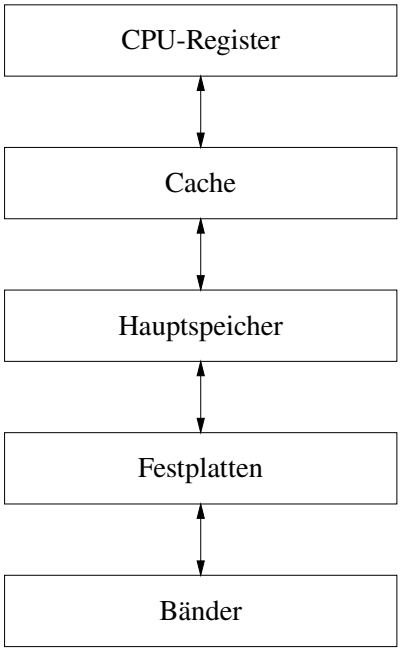
Physische Datenorganisation

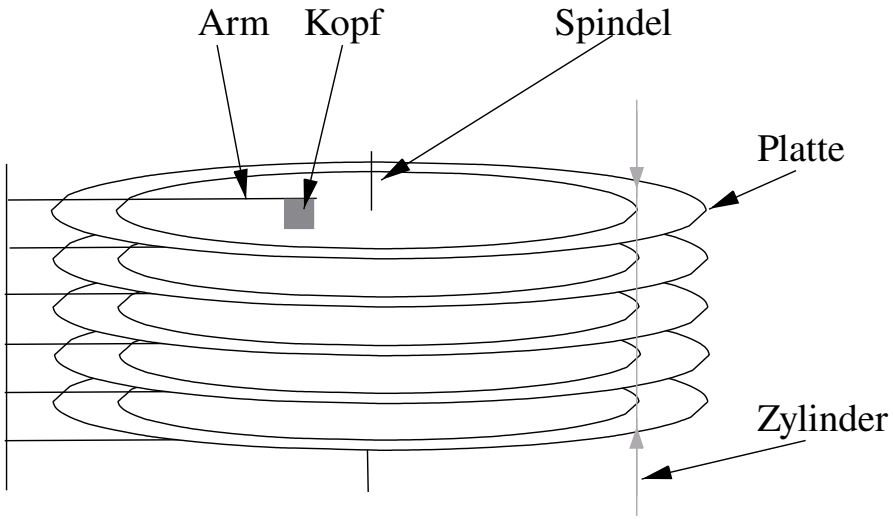
Physische Datenorganisation

Speicherhierarchie

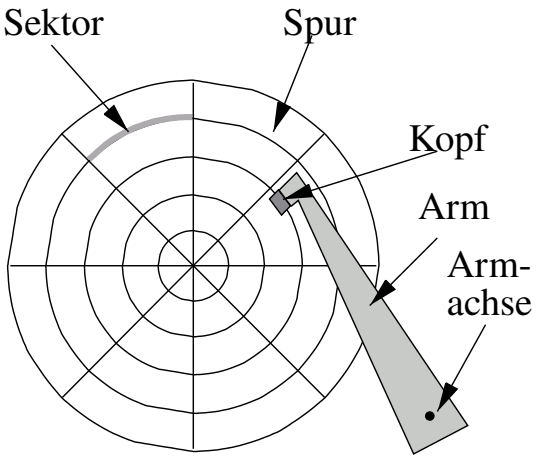
- Verschiedene Schichten der Speicherung
- Je höher in der Hierarchie, desto schneller, teurer und kleiner
- Unterschiede sind meistens in Größenordnungen
- Am wichtigsten für DBMS: Hauptspeicher und Festplatten



Aufbau einer Festplatte



a. Seitenansicht



b. Aufsicht

Aufbau einer Festplatte(2)

- Lesen eines Blocks:
  - Positionierung des Kopfes (Seek-Time)
  - Rotation zum Anfang des Blocks (Latenzzeit)
  - Lesen des Blocks (Lesezeit)

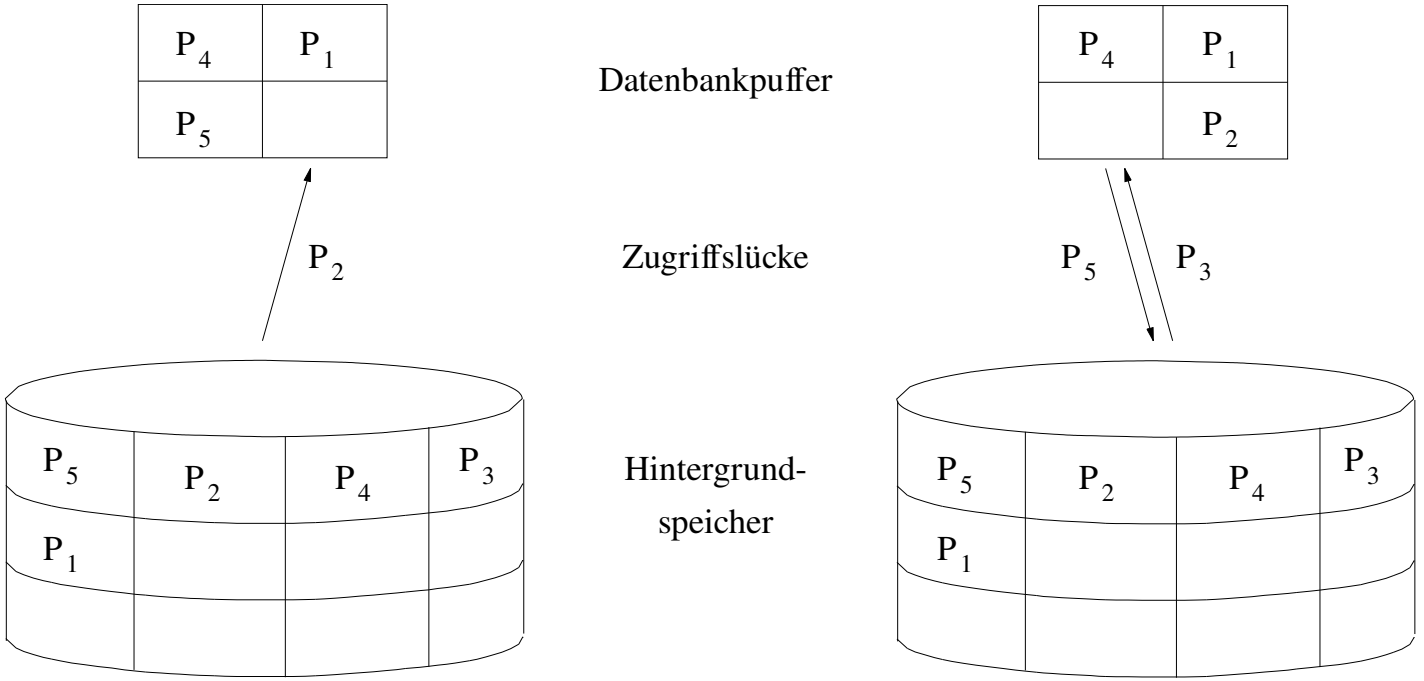
Daten sind meistens groß und wertvoll, deshalb wird häufig RAID (redundant array of inexpensive disks) verwendet.

