

Datenbanken 2

Indexstrukturen

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg



WS 2020/21

Version 17. November 2020

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Literatur und Quellen

Lektüre zum Thema "Indexstrukturen":

- Kapitel 7 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 11 in Silberschatz, Korth, and Sudarashan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Grundlagen

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Grundlagen/1

- **Index beschleunigt Zugriff**, z.B.:
 - Autorenkatalog in Bibliothek
 - Index in einem Buch
- **Index-Datei** besteht aus Datensätzen: den Index-Einträgen
- **Index-Eintrag** hat die Form (Suchschlüssel, Pointer)
 - *Suchschlüssel*: Attribut(liste) nach der Daten gesucht werden
 - *Pointer*: Pointer auf einen Datensatz (TID)
- **Suchschlüssel darf mehrfach vorkommen** (im Gegensatz zu Schlüssel von Relationen)
- Index-Datei meist viel **kleiner** als die indizierte Daten-Datei

Grundlagen/2

- **Merkmale des Index** sind:
 - Zugriffszeit
 - Zeit für Einfügen
 - Zeit für Löschen
 - Speicherbedarf
 - effizient unterstützte Zugriffsarten
- **Wichtigste Zugriffsarten** sind:
 - Punktanfragen: z.B. Person mit SVN=1983-3920
 - Mehrpunktanfragen: z.B. Personen, die 1980 geboren wurden
 - Bereichsanfragen: z.B. Personen die mehr als 100.000 EUR verdienen

Grundlagen/3

Indextypen werden nach folgenden Kriterien unterschieden:

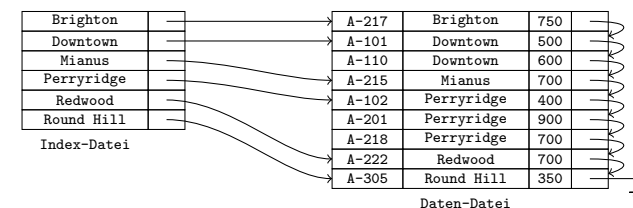
- **Ordnung der Daten- und Index-Datei:**
 - Clustered Index
 - Non-Clustered Index
- **Art der Index-Einträge:**
 - sparse Index
 - dense Index

Nicht alle Kombinationen üblich/möglich:

- **Clustered Index** ist oft sparse
- **Non-Clustered Index** ist immer dense

Clustered Index

- **Index-Datei:**
 - sequentiell geordnet nach Suchschlüssel
- **Daten-Datei:**
 - sequentiell geordnet nach Suchschlüssel
- **Effiziente Zugriffsarten:**
 - Punkt-, Mehrpunkt-, und Bereichsanfragen
 - nicht-sequentieller Zugriff (random access)
 - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

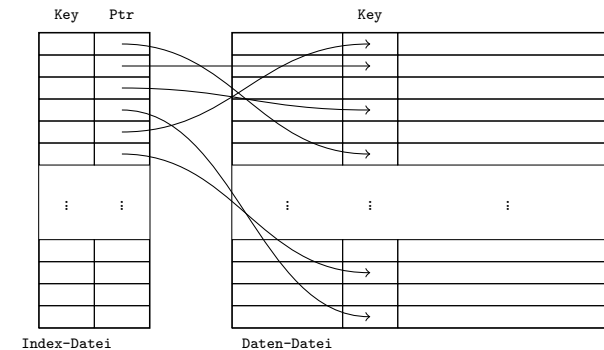


Non-Clustered Index/1

- **Clustered vs. Non-Clustered Index:**
 - nur 1 Clustered Index möglich
 - beliebig viele Non-Clustering Indizes
 - Non-Clustered Index für schnellen Zugriff auf alle Felder, die nicht Suchschlüssel des Clustered Index sind
- **Beispiel:** Konten mit Clustered Index auf Kontonummer
 - Finde alle Konten einer bestimmten Filiale.
 - Finde alle Konten mit 1000 bis 1500 EUR Guthaben.
- **Ohne Index** können diese Anfragen **nur durch sequentielles Lesen** aller Knoten beantwortet werden – sehr langsam
- **Non-Clustered Index** für schnellen Zugriff **erforderlich**

Non-Clustered Index/2

- **Index-Datei:**
 - sequentiell nach Suchschlüssel geordnet
- **Daten-Datei:**
 - *nicht* nach Suchschlüssel geordnet



Non-Clustered Index/3

- **Effiziente Zugriffsarten:**
 - sehr schnell für Punktanfragen
 - Mehrpunkt- und Bereichsanfragen: gut wenn nur kleiner Teil der Tabelle zurückgeliefert wird (wenige %)
 - besonders für nicht-sequentiellen Zugriff (random access) geeignet

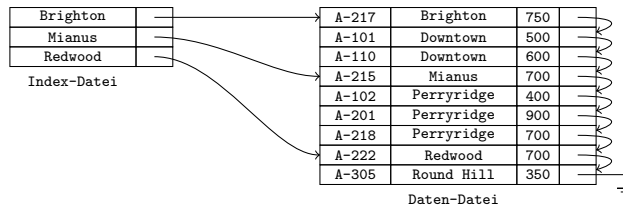
Primär- und Sekundärindex

Folgende Begriffe finden sich häufig in der Praxis:

- **Primärindex:** Clustered Index mit eindeutigem Suchschlüssel
- **Sekundärindex:** Synonym für Non-Clustered Index

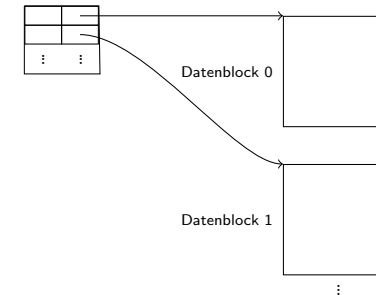
Sparse Index/1

- Sparse Index
 - ein Index-Eintrag für mehrere Datensätze
 - kleiner Index: weniger Index-Einträge als Datensätze
 - nur möglich wenn Datensätze nach Suchschlüssel geordnet sind (d.h. Clustered Index)



Sparse Index/2

- Oft enthält ein sparse Index einen Eintrag pro Block.
- Der Suchschlüssel, der im Index für eine Block gespeichert wird, ist der kleinste Schlüssel in diesem Block.



