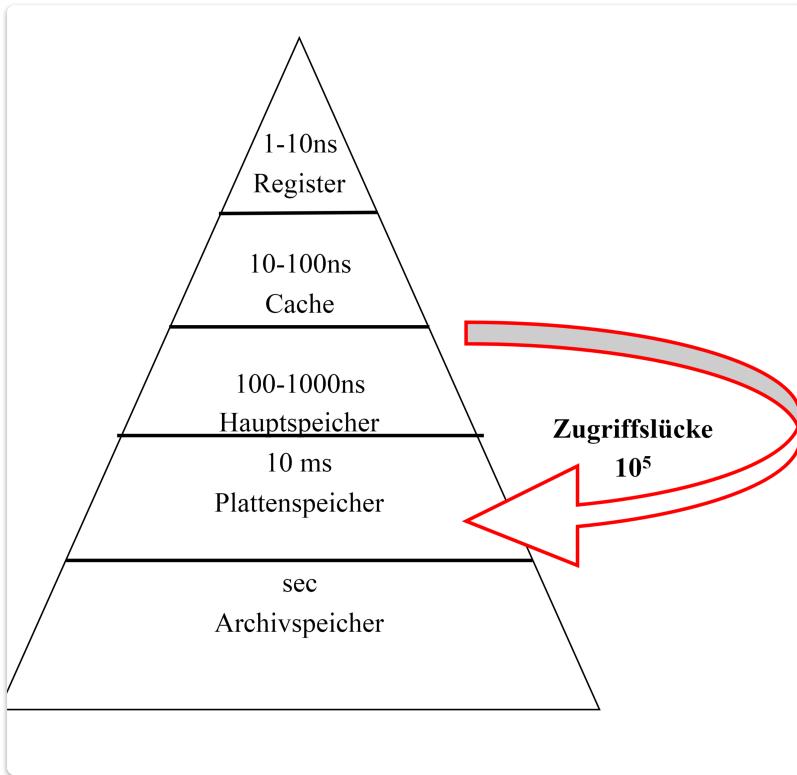


6. Physischer Datenbankentwurf

Slides: [DBS-8_Physischer Datenbankentwurf.pdf](#)

Dateiorganisation - Speicherorganisation

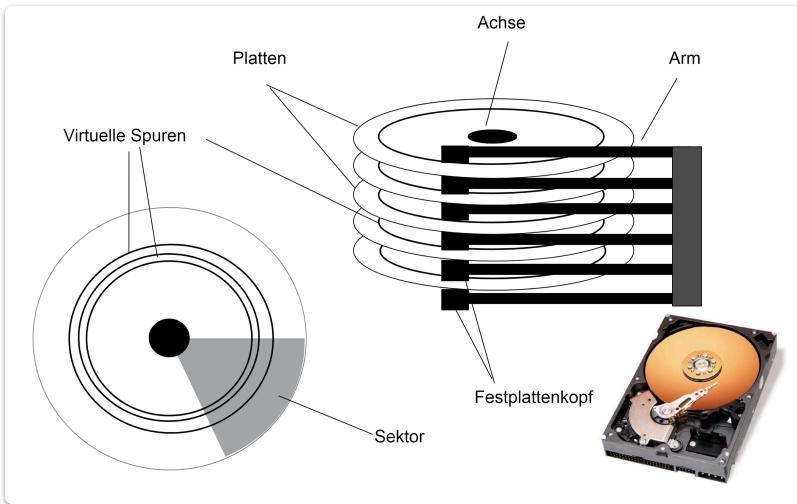
Speicherebenen



- **Primärspeicher (Volatile Storage)**
 - z.B. Cache, Hauptspeicher, Register
 - Verliert den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeiten:
 - Register: 1-10 ns
 - Cache: 10-100 ns
 - Hauptspeicher: 100-1000 ns
- **Sekundärspeicher (Non-Volatile Storage)**
 - z.B. Festplatte
 - Behält den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeit: ca. 10 ms (deutlich langsamer als Primärspeicher)
- **Tertiärspeicher (Non-Volatile Storage)**
 - z.B. Magnetbänder (Archivspeicher)
 - Behält den Inhalt, wenn der Strom abgeschaltet wird
 - Zugriffszeit: sec (noch langsamer als Sekundärspeicher)

- Je höher das Level (im Mehrebenenmodell), desto schneller der Zugriff.
- **Zugriffslücke:** Die Diskrepanz in der Zugriffszeit zwischen Hauptspeicher und Plattspeicher ist sehr groß (10^5).

Magnetische Festplatte



- Besteht aus:
 - Platten
 - Spuren (virtuell, konzentrisch auf den Platten)
 - Sektoren (segmentieren die Spuren)
 - Achse (um die sich die Platten drehen)
 - Arm mit Festplattenkopf (zum Lesen und Schreiben)
- Meist sind Datenbanken auf magnetischen Festplatten gespeichert.
- Datenbank ist oft zu groß für den Hauptspeicher.
- Plattspeicher ist persistent (nicht-flüchtig).
- Plattspeicher ist billiger als Hauptspeicher.

Zugriffsoptimierung für Festplatten

Für jeden Speicherzugriff auf die Festplatte fallen folgende Zeiten an:

- **Seek Time:** Zeit, um den Lese-/Schreibkopf zur richtigen Spur zu bewegen.
- **Rotational Delay (Latenzzeit):** Zeit, bis der gewünschte Sektor unter dem Lese-/Schreibkopf rotiert ist.
- **Transfer Time:** Zeit, um die Daten tatsächlich zu übertragen.
- Jede **Spur (Track)** ist unterteilt in **Sektoren**.
- Eine **Seite (Block/Page)** ist eine kontinuierliche Sequenz von **Sektoren** eines einzigen Tracks.
- Die kleinste Einheit, die zwischen Festplatte und Hauptspeicher transferiert wird, ist eine Seite (Block/Page).

Optimierung des Festplattenzugriffs

- **Blöcke in der Reihenfolge anordnen**, in der sie auch benötigt werden (sequentielle Anordnung minimiert Seek Time und Rotational Delay).
- **Verwandte Informationen nahe beieinander ablegen** (reduziert die Notwendigkeit großer Kopfbewegungen).

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

Funktionale Anforderungen an Datenbanksysteme

- **Tupel (Records) sequenziell abarbeiten** können.
- **Effiziente Key-Value-Suche ermöglichen**.
- **Einfügen/Löschen von Tupeln (Records)** unterstützen.

Performanzanforderungen an Datenbanksysteme

- **Wenig Speicherplatz verschwenden**.
 - **Schnelle Antwortzeiten** gewährleisten.
 - **Hoher Durchsatz von Transaktionen** ermöglichen.
-

Dateiorganisation - 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 (die Einheit für den Datentransfer zwischen Festplatte und Hauptspeicher).*

Größe eines Records

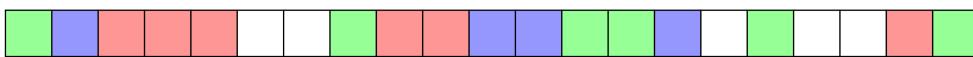
- **Feste Größe (Fixed Size):** Alle Records in einer Datei haben die gleiche Länge. Dies erleichtert die Berechnung der Speicheradresse eines bestimmten Records.
- **Variable Größe (Variable Size):** Records in einer Datei können unterschiedliche Längen haben. Dies erfordert zusätzliche Informationen, um die Grenzen zwischen Records zu identifizieren (z.B. Längenangaben).

Dateien auf Massenspeichern (normalerweise Festplatten)

- Ermöglichen schnellen Zugriff auf beliebige Tupel (im Vergleich zu sequentiellen Speichermedien wie Bändern).
- Sind Teil des Sekundärspeichers.

Beispiel

Blöcke in einem Dateisystem sind nicht unbedingt zusammenhängend (contiguous).



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 (Fixed Size Records)

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

Löschen eines Tupels i

- Es gibt verschiedene Strategien, um die durch das Löschen entstandene Lücke zu schließen

Verschieben von Tupeln:

- Verschiebe die nachfolgenden Tupel ($i+1, \dots, n$) an die Positionen $i, \dots, n-1$, um die Lücke zu schließen. Dies ist aufwendig, da viele Datensätze bewegt werden müssen.
- Oder verschiebe das letzte Tupel (n) an die Position i . Dies ist effizienter, hinterlässt aber möglicherweise eine Lücke am Ende.

Markieren der Lücken:

- Markiere die Lücke als gelöscht und fülle sie später mit neuen Tupeln auf. * **Free-List:** Eine Möglichkeit, die freien Lücken zu verwalten:
 - Markiere die erste Lücke im File-Header, um den Beginn der Liste freier Blöcke zu kennzeichnen.
 - Benutze die Lücken selbst, um auf weitere Lücken zu verweisen (verkettete Liste freier Speicherbereiche).

Variable Größe (Variable Size Records)

- Tupel haben **unterschiedliche Größen** und beanspruchen unterschiedlich viel Speicherplatz.
- Beispiel:** Attribute mit variabler Länge wie `varchar`.

