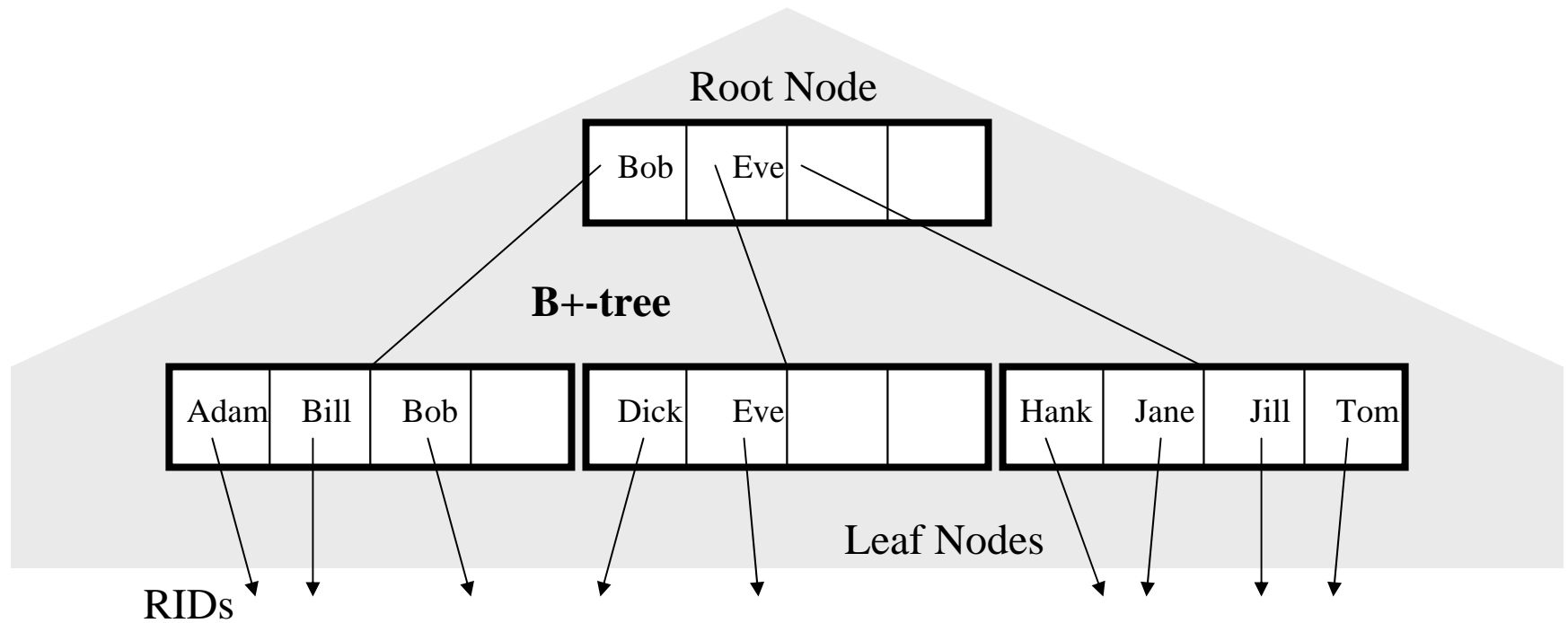
London, Paris, Madrid, Kopenhagen, Lissabon, Zürich, Frankfurt, Wien, Amsterdam, Florenz



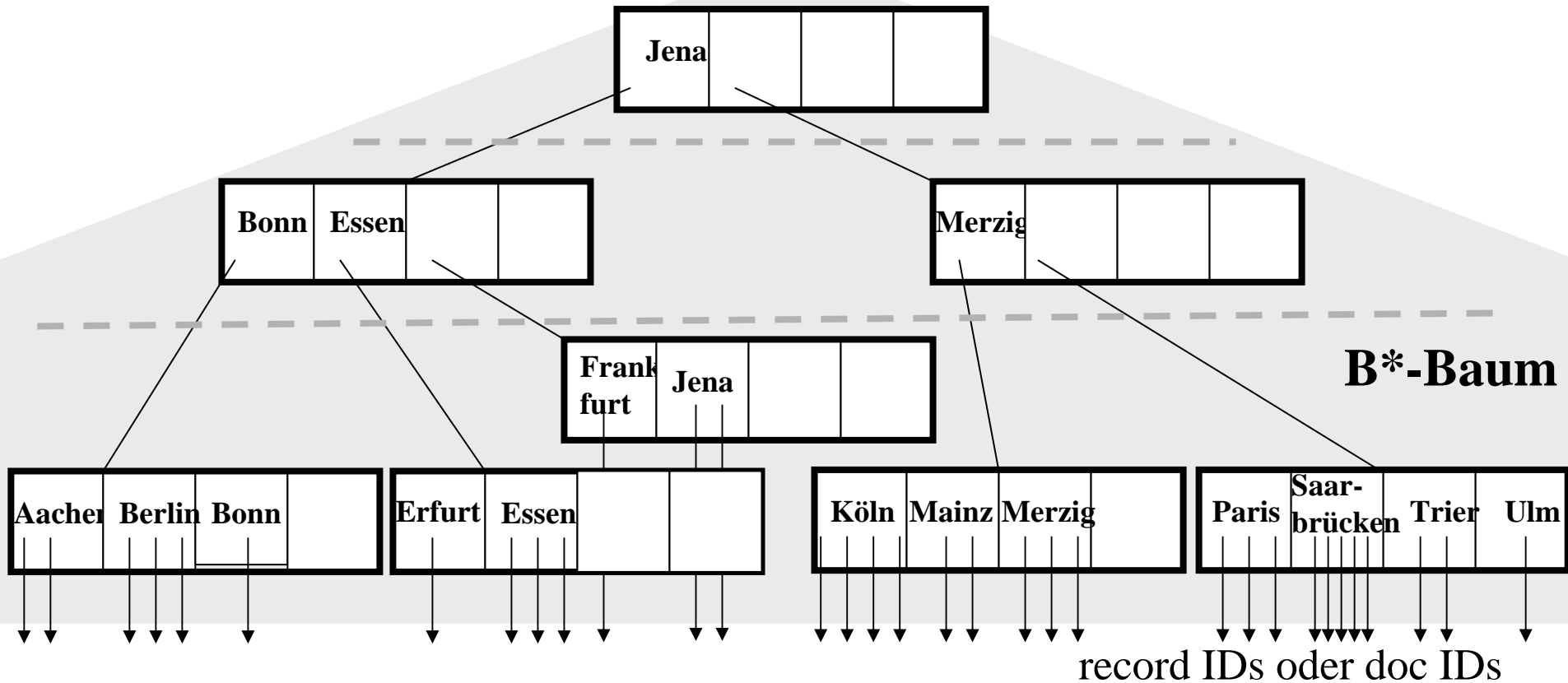
B*-Bäume: Seitenstrukturierte Mehrwegbäume

