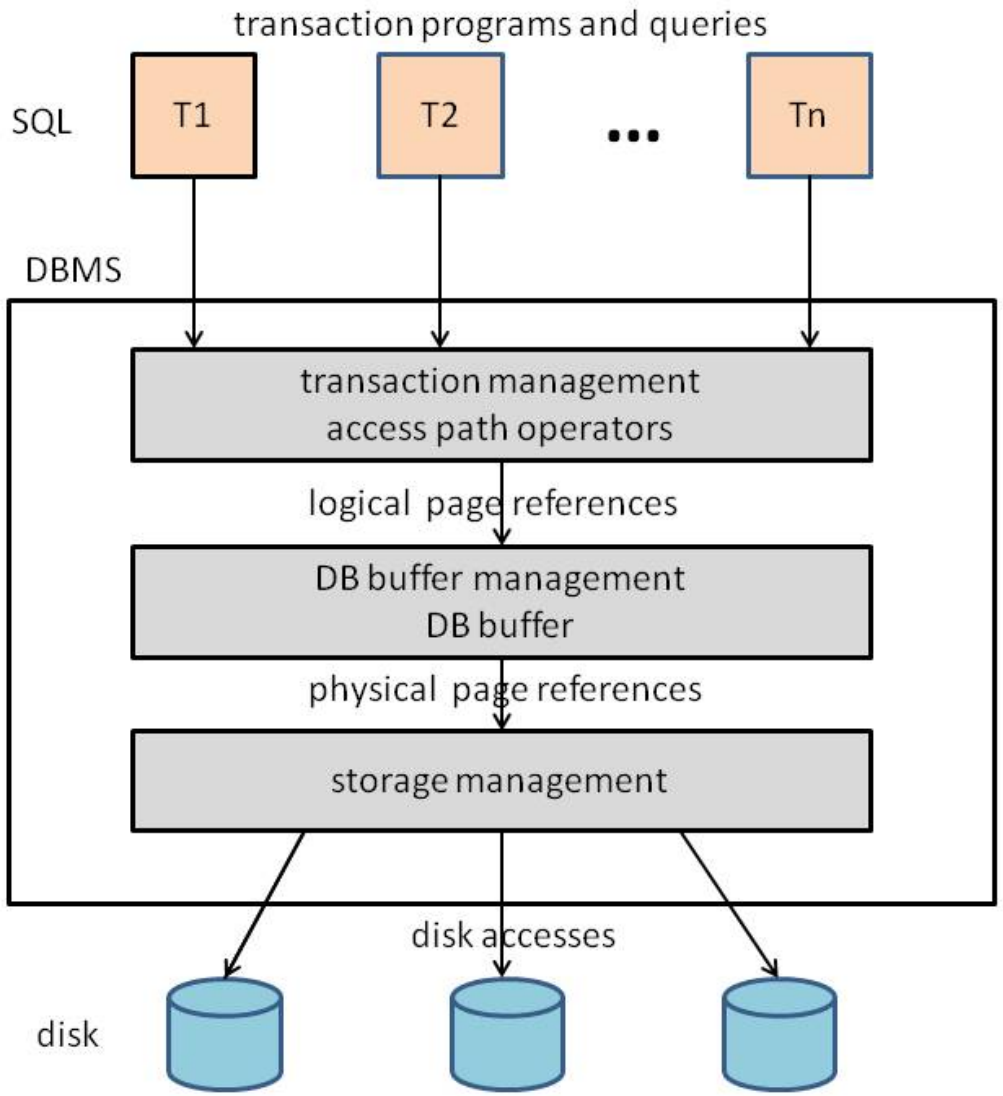
Kapitel 7: Physischer Datenbankentwurf

- Speicherung und Verwaltung der Relationen einer relationalen Datenbank so, dass eine möglichst große Effizienz der einzelnen Anwendungen auf der Datenbank ermöglicht wird.
- Vor dem Hintergrund eines logischen Schemas und den Anforderungen der geplanten Anwendungen wird ein geeignetes *physisches Schema* definiert.

Der Weg zu den Daten im Überblick



7.1 Grundlagen

Begriffe

- **Physische Datenbank**: Menge von *Dateien*
- **Datei**: Menge von gleichartigen *Sätzen*
- **Satz**: In ein oder mehrere *Felder* unterteilter Speicherbereich
- **Relationale Datenbanken**: Relationen, Tupel und Attribute

Indexstrukturen

Hilfsstrukturen zur Gewährleistung eines effizienten Zugriffs zu den Daten einer physischen Datenbank.

Zugriffseinheiten

- **Seiten** oder auch **Blöcke**
- Ein Block besteht aus einer Reihe aufeinander folgender Seiten.
- Eine Seite enthält in der Regel mehrere Tupel.
- Typischerweise hat eine Seite eine Größe von 4KB bis 8KB und ein Block eine Größe von bis zu 32KB.

Speicherhierarchie

- *Primärspeicher* (*interner Speicher*):
  Bestehend aus *Cache* und einem in Seitenrahmen unterteilten *Hauptspeicher*.
  Zugriff im Nanosekundenbereich, Größe ¡ TB.
- *Sekundärspeicher* (*externer Speicher*):
  Typischerweise in Seiten unterteilter externer Plattenpeicher.
  Zugriff im Millisekundenbereich, Größe ¿ TB.
- *Tertiärspeicher* (*Archiv*): kein direkter Zugriff möglich.

