# Physische Datenorganisation





## Zielvorstellung

Speichern von Dateien, die aus einzelnen Sätzen bestehen.





## INHALT

1. Sätze und Blöcke
2. Arten der Primärorganisation
3. Arten der Sekundärorganisation
4. Speicherstrukturen für Relationen
5. Join-Algorithmen





## 1. Sätze und Blöcke

(a) Felder und Blöcke

(b) Sätze mit konstanter bzw. variabler Länge

(c) Aufteilung der Sätze auf Blöcke

(d) Zugriff auf Sätze

## (a) Felder und Blöcke

Sätze bestehen aus *Feldern* und werden für die physische Speicherung auf *Blöcke* verteilt.

Blöcke bestehen aus:

(a) Header: speichert Kontrollinformationen

(b) Teilblöcke: enthalten jeweils einen Datensatz

| | | | | |
|---|---|---|---|---|
| Block | Header | Teilb. 1 | Teilb. 2 | Teilb. 3 |





## (b) Sätze mit konstanter und variabler Länge

Konstante Länge:

| | | | | |
|---|---|---|---|---|
| Block 1 | Satz 1 | Satz 2 | Satz 3 | Satz 4 |
| Block 2 | Satz 5 | Satz 6 | Satz 7 | Satz 8 |

Variable Länge:

| | | | | |
|---|---|---|---|---|
| Block 1 | Satz 1 | Satz 2 | Satz 3 | Satz 4 |
| Block 2 | Satz 5 | Satz 6 | Satz 7 | |





## Sätze mit konstanter Länge

Vorteil:

- einfache Suche

Nachteile:

- optionale Felder in jedem Satz
- Felder variabler Länge müssen mit maximaler Länge gespeichert werden





## Sätze mit variabler Länge

Vorteile:

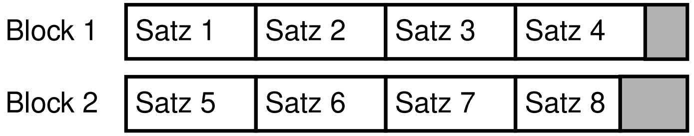
- Sätze unterschiedlicher Länge können gespeichert werden
- Optionale Felder können gespeichert werden

Nachteile:

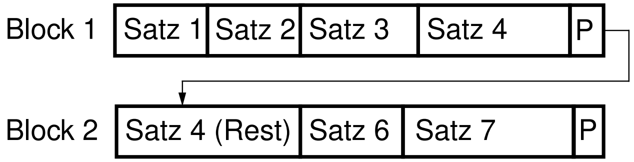
- Fragmentierung, Reorganisation notwendig (Defragmentierung)
- Höherer Suchaufwand

## (c) Aufteilung der Sätze auf Blöcke

Unspanned Records:




Spanned Records:








## Unspanned Records

Vorteile:

- Ein Satz ist in genau einem Block
- Einfache Verwaltung

Nachteile:

- Satzlänge ≤ Blocklänge
- Platzverlust





## (d) Zugriff auf Sätze

Grundlegende Zugriffsarten:

- Finden eines Satzes
- Einfügen eines Satzes
- Löschen eines Satzes

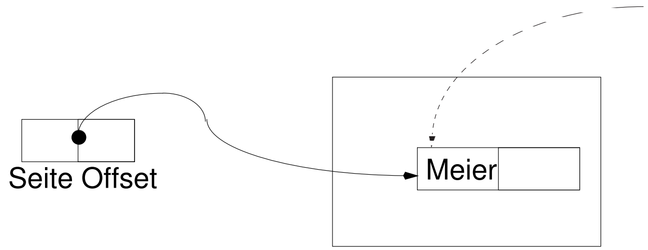
Der Zugriff erfolgt

- über den Schlüssel oder
- über die vollständige Adresse oder
- über die Blockadresse und den Schlüssel

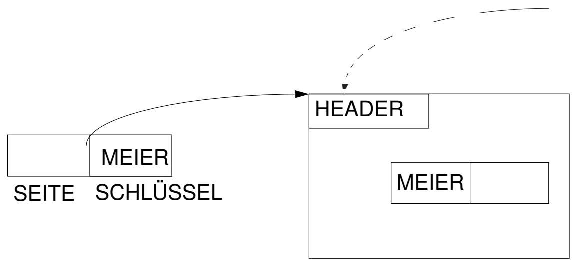




## Pinned Records



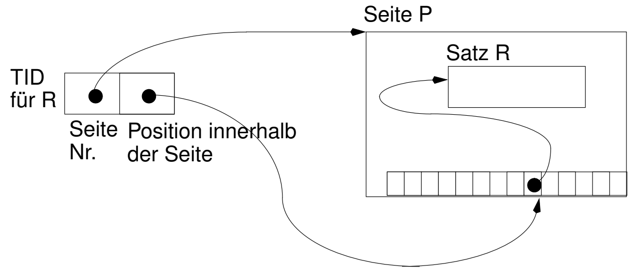
# Im Block verschiebbare Datensätze








# Tuple Identifiers (TID’s - System R)








# Vereinfachende Annahmen

- Ein Satz besteht aus einer festen Anzahl von Feldern
- Schlüssel analog Relationenmodell
- Blockzugriff über Blockadresse
- Das Speicherverwaltungsystem bietet Zugriff auf Blöcke bestimmter Größe
- Das Speicherverwaltungsystem verwaltet unbenutzte und gelöschte Blöcke





# Primär- und Sekundärorganisation

- Primärorganisation: Organisation der Hauptdatei (enthält die Daten) und der Hauptzugriffsstruktur
- Sekundärorganisation: Eine alternative Zugriffsstruktur neben der primären Zugriffsstruktur

# 2. Arten der Primärorganisation

(a) Ungeordnete Dateien

(b) Indexsequentiell

(c) Clustered

(d) Hashorganisation

(e) B*-Baum





# (a) Ungeordnete Datei

| | | | | | | |
|---|---|---|---|---|---|---|
| **6** | **3** | **5** | **4** | **18** | | |

- Einfügen
- Suchen
- Löschen





# (b) Indexsequentiell

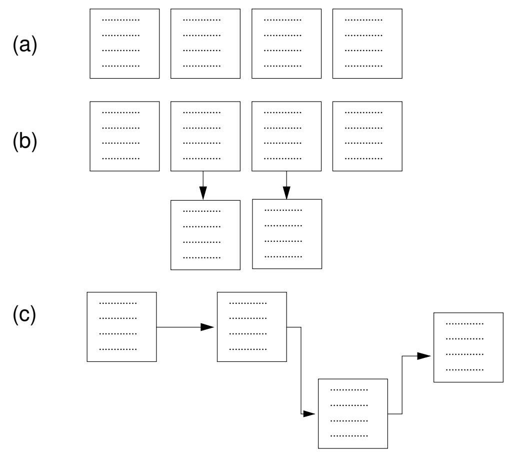
2 Dateien:

- Hauptdatei: enthält die Datensätze
- Indexdatei: ermöglicht einen effizienten Zugriff nach dem Schlüssel





# Speicherungsalternativen - Hauptdatei



## Index

Def.: Ein Index *I* zu einer Datei *D* ist eine Datei, deren Sätze Paare der Form $(v_i,b_i)$ darstellen, wobei $v_i$ der Schlüsselwert eines Satzes in *D* ist und $b_i$ die Adresse bzw. die Blockadresse dieses Satzes. *D* wird als Hauptdatei *(main file)* bezeichnet.

