

Hash-basierte Dateien

Hashing

- Hash-Verfahren ermöglichen es, die Adresse eines Datensatzes basierend auf dem Wert eines Feldes zu finden
- Idee: Verwendung einer Hashfunktion, die den Wert eines Suchschlüssels in einen Bereich von Behälternummern abbildet, um die Seite mit dem Datensatz zu finden
- Im idealen Fall: die Hashfunktion berechnet direkt die Adresse des Datensatzes
$$h:\{S_1, S_2, \dots, S_n\} \rightarrow A, h(S_i) = \text{Adresse des } i\text{-ten Datensatzes}$$
- Solche Funktionen sind schwer zu finden:
 - Alle möglichen Suchschlüsselwerte müssen von Anfang an bekannt sein
 - Für große Dateien ist es unmöglich die Bijektivität zu erhalten

Hashing – Behälter (buckets)

- Lösung: Kollisionen sind erlaubt
 - $h(S_i) = h(S_j), i \neq j$, h – Hash-Funktion
- Gilt für zwei Schlüssel S_1 und S_2 ,
dass $h(S_1) = h(S_2)$ ist, nennt man S_1 und S_2 *synonym*
- Formell ist eine Hashfunktion eine Abbildung $h: S \rightarrow B$, wobei S eine Schlüsselmenge und B eine Nummerierung der n Behälter ist
- Normalerweise ist die Anzahl der möglichen Elemente in der Schlüsselmenge viel größer als die Anzahl der Behälter ($|S| > |B|$)
→ die Hashfunktion ist nicht injektiv, sollte aber die Elemente von S gleichmäßig auf B verteilen



Probleme die bei Hashing vorkommen

- Verteilungsproblem – nachdem wir den Hashing Algorithmus ausgewählt haben, haben wir keine Kontrolle über die Verteilung der Daten in dem Speicherplatz
- Clustering Problem – wenn die Datensätze nicht gleichmäßig verteilt werden (zu viele Datensätze in einem Behälter und sehr wenige in anderen)
- Überlaufproblem – wenn die Behälter nicht groß genug sind, dann kann ein Überlauf auftreten

Hashfunktion

- Voraussetzungen für eine gute Hashfunktion:
 - Schnelle Auswertung
 - Minimiert die Anzahl der Kollisionen (verteilt die Datensätze gleichmäßig in den Behälter)
- Nehmen wir an, dass wir Datensätze in 41 Behälter verteilen wollen

Die Wahrscheinlichkeit:

- den 1sten Datensatz in einem leeren Behälter zu verteilen = $41/41$
- den 2ten Datensatz in einem leeren Behälter zu verteilen = $40/41$
- den 3ten Datensatz in einem leeren Behälter zu verteilen = $39/41$
-
- die ersten 8 Datensätze in unterschiedliche Behälter zu verteilen =
- $(41/41) * (40/41) * (39/41) * (38/41) * ... * (34/41) = 0.482 < 50\%$

Wahl einer Hashfunktion

- Methoden, die benutzt werden um eine Hashfunktion zu definieren:
 - Divisionsverfahren
 - Mittquadratmethode
 - Multiplikative Methode
 - usw.
- Typische Hashfunktionen berücksichtigen die Bit-Darstellung des Suchschlüssels um den Hashwert zu berechnen
- z.B. Für ein String Suchschlüssel, kann man die binäre Darstellungen aller Charakter addieren und die Summe wählt man als Parameter für die Hashfunktion

Wahl einer Hashfunktion

- Divisionsverfahren
 - $h(k) = k \bmod N$, wobei N die Anzahl der Behälter ist
 - Wählt man $N = 2^d$, so werden letztendlich die letzten d Bits von k als Hashwert betrachtet
 - Am günstigsten wählt man eine Primzahl für N (die nicht nahe einer Zweierpotenz liegt), um eine gute Streuung zu gewährleisten (beeinflusst alle Bits)
- Mittquadratmethode
 - Berechne den Quadrat des Suchschlüsselwertes und wähle ein paar Ziffern aus der Mitte des Quadrats

Wahl einer Hashfunktion

- Multiplikative Methode

1. Der Schlüsselwert k wird mit einer Zahl A multipliziert
2. Der ganzzahlige Anteil des Ergebnis aus Schritt 1 wird abgeschnitten \rightarrow das Ergebnis wird in das Intervall $[0,1]$ abgebildet
3. Das Ergebnis von Schritt 2 wird mit der Anzahl der Behälter m multipliziert und nach unten abgerundet

- Es gilt:

$$h(k) = \lfloor m * (k * A \bmod 1) \rfloor = \lfloor m * (k * A - \lfloor k * A \rfloor) \rfloor$$

- Eine gute Wahl: $A = (\sqrt{5} - 1)/2 = 0.61803\dots$ oder $A = (3 - \sqrt{5})/2 = 0.38196\dots$

Hashfunktion - Beispiel

- Suchschlüsselwert 'Toyota'
 - Wir nehmen die ersten zwei Charakter 'To' und berechnen die alphabetische Position \Rightarrow

