universität wien

Kapitel 9

Speicher-, Datei- und Index-Strukturen

Inhalt

9.1 Datensatz Organisation

Die Datenbank wird durch eine Sammlung von **Dateien** (**Files**) gespeichert

Jede Datei besteht aus einer Sequenz von **Datensätzen** (**Records**)

Ein Datensatz ist eine **Sequenz von Feldern**

Ansatz

- Man nehme an, dass die Datensatzlänge fix ist
- Jede Datei besteht aus Datensätzen eines speziellen Typs
- Verschiedene Dateien repräsentieren verschiedene Relationen
- Dies ist der am einfachsten zu realisierende Fall, variabel lange Datensätze werden später behandelt

Datensätze fixer Länge

Simpler Ansatz

- Speichere Datensatz i beginnend von Byte n ∗ (i – 1), wobei n die Größe eines Datensatzes ist
- Zugriff auf Datensätze ist einfach, aber Datensätze können Blockgrenzen überschreiten
  - Modifikation: Datensätze dürfen keine Blockgrenzen überschreiten

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

Löschen von Datensätzen

Drei Alternativen

1. Schiebe Datensätze i + 1, . . ., n nach i, . . . , n – 1
2. Schiebe Datensatz n nach i
3. Verschiebe keine Datensätze, verkette alle freien Datensätze in einer Frei-Liste (free list)
   - Speichere die Adresse des ersten gelöschten Datensatzes im Dateikopf (Header)
   - Verwende diesen ersten Datensatz, um die Adresse des 2. gelöschten Datensatzes zu speichern, usw.
   - Man kann sich diese gespeicherten Adressen als Zeiger vorstellen, da sie auf den Ort des Datensatzes „zeigen"

Löschen des Datensatzes 2

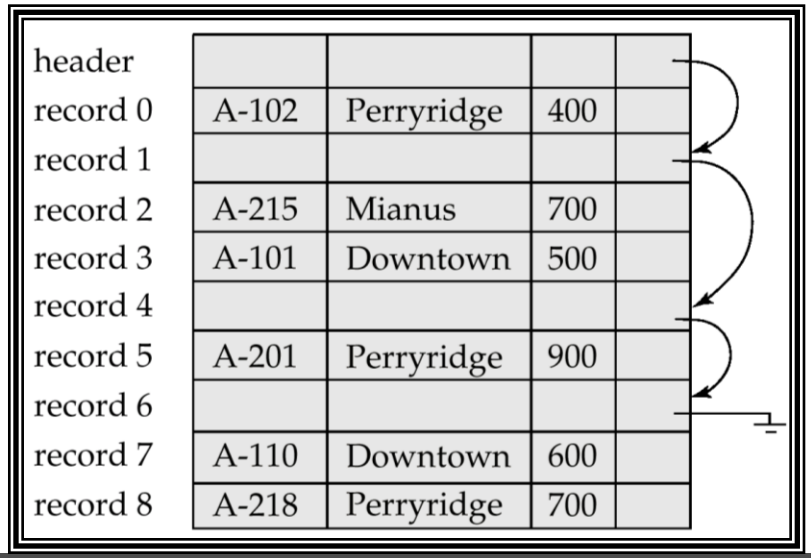
Alternative 1

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
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

Alternative 2

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	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

Alternative 3



Datensätze variabler Länge

Variabel-lange Datensätze kommen in verschiedenen Fällen vor

- Speicherung von unterschiedlichen Datensatz-Typen in einer Datei
- Datensatz-Typen, die variable Längen für ein oder mehrere Felder erlauben
- Datensatz-Typen, die wiederholende Felder erlauben

  Wurde in einigen älteren Datenmodellen verwendet

Speichermethoden

- **Byte-String** Repräsentation
- **Fixe-Längen** Repräsentation

Byte-String Repräsentation

Füge ein End-Of-Record (⊥) Kontrollzeichen an das Ende jedes Datensatzes

Datensatz ist dann ein String aufeinander folgender Bytes

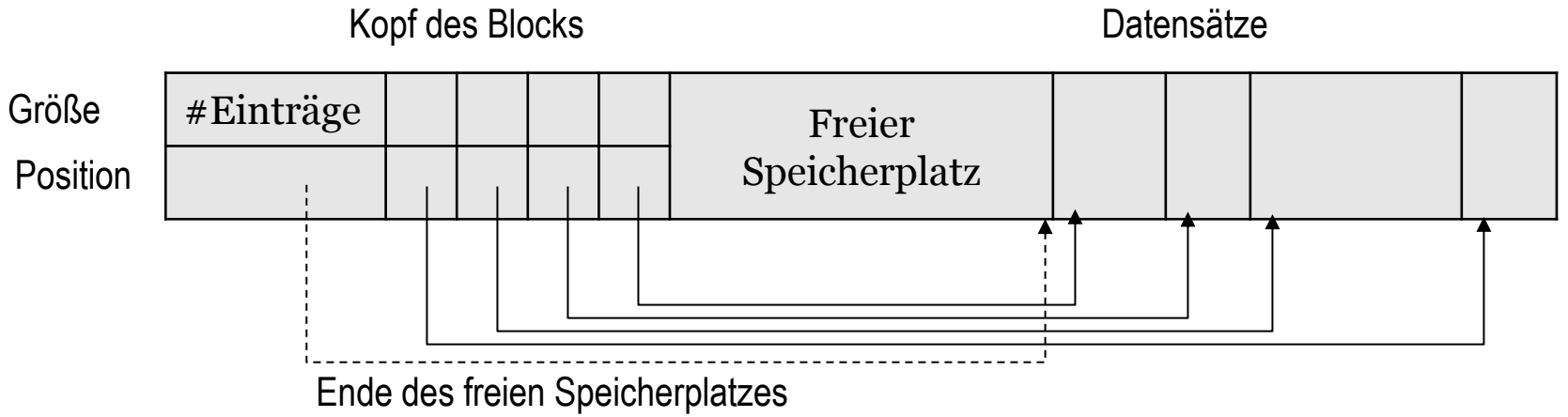
Probleme beim Löschen
-> Fragmentierung,