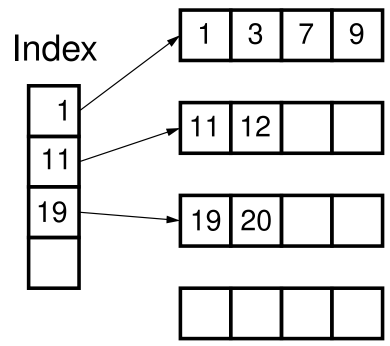
Motivation:

- Indexeintrag kleiner → mehr Einträge pro Block → weniger Lesezugriffe → schnellere Suche!
- Einsatz als Primärindex (bei sortierten Dateien) oder Sekundärindex (bei unsortierten Dateien)



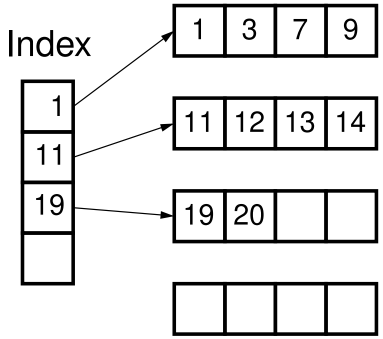


## Einfügen - Indexsequentiell




Einfügen: 14, 13

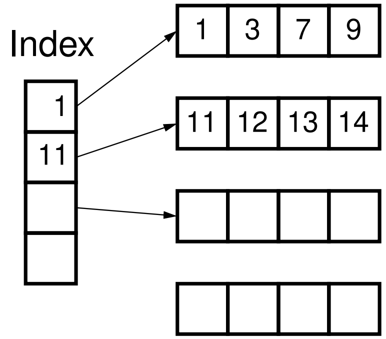
# Suchen - Indexsequentiell








# Löschen - Indexsequentiell




Löschen: 19, 20



19 19 20

20 20

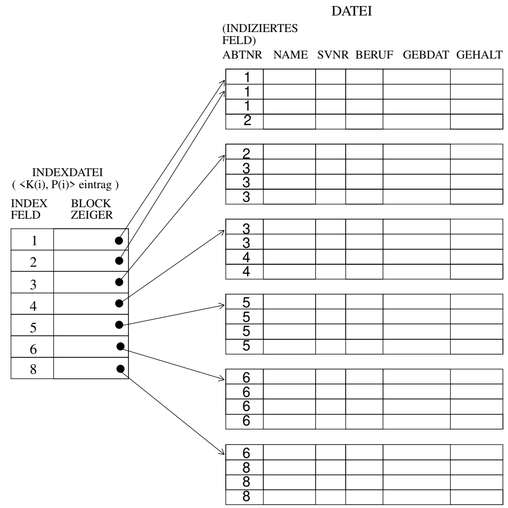
# (c) Clustered

- Abart der indexsequentiellen Speicherung
- Hauptdatei ist nicht nach dem Primärschlüssel sortiert (Sortierung nach “Schlüsselfeld”) $\rightarrow$ die indizierten Felder der Hauptdatei dürfen in unterschiedlichen Sätzen den gleichen Wert annehmen
- Einfügen, Suchen, Löschen wie bei der indexsequentiellen Form





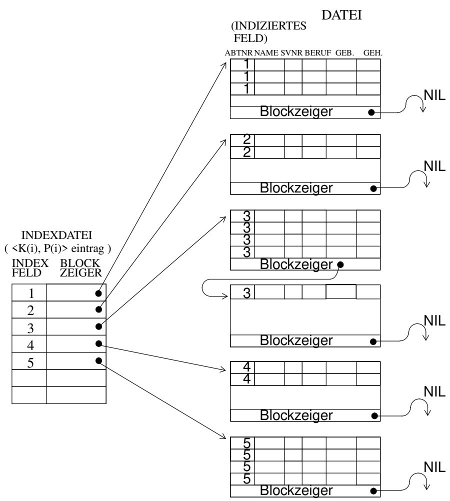
# Clustered








# Clustered








# Clustern mehrerer Dateien

Clustern mehrerer Dateien: Tupel aus verschiedenen Relationen mit gleichen Werten in bestimmten Feldern (clustering attributes) werden in benachbarten Blöcken gespeichert.

Vorteile:

- prejoin $\rightarrow$ Abfragen schneller
- geringerer Speicherbedarf - Feld wird nur einmal gespeichert

Nachteil:

Relation wird auf mehrere Blöcke verteilt $\rightarrow$ bei Abfragen innerhalb einer Relation sind mehr Zugriffe notwendig

# (d) Hashorganisation

- Allgemeines
- Hashfunktion
- Einfügen und Suchen
- Externes Hashen:
  Dynamisches, erweiterbares und lineares Hashen





# Allgemeines

Auf den Schlüssel wird eine Funktion angewendet, das Ergebnis ist die Position des Satzes innerhalb der Datei.

- Die Sätze einer Datei werden auf *Buckets* aufgeteilt
- Ein Bucket besteht aus einem oder mehreren Blöcken
- Die Hashfunktion soll “wirr” und “gleichverteilt” sein
- Die *Hashtabelle* zeigt jeweils auf den ersten Block eines Buckets





# Hashfunktion

H(s) = s mod nmax

s: Schlüssel

nmax: höchster erwünschter Hashwert, wenn möglich eine Primzahl

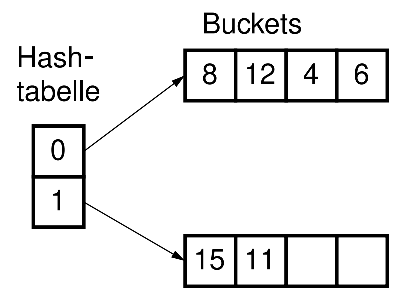
Hashfunktion für Beispiel:

H(s) = s mod 2

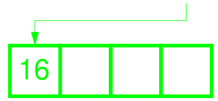




# Einfügen - Hashorganisation



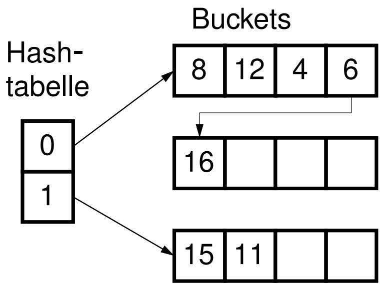

Einfügen: 13, 16




13



## Suchen - Hashorganisation








## Hashorganisation

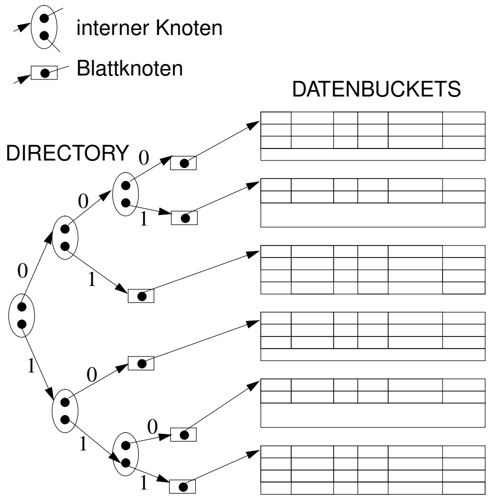
Nachteil: Dateigröße kann nur durch aufwendige Reorganisation geändert werden (Verwendung einer anderen Hashfunktion)

Abhilfe: externes Hashen

1. Dynamisches Hashen (dynamic hashing)
2. Erweiterbares Hashen (extendible hashing)
3. Lineares Hashen (linear hashing)

Voraussetzung: Schlüssel wird als Binärzahl betrachtet

# Dynamisches Hashen








# Dynamisches Hashen

Vorteil:

- Die Dateigröße paßt sich schnell und exakt den tatsächlichen Anforderungen an

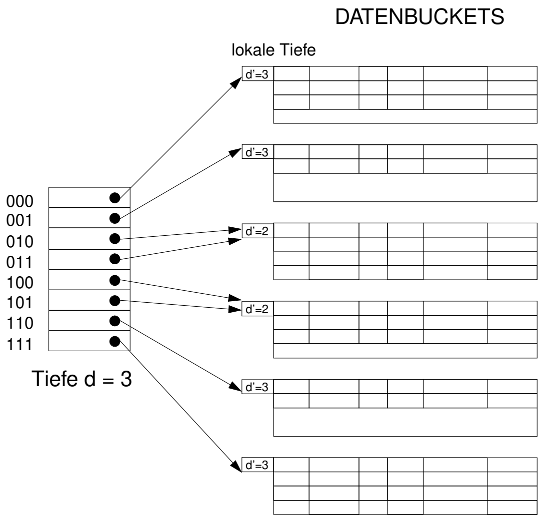
Nachteile:

- Zusätzlicher Aufwand für die Speicherung der Baumstruktur
- Bei Verwendung einer nicht gleichverteilten Hash-Funktion entsteht ein schlecht balanzierter Baum





# Erweiterbares Hashen








# Erweiterbares Hashen

Vorteil:

- Der Verwaltungsaufwand ist geringer als beim dynamischen Hashen

Nachteil:

- Das Hash-Directory paßt sich nur in vergleichsweise großen Sprüngen von überlinearer Größe an

# Lineares Hashen

Grundgedanke: es wird eine “skalierbare” Hashfunktion verwendet:

$$h_j(K) = K \bmod (2^j M)$$

Vorteil:

- Man benötigt keine externe Struktur

Nachteile:

- Hoher Verwaltungsaufwand durch komplexen Algorithmus
- Ungünstig bei schlecht verteilter Hashfunktion (viele Überläufe)





# (e) B*-Baum

Def.: Ein B*-Baum ist ein geordneter Vielweg-Baum dessen Knoten bzw. Blätter folgende Eigenschaften besitzen:

STRUKTUR:

- Jedes Blatt enthält zwischen k* und 2k* - 1 sortierte Datensätze
- Alle Blätter liegen auf gleicher Höhe
- Jeder innere Knoten enthält zwischen k und 2k - 1 Indexeinträge (Schlüssel und Zeiger auf Nachfolgeknoten). Der erste Indexeintrag jedes Knoten besteht nur aus einem Zeiger.





# B*-Baum

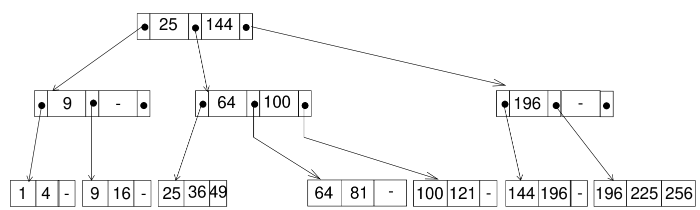
ORDNUNGSRELATION:

- Der links von einem Schlüssel s gelegene Zeiger zeigt auf einen Block, von welchem man nur zu Datensätzen mit kleinerem Schlüssel als s gelangen kann
- Der rechts von einem Schlüssel s gelegene Zeiger zeigt auf einen Block, von welchem man nur zu Datensätzen mit größerem oder gleichem Schlüssel als s gelangen kann (der Schlüssel s entspricht dem kleinsten Datensatzschlüssel des rechten Teilbaumes)

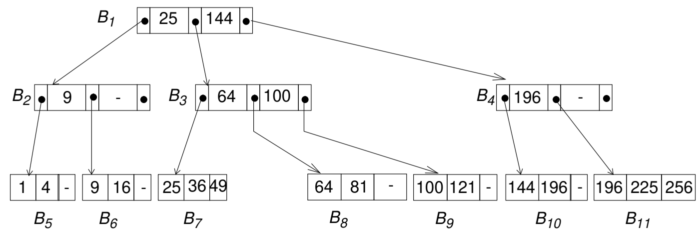




# B*Baum

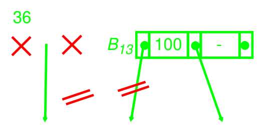


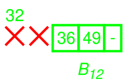
# Einfügen - B*Baum

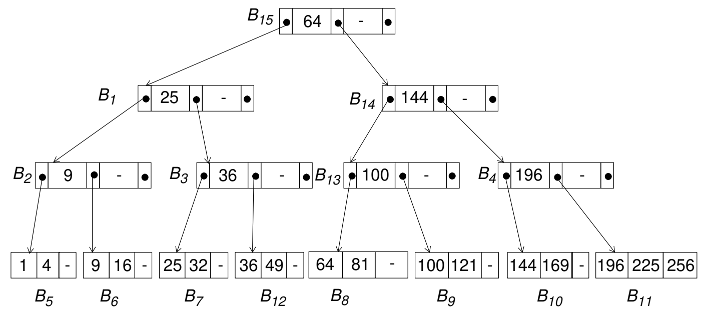

Einfügen: 32

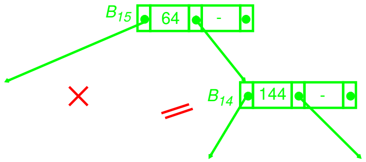




# Löschen - B*Baum

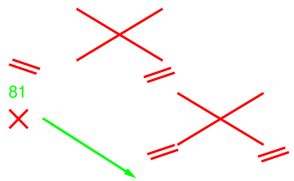





81 100 121

144 196






# 3. Arten der Sekundärorganisation

Def.: Eine alternative Zugriffsstruktur neben der primären Zugriffsstruktur (Primärorganisation)