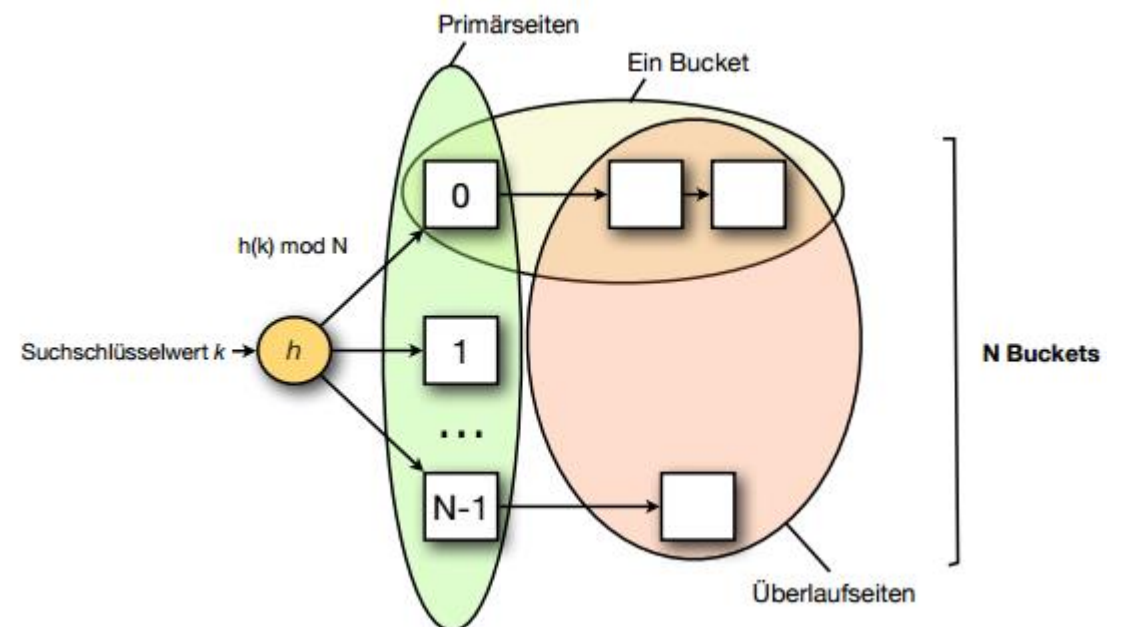
20	15
----	----
- Hashfunktionen:
 - Divisionsverfahren mit $N = 97 \rightarrow \mathbf{2015} \bmod 97 = 75$
 - Mittquadratmethode: $\mathbf{2015}^2 = 40\mathbf{60}225 \rightarrow$ nehme zwei mittleren Ziffern
 - Multiplikative Methode: $\lfloor 99 * (\mathbf{2015} * 0.61803 \bmod 1) \rfloor = 32$
- Warum benutzen wir nicht direkt 2015 als Hashwert?
 - 4 Ziffern \rightarrow 10000 mögliche Werte \rightarrow die Tabelle mit den Hashwerten würde ziemlich leer sein
 - In dem obigen Bsp. Brauchen wir 100 Hashwerte \rightarrow es kann ein Überlauf auftreten

Strategien zur Kollisionbehandlung / Überlaufbehandlung

- Mittels offener Adressierung – im Kollisionsfall nach fester Regel alternativen freien Platz in Hashtabelle suchen
- Mittels verketteter Listen – jeder Behälter enthält Zeiger auf Überlaufliste
- Mittels einer zweiten Hashfunktion (Double Hashing) – man wendet die zweite Hashfunktion auf das Ergebnis des ersten um eine neue Adresse zu bekommen
- Zeiger anstatt Datensätze speichern → in der Hash Adresse speichert man:
 - Alle Zeiger zu synonymen Datensätze – Behälter von Adressen
 - Zeiger zu dem ersten Datensatz (der dann ein Zeiger zu der nächsten enthält, usw.) – verkettete Listen von Adressen

Statisches Hashing

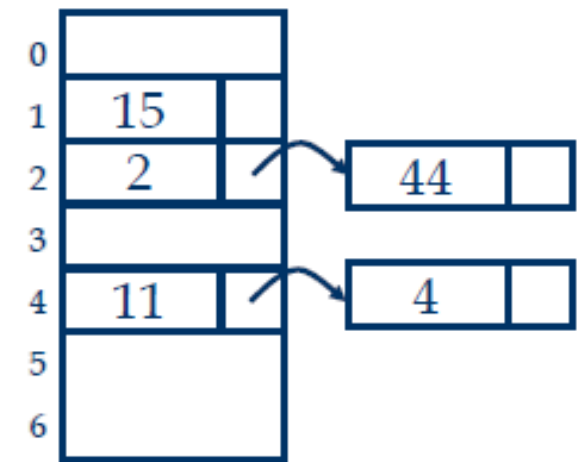
- Ein Behälter besteht aus einer Primärseite und ggf. ein oder mehreren Überlaufseiten
- Die Anzahl der Primärseiten ist von Anfang an fest und die Seiten sind sequentiell auf der Festplatte gespeichert (und nie freigegeben)
- Gegeben N Behälter, die von 0 bis $N-1$ numeriert sind, so wird k dem Behälter $h(k) \bmod N$ zugewiesen



Statisches Hashing mit unabhängigen Listen

- Alle synonyme Datensätze werden in einer verketteten Liste gespeichert
- Die Hashdatei enthält eine Liste von N Datensätze; jeder Datensatz ist Kopf einer Liste von Synonymen
- Die Reihenfolge der synonymen Datensätze in der Hashdatei kann folgende sein:
 - Die Reihenfolge der Einfügungen
 - Steigende Reihenfolge der Suchschlüsselwerte
 - Absteigende Reihenfolge der Suchfrequenz