Alternativen für variable Größe

- Wenn maximale Größe bekannt:** Abbildung auf Tupel fester Größe (führt möglicherweise zu Speicherplatzverschwendungen, wenn die tatsächliche Größe oft kleiner ist).
- Slotted-Page-Structure:** Eine effizientere Methode zur Speicherung von Tupeln variabler Größe innerhalb eines Blocks (Seite):
 - Tupel werden fortlaufend im Datenbereich des Blocks gespeichert.
 - Ein **Block Header** enthält **Pointer (Zeiger)** zu allen Tupeln innerhalb des Blocks sowie Informationen über den freien Speicherplatz.
 - Beim Zugriff auf ein Tupel wird der Pointer im Header verwendet, um die tatsächliche Position und Größe des Tupels im Datenbereich zu finden.
 - Dies ermöglicht effiziente Updates und Löschungen, da nur die Pointer im Header angepasst werden müssen, ohne die Tupel selbst verschieben zu müssen (innerhalb

des Blocks).

Organisation von Tupeln in Dateien

Bestimmen der Tupelreihenfolge innerhalb der Datei: Wie werden die Datensätze physisch in der Datei angeordnet?

Heap Files

- Tupel können an **beliebiger Position** in der Datei abgelegt werden.
- Es gibt keine spezielle Ordnung. Neue Tupel werden einfach am Ende der Datei angehängt oder in freie Lücken eingefügt.
- Suche nach einem bestimmten Tupel erfordert möglicherweise das Durchsuchen der gesamten Datei.

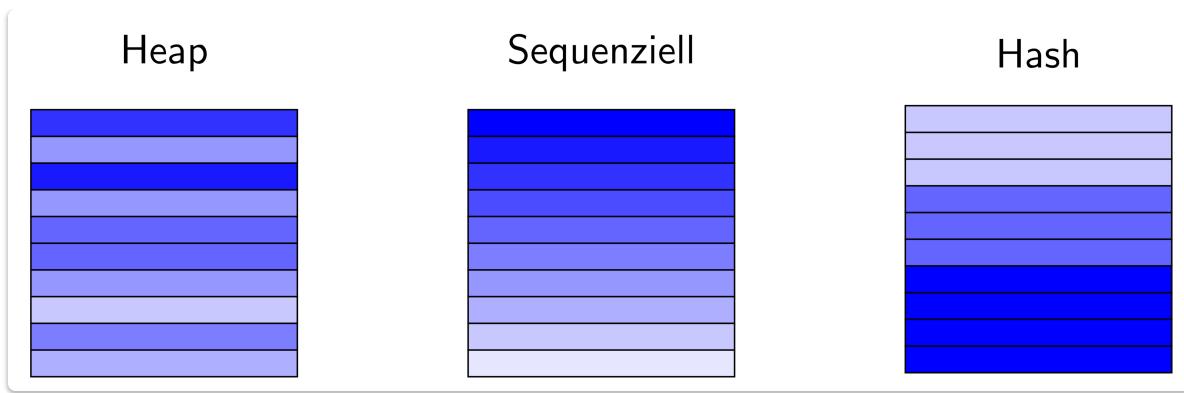
Sequentielle Dateiorganisation

- Tupel werden **sequenziell in Reihenfolge des Search Keys** (z.B. ein bestimmtes Attribut) abgelegt.
- Die physische Reihenfolge der Datensätze entspricht der logischen Ordnung basierend auf dem Suchschlüssel.
- Dies ermöglicht effizientere Bereichsabfragen basierend auf dem Suchschlüssel.
- Einfügungen und Löschungen können aufwendig sein, da möglicherweise viele Datensätze verschoben werden müssen, um die Reihenfolge beizubehalten.

Hash Files

- Benutze eine **Hashfunktion**, um die **Position eines Tupels innerhalb der Datei zu bestimmen**.
- Der Wert des Suchschlüssels wird durch die Hashfunktion abgebildet, um die Speicheradresse des Tupels zu ermitteln (ungefähre Position).
- Ermöglicht sehr schnelle Zugriffe auf einzelne Tupel basierend auf dem Suchschlüssel (im Idealfall konstanter Zeitaufwand).
- Bereichsabfragen sind in der Regel ineffizient, da die gehaschten Adressen keine direkte Ordnung widerspiegeln.

Overview



Heap

- **Keine bestimmte Reihenfolge** innerhalb der Tabelle. Die Tupel sind in keiner spezifischen Ordnung gespeichert.
- **Suche:** Alle Tupel nacheinander durchsuchen (**Linear Scan**), bis das gesuchte Tupel gefunden wird. Dies kann ineffizient sein, insbesondere bei großen Tabellen.
- **Einfügen:** Finde einen freien Slot (entweder am Ende der Datei oder eine zuvor freigegebene Lücke) und füge das neue Tupel dort ein. Das Einfügen ist in der Regel effizient.

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

Sequenziell

- Tabelle ist anhand eines **Search-Keys (Suchschlüssels)** sortiert, z.B. ID-Attribut. Die physische Reihenfolge der Tupel entspricht der Sortierreihenfolge des Suchschlüssels.
- Der Search-Key muss **nicht unbedingt der Primärschlüssel** sein, ist aber meistens ein eindeutiges Attribut.
- **Suche: Binäre Suche** bei Suche anhand des Search-Keys möglich, was deutlich effizienter ist als der Linear Scan bei Heap Files (logarithmische Komplexität).
- **Einfügen:** Erfordert möglicherweise eine **Reorganisation der Datei**, um die Sortierreihenfolge beizubehalten. Neue Tupel müssen an der richtigen Position eingefügt werden, was das Verschieben anderer Tupel erfordern kann.

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

Hashing

- **Hashfunktion bestimmt Reihenfolge:** Eine Hashfunktion wird auf den Search-Key angewendet, um die Speicheradresse (genauer gesagt, die Bucket- oder Blocknummer) für das Tupel zu bestimmen. Tupel mit ähnlichen Hashwerten werden im selben Bucket gespeichert.
- **Suche:** Benutze die Hashfunktion mit dem Wert des Search-Keys (z.B. ID-Attribut), um den **richtigen Block/Seite direkt zu finden**. Im Idealfall ist ein direkter Zugriff auf den Speicherort möglich.
- **Einfügen:** Benutze die Hashfunktion mit dem Wert des Search-Keys, um den **richtigen Block zu finden und füge das Tupel dort hinzu**. Es kann zu Kollisionen kommen, wenn unterschiedliche Suchschlüssel auf denselben Bucket abgebildet werden. Diese müssen durch spezielle Techniken (z.B. separate Verkettung) behandelt werden.

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

Indexstrukturen

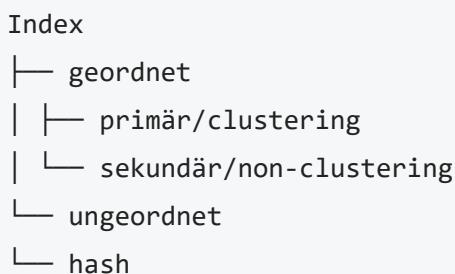
Annahmen (im Kontext von Indexstrukturen)

- 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 (Indexe müssen dynamisch verwaltet werden).

Ziel: so wenig Daten wie möglich lesen, um die Performance von Datenbankabfragen zu optimieren.

Übersicht: Indexstrukturen

Klassifikation von Indexen:



- **Geordnet:** Der Index basiert auf einer sortierten Struktur, die effiziente Bereichsabfragen ermöglicht.
 - **Primär/Clustering Index:** Die physische Speicherung der Datensätze auf der Festplatte ist an die Sortierreihenfolge des Indexschlüssels angepasst. Es kann nur einen Clustering Index pro Tabelle geben.
 - **Sekundär/Non-Clustering Index:** Der Index ist eine separate Struktur, die Pointer zu den tatsächlichen Datensätzen enthält. Die physische Speicherung der Daten ist unabhängig von der Indexreihenfolge. Es können mehrere Non-Clustering Indexe pro Tabelle existieren.
- **Ungeordnet:** Der Index basiert auf einer Hash-Struktur, die schnelle Punktabfragen ermöglicht, aber ineffizient für Bereichsabfragen ist.
 - **Hash Index:** Verwendet eine Hashfunktion, um die Speicheradresse der Datensätze zu finden (ähnlich wie bei Hash Files).

Variationen:

- **Single-Level Index:** Der Index besteht aus einer einzigen Ebene.

- **Multi-Level Index:** Der Index ist hierarchisch aufgebaut (z.B. B-Baum), um die Suche in sehr großen Datenbeständen zu beschleunigen.