- Hohler Mehrwegebaum mit hohem Fanout (\Rightarrow kleiner Tiefe)
- Knoten = Seite auf Platte
- Knoteninhalt:
 - (Sohnzeiger, Schlüssel)-Paare in inneren Knoten
 - Schlüssel (mit weiteren Daten) in Blättern
- perfekt balanciert: alle Blätter haben dieselbe Distanz zur Wurzel
- Suche effizienz $O(\log_k n/C)$ Seitenzugriffe (Platten-I/Os)
bei n Schlüsseln, Seitenkapazität C und Fanout k
pro Baumniveau: bestimme kleinsten Schlüssel $\geq q$ und
suche weiter im Teilbaum links von q
- Kosten einer Einfüge- oder Löschoperation $O(\log_k n/C)$
- mittlere Speicherplatzauslastung bei zufälligem Einfügen: $\ln 2 \approx 0.69$

B*-Baum-Beispiel



B*-Baum-Beispiel (2)



B*-Baum-Definition

Ein Mehrwegbaum heißt B*-Baum der Ordnung (m, m^*) , wenn gilt:

- Jeder Nichtblattknoten außer der Wurzel enthält mindestens $m \geq 1$ und höchstens $2m$ Schlüssel (Wegweiser).
- Ein Nichtblattknoten mit k Schlüssel x_1, \dots, x_k hat genau $k+1$ Söhne t_1, \dots, t_{k+1} , so daß
 - für alle Schlüssel s im Teilbaum t_i ($2 \leq i \leq k$) gilt $x_{i-1} < s \leq x_i$ und
 - für alle Schlüssel s im Teilbaum t_1 gilt $s \leq x_1$ und
 - für alle Schlüssel im Teilbaum t_{k+1} gilt $x_k < s$.
- Alle Blätter haben dasselbe Niveau (Distanz von der Wurzel)
- Jedes Blatt enthält mindestens $m^* \geq 1$ und höchstens $2m^*$ Schlüssel.

Achtung: Implementierungen verwenden **variabel lange Schlüssel** und eine Knotenkapazität in Bytes statt Konstanten $2m$ und $2m^*$

Sonderfall $m=m^*=1$: **2-3-Bäume** als Hauptspeicherdatenstruktur

Pseudocode für B*-Baum-Suche

Suchen von Schlüssel s in B*-Baum mit Wurzel t :

t habe k Schlüssel x_1, \dots, x_k und $k+1$ Söhne $t_1, \dots, t_{(k+1)}$

(letzteres sofern t kein Blatt ist)

Bestimme den kleinsten Schlüssel x_i , so daß $s \leq x_i$

if $s = x_i$ (für ein $i \leq k$) und t ist ein Blatt

then Schlüssel gefunden

else

if t ist kein Blatt then

if $s \leq x_i$ (für ein $i \leq k$)