Pufferverwaltung

- Anzahl benötigter Seiten einer Datenbank typischerweise erheblich größer, als die Anzahl zur Verfügung stehender Seitenrahmen.
- *Datenbankpuffer*: Im Hauptspeicher für die Datenbank verfügbare Seitenrahmen.
- $\Rightarrow$ *Pufferverwaltung*: für angeforderte Seiten der Datenbank muss ein Seitenrahmen im Puffer gefunden werden; sonst *Seitenfehler*.
- Für jeden Seitenrahmen unterhält die Pufferverwaltung die Variablen *pin-count* und *dirty*. *pin-count* gibt die Anzahl Prozesse an, die die aktuelle Seite im Rahmen angefordert haben und noch nicht freigegeben haben; *dirty* gibt an, ob der Inhalt der Seite verändert wurde.

- Es ist aus Effizienzgründen häufig sinnvoll,
  (1) geänderte Seiten nicht direkt in die Datenbank zurück zuschreiben.
  (2) ein *Prefetching* der Seiten vorzunehmen.
- Eine Änderung heißt *materialisiert*, wenn die entsprechende Seite in den externen Speicher der Datenbank zurückgeschrieben ist.

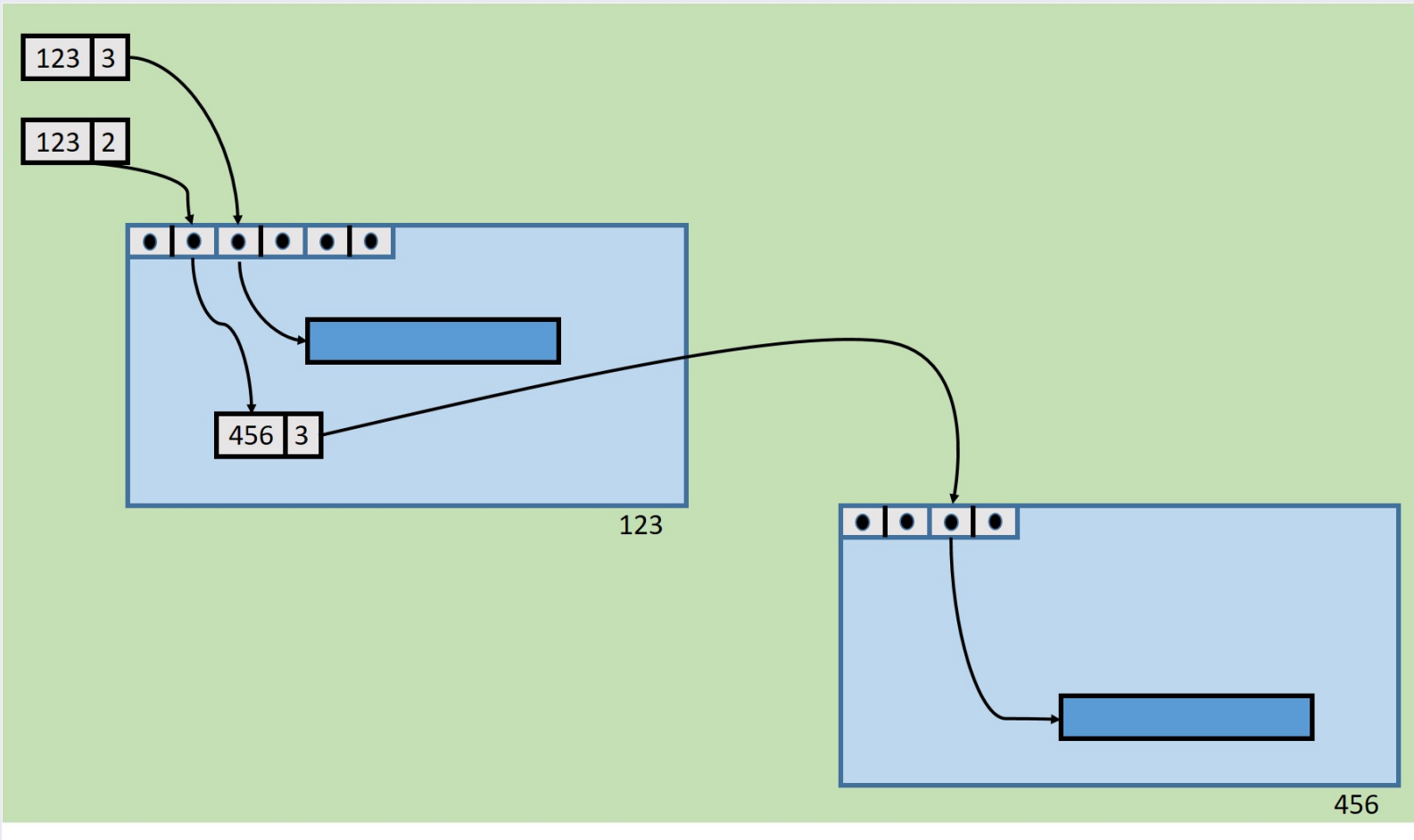
Adressierung von Tupeln/Sätzen

- *Tupelidentifikatoren* der Form $TID = (p, t)$,
  wobei p eine Seitennummer und t eine Tupelnummer innerhalb der Seite p.
- Mittels eines in der betreffenden Seite abgelegten *Adressbuchs* (*Directory*) kann in Abhängigkeit des Wertes t die Position des Tupels in der Seite gefunden werden.
- Tupel sind lediglich in ihrer Seite frei verschiebbar.
- Verschiebbarkeit über Seitengrenzen mittels *Verweisketten*.
- Tupel mit variabel langen Attributwerten verlangen *Längenzähler*,
  bzw. eine *Zeigerliste*.