(a) Index

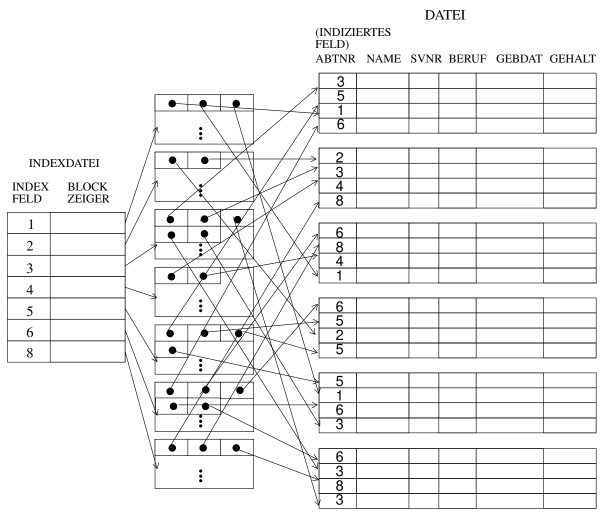
(b) Hash

(c) B*-Baum





# (a) Index








# (b) Hash

Wie Primärorganisation, nur enthalten die Buckets Verweise auf Datensätze

Anwendungsgebiet:

- Es gibt eine gleichverteilte und wirre Hashfunktion
- Es wird immer das vollständige Ordnungskriterium abgefragt
- Es werden keine Bereichsabfragen benötigt

## (c) B*-Baum

Wie Primärorganisation, nur enthalten die Blätter Verweise auf Datensätze

Anwendungsgebiet:

- bei vielen unterschiedlichen Werten
- bei Abfragen nach Teilen des Index-Feldes





## Kriterien für die Anwendung

Indiziere Attribute

- die oft in WHERE-Klausen vorkommen
- die häufig auf ihr Maximum / Minimum abgefragt werden
- die in Join-Operationen verwendet werden
- mit hoher Selektivität (viele verschiedene Attributwerte)

Indiziere nicht Attribute

- mit geringer Selektivität
- in kleinen Relationen





## 4. Speicherstrukturen für Relationen

(a) Kriterien zur Auswahl von Speicherstrukturen

(b) Füllgrad

(c) Zugriffszeitverhalten

(d) Physische Datenorganisation in kommerziellen Systemen





## (a) Kriterien: Auswahl v. Speicherstrukturen

- Ungeordnete Dateien
- Indexsequentiell
- Clustered
- Hashorganisation
- B*-Baum

## Ungeordnete Dateien

Vorteile:

- Schnellstes Einfügen
- Schnelles Löschen (wenn Satz gefunden)
- Geringer Platzbedarf (ACHTUNG: Fragmentierung)

Nachteile:

- Langsam bei Suche nach bestimmten Sätzen
- Langsam bei Bereichsabfragen
- Der Platz gelöschter Sätze kann nur durch Reorganisation wiederverwendet werden





## Ungeordnete Dateien - Anwendungsgebiet

- Kurze Dateien: zusätzlicher Verwaltungsaufwand lohnt sich nicht
- Wenn oft alle Sätze gelesen werden, z.B. bei statistischen Abfragen
- Wenn der Speichersplatz vollständig genutzt werden soll
- Bei Log-Einträgen





## Indexsequentiell

Vorteile:

- Gutes Antwortzeitverhalten bei pattern-matching, Bereichsabfragen, exakten Abfragen, Abfragen nach Teil-Schlüsseln

Nachteile:

- Hauptdatei und Index sind statisch → neue Sätze werden in Überlaufblöcken gespeichert → Suchoperationen werden langsamer. Abhilfe: zeitaufwendige Reorganisationen





## Indexsequentiell - Anwendungsgebiet

- Bei statischen Dateien (wenige Einfügeoperationen)
- Wenn flexible Abfragen notwendig sind

## Clustered - Vorteile

- Gutes Antwortzeitverhalten bei pattern-matching, Bereichsabfragen, exakten Abfragen, Abfragen nach Teil-Schlüsseln
- Sortierung muß nicht nach Primärschlüssel erfolgen
- Mehrere Dateien können verwoben werden → schneller-er Zugriff





## Clustered - Nachteile

- Langsamerer Zugriff auf Einzeldatei
- Aufwendiges Einfügen
- Aufwendiges Löschen





## Clustered - Anwendungsgebiet

- Bei statischen Dateien (wenige Einfügeoperationen)
- Wenn flexible Abfragen notwendig sind
- Wenn Dateien oft miteinander verknüpft werden





## Hashorganisation

Vorteile:

- Effizientester Zugriff auf Einzelsätze
- Schnellste Zugriffsmethode bei exakten Abfragen

Nachteile:

- Bei Bereichsabfragen muß jeder Satz gelesen werden
- Keine Sortierung möglich
- Ungünstig bei pattern-matching und Abfrage nach Teilsschlüsseln

## Hashorganisation - Anwendungsgebiet

- Wenn überwiegend exakte Abfragen nach Einzelsätzen gestellt werden
- Wenn nur wenig sortierte Abfragen benötigt werden





## B*-Baum - Vorteile

- Dynamische Anpassung $\rightarrow$ keine Überlaufprobleme
- freigewordener Platz kann ohne Reorganisation wiederverwendet werden
- Gutes Antwortzeitverhalten bei pattern-matching, Bereichsabfragen, exakte Abfragen, Abfrage nach Teil-Schlüssel
- Bereichsabfragen können effizient durchgeführt werden





## B*-Baum - Nachteile

- Hoher Verwaltungsaufwand
- Komplexere Zugriffsstruktur $\rightarrow$ Zugriff etwas langsamer als bei indexsequentiell
- Etwas höherer Platzverbrauch
- Baum muß bei gleichzeitigem Zugriff gesperrt werden





## B*-Baum - Anwendungsgebiet

- Gutes Antwortzeitverhalten bei pattern-matching, Bereichsabfragen, exakten Abfragen, Abfragen nach Teil-Schlüsseln
- Bei dynamischen Daten (viele Einfüge- und Löschoperationen)
- Wenn flexible Abfragen notwendig sind

## (b) Füllgrad

Def.: Als Füllgrad (in %) bezeichnet man

(Anzahl der erwünschten Sätze pro Seite /

Anzahl der möglichen Sätze pro Seite) * 100

Vorteil teilweiser Füllung:

- gar keine oder weniger Überläufe

Nachteil:

- Datei benötigt (100 / Füllgrad) mehr Platz





## (c) Mittleres Zugriffszeitverhalten

Blockzugriffe:

- Hashorganisation: Hashtabelle + (Bucketlänge / 2)
- Indexsequentiell (lockerer Index): $\log_2(n/ed)$
- B*-Baum: $< 1 + \log_d(n/e)$
- Dichter Index: $\log_2(n/d)$

n: Anzahl der Datensätze in der Hauptdatei

e: Anzahl der Datensätze pro Block in der Hauptdatei

d: Anzahl der Einträge pro Block in der Indexdatei





## (d) Physische DO in kommerz. Systemen

- Ingres
  - Heap: ungeordnete Datei - alle Blöcke werden gefüllt
  - Hash: (interne) Hashorganisation
  - Isam: geordnete Datei mit Primärindex
  - Btree: ungeordnete Datei, B*-Baum als Zugriffsstruktur
- Oracle
  - ungeordnete Datei,
  - B*-Baum als Zugriffsstruktur
  - Clusters
  - Hashing in Verbindung mit Definition von Clusters





## Auswahl von Speicherstrukturen

**Ziel:**

Effizienzsteigerung wichtiger Abfragen

**Einschränkungen:**

Verfügbarkeit mancher physischer Speicherstrukturen abhängig von Implementierung

**Annahme bei folgender Methode:**

Index und Clustering möglich

# NICHT VERWENDETE FOLIEN





# (e) Vorgehensweise - Überblick

1. Festlegung der Relationen und Abschätzung der Anzahl der Tupel in jeder Relation
2. Bestimmung häufiger Abfragen
3. Abschätzung der Häufigkeit jeder Abfrage
4. Erstellung eines JOIN-Graphen
5. CLUSTERs bestimmen
6. INDIZES bestimmen





# Bildung von CLUSTERs

- Bildung von Clusters bei häufiger Verwendung von Tabellen in JOINs
- keine Cluster bei häufiger Änderung von Werten, speziell der Cluster-Key-Attribute
- keine Cluster, wenn oft auf nur eine der beteiligten Tabellen zugegriffen werden muß
- Bildung von Clusters über Master-Detail-Tabellen, wo einer Zeile der Master-Tabelle mehrere Zeilen der Detail-Tabelle entsprechen
- Attribute eignen sich nicht als Cluster-Schlüssel, wenn zuviele Tupel die gleichen Attributwerte haben
- jede Tabelle darf nur in einem Cluster vorkommen





# Entscheidung über Auswahl von Clusters (1)

**Ausgangssituation:**

Verwendung einer bestimmten Tabelle in Abfragen ohne Join und Abfragen mit Join oder in mehreren Abfragen mit unterschiedlichen Joins

**Problem:**

Wird Cluster über mehrere Tabellen angelegt, wird die Ausführung von Abfragen über nur eine der Tabellen langsamer

## Entscheidung über Auswahl von Clusters (2)

**Häufigkeit der Abfragen:** Cluster über alle Tabellen einer Abfrage nur, wenn Abfrage mit Join wesentlich häufiger vorkommt als Abfrage ohne bzw. mit anderen Joins

**Gemeinsame Tabellen:** Bei gleich häufigen Abfragen sollen die gemeinsam genutzten Tabellen in einem Cluster gehalten werden

**Zahl der Zeilen:** Verwenden mehrere gleich häufige Abfragen in Joins dieselbe Tabelle, sollten die Tabellen derjenigen Abfrage in Clusters gehalten werden, die die größeren Verbundtabellen verursacht





## Bildung von Indizes

Indizes über

- Join-Attribute von Cluster-Kandidaten, die bei der Bildung der Clusters nicht berücksichtigt wurden





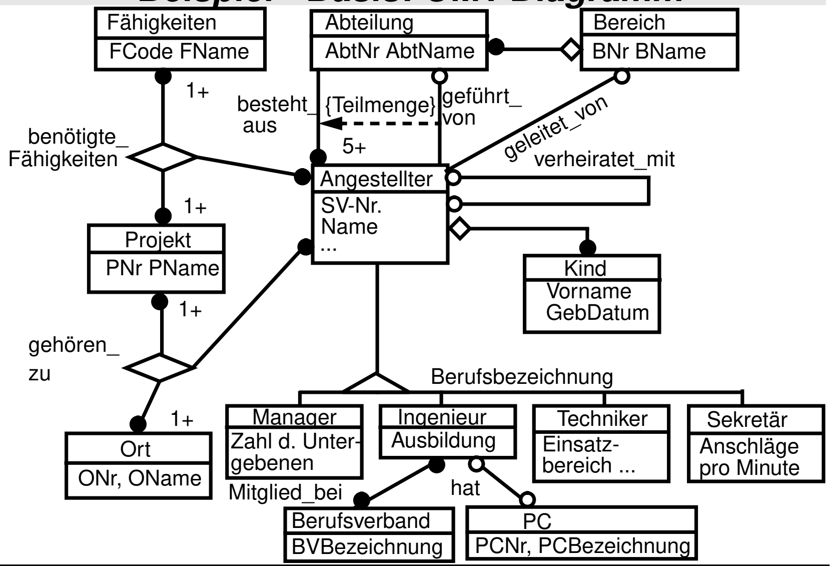
## Der JOIN-Graph

1. Aus OMT-Diagramm werden jene Klassen und Assoziationen dargestellt, die an häufigen Abfragen beteiligt sind.
2. Jede Klasse und jede Assoziation, für die ein Relationenschema definiert wurde, wird mit der geschätzten Zahl der Tupel versehen.
3. Jede Kante wird mit der Häufigkeit der Abfrage und den verbindenden Attributen (d. h. mit denjenigen Attributen, die in der Abfrage verglichen werden) versehen





## Beispiel - Basis: OMT-Diagramm



# 1 a Relationenschemata

| | |
|---|---|
| **Angestellter** | (SV-Nr, Name, ..., Berufsbezeichnung, AbtNr, verhSVNr) |
| **Manager** | (SV-Nr, Zahl d. Untergebenen, ...) |
| **Ingenieur** | (SV-Nr, Ausbildung, ..., PC-Nr.) |
| **Techniker** | (SV-Nr, Einsatzbereich, ...) |
| **Sekretär** | (SV-Nr, Anschläge p. M., ...) |
| **Berufsverband** | (BVBezeichnung, ...) |
| **PC** | (PCNr, PCBezeichnung) |
| **Kind** | (SV-Nr, Vorname, GebDatum) |
| **Bereich** | (BNr, BName, ..., Leiter-SV-Nr) |
| **Abteilung** | (AbtNr, AbtName, ..., BNr, Leiter-SV-Nr) |
| **Fähigkeiten** | (FCode, FName, ...) |
| **Projekt** | (PNr, PName, ...) |
| **Ort** | (ONr, OName, ...) |
| **Mitglied_bei** | (SV-Nr, BVBezeichnung) |
| **Benötigte_ Fähigkeiten** | (SV-Nr, PNr, FCode) |
| **Gehören_zu** | (SV-Nr, PNr, ONr) |