then suche s im Teilbaum t_i

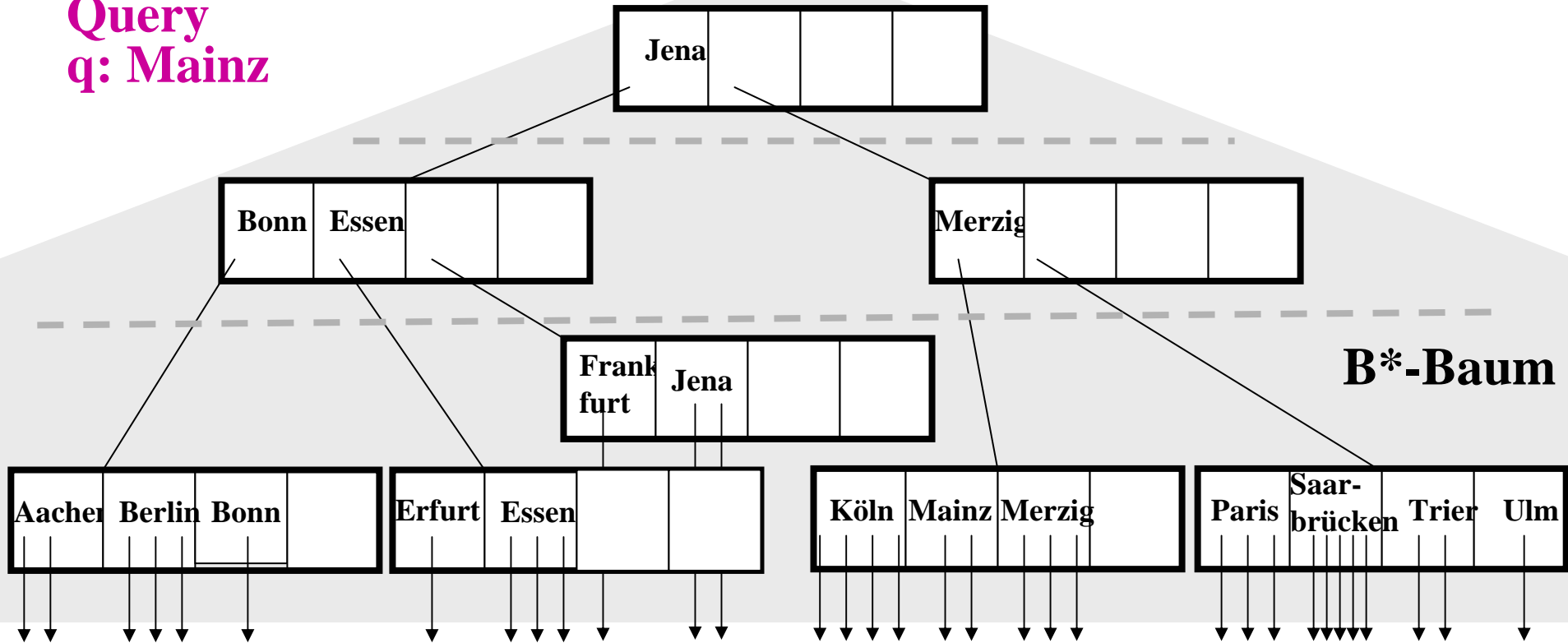
else suche s im Teilbaum $t_{(k+1)}$ fi

else Schlüssel s ist nicht vorhanden fi

fi

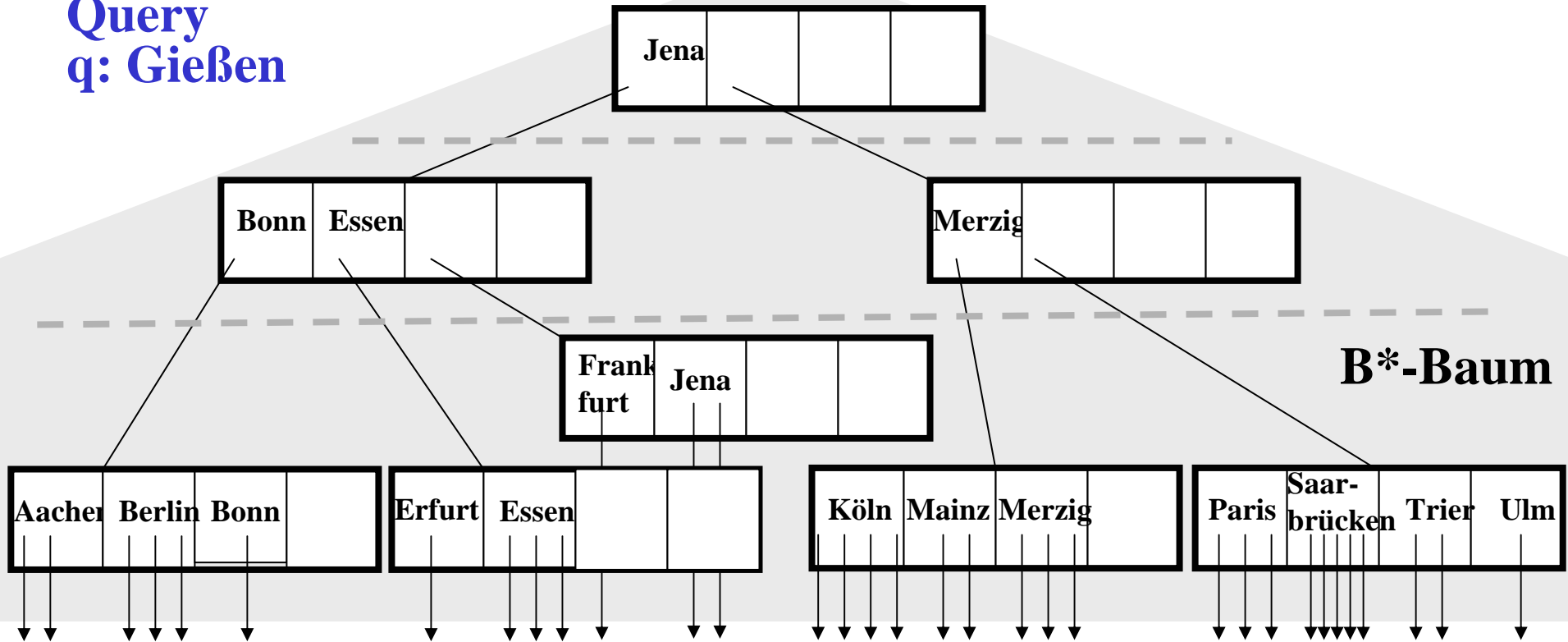
B*-Baum-Suche (1)

Query q: Mainz



B*-Baum-Suche (2)

Query q: Gießen



Pseudocode für Einfügen in B*-Baum (Grow&Post)

Suche nach einzufügendem Schlüssel e

if e ist noch nicht vorhanden then

Sei t das Blatt, bei dem die Suche erfolglos geendet hat

repeat

if t hat weniger als $2m^*$ bzw. $2m$ Schlüssel (d.h. ist nicht voll)

then füge e in t ein

else /* *Knoten-Split* */

Bestimme Median s der $2m^* + 1$ bzw. $2m + 1$ Schlüssel inkl. e

Erzeuge Bruderknoten t' /* *Grow-Phase* */

if t ist Blattknoten then

Speichere Schlüssel $\leq s$ in t und Schlüssel $> s$ in t'