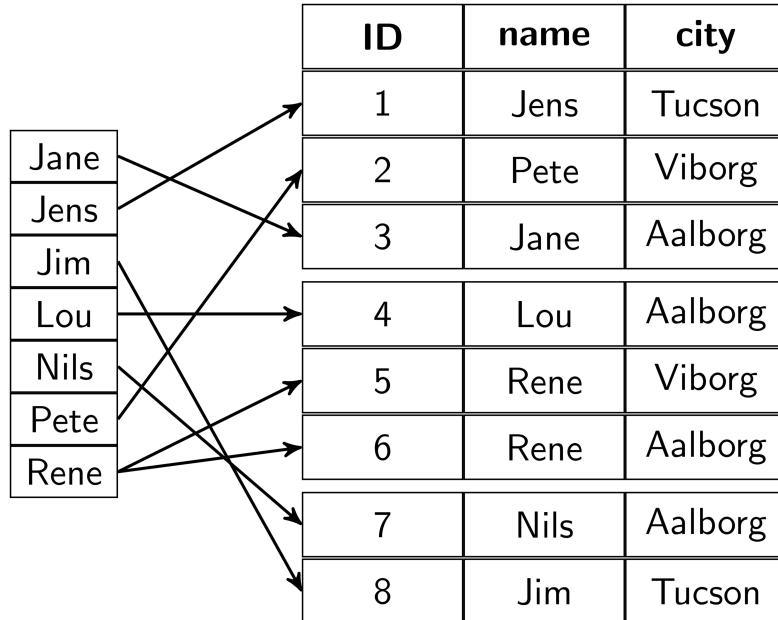
Es ist möglich, **mehrere Indexe auf der gleichen Tabelle zu definieren**, um unterschiedliche Arten von Abfragen effizient zu unterstützen. Die Wahl der Indexe hängt von den häufigsten Abfragemustern ab.

Primary Sparse Index

| ID | name | city |
|----|------|---------|
| 1 | Jens | Tucson |
| 2 | Pete | Viborg |
| 3 | Jane | Aalborg |
| 4 | Lou | Aalborg |
| 5 | Rene | Viborg |
| 6 | Rene | Aalborg |
| 7 | Nils | Aalborg |
| 8 | Jim | Tucson |

- 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

Indexstrukturen - Geordnete Indexe

Übersicht: Primär vs. Sekundär & Dense vs. Sparse Index

Primär (Primary) vs. Sekundär (Secondary)

- = Clustering vs. Non-Clustering (synonyme Begriffe)
- Entscheidende Frage: Ist die Datei nach dem Search-Key sortiert?
 - Ja: \Rightarrow Primär (Clustering) Index. Die physische Anordnung der Daten auf der Festplatte entspricht der Sortierreihenfolge des Index-Schlüssels.
 - Nein: \Rightarrow Sekundär (Non-Clustering) Index. Der Index ist eine separate Struktur, die Pointer zu den Datensätzen enthält, welche in einer anderen (oder keiner bestimmten) Reihenfolge auf der Festplatte gespeichert sind.

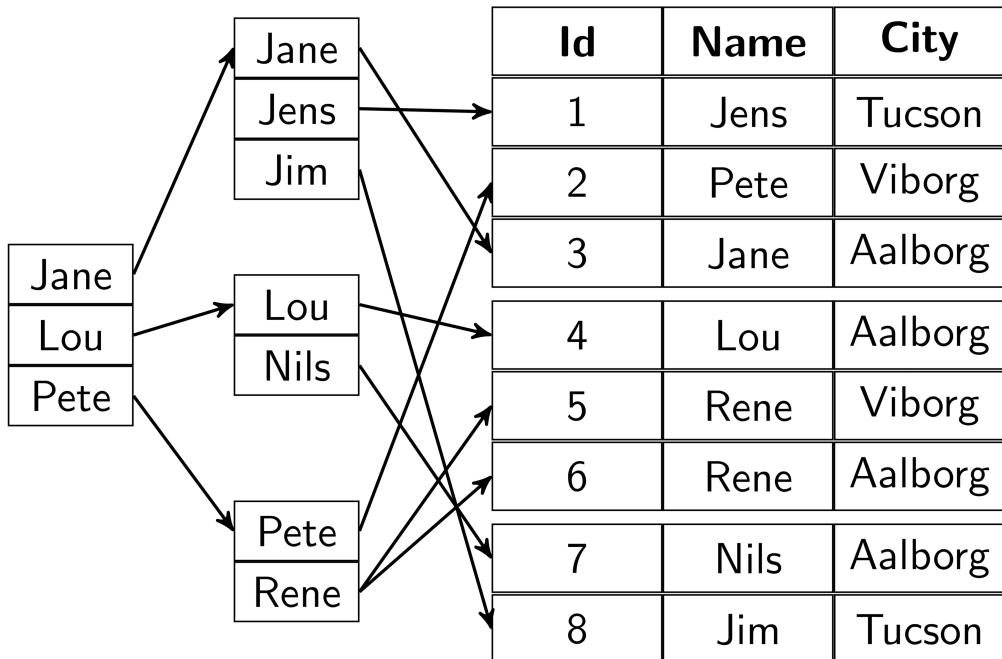
Dense vs. Sparse

- Entscheidende Frage: Gibt es einen separaten Eintrag im Index für jedes Tupel bzw. jeden vorkommenden Wert des Search-Keys?
 - Ja: \Rightarrow Dense Index (dichter Index). Jeder Datensatz (oder jeder eindeutige Wert des Suchschlüssels) hat einen Eintrag im Index, der auf den tatsächlichen Datensatz zeigt.
 - Nein: \Rightarrow Sparse Index (dünner Index). Nur für einige Werte des Suchschlüssels existiert ein Eintrag im Index. Diese Einträge zeigen auf den Block, der den Datensatz mit diesem Suchschlüssel enthält. Um einen bestimmten Datensatz zu finden, muss möglicherweise der gesamte Block durchsucht werden. Sparse Indexe sind in der Regel kleiner als Dense Indexe.

Tradeoff

- Dense Index: Ermöglicht schnelleres Finden von Tupeln in einigen Fällen, da der Indexeintrag direkt auf den Datensatz verweist.
- Sparse Index: Benötigt weniger Speicherplatz, da nicht für jeden Datensatz ein Eintrag existiert. Dies kann die Performance beim Auffinden eines Datensatzes erhöhen, da weniger Indexseiten durchsucht werden müssen, um den relevanten Block zu finden.

Multi-Level Index



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

Einschränkungen von geordneten Dateistrukturen

- **Einfügen und Löschen:**
 - Kann zu einer **teuren Reorganisation von mehreren Ebenen von sortierten Dateien** führen, um die Sortierreihenfolge beizubehalten. Dies ist besonders aufwendig bei sequenzieller Dateiorganisation.

B+-Bäume

- **Balancierte Suchbäume:** Stellen sicher, dass der Suchpfad von der Wurzel zu jedem Blattknoten ungefähr gleich lang ist, was eine effiziente Suche garantiert (logarithmische Komplexität).
- **Die Anzahl von Lookups/Levels ist für alle Einträge gleich:** Dies sorgt für eine vorhersagbare und gute Suchperformance, unabhängig vom gesuchten Schlüssel.
- **Etwas Platz auf jeder Seite/Block lassen:** Um zukünftige Einfügungen zu ermöglichen, ohne sofort eine Reorganisation durchführen zu müssen. Dies verbessert die Performance bei häufigen Einfügeoperationen. B+-Bäume sind eine häufig verwendete Indexstruktur in Datenbanksystemen aufgrund ihrer Effizienz bei Such-, Einfüge- und Löschoperationen sowie bei Bereichsabfragen

Kombinationen von Konzepten (Indexarten)

| | dense | sparse |
|----------------------|-------|--------|
| primary (clustering) | ✓ | ✓ |

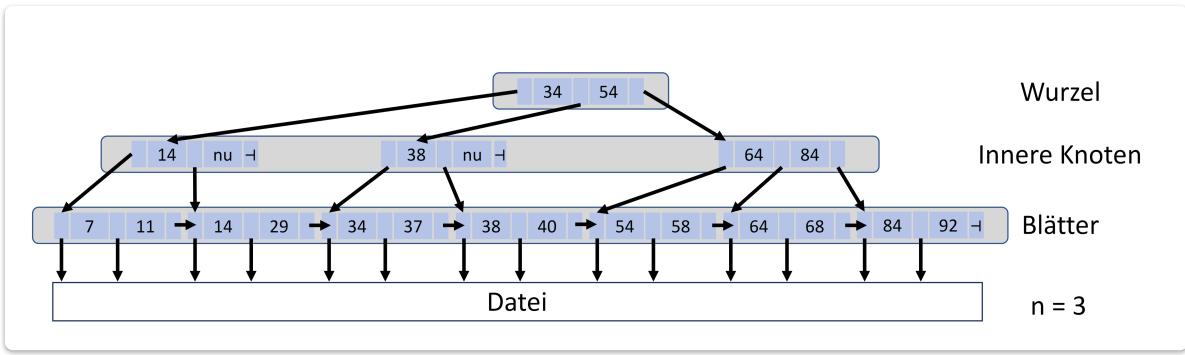
| | dense | sparse |
|----------------------------|-------|--------|
| secondary (non-clustering) | ✓ | % |

Ein Secondary Sparse Index ist nicht sinnvoll!