Probleme beim Wachsen von Datensätzen

In der Praxis Lösung mit **Slotted-page Header**

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

Byte-String Repräsentation – Slotted-page Header



Slotted-page Header (am Kopf jedes Blocks) enthalten

- Anzahl der Datensatz Einträge
- Ende des freien Speicherplatzes im Block
- Position und Größe jedes Datensatzes

Datensätze können in einer Seite herumgeschoben werden, um die Seite zusammenhängend (contiguous) ohne leere Bereiche zu erhalten

- Die Einträge im Header müssen entsprechend nachgeführt werden

Referenzen auf Datensätze sollten nicht direkt auf DS zeigen – sondern auf den Eintrag des DS im Header zeigen

Fixe-Längen Repräsentation - Platzreservierung

Repräsentation durch Datensätze fixer Länge – 2 Methoden

- Platzreservierung
- Listenrepräsentation

Platzreservierung – man kann Datensätze fixer Länge einer bekannten maximalen Länge verwenden

Ungenutzter Platz in kürzeren Datensätzen wird mit Nullen oder End-Of-Record Symbolen gefüllt.

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

Fixe-Längen Repräsentation – Listenrepräsentation

Listenrepräsentation

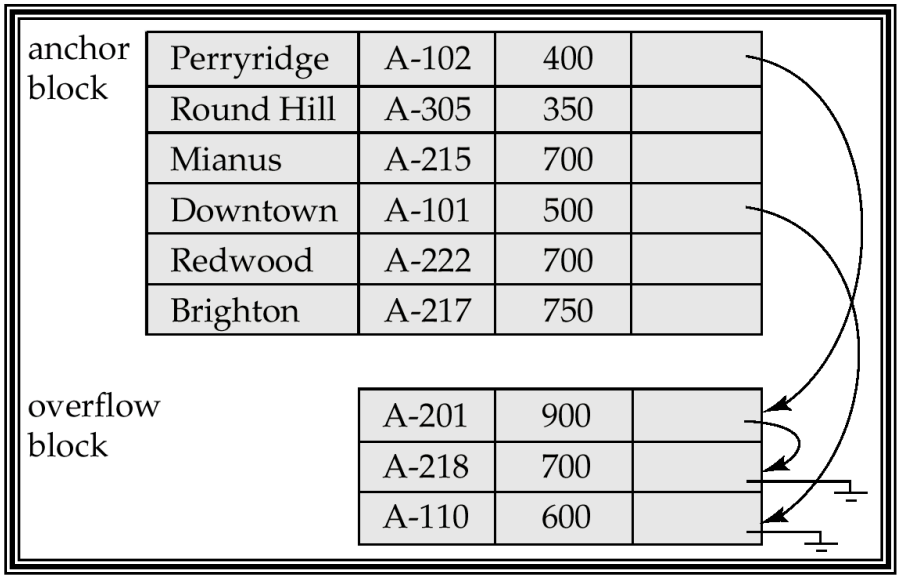
- Wird durch eine Liste von Teil-Datensätzen fixer Länge repräsentiert, die durch Zeiger verkettet sind
- Kann verwendet werden, auch wenn man die maximale Datensatz-Länge nicht kennt

Nachteil der Zeiger Struktur

- Platz wird bei allen Teil-Datensätzen außer dem ersten in der Kette verschwendet

Lösung: 2 Arten von Blöcken in der Datei

- Anker (Anchor) Block
  - enthält die ersten Teil-Datensätze der Kette
- Überlauf (Overflow) Block
  - enthält alle anderen Teil-Datensätze außer den ersten einer Kette



9.2 Dateiorganisation

Heap – ein Datensatz kann an beliebiger Stelle in der Datei stehen, dort wo Platz ist

Sequentiell – speichere Datensätze in sequentieller Reihenfolge basierend auf dem Schlüsselwert jedes Datensatzes

Hashing – der Datensatz wird in dem Block gespeichert, den eine Hashfunktion aus einigen Attributwerten des Datensatzes berechnet

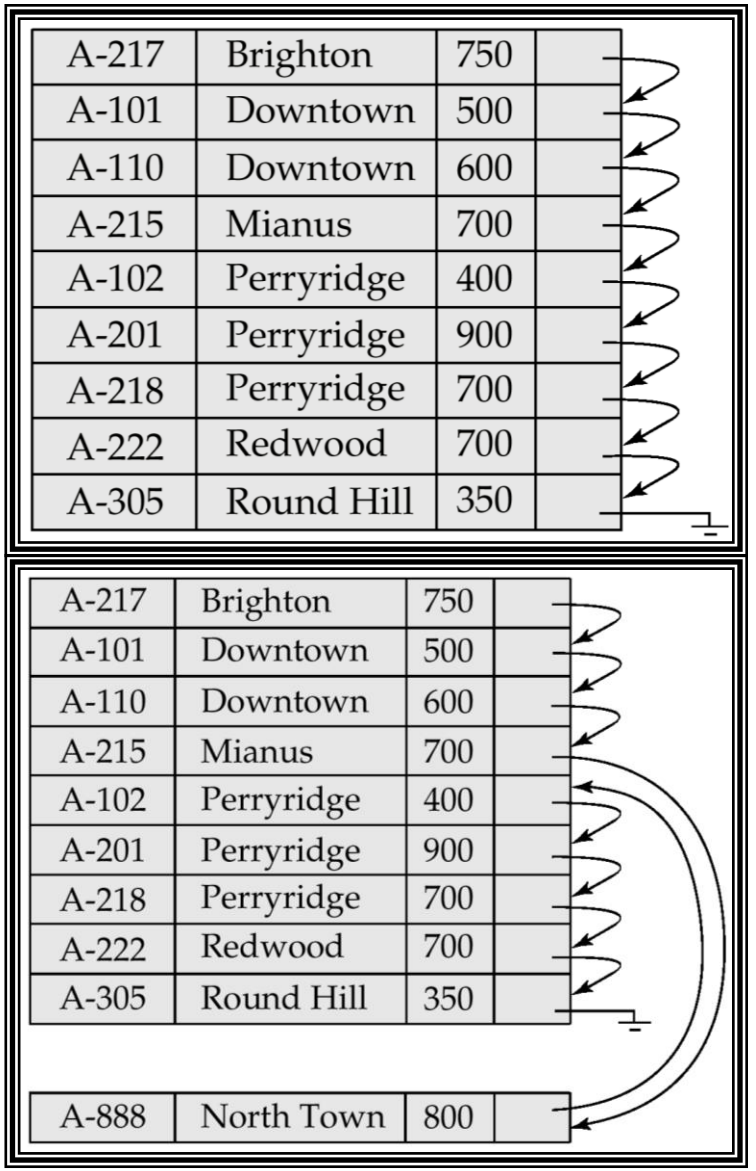
Die Datensätze der einzelnen Relationen werden meist in separaten Dateien gespeichert