else Speichere Schlüssel $< s$ (mit Sohnzeigern) in t und

Schlüssel $> s$ (mit Sohnzeigern) in t' fi

if t ist Wurzel /* *Post-Phase* */

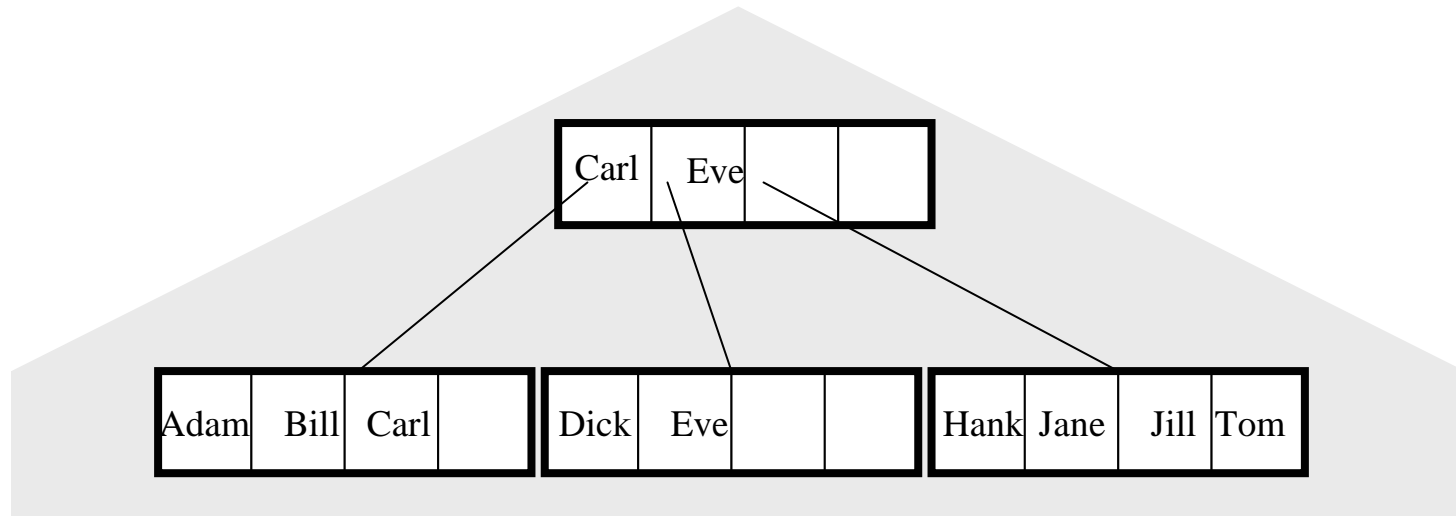
then Erzeuge neue Wurzel r mit Schlüssel s und Zeigern auf t und t'