Dense Index/1

- Dense Index:
 - Index-Eintrag (bzw. Pointer in Bucket) für jeden Datensatz in der Daten-Datei
 - dense Index kann groß werden (aber normalerweise kleiner als Daten)
 - Handhabung einfacher, da ein Pointer pro Datensatz
- Non-Clustered Index ist immer dense

Gegenüberstellung von Index-Typen

- Alle Index-Typen machen Punkt-Anfragen erheblich schneller.
- Index erzeugt Kosten bei Updates: Index muss auch aktualisiert werden.
- Dense/Sparse und Clustering/Non-Clustering:
 - Clustered Index kann dense oder sparse sein
 - Non-Clustered Index ist immer dense
- Sortiert lesen (=sequentielles Lesen nach Suchschlüssel-Ordnung):
 - mit Clustered Index schnell
 - mit Non-Clustered Index teuer, da sich aufeinander folgende Datensätze auf unterschiedlichen Blöcken befinden (können)
- Dense vs. Sparse:
 - sparse Index braucht weniger Platz
 - sparse Index hat geringere Kosten beim Aktualisieren
 - dense Index erlaubt bestimmte Anfragen zu beantworten, ohne dass Datensätze gelesen werden müssen ("covering index")

Duplikate/1

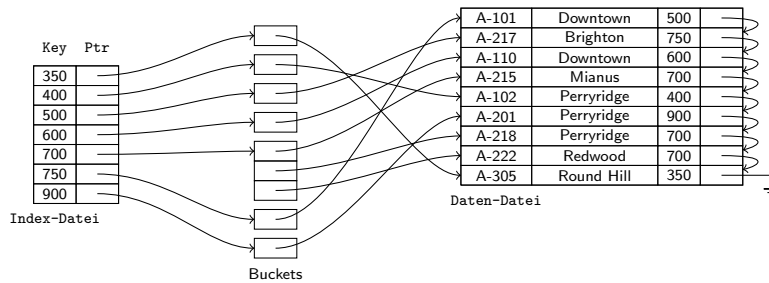
Umgang mit mehrfachen Suchschlüsseln:

(a) Doppelte Indexeinträge:

- ein Indexeintrag für jeden Datensatz
- schwierig zu handhaben, z.B. in B^+ -Baum Index

(b) Buckets:

- nur einen Indexeintrag pro Suchschlüssel
- Index-Eintrag zeigt auf ein Bucket
- Bucket zeigt auf alle Datensätze zum entsprechenden Suchschlüssel
- zusätzlicher Block (Bucket) muss gelesen werden



Duplikate/2

Umgang mit mehrfachen Suchschlüsseln:

(c) Suchschlüssel eindeutig machen:

- Einfügen: TID wird an Suchschlüssel angehängt (sodass dieser eindeutig wird)
- Löschen: Suchschlüssel und TID werden benötigt (ergibt genau 1 Index-Eintrag)
- Suche: nur Suchschlüssel wird benötigt (ergibt mehrere Index-Einträge)
- wird in der Praxis verwendet

In den restlichen Folien wird angenommen, dass Suchschlüssel eindeutig sind bzw. eindeutig erstellt wurden.

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Inhalt

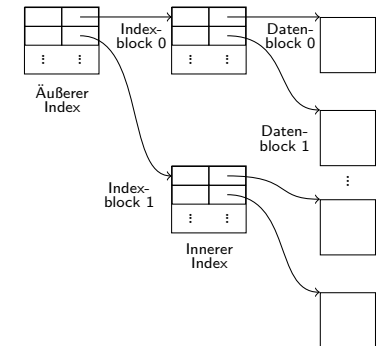
- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Mehrstufiger Index/1

- Großer Index wird teuer:
 - Index passt nicht mehr in Hauptspeicher und mehrere Block-Lese-Operationen werden erforderlich
 - binäre Suche: $\lfloor \log_2(B) \rfloor + 1$ Block-Lese-Operationen (Index mit B Blöcken)
 - eventuelle Overflow Blöcke müssen sequentiell gelesen werden
- Lösung: Mehrstufiger Index
 - Index wird selbst wieder indiziert
 - dabei wird der Index als sequentielle Daten-Datei behandelt

Mehrstufiger Index/2

- Mehrstufiger Index:
 - Innerer Index: Index auf Daten-Datei
 - Äußerer Index: Index auf Index-Datei
- Falls äußerer Index zu groß wird, kann eine weitere Index-Ebene eingefügt werden.
- Diese Art von (ein- oder mehrstufigem) Index wird auch als **ISAM** (Index Sequential Access Method) oder **index-sequentielle Datei** bezeichnet.



Mehrstufiger Index/3

- Index Suche
 - beginne beim Root-Knoten
 - finde alle passenden Einträge und verfolge die entsprechenden Pointer
 - wiederhole bis Pointer auf Datensatz zeigt (Blatt-Ebene)
- Index Update: Löschen und Einfügen
 - Indizes aller Ebenen müssen nachgeführt werden
 - Update startet beim innersten Index
 - Erweiterungen der Algorithmen für einstufige Indizes

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

B^+ -Baum/1

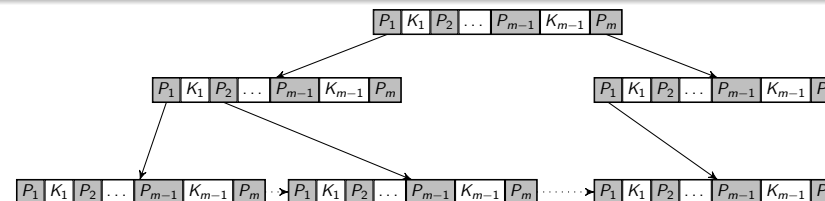
B^+ -Baum: Alternative zu index-sequentiellen Dateien:

- **Vorteile** von B^+ -Bäumen:
 - Anzahl der Ebenen wird automatisch angepasst
 - reorganisiert sich selbst nach Einfüge- oder Lösch-Operationen durch kleine lokale Änderungen
 - reorganisieren des gesamten Indexes ist nie erforderlich
- **Nachteile** von B^+ -Bäumen:
 - evtl. Zusatzaufwand bei Einfügen und Löschen
 - etwas höherer Speicherbedarf
 - komplexer zu implementieren
- Vorteile wiegen Nachteile in den meisten Anwendungen bei weitem auf, deshalb sind B^+ -Bäume die meist-verbreitete Index-Struktur

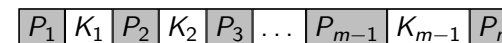
Terminologie und Notation

- Ein Paar (P_i, K_i) ist ein Eintrag
- $L[i] = (P_i, K_i)$ bezeichnet den i -ten Eintrag von Knoten L
- **Daten-Pointer**: Pointer zu Datensätzen sind nur in den Blättern gespeichert
- **Verbindung zwischen Blättern**: der letzte Pointer im Blatt, P_m , zeigt auf das nächste Blatt

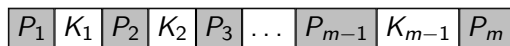
Anmerkung: Es gibt viele Varianten des B^+ -Baumes, die sich leicht unterscheiden. Auch in Lehrbüchern werden unterschiedliche Varianten vorgestellt. Für diese Lehrveranstaltung gilt der B^+ -Baum, wie er hier präsentiert wird.