# 1 b Größe der Relationen

| Relation | Zahl d. Tupel/Relation |
|---|---|
| ANGESTELLTER | 5000 |
| KIND | 10000 |
| BEREICH | 20 |
| ABTEILUNG | 100 |
| FÄHIGKEITEN | 150 |
| PROJEKT | 500 |
| ORT | 50 |
| MANAGER | 150 |
| INGENIEUR | 3500 |
| TECHNIKER | 1000 |
| SEKRETÄR | 350 |
| BERUFSVERBAND | 20 |
| PC | 3500 |
| MITGLIED_BEI | 7000 |
| BENÖTIGTE_FÄHIGKEITEN | 15000 |
| GEHÖREN_ZU | 20000 |





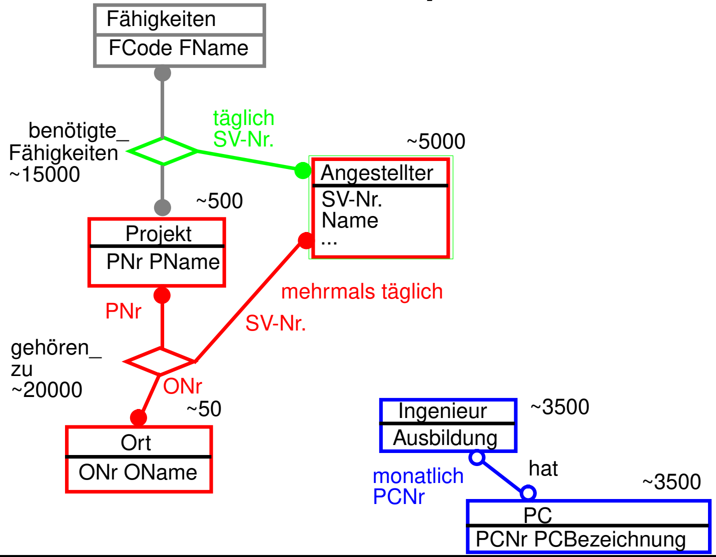
# 2, 3 Häufige Abfragen u. Häufigkeit

```
(1) monatlich:
SELECT   PCNr, PCBezeichnung
FROM     pc
WHERE    PCNr NOT IN (SELECT PCNr FROM ingenieur)
ORDER BY 1;

(2) täglich:
SELECT   a.SV-Nr, a.Name, f.FCode
FROM     Angestellter a, benötigte_Fähigkeiten f
WHERE    a.SV-Nr = f.SV-Nr
ORDER BY 1;

(3) mehrmals täglich:
SELECT   a.SV-Nr, a.Name, g.PNr, p.PName, g.ONr, o.OName
FROM     Angestellter a, gehören_zu g,
         Projekt p, Ort o
WHERE    a.SV-Nr = g.SV-Nr AND p.PNr = g.PNr
         AND o.ONr = g.ONr
ORDER BY 1;
```





# 4 JOIN-Graph



# 5 Bestimmung der Clusters (1)

```
CREATE CLUSTER ingenieur_pc_cluster
   (PCNr           INTEGER);

CREATE TABLE ingenieur
   (SVNr           CHAR(10) PRIMARY KEY,
    ausbildung     CHAR(20),
    PCNr           INTEGER REFERENCES pc (PCNr))
   CLUSTER ingenieur_pc_cluster (PCNr);

CREATE TABLE pc
   (PCNr           INTEGER PRIMARY KEY,
    PCBezeichnung  CHAR (30))
   CLUSTER ingenieur_pc_cluster (PCNr);

CREATE INDEX ingenieur_pc_index
   ON CLUSTER ingenieur_pc_cluster;
```





# 5 Bestimmung der Clusters (2)

```
CREATE CLUSTER gehören_zu_cluster
   (SV–Nr          CHAR (10),
    PNr            INTEGER,
    ONr            INTEGER);
```

**Hinweis: Reihenfolge der Attribute in der Cluster–Definition durch SELEKTIVITÄT bestimmt**

**zu Abfrage 3:**
**Angestellter bereits in Cluster "gehören_zu_cluster", daher Definition eines Index für Tabelle benötigte_Fähigkeiten**

```
CREATE INDEX Fähigkeiten_index ON
benötigte_Fähigkeiten (SV–Nr);
```