Bei einer **Cluster Dateiorganisation** können Datensätze verschiedener Relationen in derselben Datei gespeichert werden

- Motivation: speichere in Beziehung stehende Datensätze im selben Block um I/O zu minimieren

Sequentielle Dateiorganisation

- Die Datensätze in der Datei sind spezifisch nach einem Schlüsselwert **geordnet**
  - Praktisch für Applikationen, die die gesamte Datei **sequentiell** abarbeiten müssen
- **Löschen** – Einsatz von Zeiger Ketten
- **Einfügen** – stelle die Position fest, wo der Datensatz eingefügt werden muss
  - Falls Platz frei, füge dort ein
  - Falls kein freier Platz, füge den Datensatz in einen Überlauf Block ein
    - In jedem Fall müssen die Zeiger der Kette nachgeführt werden
- Notwendigkeit die Datei von Zeit zu Zeit zu **reorganisieren**, um die sequentielle Ordnung zu erhalten



Cluster Dateiorganisation

Ein simpler Datei Struktur Ansatz speichert jede Relation in einer separaten Datei

Bei der **Cluster Dateiorganisation** werden stattdessen mehrere Relationen in einer Datei gespeichert

- z.B. Cluster Organisation von *kunde* und *kreditnehmer*
- Gut für Abfragen auf *kunde* $\bowtie$ *kreditnehmer* und auf einzelnen Kunden und seine Konten
- Schlecht für Abfragen nur auf Kunde
- Führt zu variabel langen Datensätzen

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

9.3 Speicherung von Data-Dictionary, Blobs und Clobs

universität wien

Das Data Dictionary (auch System Katalog) speichert **Metadaten**, d.h. **Daten über Daten**, wie

- Informationen über Relationen
  - Namen von Relationen
  - Namen und Typen der Attribute jeder Relation
  - Namen und Definitionen von Views (Sichten)
  - Integritätsbedingungen
- Benutzer und Accounting Information, inklusive Passworte
- Statistische und deskriptive Daten
  - Anzahl der Tupel in jeder Relation
- Information über die physische Dateiorganisation
  - Wie die Relation gespeichert ist (sequentiell/hash/…)
  - Physischer Speicherort der Relation
    - Betriebssystem Namen
    - Plattenadressen der Blöcke die die Datensätze einer Relation enthalten
- Information über Indizes

Speicherung des Data Dictionary (2)

Zwei Möglichkeiten zur Speicherung der Katalog Struktur

- Spezialisierte Datenstrukturen für effizienten Zugriff, oder
- Eine Menge von Relationen basierend auf den existierenden Systemmöglichkeiten (üblicherweise der Normalfall)

Eine mögliche Katalog Repräsentation

Relation-metadata = (relation-name, num-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)

Large Objects – Große Objekte

- **Binary Large Objects** (**blobs**) und **Character Large Objects** (**clobs**)
  - Text Dokumente
  - Graphische Daten, wie Bilder, Audio und Video Daten, etc.
- Große Objekte müssen oft in einer zusammenhängenden Sequenz von Bytes gespeichert werden, wenn sie in den Hauptspeicher kommen
  - Falls ein Objekt größer als eine Seite ist, müssen zusammenhängende Pufferseiten zur Speicherung alloziert werden
  - Es kann von Vorteil sein, den direkten Zugriff auf die Daten zu verbieten und den Zugriff nur durch ein Dateisystem ähnliches API zu erlauben, um die Notwendigkeit für zusammenhängenden Speicher zu vermeiden

9.4 Indexmechanismen

Indexmechanismen **beschleunigen den Zugriff** auf Daten

- z.B. Autoren-Katalog in der Bibliothek

Suchschlüssel = Attributmenge um Datensätze in einer Datei zu finden

Index Datei besteht aus Datensätzen (Indexeinträgen) der Form

Suchschlüssel	Zeiger

- Index Dateien sind üblicherweise **deutlich kleiner** als die Originaldatei

Zwei grundsätzliche Arten von Indizes

- **Geordnete Indizes**: Suchschlüssel sind in Sortierreihenfolge gespeichert
- **Hash Indizes**: Suchschlüssel werden gleichmäßig über Blöcke (buckets) durch eine Hash-Funktion verteilt

Kennzahlen für die Bewertung von Indizes

- Effizient unterstützte Zugriffsarten
  - Datensätze mit einem angegebenem Attributwert, oder
  - Datensätze mit Attributwerten, die in ein angegebenes Intervall fallen
- Zugriffszeit, Einfügezeit, Löschzeit
- Zusätzlicher Platzverbrauch

Vergleiche VO AD: Qualitative und quantitative Merkmale von Datenstrukturen

Geordnete Indizes

In einem geordneten Index werden die Einträge entsprechend der Ordnung ihrer Suchschlüssel gespeichert

- Suchschlüssel werden miteinander verglichen

Primärindex: der Index, der in der sequentiellen Datei die Ordnung bestimmt

Man nennt ihn auch **Clustering Index**

- Der Suchschlüssel eines Primärindex ist üblicherweise der Primärschlüssel, muss es aber nicht sein