else Betrachte Vater von t als neues t und s (mit Zeiger auf t') als e fi

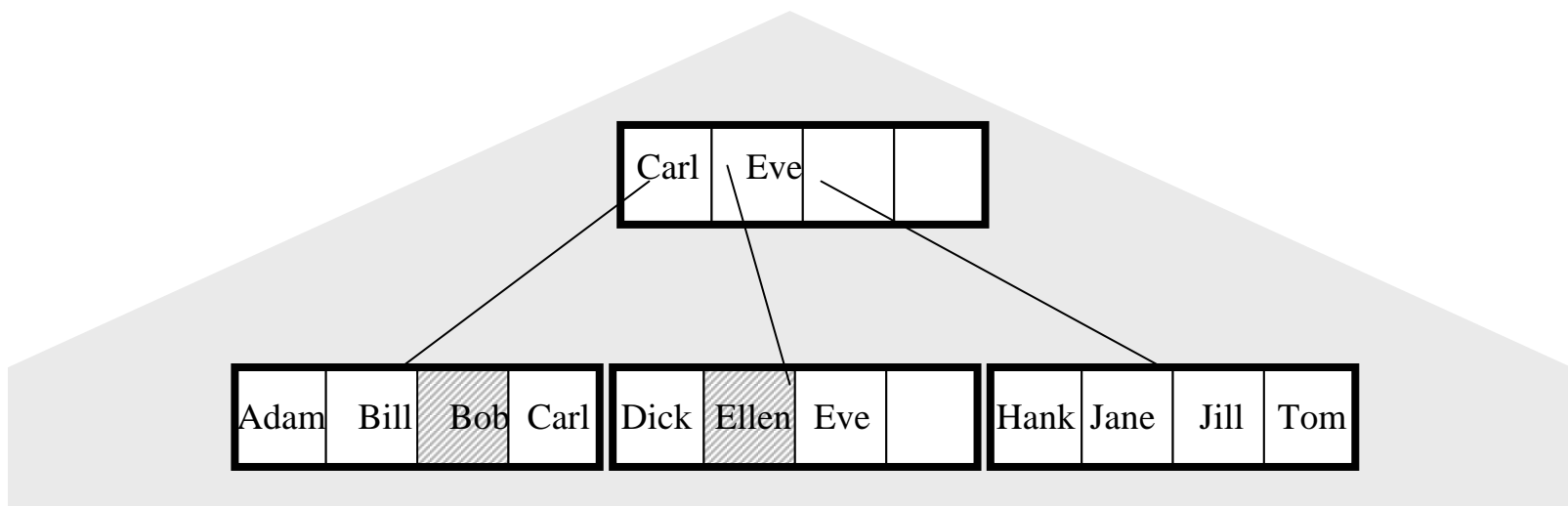
fi

until kein Knoten-Split mehr erfolgt

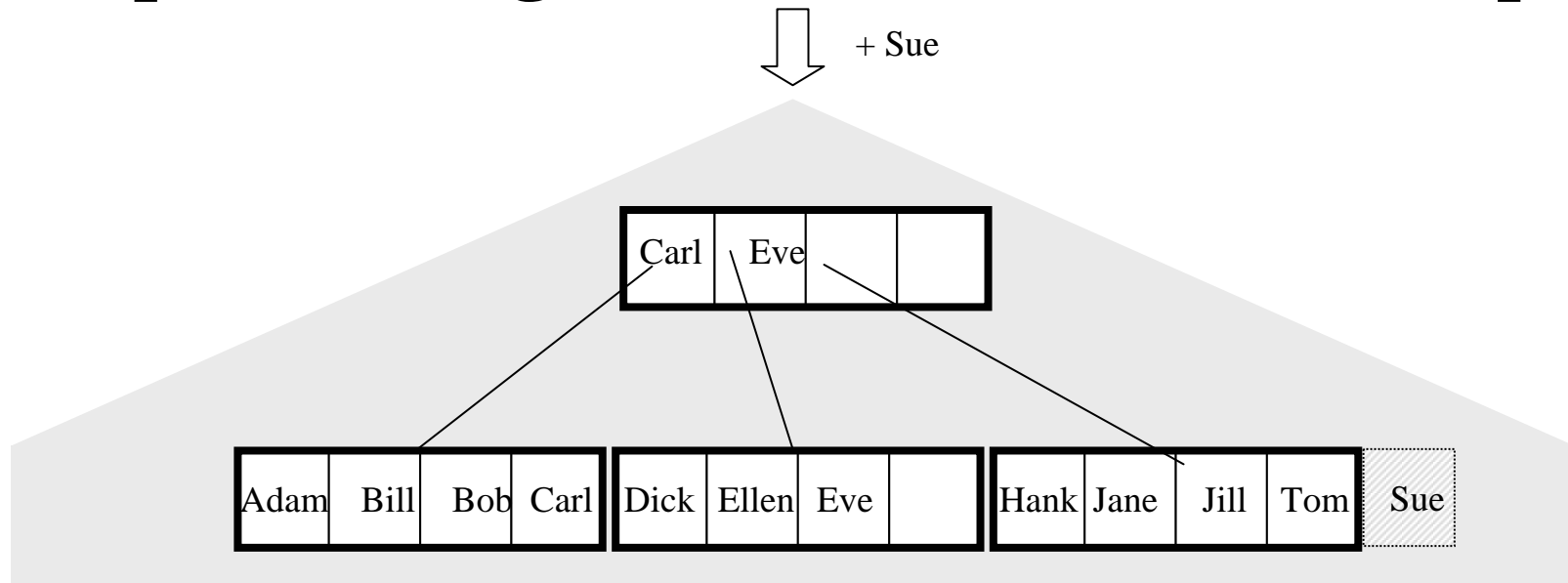
Beispiel: Einfügen in B*-Baum



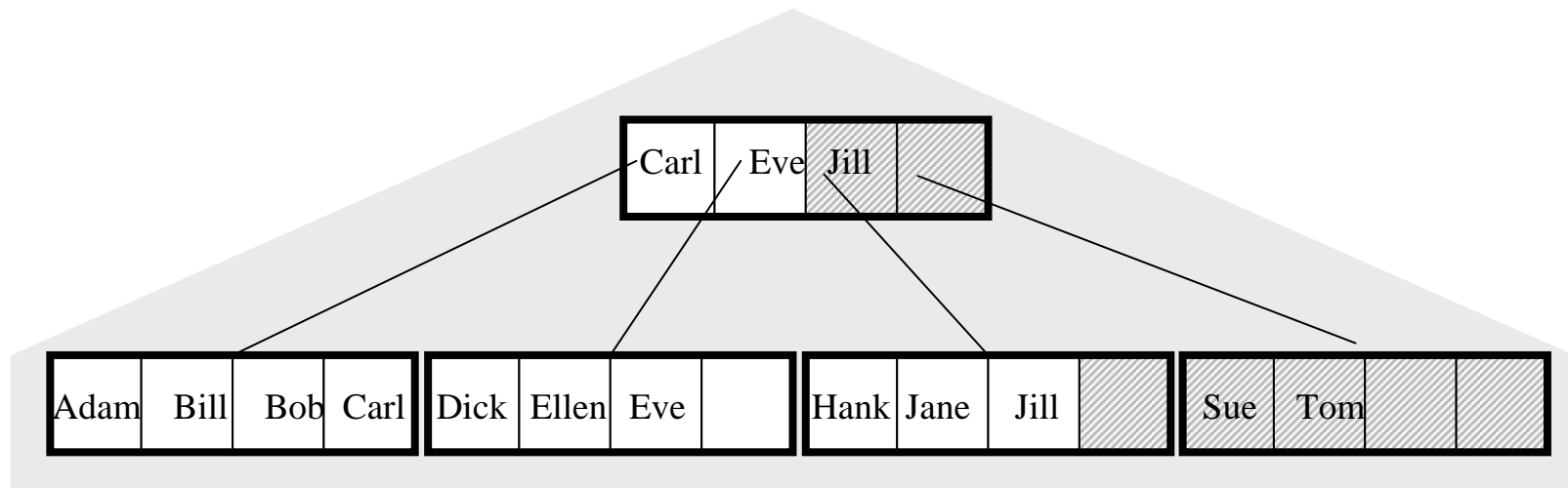
↓ + Ellen, + Bob



Beispiel: Einfügen in B*-Baum mit Blatt-Split