Wichtigste RAID-Typen:

- RAID0: Striping (kein “echtes” RAID)
- RAID1: Spiegelung
- RAID0+1: Striping und Spiegelung
- RAID3/4: Striping mit Parity-Platte
- RAID5: Striping mit verteilter Parity

Es gibt noch mehr RAID-Typen, z.B. RAID6 für mehr Ausfallsicherheit.

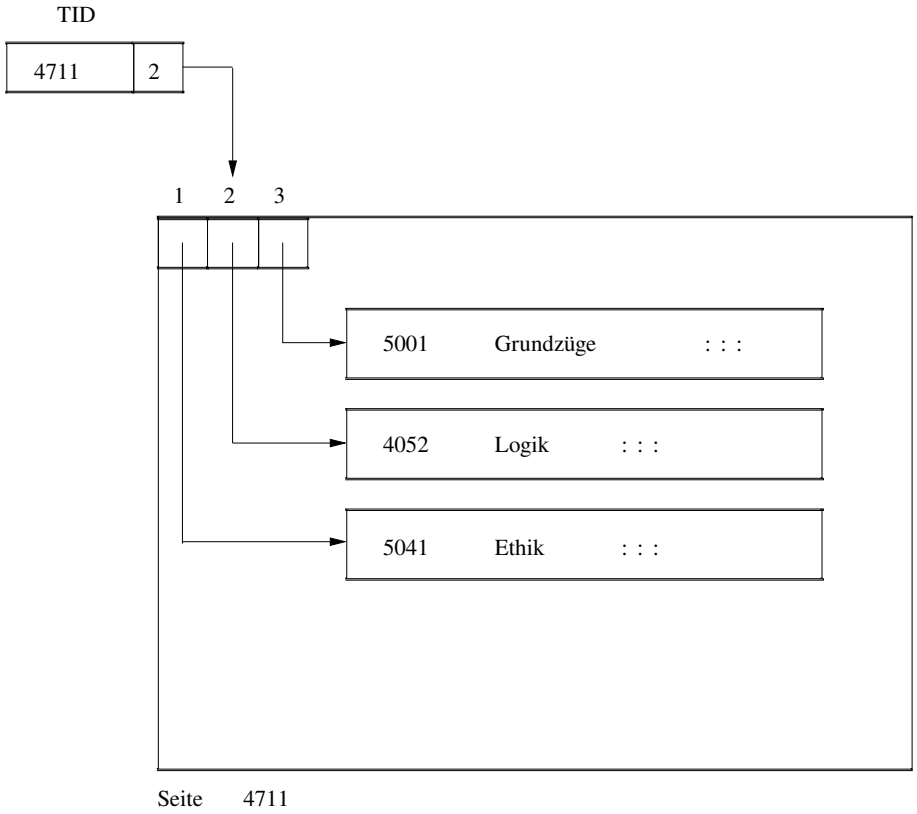
Datenbankpuffer



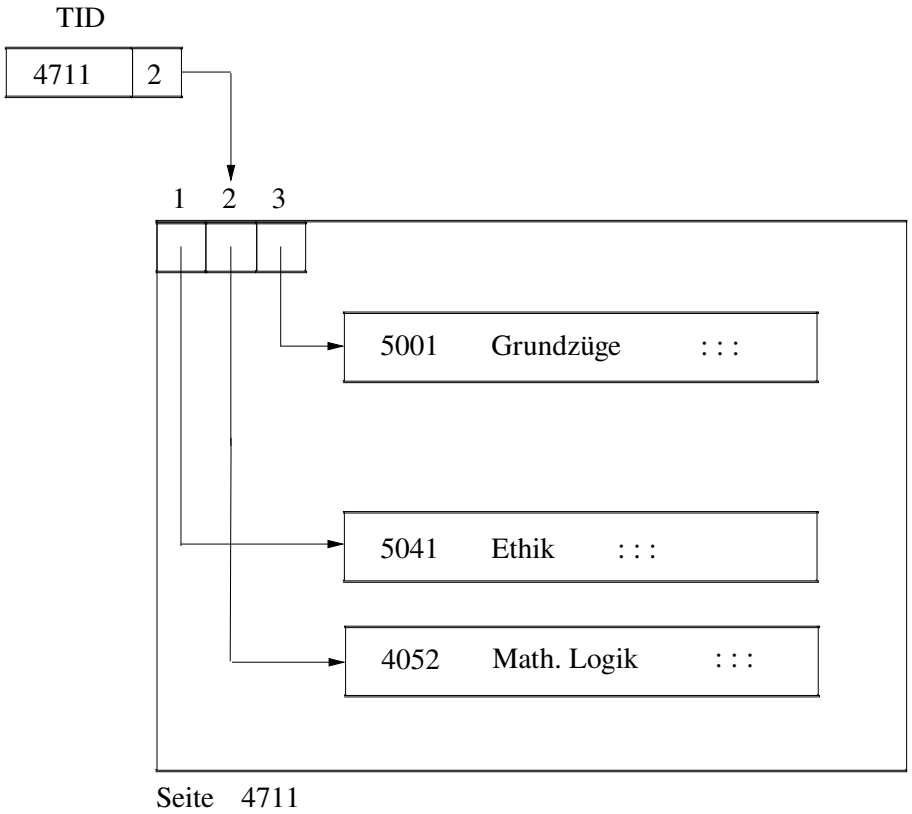
Speicherung von Relationen

- Die Tupel einer Relation werden auf mehreren Seiten im Hintergrundspeicher gespeichert
- Jede Seite enthält interne Datensatztabelle mit Verweisen auf die Tupel innerhalb der Seite (slotted pages)
- Tupel werden über *Tupel-Identifikatoren* referenziert

