Datenbanken 2

Einführung, Physische Datenorganisation

Nikolaus Augsten

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

database
research group

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 9. Oktober 2018

Inhalt

Inhalt

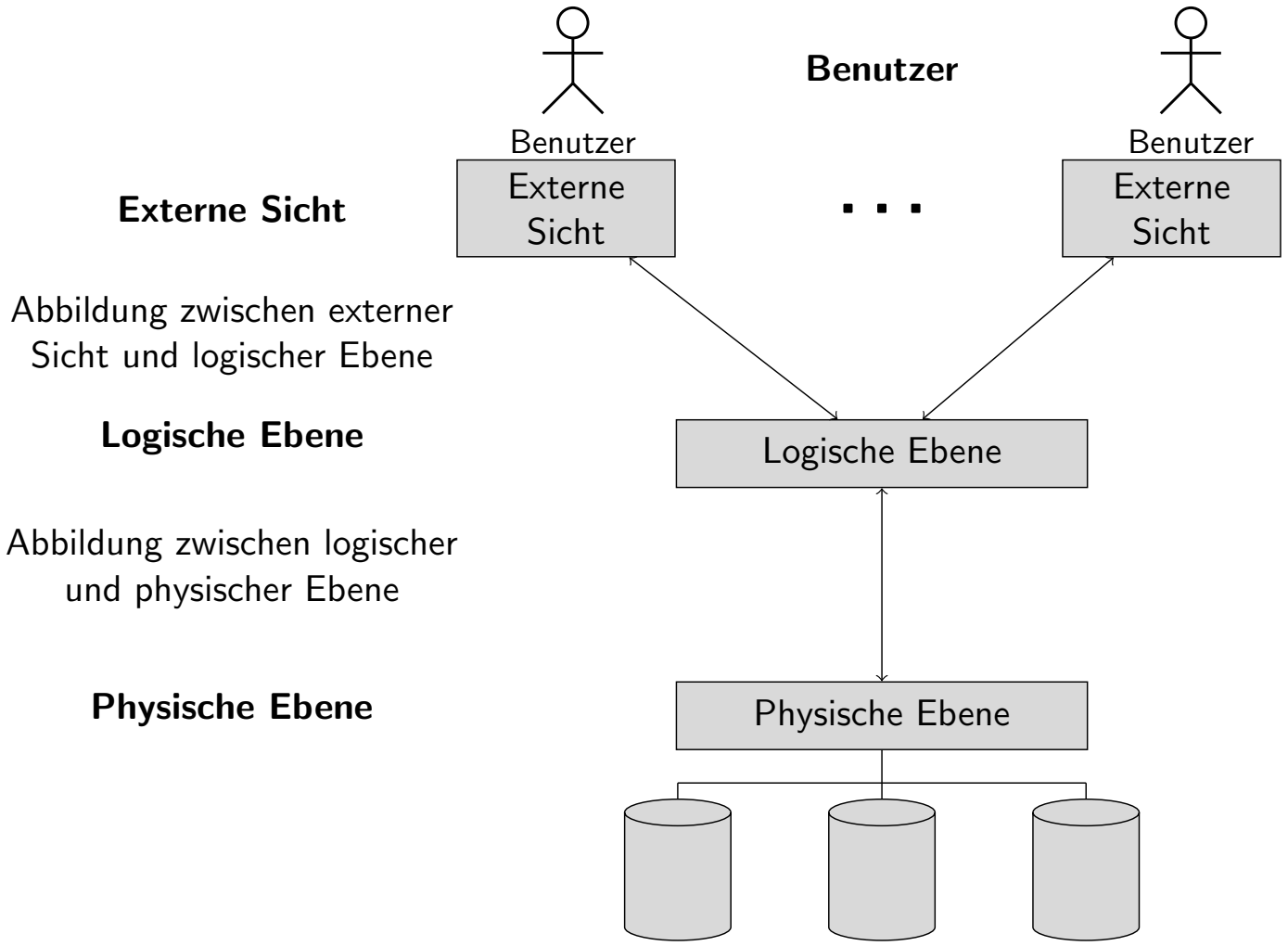
Alle Infos zu Vorlesung und Proseminar:

http://dbresearch.uni-salzburg.at/teaching/2018ws/db2/

Was erwartet Sie inhaltlich?

- **Datenbanken 1**: Logische Ebene
  - Konzeptioneller Entwurf (ER)
  - Relationale Algebra
  - SQL
  - Relationale Entwurfstheorie
- **Datenbanken 2**: Physische Ebene
  - Wie baue (programmiere) ich ein Datenbanksystem?
    - Daten müssen physisch gespeichert werden
    - **Datenstrukturen** und Zugriffs-**Algorithmen** müssen gefunden werden
    - SQL-Anfragen müssen in ausführbare Programme umgesetzt werden
  - Es geht um **Effizienz** (schneller ist besser)

Die ANSI/SPARC Drei-Ebenen Architektur



Inhaltsübersicht Datenbanksysteme

1. **Physische Datenorganisation**
   - Speichermedien, Dateiorganisation
   - Kapitel 7 in Kemper und Eickler
   - Chapter 10 in Silberschatz et al.
2. **Indexstrukturen**
   - Sequentielle Dateien, B^+ Baum, Statisches Hashing, Dynamisches Hashing, Mehrere Suchschlüssel, Indizes in SQL
   - Kapitel 7 in Kemper und Eickler
   - Chapter 11 in Silberschatz et al.
3. **Anfragebearbeitung**
   - Effiziente Implementierung der (relationalen) Operatoren
   - Kapitel 8 in Kemper und Eickler
   - Chapter 12 in Silberschatz et al.
4. **Anfrageoptimierung**
   - Äquivalenzregeln und Äquivalenzumformungen, Join Ordnungen
   - Kapitel 8 in Kemper und Eickler
   - Chapter 13 in Silberschatz et al.

Silberschatz 1/1/6
Cp 10+11 [Referenz]

Inhalt

Speichermedien/1

- Verschiedene Arten von Speichermedien sind für Datenbanksysteme relevant.
- Speichermedien lassen sich in Speicherhierarchie anordnen.
- Klassifizierung der Speichermedien nach:
  - Zugriffsgeschwindigkeit
  - Kosten pro Dateneinheit
  - Verlässlichkeit
    - Datenverlust durch Stromausfall oder Systemabsturz
    - Physische Fehler des Speichermediums
  - Flüchtige vs. persistente Speicher
    - Flüchtig (volatile): Inhalt geht nach Ausschalten verloren
    - Persistent (non-volatile): Inhalt bleibt auch nach Ausschalten

Speichermedien/2

- Cache
  - flüchtig
  - am schnellsten und am teuersten
  - von System Hardware verwaltet

- Hauptspeicher (RAM)
  - flüchtig
  - schneller Zugriff (x0 bis x00 ns; 1 ns $= 10^{-9}$s)
  - meist zu klein (oder zu teuer) um gesamte Datenbank zu speichern
    - mehrere GB weit verbreitet
    - Preise derzeit ca. 7.5 EUR/GB (DRAM)

Speichermedien/3

- Flash memory (SSD)
  - persistent
  - lesen ist sehr schnell (x0 bis x00 μs; 1 μs $= 10^{-6}$s)
  - hohe sequentielle Datentransferrate (bis 500 MB/s)
  - nicht-sequentieller Zugriff nur ca. 25% langsamer
  - Schreibzugriff langsamer und komplizierter
    - Daten können nicht überschrieben werden, sondern müssen zuerst gelöscht werden
    - nur beschränkte Anzahl von Schreib/Lösch-Zyklen sind möglich
  - Preise derzeit ca. 0.2 EUR/GB
  - Speichermedien: NAND Flash Technologie (Firmware: auch NOR)
  - weit verbreitet in Embedded Devices (z.B. Digitalkamera)
  - auch als EEPROM bekannt (Electrically Erasable Programmable Read-Only Memory)

Speichermedien/4

- Festplatte
  - persistent
  - Daten sind auf Magnetscheiben gespeichert, mechanische Drehung
  - sehr viel langsamer als RAM (Zugriff im ms-Bereich; 1 ms $= 10^{-3}$s)
  - sequentielles Lesen: 25–100 MB/s
  - billig: Preise teils unter 30 EUR/TB
  - sehr viel mehr Platz als im Hauptspeicher; derzeit x00 GB - 14 TB
  - Kapazitäten stark ansteigend (Faktor 2 bis 3 alle 2 Jahre)
  - Hauptmedium für Langzeitspeicher: speichert gesamte Datenbank
  - für den Zugriff müssen Daten von der Platte in den Hauptspeicher geladen werden
  - direkter Zugriff, d.h., Daten können in beliebiger Reihenfolge gelesen werden
  - Diskette vs. Festplatte

Speichermedien/5

- Optische Datenträger
  - persistent
  - Daten werden optisch via Laser von einer drehenden Platte gelesen
  - lesen und schreiben langsamer als auf magnetischen Platten
  - sequentielles Lesen: 1 Mbit/s (CD) bis 400 Mbit/s (Blu-ray)
  - verschiedene Typen:
    - CD-ROM (640 MB), DVD (4.7 bis 17 GB), Blu-ray (25 bis 129 GB)
    - write-once, read-many (WORM) als Archivspeicher verwendet
    - mehrfach schreibbare Typen vorhanden (CD-RW, DVD-RW, DVD-RAM)
  - Jukebox-System mit austauschbaren Platten und mehreren Laufwerken sowie einem automatischen Mechanismus zum Platten wechseln – "CD-Wechsler" mit hunderten CD, DVD, oder Blu-ray disks

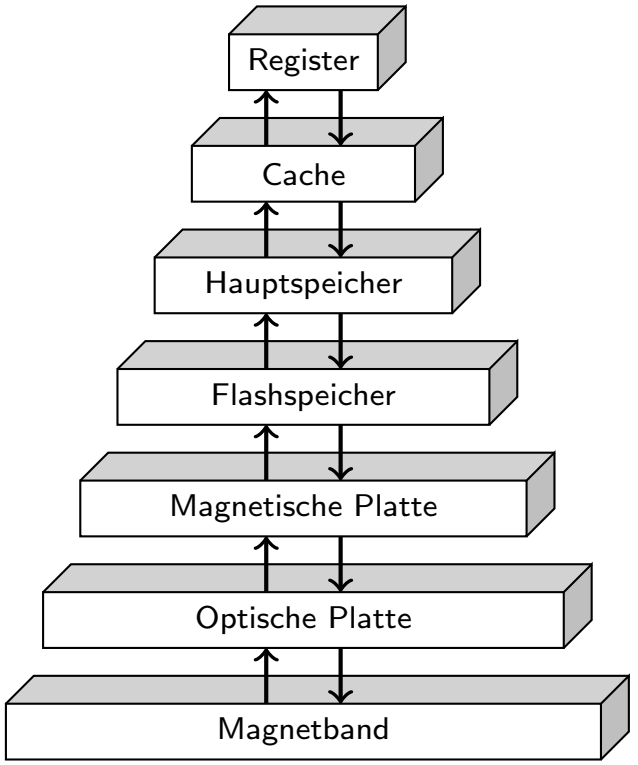
Speichermedien/6

- Band
  - persistent
  - Zugriff sehr langsam, da sequentieller Zugriff
  - Datentransfer jedoch z.T. wie Festplatte (z.B. 120 MB/s, komprimiert 240MB/s)
  - sehr hohe Kapazität (mehrere TB)
  - sehr billig (ab 10 EUR/TB)
  - hauptsächlich für Backups genutzt
  - Band kann aus dem Laufwerk genommen werden
  - Band Jukebox für sehr große Datenmengen
    - x00 TB (1 terabyte = 10^{12} bytes) bis Petabyte (1 petabyte = 10^{15} bytes)

Speichermedien/7

- Speichermedien können hierarchisch nach Geschwindigkeit und Kosten geordnet werden:

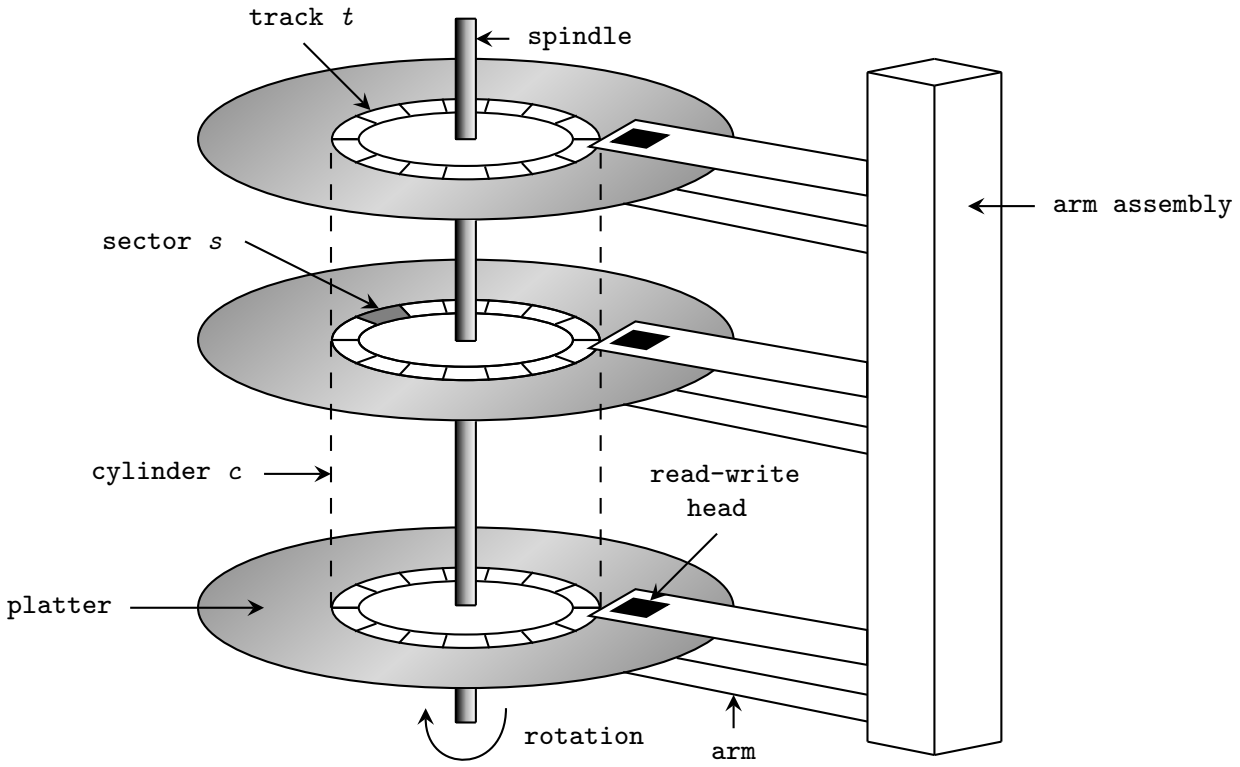
- Primärspeicher: flüchtig, schnell, teuer
  - z. B. Cache, Hauptspeicher
- Sekundärspeicher: persistent, langsamer und günstiger als Primärspeicher
  - z. B. Magnetplatten, Flash Speicher
  - auch Online-Speicher genannt
- Tertiärspeicher: persistent, sehr langsam, sehr günstig
  - z. B. Magnetbänder, optischer Speicher
  - auch Offline-Speicher genannt



- Datenbank muss mit Speichermedien auf allen Ebenen umgehen

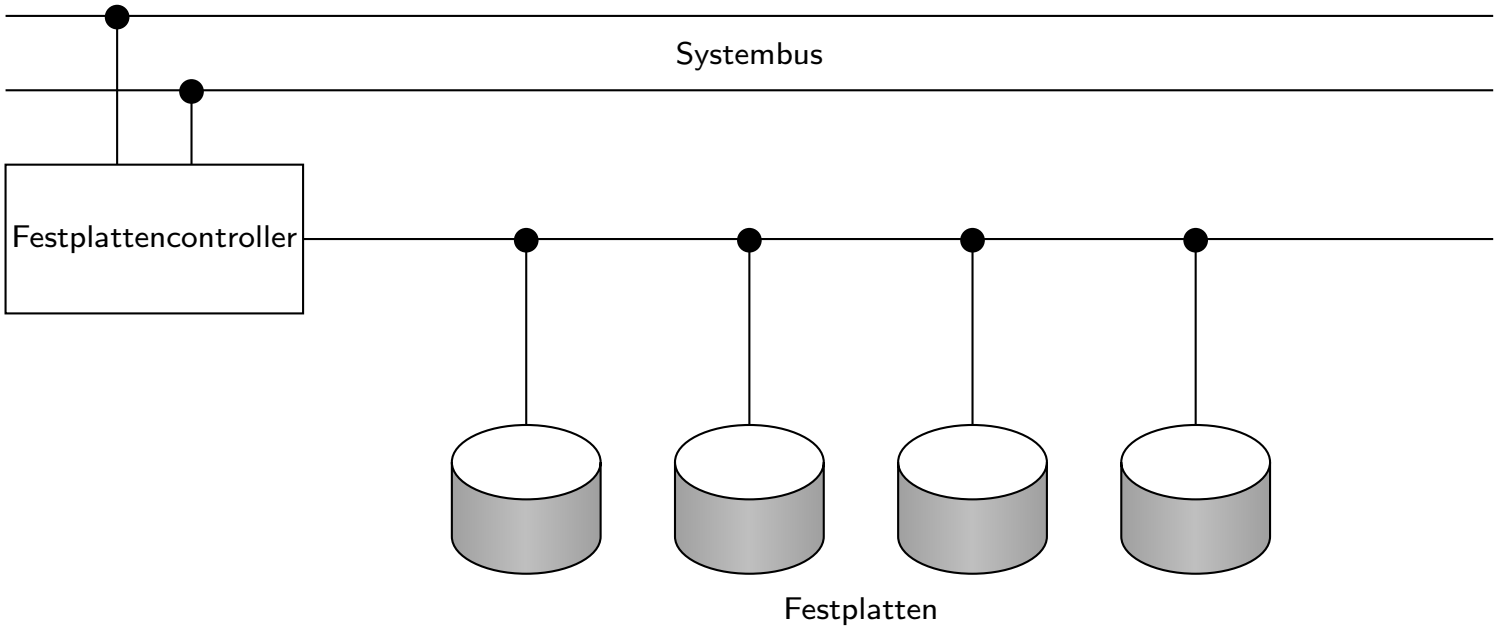
Festplatten/1

- Meist sind Datenbanken auf magnetischen Platten gespeichert, weil:
  - die Datenbank zu groß für den Hauptspeicher ist
  - der Plattenspeicher persistent ist
  - Plattenspeicher billiger als Hauptspeicher ist
- Schematischer Aufbau einer Festplatte:



Festplatten/2

- Controller: Schnittstelle zwischen Computersystem und Festplatten:
  - übersetzt high-level Befehle (z.B. bestimmten Sektor lesen) in Hardware Aktivitäten (z.B. Disk Arm bewegen und Sektor lesen)
  - für jeden Sektor wird Checksum geschrieben
  - beim Lesen wird Checksum überprüft



Festplatten/3

Drei Arbeitsvorgänge für Zugriff auf Festplatte:

- Spurwechsel (seek time): Schreib-/Lesekopf auf richtige Spur bewegen
- Latenz (rotational latency): Warten bis sich der erste gesuchte Sektor unter dem Kopf vorbeibewegt
- Lesezeit: Sektoren lesen/schreiben, hängt mit Datenrate (data transfer rate) zusammen

$$\text{Zugriffszeit} = \text{Spurwechsel} + \text{Latenz} + \text{Lesezeit}$$

Festplatten/4

Performance Parameter von Festplatten

- Spurwechsel: gerechnet wird mit mittlerer Seek Time (=1/2 worst case seek time, typisch 2-10ms)
- Latenz:
  - errechnet sich aus Drehzahl (5400rpm-15000rpm)
  - rpm = revolutions per minute
  - Latenz [s] = 60 / Drehzahl [rpm]
  - mittlere Latenz: 1/2 worst case (2ms-5.5ms)
- Datenrate: Rate mit der Daten gelesen/geschrieben werden können (z.B. 25-100 MB/s)
- Mean time to failure (MTTF): mittlere Laufzeit bis zum ersten Mal ein Hardware-Fehler auftritt
  - typisch: mehrere Jahre
  - keine Garantie, nur statistische Wahrscheinlichkeit

Festplatten/5

- Block: (auch "Seite") zusammenhängende Reihe von Sektoren auf einer bestimmten Spur
- Interblock Gaps: ungenützter Speicherplatz zwischen Sektoren
- ein Block ist eine logische Einheit für den Zugriff auf Daten.
  - Daten zwischen Platte und Hauptspeicher werden in Blocks übertragen
  - Datenbank-Dateien sind in Blocks unterteilt
  - Block Größen: 4-16 kB
    - kleine Blocks: mehr Zugriffe erforderlich
    - große Blocks: Ineffizienz durch nur teilweise gefüllte Blocks

Integrierte Übung 1.1

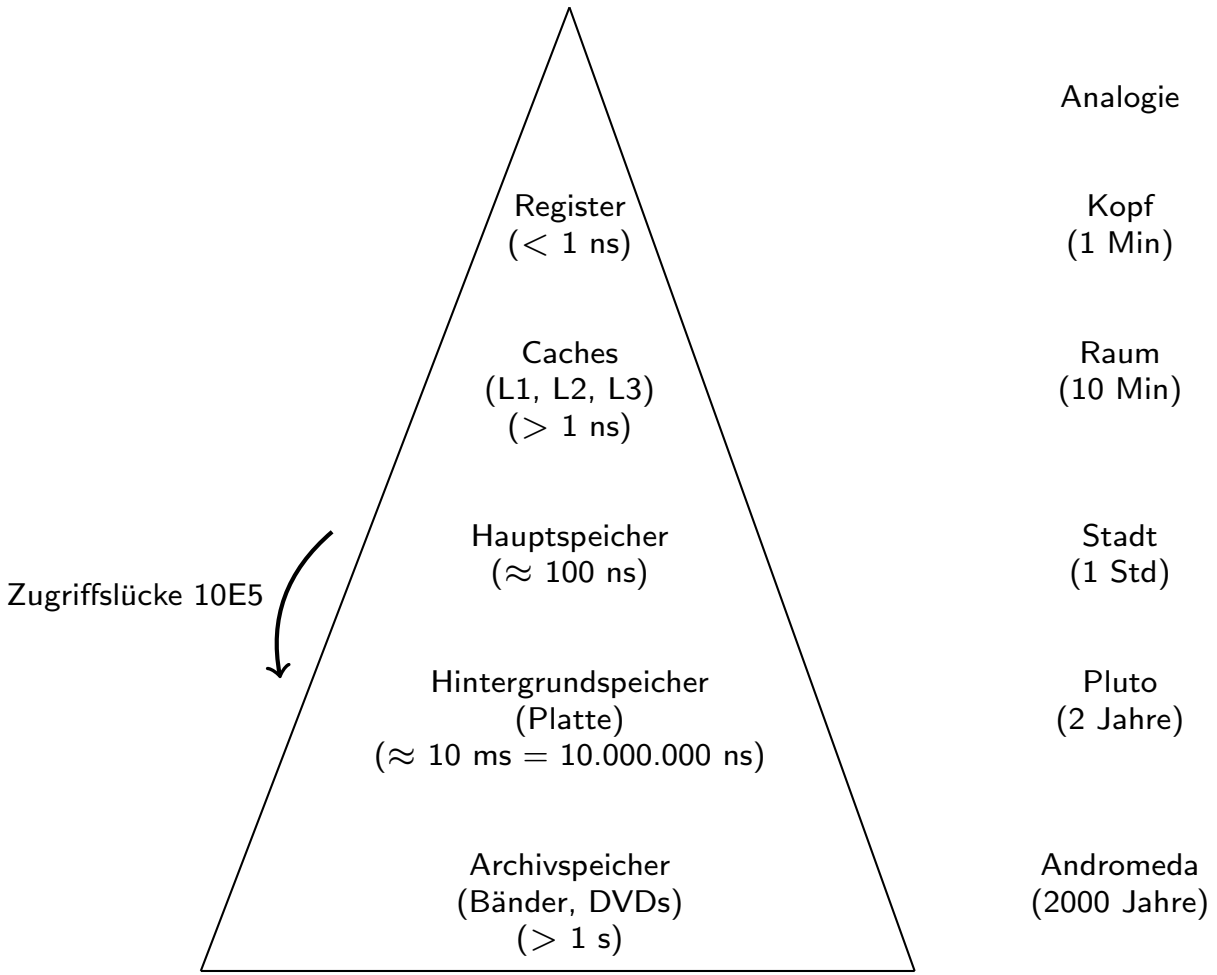
Betrachte folgende Festplatte: Sektor-Größe $B = 512$ Bytes, Sektoren/Spur $S = 20$, Spuren pro Scheibenseite $T = 400$, Anzahl der beidseitig beschriebenen Scheiben $D = 15$, mittlerer Spurwechsel $sp = 30ms$, Drehzahl $dz = 2400rpm$ (Interblock Gaps werden vernachlässigt).

Bestimme die folgenden Werte:

a) Kapazität der Festplatte
b) mittlere Zugriffszeit (1 Sektor lesen)

Inhalt

Speicherhierarchie



Platten Zugriff Optimieren

- Wichtiges Ziel von DBMSs: Transfer von Daten zwischen Platten und Hauptspeicher möglichst effizient gestalten.
  - optimieren/minimieren der Anzahl der Zugriffe
  - minimieren der Anzahl der Blöcke
  - so viel Blöcke als möglich im Hauptspeicher halten ($\rightarrow$ Puffer Manager)
- Techniken zur Optimierung des Block Speicher Zugriffs:
  1. Disk Arm Scheduling
  2. Geeignete Dateistrukturen
  3. Schreib-Puffer und Log Disk

Block Speicher Zugriff/1

- Disk Arm Scheduling: Zugriffe so ordnen, dass Bewegung des Arms minimiert wird.
- Elevator Algorithm (Aufzug-Algorithmus):
  - Disk Controller ordnet die Anfragen nach Spur (von innen nach außen oder umgekehrt)
  - Bewege Arm in eine Richtung und erledige alle Zugriffe unterwegs bis keine Zugriffe mehr in diese Richtung vorhanden sind
  - Richtung umkehren und die letzten beiden Schritte wiederholen

Block Speicher Zugriff/2

- Datei Organization: Daten so in Blöcken speichern, wie sie später zugegriffen werden.
  - z.B. verwandte Informationen auf benachbarten Blöcken speichern
- Fragmentierung: Blöcke einer Datei sind nicht hintereinander auf der Platte abgespeichert
  - Gründe für Fragmentierung sind z.B.
    - Daten werden eingefügt oder gelöscht
    - die freien Blöcke auf der Platte sind verstreut, d.h., auch neue Dateien sind schon zerstückelt
  - sequentieller Zugriff auf fragmentierte Dateien erfordert erhöhte Bewegung des Zugriffsarms
  - manche Systeme erlauben das Defragmentieren des Dateisystems

Block Speicher Zugriff/3

Schreibzugriffe können asynchron erfolgen um Throughput (Zugriffe/Sekunde) zu erhöhen

- Persistente Puffer: Block wird zunächst auf persistenten RAM (RAM mit Batterie-Backup oder Flash Speicher) geschrieben; der Controller schreibt auf die Platte, wenn diese gerade nicht beschäftigt ist oder der Block zu lange im Puffer war.
  - auch bei Stromausfall sind Daten sicher
  - Schreibzugriffe können geordnet werden um Bewegung des Zugriffsarms zu minimieren
  - Datenbank Operationen, die auf sicheres Schreiben warten müssen, können fortgesetzt werden
- Log Disk: Eine Platte, auf die der Log aller Schreibzugriffe sequentiell geschrieben wird
  - wird gleich verwendet wie persistenter RAM
  - Log schreiben ist sehr schnell, da kaum Spurwechsel erforderlich
  - erfordert keine spezielle Hardware

Puffer Manager/1

- Puffer: Hauptspeicher-Bereich für Kopien von Platten-Blöcken
- Puffer Manager: Subsystem zur Verwaltung des Puffers
  - Anzahl der Platten-Zugriffe soll minimiert werden
  - ähnlich der virtuellen Speicherverwaltung in Betriebssystemen

Puffer Manager/2

- Programm fragt Puffer Manager an, wenn es einen Block von der Platte braucht.
- Puffer Manager Algorithmus:
  1. Programm fordert Plattenblock an.
  2. Falls Block nicht im Puffer ist:
     - Der Puffer Manager reserviert Speicher im Puffer (wobei nötigenfalls andere Blöcke aus dem Puffer geworfen werden).
     - Ein rausgeworfener Block wird nur auf die Platte geschrieben, falls er seit dem letzten Schreiben auf die Platte geändert wurde.
     - Der Puffer Manager liest den Block von der Platte in den Puffer.
  3. Der Puffer Manager gibt dem anfordernden Programm die Hauptspeicheradresse des Blocks im Puffer zurück.
- Es gibt verschiedene Strategien zum Ersetzen von Blöcken im Puffer.

Ersetzstrategien für Pufferseiten/1

- LRU Strategie (least recently used): Ersetze Block der am längsten nicht benutzt wurde.
  - Idee: Zugriffsmuster der Vergangenheit benutzten um zukünfiges Verhalten vorherzusagen
  - erfolgreich in Betriebssystemen eingesetzt
- MRU Strategie: (most recently used): Ersetze zuletzt benutzten Block als erstes.
  - LRU kann schlecht für bestimmte Zugriffsmuster in Datenbanken sein, z.B. wiederholtes Scannen von Daten
- Anfragen in DBMSs haben wohldefinierte Zugriffsmuster (z.B. sequentielles Lesen) und das DBMS kann die Information aus den Benutzeranfragen verwenden, um zukünftig benötigte Blöcke vorherzusagen.

LRU / MRU.

Ersetzstrategien für Pufferseiten/2

- Pinned block: Darf nicht aus dem Puffer entfernt werden.
  - z.B. der R-Block, bevor alle Tupel von S bearbeitet sind
- Toss Immediate Strategy: Block wird sofort rausgeworfen, wenn das letzte Tupel bearbeitet wurde.
  - z.B. der R Block sobald das letzte Tupel von S bearbeitet wurde
- Gemischte Strategie mit Tipps vom Anfrageoptimierer ist am erfolgreichsten.

Ersetzstrategien für Pufferseiten/3

- Beispiel: Berechne Join mit Nested Loops

```
für jedes Tupel tr von R:
  für jedes Tupel ts von S:
    wenn ts und tr das Join-Prädikat erfüllen, dann ...
```

- Verschiedene Zugriffsmuster für R und S
  - ein R-Block wird nicht mehr benötigt, sobald das letzte Tupel des Blocks bearbeitet wurde; er sollte also sofort entfernt werden, auch wenn er gerade erst benutzt worden ist
  - ein S-Block wird nochmal benötigt, wenn alle anderen S-Blöcke abgearbeitet sind

Integrierte Übung 1.2

Zwischen R (2 Blöcke) und S (3 Blöcke) soll einen Nested Loop Join ausgeführt werden. Jeder Block enthält nur 1 Tupel.

Der Puffer fasst 3 Blöcke.

Betrachte den Puffer während des Joins und zähle die Anzahl der geladenen Blöcke für folgende Puffer-Strategien:

- LRU
- MRU + Pinned Block (für aktuellen Block von R)
- MRU + Pinned Block (für aktuellen Block von R) + Toss Immediate (für abgearbeiteten Block von R)

Welche Strategie eignet sich besser?

Ersetzstrategien für Pufferseiten/4

Informationen für Ersatzstrategien in DBMSs:

- Zugriffspfade haben wohldefinierte Zugriffsmuster (z.B. sequentielles Lesen)
- Information im Anfrageplan um zukünftige Blockanfragen vorherzusagen
- Statistik über die Wahrscheinlichkeit, dass eine Anfrage für eine bestimmte Relation kommt
  - z.B. das Datenbankverzeichnis (speichert Schema) wird oft zugegriffen
  - Heuristik: Verzeichnis im Hauptspeicher halten

Inhalt

Datei Organisation

- Datei: (file) aus logischer Sicht eine Reihe von Datensätzen
  - ein *Datensatz* (record) ist eine Reihe von Datenfeldern
  - mehrere Datensätze in einem Platten-Block
  - *Kopfteil* (header): Informationen über Datei (z.B. interne Organisation)
- Abbildung von Datenbank in Dateien:
  - eine Relation wird in eine Datei gespeichert
  - ein Tupel entspricht einem Datensatz in der Datei
- Cooked vs. raw files:
  - cooked: DBMS verwendet Dateisystem des Betriebssystems (einfacher, code reuse)
  - raw: DBMS verwaltet Plattenbereich selbst (unabhängig von Betriebssystem, bessere Performance, z.B. Oracle)
- Fixe vs. variable Größe von Datensätzen:
  - fix: einfach, unflexibel, Speicher-ineffizient
  - variabel: komplizierter, flexibel, Speicher-effizient

Fixe Datensatzlänge/1

- Speicheradresse: i-ter Datensatz wird ab Byte $m * (i - 1)$ gespeichert, wobei m die Größe des Datensatzes ist
- Datensätze an der Blockgrenze:
  - *überlappend:* Datensätze werden an Blockgrenze geteilt (zwei Blockzugriffe für geteilten Datensatz erforderlich)
  - *nicht-überlappend:* Datensätze dürfen Blockgrenze nicht überschreiten (freier Platz am Ende des Blocks bleibt ungenutzt)
- mehrere Möglichkeiten zum Löschen des i-ten Datensatzes:

  (a) verschiebe Datensätze $i + 1, ..., n$ nach $i, ..., n - 1$

  (b) verschiebe letzten Datensatz im Block nach i

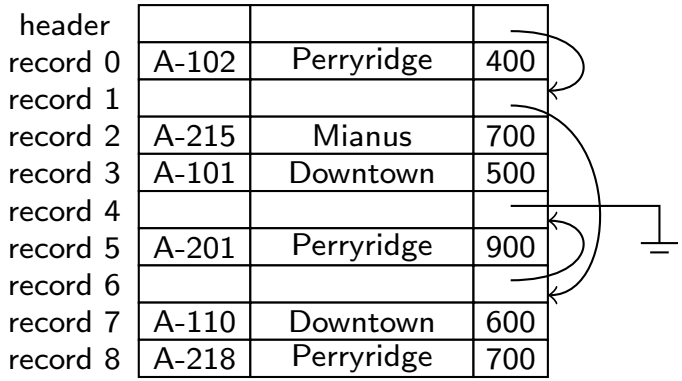
  (c) nicht verschieben, sondern "Free List" verwalten

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixe Datensatzlänge/2

- Free List:
  - speichere Adresse des ersten freien Datensatzes im Kopfteil der Datei
  - freier Datensatz speichert Pointer zum nächsten freien Datensatz

  → der Speicherbereich des gelöschten Datensatzes wird für Free List Pointer verwendet
- Beispiel: Free List nach löschen der Datensätze 4, 6, 1 (in dieser Reihenfolge)

