

184.686 VU Database Systems

Physischer Datenbankentwurf

Katja Hose

Institut für Logic and Computation

Sommersemester 2024



Informatics

Lernziele

Lernziele

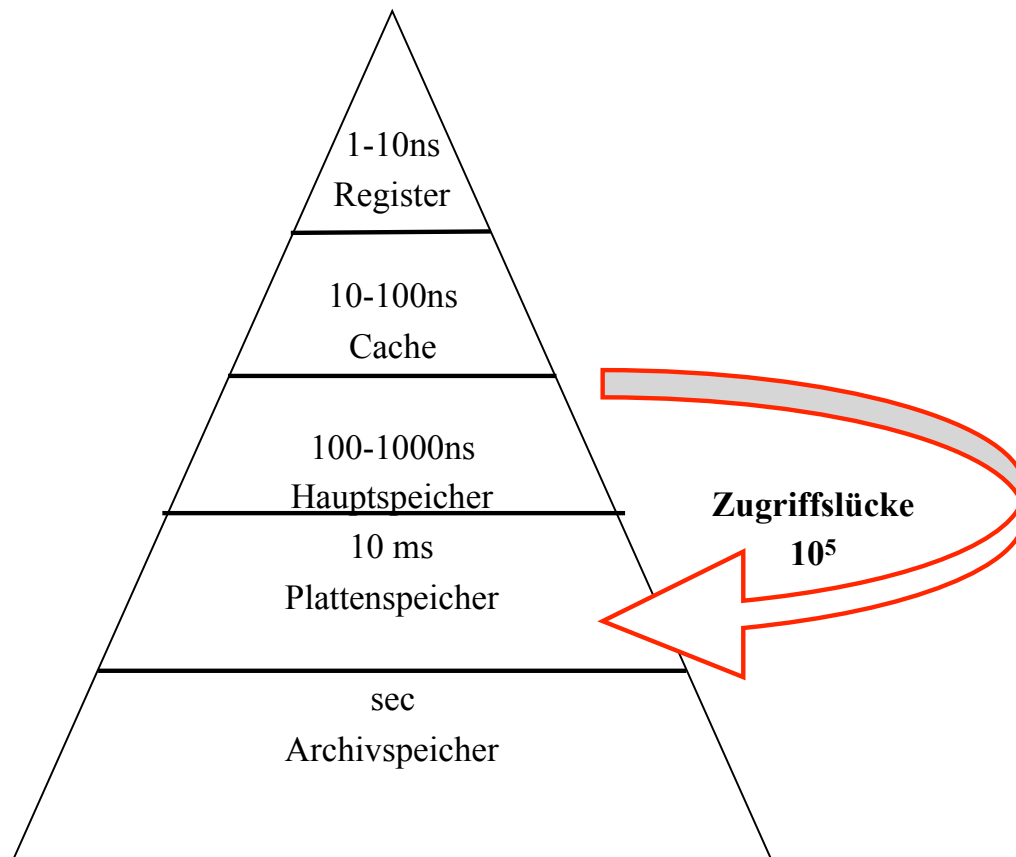
- Verstehen, wie Tabellen in Dateien gespeichert werden
- Grundzüge der Indexierung verstehen
- Indexe und Filestrukturen zum einfachen Tuning von SQL-Anfragen nutzen

Motivation

- Effiziente Anfrageausführung ist ein entscheidender Faktor in vielen Anwendungen

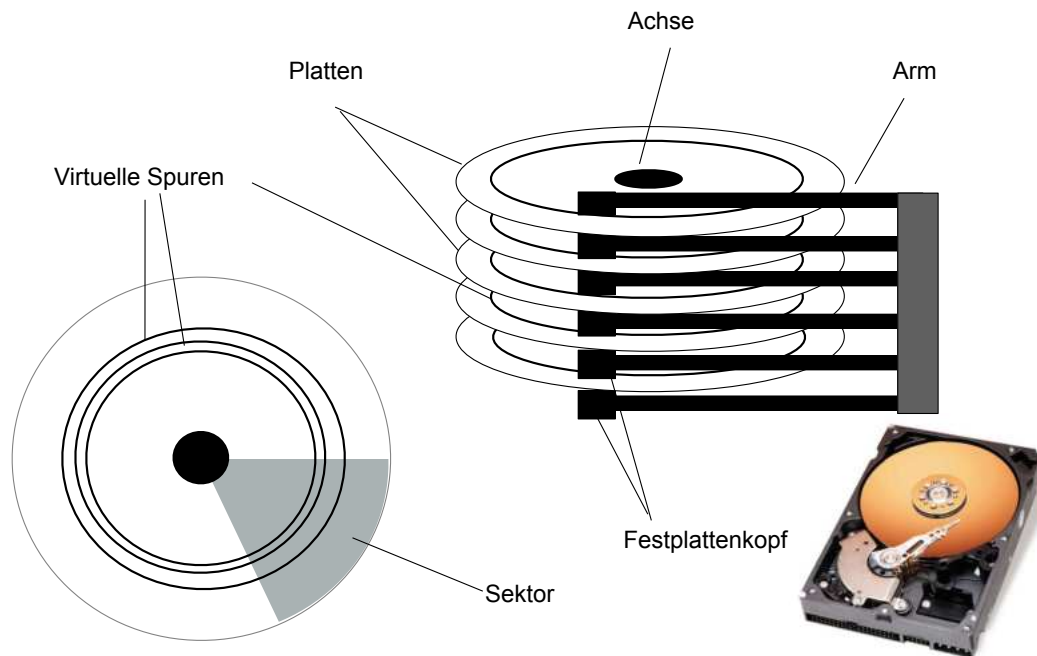
- 1 Dateiorganisation
 - Speicherorganisation
 - Dateien und Tupel (Records)
- 2 Indexstrukturen
 - Geordnete Indexe
 - B⁺-Bäume
 - Hashing
- 3 Design Tuning

Speicherebenen



- Primärspeicher (Volatile Storage) z.B. Cache, Hauptspeicher, Register: verliert den Inhalt, wenn der Strom abgeschaltet wird
- Sekundärspeicher (Non-Volatile Storage) z.B. Festplatte: behält den Inhalt, wenn der Strom abgeschaltet wird
- Tertiärspeicher (Non-Volatile Storage) z.B. Magnetbänder (Archivspeicher): behält den Inhalt, wenn der Strom abgeschaltet wird
- Je höher das Level, je schneller der Zugriff

Magnetische Festplatte



Meist sind Datenbanken auf magnetischen Platten gespeichert

- Datenbank zu groß für den Hauptspeicher
- Plattenspeicher ist persistent
- Plattenspeicher ist billiger als Hauptspeicher

Zugriffsoptimierung für Festplatten

Für jeden Speicherzugriff auf Festplatte



- Jede **Spur (Track)** ist unterteilt in **Sektoren**
- Eine **Seite (Block/Page)** ist eine kontinuierliche **Sequenz von Sektoren** eines einzigen Tracks
 - Kleinste Einheit, die zwischen Festplatte und Hauptspeicher transferiert wird

Optimierung des Festplattenzugriffs

- Blöcke in der Reihenfolge anordnen, in der sie auch benötigt werden
- Verwandte Informationen nahe beieinander ablegen

Optimierung des Festplattenzugriffs

Grundsätzlich werden relationale Daten als Sequenzen von Bits auf Festplatten gespeichert.

Funktionale Anforderungen

- Tupel (Records) sequenziell abarbeiten
- Effiziente Key-Value-Suche
- Einfügen/Löschen von Tupeln (Records)

Performanzanforderungen

- Wenig Speicherplatz verschwenden
- Schnelle Antwortzeiten
- Hoher Durchsatz von Transaktionen

Outline

- 1 Dateiorganisation
 - Speicherorganisation
 - Dateien und Tupel (Records)

Dateiorganisation

Speichern von Datenbanken auf Festplatten

- Eine Datenbank wird als Menge von **Dateien (Files)** gespeichert
- Jede Datei enthält eine Menge von **Tupeln (Records)**
- Ein Tupel/Record enthält eine bestimmte Anzahl von **Feldern (Fields)**

Mehrere Records werden in **Seiten/Blöcken (Pages/Blocks)** zusammengefasst

Größe eines Records

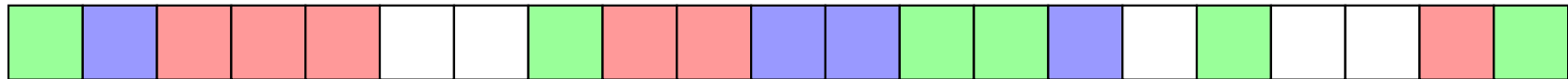
- Feste Größe (Fixed Size)
- Variable Größe (Variable Size)

Dateien werden auf Massenspeichern (normalerweise Festplatten) abgelegt

- Schneller Zugriff auf beliebige Tupel
- Sekundärspeicher

Beispiel

Blöcke in einem Dateisystem sind nicht unbedingt zusammenhängend (contiguous)



Relation A

Relation B

Relation C

freie Blöcke

Geschwindigkeit

- Geschwindigkeit des Lesezugriffs auf einen Block (ein I/O-Zugriff):
 - ≈ 10 msec für einen nicht zusammenhängenden Block
 - ≈ 1 msec für einen zusammenhängenden Block
- DBMS/OS kann Blöcke reorganisieren, um den Zusammenhang wiederherzustellen

Feste Größe

Alle Tupel haben die **gleiche** Länge (Größe),
auch wenn sie nicht den ganzen reservierten Speicherplatz benötigen

Lese Tupel i

- Lese Tupel an Position: $\text{Tupellänge (-größe)} \times i$

Feste Größe

Alle Tupel haben die **gleiche** Länge (Größe),
auch wenn sie nicht den ganzen reservierten Speicherplatz benötigen

Lösche Tupel i

- Verschiebe Tupel $i+1, \dots, n$ an Position $i, \dots, n-1$ um die Lücke zu schließen

oder

- Verschiebe Tupel n nach i

oder

- Markiere die Lücken und fülle sie später auf
 - Markiere die erste Lücke im File Header
 - Benutze die Lücken, um weitere Lücken zu referenzieren (Free-List)

Variable Größe

Tupel haben **unterschiedliche** Größen
und beanspruchen unterschiedlich viel Speicherplatz