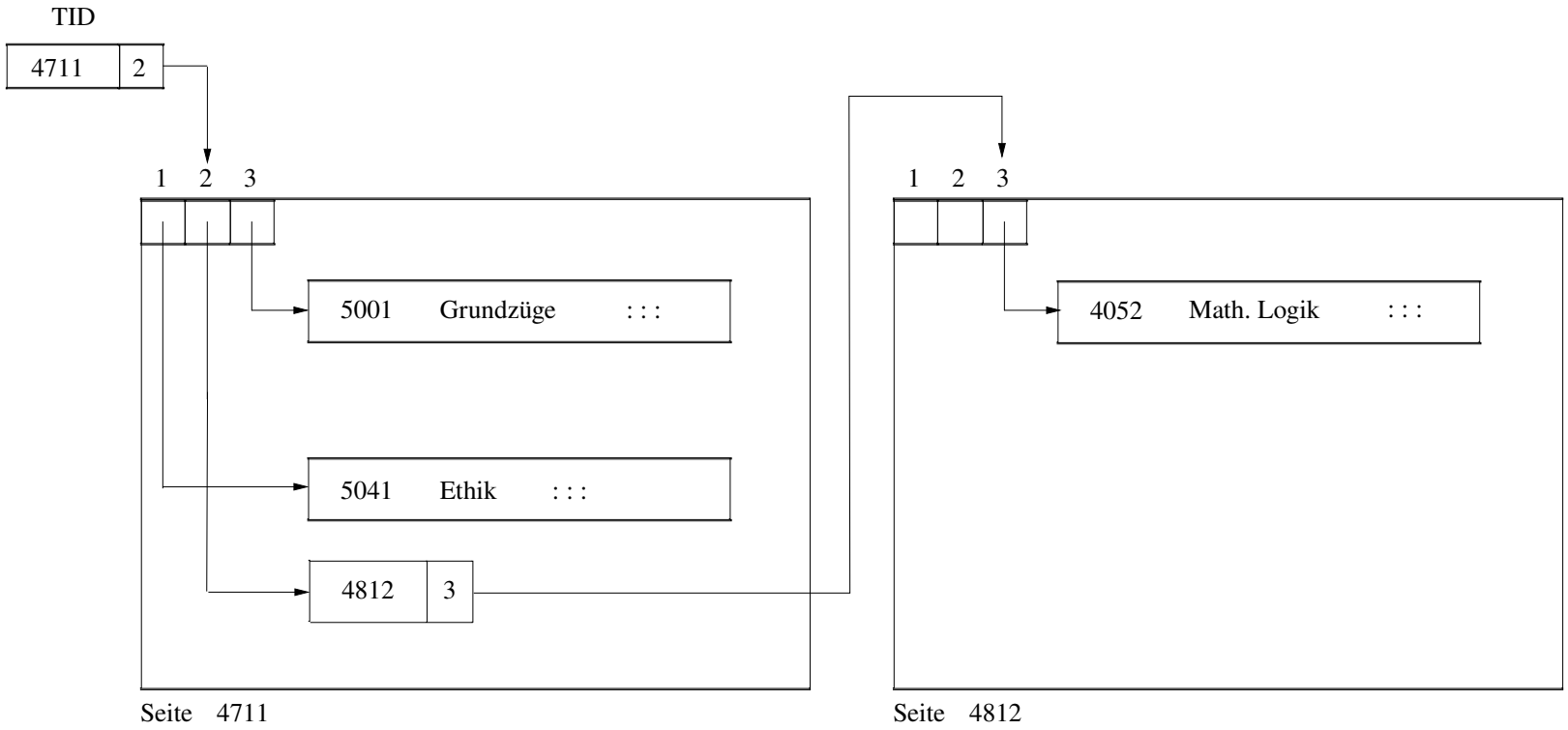
Das TID-Konzept



Verdrängung innerhalb Seite



Verdrängung auf andere Seite



Datentransfer

- Die Tupel aller Relationen nacheinander in den Hauptspeicher zu holen, ist einfachste Art, Anfrage zu bearbeiten
- Ist leider auch mit die teuerste
- Bei näherer Betrachtung stellt man folgendes fest:
  - Oft erfüllt nur ein Bruchteil der Tupel die Anfragebedingungen
  - Anfragen haben oft ähnliche Prädikate
  - Festplatten erlauben wahlfreien Zugriff

- Indexstrukturen nutzen diese Eigenschaften von Anfragen aus, um das transferierte Datenvolumen klein zu halten
- Sie erlauben schnellen assoziativen Zugriff auf die Daten
- Nur der Teil der Daten, der zur Beantwortung der Anfrage wirklich gebraucht wird, wird in den Hauptspeicher geholt
- Zwei bedeutende Indexierungsansätze
  - Hierarchisch (Bäume)
  - Partitionierung (Hashing)