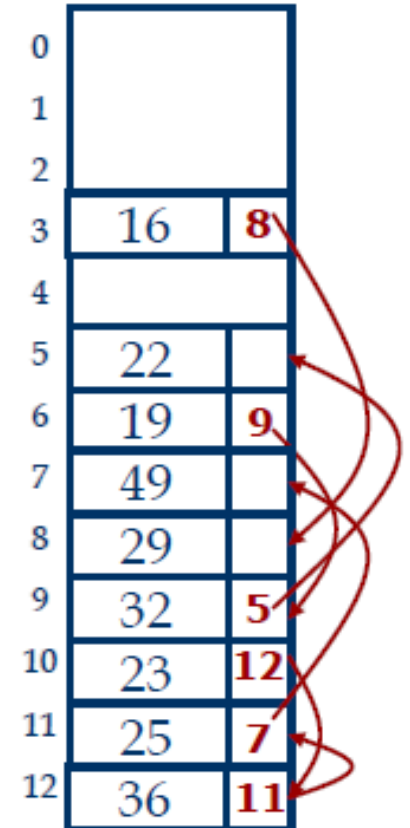
k	$h(k) = k \bmod 7$
11	4
2	2
44	2
4	4
15	1



Statisches Hashing mit verzahnten Listen

- Keine Überlaufseiten
- Einfügen eines Datensatzes mit Schlüsselwert k :
 - Falls der Slot an der Adresse $h(k)$ frei ist, dann speichere den Datensatz
 - Falls der Slot nicht frei ist, dann:
 - Suche von unten nach oben den ersten freien Slot und speichere den Datensatz
 - Füge den Slot am Ende der Liste die den Slot $h(k)$ enthält
- Beispiel

k	$h(k) = k \bmod 13$
16	3
23	10
36	10
25	12
19	6
32	6
29	3
49	10
22	9

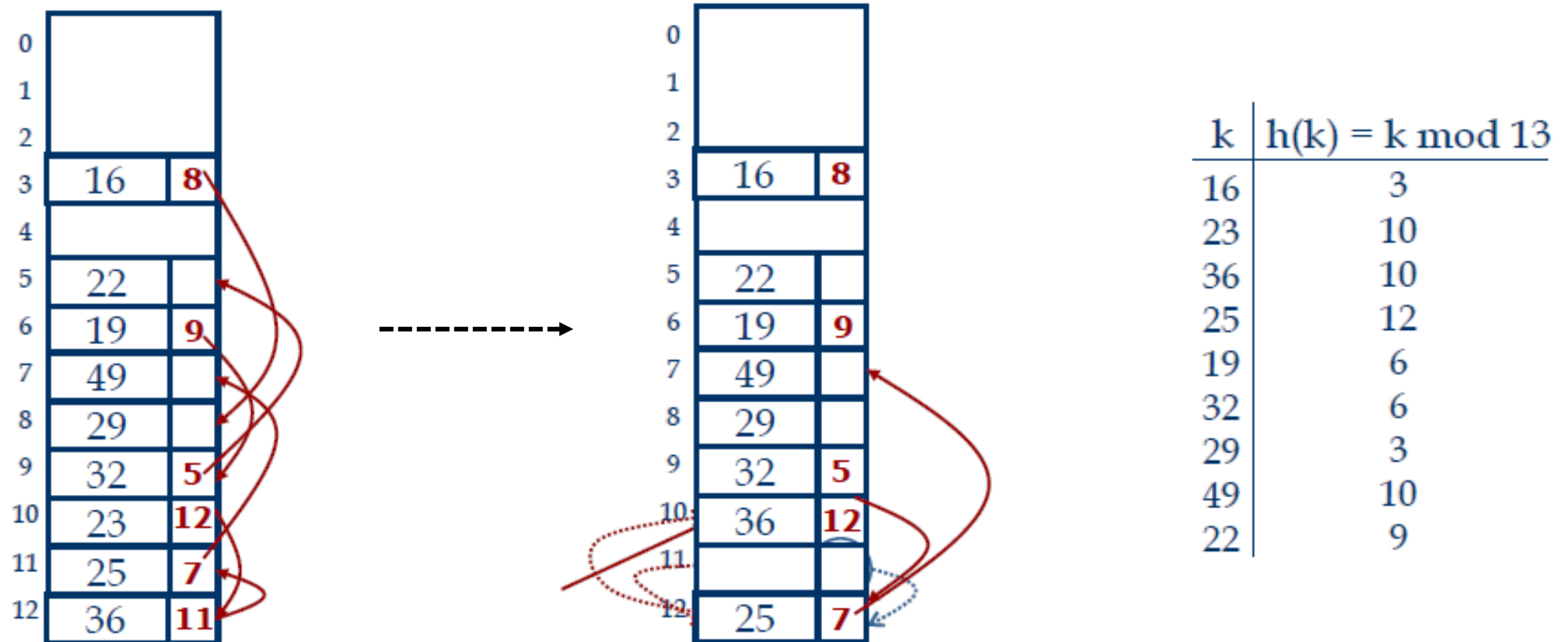


Statisches Hashing mit verzahnten Listen

- Löschen eines Datensatzes mit Schlüsselwert k :
 - Falls der Slot an der Adresse $h(k)$ frei ist → Fehlermeldung
 - Falls der Slot nicht frei ist, dann:
 1. Finde und lösche den Datensatz (mit Hilfe der Zeiger)
 2. Suche, mit Hilfe der Zeiger, ein Datensatz r mit $h(k_r) = h(k)$
 - Wenn es einen solchen Datensatz gibt, dann verschiebe es in den aktuellen Slot
 3. Wiederhole Schritt 2 für den neuen leeren Slot **oder**
Kopiere den Zeiger des leeren Slots in den davorstehender Slot in der Liste (wenn es einen gibt)