- Da die Tupel **nicht anhand des Search-Keys sortiert** sind (bei einem sekundären Index ist die physische Reihenfolge der Daten unabhängig vom Indexschlüssel), können wir anhand des ersten Tupels einer Seite **keine Rückschlüsse auf die restlichen Tupel der Seite** schließen. Ein Eintrag im Sparse Index würde nur auf den Anfang einer Seite zeigen, aber wir wüssten nicht, wo sich die anderen Tupel mit dem gesuchten Schlüssel (oder einem ähnlichen Schlüssel) auf dieser Seite befinden.
 - **Daher sind Secondary Indexes immer dense.** Für jeden eindeutigen Wert des Suchschlüssels (oder für jede Seite, je nach Implementierung) muss ein Eintrag im Index vorhanden sein, um direkt zum entsprechenden Datensatz oder zur entsprechenden Seite navigieren zu können.
 - **Clustering dense:** Indexeintrag für **jeden vorkommenden Wert des Search-Keys** \Rightarrow der Indexeintrag zeigt auf das **erste Tupel** mit diesem Wert (da die Daten physisch sortiert sind).
 - **Non-clustering dense:** Indexeinträge für **alle Tupel** (oder für jede Seite, die Tupel mit dem entsprechenden Schlüssel enthält). Da die Daten nicht sortiert sind, muss der Index jeden einzelnen Datensatz (oder jede relevante Seite) adressieren.
-

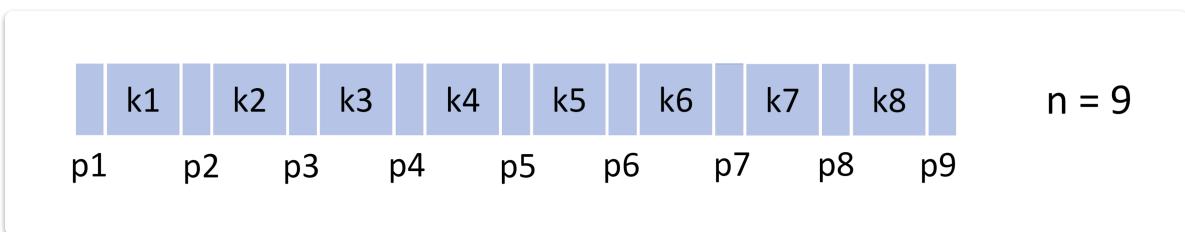
Indexstrukturen - B^+ Bäume

B^+ Baum Beispiel



- **Verzweigungsgrad/Ordnung (Branching Factor, Fanout) n :** Die maximale Anzahl von Kindknoten, die ein interner Knoten in einem B^+ -Baum haben kann. Dieser Parameter beeinflusst die Höhe des Baumes und somit die Anzahl der Zugriffe, die für eine Suche erforderlich sind. Ein höherer Verzweigungsgrad führt zu einem flacheren Baum.
- **\dashv = unbefüllter Pointer:** Markiert eine Stelle in einem Knoten, an der kein Kindknoten oder kein Datensatz-Pointer gespeichert ist.
- **nu (not used):** Bezeichnet einen nicht benutzten Eintrag innerhalb eines Knotens. Dies kann auftreten, wenn Knoten nicht vollständig gefüllt sind, um Einfügungen zu erleichtern.
- **Pointer auf Blattebene zeigen auf Positionen in der Datei:** In B^+ -Bäumen enthalten die Blattknoten die eigentlichen Daten-Einträge (oder Pointer zu den Daten, im Fall von sekundären Indexen). Diese Pointer verweisen auf die physische Speicheradresse der Datensätze auf der Festplatte.

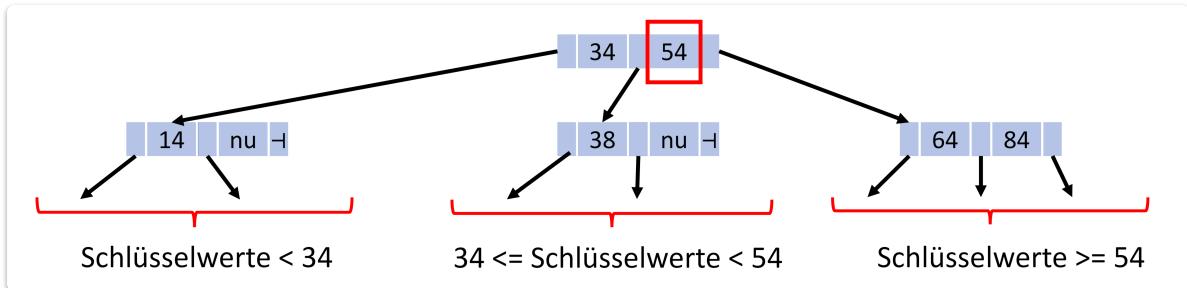
B^+ -Baum Knotenstruktur



- **Wurzel, innere Knoten und Blätter haben die gleiche Struktur.**
- **Jeder Knoten hat n Pointer p_i .** Hier ist $n = 9$. Die Pointer in internen Knoten zeigen auf Kindknoten. In Blattknoten zeigen die Pointer auf Datensätze (oder Speicheradressen der Datensätze) oder, im Fall des letzten Pointers, auf den nächsten Blattknoten.
- **Jeder Knoten hat at most ($n - 1$) Search-Key-Werte k_j .** Hier also maximal $9 - 1 = 8$ Schlüsselwerte. Die Schlüsselwerte in internen Knoten dienen als Wegweiser, um den richtigen Unterbaum für die Suche zu finden. In Blattknoten sind sie die tatsächlichen Suchschlüssel der gespeicherten Datensätze (oder eine Kopie davon).

- Der letzte Pointer auf Blattebene zeigt auf den nächsten Blattknoten in Search-Key-Reihenfolge. Dies ist ein wichtiges Merkmal von B+-Bäumen, das effiziente sequentielle Durchläufe (Bereichsabfragen) der Daten ermöglicht, ohne zum Wurzelknoten zurückkehren zu müssen. Die Blattknoten sind also einfach verkettet.

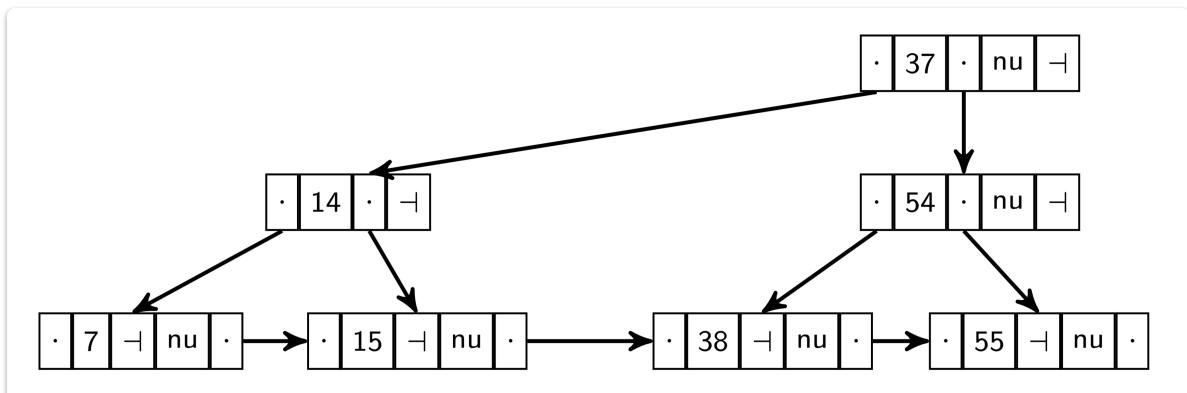
B+-Baum Eigenschaften



- Ordnung:**
 - Die Werte in jedem Knoten sind geordnet: $k_i < k_j$, wenn $i < j$. (Innerhalb eines Knotens sind die Schlüsselwerte sortiert.)
 - Teilbäume sind geordnet. (Alle Schlüsselwerte im linken Unterbaum eines Knotens sind kleiner als der kleinste Schlüsselwert im Knoten, und alle Schlüsselwerte im rechten Unterbaum sind größer oder gleich.)
- Balanciert:** Alle Pfade von der Wurzel zu den Blattknoten haben die gleiche Länge. (Dies garantiert eine logarithmische Suchzeit, da alle Blätter die gleiche Tiefe haben.)
- Verzweigung:** Jeder innere Knoten (nicht die Wurzel) hat zwischen $\lceil \frac{n}{2} \rceil$ und n Kinder. (n ist der Verzweigungsgrad/Ordnung.)
 - Ausnahme: Der Wurzelknoten hat zwischen 2 und n Kinder (wenn er nicht selbst ein Blatt ist).
- Blattknoten:** Haben zwischen $\lceil \frac{n-1}{2} \rceil$ und $n - 1$ Pointer auf Tupel (oder deren Speicheradressen) in der Datei und 1 Pointer auf den nächsten Blattknoten (für effiziente Bereichsabfragen).

Ein minimaler B+-Baum

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



- Für $n = 3$ bedeutet das:
 - Innere Knoten (nicht Wurzel) müssen mindestens $\lceil \frac{3}{2} \rceil = 2$ Kinder haben.
 - Blattknoten müssen mindestens $\lceil \frac{3-1}{2} \rceil = 1$ Schlüsselwert haben.
 - Die Wurzel kann weniger als 2 Kinder haben, wenn sie ein Blatt ist oder die einzige Wurzel.