Beispiel: Attribute mit variabler Länge/Größe wie varchar

Alternativen

- Wenn maximale Größe bekannt:
auf Tupel mit fester Größe abbilden
- Slotted-Page-Structure
 - Tupel fortlaufend speichern
 - Block Header enthält Pointer für alle Tupel

Organisation von Tupeln in Dateien

Bestimmen der Tupelreihenfolge innerhalb der Datei

Heap Files

- Tupel können an beliebiger Position in der Datei abgelegt werden

Sequenzielle Dateiorganisation

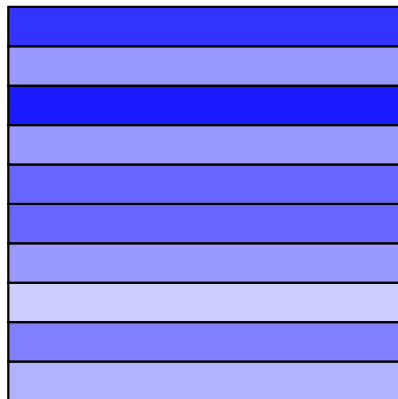
- Tupel werden sequentiell in Reihenfolge des Search Keys (z.B. ein Attribut) abgelegt

Hash Files

- Benutze eine Hashfunktion um die Position eines Tupels innerhalb der Datei zu bestimmen

Overview: file structures

Heap



Sequenziell



Hash



Heap

3	Larsen	E117
6	Rose	E167
8	Lazy	E176
1	Aaen	E111
4	Ravn	E161
7	Torp	E171
2	Dolog	E116
5	Srba	E166

- Keine bestimmte Reihenfolge innerhalb der Tabelle
- Suche: Alle Tupel nacheinander durchsuchen (Linear Scan)
- Einfügen: Finde einen freien Slot

Sequenziell

1	Aaen	E111
2	Dolog	E116
3	Larsen	E117
4	Ravn	E161
5	Srba	E166
6	Rose	E167
7	Torp	E171
8	Lazy	E176

- Tabelle ist anhand des eines Search Keys sortiert, z.B. ID Attribut
- Der Search Key muss nicht unbedingt dem Primärschlüssel entsprechen – tut es aber meistens
- Suche: Binäre Suche bei Suche anhand des Search Keys
- Einfügen: Reorganisation der Datei'

Hashing

3	Larsen	E117
6	Rose	E167
4	Ravn	E161
1	Aaen	E111
7	Torp	E171
5	Srba	E166
2	Dolog	E116
8	Lazy	E176

- Hashfunktion bestimmt Reihenfolge. z.B. $\text{mod}(\text{ID}, 3)$
- Suche: benutze die Hashfunktion mit dem Wert des Search Keys (z.B. ID Attribut), um den richtigen Block/Seite zu finden
- Einfügen: benutze die Hashfunktion mit dem Wert des Search Keys, um den richtigen Block zu finden und füge hinzu

- 2 Indexstrukturen
 - Geordnete Indexe
 - B^+ -Bäume
 - Hashing

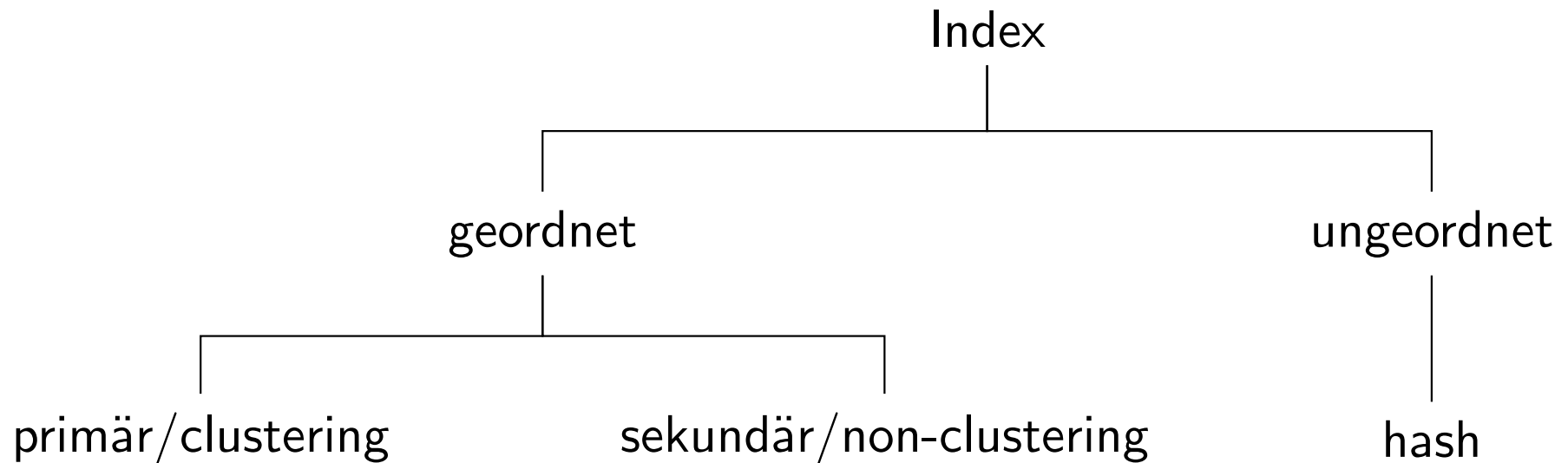
Annahmen

- In einer Datenbank werden viele Tupel abgelegt
- Viele Anfragen greifen je auf eine kleine Menge von Tupeln zu
- Tupel müssen verändert werden können; Insert, Update und Delete
- Die Datenbank kann nicht im Offline-Zustand versetzt werden, um eine Reorganisation durchzuführen