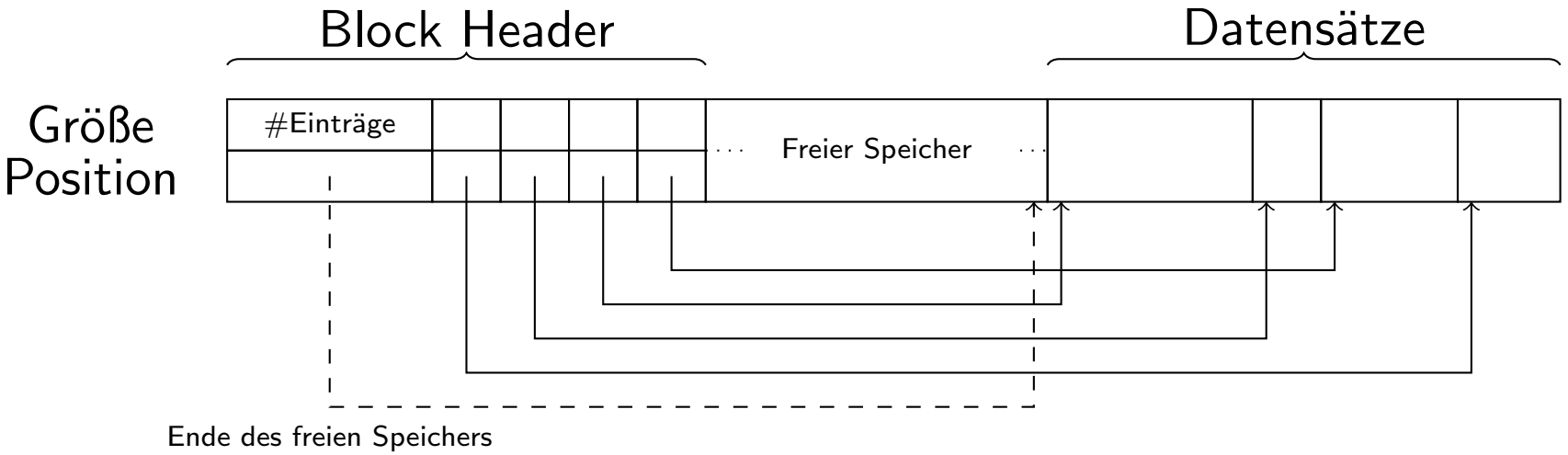


Variable Datensatzlänge/1

- Warum Datensätze mit variabler Größe?
  - Datenfelder variabler Länge (z.B., VARCHAR)
  - verschiedene Typen von Datensätzen in einer Datei
  - Platz sparen: z.B. in Tabellen mit vielen null-Werten (häufig in der Praxis)
- Datensätze verschieben kann erforderlich werden:
  - Datensätze können größer werden und im vorgesehenen Speicherbereich nicht mehr Platz haben
  - neue Datensätze werden zwischen existierenden Datensätzen eingefügt
  - Datensätze werden gelöscht (leere Zwischenräume verhindern)
- Pointer soll sich nicht ändern:
  - alle existierenden Referenzen zum Datensatz müssten geändert werden
  - das wäre kompliziert und teuer
- Lösung: Slotted Pages (TID-Konzept)

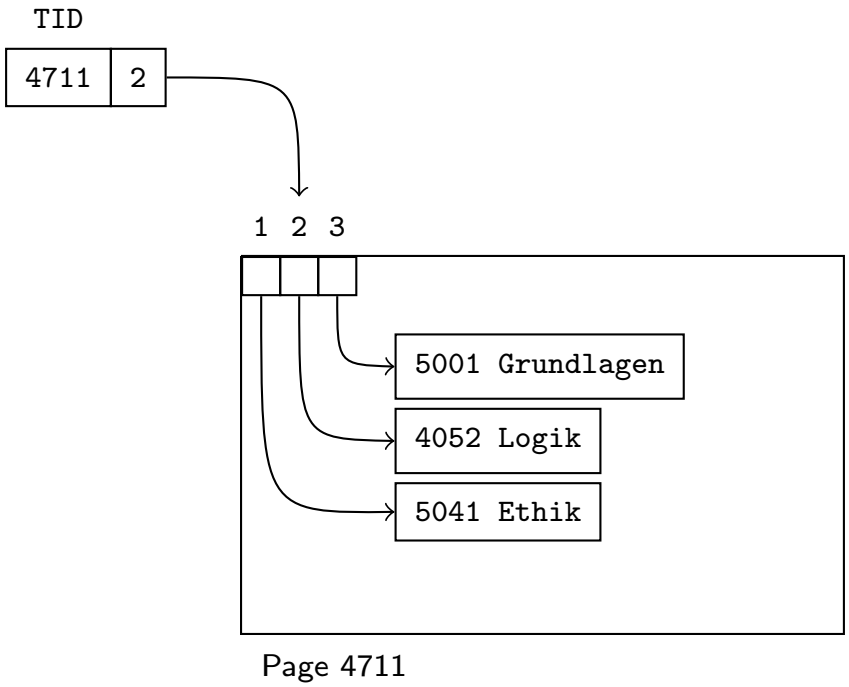
Slotted Pages/1

- Slotted Page:
  - Kopfteil (header)
  - freier Speicher
  - Datensätze
- Kopfteil speichert:
  - Anzahl der Datensätze
  - Ende des freien Speichers
  - Größe und Pointer auf Startposition jedes Datensatzes



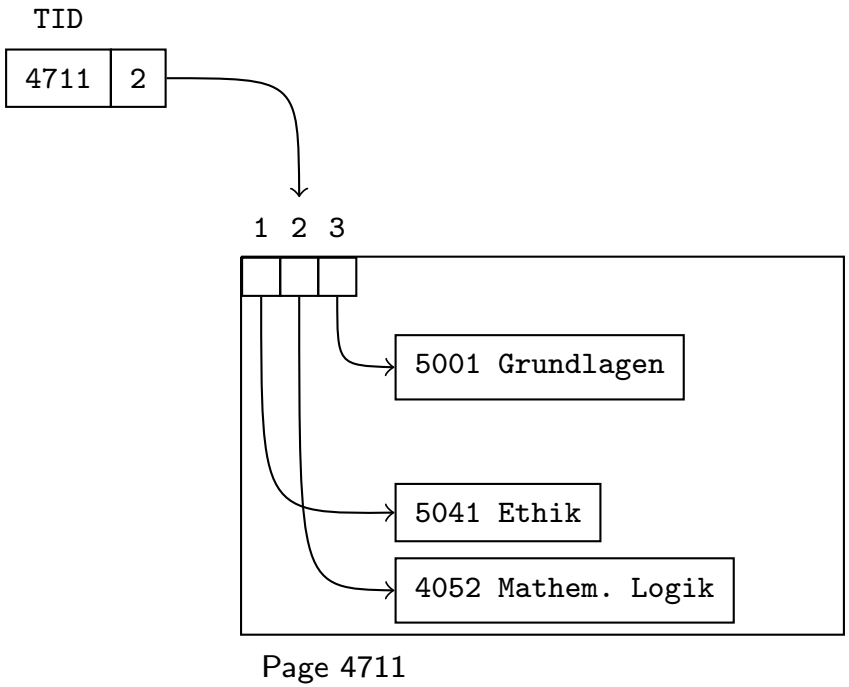
Slotted Pages/2

- TID: Tuple Identifier besteht aus
  - Nummer des Blocks (page ID)
  - Offset des Pointers zum Datensatz
- Datensätze werden nicht direkt adressiert, sondern über TID



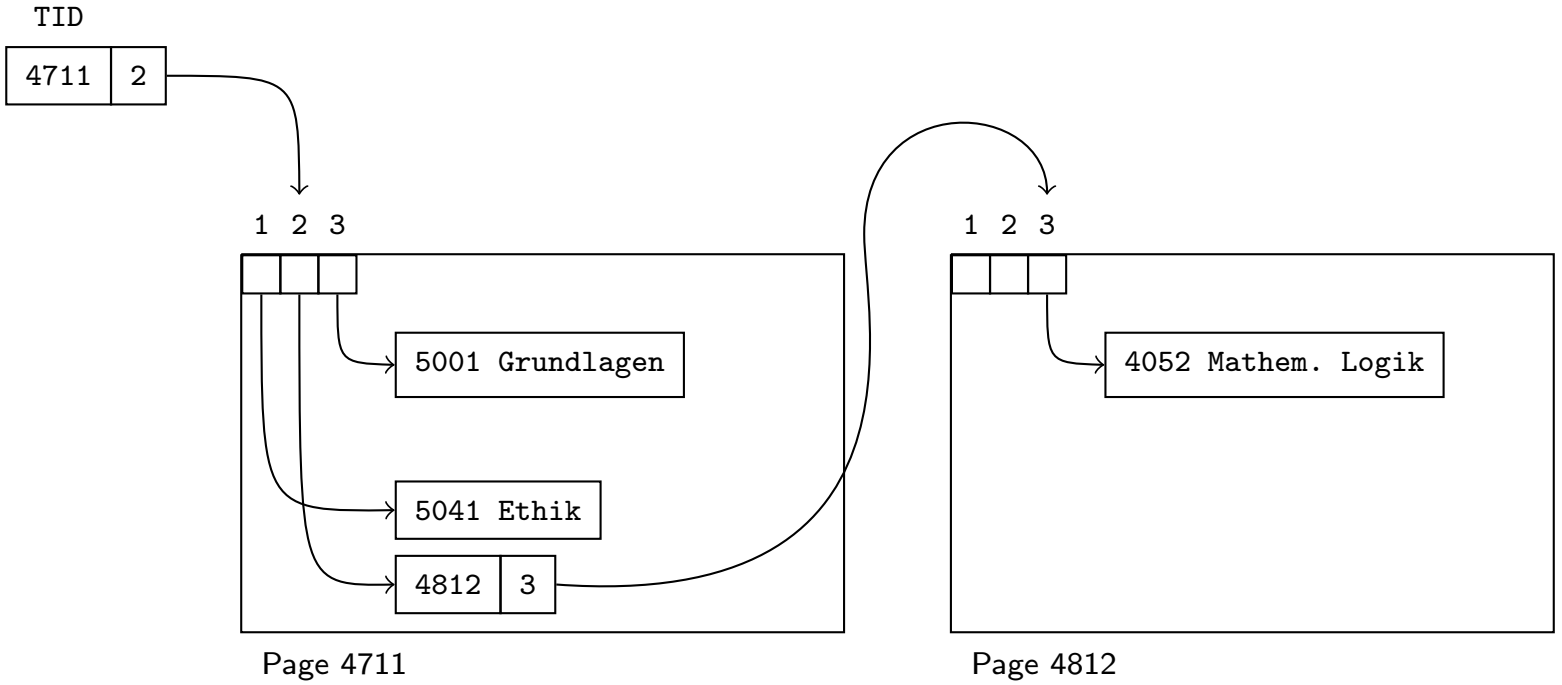
Slotted Pages/3

- Verschieben innerhalb des Blocks:
  - Pointer im Kopfteil wird geändert
  - TID ändert sich nicht



Slotted Pages/4

- Verschieben zwischen Blöcken:
  - Datensatz wird ersetzt durch Referenz auf neuen Block, welche nur intern genutzt wird
  - Zugriff auf Datensatz erfordert das Lesen von zwei Blöcken
  - TID des Datensatzes ändert sich nicht
  - weitere Verschiebungen modifizieren stets die Referenz im ursprünglichen Block (d.h. es entsteht keine verkettete Liste)



Organisation von Datensätzen in Dateien/1

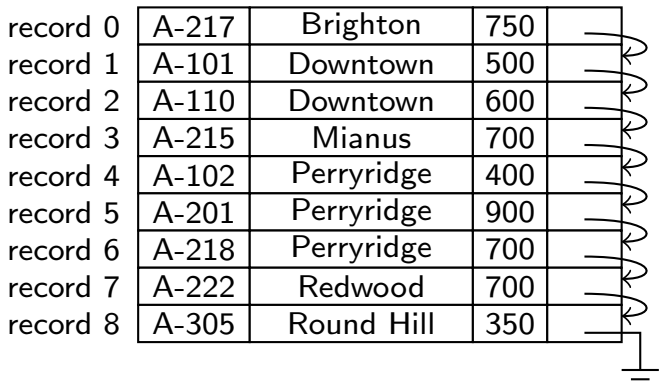
Verschiedene Ansätze, um Datensätze in Dateien logisch anzuordnen (primary file organisation):

- Heap Datei: ein Datensatz kann irgendwo gespeichert werden, wo Platz frei ist, oder er wird am Ende angehängt
- Sequentielle Datei: Datensätze werden nach einem bestimmten Datenfeld sortiert abgespeichert
- Hash Datei: der Hash-Wert für ein Datenfeld wird berechnet; der Hash-Wert bestimmt, in welchem Block der Datei der Datensatz gespeichert wird

Normalerweise wird jede Tabelle in eigener Datei gespeichert.

Organisation von Datensätzen in Dateien/2

- Sequentielle Datei: Datensätze nach Suchschlüssel (ein oder mehrere Datenfelder) geordnet
  - Datensätze sind mit Pointern verkettet
  - gut für Anwendungen, die sequentiellen Zugriff auf gesamte Datei brauchen
  - Datensätze sollten – soweit möglich – nicht nur logisch, sondern auch physisch sortiert abgelegt werden
- Beispiel: Konto(KontoNr, **FilialName**, Kontostand)



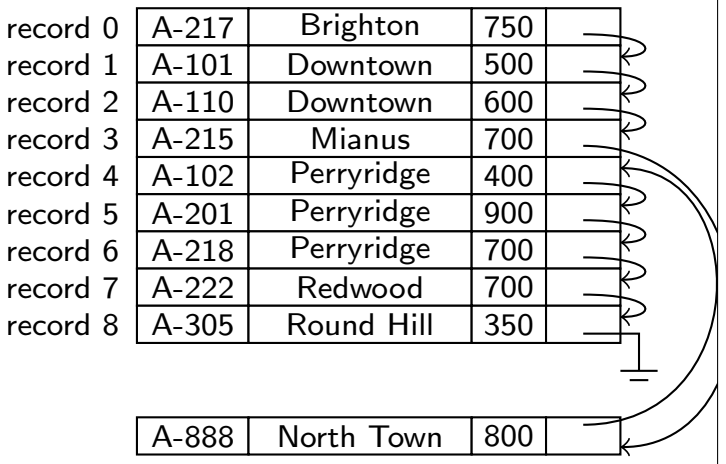
Organisation von Datensätzen in Dateien/3

- Physische Ordnung erhalten ist schwierig.
- Löschen:
  - Datensätze sind mit Pointern verkettet (verkettete Liste)
  - gelöschter Datensatz wird aus der verketteten Liste genommen

  → leere Zwischenräume reduzieren Datendichte

- Einfügen:
  - finde Block, in den neuer Datensatz eingefügt werden müsste
  - falls freier Speicher im Block: einfügen
  - falls zu wenig freier Speicher: Datensatz in Überlauf-Block (overflow block) speichern

  → Tabelle sortiert lesen erfordert nicht-sequentiellen Blockzugriff



- Datei muss von Zeit zu Zeit reorganisiert werden, um physische Ordnung wieder herzustellen

Datenbankverzeichnis/1

- Datenbankverzeichnis (Katalog): speichert Metadaten
  - Informationen über Relationen
    - Name der Relation
    - Name und Typen der Attribute jeder Relation
    - Name und Definition von Views
    - Integritätsbedingungen (z.B. Schlüssel und Fremdschlüssel)
  - Benutzerverwaltung
  - Statistische Beschreibung der Instanz
    - Anzahl der Tupel in der Relation
    - häufigste Werte
  - Physische Dateiorganisation
    - wie ist eine Relation gespeichert (sequentiell/Hash/...)
    - physischer Speicherort (z.B. Festplatte)
    - Dateiname oder Adresse des ersten Blocks auf der Festplatte
  - Information über Indexstrukturen

Datenbankverzeichnis/2

- Physische Speicherung des Datenbankverzeichnisses:
  - spezielle Datenstrukturen für effizienten Zugriff optimiert
  - Relationen welche bestehende Strategien für effizienten Zugriff nutzen
- Beispiel-Relationen in einem Verzeichnis (vereinfacht):
  - RELATION-METADATA(relation-name, number-of-attributes, storage-organization, location)
  - ATTRIBUTE-METADATA(attribute-name, relation-name, domain-type, position, length)
  - USER-METADATA(user-name, encrypted-password, group)
  - INDEX-METADATA(index-name, relation-name, index-type,index-attributes)
  - VIEW-METADATA(view-name, definition)
- PostgreSQL (ver 9.3): mehr als 70 Relationen:
  `http://www.postgresql.org/docs/9.3/static/catalogs-overview.html`

Datenbanken 2

Indexstrukturen

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

FB Computerwissenschaften

Universität Salzburg

database research group

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 9. Oktober 2018

Inhalt

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

Inhalt

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 Schlüsseln 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:

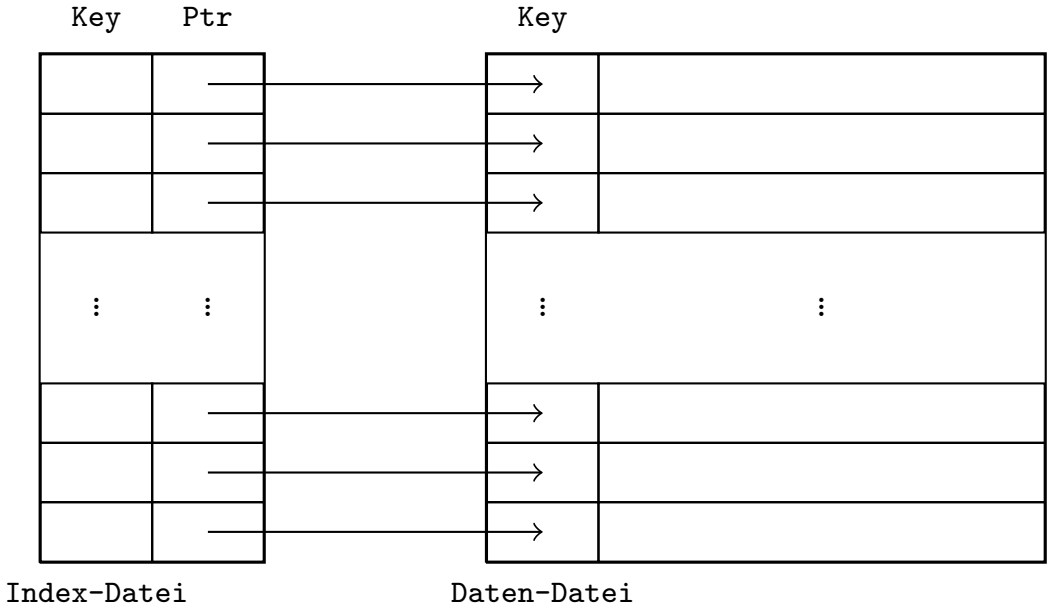
- Ordnung der Daten- und Index-Datei:
  - Primärindex
  - Clustered Index
  - Sekundärindex
- Art der Index-Einträgen:
  - sparse Index
  - dense Index