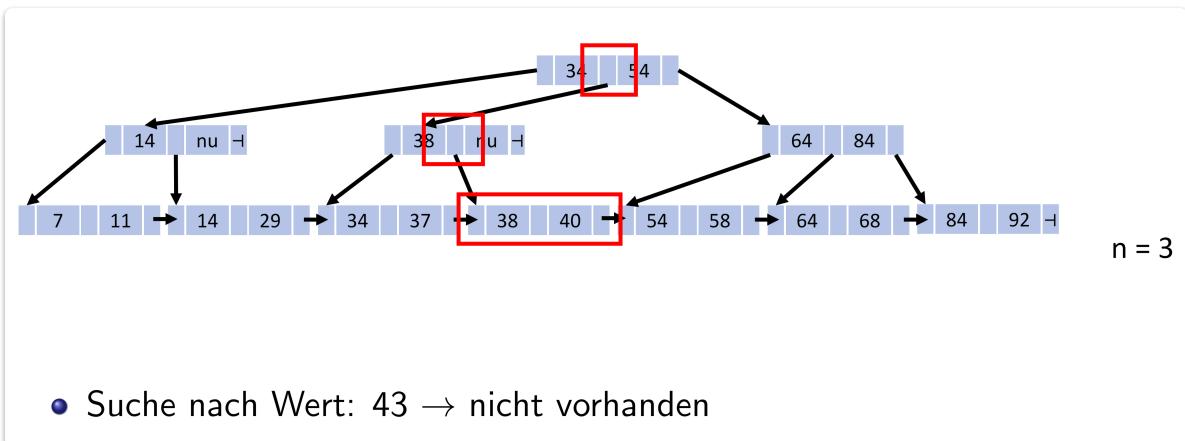
Der gezeigte Baum illustriert diese Minimalbedingungen. Beachte die Verkettung der Blattknoten für sequenzielle Zugriffe.

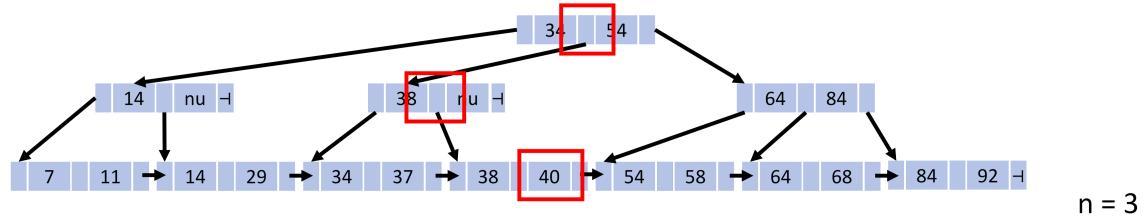
Verwendung von B⁺-Bäumen in DBMS

- **Knotengröße:** Jeder Knoten hat die Größe eines I/O-Blocks (optimiert für den Datentransfer zwischen Speicher und Festplatte).
- **Füllgrad:** Ein Knoten ist zu mindestens 50% gefüllt (gewährleistet eine gewisse Speichereffizienz).
- **Baumstruktur:** Ein B⁺-Baum ist in der Regel sehr flach, d.h. die Suche erfordert nur wenige wahlfreie Zugriffe (schnellere Suchzeiten).
- **Caching:** Die ersten 1-2 Ebenen des Baums sind in der Regel im Hauptspeicher "gecached" (ermöglicht sehr schnelle Zugriffe auf häufig benötigte Indexinformationen).
- **Logische vs. physikalische Nähe:** "Logisch" nahe bedeutet nicht unbedingt "physisch" nahe (die physische Anordnung auf der Festplatte kann fragmentiert sein). Das Lesen eines Knotens erfordert typischerweise einen I/O-Zugriff.
- **Innere Knoten:** Innere Knoten entsprechen einer Hierarchie von *sparse indexes* (enthalten nur wenige, aber repräsentative Schlüssel, um den Suchraum einzuschränken).

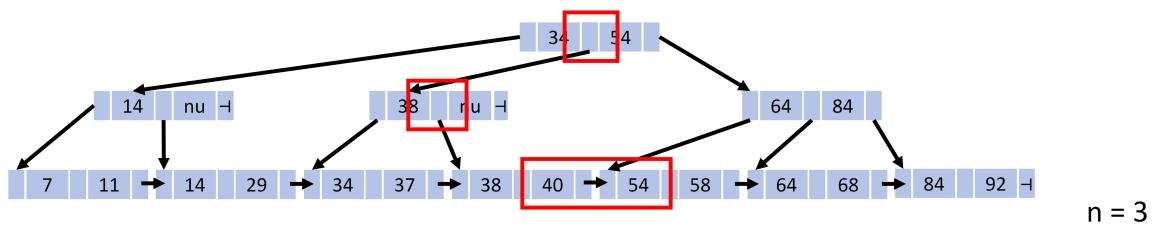
Uniqueness-Constraints: Uniqueness-Constraints auf Attributen in einer Datenbank werden durch B⁺-Bäume realisiert → **Primärschlüssel** (B⁺-Bäume eignen sich gut zur Durchsetzung von Eindeutigkeitsbedingungen und für effiziente Bereichsabfragen).

Beispiel zum Lookup



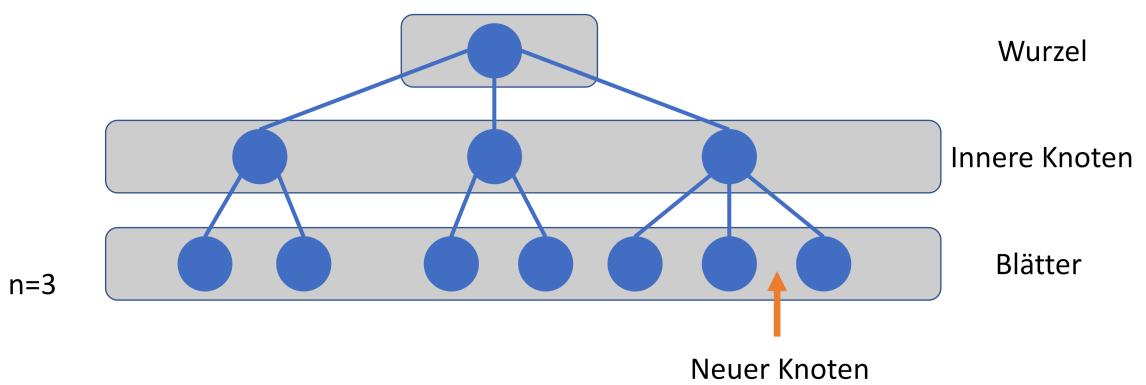


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden

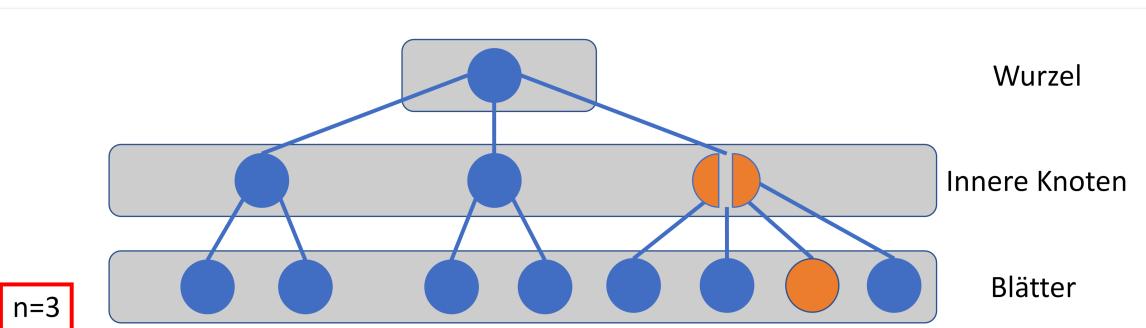


- Suche nach Wert: 43 → nicht vorhanden
- Suche nach Wert: 40 → gefunden
- Suche nach Bereich: 40-55 → gefunden