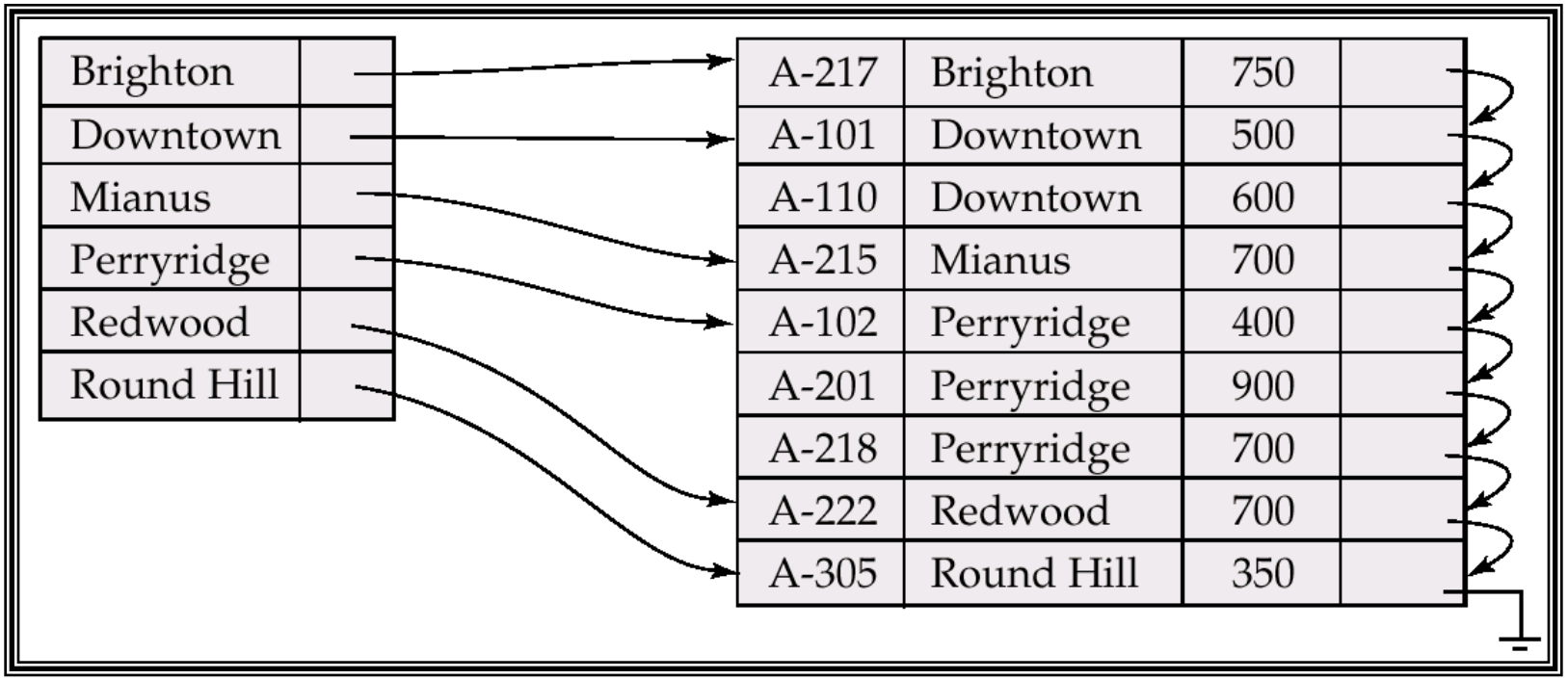
Sekundärindex: ein Index der eine vom Primärschlüssel unterschiedliche Ordnung spezifiziert

- Nennt man Non-Clustering Index

Index-sequentielle Datei

Geordnete sequentielle Datei mit einem Primärindex

In der Darstellung mit dichtem Index



Dichter und dünner Index

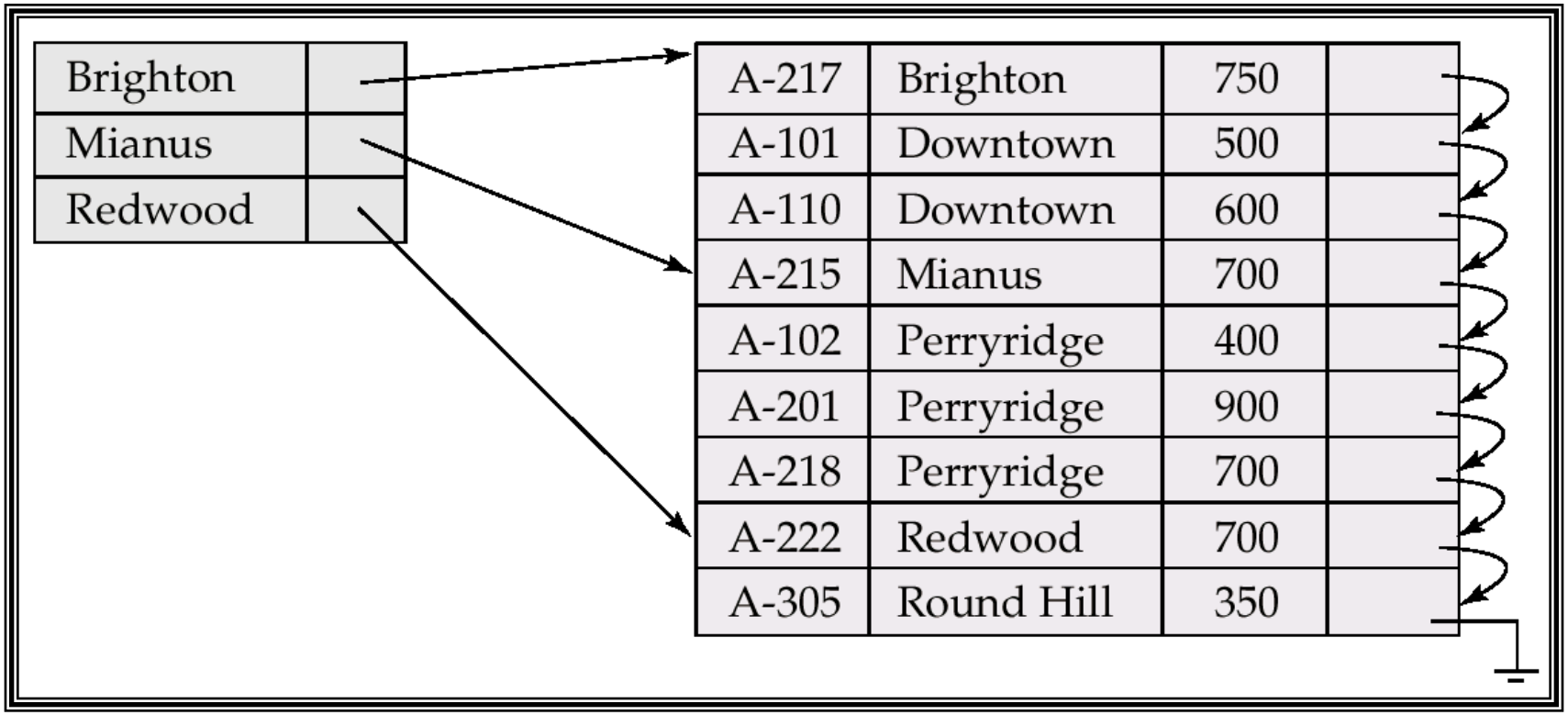
Dichter Index

- Indexeinträge existieren für jeden Suchschlüssel der Datei
  - Siehe Beispiel letzte Folie