 B^+ -Baum/2

- **Knoten mit Grad m** : enthält bis zu $m - 1$ Suchschlüssel und m Pointer
 - Knotengrad $m > 2$ entspricht der maximalen Anzahl der Pointer
 - Suchschlüssel im Knoten sind sortiert
 - Knoten (außer Wurzel) sind mindestens halb voll
- **Wurzelknoten**:
 - als Blattknoten: 0 bis $m - 1$ Suchschlüssel
 - als Nicht-Blattknoten: mindestens 2 Kinder
- **Innerer Knoten**: $\lceil m/2 \rceil$ bis m Kinder (=Anzahl Pointer)
- **Blattknoten**: $\lceil (m - 1)/2 \rceil$ bis $m - 1$ Suchschlüssel bzw. Daten-Pointer
- **balancierter Baum**: alle Pfade von der Wurzel zu den Blättern sind gleich lang (maximal $\lceil \log_{\lceil m/2 \rceil}(L) \rceil$ Kanten für L Blattknoten)

 B^+ -Baum Knotenstruktur/1**Blatt-Knoten:**

- K_1, \dots, K_{m-1} sind Suchschlüssel
- P_1, \dots, P_{m-1} sind Daten-Pointer
- Suchschlüssel sind sortiert: $K_1 < K_2 < K_3 < \dots < K_{m-1}$
- Daten-Pointer P_i , $1 \leq i \leq m - 1$, zeigt auf einen Datensatz mit Suchschlüssel K_i
- P_m zeigt auf das nächste Blatt in Suchschlüssel-Ordnung

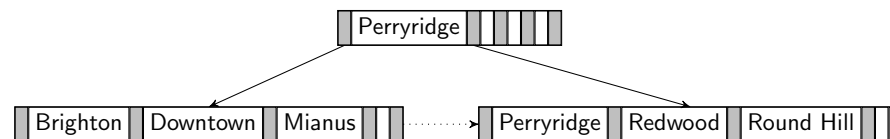
B^+ -Baum Knotenstruktur/2

Innere Knoten:

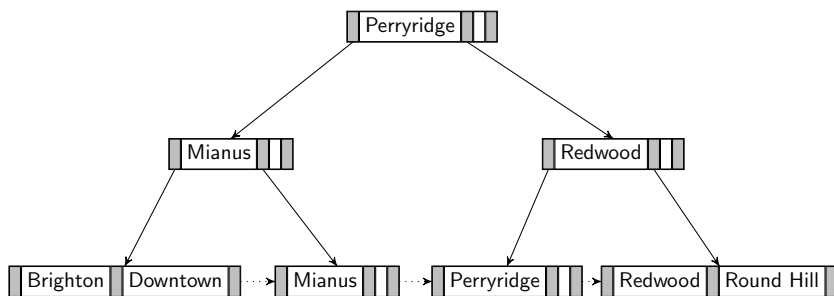
- Stellen einen **mehrstufigen sparse Index** auf die Blattknoten dar
- Suchschlüssel im Knoten sind **eindeutig**
- P_1, \dots, P_m sind **Pointer zu Kind-Knoten**, d.h., zu Teilbäumen
- Alle **Suchschlüssel k** im Teilbaum von P_i haben folgende Eigenschaften:
 - $i = 1: k < K_1$
 - $1 < i < m: K_{i-1} \leq k < K_i$
 - $i = m: k \geq K_{m-1}$

Beispiel: B^+ -Baum/1

- **Index auf Konto-Relation** mit Suchschlüssel Filiale
- B^+ -Baum mit Knotengrad $m = 5$:
 - Wurzel: mindestens 2 Pointer zu Kind-Knoten
 - Innere Knoten: $\lceil m/2 \rceil = 3$ bis $m = 5$ Pointer zu Kind-Knoten
 - Blätter: $\lceil (m-1)/2 \rceil = 2$ bis $m-1 = 4$ Suchschlüssel

Beispiel: B^+ -Baum/2

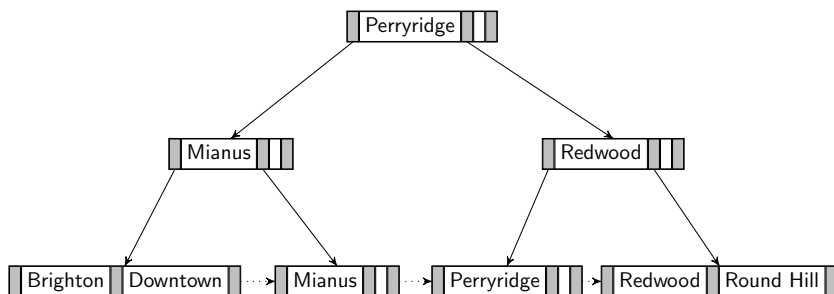
- B^+ -Baum für Konto-Relation (Knotengrad $m = 3$)
 - Wurzel: mindestens 2 Pointer zu Kind-Knoten
 - Innere Knoten: $\lceil m/2 \rceil = 2$ bis $m = 3$ Pointer zu Kind-Knoten
 - Blätter: $\lceil (m-1)/2 \rceil = 1$ bis $m-1 = 2$ Suchschlüssel

Suche im B^+ -Baum/1

- **Algorithmus: Suche** alle Datensätze mit Suchschlüssel k (Annahme: dichte B^+ -Baum Index):
 1. $C \leftarrow$ Wurzelknoten
 2. **while** C keine Blattknoten **do**
 - suche im Knoten C nach dem größten Schlüssel $K_i \leq k$
 - if** ein Schlüssel $K_i \leq k$ existiert
 - then** $C \leftarrow$ Knoten auf den P_{i+1} zeigt
 - else** $C \leftarrow$ Knoten auf den P_1 zeigt
 3. **if** es gibt einen Schlüssel K_i in C sodass $K_i = k$
 - then** folge Pointer P_i zum gesuchten Datensatz (oder Bucket)
 - else** kein Indexeintrag mit Suchschlüssel k existiert