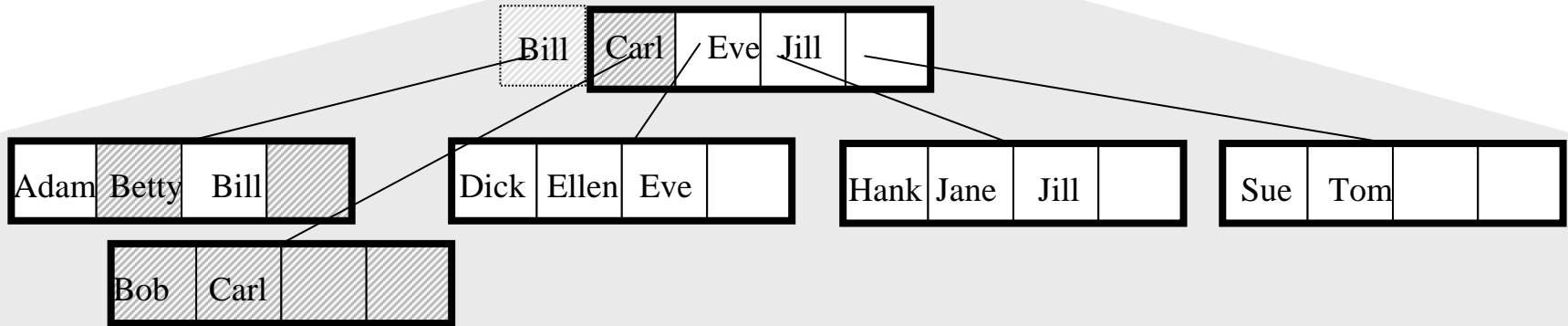


Leaf Node Split ↓

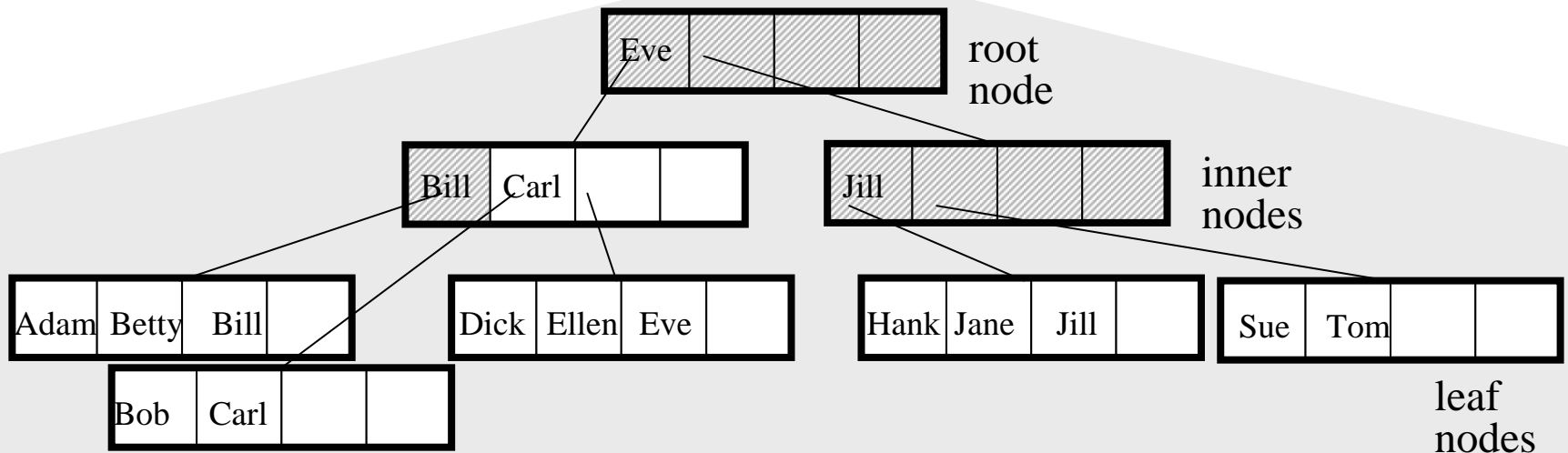


Beispiel: Einfügen in B*-Baum mit Wurzel-Split

↓ + Betty



Root Node Split

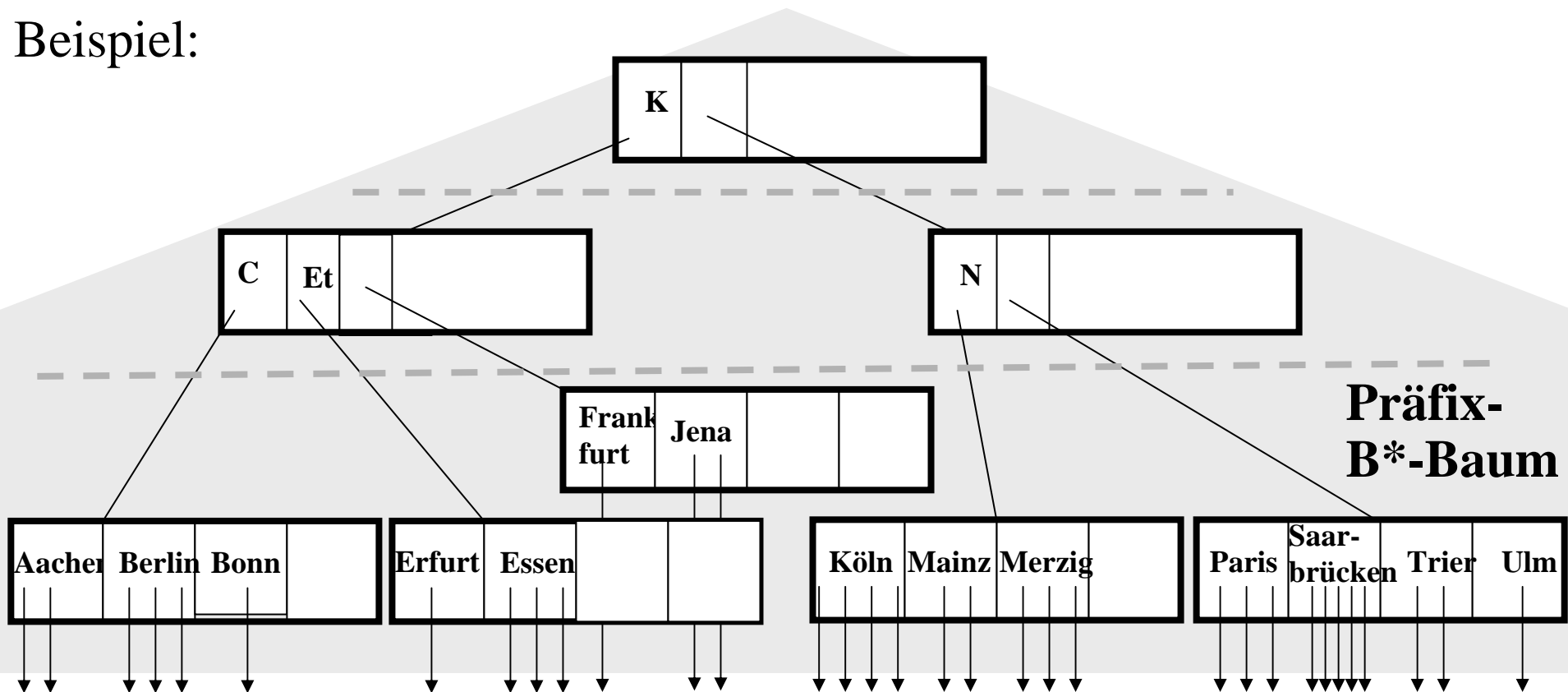


Präfix-B*-Bäume für Strings als Schlüssel

Schlüssel in inneren Knoten sind nur **Wegweiser (Router)** zur Partitionierung des Schlüsselraums.