Ziel: so wenig Daten wie möglich lesen

Übersicht

Klassifikation

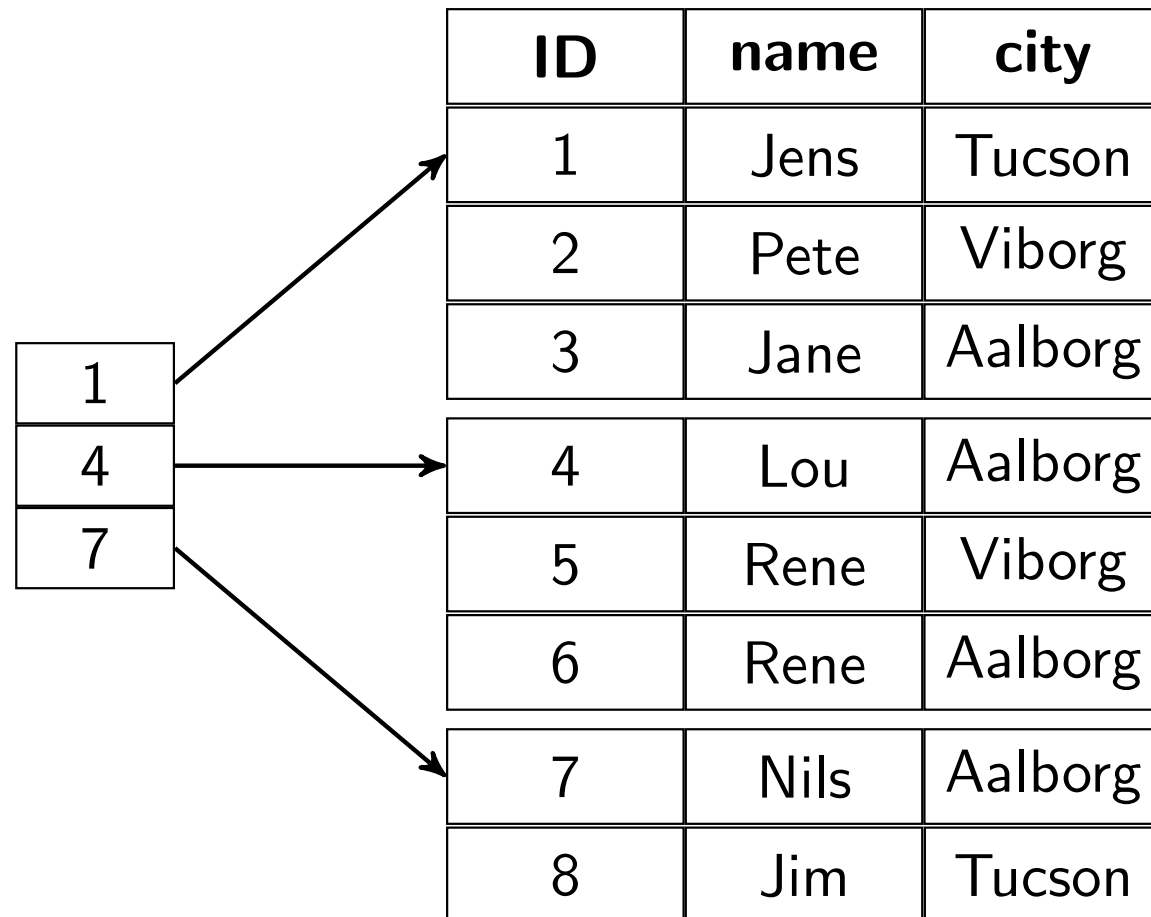


Variationen

- Single-Level Index
- Multi-Level Index

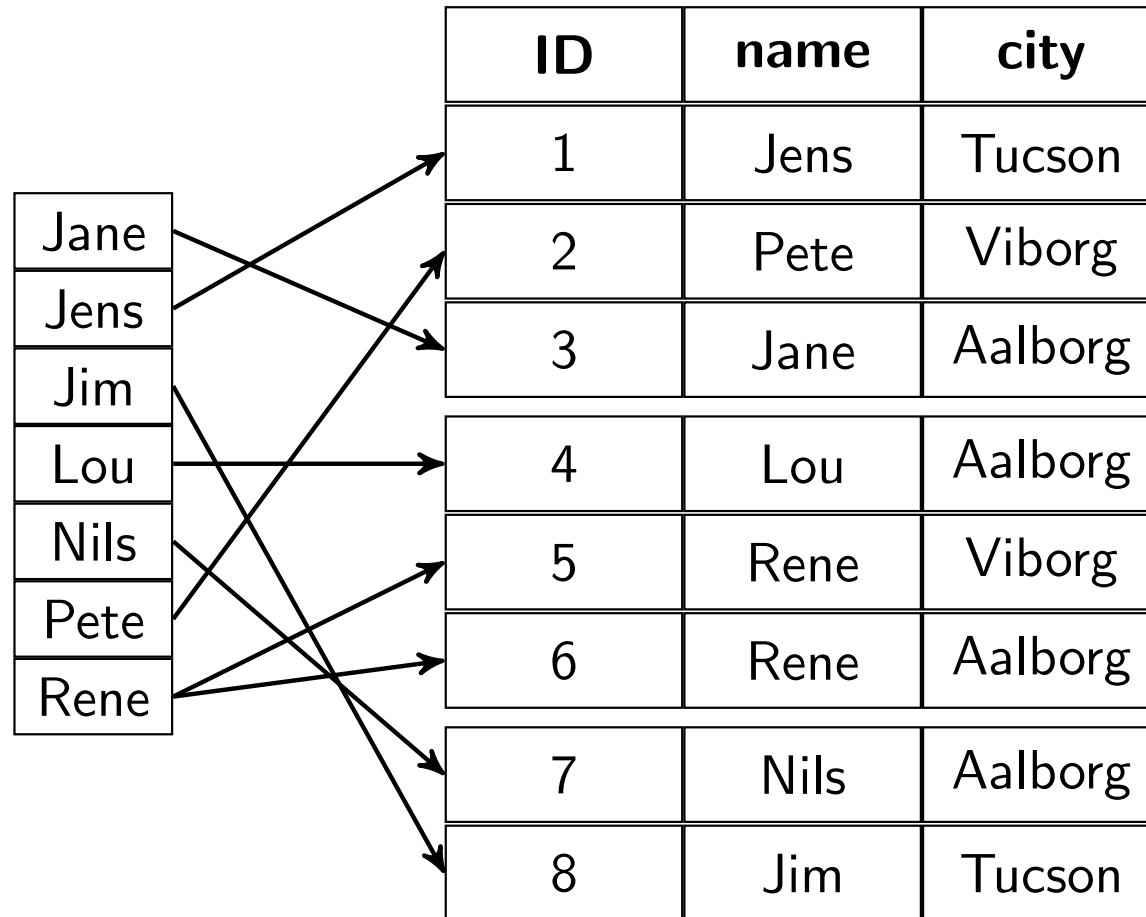
Es ist möglich, mehrere Indexe auf der gleichen Tabelle zu definieren.

Primary Sparse Index



- Definiert auf einer nach dem Search Key sortierten Datei
- Ein Eintrag pro Seite/Block in der Datei

Secondary Dense Index



- Definiert auf einer nicht nach dem Search Key sortierten Datei
- Ein Eintrag pro Tupel

Übersicht

Primär (primary) vs. sekundär (secondary)

- = Clustering vs. Non-Clustering
- Ist die Datei nach dem Search Key sortiert?
- Ja → Primär (Clustering)
- Nein → Sekundär (Non-Clustering)

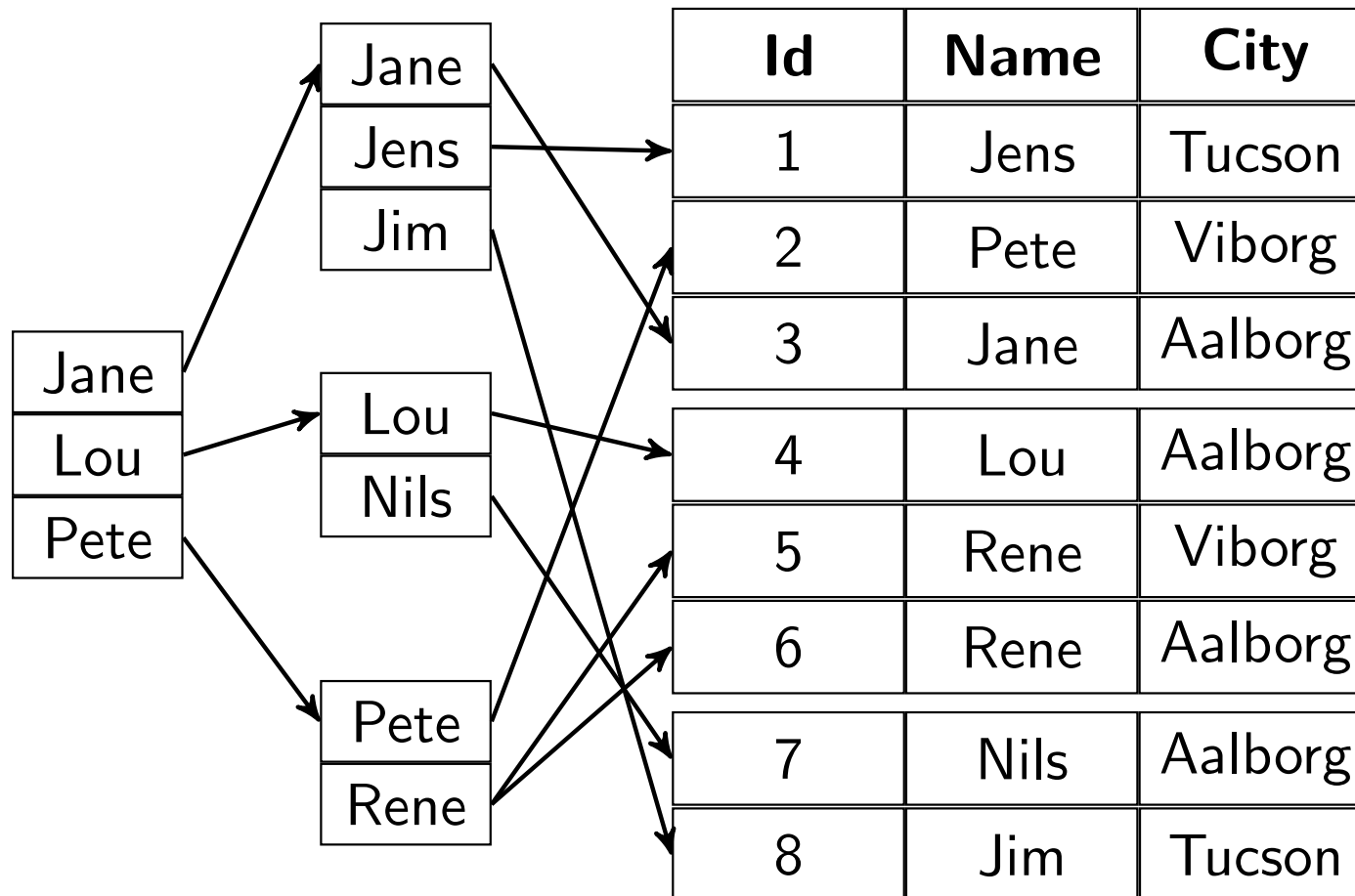
Dense vs. sparse

- Ein separater Eintrag für jedes Tupel bzw. jeden vorkommenden Wert des Search Keys?
- Ja → dense
- Nein → sparse

Tradeoff

- Dense Index: schnelleres Finden von Tupeln
- Sparse Index: weniger Speicherplatz

Multi-Level Index



- Ziel: der äußere Index (sparse) passt in den Hauptspeicher
- Der Index kann mehr als 2 Ebenen haben

Multi-Level Index

Einschränkungen von geordneten Dateistrukturen

- Einfügen und löschen
 - Teils teure Reorganisation von mehreren Ebenen von sortierten Dateien

B⁺-Bäume

- Balancierte Suchbäume
 - Die Anzahl von Lookups/Levels ist für alle Einträge gleich
- Etwas Platz auf jeder Seite/Block lassen

Kombinationen von Konzepten

	dense	sparse
primary (clustering)	✓	✓
secondary (non-clustering)	✓	%

Ein Secondary Sparse Index ist nicht sinnvoll!

- Was wäre die Folge eines solchen Indexes?

Kombinationen von Konzepten

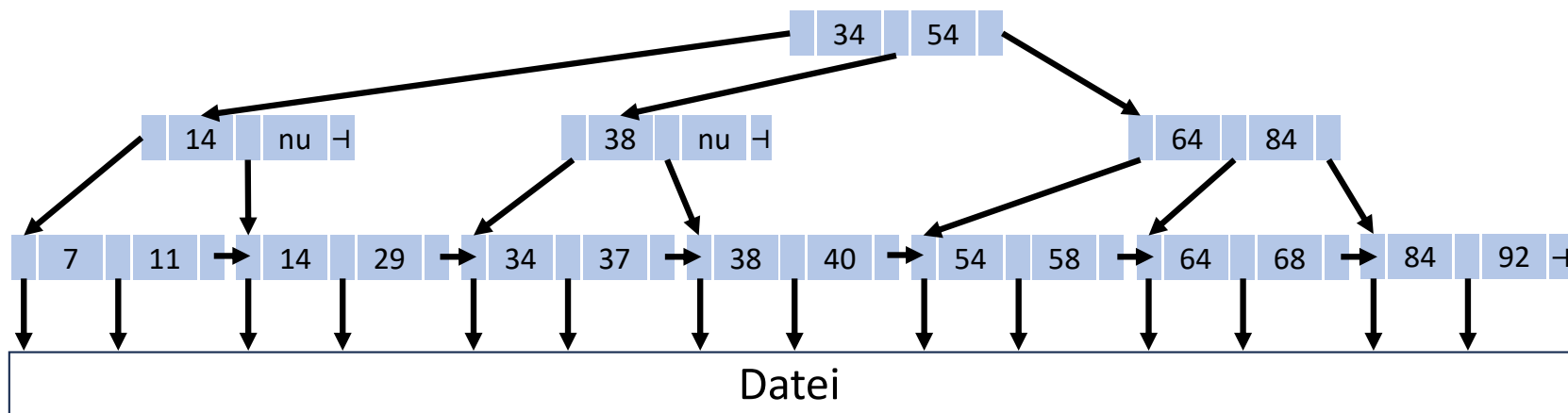
	dense	sparse
primary (clustering)	✓	✓
secondary (non-clustering)	✓	%

Ein Secondary Sparse Index ist nicht sinnvoll!