logische vs. physische Adressierung

- TIDs werden zur *logischen* Adressierung verwendet:
  Seitennummer p wird beispielsweise abgeleitet mittels Hashing des Primärschlüsselwertes auf $\{1, \ldots, N\}$, wobei N die Anzahl logischer Seiten (logischer Adressraum). Alternativ: Verwendung eines Index.
- Zur Realisierung eines Datenzugriffs werden TIDs auf physische Adressen (Dateien, Plattenspeicher) abgebildet, beispielsweise mittels eines Directories.

Adressierung von Tupeln in Seiten



Exkurs (A) Tupel- vs. Spaltenorientierte Speicherung

$\Longrightarrow$ für analytische Datenbankanwendungen, in denen sehr große Tabellen verarbeitet werden, jedoch nur wenige Attribute relevant sind.

Row vs. Columnar Store

table:

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

row store:

A1	B1	C1	A2	B2	C2	A3	B3	C3

column store:

A1	A2	A3	B1	B2	B3	C1	C2	C3

Advantages:

- Column block iteration, column-specific compression,
- late tuple materialization.

Disadvantage: Updates

Exkurs (B) Externes Sortieren

Sei *Sort*() ein Verfahren zum Sortieren innerhalb des Interspeichers.
Gesucht ist ein externes Sortierverfahren *extSort*() unter Verwendung von *Sort*(), mittels dem eine beliebige extern abgelegte Relation sortiert werden kann.

Wir nehmen an, dass die zu sortierende Datei nicht komplett in den Hauptspeicher passt.

Seien k Pufferseiten verfügbar und sei N die Anzahl Seiten der Datei.

Durchlauf 0: Es werden jeweils k aufeinander folgende Seiten der Datei gelesen, intern mittels *Sort*() sortiert und wieder ausgegeben. Es werden $\lceil N/k \rceil$ sortierte Teilfolgen (sog. *Runs*) ausgegeben.

Durchlauf $i = 1, 2, \ldots$: Verwendet $k - 1$ Puffer-Seiten für die Eingabe und die verbleibende Seite für die Ausgabe. Mische $k - 1$ Runs, die innerhalb des vorherigen Durchlaufs erzeugt wurden, zu einem neuen Run.

Analyse Anzahl Seitenzugriffe

(a) Es werden $\lceil \log_{k-1} \lceil \frac{N}{k} \rceil \rceil + 1$ Durchläufe benötigt; in jedem Durchlauf werden alle N Seiten gelesen und geschrieben.

Anzahl Seitenzugriffe:

$$(\lceil \log_{k-1} \lceil \frac{N}{k} \rceil \rceil + 1)\ 2 \cdot N$$

(b) Sei $N = 10^6$, $k = 100$.

Anzahl Seitenzugriffe:

$$(\log_{99} 10^4 + 1) \cdot 2 \cdot 10^6 \approx 6 \cdot 10^6$$

7.2 Dateiorganisationsformen

Zugriffsarten

- *Durchsuchen* (*scan*) einer gesamten Relation.
- *Suchen* (*search*): Lesen derjenigen Tupel, die eine über ihren Attributen formulierte Bedingung erfüllen.
- *Bereichssuchen* (*range search*): In den Bedingungen über den Attributen stehen nicht ausschließlich Gleichheitsoperatoren, sondern beliebige andere arithmetische Vergleichsoperatoren.
- *Einfügen* (*insert*) eines Tupels in eine Seite.
- *Löschen* (*delete*) eines Tupels in einer Seite.
- *Ändern* (*update*) des Inhalts eines Tupels einer Seite.

Haufenorganisation

- Zufällige Anordnung der Tupel in den Seiten.
- Suchen eines Tupels benötigt im Mittel $\frac{N}{2}$ Seitenzugriffe.
- Einfügen eines neuen Tupels benötigt 2 Seitenzugriffe.
- Löschen und Ändern eines Tupels benötigt $\frac{N}{2} + 1$ Seitenzugriffe.

Ein *Suchschlüssel* ist die einem Index zugrunde liegende Attributkombination.

Suchbaumindex

- Die Tupel sind nach dem Suchschlüssel sortiert.
- *Suchen* eines Tupels benötigt eine logarithmische Anzahl Seitenzugriffe.
- *Einfügen*, *Löschen* und *Ändern* eines Tupels: siehe 7.4

Hashindex

- Mittels einer Hashfunktion wird einem Suchschlüsselwert eine Seitennummer zugeordnet.
- *Suchen* eines Tupels benötigt in der Regel ein bis zwei Seitenzugriffe.
- *Einfügen*, *Löschen* und *Ändern* eines Tupels benötigt in der Regel zwei bis drei Seitenzugriffe: siehe 7.5