Hierarchische Indexe

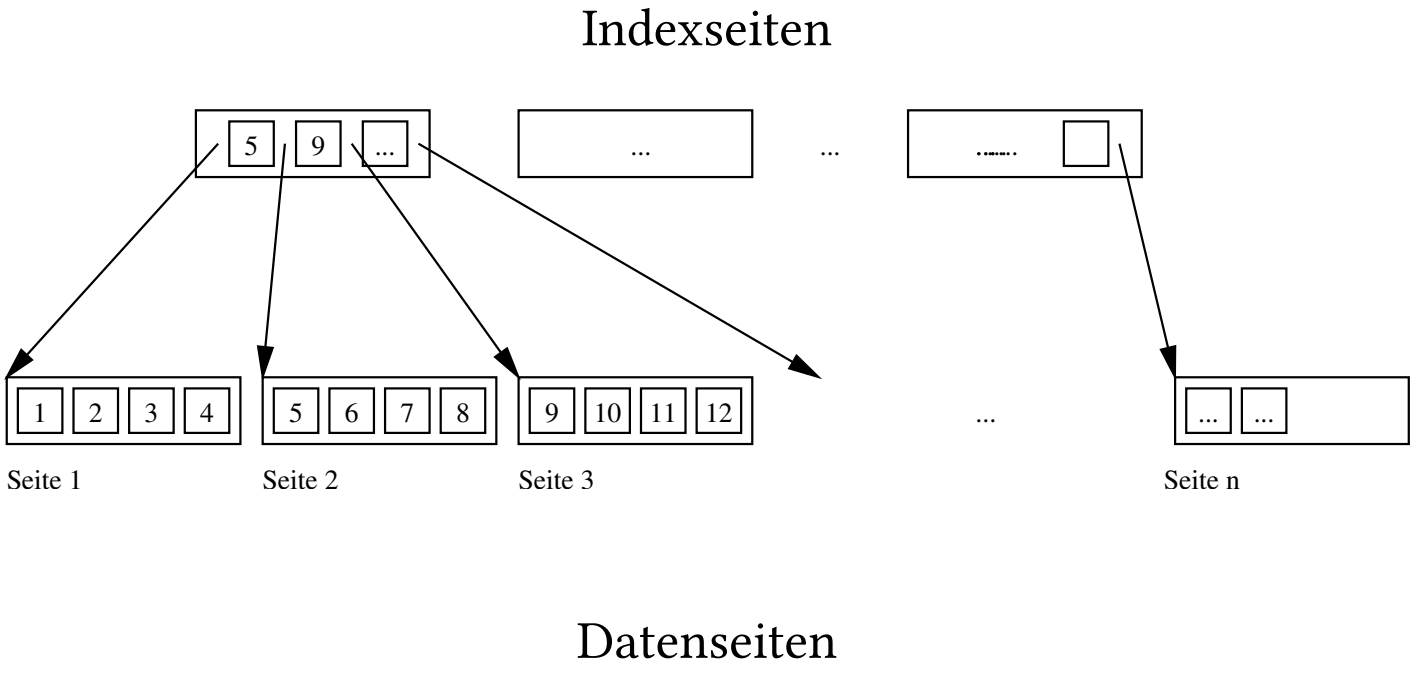
- Wir betrachten zwei hierarchische Indexstrukturen:
  - ISAM (Index-Sequential Access Method)
  - B-Bäume

ISAM

- ISAM war Vorgänger von B-Bäumen
- Hauptidee ist, die Tupel auf dem indexierten Attribut zu sortieren und eine Indexdatei darüber anzulegen
- Ähnlich wie ein Daumenindex an der Seite eines Buches, durch den man schnell durchblättern kann

Beispiel

- Der Studierende mit der Matrikelnummer 13542 wird gesucht
- Alle Tupel der Relation `Studenten` sind sortiert nach MatrNr



Beispiel(2)

- Während der Anfragebearbeitung geht man durch die Indexseiten und sucht die Stelle, an der 13542 passt
- Von dort aus wird die referenzierte Datenseite geholt
- Vorteil: die Anzahl der Indexseiten ist normalerweise sehr viel kleiner als die Anzahl der Datenseiten, d.h. es wird I/O gespart
- Es können auch Bereichsanfragen beantwortet werden (z.B. bei einer Suche nach allen MatrNr zwischen 765 und 1232: zuerst die erste passende Datenseite finden und von dort aus sequentiell durch die Datenseiten bis MatrNr 1232 laufen)

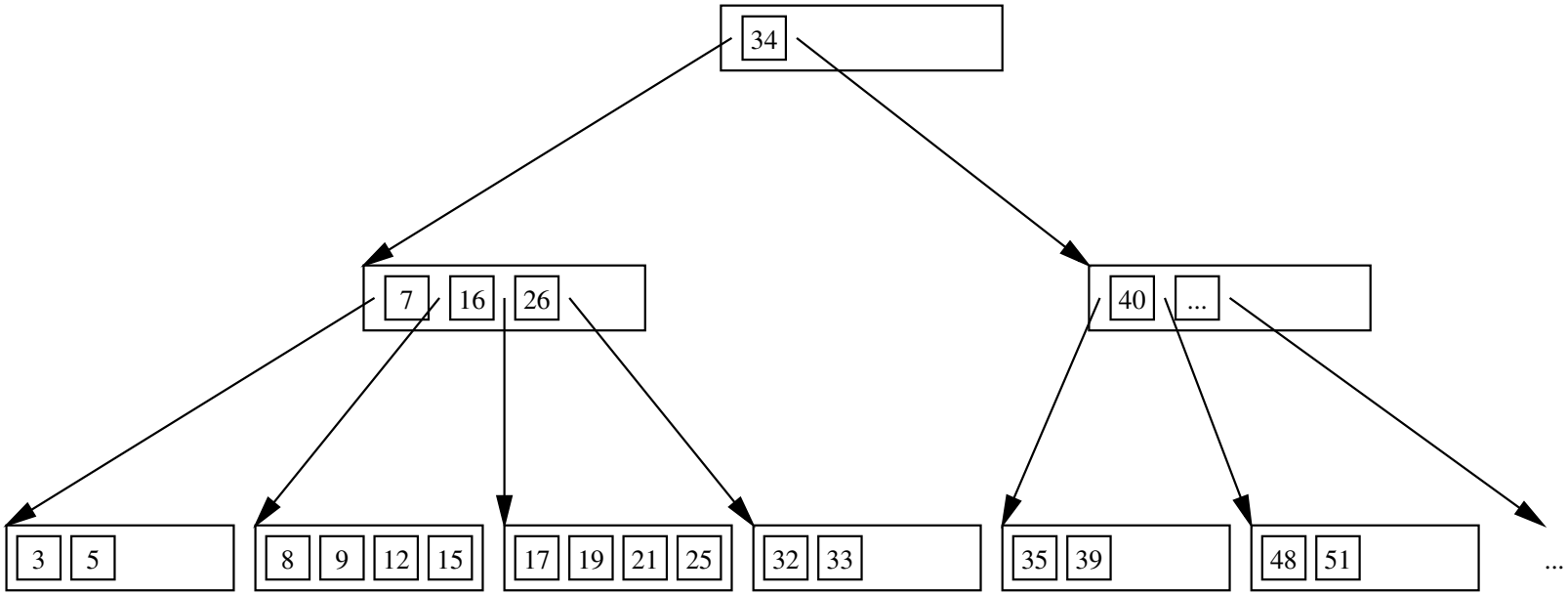
Probleme mit ISAM

- Obwohl Suche auf ISAM einfach und schnell ist, kann die Instandhaltung des Indexes teuer werden
- Wenn ein Tupel auf eine gefüllte Datenseite eingefügt werden soll, muss Platz geschaffen werden: die Datenseite wird auf zwei Seiten aufgeteilt (wir müssen Sortierung beibehalten)
- Das erzeugt wiederum einen neuen Eintrag auf einer Indexseite
- Wenn auf der Indexseite auch kein Platz mehr ist, müssen die Einträge verschoben werden, um Platz zu schaffen