- Da Tupel nicht anhand des Search Keys sortiert sind, können wir anhand des ersten Tupels einer Seite keine Rückschlüsse auf die restlichen Tupel der Seite schließen.
- Daher sind Secondary Indexes immer dense.
- Clustering dense:
Indexeintrag für jeden vorkommenden Wert des Search Keys → das erste Tupel
- Non-clustering dense:
Indexeinträge für alle Tupel

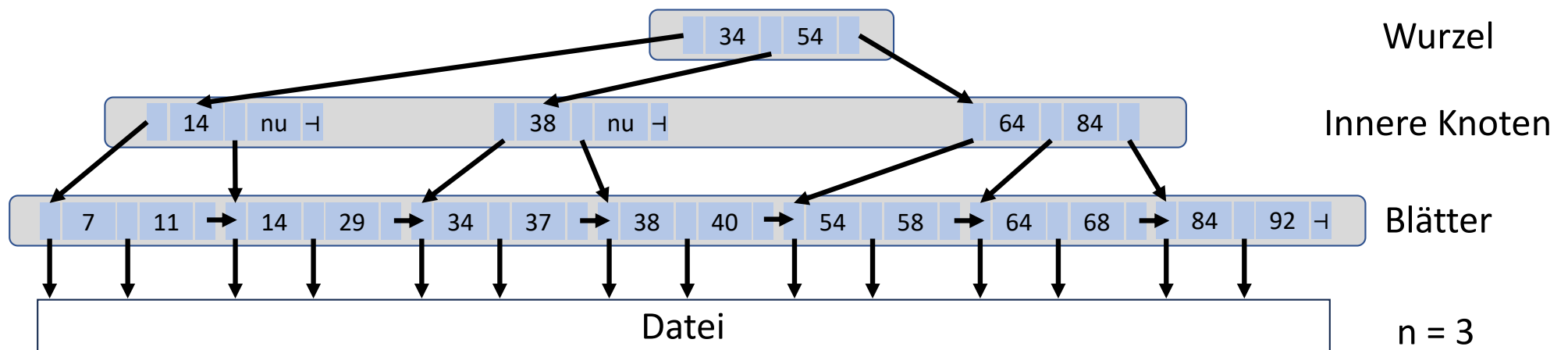
2 Indexstrukturen

- Geordnete Indexe
- B^+ -Bäume
- Hashing

B⁺-Baum Beispiel

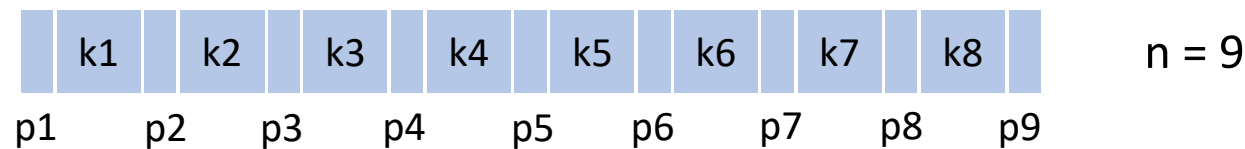
n = 3

- Verzweigungsgrad/Ordnung (Branching Factor, Fanout) n
- \dashv = unbenutzter Pointer
- nu (not used): nicht benutzter Eintrag eines Knotens
- Pointer auf Blattebene zeigen auf Positionen in der Datei

B⁺-Baum Beispiel

- Verzweigungsgrad/Ordnung (Branching Factor, Fanout) n
- \dashv = unbenutzter Pointer
- nu (not used): nicht benutzter Eintrag eines Knotens
- Pointer auf Blattebene zeigen auf Positionen in der Datei

B^+ -Baum Knotenstruktur

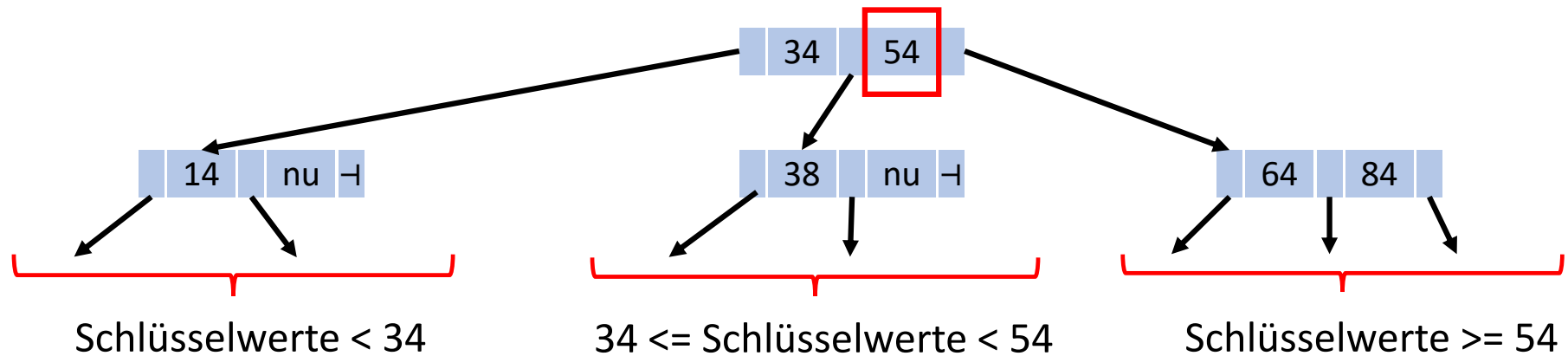


Wurzel, innere Knoten und Blätter haben die gleiche Struktur

- Jeder Knoten hat 9 (n) **Pointer** p_i
- Jeder Knoten hat 8 ($n - 1$) **Search-Key**-Werte k_j

Der letzte Pointer auf Blattebene zeigt auf den nächsten Blattknoten in Search-Key-Reihenfolge.

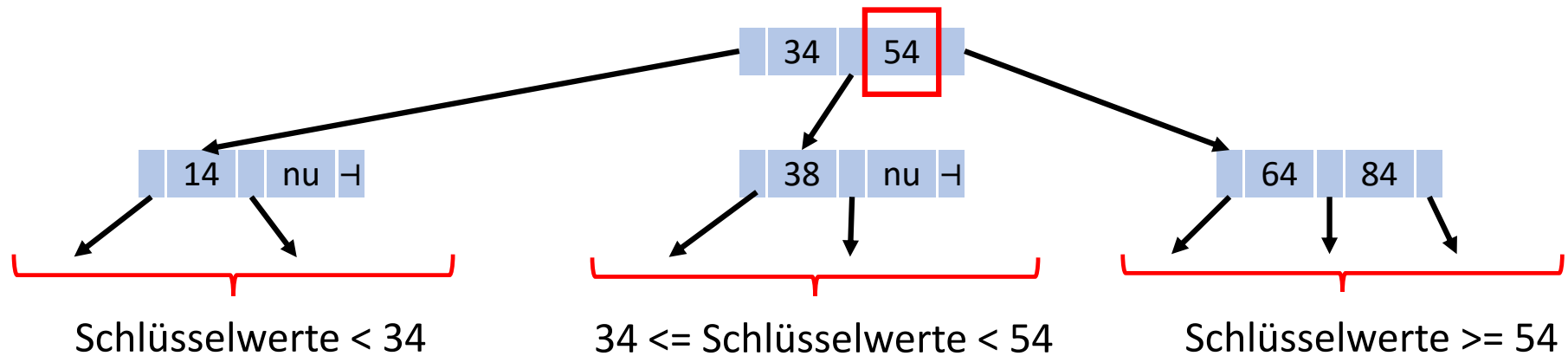
B⁺-Baum Eigenschaften



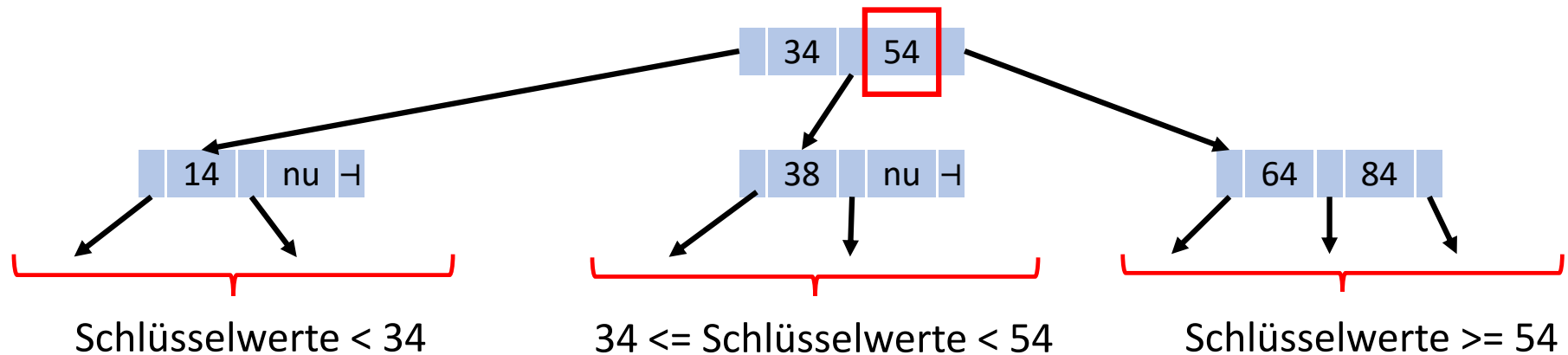
Ordnung

- Die Werte in jedem Knoten sind geordnet, d.h. $k_i < k_j$, wenn $i < j$
- Teilbäume sind geordnet

B⁺-Baum Eigenschaften



Balanciert, d.h. alle Pfade von der Wurzel zu den Blattknoten haben die gleiche Länge