7.3 Indexstrukturen

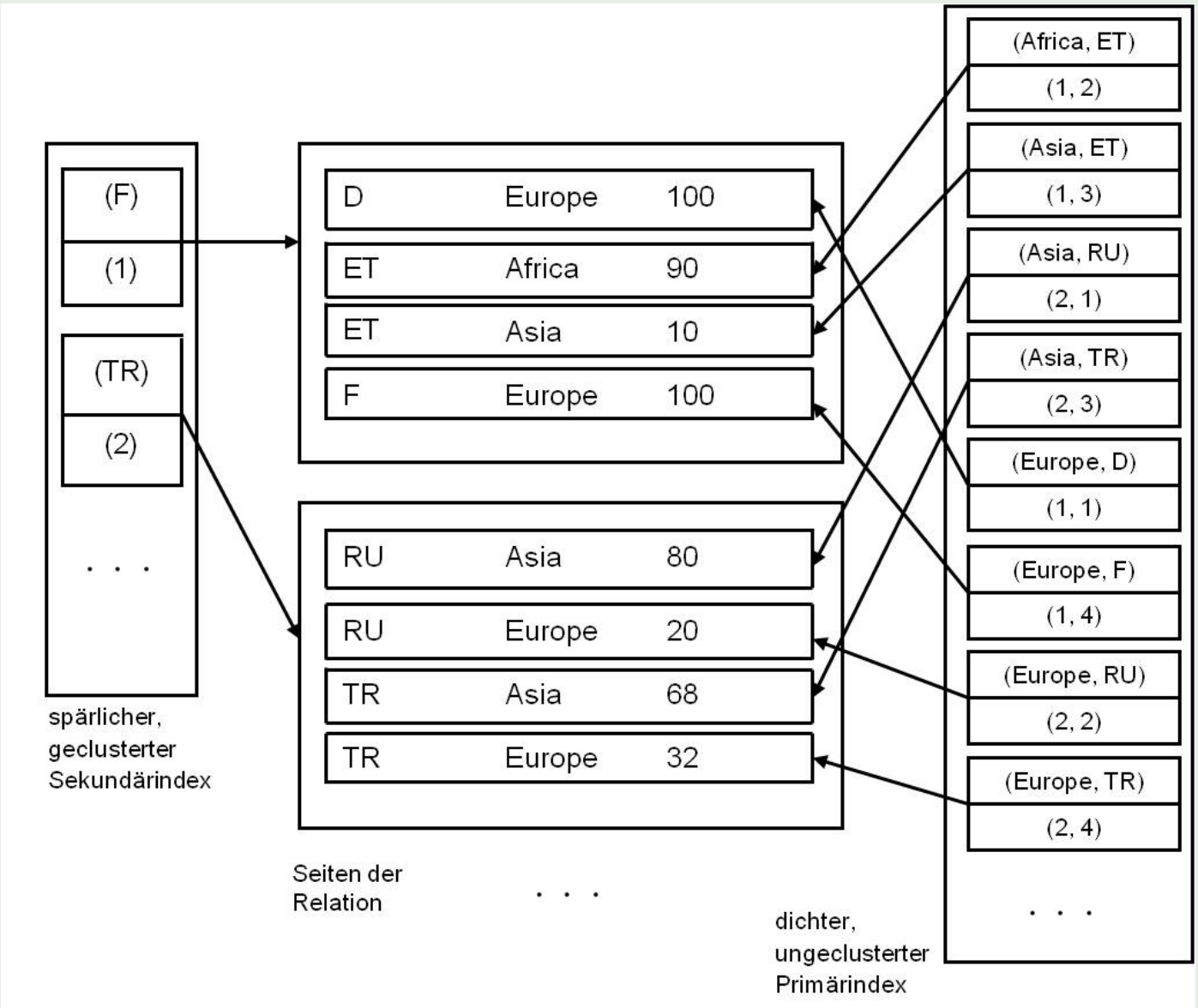
Primär- und Sekundärindex

- Bei einem *Primärindex* ist der Primärschlüssel der Relation im Suchschlüssel enthalten.
- *Sekundärindex* anderenfalls. Ein Suchschlüsselwert identifiziert hier i.A. eine Menge von Tupeln.

weitere Indexformen

- *dicht*: pro Suchschlüsselwert der Tupel der betreffenden Relation einen Verweis
- *spärlich*: pro Intervall von Suchschlüsselwerten einen Verweis
- *geballt (engl. clustered)*: logisch zusammengehörende Tupel sind auch physisch benachbart gespeichert.

Suchschlüssel (Land) vs. Suchschlüssel (Kontinent, Land) angewendet auf Relation Lage



- Zu einer Relation existieren typischerweise mehrere Indexstrukturen über jeweils unterschiedlichen Suchschlüsseln.
- Eine Relation heißt *invertiert* bzgl. einem Attribut, wenn ein dichter Sekundärindex zu diesem Attribut existiert,
- Eine Relation heißt *voll invertiert*, wenn zu jedem Attribut ein dichter Sekundärindex existiert.