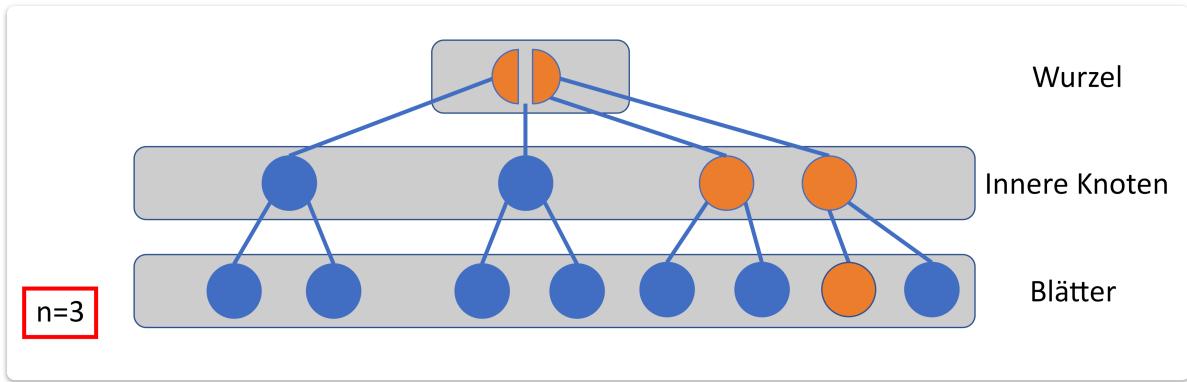
Einfügen beim B^+ Baum



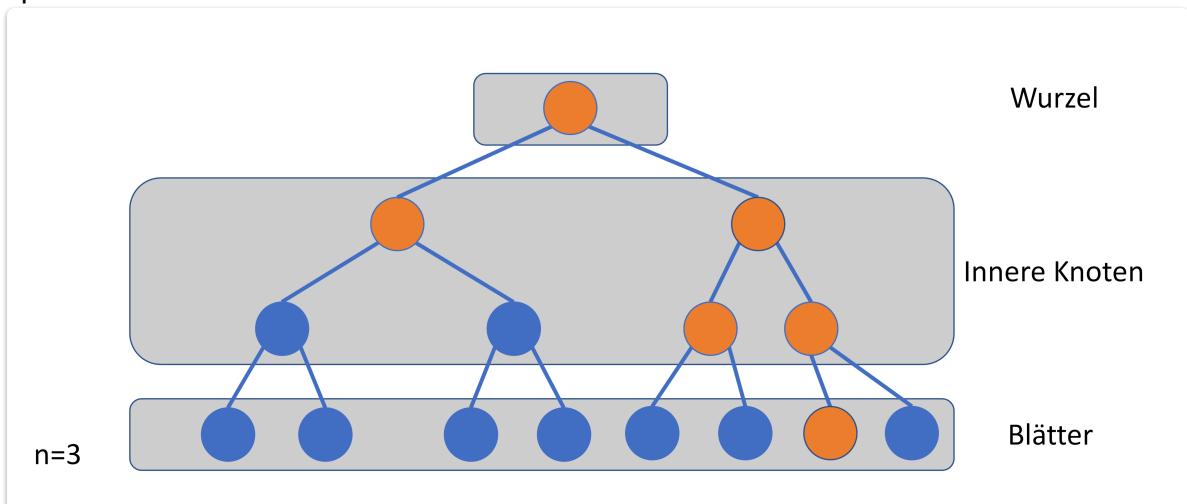
Wir wollen hier den neuen Knoten einfügen



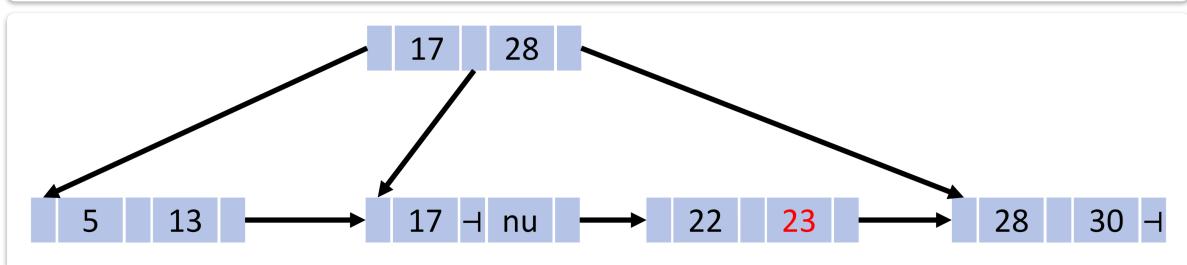
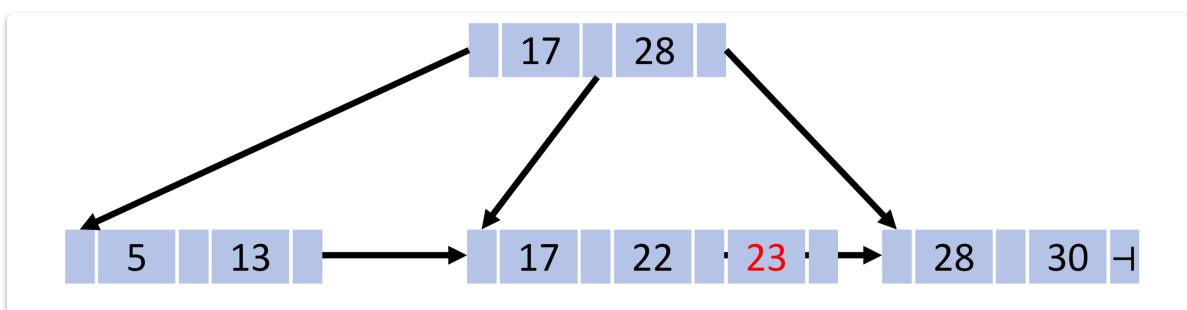
Da unser $n = 3$ müssen wir eine Ebene darüber in 2 teilen

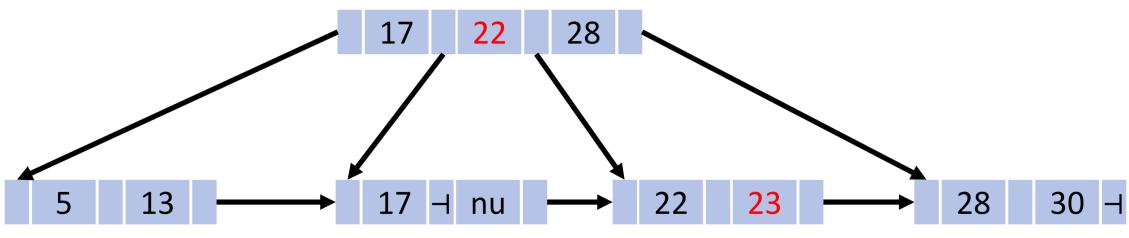


Dann haben wir das selbe Problem in einer Ebene darüber. Da müssen wir auch ein mal spalten und haben am Ende eine Ebene mehr



Beispiel: Einfügen von 23





Mehr Beispiele: [DBS-8_Physischer Datenbankentwurf, p.32](#)

B⁺-Baum Löschen

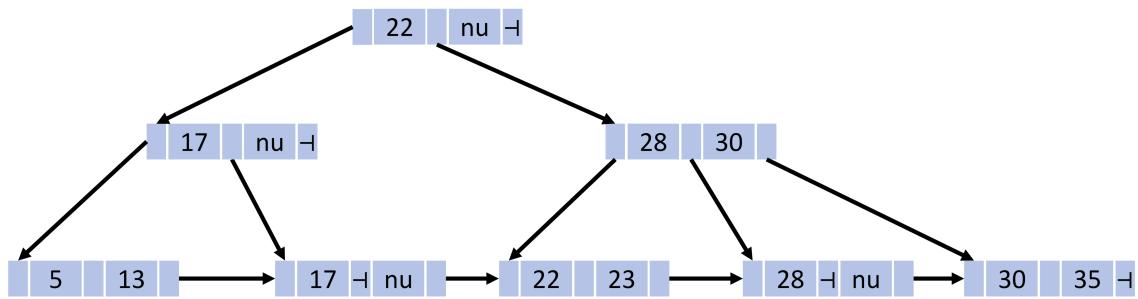
Löschen von Schlüsselwert k:

- **Suche:** Finde das Blatt b , das den Schlüsselwert k enthält.
- **Löschen (bei ausreichendem Füllgrad):**
 - Falls b genügend gefüllt bleibt (mindestens $\lceil \frac{n}{2} \rceil$ Einträge, wobei n die maximale Anzahl an Einträgen pro Knoten ist), lösche k .
 - **Wichtig:** Innere Knoten können dann Schlüssel enthalten, die nicht mehr in den Blättern existieren (diese dienen weiterhin als Wegweiser).
- **Verschmelzen oder Umverteilen (bei Unterschreitung des minimalen Füllgrads):**
 - **Verschmelzen:**
 - Falls Verschmelzen mit einem Nachbarknoten möglich ist (der Nachbarknoten hat nicht den minimalen Füllgrad überschritten, sodass nach dem verschmelzen beider Knoten der minimale Füllgrad nicht unterschritten wird), verschmelze die Knoten und passe den Pointer im Elternknoten an.
 - **Umverteilung:**
 - Falls Verschmelzen nicht möglich ist (kein geeigneter Nachbarknoten vorhanden), versuche eine Neuverteilung der Schlüssel und Pointer über den Elternknoten (ein Schlüssel wird vom Nachbarn "ausgeliehen").
- **Kaskadierendes Verschmelzen:** Verschmelzen muss unter Umständen auch in höheren Ebenen des Baumes erfolgen, wenn durch das Verschmelzen in einer unteren Ebene der minimale Füllgrad im Elternknoten unterschritten wird.
- **Baumtiefe:** Die Tiefe des Baumes kann sich um eine Ebene verringern, wenn die Wurzel nach einer Verschmelzungsoperation nur noch einen oder keinen Kindknoten hat.