Dünner Index

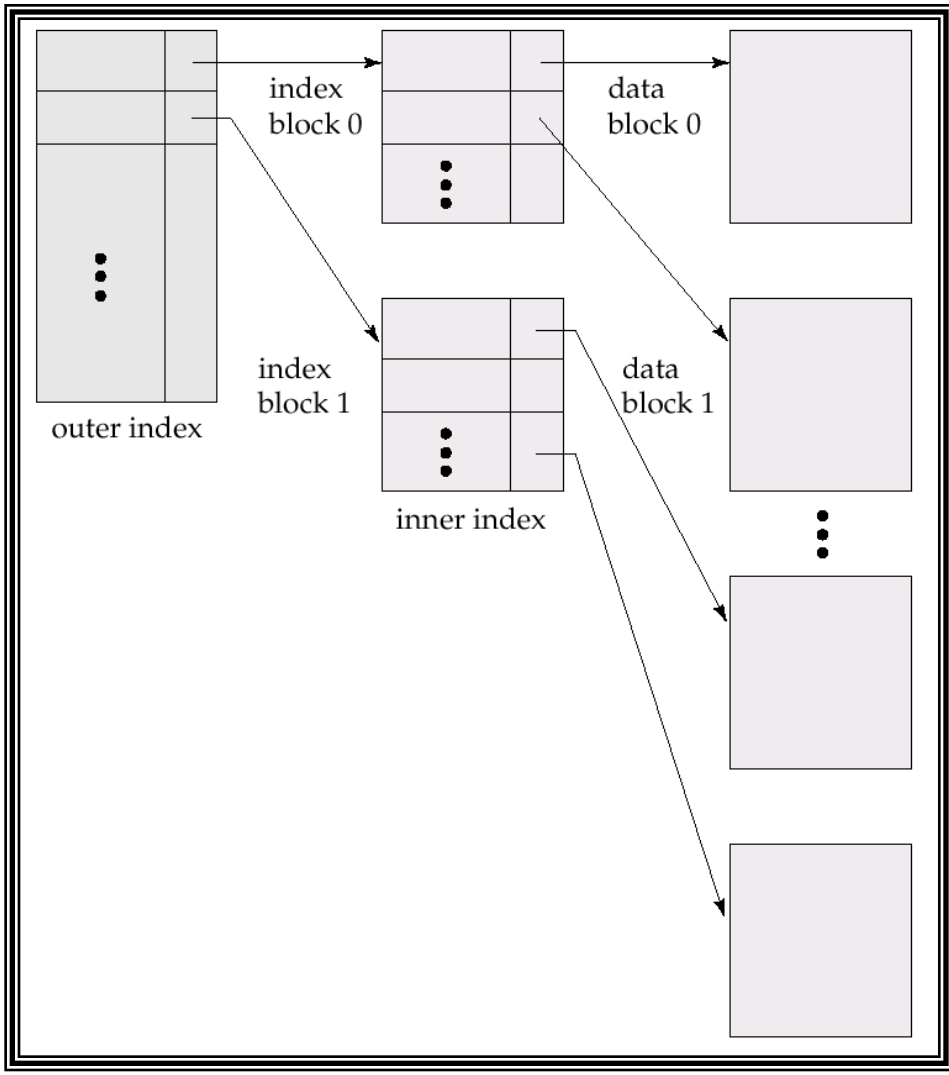
- Enthält Indexeinträge nur für einige Suchschlüssel
- Anwendbar, wenn Datensätze sequentiell nach dem Suchschlüssel **sortiert** sind
- **Suche** nach Datensatz mit Suchschlüssel K
  1. Finde Indexeintrag mit größtem Suchschlüssel < K
  2. Starte sequentielle Suche vom Datensatz, auf den der Indexeintrag zeigt
- Benötigt weniger Administrationsaufwand für Einfügen und Löschen
- Generell langsamer in der Suche als dichter Index
- Guter **Kompromiss**
  - Dünner Index mit einem **Indexeintrag für jeden Block** in der Datei, entsprechend dem kleinsten Suchschlüsselwert im Block
  - (vergleiche mit **Intervallbasierten Baumstrukturen**, VO AD)

Beispiel für Dünnen Index



Mehr-Ebenen (Multilevel) Index

- Wenn Primärindex nicht in den Hauptspeicher passt, werden die Zugriffe teuer
- Um die Zugriffe zu Indexeinträgen zu minimieren, behandle Primärindex auf der Platte als sequentielle Datei und baue darüber einen dünnen Index
  - Äußerer (Outer) Index – dünner Index auf dem Primärindex
  - Innerer (Inner) Index – die Primärindexdatei
- Falls selbst der äußere Index zu groß für den Hauptspeicher wird, kann eine weitere Ebene eingefügt werden, usw.
- Die Indizes auf allen Ebenen müssen beim Einfügen und Löschen nachgeführt werden
- aka **Index-sequentielle Datei**