7.3.1 Baum-Indexstrukturen

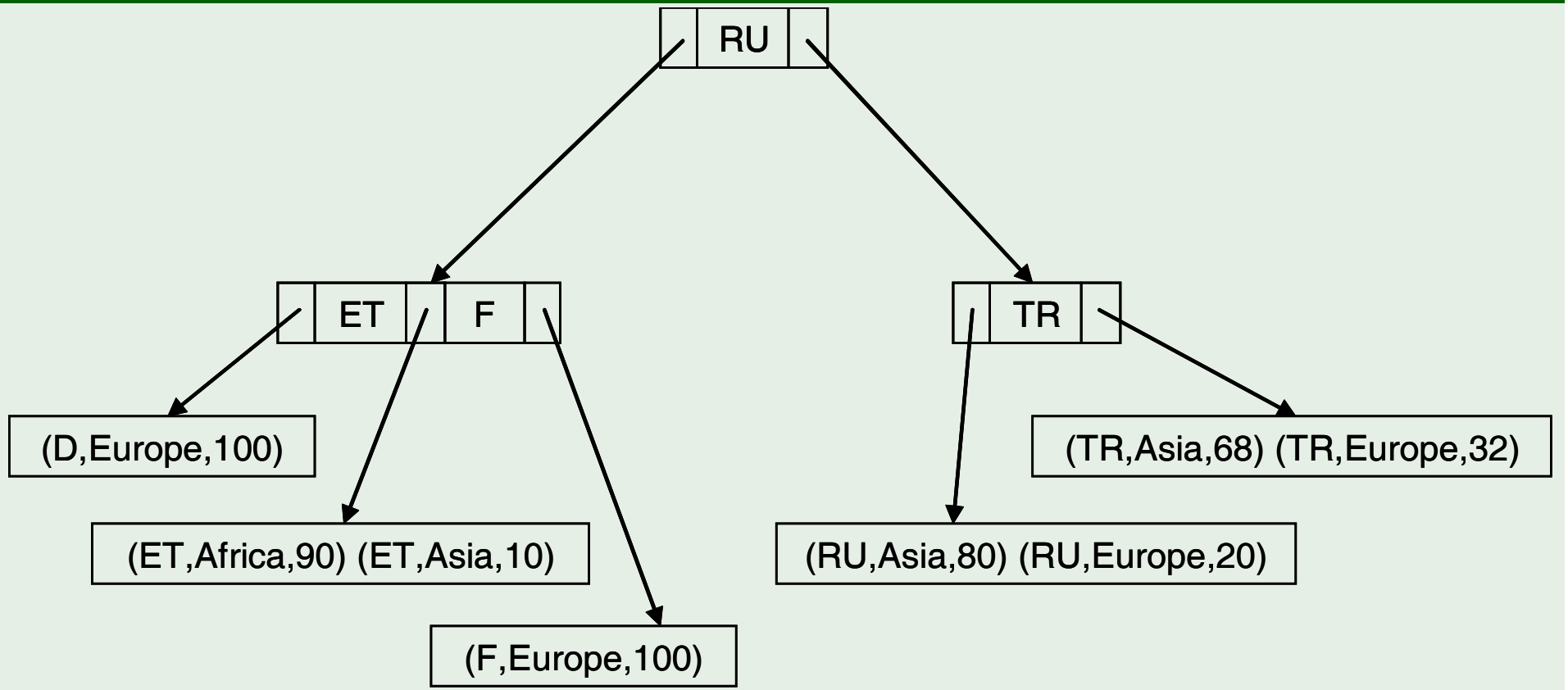
B^+-Baum[1] der Ordnung (m, l); $m > 2$, $l > 1$

- Die Wurzel ist entweder ein Blatt oder hat mindestens zwei direkte Nachfolger.
- Jeder innere Knoten außer der Wurzel hat mindestens $\lceil \frac{m}{2} \rceil$ und höchstens m direkte Nachfolger.
- Die Länge des Pfades von der Wurzel zu einem Blatt ist für alle Blätter gleich.
- Die inneren Knoten haben die Form $(p_0, k_1, p_1, k_2, p_2, \ldots, k_n, p_n)$, wobei $\lfloor \frac{m}{2} \rfloor \leq n \leq m - 1$. Es gilt:
  - p_i ist ein Zeiger auf den $i + 1$-ten direkten Nachfolger und jedes k_i ist ein Suchschlüsselwert, $0 \leq i \leq n$.
  - Die Suchschlüsselwerte sind geordnet, d.h. $k_i < k_j$, für $1 \leq i < j \leq n$.
  - Alle Suchschlüsselwerte im linken Teilbaum von k_i sind kleiner als k_i, im rechten Teilbaum von k_i sind sie größer gleich k_i, $1 \leq i \leq n$.
- Die Blätter haben die Form $(k_1^*, k_2^*, \ldots, k_g^*)$, wobei $\lfloor \frac{l}{2} \rfloor \leq g \leq l$ und k_i^* das Tupel mit Suchschlüsselwert k_i und $k_i \leq k_j$ für $i < j$.

[1] Erweiterung des B-Baums.

Beispiel: B^+-Baum $(3, 2)$

spärlicher Sekundärindex über Relation Lage zu Attribut LCode.