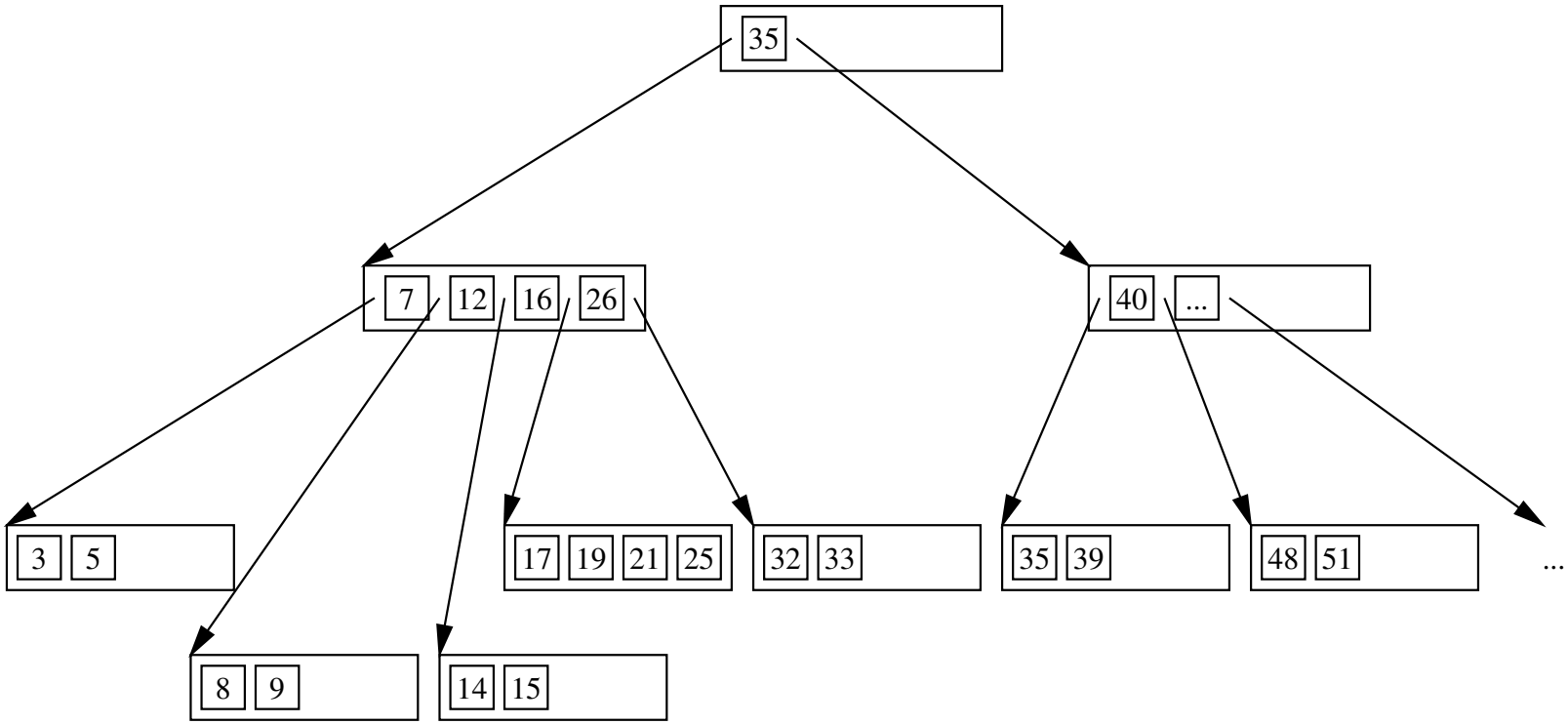
Weitere Probleme

- Obwohl die Anzahl der Indexseiten kleiner als die Anzahl der Datenseiten, kann Durchlauf der Indexseiten trotzdem lange dauern
- Idee: warum richtet man nicht Indexseiten für die Indexseiten ein?
- Das ist im Prinzip die Idee eines B-Baums