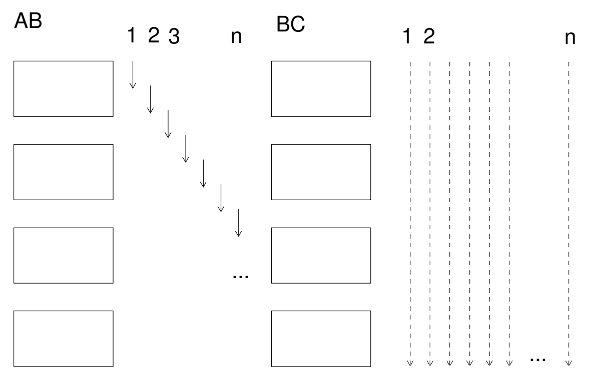
# Join-Algorithmen

1. Nested-loop Join
2. Join mit Hilfe eines Index
3. Sort-Merge Join
4. Hash Join

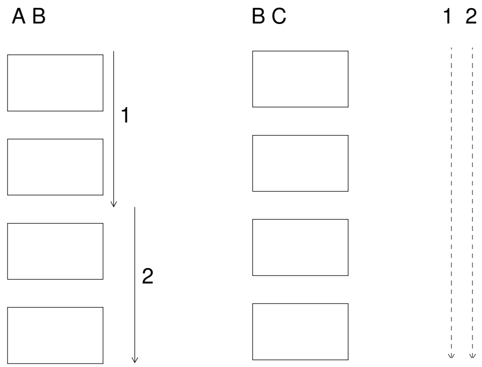




# Nested-loop Join: primitives Verfahren



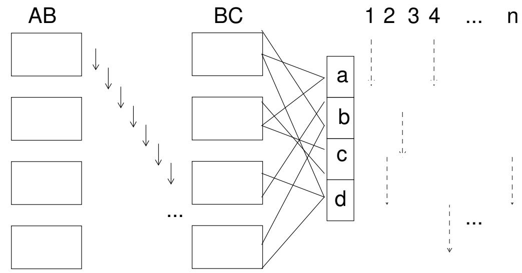
# Nested-loop Join: verbesserte Lösung







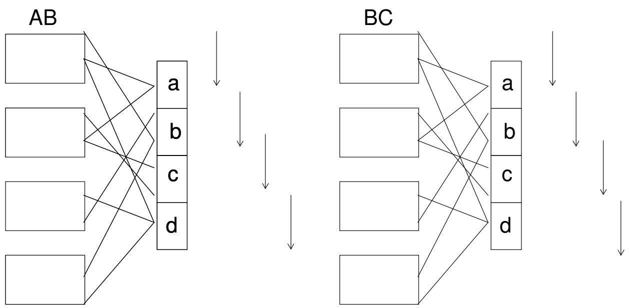

# Join mit Hilfe eines Index über BC







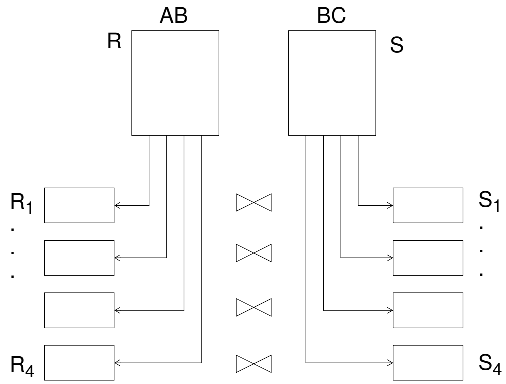

# Sort-Merge Join mit Index über beide Dateien

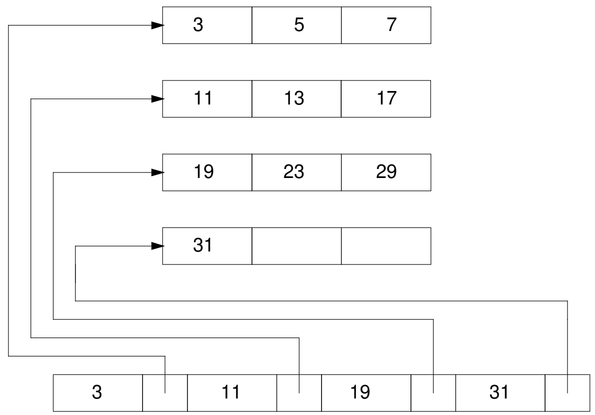




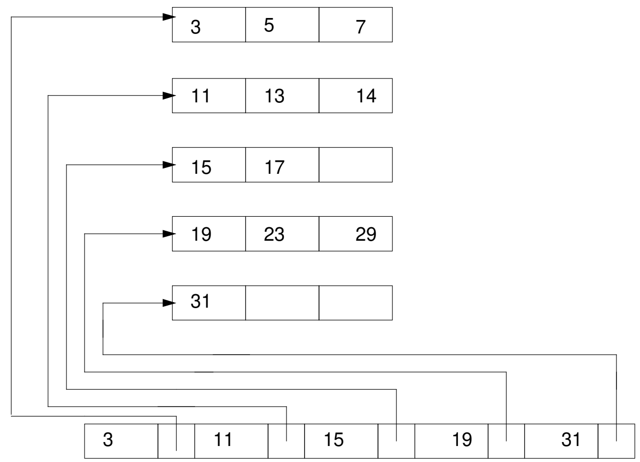



# Hash-Join

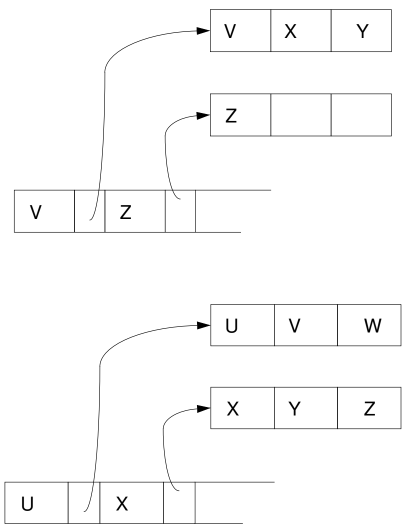



3 5 7
11 13 17
19 23 29
31
3 11 19 31

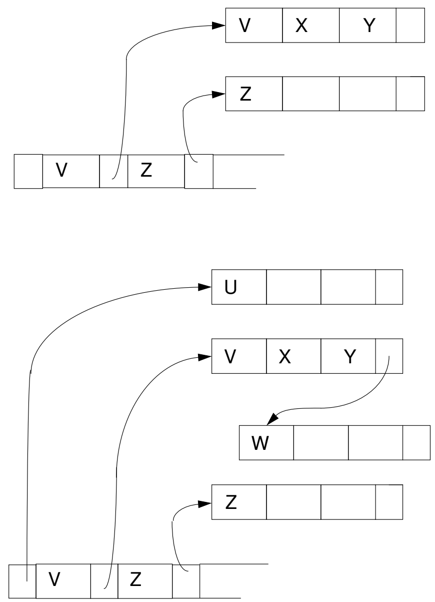


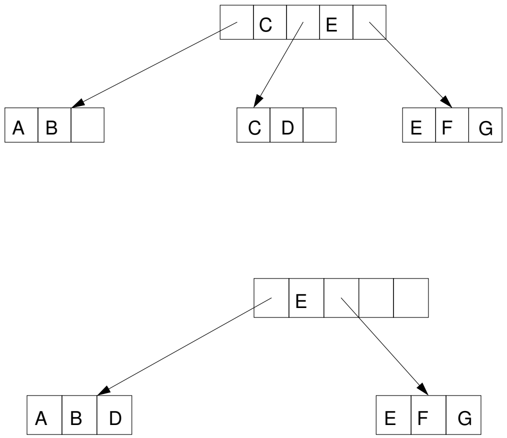

3 5 7
11 13 14
15 17
19 23 29
31
3 11 15 19 31




V X Y
Z
V Z
U V W
X Y Z
U X

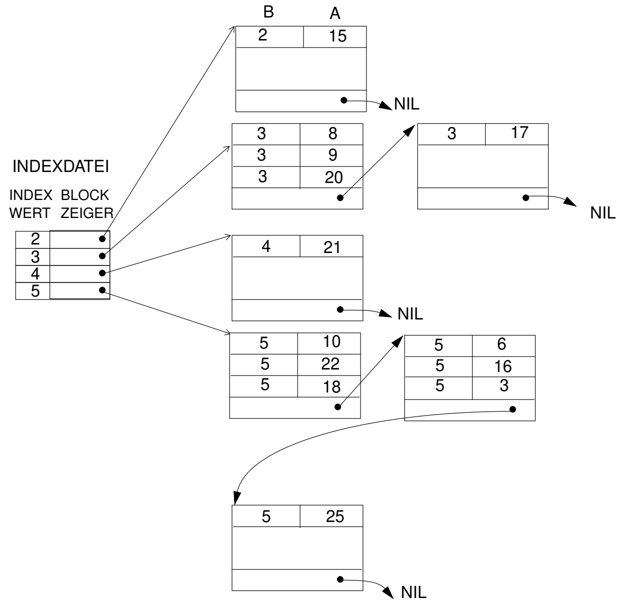



V X Y
Z
V Z
U
V X Y
W
Z
V Z



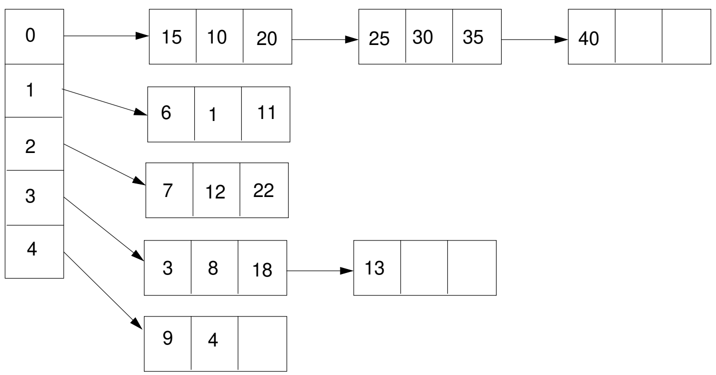
C E
A B
C D
E F G
E
A B D
E F G



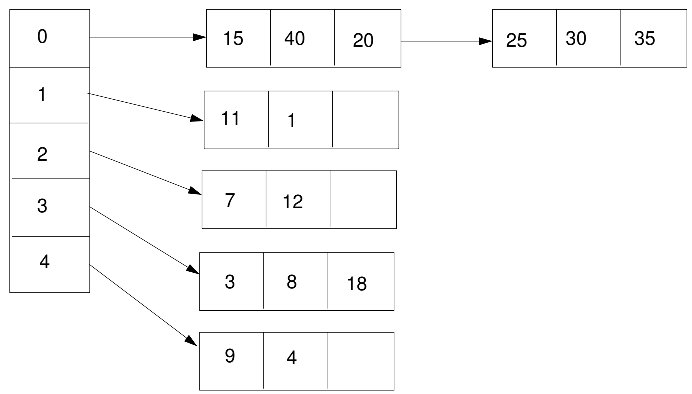
B A
2 15
NIL
3 8
3 9
3 20
3 17
NIL
INDEXDATEI
INDEX WERT
BLOCK ZEIGER
2
3
4
5
4 21
NIL
5 10
5 22
5 18
5 6
5 16
5 3
5 25
NIL



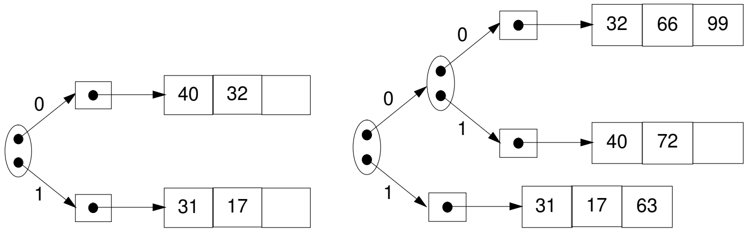



0
1
2
3
4
15 10 20