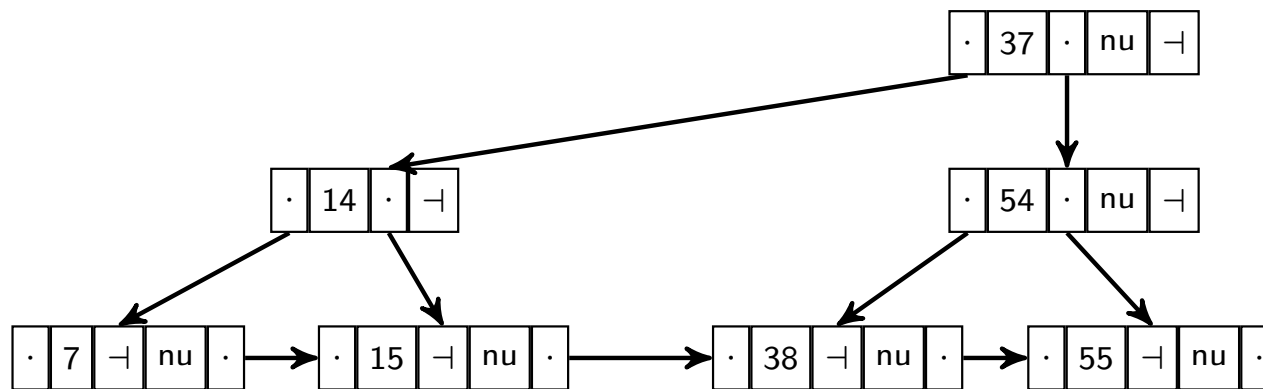
B⁺-Baum Eigenschaften

Verzweigung: Jeder Knoten hat zwischen $\lceil \frac{n}{2} \rceil$ und n Kinder

- Ausnahme: der Wurzelknoten hat zwischen 2 und n Kinder
- Blattknoten haben zwischen $\lceil \frac{n-1}{2} \rceil$ und $n - 1$ Pointer auf Tupel in Dateien und 1 Pointer auf den nächsten Blattknoten

Ein minimaler B^+ -Baum

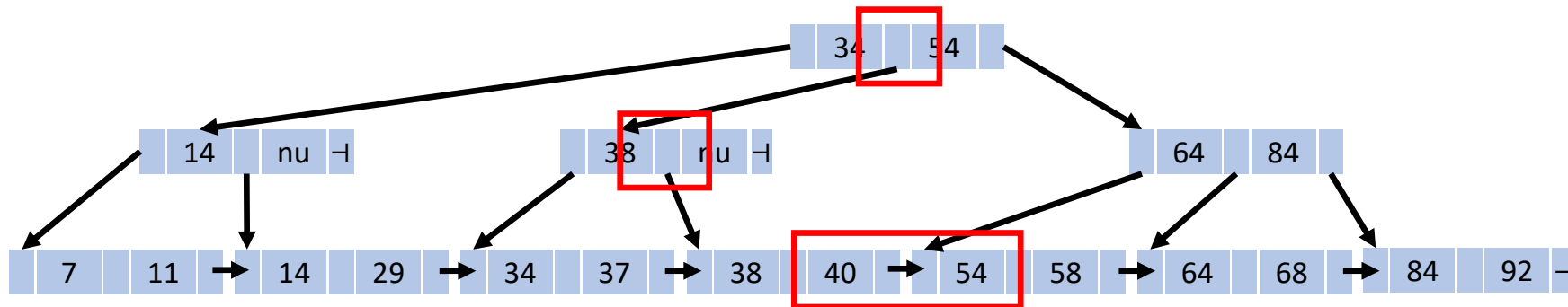
Ein minimal gefüllter B^+ -Baum für $n=3$



Verwendung von B^+ -Bäumen in DBMS

- Jeder Knoten hat die Größe eines I/O Blocks
- Ein Knoten ist zu mindestens 50% gefüllt
- Ein B^+ Baum ist i.d.R. sehr flach, d.h. Suche verursacht nur wenige wahlfreie Zugriffe
- Zudem, die ersten 1-2 Ebenen des Baums sind i.d.R. im Hauptspeicher “cached”
- “Logisch” nahe bedeutet nicht unbedingt “physisch nahe”
Das Lesen eines Knotens erfordert typischerweise einen I/O-Zugriff
- Innere Knoten entsprechen einer Hierarchie von sparse Indexes

Uniqueness-Constraints auf Attributen in einer Datenbank werden durch B^+ -Bäume realisiert → Primärschlüssel

B⁺-Baum Suche (Lookup)

n = 3

- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden
- Suche nach Bereich: 40-55 → gefunden

B⁺-Baum Einfügen

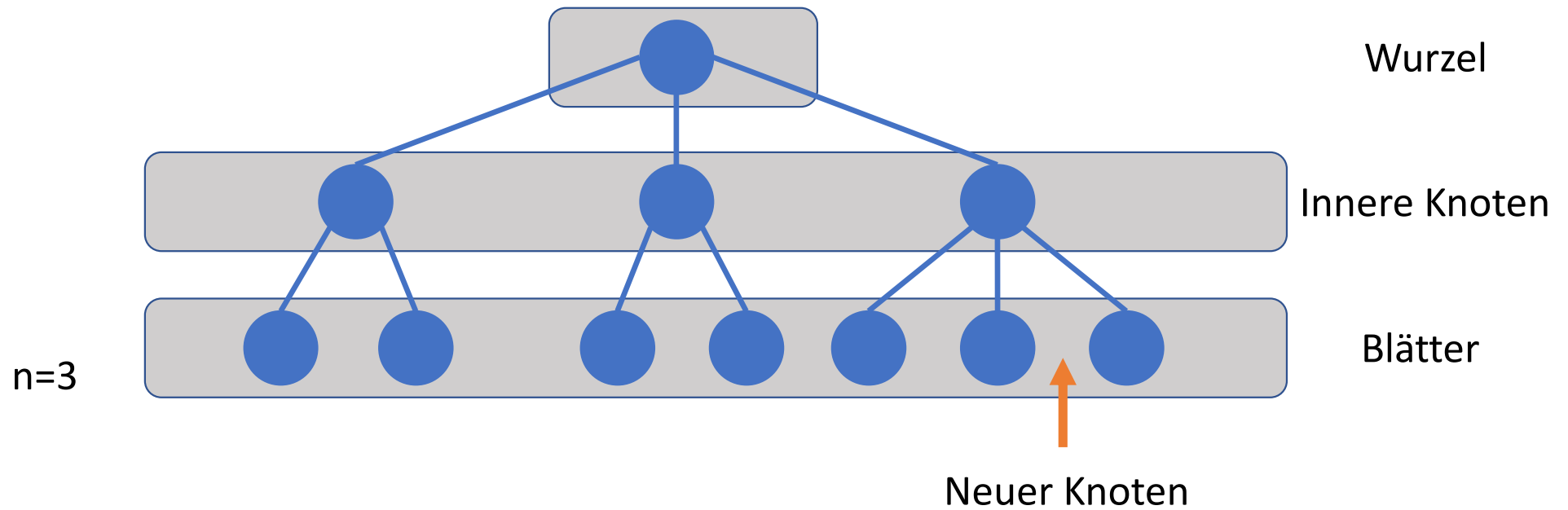
Einfügen von Schlüsselwert k

- Suche: Finde das Blatt b, das den Schlüsselwert k enthalten müsste
- Falls b genügend Platz hat, füge k ein
- Sonst spalte Knoten b, teile Schlüssel unter den zwei Knoten auf, ändere den Eintrag im Elternknoten

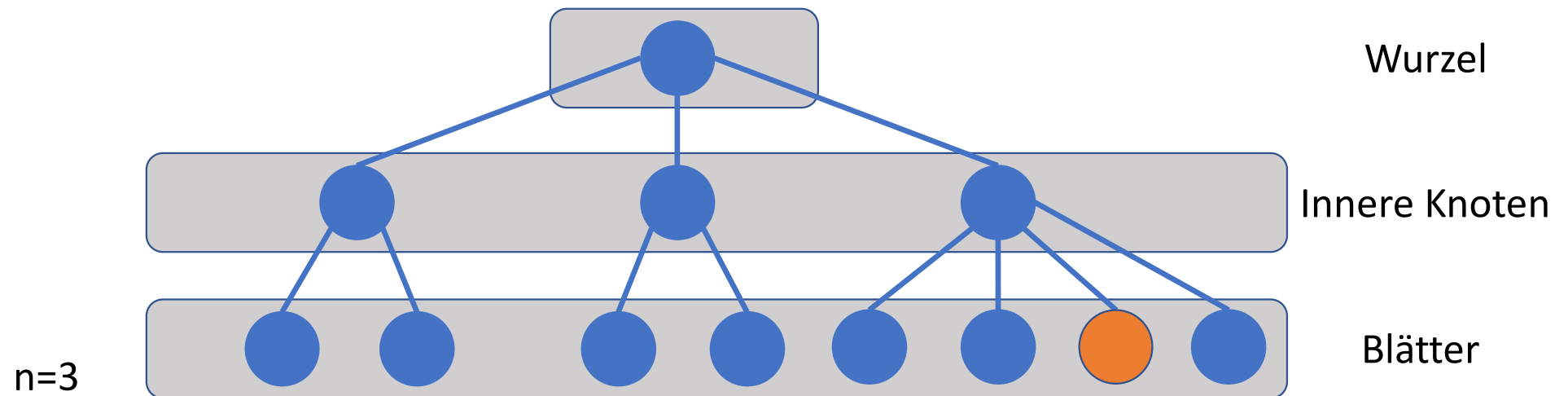
Spalten von Knoten muss unter Umständen rekursiv nach oben erfolgen