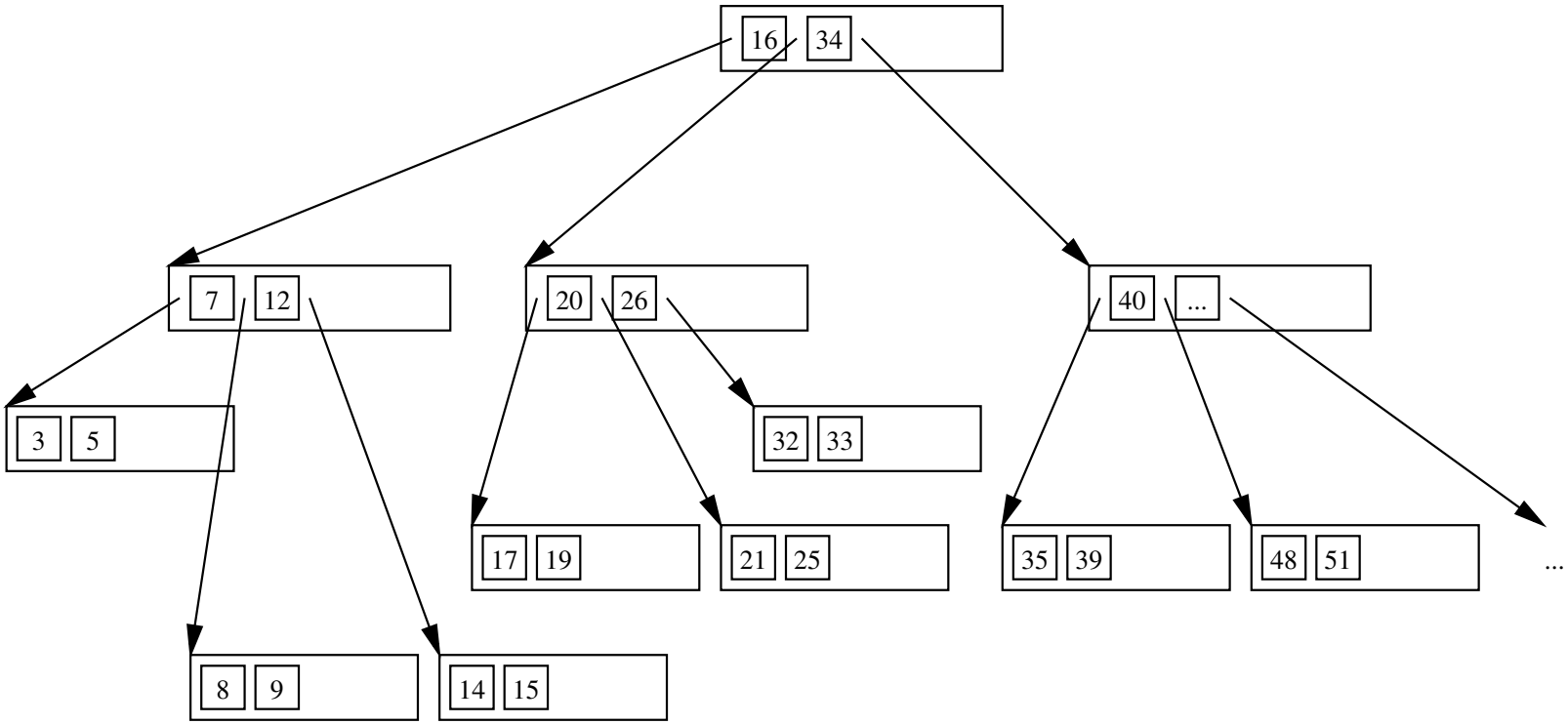
B-Baum (Beispiel)



Einfügen von 14



Einfügen von 20



Eigenschaften eines B-Baums

- Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge
- Jeder Knoten (außer der Wurzel) hat mindestens i und höchstens $2i$ Einträge (in obigem Beispiel $i = 2$)
- Die Einträge in jedem Knoten sind sortiert
- Jeder Knoten (außer Blätter) mit n Einträgen hat $n + 1$ Kinder

Eigenschaften eines B-Baums(2)

- Seien $p_0, k_1, p_1, k_2, ...k_n, p_n$ die Einträge in einem Knoten (p_j sind Zeiger, k_j Schlüssel)
- Dann gilt folgendes:
  - Der Unterbaum, der von p_0 referenziert wird, enthält nur Schlüssel kleiner als k_1
  - p_j zeigt auf einen Unterbaum mit Schlüsseln zwischen k_j und k_{j+1}
  - Der Unterbaum, der von p_n referenziert wird, enthält nur Schlüssel größer als k_n

Einfügealgorithmus

1. Finde den richtigen Blattknoten, um den neuen Schlüssel einzufügen
2. Füge Schlüssel dort ein
3. Falls kein Platz mehr da
   3.1. Teile Knoten und ziehe Median heraus
   3.2. Füge alle Knoten kleiner als Median in linken Knoten, alle größer als Median in rechten Knoten
   3.3. Füge Median in Elternknoten ein und passe Zeiger an

Einfügealgorithmus(2)

1. Falls kein Platz in Elternknoten
   - Falls Wurzelknoten, kreiere neuen Wurzelknoten und füge Median ein, passe Zeiger an
   - Ansonsten, wiederhole 3. mit Elternknoten

Löschalgorithmus

- In einem Blattknoten kann ein Schlüssel einfach gelöscht werden
- In einem inneren Knoten muss Verbindung zu den Kindern bestehen bleiben
  - Deshalb wird der nächstgrößere (oder nächstkleinere) Schlüssel gesucht (in entsprechendem Kindknoten)
  - Dieser Schlüssel wird an die Stelle des gelöschten Schlüssels geschrieben

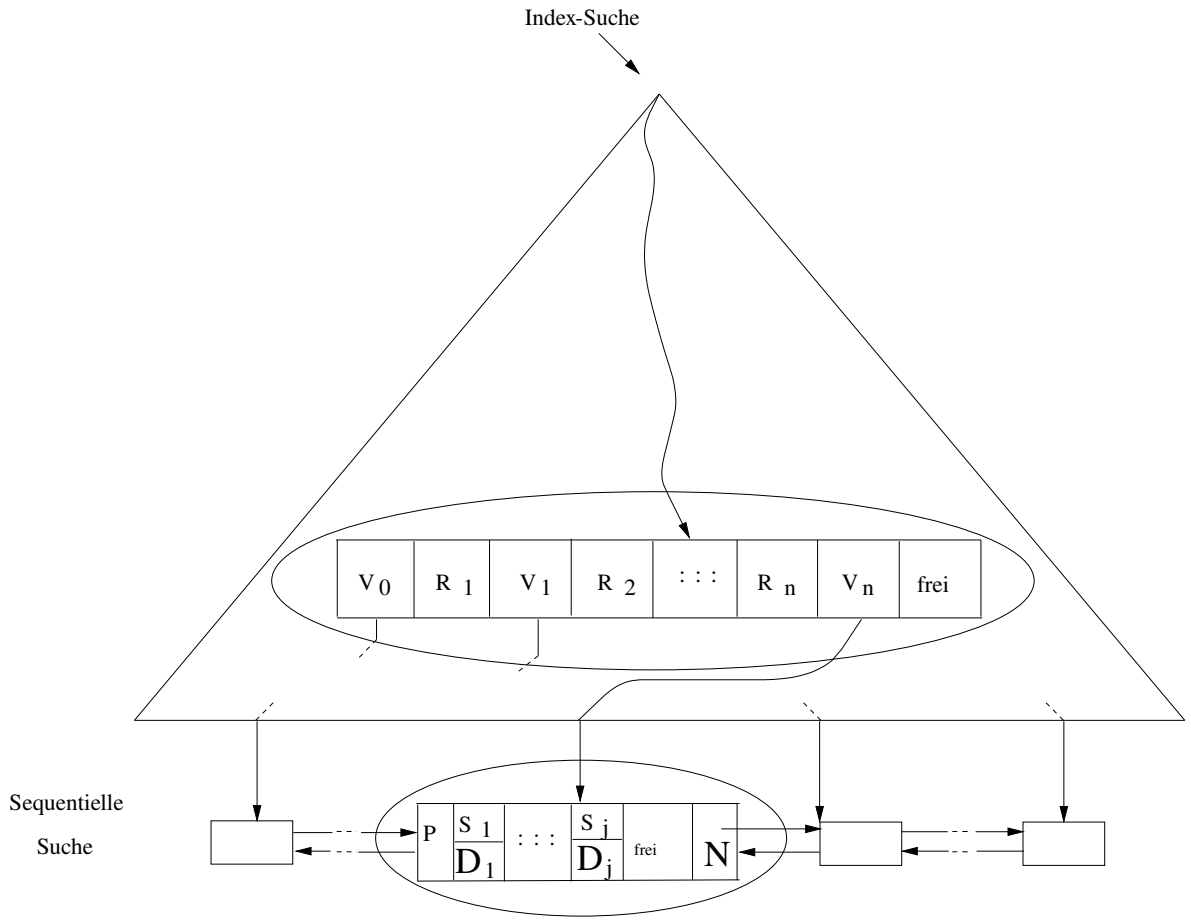
Löschalgorithmus(2)