Statt $x_i = \max\{s: s \text{ ist ein Schlüssel im Teilbaum } t_i\}$ genügt ein (kürzerer) Wegweiser x_i' mit $s_i \leq x_i' < x_{i+1}$ für alle s_i in t_i und alle s_{i+1} in t_{i+1} .
Eine Wahl wäre $x_i' = \text{kürzester String mit der o.a. Eigenschaft.}$
→ höherer Fanout, potentiell kleinere Baumhöhe

Beispiel:



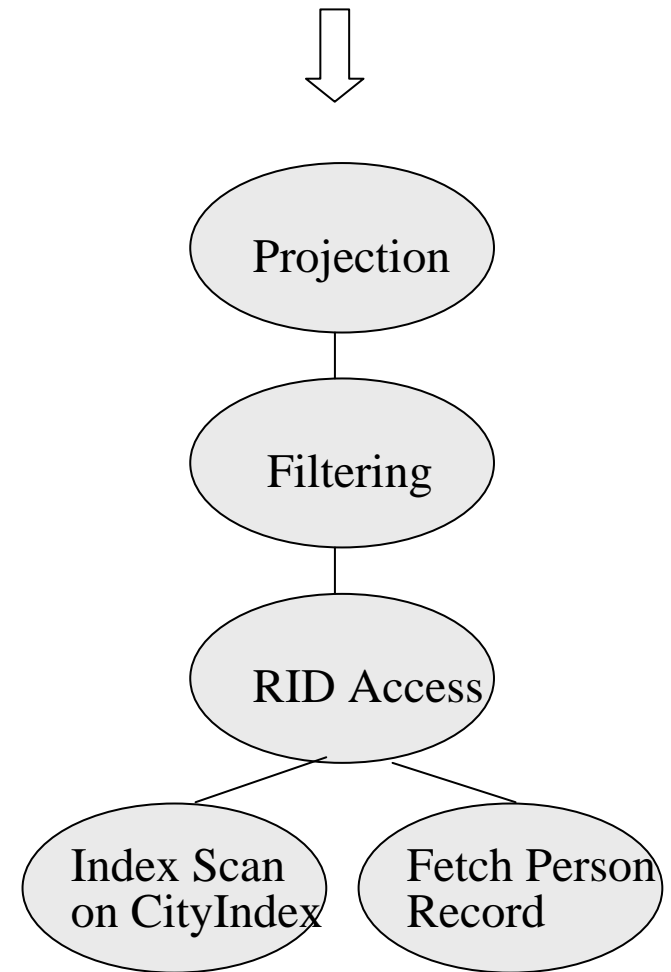
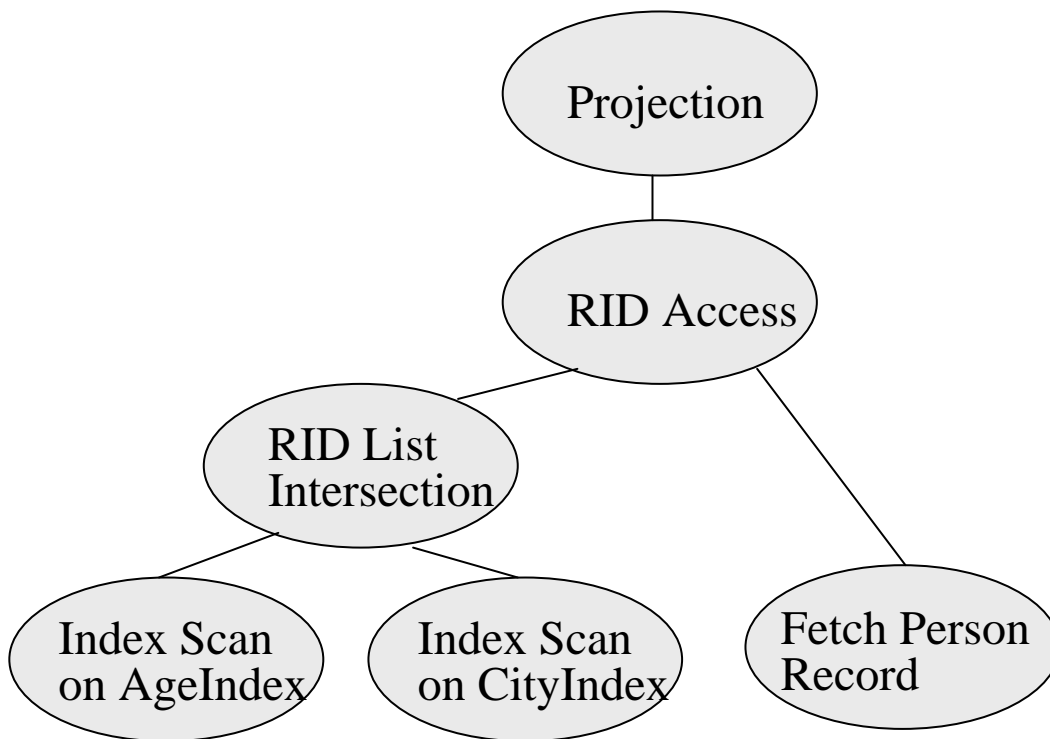
14.3 Wie Anfragen ausgeführt werden

Interne Repräsentation einer Anfrage bzw. eines Ausführungsplans als Operatorbaum mit algebraischen Operatoren der Art:

- Table-Scan
- RID-Zugriff
- Index-Scan
- Sortieren
- Durchschnitt, Vereinigung, Differenz
- Filter (Selektion)
- Projektion für Mengen
- Projekten für Multimengen
- usw.