Sekundärindizes

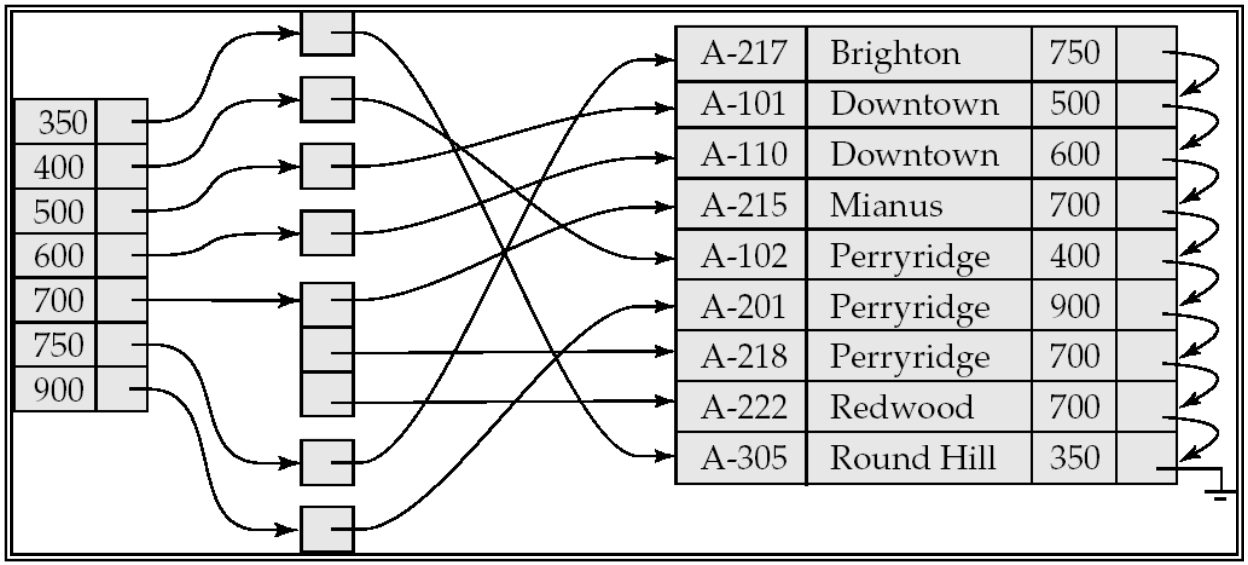
Häufig will man alle Datensätze finden, die eine spezielle Bedingung erfüllen, wobei die Felder der Bedingung aber nicht der Suchschlüssel des Primärindex sind

- Beispiel: In der *konto* Relation, die sequentiell nach der Kontonummer gespeichert ist, will man alle Konten einer speziellen Filiale finden

Sekundärindizes müssen Einträge für jeden Suchschlüsselwert besitzen (dichter Index)

- Indexeintrag zeigt auf einen Block, der Zeiger auf alle eigentlichen Datensätze mit dem spezifischen Suchschlüssel enthält

Beispiel:
Sekundärindex auf *kontostand*
Feld von *konto*



Primär- und Sekundärindex

Indizes liefern deutliche Vorteile für die Suche nach Datensätzen

Wenn eine Datei geändert wird, muss jeder Index auf der Datei nachgeführt werden, was großen Aufwand für Datenbank Änderungen bedeutet

Sequentieller Scan über dem Primärindex ist effizient, aber über Sekundärindex aufwändig

- Jeder Datensatz-Zugriff kann einen neuen Block auf der Platte referenzieren

B+-Baum Index Datei

B+-Baum Indizes sind eine Alternative zu den Index-sequentiellen Dateien

Nachteil von Index-sequentiellen Dateien

- Leistungsfähigkeit lässt nach, wenn die Datei wächst, da viele Überlauf-Blöcke erzeugt werden
- Periodische **Reorganisation** notwendig

Vorteil von **B+-Baum Indizes**

- **Reorganisieren sich automatisch** selbst durch kleine, lokale Änderungen beim Einfügen und Löschen
- Keine Reorganisation der gesamten Datei notwendig