Eigenschaften:

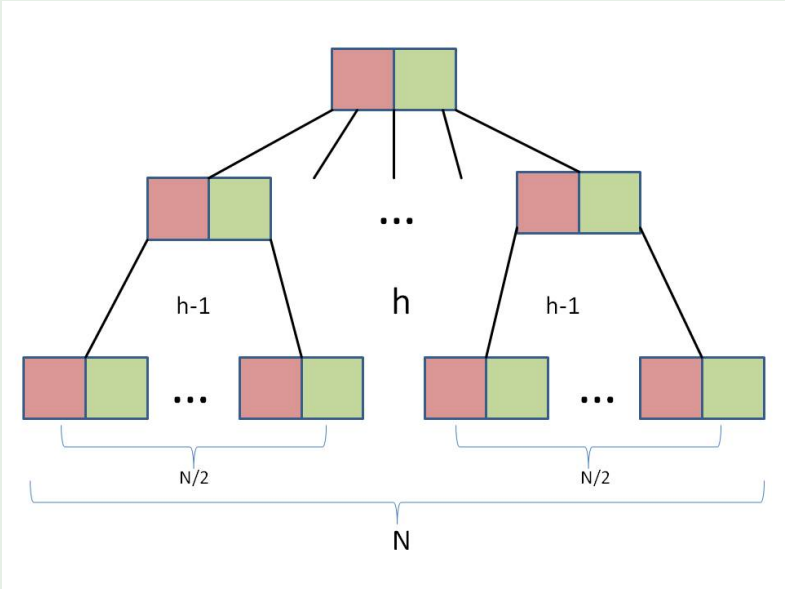
- Höhe des Baumes h: $\lceil log_m N \rceil \leq h \leq \lfloor 1 + log_{\frac{m}{2}} \frac{N}{2} \rfloor$, N Anzahl Blätter des Baumes.
- Anzahl Externzugriffe für Suchen, Einfügen und Löschen: $O(h)$.

B^+-Baum in der Praxis

- Typisch:
  Ordnung $m = 200$, Füllungsgrad 67%, Verzweigungsgrad 133, Seitengröße 8 KB.
- $h = 1$: 133 Blätter $\sim$ 1 Mbyte,
  $h = 2$: $133^2 = 17.689$ Blätter $\sim$ 133 Mbyte,
  $h = 3$: $133^3 = 2.352.637$ Blätter $\sim$ 17 Gbyte
  $h = 4$: $133^4 = 312.900.700$ Blätter $\sim$ 2 Tbyte.
- Kann u.U. bis zur Höhe 2 im Internspeicher gehalten werden.

Um die Effizienz von Zugriffen in Sortierfolge zu erhöhen, werden die Blätter in der Sortierfolge benachbarter Tupel miteinander verkettet.

Abschätzung der Höhe:

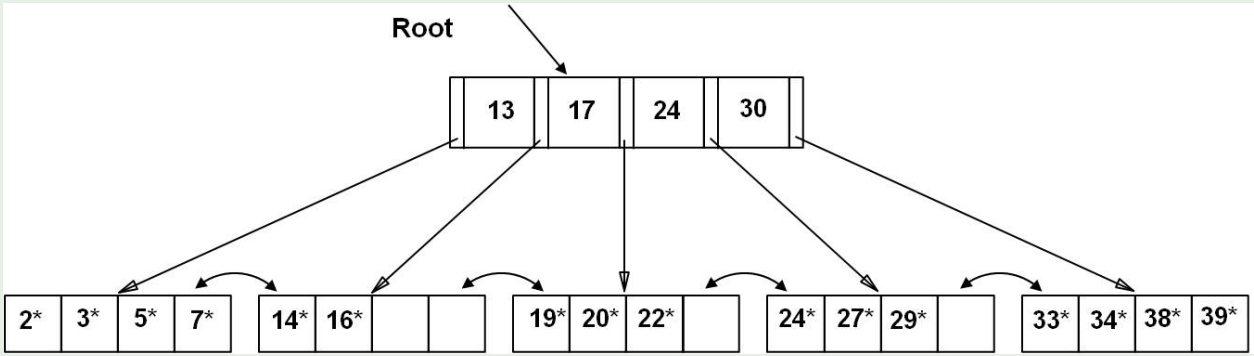


Annahme: m ganzzahlig

$$h_{min} = \lceil log_m N \rceil$$

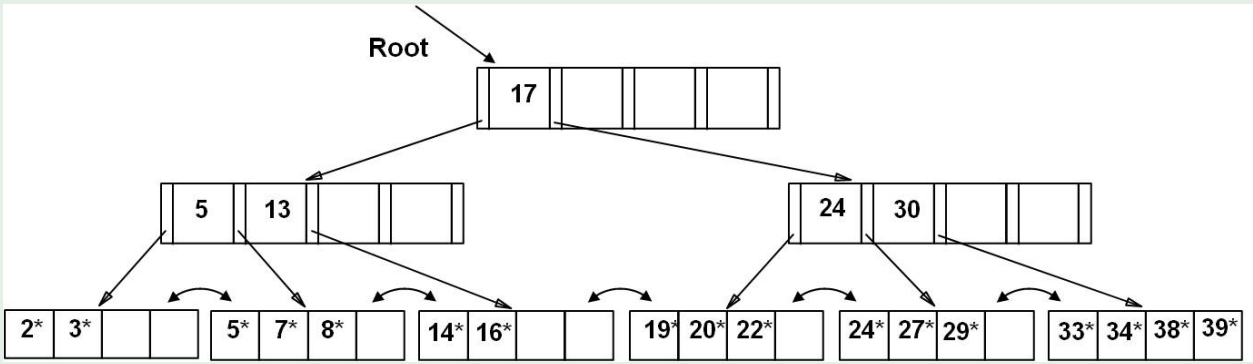
$$h_{max} = 1 + \lfloor log_{\frac{m}{2}} \frac{N}{2} \rfloor$$