Statisches Hashing mit verzahnten Listen

- Beispiel: lösche den Datensatz mit Schlüsselwert 23



Statisches Hashing mit offener Adressierung

- Die Hashdatei enthält nur Dateneinträge (keine Zeiger zu weitere Seiten)
- Für kollidierende Schlüssel wird ein freier Eintrag in der Tabelle gesucht
- *Sondierungsreihenfolge* bestimmt für jeden Schlüssel, in welcher Reihenfolge alle Hashtabelleneinträge auf einen freien Platz durchsucht werden
- z.B. Lineares Sondieren : $h(k), h(k)+1, h(k)+2, \dots, N-1, 0, \dots, h(k)-1$

Statisches Hashing mit offener Adressierung

- Einfügen eines Datensatzes mit Schlüsselwert k :
 - Falls der Slot an der Adresse $h(k)$ frei ist, dann speichere den Datensatz
 - Falls der Slot nicht frei ist, dann suche einen freien Slot an die Adressen: $h(k)+1, h(k)+2, \dots, N-1, 0, \dots, h(k)-1$
- Gut für 75% Belegung
- Beispiel:

k	$h(k) = k \bmod 13$
5	5
21	8
24	11
22	9
23	10
34	8
35	9

0	35
1	
2	
3	
4	
5	5
6	
7	
8	21
9	22
10	23
11	24
12	34

Statisches Hashing mit offener Adressierung

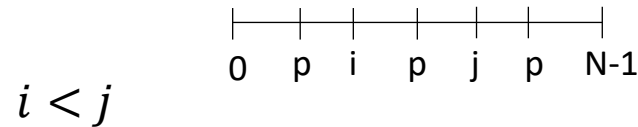
- Löschen eines Datensatzes mit Schlüsselwert k :
 - Problem: Löscht man z.B. h_0 aus der Folge h_0, h_1, h_2 , so kann h_2 nicht mehr gefunden werden
 - Lösungen:
 - A. Ersetze den zu löschenden Eintrag durch einen „Wächter“ (special code character).

Alle Operationen werden dann angepasst: die Suche schaut über den Wächter hinweg, so als ob dort ein gültiger Wert steht. Bei einer Einfüge-Operation kann der Wächter durch einen Neueintrag ersetzt werden.

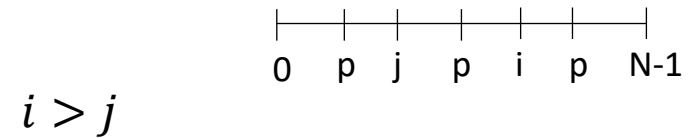
Statisches Hashing mit offener Adressierung

B. Lösche den Datensatz und verschiebe die anderen Datensätze.

- Seien i , j und p Adressen, so dass:
 - i ist die Adresse des Datensatzes den wir löschen wollen
 - Zwischen i und j gibt es keine freie Slots
 - $h(k_j) = h(k_p) \rightarrow$ der Datensatz an der Adresse j sollte an die Adresse p gespeichert werden
- Es gibt folgende Fälle:



$i < p \leq j \rightarrow$ nicht verschieben
 $j \leq p \leq N - 1 \rightarrow$ verschiebe Datensatz
von der Adresse j zu der Adresse i
 $0 \leq p \leq i \rightarrow$ verschiebe Datensatz
von der Adresse j zu der Adresse i



$0 < p \leq j \rightarrow$ nicht verschieben
 $j < p \leq i \rightarrow$ verschiebe Datensatz
von der Adresse j zu der Adresse i
 $i < p \leq N - 1 \rightarrow$ nicht verschieben

Statisches Hashing - Zusammenfassung

- Hashfunktion verteilt die Datensätze über N Behälter (Anzahl steht fest)
- **Statisches** Hashing ist für eine reale Datenbank nicht effizient → eine einmal angelegte Hash-Tabelle kann nicht effizient vergrößert werden
- Wenn viele Einfügeoperationen erwartet werden, gibt es zwei Möglichkeiten:
 - Von vornherein viel Platz für die Tabelle reserviert → viel freier Platz umsonst, da die Primärseiten nie freigegeben werden
 - Es entstehen im Laufe der Zeit immer längere Überlaufketten → können nur durch Änderung der Hashfunktion und aufwendige Reorganisation der Tabelle beseitigt werden
- Lösung des Problems: **dynamisches** Hashing, **erweiterbares** Hashing und **lineares** Hashing

Erweiterbares Hashing

- Problem: Die Behälter (Primärseiten) sind voll
- Lösung: Die Datei wird reorganisiert und die Anzahl von Behälter verdoppelt
 - Lesen und Schreiben aller Seite ist aber teuer
 - Idee:
 - benutze ein Verzeichnis von Behälter
 - Müsste ein neuer Datensatz in einen bereits vollen Behälter eingetragen werden, so wird er aufgeteilt → keine Änderungen nötig bei den anderen Behälter und keine Überlaufseite nötig
 - Das Verzeichnis von Behälter ist viel kleiner als die ganze Datei → die Verdoppelung ist viel billiger