Suche im B^+ -Baum/2

- **Beispiel:** Finde alle Datensätze mit Suchschlüssel $k = \text{Mianus}$
 - Beginne mit dem Wurzelknoten
 - Kein Schlüssel $K_i \leq \text{Mianus}$ existiert, also folge P_1
 - $K_1 = \text{Mianus}$ ist der größte Suchschlüssel $K_i \leq \text{Mianus}$, also folge P_2
 - Suchschlüssel Mianus existiert, also folge dem ersten Datensatz-Pointer P_1 um zum Datensatz zu gelangen

Suche im B^+ -Baum/3

- **Suche durchläuft Pfad** von Wurzel bis Blatt:
 - Länge des Pfads höchstens $\lceil \log_{\lceil m/2 \rceil}(L) \rceil$ für L Blattknoten
 $\Rightarrow \lceil \log_{\lceil m/2 \rceil}(L) \rceil + 1$ Blöcke¹ müssen gelesen werden
 - sind die Blattknoten nur minimal voll ($\lceil (m-1)/2 \rceil$), ergibt sich die maximale Anzahl der Blattknoten: $L = \left\lceil \frac{K}{\lceil (m-1)/2 \rceil} \right\rceil$
 - Wurzelknoten bleibt in der Praxis im Hauptspeicher, oft auch dessen Kinder, dadurch werden 1–2 Block-Zugriffe pro Suche gespart
- **Suche effizienter als in sequentiellem Index:**
 - bis zu $\lfloor \log_2(B) \rfloor + 1$ Blöcke¹ lesen im einstufigen sequentiellen Index (binäre Suche, Index mit B Blöcken, $B = \lceil K/(m-1) \rceil$)

¹nur Index Blöcke werden gezählt, Datenzugriff hier nicht berücksichtigt

Integrierte Übung 2.1

Es soll ein Index mit $K = 10^6$ verschiedenen Suchschlüsseln erstellt werden. Ein Knoten kann maximal 200 Schlüssel mit den entsprechenden Pointern speichern. Es soll nach einem bestimmten Suchschlüssel k gesucht werden.

- Wie viele Block-Zugriffe erfordert ein B^+ -Baum Index maximal, wenn kein Block im Hauptspeicher ist?
- Wie viele Block-Zugriffe erfordert ein einstufiger, sequentieller Index mit binärer Suche?

Einfügen in B^+ -Baum/1

- **Datensatz mit Suchschlüssel k einfügen:**
 - füge Datensatz in Daten-Datei ein (ergibt Pointer)
 - finde Blattknoten für Suchschlüssel k
 - falls** im Blatt noch Platz ist **dann:**
 - füge (Pointer, Suchschlüssel)-Paar so in Blatt ein, dass Ordnung der Suchschlüssel erhalten bleibt
 - sonst** (Blatt ist voll) teile Blatt-Knoten:
 - sortiere alle Suchschlüssel (einschließlich k)
 - die Hälfte der Suchschlüssel bleiben im alten Knoten
 - die andere Hälfte der Suchschlüssel kommt in einen neuen Knoten
 - füge den kleinsten Eintrag des neuen Knotens in den Eltern-Knoten des geteilten Knotens ein
 - falls** Eltern-Knoten voll ist **dann:**
 - teile den Knoten und propagiere Teilung nach oben, sofern nötig

Einfügen in B^+ -Baum/2

- Aufteilvorgang:

- falls nach einer Teilung der neue Schlüssel im Elternknoten nicht Platz hat wird auch dieser geteilt
- im schlimmsten Fall wird der Wurzelknoten geteilt und der B^+ -Baum wird um eine Ebene tiefer

Algorithmus: Einfügen in B^+ -Baum/1 – Blattknoten

→ Knoten L , Suchschlüssel k , Pointer p zu Datensatz

Algorithm 1: B+Tree-Insert(L, k, p)

if L has fewer than $m - 1$ key values then

 insert(k, p) into L

else

$T \leftarrow L \cup (k, p)$; sort T ; // Knoten teilen
 create new node L' ; // temporärer Speicher

$L'.p_m \leftarrow L.p_m$; // füge neuen Knoten L' in
 remove all entries from L ; // ...verkettete Liste
 $L.p_m \leftarrow L'$; // ...von Blättern ein

 copy $T.p_1$ through $T.k_{\lceil m/2 \rceil}$ into L ; // Verteile Einträge auf L
 copy $T.p_{\lceil m/2 \rceil + 1}$ through $T.k_m$ into L' ; // ...und L'

$k' \leftarrow T.k_{\lceil m/2 \rceil + 1}$; // neuen Knoten L' mit Schlüssel k'
 B+Tree-InsertInParent(L, k', L'); // ...in Elternknoten verlinken

Algorithmus: Einfügen in B^+ -Baum/2 – Innerer Knoten**Algorithm 2:** B+Tree-InsertInParent(L, k, L')

if L is root then // Wurzel teilen
 create new root with children L, L' and key value k ;
 return;

$P \leftarrow \text{parent}(L)$; // Paar (k, p) soll in Knoten P
 $p \leftarrow \text{pointer to } L'$; // ...eingefügt werden

if P has fewer than m pointers then // Teilen nicht erforderlich
 insert(k, p) into P ;

else // Elternknoten teilen

$T \leftarrow P \cup (k, p)$;
 erase all entries from P ; // alter Elternknoten
 create new node P' ; // neuer Elternknoten

 copy $T.p_1$ through $T.p_{\lceil m/2 \rceil}$ into P ; // Verteile Einträge auf P
 copy $T.p_{\lceil m/2 \rceil + 1}$ through $T.p_{m+1}$ into P' ; // ...und P'

$k' \leftarrow T.k_{\lceil m/2 \rceil}$; // neuen Knoten P' mit Schlüssel k' in
 B+Tree-InsertInParent(P, k', P'); // ...Großelternknoten verlinken