- Wenn die Wurzel gespalten wird, erhöht sich die Tiefe des Baumes um eine Ebene

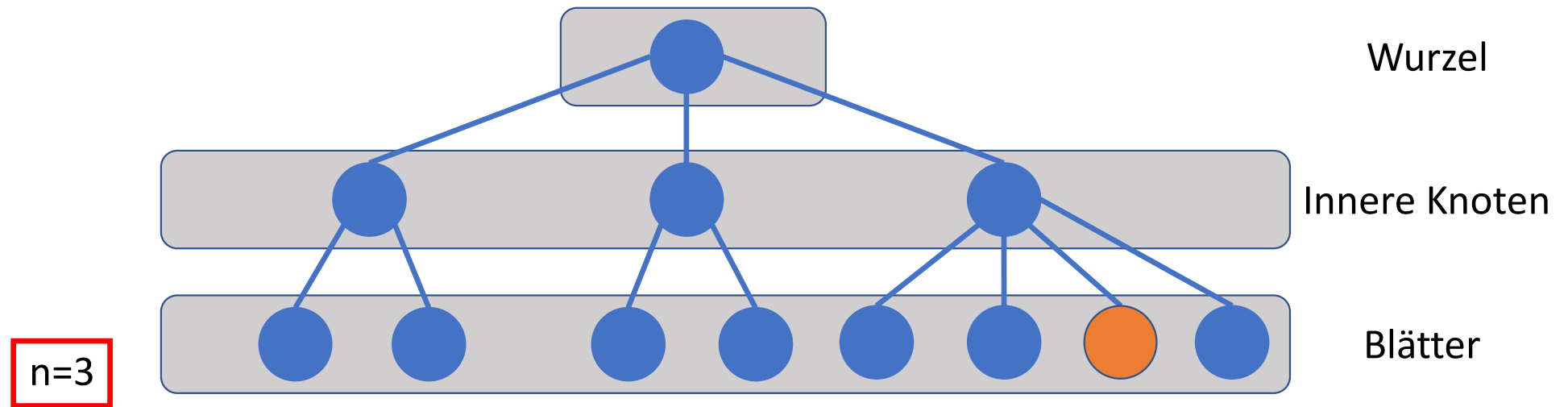
B⁺-Baum Spalten von Knoten



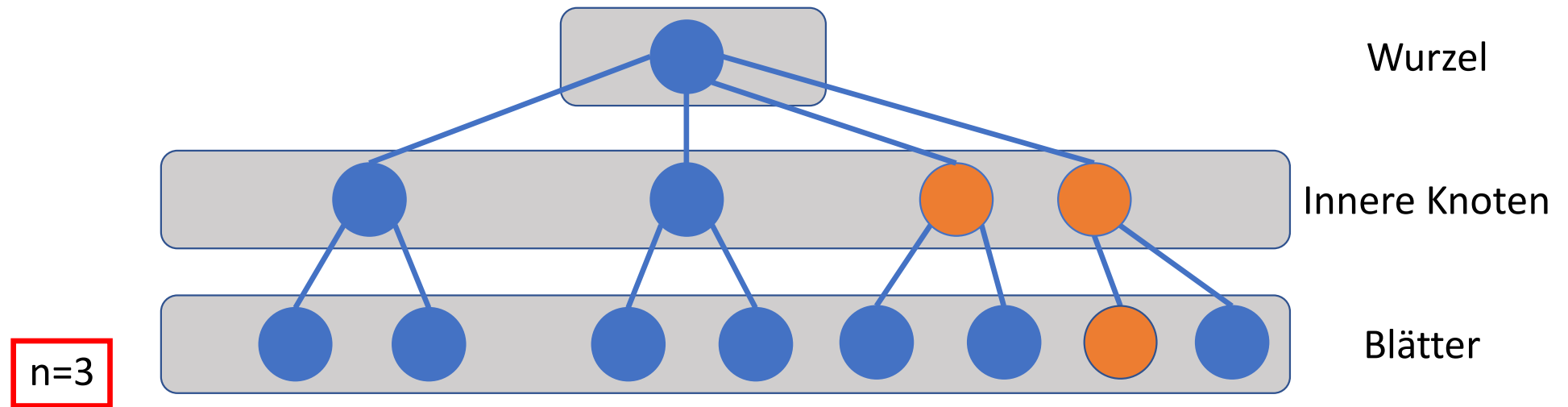
B^+ -Baum Spalten von Knoten



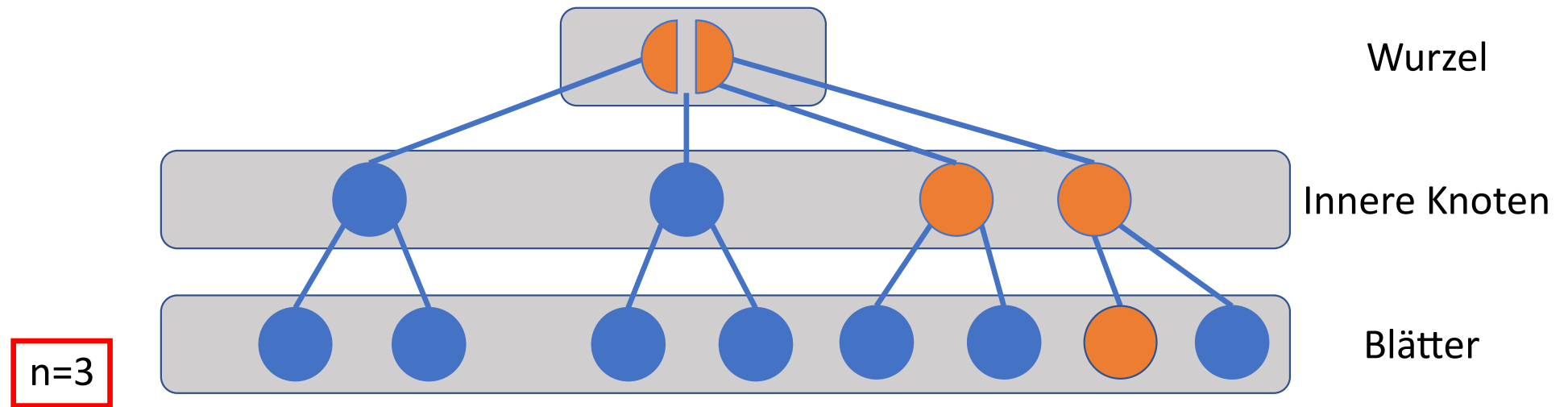
B⁺-Baum Spalten von Knoten



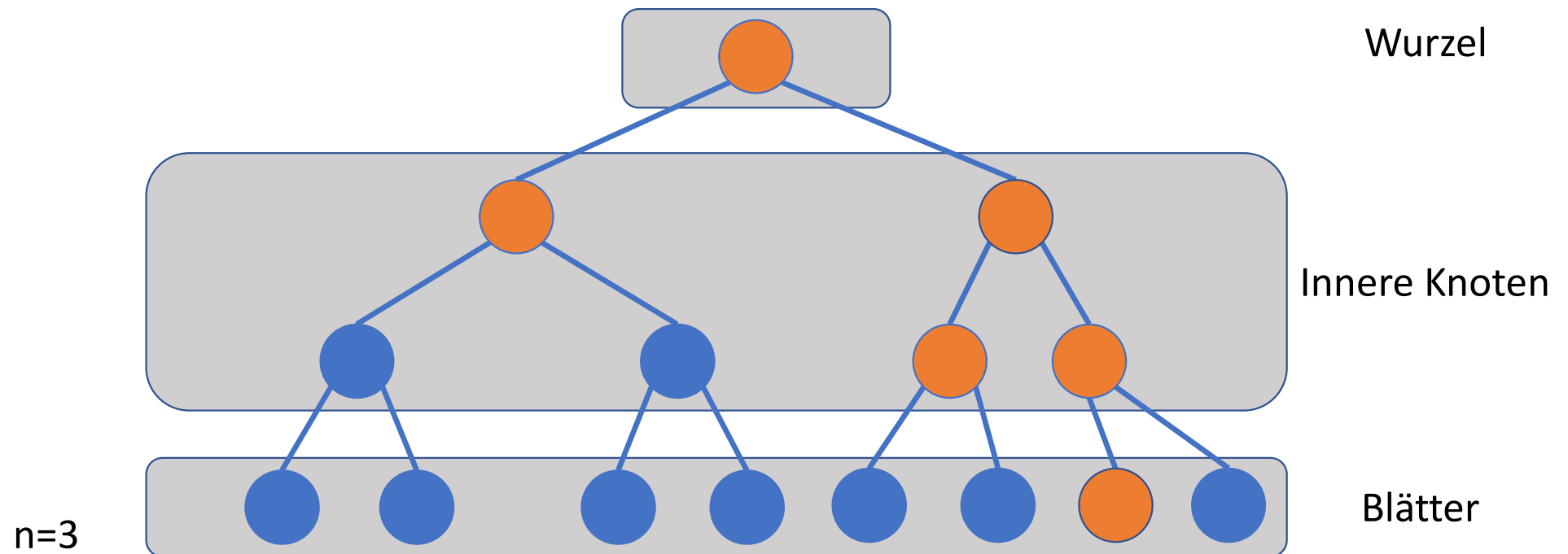
B⁺-Baum Spalten von Knoten



B⁺-Baum Spalten von Knoten

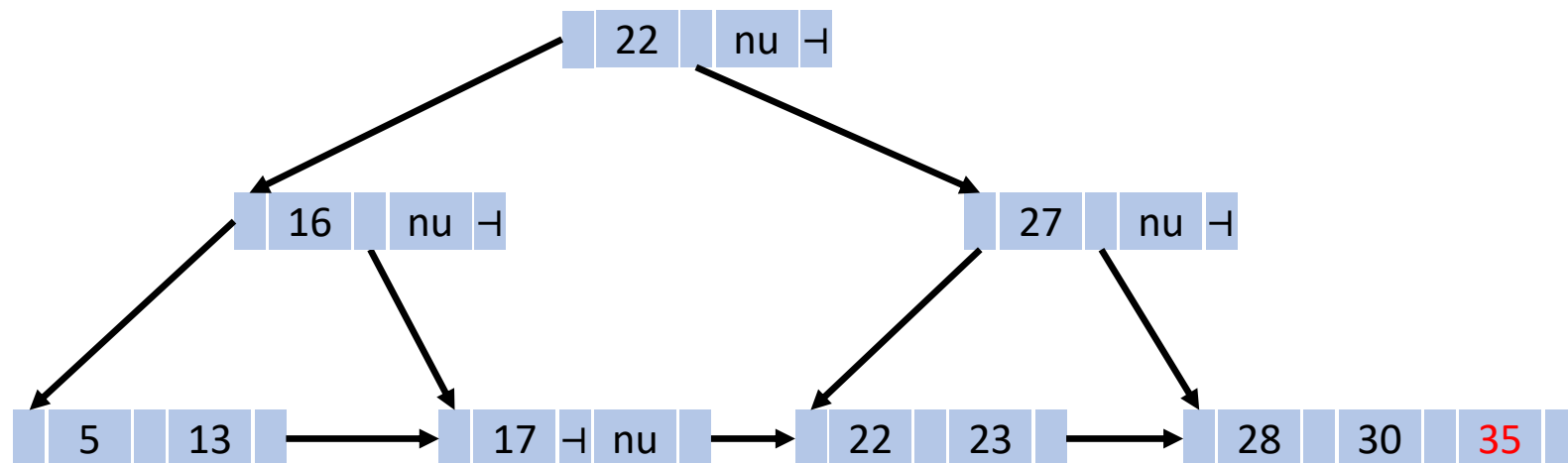


B⁺-Baum Spalten von Knoten



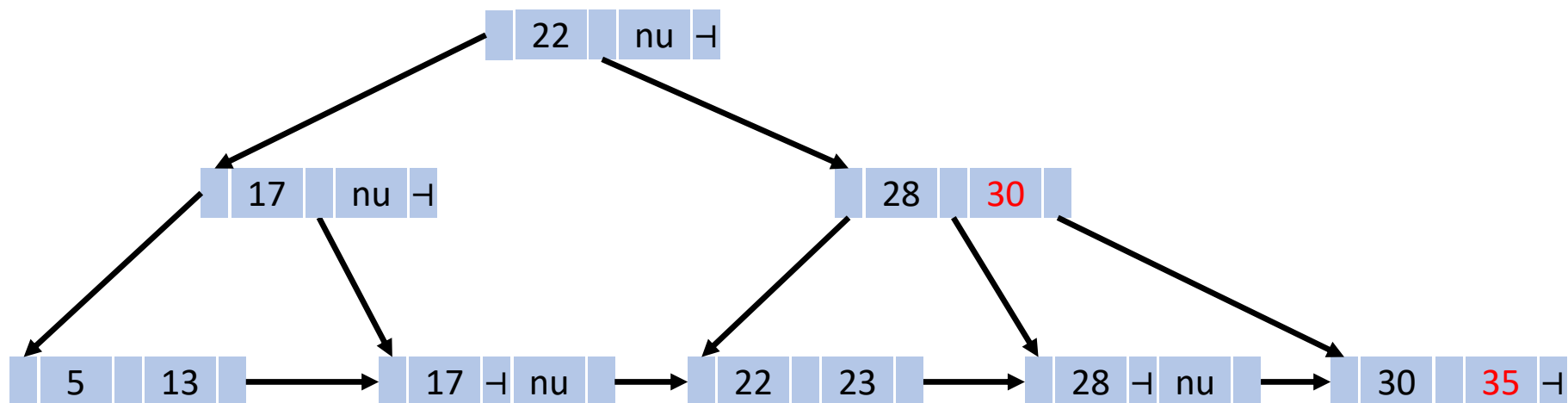
B⁺-Baum Einfügen

n=3, Einfügen von 35



B⁺-Baum Einfügen

n=3, Einfügen von 35



B⁺-Baum Löschen

Löschen von Schlüsselwert k

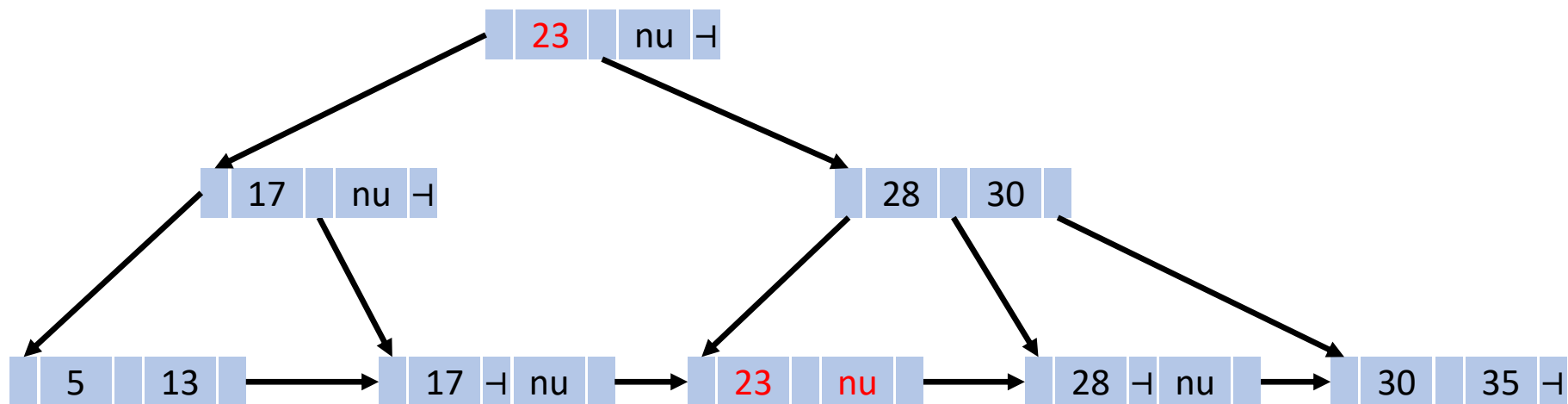
- Suche: Finde das Blatt b, das den Schlüsselwert k enthält
- Falls b genügend gefüllt bleibt min. $\lceil \frac{n}{2} \rceil$, lösche k