25 30 35
40
6 1 11
7 12 22
3 8 18
13
9 4



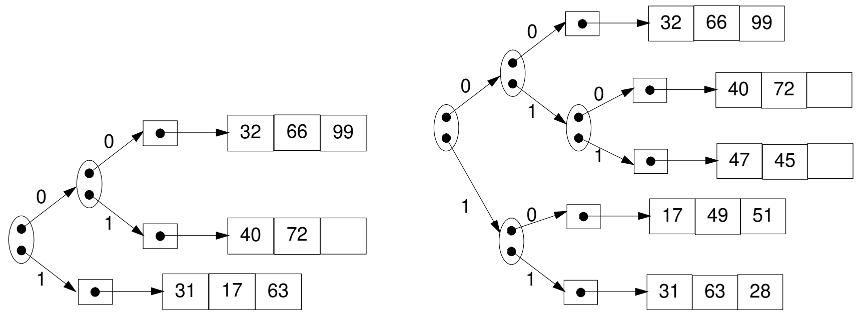



0
1
2
3
4
15 40 20
25 30 35
11 1
7 12
3 8 18
9 4


0
1
40 32
31 17
0
0
1
1
32 66 99
40 72
31 17 63

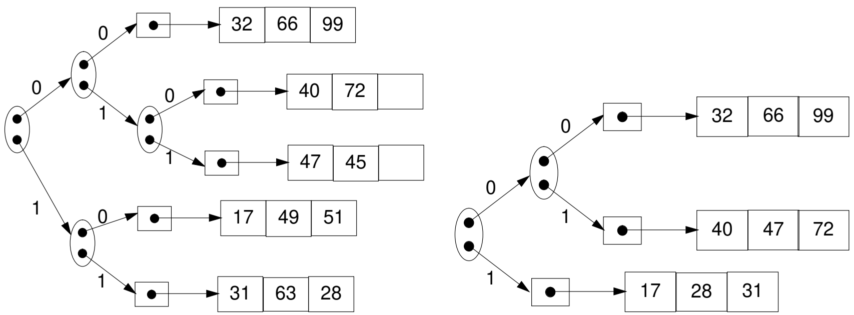





32 66 99
40 72
31 17 63
32 66 99
40 72
47 45
17 49 51
31 63 28

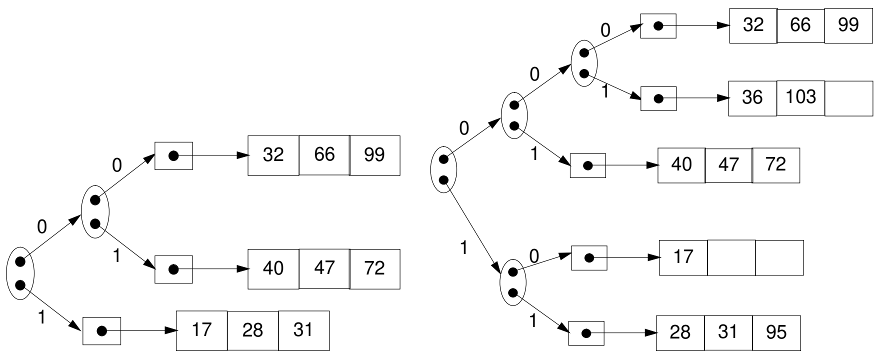





32 66 99
40 72
47 45
17 49 51
31 63 28
32 66 99
40 47 72
17 28 31






32 66 99
40 47 72
17 28 31
32 66 99
36 103
40 47 72
17
28 31 95

*Beispiel:*

Gegeben ist eine geordnete Datei mit 35000 Sätzen. Jeder Satz enthält 102 Bytes, 26 davon hat der Schlüssel. Ein Block besteht aus 1024 Bytes. Ein Zeiger zu einem Block oder Unterblock ist 4 Bytes lang.

Für eine B*-Baum Organisation werden alle Blöcke so weit wie möglich gefüllt. Berechnen Sie die Anzahl der Blöcke, die für den Index benötigt werden. Erläutern Sie den Rechenvorgang!

*Lösung:*

3500 Blöcke für die Datensätze:

$$\text{Sätze pro Block} = \left\lfloor \frac{\text{1024 Byte pro Block}}{\text{102 Byte pro Satz}} \right\rfloor = 10$$

$$\text{Anzahl Blöcke} = \left\lceil \frac{\text{35000 Sätze}}{\text{10 Sätze pro Block}} \right\rceil = \text{3500 Blöcke}$$

35 Einträge pro Indexblock: (26 + 4)*34 = 1020 + 4 = 1024