Erweiterbares Hashing

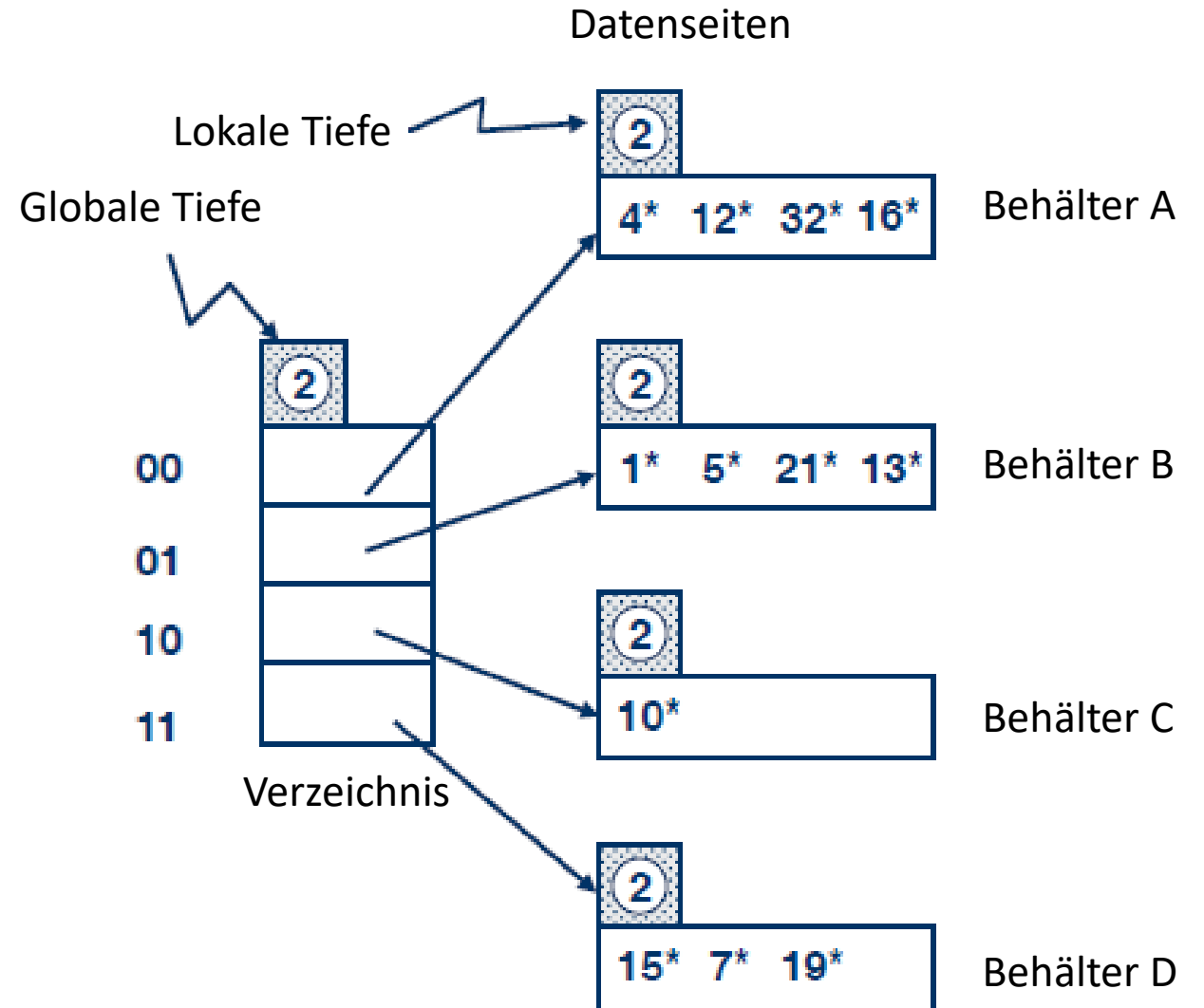
- Wichtig ist wie die Hashfunktion angepasst wird
- Der Wert $h(x)$ wird binär dargestellt und nur ein Präfix dieser binären Darstellung berücksichtigt
 $h(x) = dp$, wobei dp die Binärdarstellung ist, in zwei eingeteilt
- d gibt die Position des Behälters im Verzeichnis an (p wird zur Zeit nicht benutzt)
- Die Größe von d wird die **globale Tiefe t** genannt
- Die **lokale Tiefe t'** eines Behälters gibt an, wieviele Bits des Schlüssels für diesen Behälter tatsächlich verwendet werden

Erweiterbares Hashing

- Wenn ein Behälter voll ist und aufgeteilt werden muss, dann erfolgt die Aufteilung anhand eines weiteren Bits des bisher unbenutzten Teil p
- Ist die globale Tiefe nicht ausreichend, um den Verweis auf den neuen Behälter eintragen zu können, muss das Verzeichnis verdoppelt werden
- Eine Verdoppelung des Verzeichnisses erfolgt also, wenn nach einer Aufteilung eines Behälters die lokale Tiefe größer als die globale Tiefe ist

