

# Datenbanken 2

## Interne Dateiorganisation

Nikolaus Augsten  
nikolaus.augsten@plus.ac.at  
FB Informatik  
Universität Salzburg



WS 2022/23  
Version 11. Oktober 2022

# Inhalt

- 1 Sequentielle Dateien
- 2 Statisches Hashing
- 3 Dynamisches Hashing

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.

- 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 Dateioorganisation
    - wie ist eine Relation gespeichert (sequentiell/Hash/...)
    - physischer Speicherort (z.B. Festplatte)
    - Dateiname oder Adresse des ersten Blocks auf der Festplatte
  - Information über Indexstrukturen

- **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>

# Inhalt

- 1 Sequentielle Dateien
- 2 Statisches Hashing
- 3 Dynamisches Hashing

# Sequentielle Datei/1

- **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)

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



# Sequentielle Datei/2

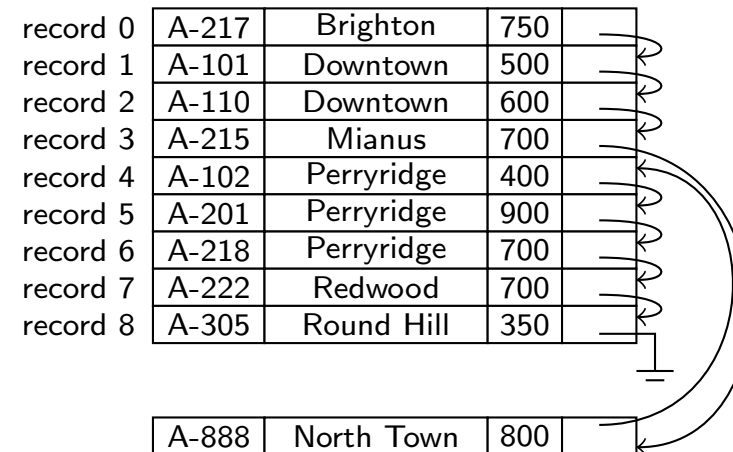
- Physische Ordnung erhalten ist schwierig.
  - Löschen:
    - Datensätze sind mit Pointern verkettet (verkettete Liste)
    - gelöschter Datensatz wird aus der verketteten Liste genommen
- leere Zwischenräume reduzieren Datendichte

- 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





# Inhalt

- 1 Sequentielle Dateien
- 2 Statisches Hashing**
- 3 Dynamisches Hashing

# 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 Blöcke 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.,  $\text{code}(B)=2$ .
- Hash Funktion  $h$ 
  - Summe der Codes aller Zeichen modulo 10:
  - $h(\text{Perryridge}) = 125 \bmod 10 = 5$
  - $h(\text{Round Hill}) = 113 \bmod 10 = 3$   
( $\text{code}(' ') = 0$ )
  - $h(\text{Brighton}) = 93 \bmod 10 = 3$

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	Perryridge	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

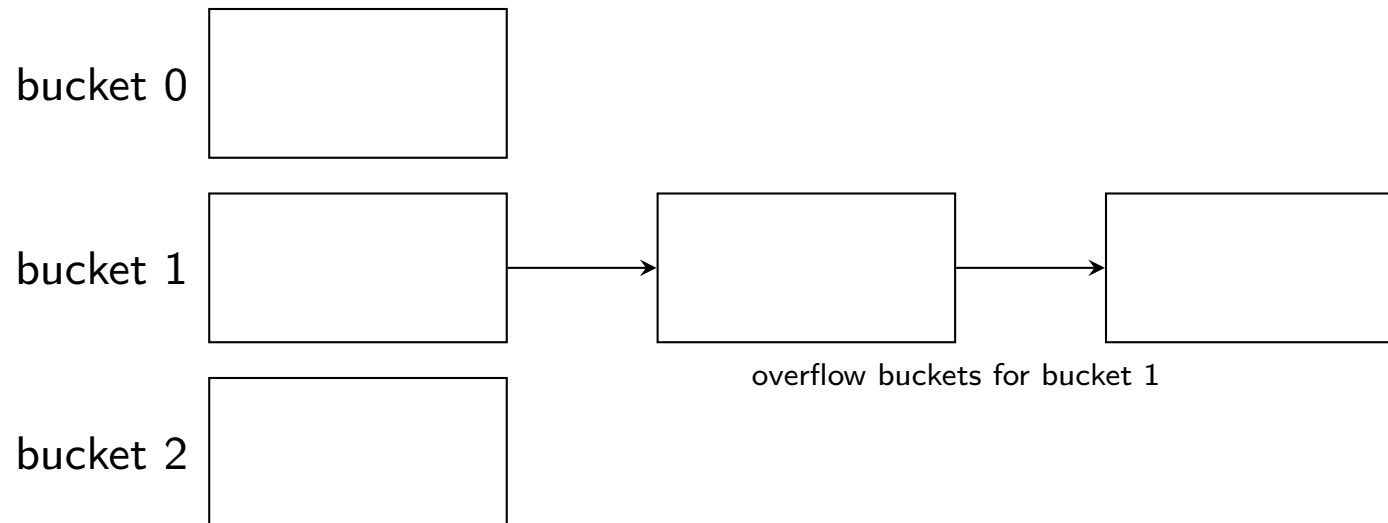
- **Beispiel:** 26 Buckets und eine Hash Funktion welche Filialnamen die mit dem  $i$ -ten Buchstaben beginnen dem Bucket  $i$  zuordnet.
  - keine Gleichverteilung, da es in der Domäne der Filialnamen (Menge aller möglichen Filialnamen) vermutlich mehr Filialen gibt die mit B beginnen als mit X.
- **Beispiel:** 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
- **Typische Hash Funktion:** Berechnung auf interner Binärdarstellung des Suchschlüssels, z.B. für String  $s$  mit  $n$  Zeichen,  $b$  Buckets:
  - $(s[0] + s[1] + \dots + s[n - 1]) \bmod b$ , oder
  - $(31^{n-1}s[0] + 31^{n-2}s[1] + \dots + s[n - 1]) \bmod b$