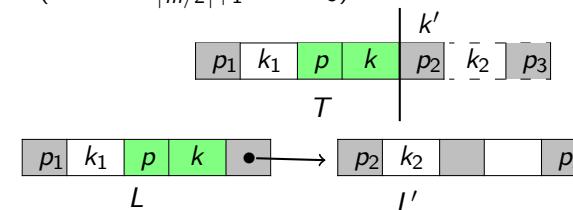
Blatt teilen/1

Kopiere L nach T und füge (k, p) ein: $p_1 \ k_1 \ p_2 \ k_2 \ p_3 \quad m = 3$

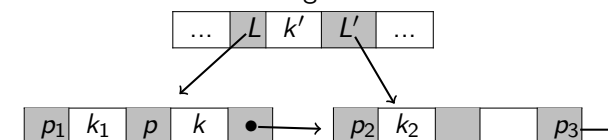
1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

T $p_1 \ k_1 \ p \ k \ p_2 \ k_2 \ p_3$

2. Teilen ($k' = T.k_{\lceil m/2 \rceil + 1} = T.k_3$)



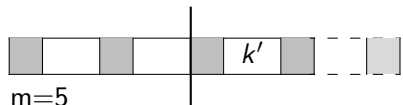
3. (k', L') in Elternknoten von L einfügen



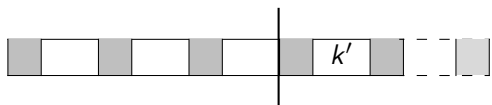
Blatt teilen/2

$$k' = T.k_{\lceil m/2 \rceil + 1}$$

- m gerade, z.B.: m=4



- m ungerade, z.B.: m=5



Innere Knoten teilen/1

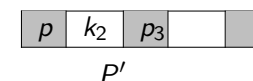
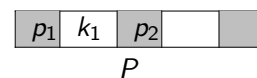
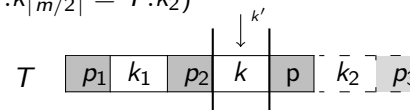
$$P \quad p_1 \quad k_1 \quad p_2 \quad k_2 \quad p_3$$

Kopiere P nach T und füge (k, p) ein:

1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

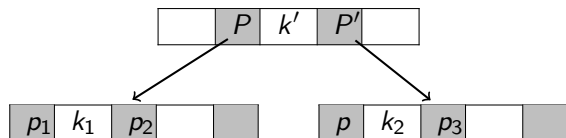
$$T \quad p_1 \quad k_1 \quad p_2 \quad k \quad p \quad k_2 \quad p_3$$

2. Teilen ($k' = T.k_{\lceil m/2 \rceil} = T.k_2$)



Innere Knoten teilen/2

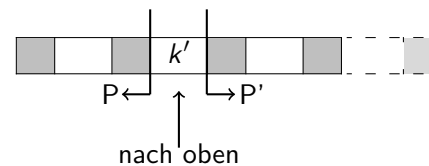
3. (k', P') in Elternknoten von P einfügen



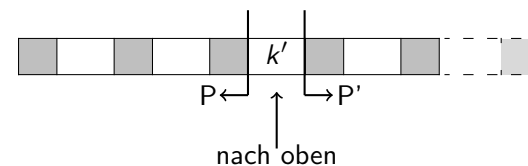
Innere Knoten teilen/3

$$k' = T.k_{\lceil m/2 \rceil}$$

- m gerade, z.B.: m=4

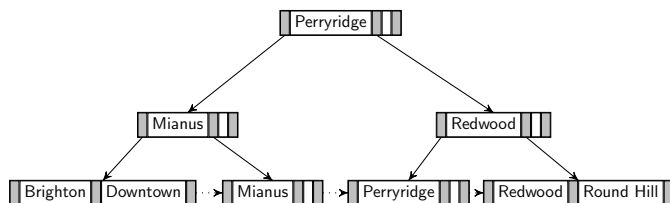


- m ungerade, z.B.: m=5

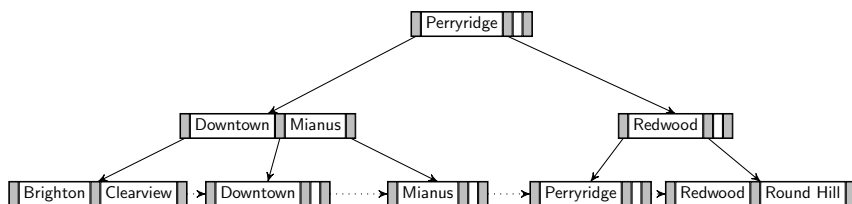


Beispiel: Einfügen in B^+ -Baum/1

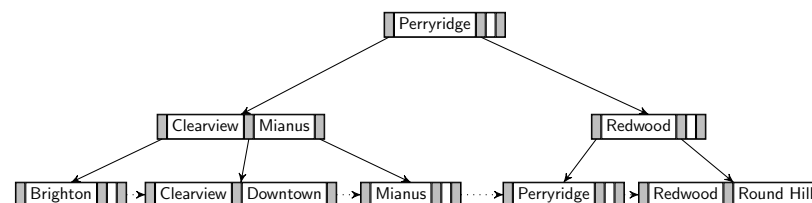
- B^+ -Baum vor Einfügen von *Clearview*



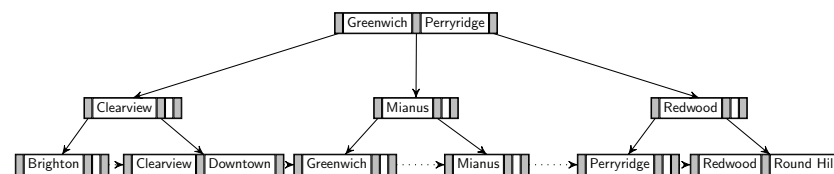
- B^+ -Baum nach Einfügen von *Clearview*

Beispiel: Einfügen in B^+ -Baum/2

- B^+ -Baum vor Einfügen von *Greenwich*



- B^+ -Baum nach Einfügen von *Greenwich*

Löschen von B^+ -Baum/1

Datensatz mit Suchschlüssel k löschen:

- finde Blattknoten mit Suchschlüssel k
- lösche k von Knoten
- falls** Knoten durch Löschen von k zu wenige Einträge hat:
 - Einträge im Knoten und einem Geschwisterknoten passen in 1 Knoten **dann**:
 - vereine** die beiden Knoten in einen einzigen Knoten (den linken, falls er existiert; ansonsten den rechten) und lösche den anderen Knoten
 - lösche den Eintrag im Elternknoten der zwischen den beiden Knoten ist und wende Löschen rekursiv an
 - Einträge im Knoten und einem Geschwisterknoten passen *nicht* in 1 Knoten **dann**:
 - verteile** die Einträge zwischen den beiden Knoten sodass beide die minimale Anzahl von Einträgen haben
 - aktualisiere den entsprechenden Suchschlüssel im Eltern-Knoten