B^+-Baum $(5, 4)$. Einfügen eines Tupels mit Schlüsselwert 8

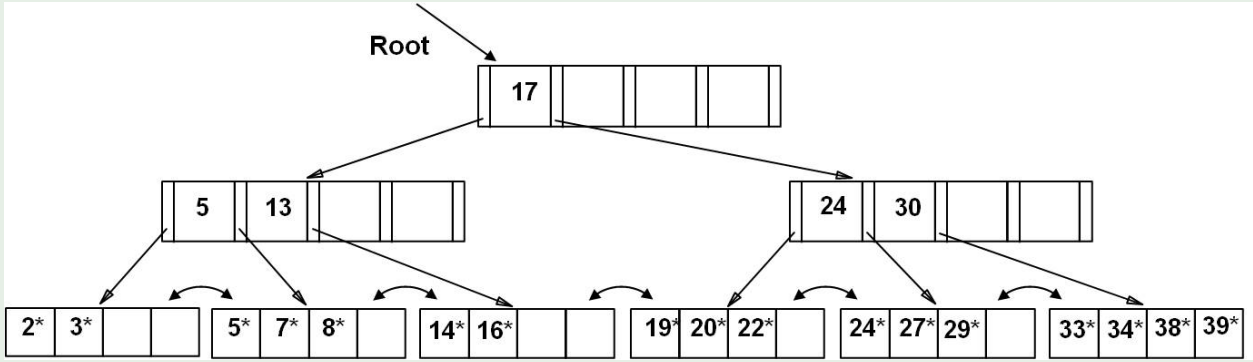


- Einfügen in das zugehörige Blatt mit möglichem Überlauf.
- Bei Überlauf durch Aufteilen ein neues Blatt bzw. einen neuen Knoten erzeugen und die zugehörige Verzweigungsinformation in den Elternknoten rekursiv einfügen.
- Die Höhe des Baumes kann bei diesem Verfahren um maximal 1 wachsen.

Ergebnis

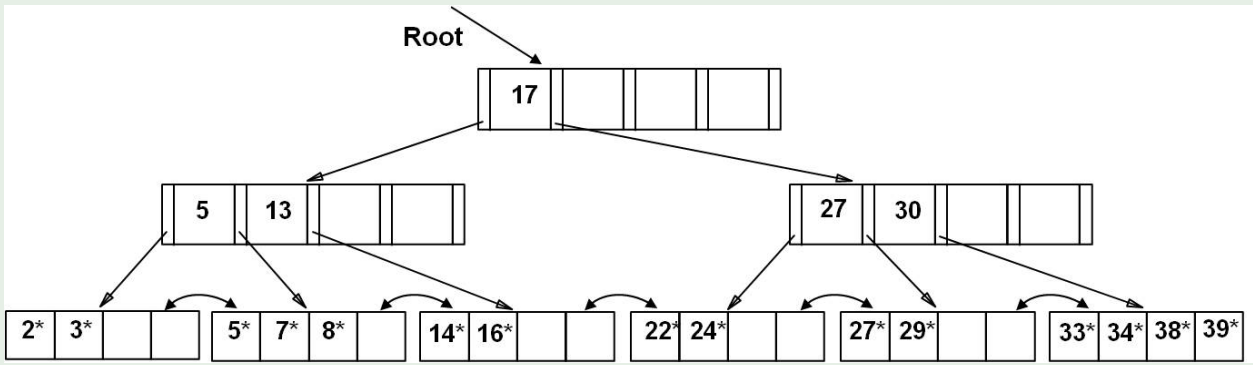


Löschen der Tupel mit Schlüsselwerten 19 und 20

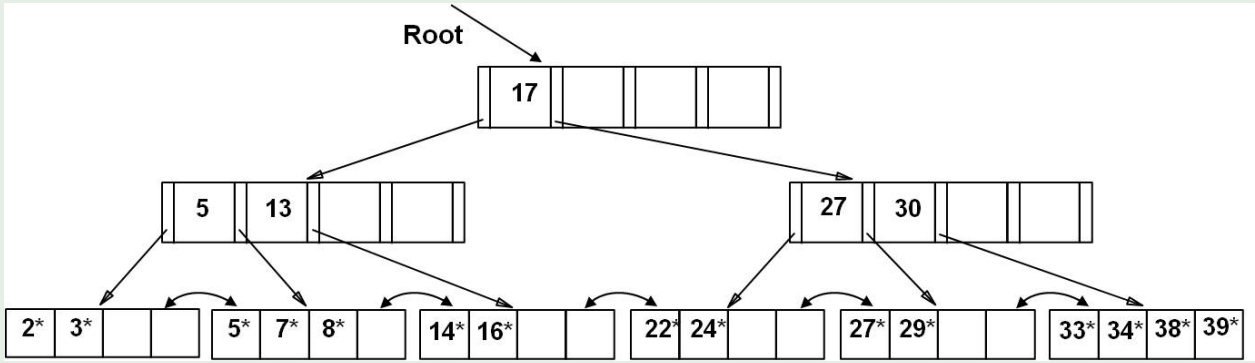


- Löschen des Tupels 19 ist unproblematisch.
- Nachfolgendes Löschen des Tupels 20 erzeugt einen Unterlauf im Blatt.
- Falls möglich: Neuverteilen der Tupel mit einem Nachbarblatt derart, dass beide Blätter die Mindestfüllung erreichen.
- Falls dies gelingt: Verzweigungsinformation im Elternknoten entsprechend anpassen.