Innere Knoten können dann Schlüssel enthalten, die nicht mehr in den Blättern existieren
- Sonst verschmelze Knoten
 - Falls Verschmelzen mit Nachbarknoten möglich, verschmelzen und Pointer im Elternknoten anpassen
 - Falls Verschmelzen nicht möglich, Neuverteilung der Pointer über Elternknoten

Verschmelzen muss unter Umständen auch in höheren Ebenen erfolgen

- Die Tiefe des Baumes kann sich um eine Ebene verringern

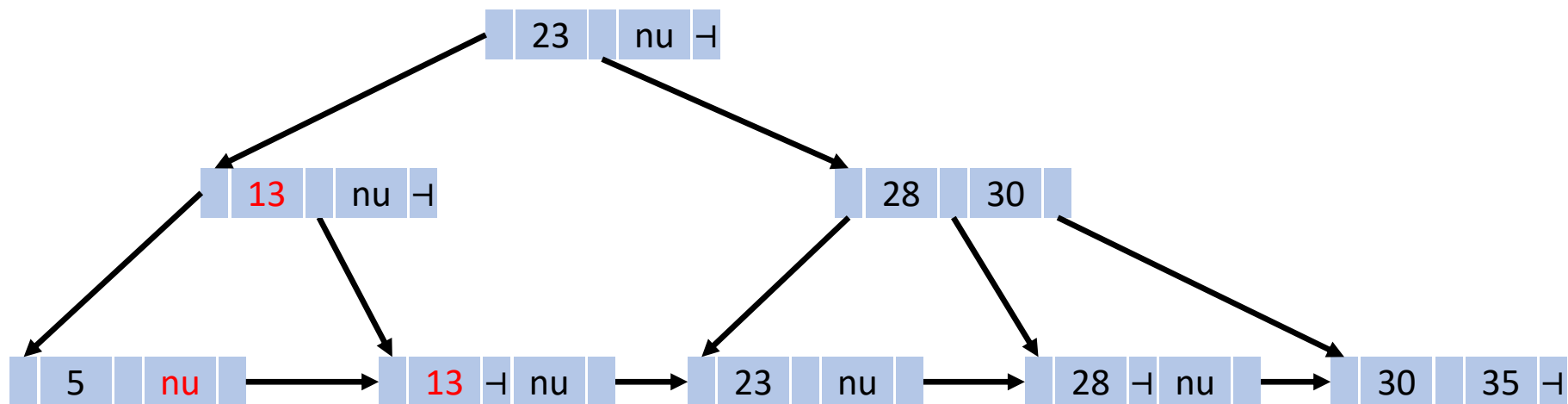
B⁺-Baum Löschen

n=3, Löschen von 22



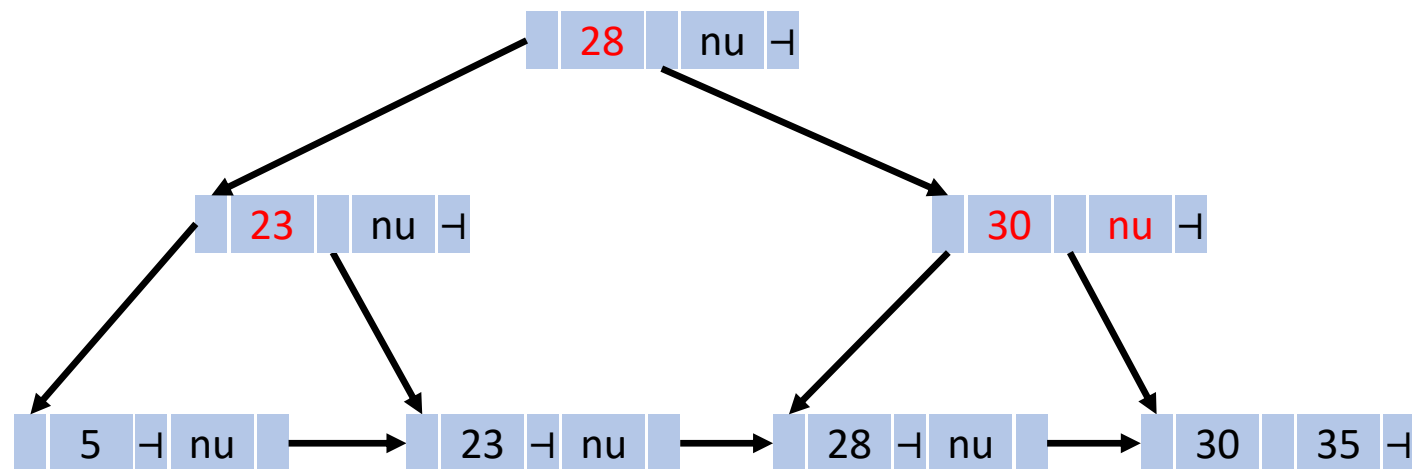
B⁺-Baum Löschen

n=3, Löschen von 17



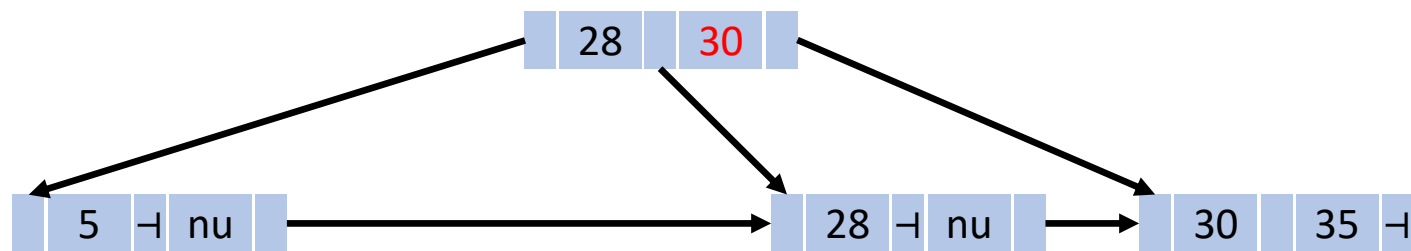
B⁺-Baum Löschen

n=3, Löschen von 13



B⁺-Baum Löschen

n=3, Löschen von 23



B⁺-Bäume online

Interaktive Webapplikationen

- <http://goneill.co.nz/btree-demo.php>
- <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>
- ...