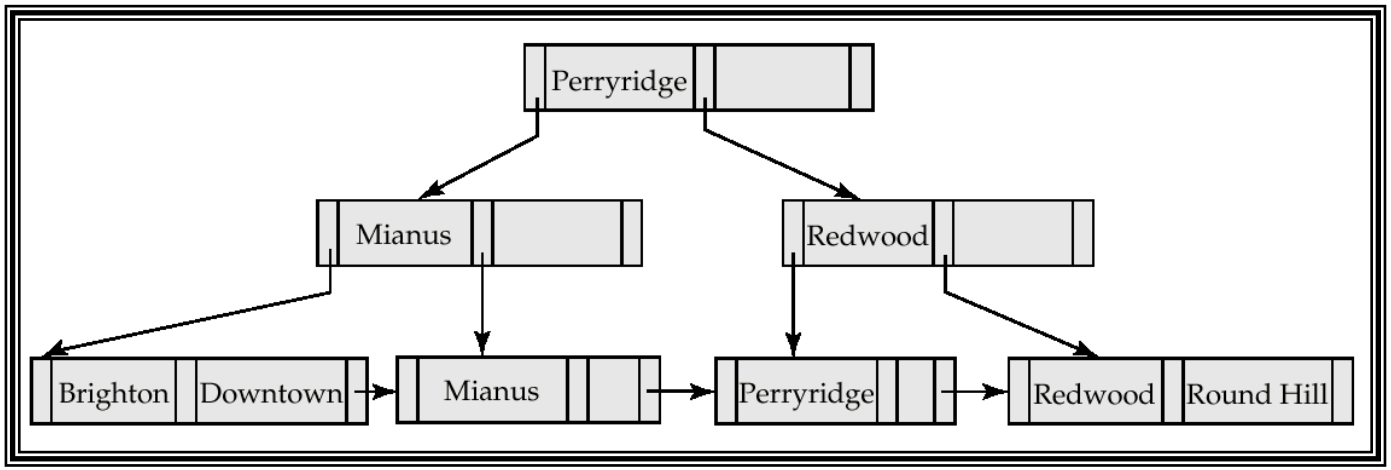
Nachteil von B+-Baum Indizes

- Aufwand beim Einfügen und Löschen, Speicherplatzaufwand

Vorteile wiegen aber die Nachteile bei Weitem auf

Interne und Externe Knoten im B+-Baum

B+-Baum für *konto* Datei



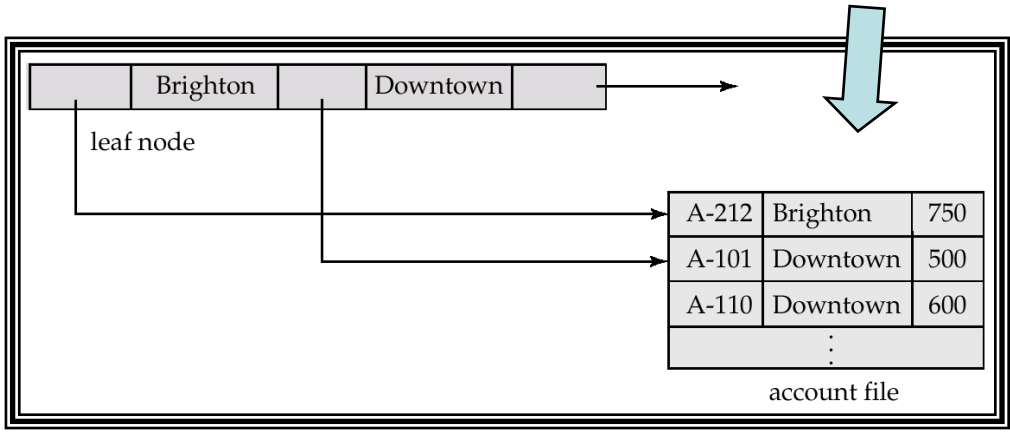
Vergleiche mit VO AD!

Externe Knoten

Enthalten die zu verwaltende Information realisiert durch Blattknoten

Indexknoten

- realisiert durch **interne Knoten**
- Erlauben effizienten Zugriff auf die **externen Knoten**
- Definieren die **Intervallbereiche** der Elemente, die im zugeordneten Teilbaum gespeichert sind.



Beobachtungen über B^+-Bäume

Da die Verbindung zwischen Blöcken mit Zeigern geschieht, müssen logisch nahe Blöcke nicht physisch nah gespeichert werden

Die Nicht-Blatt Ebenen eines B+-Baums formen die Hierarchie eines dünnen Index

Der B+-Baum enthält relativ wenige Ebenen (logarithmisch zur Gesamtdatei)

Effiziente Algorithmen -> O(log n)

- Ein Knoten ist üblicherweise so groß wie ein Platten Block, z.B. typisch wären 4 KBytes und maximal 100 Indexwerte pro Knoten (~40 Bytes pro Index Eintrag; Ordnung des Baums 50)
- Mit 1 Million Suchschlüsselwerten folgt dann: $\lceil log_{51}(1{,}000{,}000) \rceil = 4$ Knoten müssen zugegriffen werden

Hashing

Bei einer **Hash Dateiorganisation** berechnet man das **Bucket** eines Datensatzes direkt aus seinem Suchschlüssel durch eine Hash-Funktion

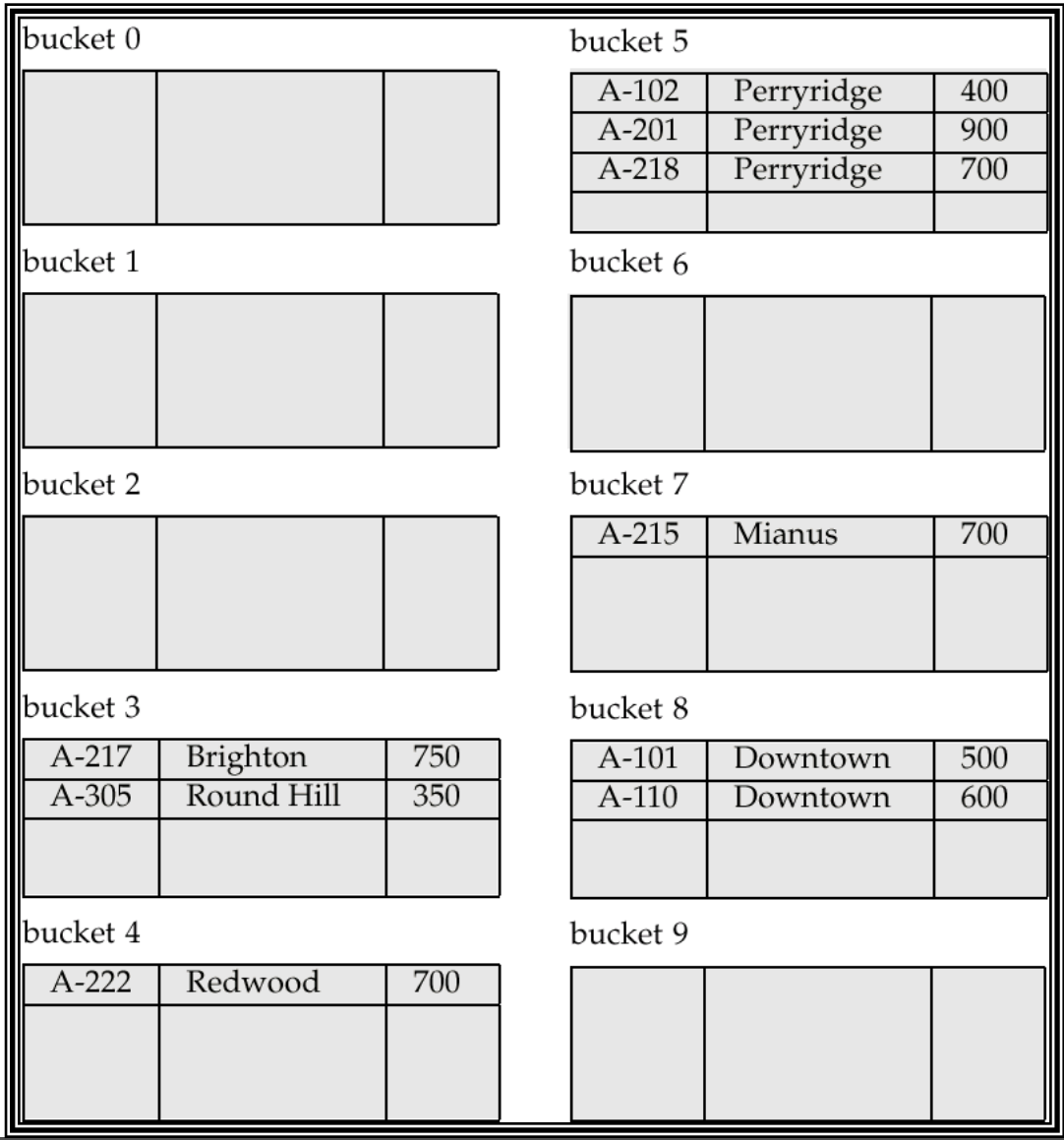
- Ein **Bucket** ist eine Speichereinheit, die einen oder mehrere Datensätze enthält (typischerweise ein Platten Block)