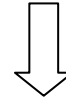
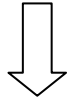
Ausführungspläne (Operatorbäume): Beispiel 1

Select Name, City, Zipcode, Street
From Person
Where Age < 30
And City = "Austin" ↓



Ausführungspläne (Operatorbäume): Beispiel 1

Select Name, City, Zipcode, Street
From Person
Where Age < 30
And City = "Austin"



in Oracle8i:

SELECT STATEMENT
TABLE ACCESS BY ROWID Person
INTERSECT
INDEX RANGE SCAN AgeIndex
INDEX RANGE SCAN CityIndex

SELECT STATEMENT
FILTER
TABLE ACCESS BY ROWID Person
INDEX RANGE SCAN CityIndex

Query-Optimierung

Weitere interne Operatoren als algorithmische Alternativen:

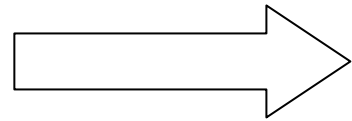
- Nested-Loop-Join
- Merge-Join
- Hash-Join
- Gruppierung mit Aggregation
- Hash-Gruppierung mit Aggregation
- usw.

Der Query-Optimierer

- generiert algebraisch äquivalente Ausführungspläne,
- bewertet deren Ausführungskosten (insbesondere #Plattenzugriffe)
- und wählt den (vermutlich) besten Plan aus.

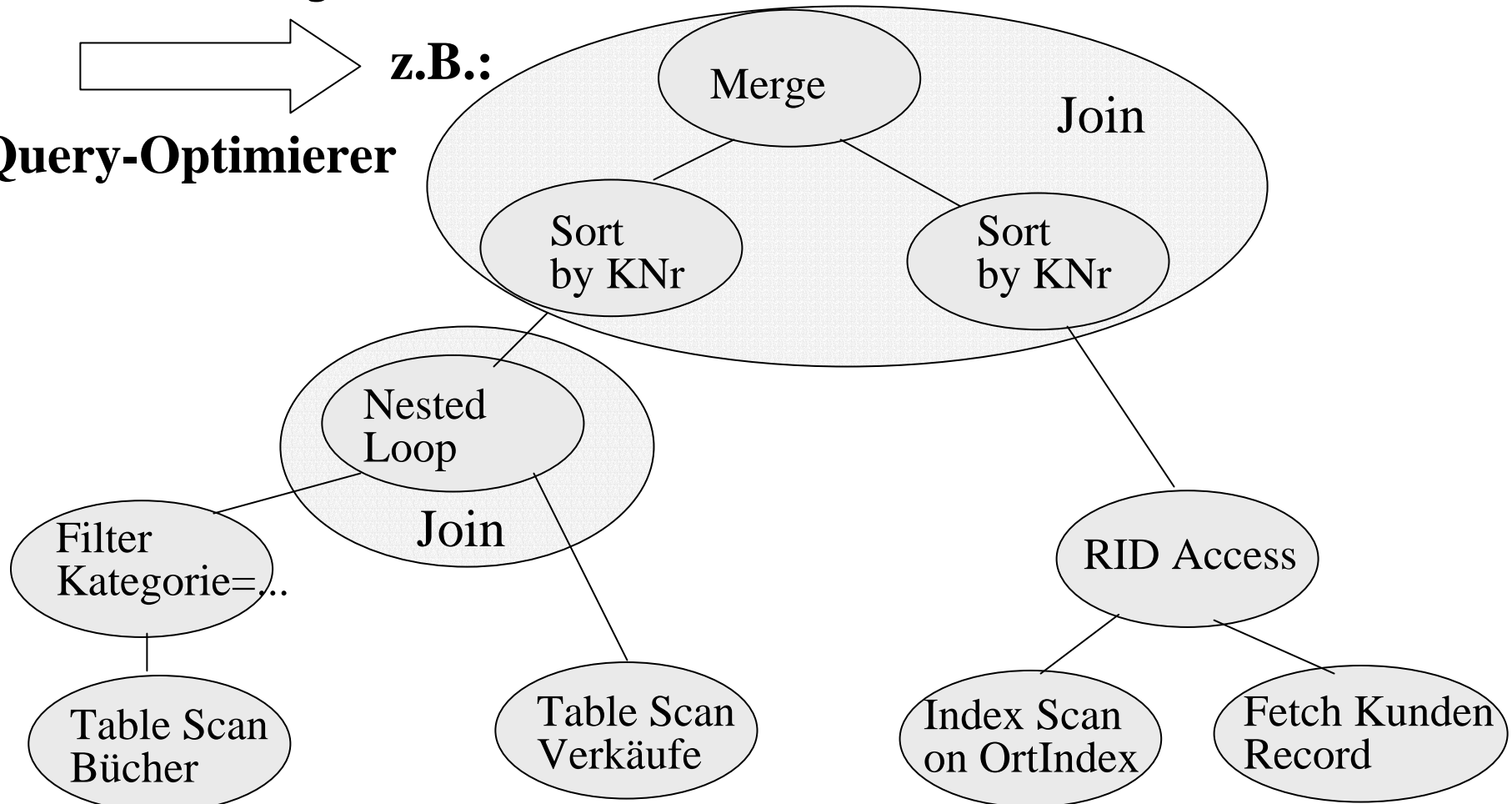
Ausführungspläne (Operatorbäume): Beispiel 2

**SELECT KNr, Name FROM Kunden K, Verkäufe V, Bücher B
WHERE K.Ort = „Saarbrücken“
AND K.KNr = V.KNr AND V.ISBN = B.ISBN
AND B.Kategorie = „Politik“**



z.B.:

Query-Optimierer



Ausführungspläne (Operatorbäume): Beispiel 2

```
SELECT KNr, Name FROM Kunden K, Verkäufe V, Bücher B
WHERE K.Ort = „Saarbrücken“
AND K.KNr = V.KNr AND V.ISBN = B.ISBN
AND B.Kategorie = „Politik“
```

Repräsentation in Oracle8i:

SELECT STATEMENT

MERGE JOIN

SORT

NESTED LOOP

FILTER

TABLE ACCESS FULL

Bücher

TABLE ACCESS FULL

Verkäufe

SORT

TABLE ACCESS BY ROWID

Kunden

INDEX RANGE SCAN

OrtIndex