Löschen von B^+ -Baum/2

- Vereinigung**:
 - Vereinigung zweier Knoten propagiert im Baum nach oben bis ein Knoten mit mehr als $\lceil m/2 \rceil$ Kindern gefunden wird
 - falls die Wurzel nach dem Löschen nur mehr ein Kind hat, wird sie gelöscht und der Kind-Knoten wird zur neuen Wurzel

Algorithmus: Löschen im B⁺-Baum

Algorithm 3: B+TreeDelete(L, k, p)

```

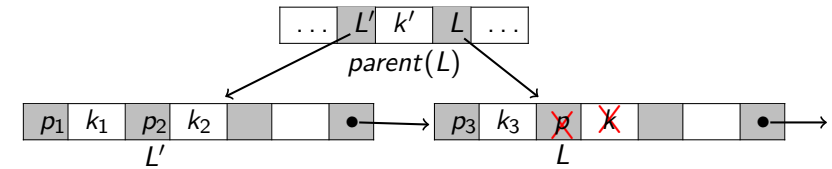
delete(k, p) from L
if L is root and has only one remaining child then
  └ make the child the new root and delete L
else if L has too few values/pointers then
  L' ← previous sibling of L [next, if there is no previous];
  k' ← value between L and L' in parent(L);
  if entries in L and L' can fit in a single node then // vereinigen
    if L is a predecessor of L' then swap L with L';
    if L is not a leaf then L' ← L' ∪ k' and all (ki, pi) from L;
    else L' ← L' ∪ all (ki, pi) from L;
    B+TreeDelete(parent(L), k', L);
  else // verteilen
    if L' is a predecessor of L then
      if L is a nonleaf node then
        remove the last (k, p) of L';
        insert the former last p of L' and k' as the first pointer and value in L;
      else move the last (p, k) of L' as the first pointer and value to L;
      replace k' in parent(L) by the former last k of L';
    else symmetric to the then case (switch first ↔ last,...);
  
```

Löschen aus Blatt/1

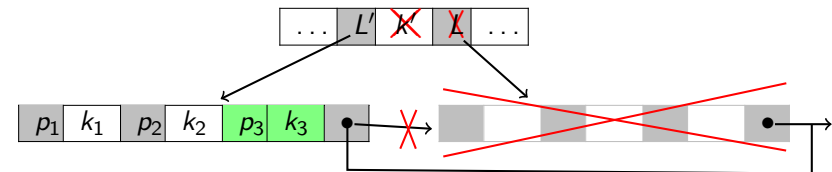
(k, p) wird aus L gelöscht:

1. Vereinigen (m = 4)

Vorher:



Nachher:

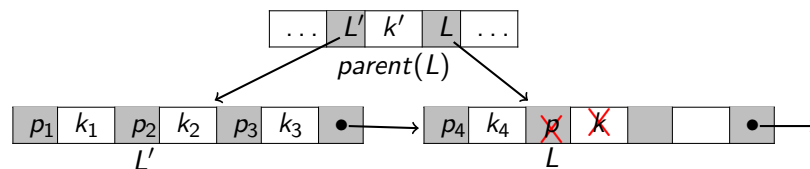


Löschen aus Blatt/2

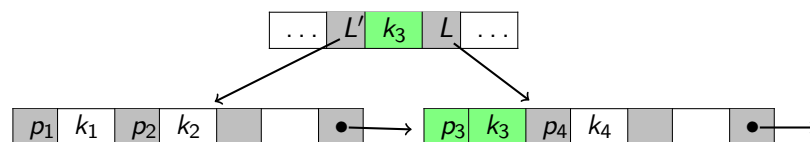
(k, p) wird aus L gelöscht:

2. Verteilen (m = 4)

Vorher:



Nachher:

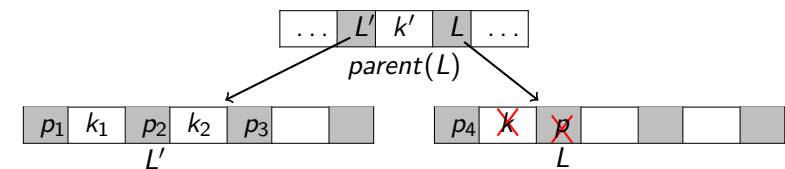


Löschen aus innerem Knoten/1

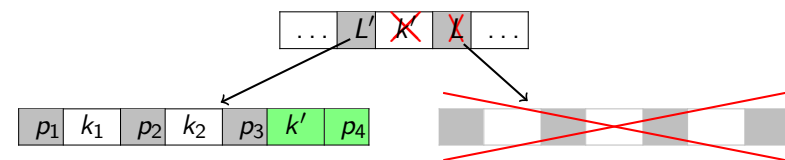
(k, p) wird aus L gelöscht:

1. Vereinigen (m = 4)

Vorher:



Nachher:

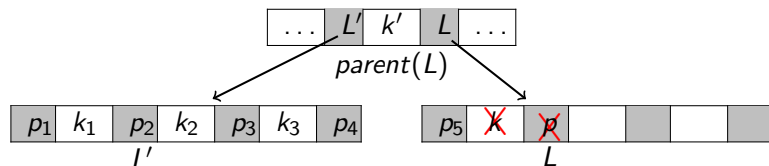


Löschen aus innerem Knoten/2

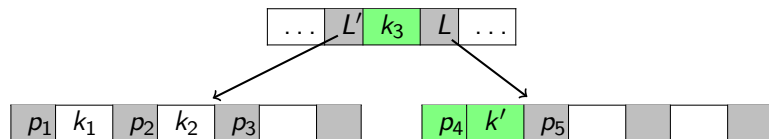
(k, p) wird aus L gelöscht:

2. Verteilen (m = 4)

Vorher:

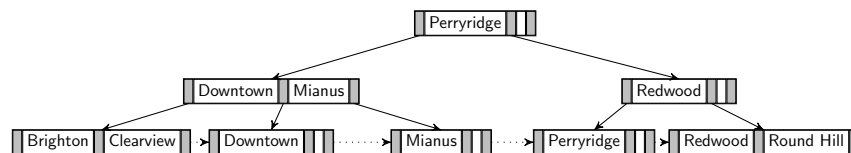


Nachher:

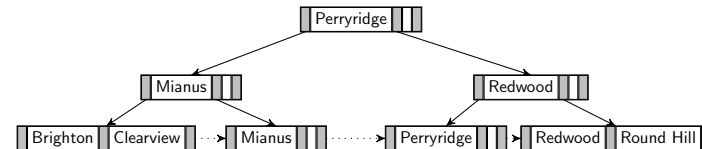


Beispiel: Löschen von B⁺-Baum/1

• Vor Löschen von *Downtown*:



• Nach Löschen von *Downtown*:

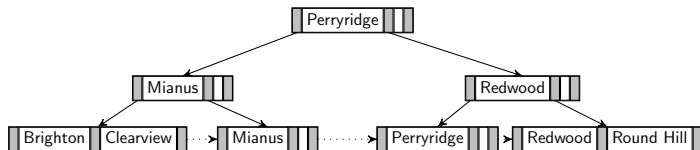


• Nach Löschen des Blattes mit *Downtown* hat der Elternknoten noch genug Pointer.

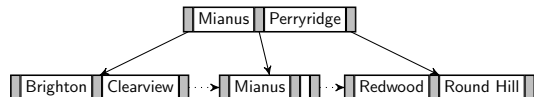
• Somit propagiert Löschen nicht weiter nach oben.

Beispiel: Löschen von B⁺-Baum/2

• Vor Löschen von *Perryridge*:



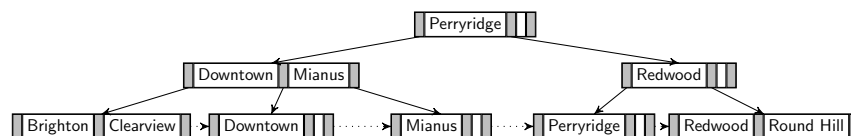
• Nach Löschen von *Perryridge*:



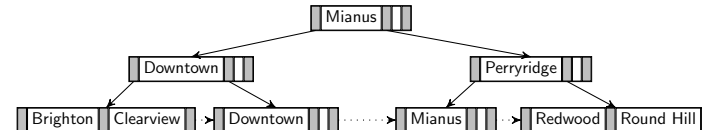
- Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und wird mit dem (rechten) Nachbarknoten **vereinigt**.
- Dadurch hat der Elternknoten zu wenig Pointer und wird mit seinem (linken) Nachbarknoten **vereinigt** (und ein Eintrag wird vom gemeinsamen Elternknoten gelöscht).
- Die Wurzel hat jetzt nur noch 1 Kind und wird gelöscht.

Beispiel: Löschen von B⁺-Baum/3

• Vor Löschen von *Perryridge*:



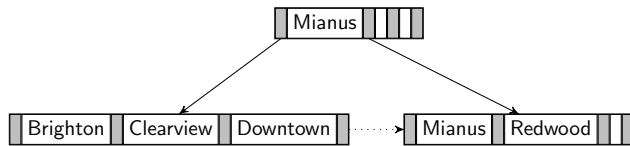
• Nach Löschen von *Perryridge*:



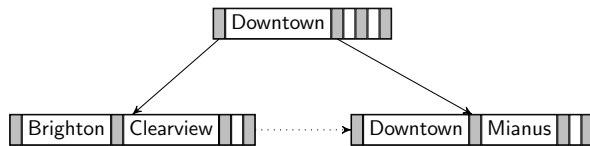
- Elternknoten von Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und erhält einen Pointer vom linken Nachbarn (**Verteilung** von Einträgen).
- Schlüssel im Elternknoten des Elternknotens (Wurzel in diesem Fall) ändert sich ebenfalls.

Beispiel: Löschen von B^+ -Baum/4

- Vor Löschen von *Redwood*:



- Nach Löschen von *Redwood*:



- Knoten von Blatt mit *Redwood* hat durch Löschen zu wenig Einträge und erhält einen Eintrag vom linken Nachbarn (*Verteilung* von Einträgen).
- Schlüssel im Elternknoten (Wurzel in diesem Fall) ändert sich ebenfalls.

Zusammenfassung B^+ -Baum

- **Knoten mit Pointern verknüpft:**
 - logisch nahe Knoten müssen nicht physisch nahe gespeichert sein
 - erlaubt mehr Flexibilität
 - erhöht die Anzahl der nicht-sequentiellen Zugriffe
- **B^+ -Bäume sind flach:**
 - maximale Tiefe $\lceil \log_{\lceil m/2 \rceil}(L) \rceil$ für L Blattknoten
 - m ist groß in der Praxis (z.B. $m = 200$)
- **Suchschlüssel als "Wegweiser":**
 - einige Suchschlüssel kommen als Wegweiser in einem oder mehreren inneren Knoten vor
 - zu einem Wegweiser gibt es nicht immer einen Suchschlüssel in einem Blattknoten (z.B. weil der entsprechende Datensatz gelöscht wurde)
- **Einfügen und Löschen sind effizient:**
 - nur $O(\log(K))$ viele Knoten müssen geändert werden
 - Index degeneriert nicht, d.h. Index muss nie von Grund auf rekonstruiert werden

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Statisches Hashing

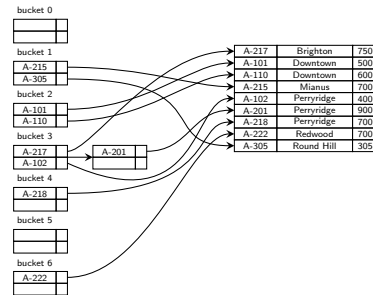
- **Nachteile von ISAM und B^+ -Baum Indizes:**
 - B^+ -Baum: Suche muss Indexstruktur durchlaufen
 - ISAM: binäre Suche in großen Dateien
 - das erfordert zusätzliche Zugriffe auf Plattenblöcke
- **Hashing:**
 - erlaubt es auf Daten direkt und ohne Indexstrukturen zuzugreifen
 - kann auch zum Bauen eines Index verwendet werden

Hash Index

- **Hash Index:** organisiert (Suchschlüssel,Pointer) Paare als Hash Datei
 - Pointer zeigt auf Datensatz
 - Suchschlüssel kann mehrfach vorkommen

- **Beispiel:** Index auf Konto-Relation

- Hash Funktion h : Quersumme der Kontonummer modulo 7
- Beachte: Konto-Relation ist nach Filialnamen geordnet



- Hash Index ist immer **Non-Clustered Index**:
 - ist deshalb immer "dense"
 - Primär- bzw. Clustered Hash Index entspricht einer Hash Datei Organisation (zusätzliche Index-Ebene überflüssig)