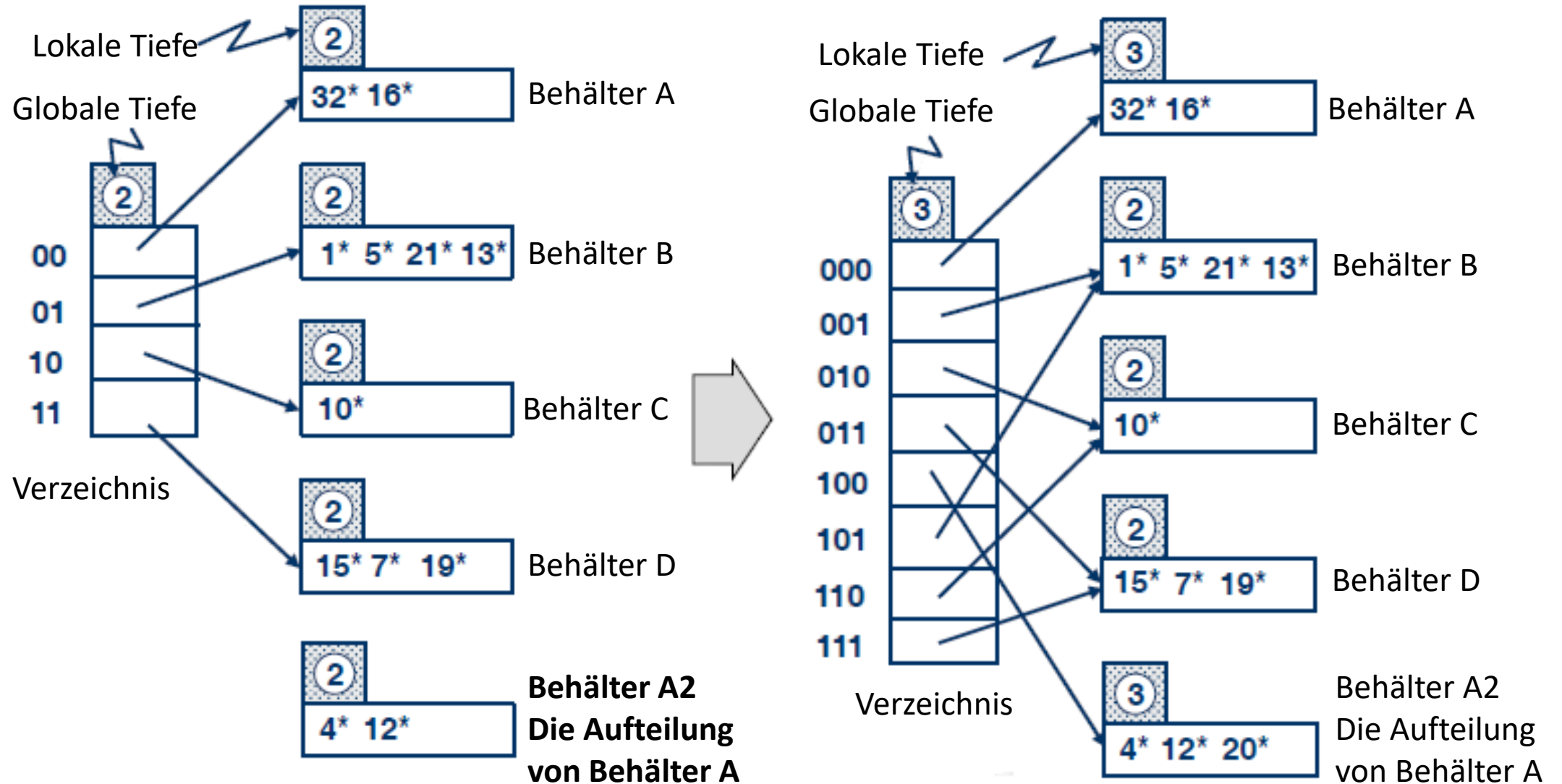
Erweiterbares Hashing - Beispiel

- Um den Behälter für x zu finden, berücksichtige die letzten t Bits aus $h(x)$
- $t = t' = 2$
- $h(5) = 5 = 101b \rightarrow$ in dem Behälter verweist von 01
- Füge k ein: $h(k) = 20 = 10100b \rightarrow$ Behälter 00



Erweiterbares Hashing - Beispiel

- Füge k ein: $h(k) = 20 = 10100b \rightarrow$ Behälter 00 \rightarrow Verzeichnis verdoppeln



Erweiterbares Hashing - Beispiel

- Beim Einfügen von $h(k) = 20 = 10100b$:
 - Die letzten **2** Bits 00 sagen uns, dass k im Behälter A oder A2 gehört
 - Die letzten **3** Bits sagen uns in welchen der zwei Behälter es gehört
- Globale Tiefe t – die Anzahl der Bits die gebraucht werden um den Behälter zu lokalisieren → vor dem Einfügen $t = 2$, nach dem Einfügen $t = 3$
- Lokale Tiefe t' eines Behälters – die Anzahl der Bits tatsächlich benutzt → in dem Beispiel $t' = 2$ oder $t' = 3$
- Da nach dem Einfügen $t' > t$ ist → Verdoppelung des Verzeichnisses