# 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 Verteilung der Suchschlüssel)
  - Hash Funktion erzeugt ungleichmäßige Verteilung
- 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 (besser geeignet für Hauptspeicher, empfindlicher gegenüber Hash Funktion, max. Füllgrad von ca. 70%)



# Inhalt

- 1 Sequentielle Dateien
- 2 Statisches Hashing
- 3 Dynamisches Hashing**

# 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** der Hash Datei als einziger Ausweg:
  - Hash Datei 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

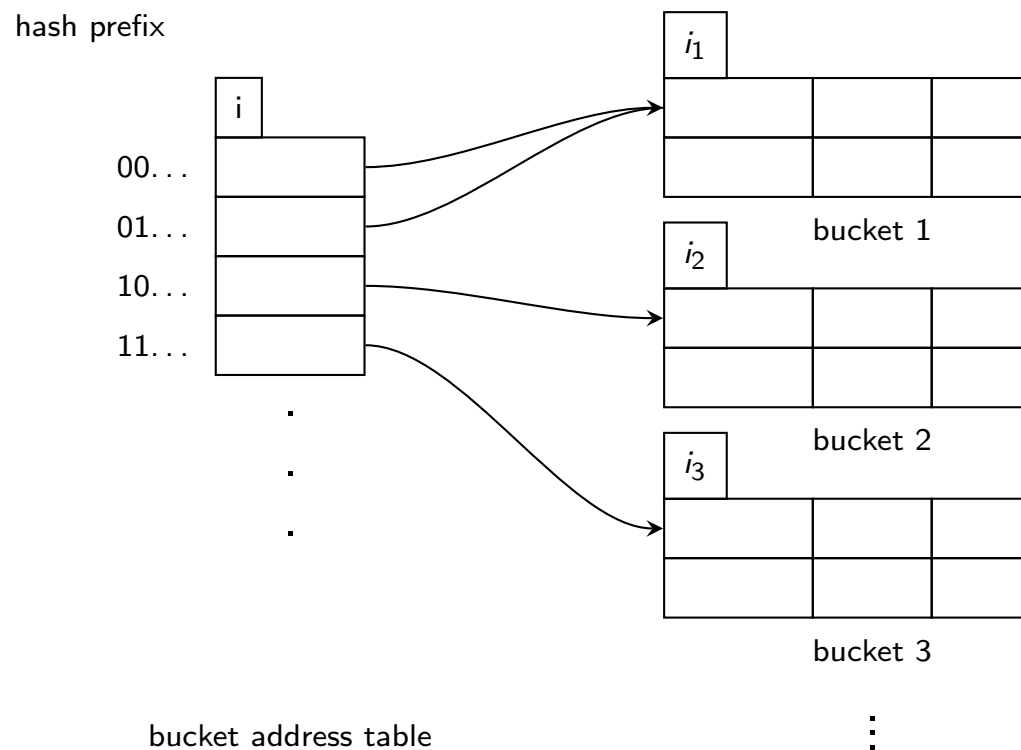
- **Dynamisches Hashing** (dynamic hashing): Hash Funktion wird dynamisch angepasst.
- **Erweiterbares Hashing** (extendible hashing): Eine Form des dynamischen Hashing.

# Erweiterbares Hashing

- Hash Funktion  $h$  berechnet Hash Wert für sehr viele Buckets:
  - eine  $b$ -Bit Integer Zahl
  - typisch  $b = 32$ , also  $\sim 4$  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 entsprechenden 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
- setze die lokalen Tiefen  $i_z$  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  $z$ )

**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

- **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 2.1

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 sind
  4. Verzeichnis kann verkleinert werden, wenn  $i_j < i$  für alle Buckets  $j$

# Integrierte Übung 2.2

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