B^+ -Baum vs. Hash Index

- Hash Index degeneriert wenn es sehr viele identische (Hashwerte für) Suchschlüssel gibt – Overflows!
- Im Average Case für Punktanfragen in n Datensätzen:
 - Hash index: $O(1)$ (sehr gut)
 - B^+ -Baum: $O(\log n)$
- Worst Case für Punktanfragen in n Datensätzen:
 - Hash index: $O(n)$ (sehr schlecht)
 - B^+ -Baum: $O(\log n)$
- Anfragetypen:
 - Punktanfragen: Hash und B^+ -Baum
 - Mehrpunktanfragen: Hash und B^+ -Baum
 - Bereichsanfragen: Hash Index nicht brauchbar

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 **Mehrschlüssel Indizes**
- 5 Indizes in SQL

Zugriffe über mehrere Suchschlüssel/1

- Wie kann Index verwendet werden, um folgende Anfrage zu beantworten?


```
select AccNr
from account
where BranchName = "Perryridge" and Balance = 1000
```
- Strategien mit mehreren Indizes (jeweils 1 Suchschlüssel):
 - a) $BranchName = "Perryridge"$ mit Index auf $BranchName$ auswerten; auf Ergebnis-Datensätzen $Balance = 1000$ testen.
 - b) $Balance = 1000$ mit Index auf $Balance$ auswerten; auf Ergebnis-Datensätzen $BranchName = "Perryridge"$ testen.
 - c) Verwende $BranchName$ Index um Pointer zu Datensätzen mit $BranchName = "Perryridge"$ zu erhalten; verwende $Balance$ Index für Pointer zu Datensätzen mit $Balance = 1000$; berechne die Schnittmenge der beiden Pointer-Mengen.

Zugriffe über mehrere Suchschlüssel/2

- Nur die dritte Strategie nützt das Vorhandensein mehrerer Indizes.
- Auch diese Strategie kann eine schlechte Wahl sein:
 - es gibt viele Konten in der "Perryridge" Filiale
 - es gibt viele Konten mit Kontostand 1000
 - es gibt nur wenige Konten die beide Bedingungen erfüllen
- Effizientere Indexstrukturen müssen verwendet werden:
 - (traditionelle) Indizes auf kombinierten Schlüsseln
 - spezielle mehrdimensionale Indexstrukturen, z.B., Grid Files, Quad-Trees, Bitmap Indizes.

Zugriffe über mehrere Suchschlüssel/3

- Annahme: Geordneter Index mit kombiniertem Suchschlüssel (*BranchName*, *Balance*)
- Kombinierte Suchschlüssel haben eine **Ordnung** (*BranchName* ist das erstes Attribut, *Balance* ist das zweite Attribut)
 - Folgende Bedingung wird effizient behandelt (alle Attribute):
where *BranchName* = "Perryridge" **and** *Balance* = 1000
 - Folgende Bedingung wird effizient behandelt (Prefix):
where *BranchName* = "Perryridge"
 - Folgende Bedingung ist ineffizient (kein Prefix der Attribute):
where *Balance* = 1000

Inhalt

- 1 Grundlagen
- 2 Sequentielle Indextypen
 - ISAM Index
 - B^+ -Baum
- 3 Hash Index
- 4 Mehrschlüssel Indizes
- 5 Indizes in SQL

Index Definition in SQL

- SQL-92 definiert keine Syntax für Indizes da diese nicht Teil des logischen Datenmodells sind.
- Jedoch alle Datenbanksysteme stellen Indizes zur Verfügung.
- Index erzeugen:
create index <IdxName> **on** <RelName> (<AttrList>)
z.B. **create index** BrNaldx **on** branch (branch-name)
- **Create unique index** erzwingt eindeutige Suchschlüssel und definiert indirekt ein Schlüsselattribut.
- Primärschlüssel (**primary key**) und Kandidatenschlüssel (**unique**) werden in SQL bei der Tabellendefinition spezifiziert.
- Index löschen:
drop index <index-name>
z.B. **drop index** BrNaldx

Beispiel: Indizes in PostgreSQL

- **CREATE [UNIQUE] INDEX** name **ON** table_name
"(" col [DESC] { "," col [DESC] } ")" [...]
- **Beispiele:**
 - **CREATE INDEX** MajIdx **ON** Enroll (Major);
 - **CREATE INDEX** MajIdx **ON** Enroll **USING HASH** (Major);
 - **CREATE INDEX** MajMinIdx **ON** Enroll (Major, Minor);

Indexes in Oracle

- **B⁺-Baum Index in Oracle:**
CREATE [UNIQUE] INDEX name **ON** table_name
"(" col [DESC] { "," col [DESC] } ")" [**pctfree** n] [...]
- **Anmerkungen:**
 - pct_free gibt an, wieviel Prozent der Knoten anfangs frei sein sollen.
 - UNIQUE sollte nicht verwendet werden, da es ein logisches Konzept ist.
 - Oracle erstellt einen B⁺-Baum Index für jede **unique** oder **primary key** definition bei der Erstellung der Tabelle.
- **Beispiele:**
CREATE TABLE BOOK (
ISBN **INTEGER**, Author **VARCHAR2 (30)**, ...);
CREATE INDEX book_auth **ON** book(Author);

Anmerkungen zu Indizes in Datenbanksystemen

- Indizes werden **automatisch nachgeführt** wenn Tupel eingefügt, geändert oder gelöscht werden.
- Indizes **verlangsamen** deshalb Änderungsoperationen.
- Einen **Index zu erzeugen** kann lange dauern.
- **Bulk Load:** Es ist (viel) effizienter, zuerst die Daten in die Tabelle einzufügen und nachher alle Indizes zu erstellen als umgekehrt.

Zusammenfassung

- **Index Typen:**
 - Clustering vs. Non-Clustered Index
 - Dense oder Sparse
- **B⁺-Baum:**
 - universelle Indexstruktur, auch für Bereichsanfragen
 - Garantien zu Tiefe, Füllgrad und Effizienz
 - Einfügen und Löschen
- **Hash Index:**
 - statisches und erweiterbares Hashing
 - kein Index für Primärschlüssel nötig
 - gut für Prädikate mit "="
- **Mehrschlüssel Indizes:** schwieriger, da es keine totale Ordnung in mehreren Dimensionen gibt
- Indizes in **SQL**