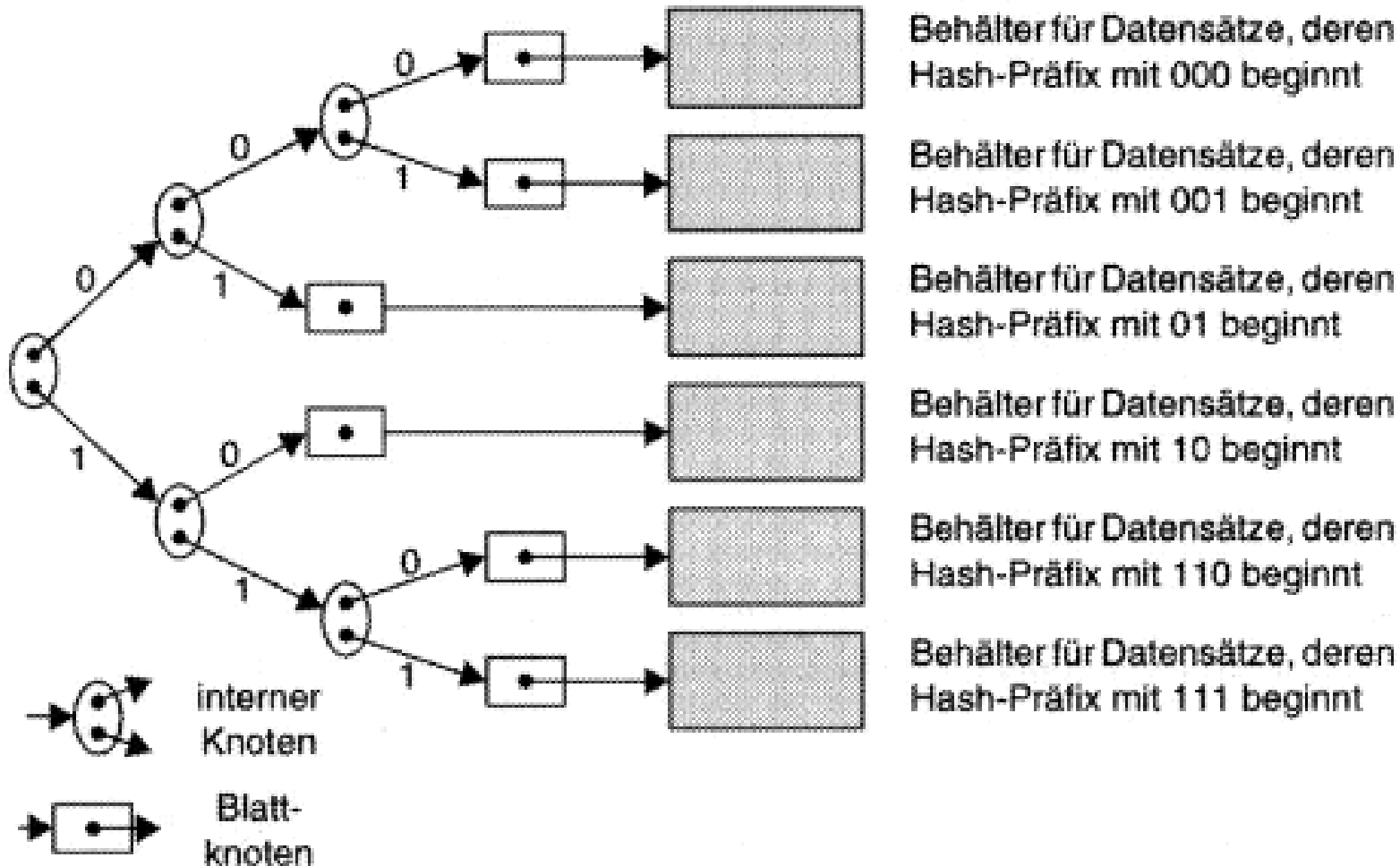
Erweiterbares Hashing

- Wenn das Verzeichnis im Hauptspeicher passt, dann kann man eine Gleichheitsanfrage mit einem Festplattenzugriff beantworten, da es keine Überlaufseiten gibt
- Ansonsten muss die jeweilige Verzeichnisseite vom Speicher geladen werden, und es sind dann insgesamt zwei Seitenzugriffe erforderlich
- Viele Datensätze mit demselben Hashwert können Probleme verursachen
- Werden Daten gelöscht, ist es möglich Behälter wieder zu verschmelzen oder sogar das Verzeichnis zu halbieren
- Im Vergleich zum statischen Hashing → speicherplatzsparender (passt sich dem Speicherplatzbedarf dynamisch an)

Dynamisches Hashing

- Die Idee ist die gleiche wie beim erweiterbares Hashing, aber es wird eine andere Art von Verzeichnisstruktur benutzt
 - Verzeichnisstruktur beim erweiterbares Hashing → ein Array mit 2^d Behälter, wobei d die globale Tiefe ist
 - Verzeichnisstruktur beim dynamisches Hashing → Verzeichnisbaum