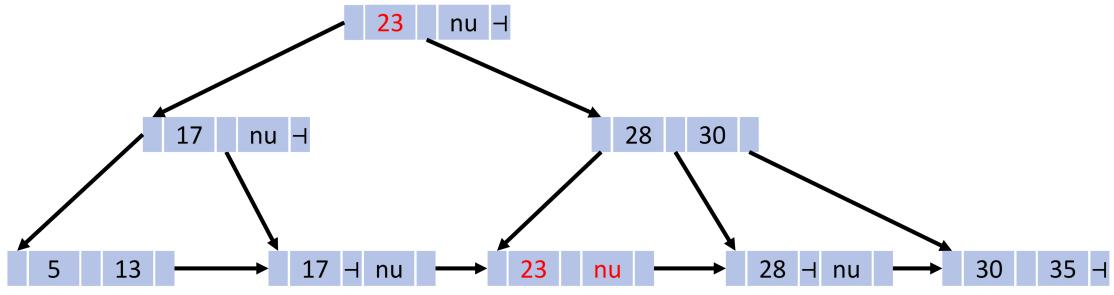
Beispiele:

Löschen von 22

n=3, Löschen von 22

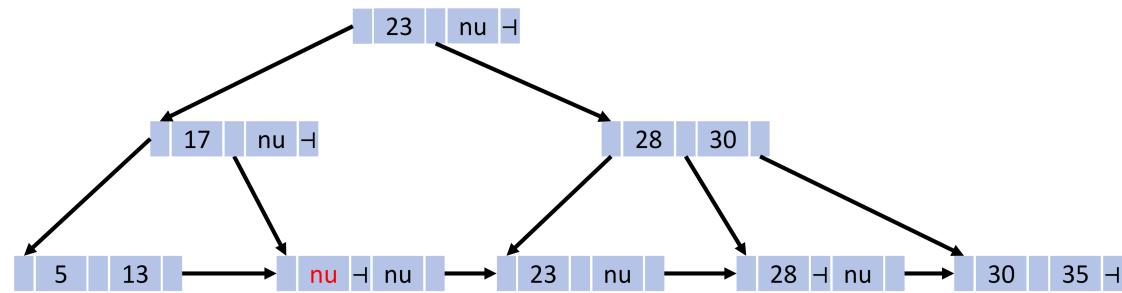


n=3, Löschen von 22

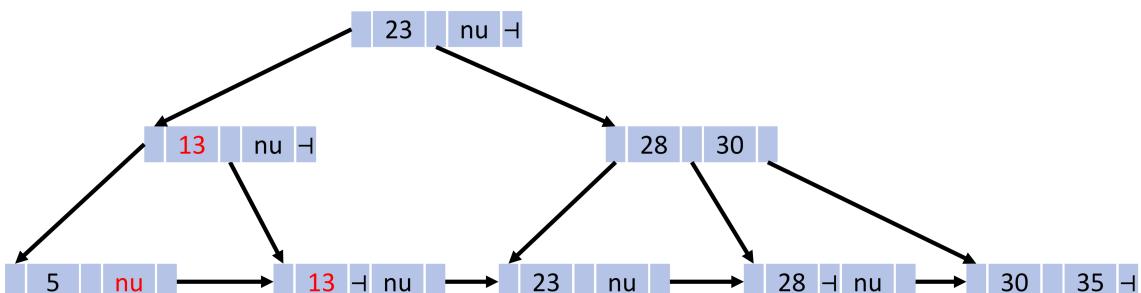


Löschen von 17

n=3, Löschen von 17

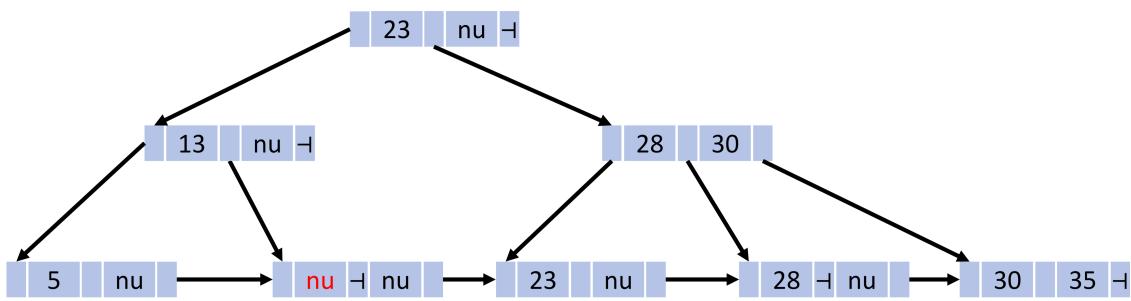


n=3, Löschen von 17

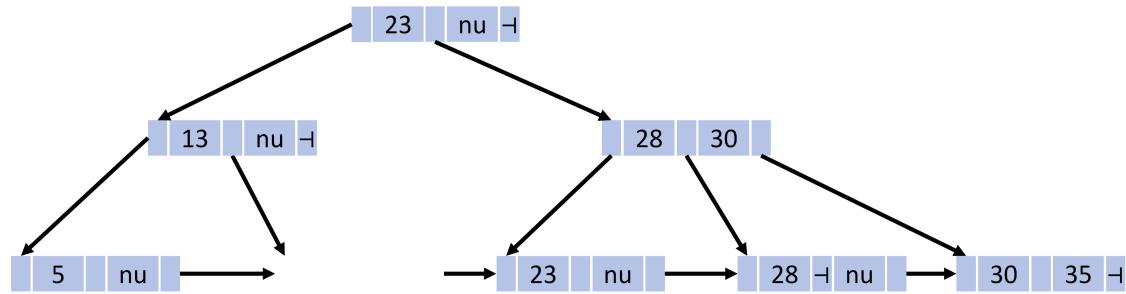


Löschen von 13

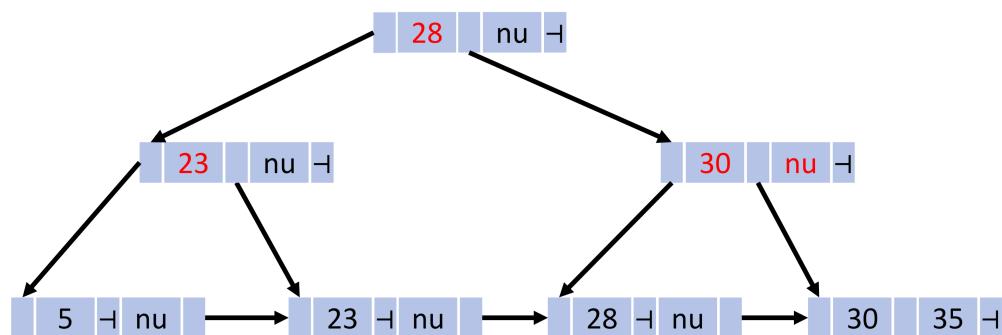
n=3, Löschen von 13



n=3, Löschen von 13

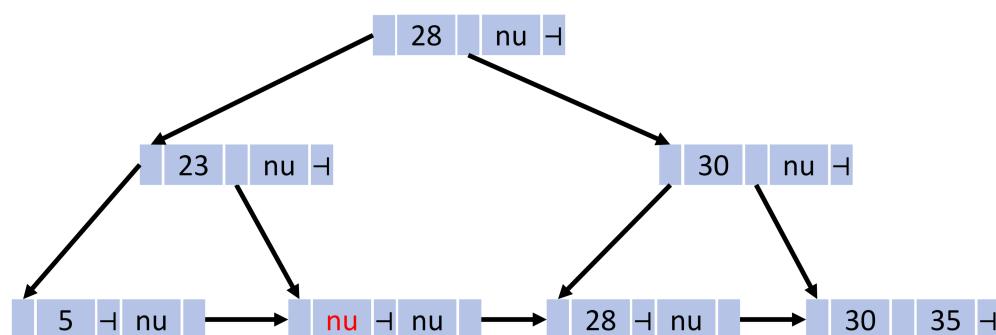


n=3, Löschen von 13

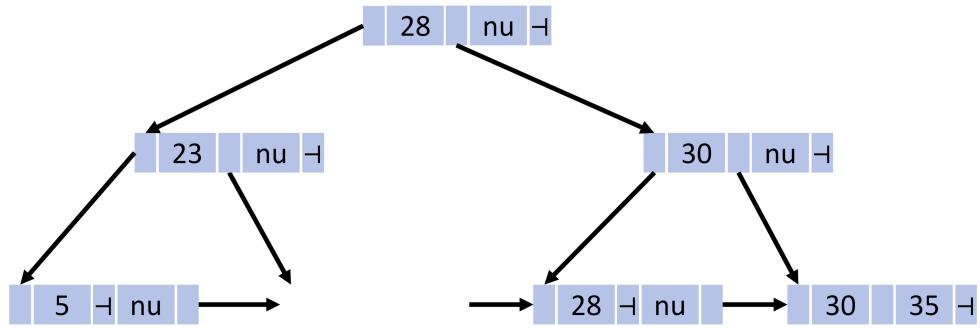


Löschen von 23

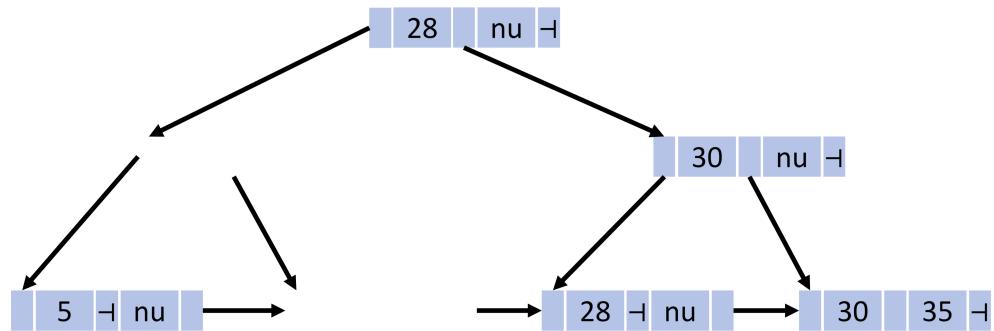
n=3, Löschen von 23



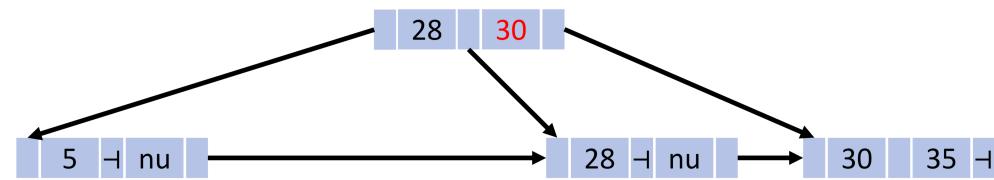
n=3, Löschen von 23



n=3, Löschen von 23



n=3, Löschen von 23



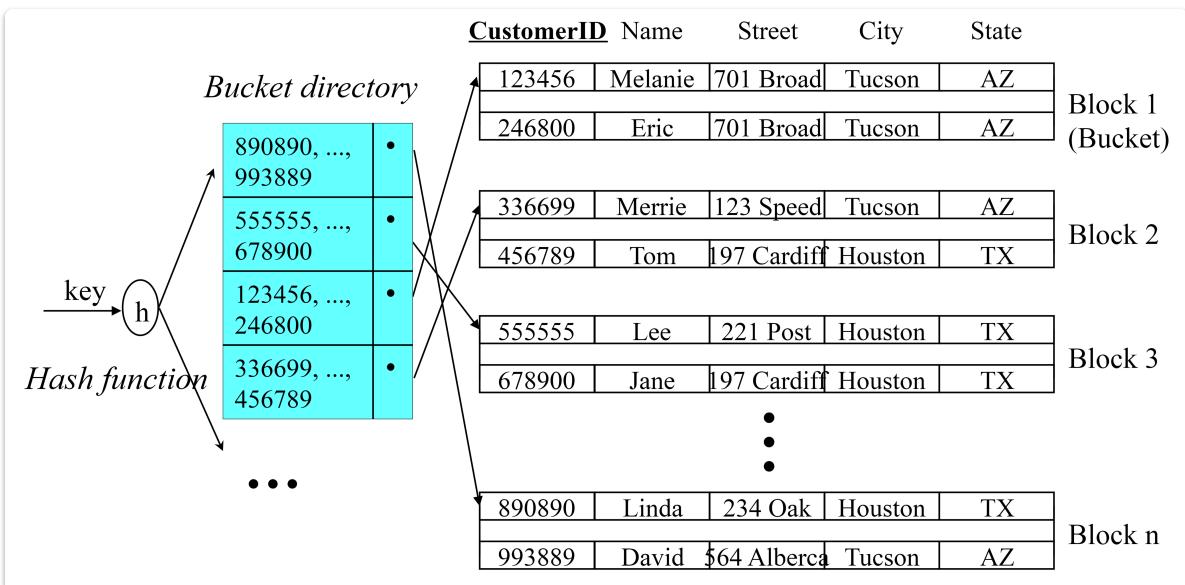
Indexstrukturen - Hashing

Statischer Hash-Index

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

- **Hashfunktion:**

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



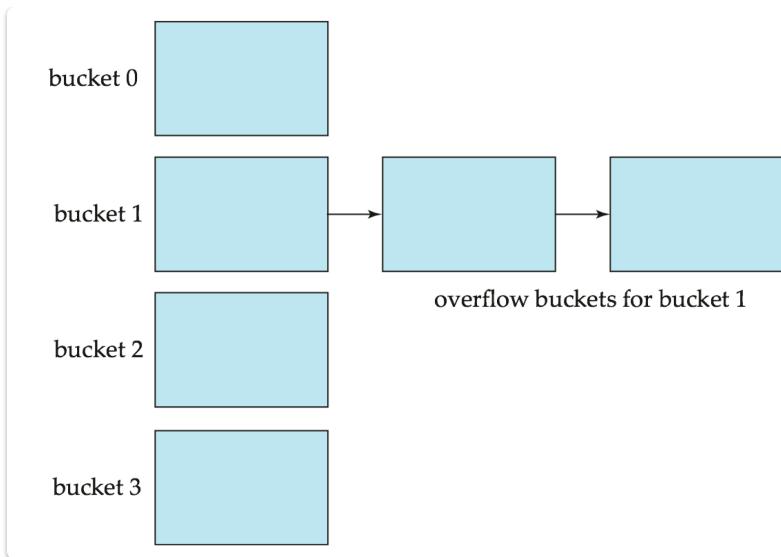
(Die Abbildung zeigt eine Tabelle mit Kundendaten, eine Hashfunktion h , ein Bucket Directory, das Hashes auf Buckets abbildet, und die eigentlichen Datenblöcke (Buckets) mit den Datensätzen.)

Suche:

- Ein Zugriff auf das **Bucket Directory** (um das zugehörige Bucket zu finden).
- Ein Zugriff auf die **Datei** (genauer: auf den Datenblock/das Bucket), um die gesuchten Datensätze zu finden.

Gute Performance hängt von einer guten Hashfunktion ab:

- Bucket kann überfüllt sein.
- Zu viele Tupel werden auf das gleiche Bucket abgebildet ⇒ **Kollision**.
- **Lösung:**
 - **Overflow-Buckets:** Zusätzliche Buckets, die verkettet werden, um Überläufe aufzufangen.
 - **Overflow-Chains:** Verkettung von Blöcken innerhalb eines Buckets oder von Overflow-Buckets.



Statischer Hash-Index

- **Open vs. Closed Hashing:**
 - **Open Hashing:**
 - Overflow Chains mit weiteren Overflow Buckets (jeder Bucket kann eine verkettete Liste von Überlauf-Buckets haben).
 - **Closed Hashing:**
 - Feste Anzahl von Buckets.
 - Beim Überlaufen muss eines der existierenden Buckets verwendet werden (z.B. durch Linear Probing, weitere Hashfunktionen).

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, Performance leidet.
- **Initiale Bucketanzahl zu groß:**
 - ⇒ Underflow, Speicherplatz wird verschwendet.