Ergebnis

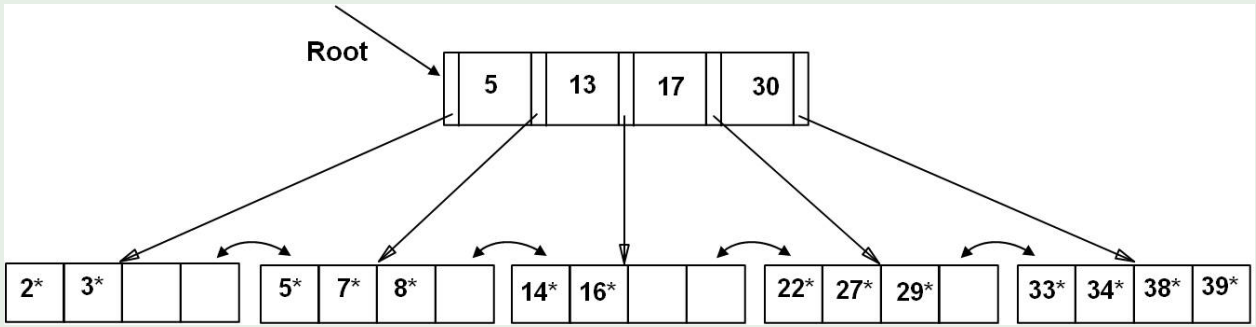


Löschen des Tupels mit Schlüsselwert 24



- Löschen des Tupels 24 erzeugt einen Unterlauf im Blatt.
- Neuverteilen der Tupel mit einem Nachbarblatt derart, dass beide Blätter die Mindestfüllung erreichen, ist nicht möglich.
- Somit das Blatt mit einem Nachbarblatt verschmelzen und rekursiv die Verzweigungsinformation im Elternknoten löschen.
- Die Höhe des Baumes kann sich um maximal 1 verringern.

Ergebnis



geclustert und ungeclusterter Index

- Sind die Tupel in den Blättern enthalten, realisiert der B^+-Baum einen geclusterten Index.
- B^+-Bäume sind ebenfalls geeignet für ungeclusterte Primär- und Sekundärindices: die Einträge k^* in den Blättern sind dann der Form $(k, \{TID\})$, wobei $\{TID\}$ die Menge der Adressen der Tupel mit Schlüsselwert k.

Bereichsanfragen

B^+-Bäume sind auch geeignet für Bereichsanfragen der Form (k *op* *value*) mit arithmetischem Vergleichsoperator *op*.

geeignet für mehrattributige Suchschlüssel?

- Konkatenation der einzelnen Attributwerte. In welcher Reihenfolge?
- Unabhängige Indexstrukturen über den einzelnen Attributen mit nachfolgender Durchschnittsbildung der Resultatsmengen. Effizienz?
- $\Rightarrow$ mehrdimensionale Zugriffsstrukturen

7.3.2 Hash-Indexstrukturen

- Eine gegebene Relation wird tupelweise mittels einer Streuungsfunktion (*Hashfunktion*) h in M Seiten $(0, \ldots, M-1)$ aufgeteilt, z.B. $h(k) = k \ mod \ M$. Die Hashfunktion sollte die Suchschlüsselwerte möglichst gleichverteilt auf die M Seiten abbilden.
- Direkter Zugriff zu den Tupeln einer Relation mit einem einzigen Externzugriff, sofern hinreichend viele Seiten zur Verfügung gestellt werden und die Hashfunktion annähernd eine Gleichverteilung der Tupel in den Seiten bewirkt.
- Werden mehr Tupel einer Seite zugeordnet als diese aufnehmen kann, so müssen der Seite *Überlaufseiten* zugeordnet werden.
- Typischerweise werden die Seiten im Mittel beim Aufbau eines Hashindex nur zu 80% gefüllt.

Bemerkung

- Zugriff zu den Tupeln gemäß einer Sortierfolge wird nicht unterstützt.
- Bereichsanfragen können nicht unterstützt werden.
- Keine Clusterung.
- Sekundärindex & mehrattributige Schlüssel analog zu B-Baum.

7.4 empfohlene Lektüre

ORGANIZATION AND MAINTENANCE OF LARGE

ORDERED INDICES

by

R. Bayer

and

E. McCreight

Mathematical and Information Sciences Report No. 20

Mathematical and Information Sciences Laboratory

BOEING SCIENTIFIC RESEARCH LABORATORIES

July 1970

ABSTRACT

Organization and maintenance of an index for a dynamic random access file is considered. It is assumed that the index must be kept on some pseudo random access backup store like a disc or a drum. The index organization described allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ where I is the size of the index and k is a device dependent natural number such that the performance of the scheme becomes near optimal. Storage utilization is at least 50% but generally much higher. The pages of the index are organized in a special data-structure, so-called B-trees. The scheme is analyzed, performance bounds are obtained, and a near optimal k is computed. Experiments have been performed with indices up to 100,000 keys. An index of size 15,000 (100,000) can be maintained with an average of 9 (at least 4) transactions per second on an IBM 360/44 with a 2311 disc.

Key Words and Phrases: Data structures, random access files, dynamic index maintenance, key insertion, key deletion, key retrieval, paging, information retrieval.

2

[2] In: Acta Informatica 1: 173-189 (1972). Kann als Report gegoogelt werden.