- Nach Löschen eines Schlüssels kann ein Knoten unterbelegt sein (weniger als i Einträge haben)
- Dann wird dieser Knoten mit einem Nachbarknoten verschmolzen
- Das kann eine Unterbelegung im Elternknoten hervorrufen, d.h. Elternknoten muss ebenfalls verschmolzen werden
- Da dieses Verfahren relativ aufwendig ist und Datenbanken eher wachsen als schrumpfen, wird diese Verschmelzung oft nicht realisiert

B^+-Bäume

- Die Performanz eines B-Baums hängt stark von der Höhe des Baumes ab, deswegen wollen wir hohen Verzweigungsgrad der inneren Knoten
- Abspeichern von Daten in inneren Knoten reduziert den Verzweigungsgrad
- B^+-Bäume speichern lediglich Referenzschlüssel in inneren Knoten, die Daten selbst werden in Blattknoten gespeichert
- Meistens sind die Blattknoten noch verkettet, um schnelle sequentielle Suche zu ermöglichen

Schematische Darstellung



Präfix-B$^+$-Bäume

- Weitere Verbesserung ist der Einsatz von Referenzschlüsselpräfixen, z.B. bei langen Zeichenketten
- Es muss nur irgendein Referenzschlüssel gefunden werden, der linken vom rechten Teilbaum trennt:
  - Müller $\leq$ P $<$ Schmidt
  - Systemprogramm $\leq$? $<$ Systemprogrammierer

Partitionierung

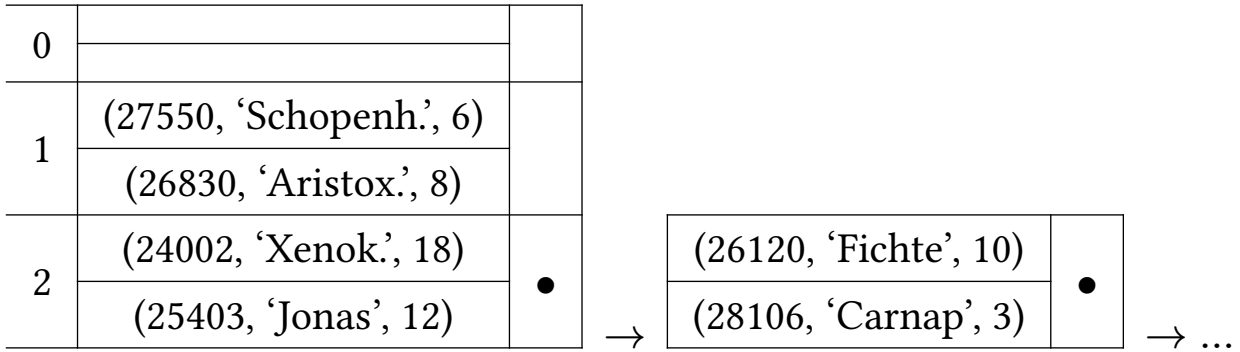
- Bäume brauchen im Schnitt $\log_k(n)$ Seitenzugriffe, um ein Datenelement zu lesen (k =Verzweigungsgrad, n=Anzahl indexierter Datensätze)
- Hashtabellen (partitionierende Verfahren) benötigen im Schnitt zwei Seitenzugriffe

- Hashfunktion $h(x) = x \mod 3$

0	
1	(27550, ‘Schopenhauer’, 6)
2	(24002, ‘Xenokrates’, 18)
	(25403, ‘Jonas’, 12)

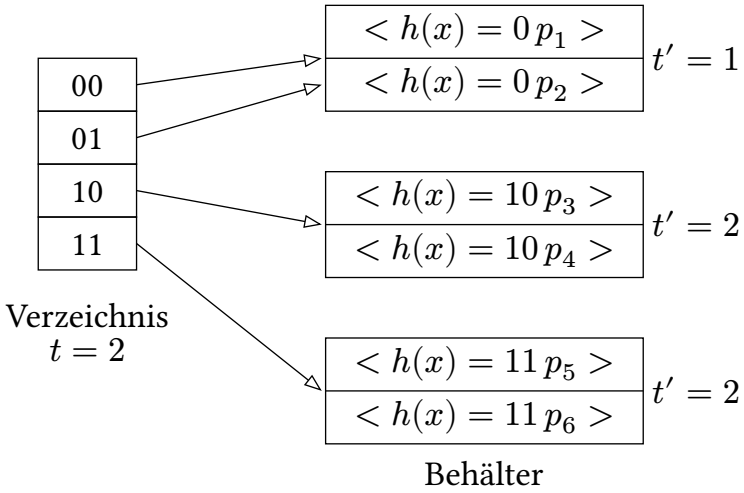
Hashing(2)