Dynamisches Hashing



Lineares Hashing

- Idee: erlaubt einer Hash-Datei, ohne Verwendung einer Verzeichnisstruktur dynamisch zu wachsen und zu schrumpfen
- Dieses Schema benutzt eine Familie von Hashfunktionen h_0, h_1, \dots
- Der Wertebereich einer Funktion h_{i+1} ist doppelt so groß wie der Wertebereich der Vorgängerkfunktion h_i
→ falls h_i einen Indexeintrag auf einen von N Behälter abbildet, so bildet h_{i+1} den Eintrag auf einen von $2N$ Behälter ab
- Überlaufseiten werden benutzt
- Gewährt eine gewisse Flexibilität bei der Entscheidung, wann ein Behälter geteilt wird
- Der Übergang von einer Hashfunktion h_i zu h_{i+1} entspricht der Verdoppelung des Verzeichnisses beim erweiterbaren Hashing

Lineares Hashing - Beispiel

- Größe der Behälter: 4
- Level: 0
- Nächstes Behälter zu verdoppeln: 0
- Füge folgende Werte ein: 37 = 100101

h_0				
00	32 (100000)	44 (101100)	36 (100100)	
01	9 (1001)	25 (11001)	5 (0101)	
10	14 (1110)	18 (10010)	10 (1010)	30 (11110)
11	31 (11111)	35 (100011)	7 (0111)	11 (1011)

Lineares Hashing - Beispiel

- Größe der Behälter: 4
- Level: 0
- Nächstes Behälter zu verdoppeln: 0
- Füge folgende Werte ein: 37 = 100101, 43 = 101011

h_0				
00	32 (100000)	44 (101100)	36 (100100)	
01	9 (1001)	25 (11001)	5 (0101)	37 (100101)
10	14 (1110)	18 (10010)	10 (1010)	30 (11110)
11	31 (11111)	35 (100011)	7 (0111)	11 (1011)

43 (101011)

Lineares Hashing - Beispiel

- Größe der Behälter: 4
- Level: 0
- Nächstes Behälter zu verdoppeln: **1**
- Füge folgende Werte ein: 29 = 11101

h_1	h_0					
000	00	32 (100000)				
	01	9 (1001)	25 (11001)	5 (0101)	37 (100101)	
	10	14 (1110)	18 (10010)	10 (1010)	30 (11110)	
	11	31 (11111)	35 (100011)	7 (0111)	11 (1011)	43 (101011)
100	00	44 (101100)	36 (100100)			

Lineares Hashing - Beispiel

- Größe der Behälter: 4
- Level: 0
- Nächstes Behälter zu verdoppeln: **2**

h_1	h_0					
000	00	32 (100000)				
001	01	9 (1001)	25 (11001)			
	10	14 (1110)	18 (10010)	10 (1010)	30 (11110)	
	11	31 (11111)	35 (100011)	7 (0111)	11 (1011)	43 (101011)
100	00	44 (101100)	36 (100100)			
101	01	5 (0101)	37 (100101)	29 (11101)		

Hash-basierte Indexe

- Vorteile:
 - „Unschlagbar“, wenn es um Gleichheitsanfragen geht
`SELECT * FROM R WHERE A = k`
 - Schneller Zugriff auf Daten wenn man bestimmte Informationen schon kennt (Suchschlüsselwert)
 - Weitere Anfrageoperationen, die eine Menge von Gleichheitsprüfungen durchführen, profitieren von Hash-Indexe

Hash-basierte Indexe

- Nachteile:
 - Es kann nur ein Hashindex geben auf einem Suchschlüssel (man muss eine Hash-Methode auswählen)
 - Die sequentielle Reihenfolge der Datensätze im Speicherplatz hat keine Bedeutung
 - Es können Blöcke von leeren Slots in einer Datei geben → ungleichformige Ladezeit
 - Keine Unterstützung bei Bereichsanfragen
 - Keine Unterstützung bei Anfragen wo man den Wert eines anderen Feldes außer dem Suchschlüssel kennt
 - Nicht empfohlen, wenn sich die Suchschlüsselwerte oft ändern

SQL Indexe

- SQL hat kein Standard für die Indexerstellung definiert
- Es gibt unterschiedliche Varianten von SQL Befehle um einen Index zu erstellen
- Allgemeine Syntax:

```
CREATE [UNIQUE] INDEX indexName  
ON tableName (colName [ASC|DESC] [, ...])
```

```
DROP INDEX indexName;
```

- UNIQUE – alle Werte in dem Index sind eindeutig
- ASC/DESC – in welcher Reihengolge die Datensätze in dem Index sortiert sind

SQL Indexe - Beispiel

- Erstelle einen Index für die Tabelle `Student` auf das Attribut `ID`

```
CREATE UNIQUE INDEX idxStudent ON Student (ID)
```

- `UNIQUE` versichert, dass die Werte des Attributes `ID` in der Tabelle (und in dem Index) eindeutig sind
- Wenn das Attribut `ID` nicht eindeutig ist, dann wird bei dem Erstellen des Index eine Fehlermeldung ausgegeben

SQL Indexe - Beispiel

- Erstelle einen geclusterten Index auf das Attribut `Major` der Tabelle `Student`

```
Create INDEX clMajor ON Student(Major) CLUSTER
```

- Ein geclusterter Index kann auf dem Primärschlüssel oder auf einem anderen Attribut erstellt werden
- Geclustert mit Alternative (1) → die Datei wird selber nach dem Suchschlüssel sortiert

```
Create INDEX clMajor ON Student(Major, Year)
```

- Erstellt einen Index auf zwei Attributen

SQL Indexe - Beispiel

- Für Hash-Index Beispiele siehe <https://www.mssqltips.com/sqlservertip/3099/understanding-sql-server-memoryoptimized-tables-hash-indexes/>