Die **Hash-Funktion** h ist eine (im allgemeinen nicht injektive) Funktion über der Menge aller Suchschlüssel auf die Menge aller Bucket Adressen

- Die Hash-Funktion verwendet man zur Suche, Einfügen und Löschen von Datensätzen
- Datensätze mit unterschiedlichen Suchschlüsseln können auf dasselbe Bucket abgebildet werden, wodurch das Bucket sequentiell abgesucht werden muss

Bespiel einer Hash Dateiorganisation

- Hash Dateiorganisation des *konto* Files, mit *filiale_name* als Suchschlüssel
- Die Hash-Funktion h liefert die Summe der binären Repräsentation der Zeichen modulo 10
  - z.B.
    - h(Perryridge) = 5
    - h(Round Hill) = 3
    - h(Brighton) = 3



Kollision

Eine **Kollision** tritt auf, wenn zwei oder mehrere Suchschlüsselwerte auf dieselbe Tabellenposition abgebildet (gehasht) werden.

Eine Kollision löst eine notwendige **Kollisionsbehandlung** aus.

- D.h., für den kollidierenden Suchschlüsselwert muss ein Ausweichsplatz gefunden werden.

Maßnahmen zur Kollisionsbehandlung sind meist recht aufwändig und beeinträchtigen die Effizienz von Hashverfahren.

- der Kollisionspfad ist definiert durch die Ausweichplätze aller Elemente, die auf den gleichen Platz gehasht wurden
- Wir kennen 2 simple Kollisionsbehandlungen
  - **Separate Chaining** und **Double Hashing**

Hash Indizes

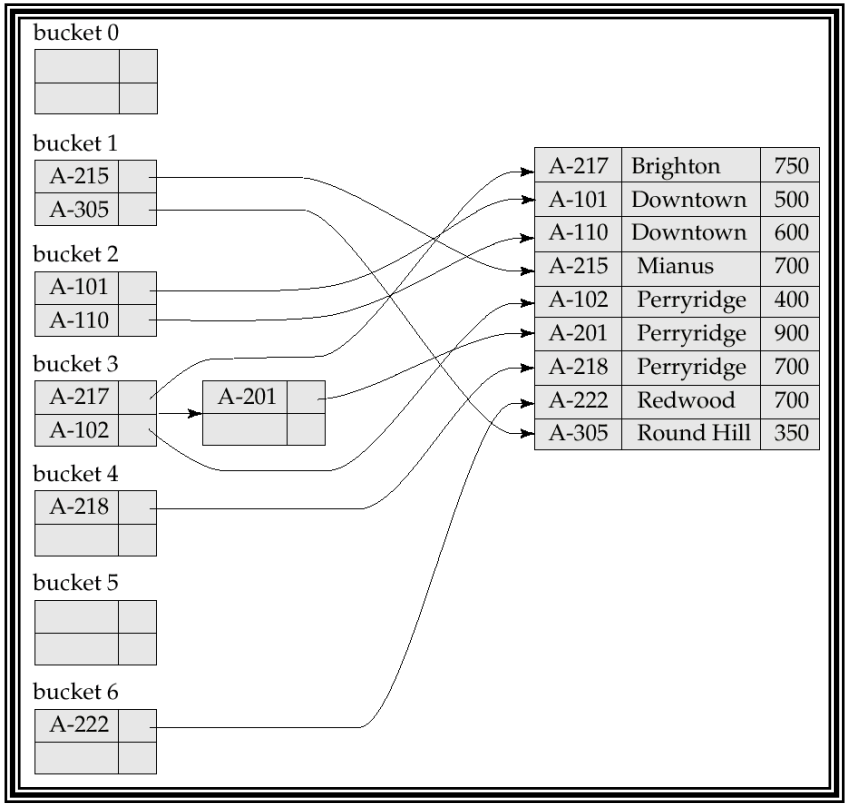
Hashing kann nicht nur für die Dateiorganisation verwendet werden, sondern auch für die Index Erzeugung

Ein **Hash Index** organisiert die Suchschlüssel mit ihren zugeordneten Datensatz-Zeigern in einer Hash Datei

- Hash Indizes können immer nur Sekundärindizes sein

Kein Clustering Index

- Falls die Datei selbst als Hash Datei organisiert ist, ist ein Hash Index unnötig



Dynamisches Hashing

Gut für Datenbanken die wachsen und schrumpfen

Dynamische Hash-Verfahren versuchen im Falle von Kollisionen die ursprüngliche Hashtabelle zu erweitern

- Anlegen von Überlaufbereichen (z.B. lineare Listen)
- Vergrößerung des Primärbereichs (ursprüngliche Tabelle)

„Two-Disk-Access“ Prinzip

- Finden eines Suchschlüssel mit maximal 2 Platten Zugriffen

Hashverfahren für externe Speichermedien

- Linear Hashing
  - Verzeichnisloses Hashverfahren
  - Primärbereiche, Überlaufbereiche
- Extendible Hashing
  - Verzeichnis mit binärer Expansion
  - Primärbereiche
- Bounded Index Size Extendible Hashing
  - Verzeichnis mit beschränkter Größe
  - Binäre Expansion der Blöcke

Kennen wir aus AlgoDat!

Vergleich von Geordneten Indizes und Hashing

Kosten der periodischen Reorganisation

Häufigkeit des Einfügens und Löschen

Optimierungen der Durchschnittlichen Zugriffszeit auf Kosten der worst-case Zeit

Erwartete Abfragetypen

- Hashing ist üblicherweise besser bei Zugriff auf Datensätze über ihren Suchschlüsselwert (Exact Match Query)
- Bei Bereichsabfragen sind geordnete Indizes vorzuziehen (Interval Query)