Lösungsansätze:

- **Periodische Reorganisation:** Anpassen der Hashfunktion und der Bucketanzahl in regelmäßigen Abständen (kann ressourcenintensiv sein und zu Ausfallzeiten führen).
- **Dynamic Hashing:** Ermöglicht Modifikationen zu einem späteren Zeitpunkt (z.B. durch Erweitern oder Verkleinern der Bucketanzahl on-the-fly).

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:

- **System-Katalog hat veraltete Informationen** (der Query-Optimierer trifft Entscheidungen basierend auf möglicherweise falschen Statistiken).
- **Optimierer könnte annehmen, dass die Tabelle klein ist** (bei kleinen Tabellen kann ein Full Table Scan effizienter sein als die Indexnutzung).
- **"Gute" und "schlechte" Typen von Anfragen (bezüglich Indexnutzung):**
 - `SELECT * FROM EMP WHERE salary/12 > 4000` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).
 - `SELECT * FROM EMP WHERE salary > 48000` (Bereichsabfragen können Indexe gut nutzen).
 - `SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G'` (Funktionen auf indizierten Spalten verhindern oft die Indexnutzung).
 - `SELECT * FROM EMP WHERE name LIKE 'G%'` (führende Wildcards können Indexnutzung einschränken, nachfolgende Wildcards oft nicht).
 - `SELECT * FROM EMP WHERE name = 'Smith'` (Punktuelle Suchen auf indizierten Spalten sind ideal für Indexe).
 - `SELECT * FROM EMP WHERE salary IS NULL` (Indexe können oft auch für IS NULL-Bedingungen genutzt werden).
- **Geschachtelte SQL-Anfragen** (die Effizienz der Indexnutzung kann von der Struktur der geschachtelten Anfrage abhängen).
- **Negationen** (`NOT`, `!=`) (können die Indexnutzung erschweren).
- **Anfragen mit OR** (die Indexnutzung bei OR-Bedingungen hängt von der Verfügbarkeit von Indexen auf den beteiligten Spalten ab und wie der Optimierer die Anfrage

umformuliert).

Zusammenfassung

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

Kompromiss zwischen Speicherplatz und Zeit: Intelligente Nutzung von zusätzlichem Speicherplatz erhöht im Allgemeinen die Performance (z.B. durch Caching von Indexen).

Kompromiss zwischen Select-Anfragen und Updates: Wenn Maßnahmen zur Beschleunigung von Select-Anfragen getroffen werden, geschieht dies meist zum Nachteil von Updates – und umgekehrt (z.B. viele Indexe beschleunigen Suchen, verlangsamen aber Schreiboperationen).