Nicht alle Kombinationen üblich/möglich:

- Primärindex ist oft sparse
- Sekundärindex ist immer dense

Primärindex/1

- Primärindex:
  - Datensätze in der Daten-Datei sind nach Suchschlüssel sortiert
  - Suchschlüssel ist eindeutig, d.h., Suche nach 1 Schlüssel ergibt (höchstens) 1 Tupel

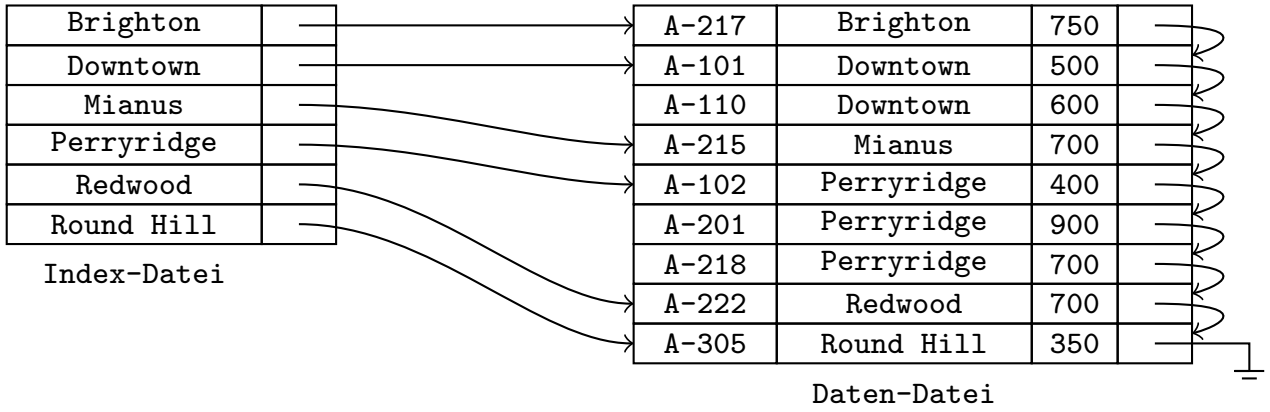


Primärindex/2

- Index-Datei:
  - sequentiell geordnet nach Suchschlüssel
- Daten-Datei:
  - sequentiell geordnet nach Suchschlüssel
  - jeder Suchschlüssel kommt nur 1 mal vor
- Effiziente Zugriffsarten:
  - Punkt- und Bereichsanfragen
  - nicht-sequentieller Zugriff (random access)
  - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

Clustered Index

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

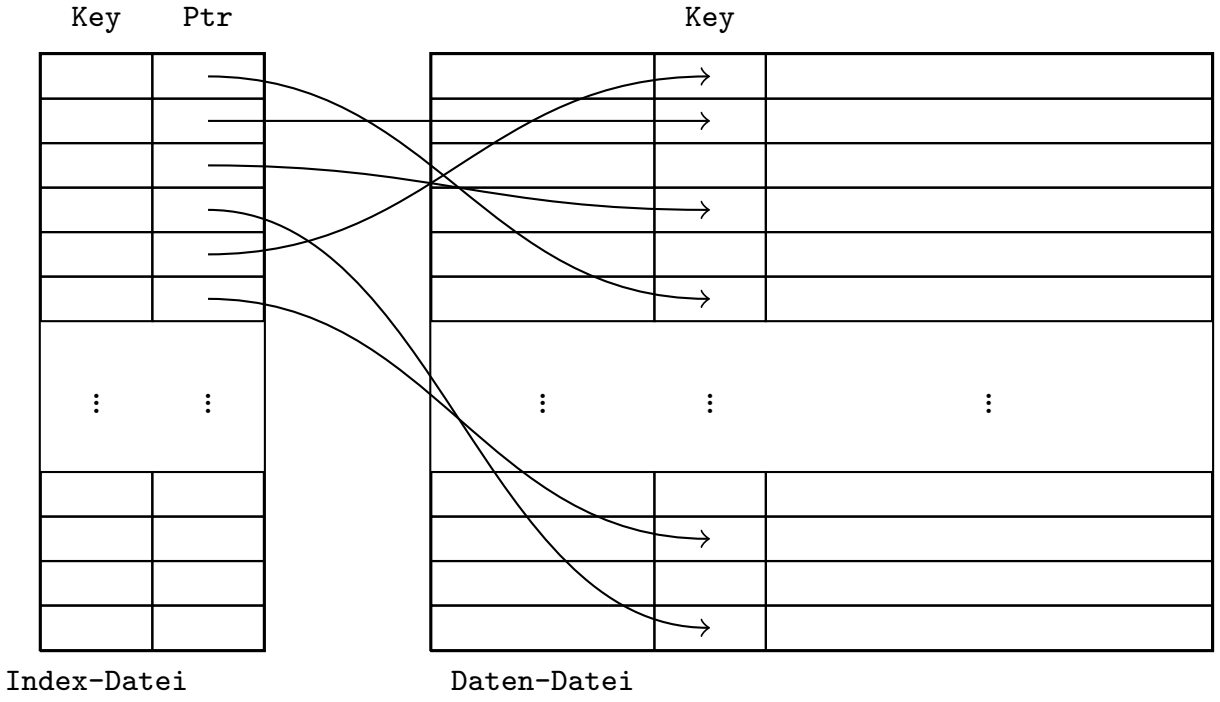


Sekundärindex/1

- Primär- vs. Sekundärindex:
  - nur 1 Primärindex (bzw. Clustered Index) möglich
  - beliebig viele Sekundärindizes
  - Sekundärindex für schnellen Zugriff auf alle Felder, die nicht Suchschlüssel des Primärindex sind
- Beispiel: Konten mit Primärindex 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
- Sekundärindex für schnellen Zugriff erforderlich

Sekundärindex/2

- Index-Datei:
  - sequentiell nach Suchschlüssel geordnet
- Daten-Datei:
  - Suchschlüssel kann *mehrfach* vorkommen
  - *nicht* nach Suchschlüssel geordnet



Sekundärindex/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

Duplikate/1

Umgang mit mehrfachen Suchschlüsseln:

(a) Doppelte Indexeinträge:

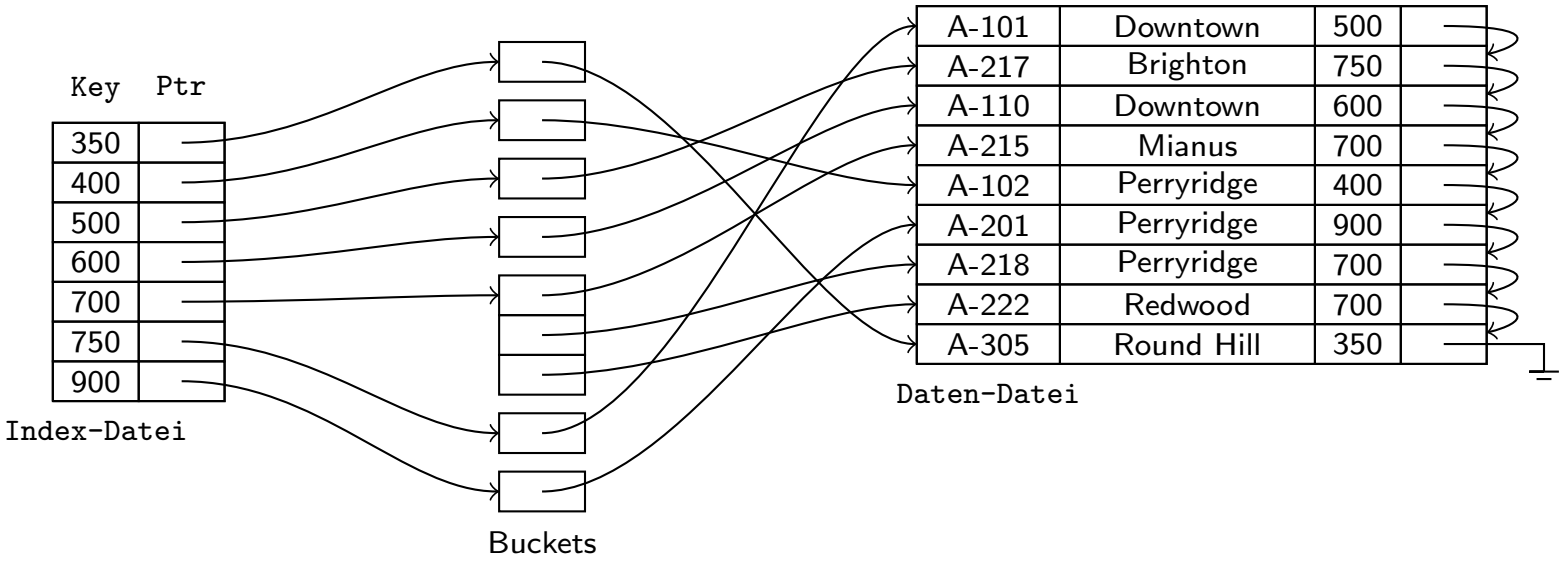
- ein Indexeintrag für jeden Datensatz

→ 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:

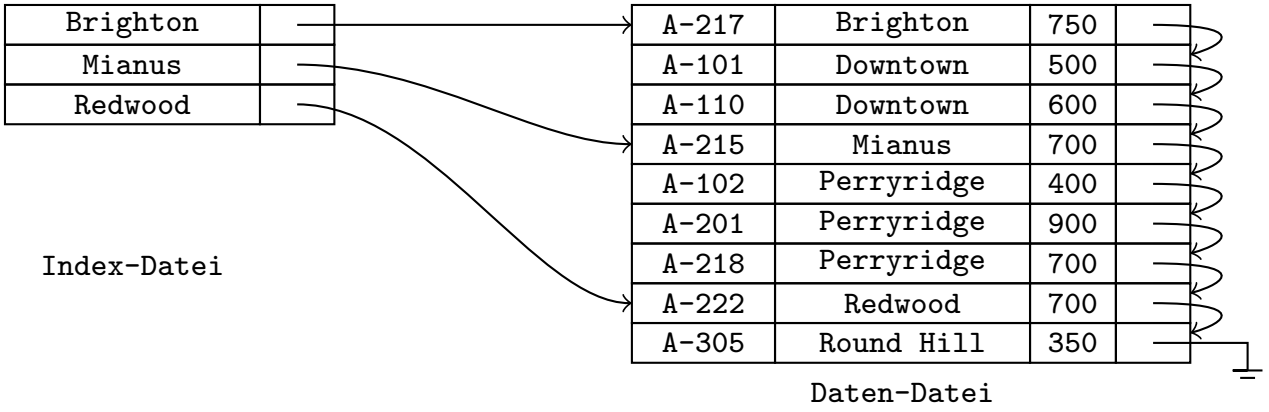
(c) Suchschlüssel eindeutig machen:

- 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)

$\rightarrow$ wird in der Praxis verwendet

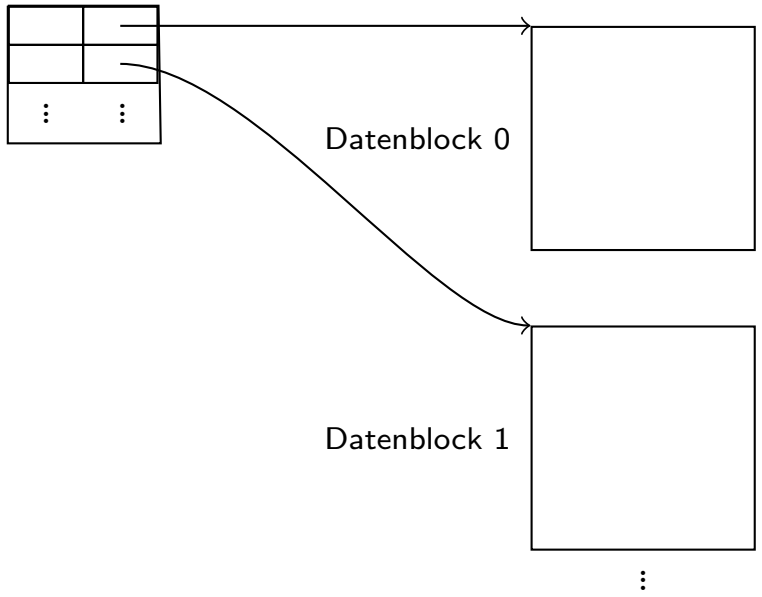
Sparse Index/1

- 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. Primärindex oder 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
- Sekundärindex ist immer dense

Gegenüberstellung von Index-Typen

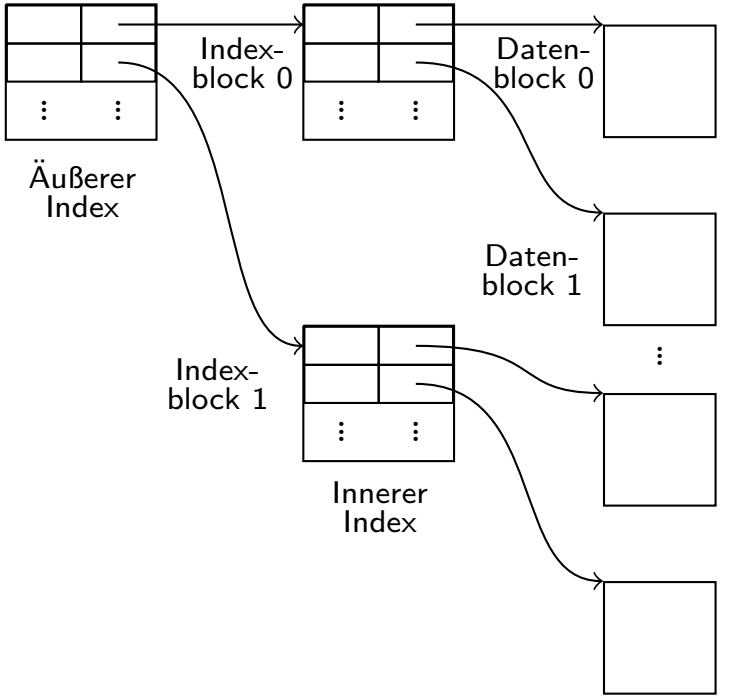
- Alle Index-Typen machen Punkt-Anfragen erheblich schneller.
- Index erzeugt Kosten bei Updates: Index muss auch aktualisiert werden.
- Dense/Sparse und Primär/Sekundär:
  - Primärindex kann dense oder sparse sein
  - Sekundärindex ist immer dense
- Sortiert lesen (=sequentielles Lesen nach Suchschlüssel-Ordnung):
  - mit Primärindex schnell
  - mit Sekundärindex teuer, da sich aufeinander folgende Datensätze auf unterschiedlichen Blocks 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")

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 Blocks)
  - eventuelle Overflow Blocks 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

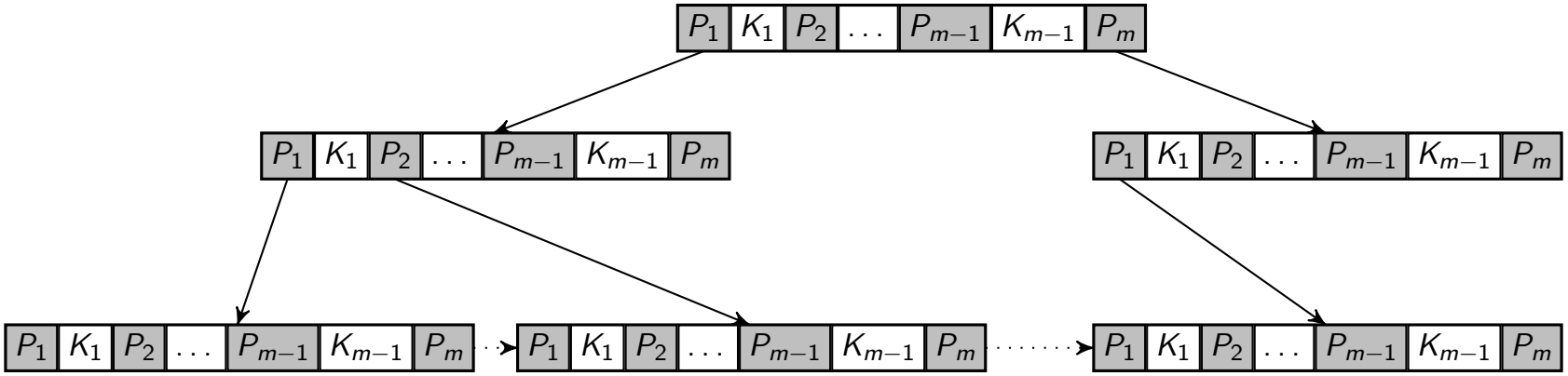
Inhalt

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

B^+-Baum/2



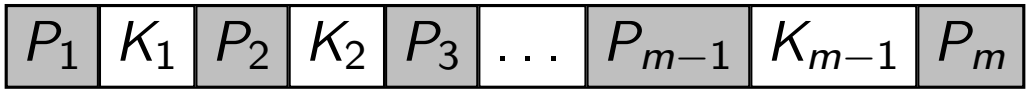
- Knoten mit Grad m: enthält bis zu $m-1$ Suchschlüssel und m Pointer
  - Knotengrad $m > 2$ entspricht der maximalen Anzahl der Pointer
  - Suchschlüssel im Knoten sind sortiert
  - Knoten (außer Wurzel) sind mindestens halb voll
- Wurzelknoten:
  - als Blattknoten: 0 bis $m-1$ Suchschlüssel
  - als 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)

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.