Disclaimer

Es gibt verschiedene Implementierungen mit eventuell auch Fehlern

2 Indexstrukturen

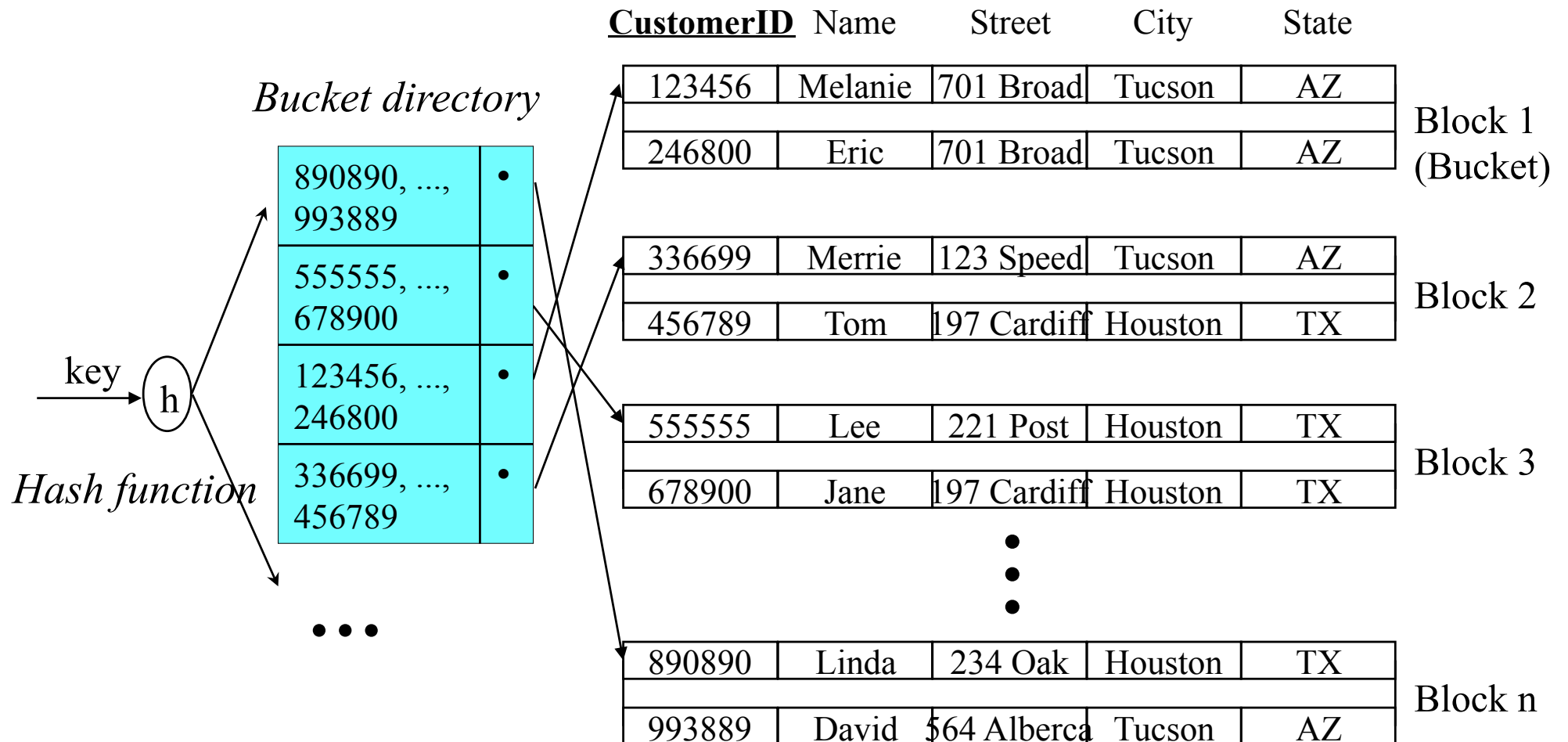
- Geordnete Indexe
- B⁺-Bäume
- Hashing

Statischer Hash Index

Erstellen eines Indexes auf Basis einer Hashfunktion
anstatt auf Basis eines Search Keys

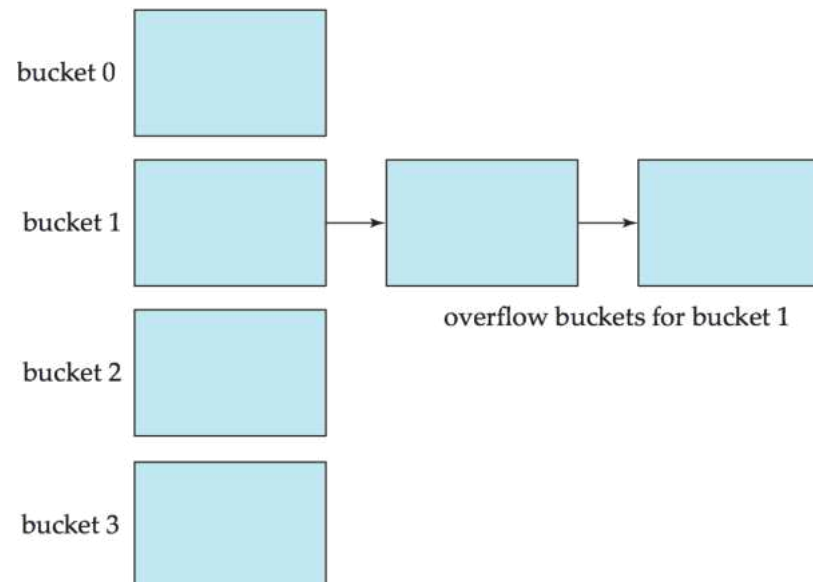
- Wahl einer geeigneten **Hashfunktion** h
- Hashfunktion h auf Search Key Wert k anwenden $h(k)$
- Reserviere ein **Bucket** (Block/Seite) für jeden Wert von $h(k)$

Statischer Hash Index



Statischer Hash Index

- Suche
 - Ein Zugriff auf das Bucket Directory
 - Ein Zugriff auf die Datei
- Gute Performanz hängt von einer guten Hashfunktion ab
- Bucket kann überfüllt sein
 - Zu viele Tupel werden auf das gleiche Bucket abgebildet
 - Lösung: Overflow Buckets und Overflow Chains



Statischer Hash Index

- Suche
 - Ein Zugriff auf das Bucket Directory
 - Ein Zugriff auf die Datei
- Gute Performanz hängt von einer guten Hashfunktion ab
- Bucket kann überfüllt sein
 - Zu viele Tupel werden auf das gleiche Bucket abgebildet
 - Lösung: Overflow Buckets und Overflow Chains

Open vs. Closed Hashing

- Open Hashing
 - Overflow Chains mit weiteren Overflow Buckets
- Closed Hashing
 - Feste Anzahl von Buckets
 - Beim Überlaufen muss eines der existierenden Buckets verwendet werden (Linear Probing, weitere Hashfunktionen, etc.)

Static Hashing

Problem bei Static Hashing:
Hashfunktion und Anzahl der Buckets muss bereits beim Erstellen bestimmt werden

Datenbanken wachsen und schrumpfen mit der Zeit!

- Initiale Bucketanzahl zu gering
→ Viele Overflows, Performanz leidet
- Initiale Bucketanzahl zu groß
→ Underflow, Speicherplatz wird verschwendet

Lösungsansätze

- Periodische Reorganisation
- Dynamic Hashing:
ermöglicht Modifikationen zu einem späteren Zeitpunkt

3 Design Tuning

Design Tuning

Optimierungsgegenstände

- Clustering Index vs. Hashing
- Sparse vs. dense Index
- Clustering vs. non-clustering Index
- Beschleunigung von Joins durch Indexe

Grundsätzliche Fragestellungen

- Sind die Kosten für eine periodische Reorganisation akzeptabel?
- Wie viele Updates gibt es wirklich?
- Optimierungsziel: durchschnittl. Laufzeit oder Worst-Case-Optimierung?
- Welche Arten von Anfragen werden erwartet?

Nutzung von Indexen

Manchmal werden existierende Indexe nicht verwendet

Wann und warum?

Nutzung von Indexen

Manchmal werden existierende Indexe nicht verwendet

- System-Catalog hat veraltete Informationen
Optimierer könnte annehmen, dass die Tabelle klein ist
- „Gute“ und „schlechte“ Typen von Anfragen
 - `SELECT * FROM EMP WHERE salary/12 > 4000`
 - `SELECT * FROM EMP WHERE salary > 48000`
 - `SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G'`
 - `SELECT * FROM EMP WHERE name LIKE 'G%'`
 - `SELECT * FROM EMP WHERE name = 'Smith'`
 - `SELECT * FROM EMP WHERE salary IS NULL`
- Geschachtelte SQL-Anfragen
- Negationen
- Anfragen mit OR

Zusammenfassung

- Speicherhierarchie und Festplatten
- Tupel mit variabler Länge machen die Dateiorganisation komplexer
- Keine Dateiorganisation ist die beste für alle Anwendungen
- Clustering vs. non-clustering Indexe
- Sparse vs. dense Indexe
- Single-level and multi-level Indexe
- Der B^+ -Baum ist der wichtigste Index für Datenbanksysteme
- Tuning hängt von vielen Aspekten ab, z.B. Query Load, Charakteristika der Daten, Systemvoraussetzungen etc.

Zusammenfassung

Kompromiss zwischen Speicherplatz und Zeit

Intelligente Nutzung von zusätzlichem Speicherplatz erhöht im Allgemeinen die Performanz.

Kompromiss zwischen Select-Anfragen und Updates

Wenn Maßnahmen zur Beschleunigung von Select-Anfragen getroffen werden, geschieht das meist zum Nachteil von Updates – und umgekehrt.