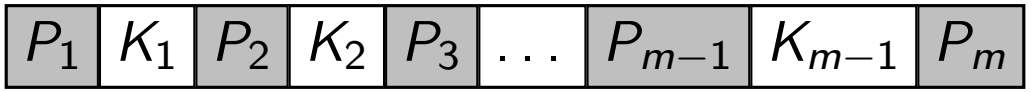
B^+-Baum Knotenstruktur/1



Blatt-Knoten:

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

B^+-Baum Knotenstruktur/2

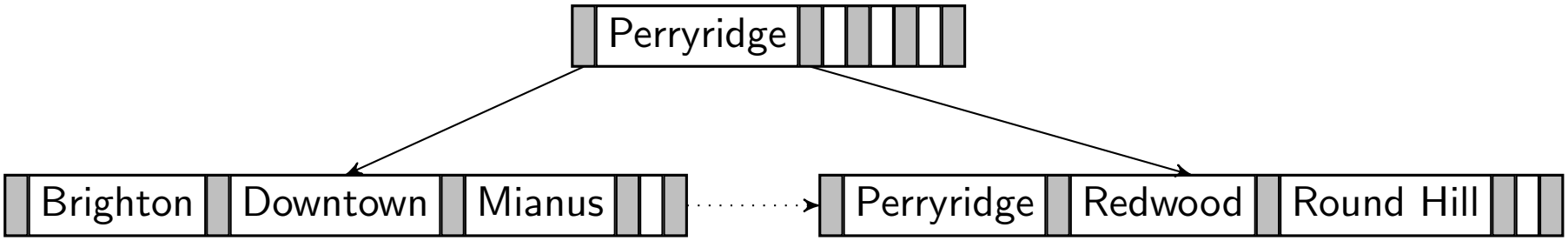


Innere Knoten:

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

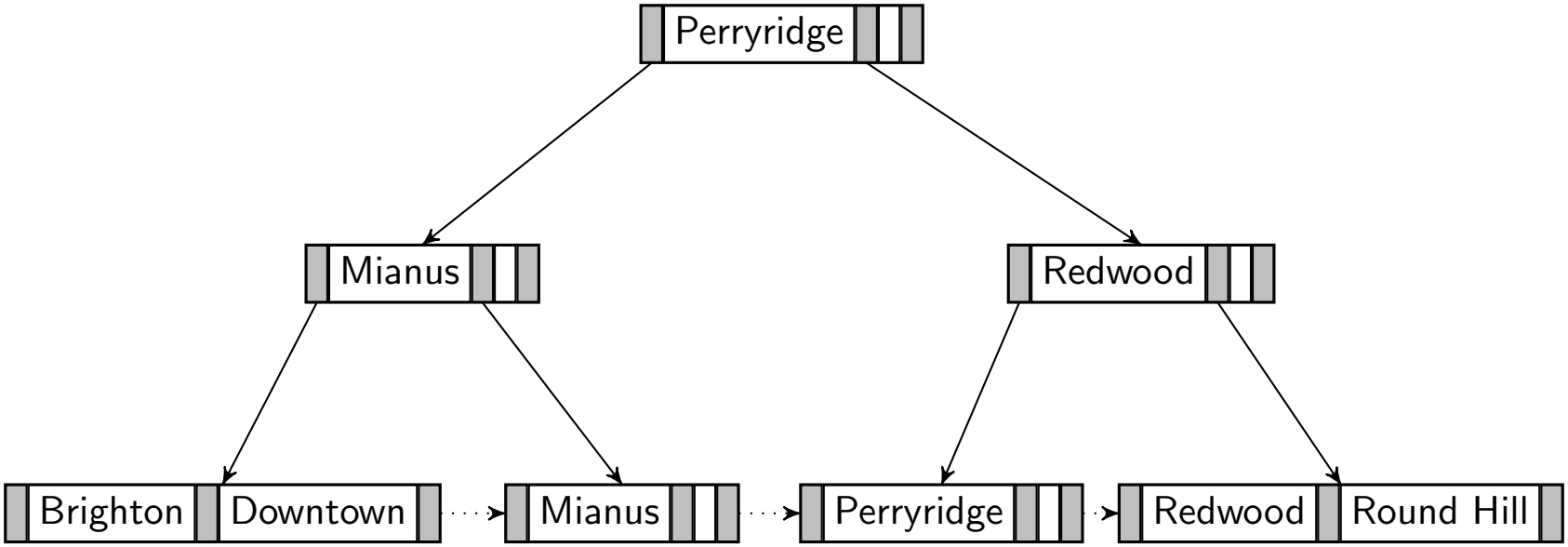
Beispiel: B^+-Baum/1

- 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

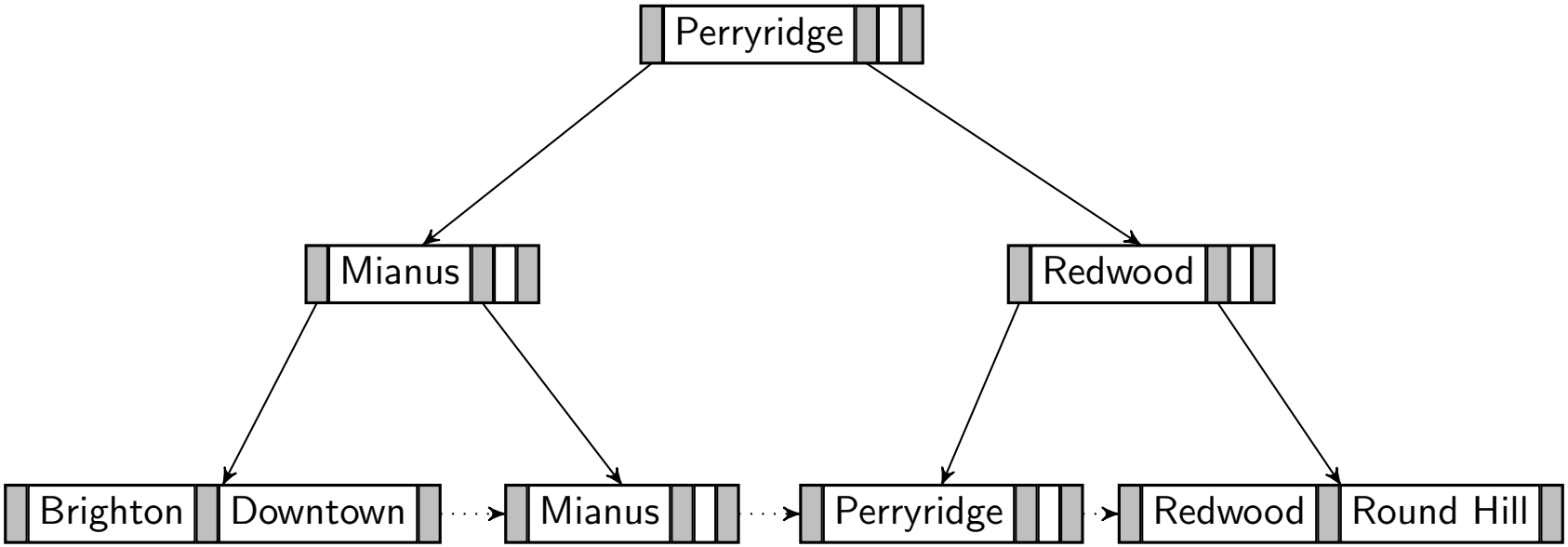


Suche im B^+-Baum/1

- Algorithmus: Suche alle Datensätze mit Suchschlüssel k (Annahme: dense 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 Datensatz mit Suchschlüssel k existiert

Suche im B^+-Baum/2

- Beispiel: Finde alle Datensätze mit Suchschlüssel $k =$*Mianus*
  - Beginne mit dem Wurzelknoten
  - Kein Schlüssel $K_i \leq$*Mianus* existiert, also folge P_1
  - $K_1 =$*Mianus* ist der größte Suchschlüssel $K_i \leq$*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
    $\Rightarrow \lceil log_{\lceil m/2 \rceil}(L) \rceil + 1$ Blocks[1] 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 im Hauptspeicher, oft auch dessen Kinder, dadurch werden 1–2 Block-Zugriffe pro Suche gespart

- Suche effizienter als in sequentiellem Index:
  - bis zu $\lfloor log_2(B) \rfloor + 1$ Blocks[1] lesen im einstufigen sequentiellen Index (binäre Suche, Index mit B Blocks, $B = \lceil K/(m-1) \rceil$)

[1] nur Index Blocks werden gezählt, Datenzugriff hier nicht berücksichtigt

Integrierte Übung 2.1

Es soll ein Index mit 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.

a) Wie viele Block-Zugriffe erfordert ein B^+-Baum Index maximal, wenn kein Block im Hauptspeicher ist?

b) Wie viele Block-Zugriffe erfordert ein einstufiger, sequentieller Index mit binärer Suche?

Einfügen in B^+-Baum/1

- Datensatz mit Suchschlüssel k einfügen:
  1. füge Datensatz in Daten-Datei ein (ergibt Pointer)
  2. finde Blattknoten für Suchschlüssel k
  3. **falls** im Blatt noch Platz ist **dann**:
     - füge (Pointer, Suchschlüssel)-Paar so in Blatt ein, dass Ordnung der Suchschlüssel erhalten bleibt
  4. **sonst** (Blatt ist voll) teile Blatt-Knoten:
     a) sortiere alle Suchschlüssel (einschließlich k)
     b) die Hälfte der Suchschlüssel bleiben im alten Knoten
     c) die andere Hälfte der Suchschlüssel kommt in einen neuen Knoten
     d) füge den kleinsten Eintrag des neuen Knotens in den Eltern-Knoten des geteilten Knotens ein
     e) **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

$\rightarrow$ Knoten L, Suchschlüssel k, Pointer p (zu Datensatz oder Knoten)

Algorithm 1: B+TreeInsert(L, k, p)

if *L has less than $m-1$ key values* **then**
  insert(k, p) into L

else // Knoten teilen
  $T \leftarrow L \cup (k, p)$; // temporärer Speicher
  create new node L';
  $L'.p_m \leftarrow L.p_m$;
  $L \leftarrow \emptyset$;
  $L.p_m \leftarrow L'$;
  copy $T.p_1$ through $T.k_{\lceil m/2 \rceil}$ into L;
  copy $T.p_{\lceil m/2 \rceil+1}$ through $T.k_m$ into L';
  $k' \leftarrow T.k_{\lceil m/2 \rceil+1}$;
  B+TreeInsertInParent(L, k', L');

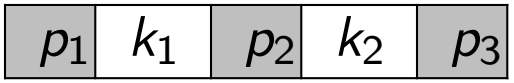
Algorithmus: Einfügen in B^+-Baum/2

Algorithm 2: B+TreeInsertInParent(L, k, L')

if *L is root* **then**
  create new root with children L, L' and value k;
  return;
$P \leftarrow$ parent(L);
if *P has less than m pointers* **then**
  insert(k, L') into P;
else // Knoten teilen
  $T \leftarrow P \cup (k, L')$;
  erase all entries from P;
  create new node P';
  copy $T.p_1$ through $T.p_{\lceil m/2 \rceil}$ into P;
  copy $T.p_{\lceil m/2 \rceil + 1}$ through $T.p_{m+1}$ into P';
  $k' \leftarrow T.k_{\lceil m/2 \rceil}$;
  B+TreeInsertInParent(P, k', P');

Blatt teilen/1

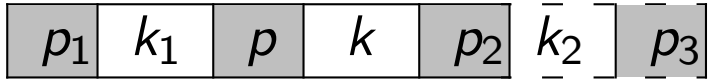
Kopiere L nach T und füge (k, p) ein:



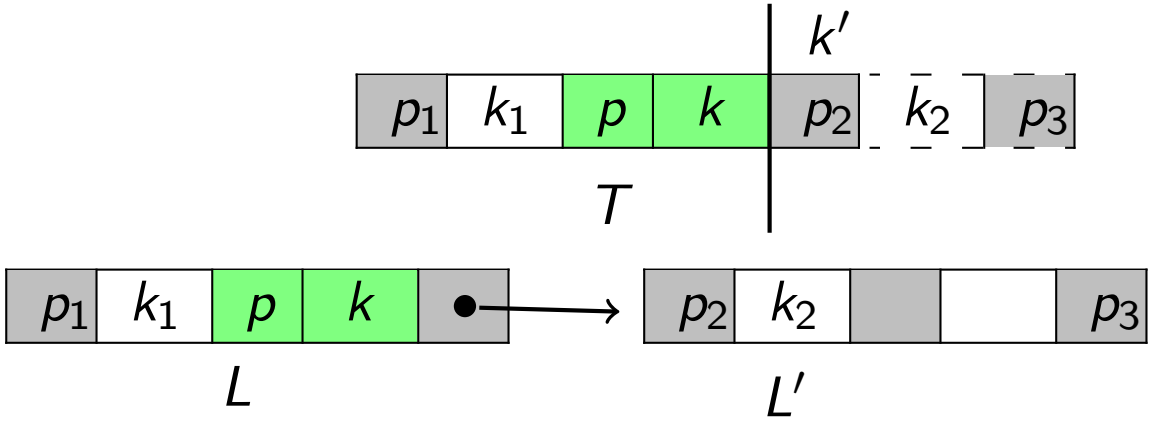
$m = 3$

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

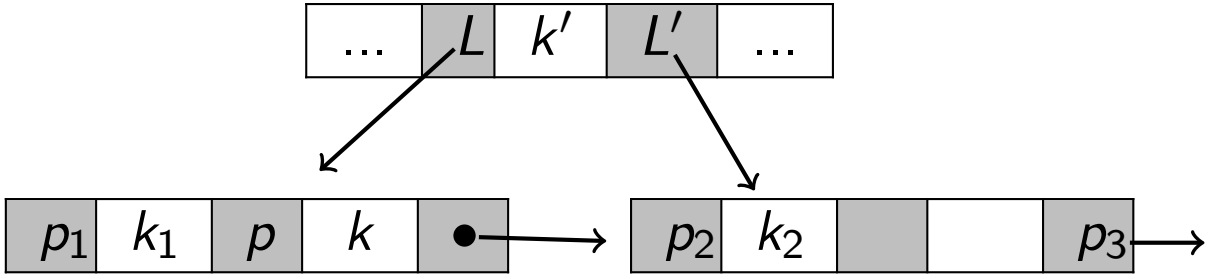
   T



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



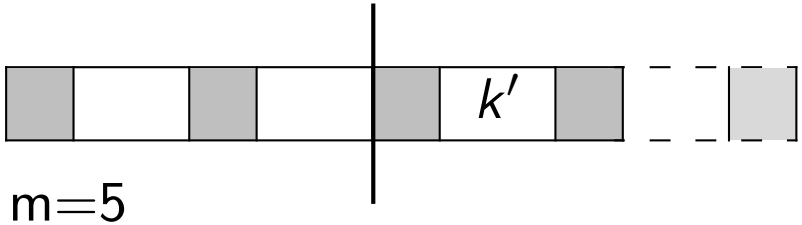
3. (k', L') in Elternknoten von L einfügen



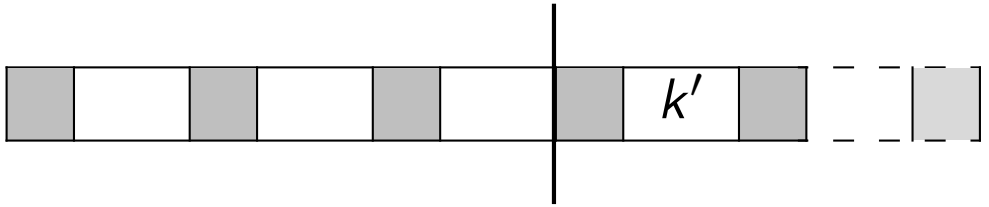
Blatt teilen/2

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

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



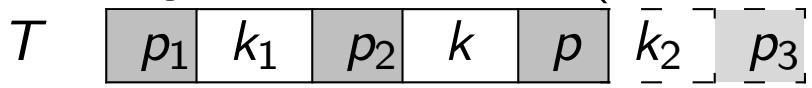
- m ungerade, z.B.: m=5



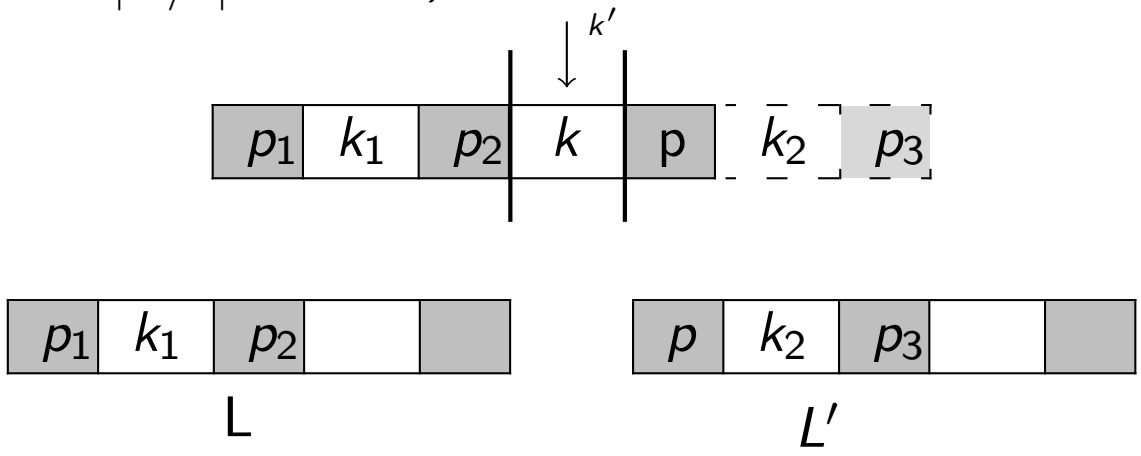
Innere Knoten teilen/1

Kopiere P nach T und füge (k, p) ein:

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

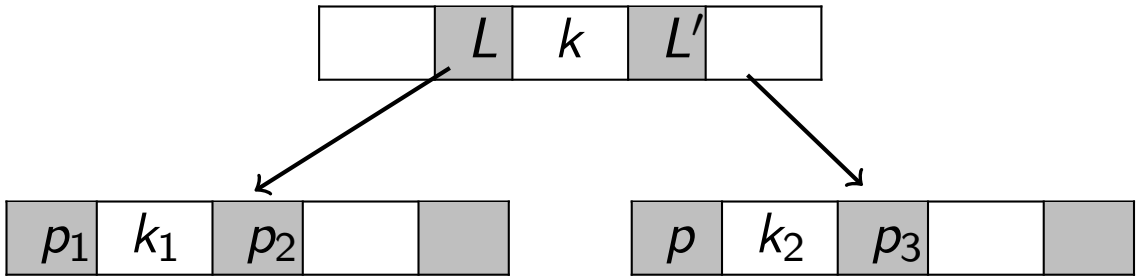


2. Teilen ($k' = T.k_{\lceil m/2 \rceil} = T.k_2$)



Innere Knoten teilen/2

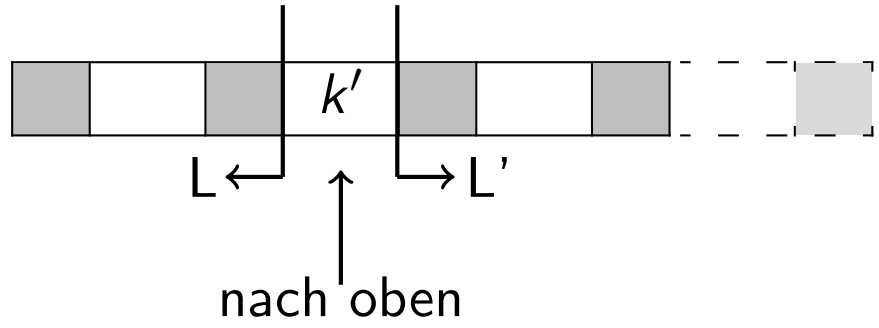
3. (k', L') in Elternknoten von L einfügen



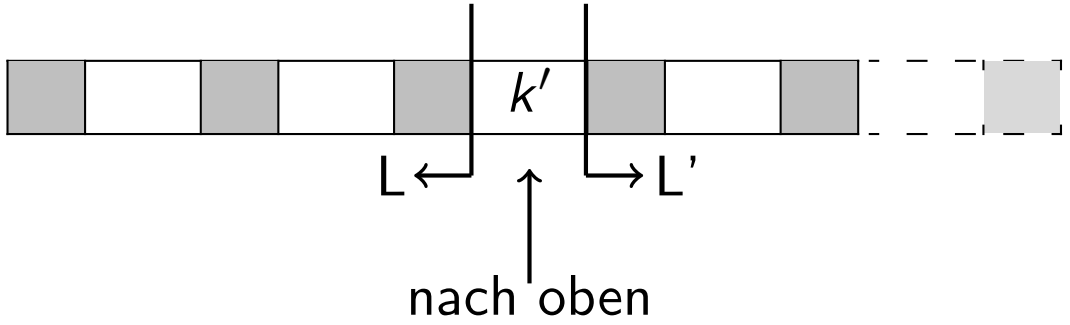
Innere Knoten teilen/3

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

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

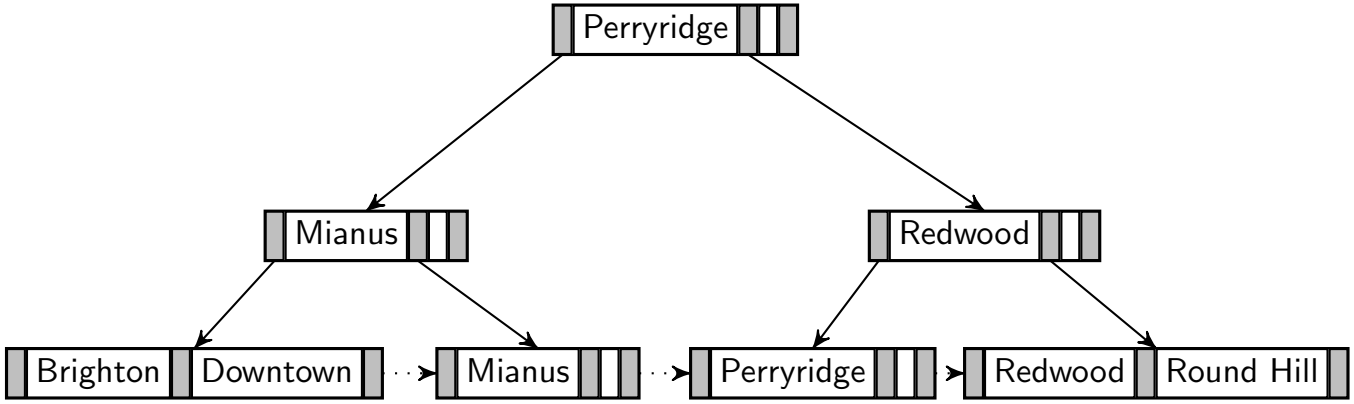


- m ungerade, z.B.: m=5

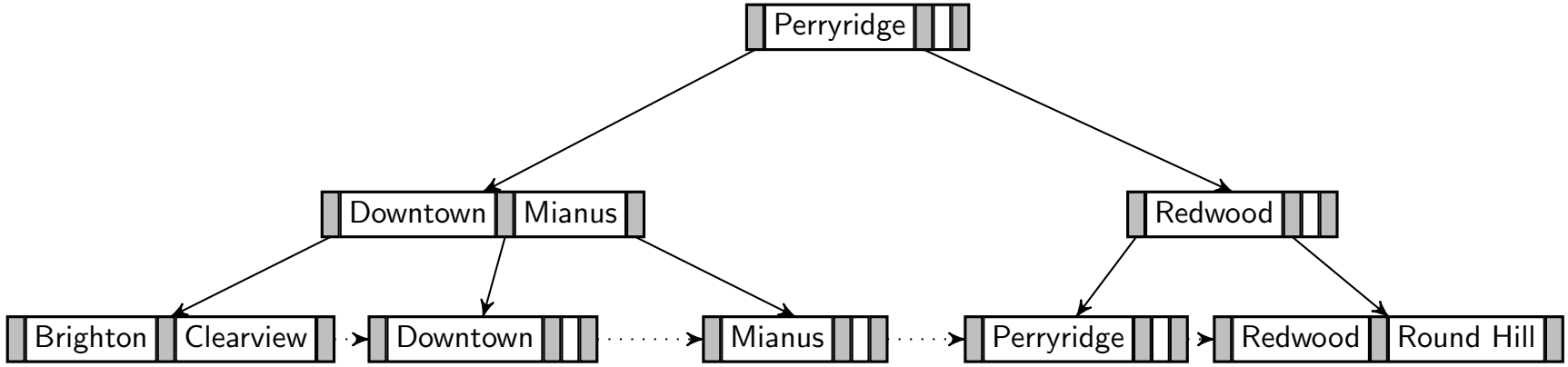


Beispiel: Einfügen in B^+-Baum/1

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

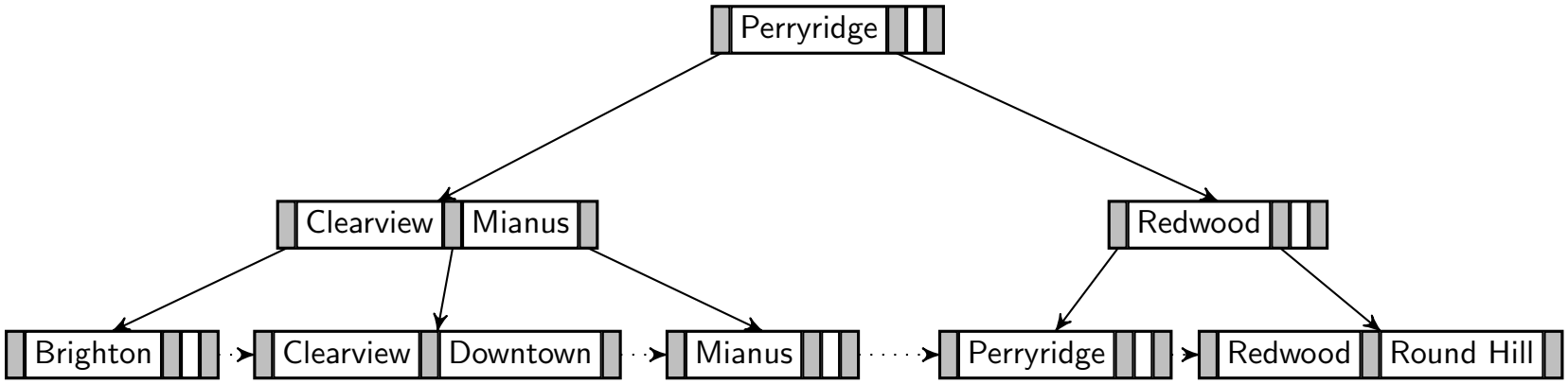


- B^+-Baum nach Einfügen von *Clearview*

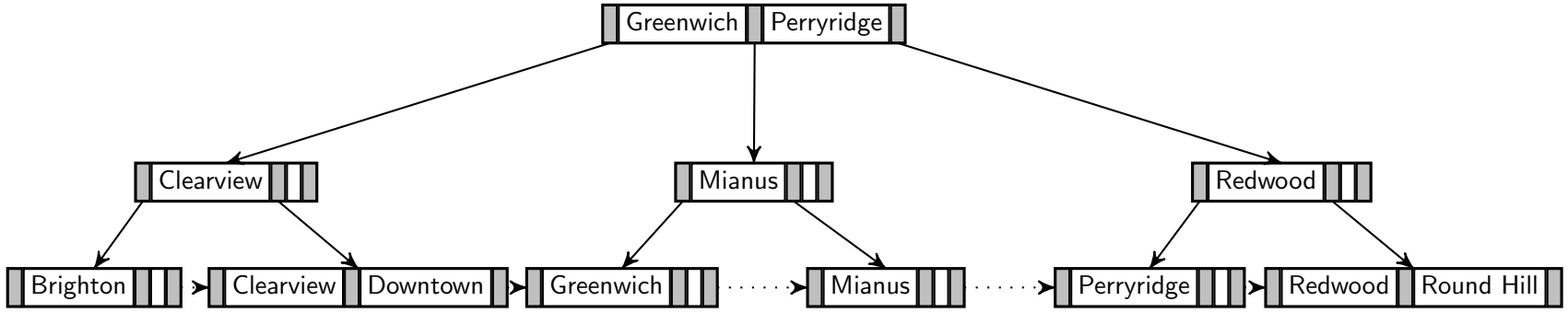


Beispiel: Einfügen in B^+-Baum/2

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



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



Löschen von B^+-Baum/1

Datensatz mit Suchschlüssel k löschen:

1. finde Blattknoten mit Suchschlüssel k
2. lösche k von Knoten
3. **falls** Knoten durch Löschen von k zu wenige Einträge hat:
   a. Einträge im Knoten und einem Geschwisterknoten passen in 1 Knoten **dann**:
      - **vereinige** 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
   b. 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

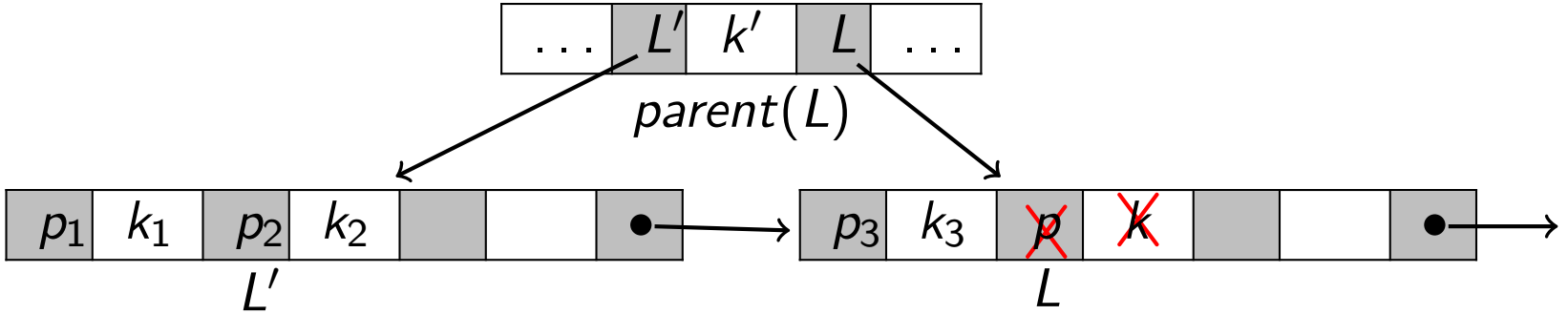
Algorithm 3: B+TreeDelete(L, k, p)

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**
  $L' \leftarrow$ previous sibling of L [next, if there is no previous];
  $k' \leftarrow$ 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** $L' \leftarrow L' \cup k'$ and all (k_i, p_i) from L;
    else $L' \leftarrow L' \cup$ all (k_i, p_i) 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 $\leftrightarrow$ last,...);

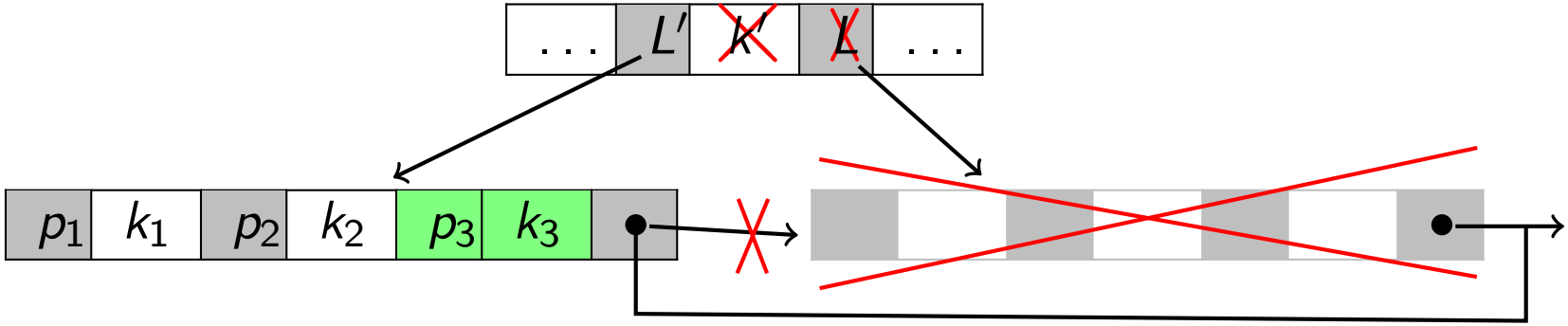
Löschen aus Blatt/1

(k, p) wird aus L gelöscht:

1. Vereinigen $(m = 4)$
   Vorher:



   Nachher:

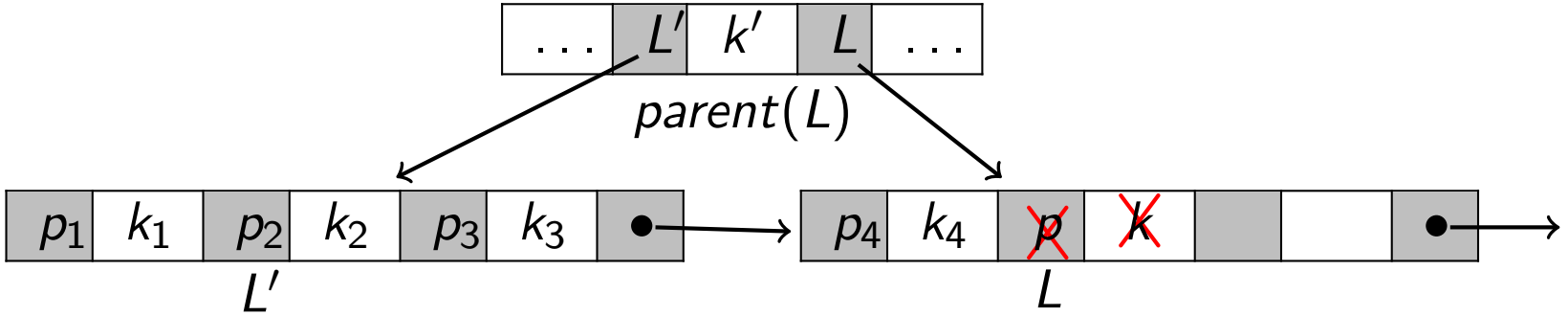


Löschen aus Blatt/2

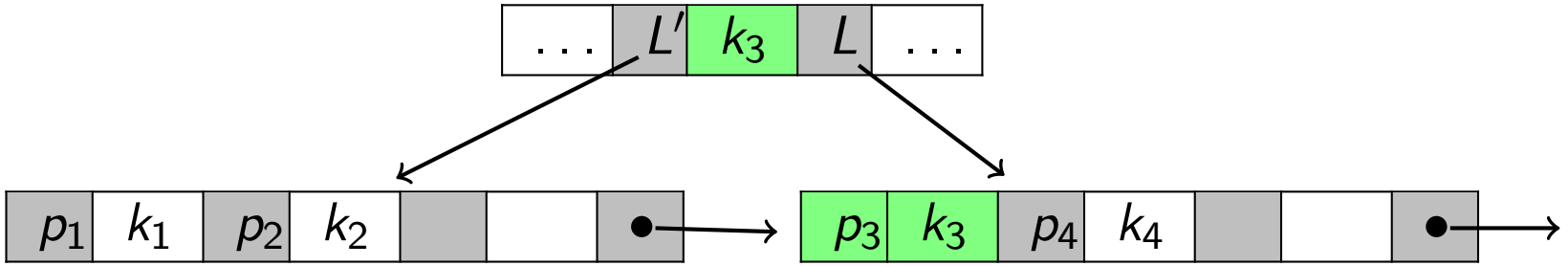
(k, p) wird aus L gelöscht:

2. Verteilen $(m = 4)$

   Vorher:



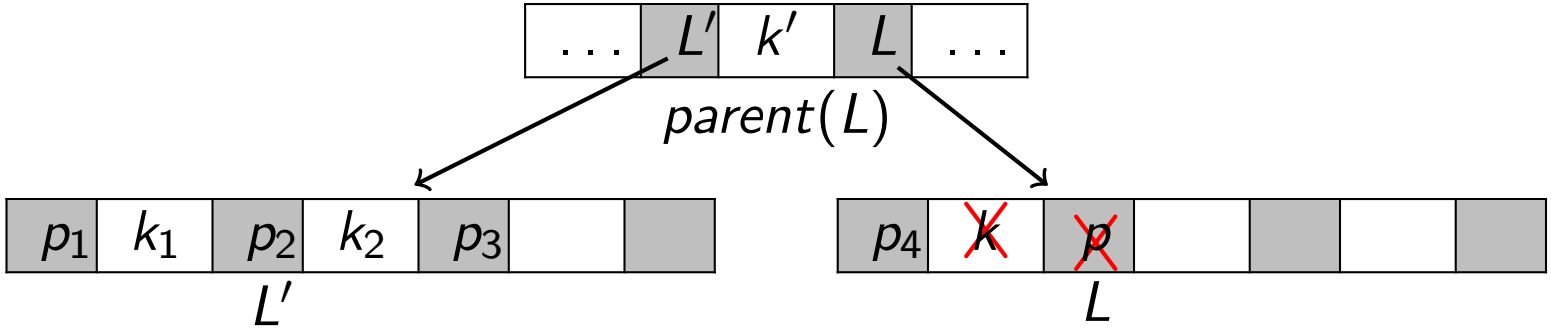
   Nachher:



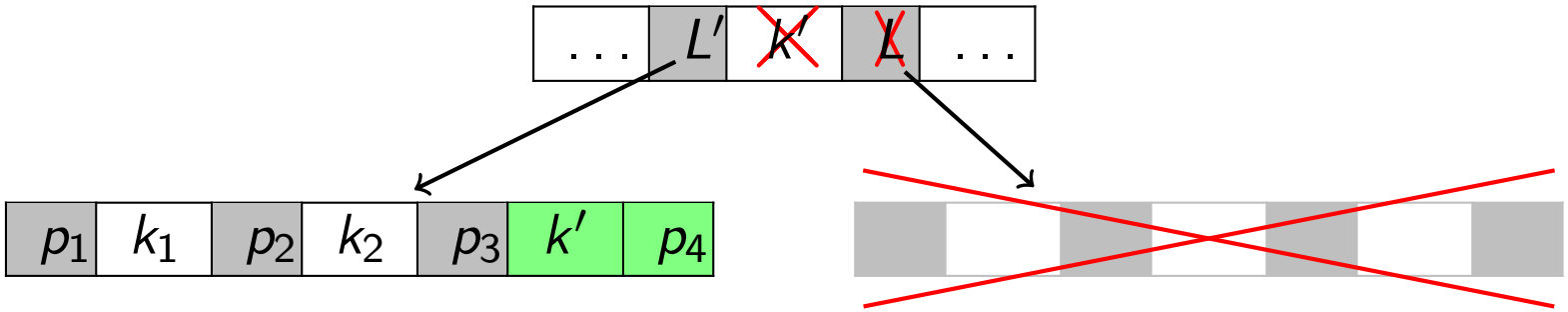
Löschen aus innerem Knoten/1

(k, p) wird aus L gelöscht:

1. Vereinigen $(m = 4)$
   Vorher:



   Nachher:

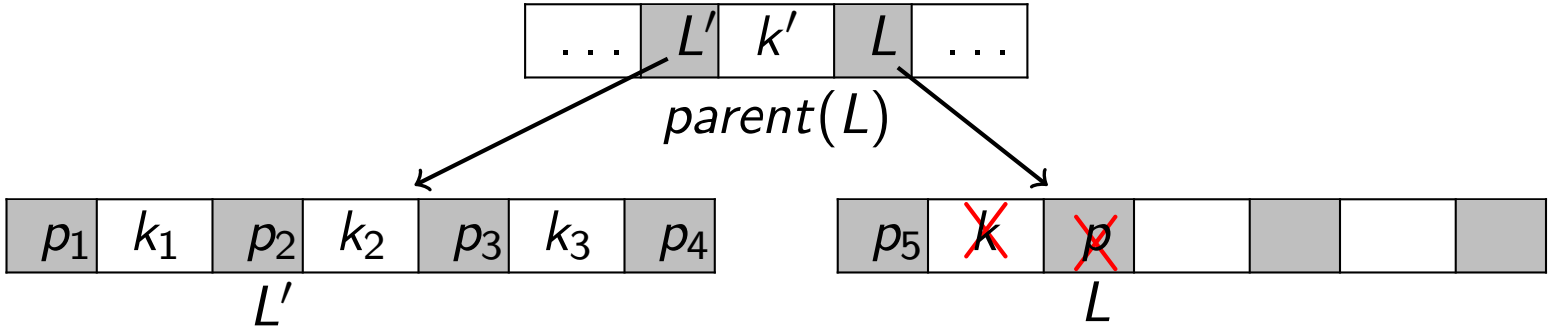


Löschen aus innerem Knoten/2

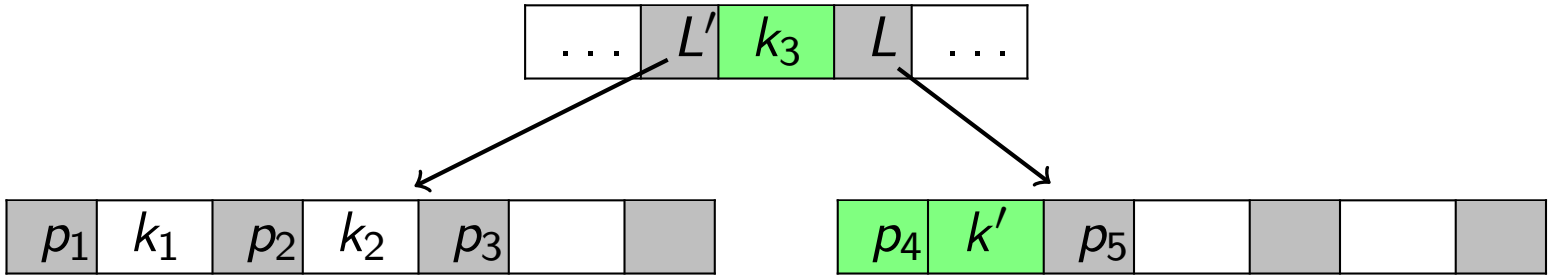
(k, p) wird aus L gelöscht:

2. Verteilen $(m = 4)$

   Vorher:

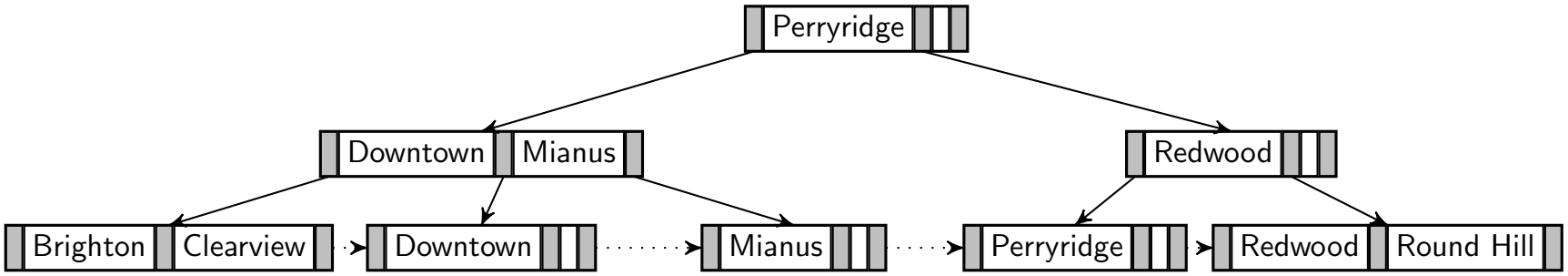


   Nachher:

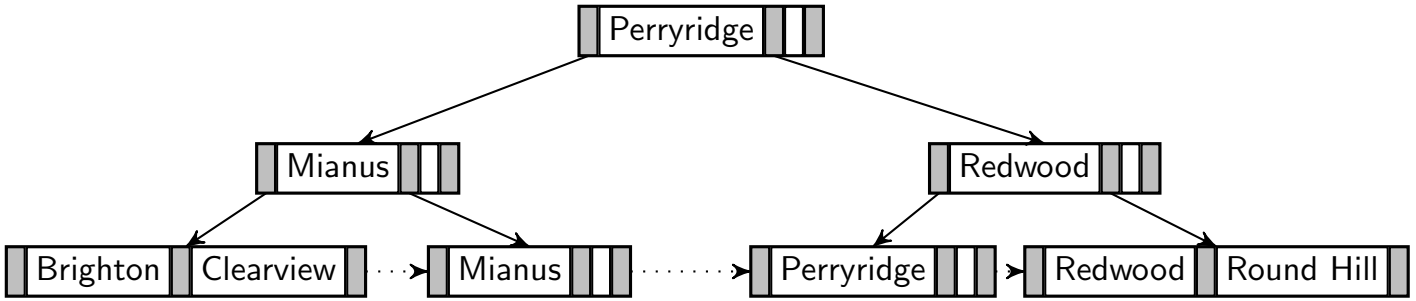


Beispiel: Löschen von B^+-Baum/1

- Vor Löschen von *Downtown*:



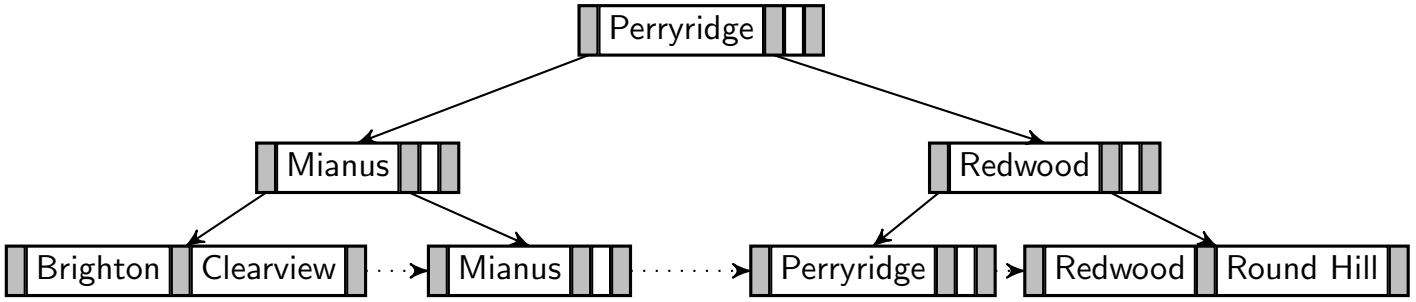
- Nach Löschen von *Downtown*:



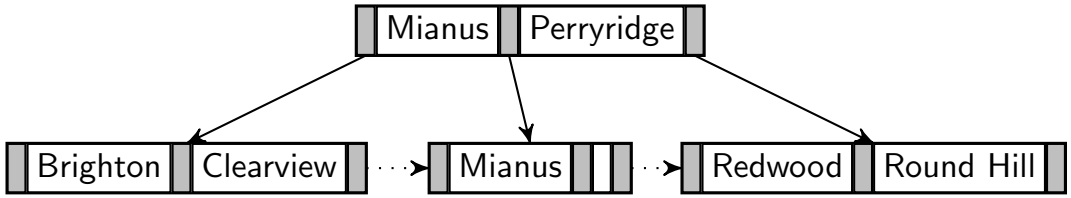
- Nach Löschen des Blattes mit *Downtown* hat der Elternknoten noch genug Pointer.
- Somit propagiert Löschen nicht weiter nach oben.

Beispiel: Löschen von B^+-Baum/2

- Vor Löschen von *Perryridge*:



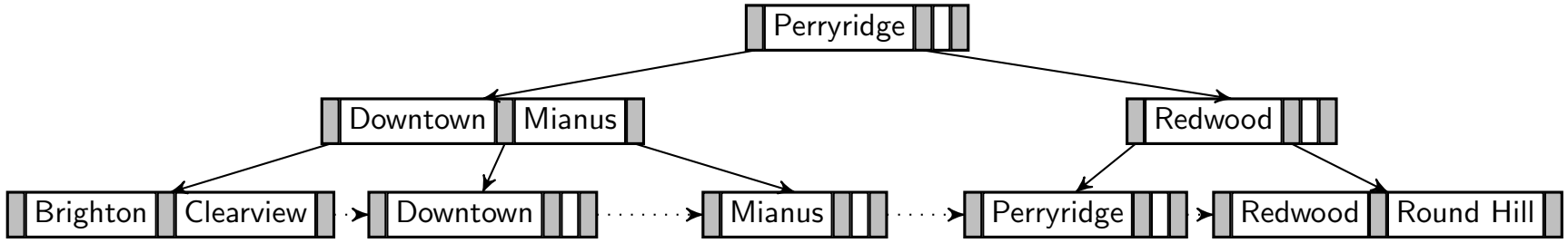
- Nach Löschen von *Perryridge*:



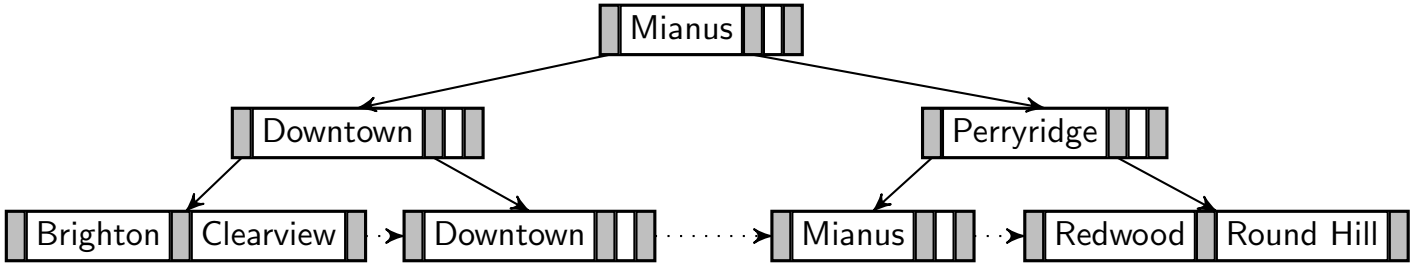
- Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und wird mit dem (rechten) Nachbarknoten vereinigt.
- Dadurch hat der Elternknoten zu wenig Pointer und wird mit seinem (linken) Nachbarknoten vereinigt (und ein Eintrag wird vom gemeinsamen Elternknoten gelöscht).
- Die Wurzel hat jetzt nur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

Beispiel: Löschen von B^+-Baum/3

- Vor Löschen von *Perryridge*:



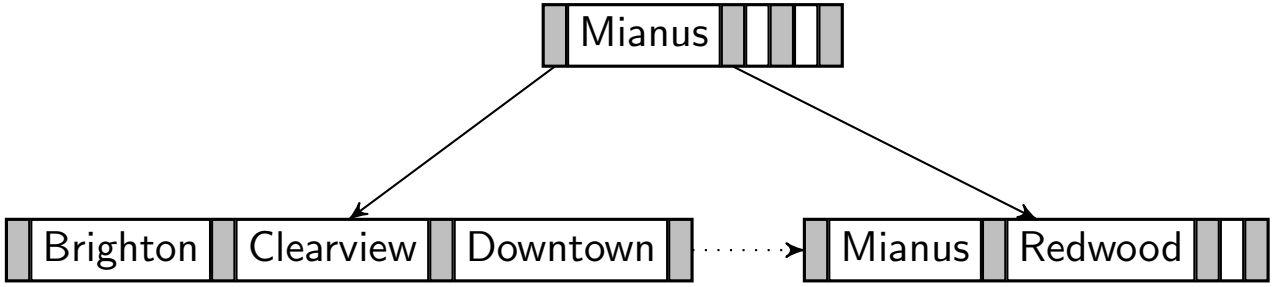
- Nach Löschen von *Perryridge*:



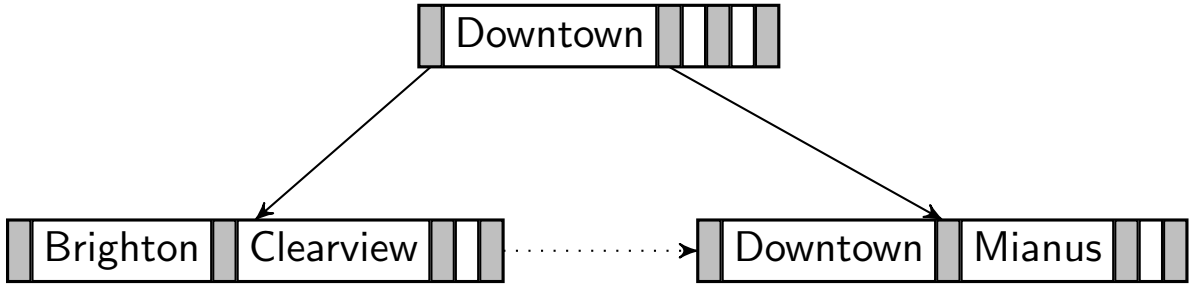
- Elternknoten von Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und erhält einen Pointer vom linken Nachbarn (Verteilung von Einträgen).
- Schlüssel im Elternknoten des Elternknotens (Wurzel in diesem Fall) ändert sich ebenfalls.

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

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 Datei Organisation

- Statisches Hashing ist eine Form der Dateiorganisation:
  - Datensätze werden in Buckets gespeichert
  - Zugriff erfolgt über eine Hashfunktion
  - Eigenschaften: konstante Zugriffszeit, kein Index erforderlich
- Bucket: Speichereinheit die ein oder mehrere Datensätze enthält
  - ein Block oder mehrere benachbarte Blocks auf der Platte
  - alle Datensätze mit bestimmtem Suchschlüssel sind im selben Bucket
  - Datensätze im Bucket können verschiedene Suchschlüssel haben
- Hash Funktion h: bildet Menge der Suchschlüssel K auf Menge der Bucket Adressen B ab
  - wird in konstanter Zeit (in der Anzahl der Datensätze) berechnet
  - mehrere Suchschlüssel können auf dasselbe Bucket abbilden
- Suchen eines Datensatzes mit Suchschlüssel:
  - verwende Hash Funktion um Bucket Adresse aufgrund des Suchschlüssels zu bestimmen
  - durchsuche Bucket nach Datensätzen mit Suchschlüssel

Beispiel: Hash Datei Organisation

- Beispiel: Organisation der Konto-Relation als Hash Datei mit Filialname als Suchschlüssel.
- 10 Buckets
- Numerischer Code des i-ten Zeichens im 26-Buchstaben-Alphabet wird als i angenommen, z.B., code(B)=2.
- Hash Funktion h
  - Summe der Codes aller Zeichen modulo 10:
  - h(Perryridge) = 125 mod 10 = 5
  - h(Round Hill) = 113 mod 10 = 3 (code(' ')=0)
  - h(Brighton) = 93 mod 10 = 3

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryrdige	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Funktionen/1

- Die Worst Case Hash Funktion bildet alle Suchschlüssel auf das gleiche Bucket ab.
  - Zugriffszeit wird linear in der Anzahl der Suchschlüssel.
- Die Ideale Hash Funktion hat folgende Eigenschaften:
  - Die Verteilung ist uniform (gleichverteilt), d.h. jedes Bucket ist der gleichen Anzahl von Suchschlüsseln aus der Menge aller Suchschlüssel zugewiesen.
  - Die Verteilung ist random (zufällig), d.h. im Mittel erhält jedes Bucket gleich viele Suchschlüssel unabhängig von der Verteilung der Suchschlüssel.

Hash Funktionen/2

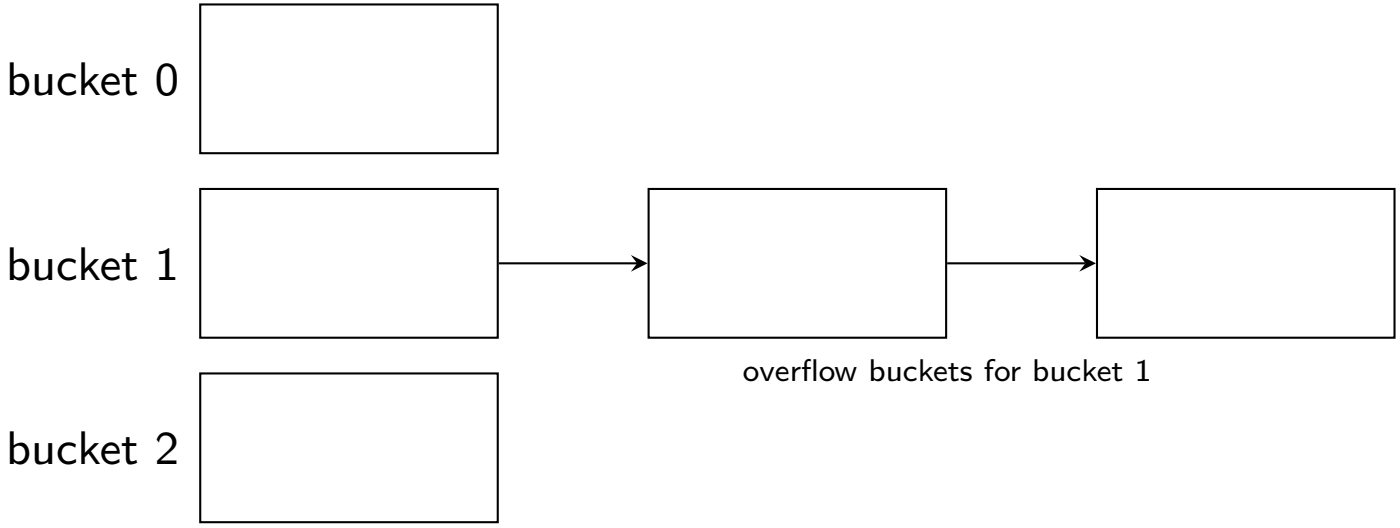
- Beispiel: 26 Buckets und eine Hash Funktion welche Filialnamen die mit dem i-ten Buchstaben beginnen dem Bucket i zuordnet.
  - keine Gleichverteilung, da es für bestimmte Anfangsbuchstaben erwartungsgemäß mehr Suchschlüssel gibt, z.B. erwarten wir mehr Filialen die mit B beginnen als mit Q.
- Beispiel: Hash Funktion die Kontostand nach gleich breiten Intervallen aufteilt: 1 - 10000 $\rightarrow$ 0, 10001 - 20000 $\rightarrow$ 1, usw.
  - uniform, da es für jedes Bucket gleich viele mögliche Werte von Kontostand gibt
  - nicht random, da Kontostände in bestimmten Intervallen häufiger sind, aber jedem Intervall 1 Bucket zugeordnet ist
- Typsiche Hash Funktion: Berechnung auf interner Binärdarstellung des Suchschlüssels, z.B. für String s mit n Zeichen, b Buckets:
  - $(s[0] + s[1] + \ldots + s[n-1]) \bmod b$, oder
  - $(31^{n-1}s[0] + 31^{n-2}s[1] + \ldots + s[n-1]) \bmod b$

Bucket Overflow/1

- Bucket Overflow: Wenn in einem Bucket nicht genug Platz für alle zugehörigen Datensätze ist, entsteht ein Bucket Overflow. Das kann aus zwei Gründen geschehen:
  - zu wenig Buckets
  - Skew: ungleichmäßige Verteilung der Hashwerte
- Zu wenig Buckets: die Anzahl n_B der Buckets muss größer gewählt werden als die Anzahl der Datensätze n geteilt durch die Anzahl der Datensätze pro Bucket f: $n_B > n/f$
- Skew: Ein Bucket ist überfüllt obwohl andere Buckets noch Platz haben. Zwei Gründe:
  - viele Datensätze haben gleichen Suchschlüssel (ungleichmäßige Verteiltung der Suchschlüssel)
  - Hash Funktion erzeugt ungleichmäßige Verteiltung
- Obwohl die Wahrscheinlichkeit für Overflows reduziert werden kann, können Overflows nicht gänzlich vermieden werden.
  - Overflows müssen behandelt werden
  - Behandlung durch Overflow Chaining

Bucket Overflow/2

- Overflow Chaining (closed addressing)
  - falls ein Datensatz in Bucket b eingefügt wird und b schon voll ist, wird ein Overflow Bucket b' erzeugt, in das der Datensatz gespeichert wird
  - die Overflow Buckets für Bucket b werden in einer Liste verkettet
  - für einen Suchschlüssel in Bucket b müssen auch alle Overflow Buckets von b durchsucht werden



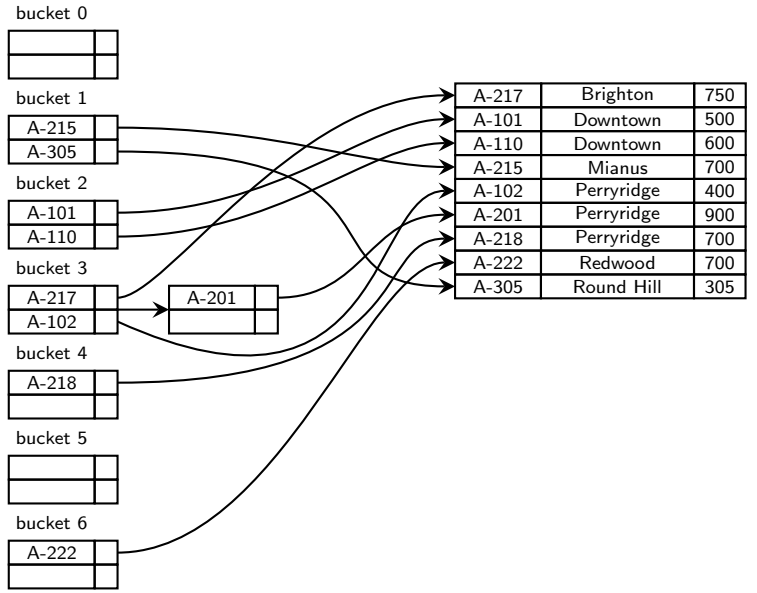
Bucket Overflow/3

- Open Addressing: Die Menge der Buckets ist fix und es gibt keine Overflow Buckets.
  - überzählige Datensätze werden in ein anderes (bereits vorhandenes) Bucket gegeben, z.B. das nächste das noch Platz hat (linear probing)
  - wird z.B. für Symboltabellen in Compilern verwendet, hat aber wenig Bedeutung in Datenbanken, da Löschen schwieriger ist

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

Inhalt

Probleme mit Statischem Hashing

- Richtige Anzahl von Buckets ist kritisch für Performance:
  - zu wenig Buckets: Overflows reduzieren Performance
  - zu viele Buckets: Speicherplatz wird verschwendet (leere oder unterbesetzte Buckets)
- Datenbank wächst oder schrumpft mit der Zeit:
  - großzügige Schätzung: Performance leidet zu Beginn
  - knappe Schätzung: Performance leidet später
- Reorganisation des Index als einziger Ausweg:
  - Index mit neuer Hash Funktion neu aufbauen
  - sehr teuer, während der Reorganisation darf niemand auf die Daten schreiben
- Alternative: Anzahl der Buckets dynamisch anpassen

Dynamisches Hashing

- Dynamisches Hashing (dynamic hashing): Hash Funktion wird dynamisch angepasst.

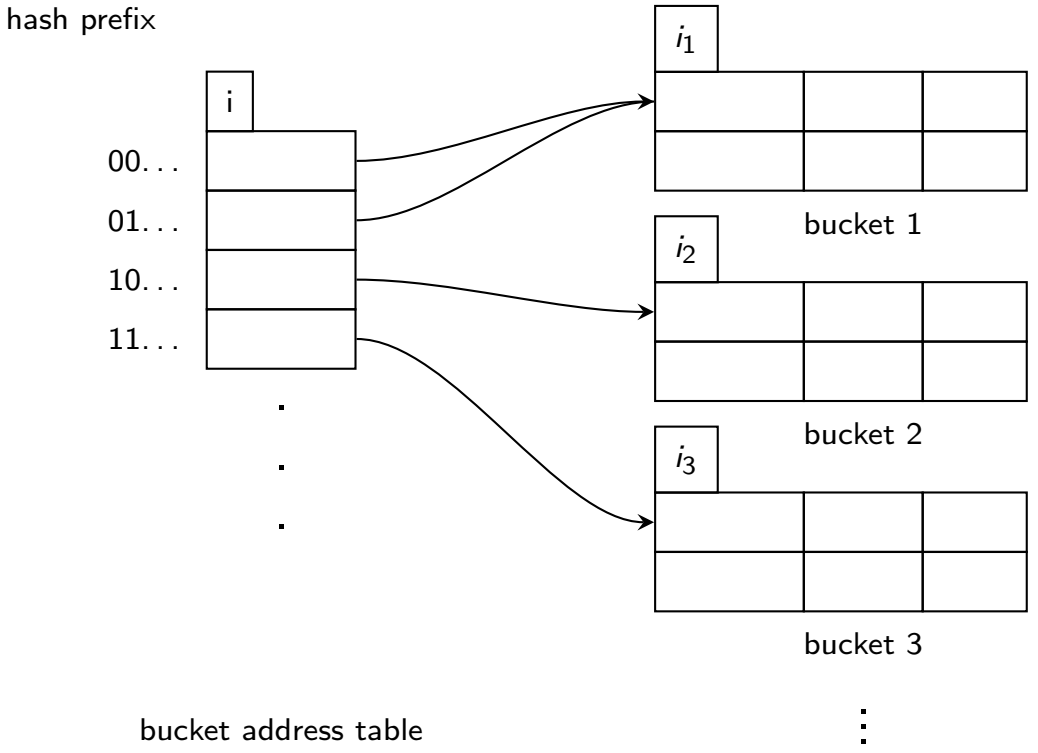
- Erweiterbares Hashing (extendible hashing): Eine Form des dynamischen Hashing.

Erweiterbares Hashing

- Hash Funktion h berechnet Hash Wert für sehr viele Buckets:
  - eine b-Bit Integer Zahl
  - typisch $b = 32$, also ~ 4 Milliarden (mögliche) Buckets
- Hash-Prefix:
  - nur die i höchstwertigen Bits (MSB) des Hash-Wertes werden verwendet
  - $0 \leq i \leq b$ ist die *globale Tiefe*
  - i wächst oder schrumpft mit Datenmenge, anfangs $i = 0$
- Verzeichnis: (directory, bucket address table)
  - Hauptspeicherstruktur: Array mit 2^i Einträgen
  - Hash-Prefix indiziert einen Eintrag im Verzeichnis
  - jeder Eintrag verweist auf ein Bucket
  - mehrere aufeinanderfolgende Einträge im Verzeichnis können auf dasselbe Bucket zeigen

Erweiterbares Hashing

- Buckets:
  - Anzahl der Buckets $\leq 2^i$
  - jedes Bucket j hat eine *lokale Tiefe* i_j
  - falls mehrere Verzeichnis-Pointer auf dasselbe Bucket j zeigen, haben die ensprechenden Hash Werte dasselbe i_j Prefix.
- Beispiel: $i = 2,\ i_1 = 1,\ i_2 = i_3 = 2,$



Erweiterbares Hashing: Suche

- Suche: finde Bucket für Suchschlüssel K
  1. berechne Hash Wert $h(K) = X$
  2. verwende die i höchstwertigen Bits (Hash Prefix) von X als Adresse ins Verzeichnis
  3. folge dem Pointer zum entsprechenden Bucket

Erweiterbares Hashing: Einfügen

- Einfügen: füge Datensatz mit Suchschlüssel K ein
  1. verwende Suche um richtiges Bucket j zu finden
  2. **If** genug freier Platz in Bucket j **then**
     - füge Datensatz in Bucket j ein
  3. **else**
     - teile Bucket und versuche erneut

Erweiterbares Hashing: Bucket teilen

- Bucket j teilen um Suchschlüssel K einzufügen

  If $i > i_j$ (mehrere Pointer zu Bucket j) **then**
  - lege neues Bucket z an und setze i_z und i_j auf das alte $i_j + 1$
  - aktualisiere die Pointer die auf j zeigen (die Hälfte zeigt nun auf z)
  - lösche alle Datensätze von Bucket j und füge sie neu ein (sie verteilen sich auf Buckets j und z)
  - versuche K erneut einzufügen

  Else if $i = i_j$ (nur 1 Pointer zu Bucket j) **then**
  - erhöhe i und verdopple die Größe des Verzeichnisses
  - ersetze jeden alten Eintrag durch zwei neue Einträge die auf dasselbe Bucket zeigen
  - versuche K erneut einzufügen

- Overflow Buckets müssen nur erzeugt werden, wenn das Bucket voll ist und die Hashwerte aller Suchschlüssel im Bucket identisch sind (d.h., teilen würde nichts nützen)

Integrierte Übung 2.2

Betrachten Sie die folgende Hashfunktion:

Schlüssel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Nehmen Sie Buckets der Größe 2 an und erweiterbares Hashing mit einem anfangs leeren Verzeichnis. Zeigen Sie die Hashtabelle nach folgenden Operationen:

- füge 1 Brighton und 2 Downtown Datensätze ein
- füge 1 Mianus Datensatz ein
- füge 1 Redwood Datensatz ein
- füge 3 Perryridge Datensätze ein

Erweiterbares Hashing: Löschen

- Löschen eines Suchschlüssels K
  1. suche Bucket j für Suchschlüssel K
  2. entferne alle Datensätze mit Suchschlüssel K
  3. Bucket j kann mit Nachbarbucket(s) verschmelzen falls
     - alle Suchschlüssel in einem Bucket Platz finden
     - die Buckets dieselbe lokale Tiefe i_j haben
     - die $i_j - 1$ Prefixe der entsprechenden Hash-Werte identisch ist
  4. Verzeichnis kann verkleinert werden, wenn $i_j < i$ für alle Buckets j

Integrierte Übung 2.3

Betrachten Sie die folgende Hashfunktion:

Schlüssel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Gehen Sie vom Ergebnis der vorigen Übung aus und führen Sie folgende Operationen durch:

- 1 Brighton und 1 Downtown löschen
- 1 Redwood löschen
- 2 Perryridge löschen

Erweiterbares Hashing: Pro und Kontra

- Vorteile von erweiterbarem Hashing
  - bleibt effizient auch wenn Datei wächst
  - Overhead für Verzeichnis ist normalerweise klein im Vergleich zu den Einsparungen an Buckets
  - keine Buckets für zukünftiges Wachstum müssen reserviert werden
- Nachteile von erweiterbarem Hashing
  - zusätzliche Ebene der Indirektion – macht sich bemerkbar, wenn Verzeichnis zu groß für den Hauptspeicher wird
  - Verzeichnis vergrößern oder verkleinern ist relativ teuer

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

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

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** BrNaIdx **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** BrNaIdx

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);

- Hash-partitionierter Index in Oracle:

  CREATE INDEX CustLNameIX **ON** customers (LName) **GLOBAL PARTITION BY HASH** (LName) **PARTITIONS** 4;

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:
  - Primary, Clustering und Sekundär
  - 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