- Kollisionsbehandlung



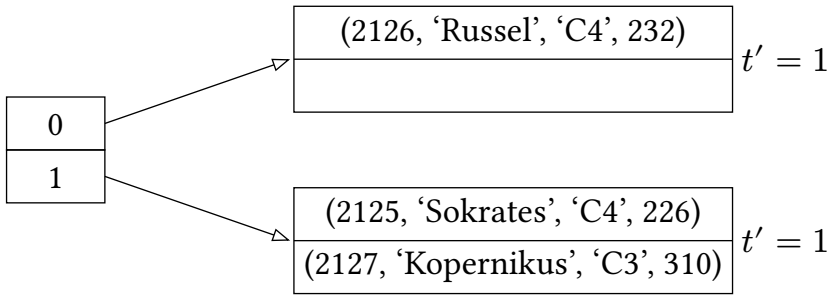
- Ineffizient bei nicht vorhersehbarer Datenmenge

Erweiterbares Hashing



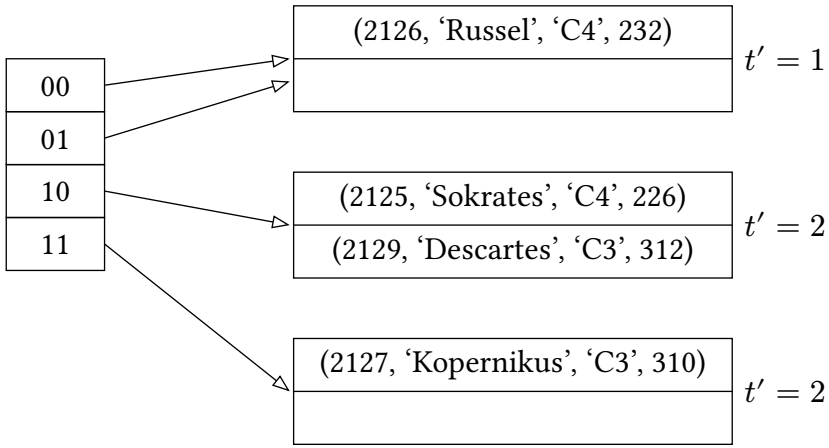
Konkretes Beispiel

	$h(x)$	
x	d	p
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001



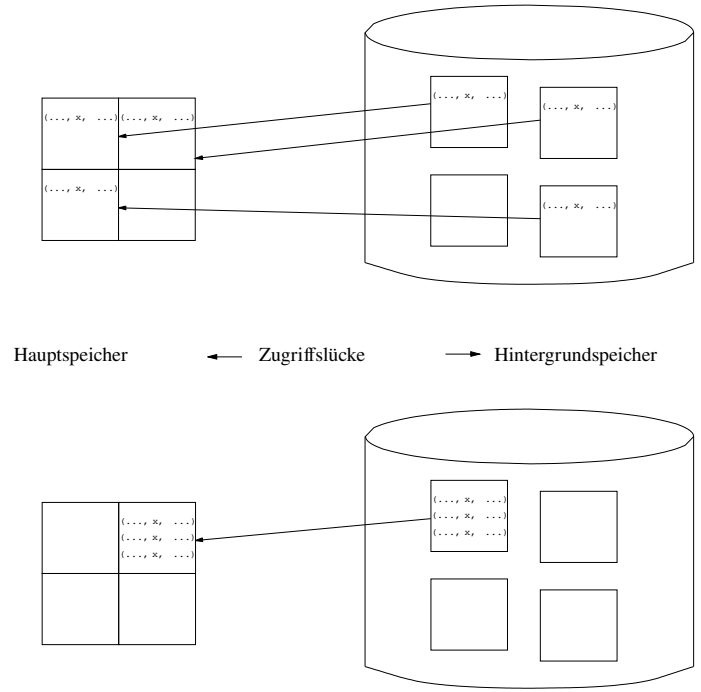
(2129, Descartes, ...) einfügen

	$h(x)$	
x	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001

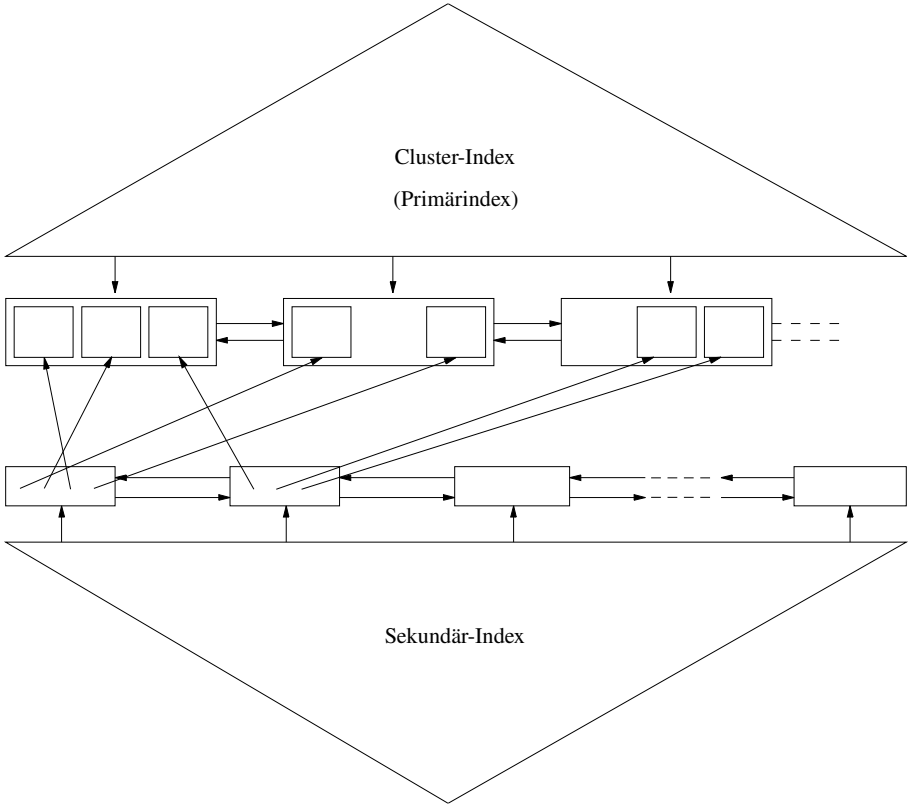


Ballung (Clustering)

```
select *
from   R
where  A = x;
```



Indexe und Ballung



Verzahnte Ballung

			Seite P_i
2125	Sokrates	C4	226
5041	Ethik	4	2125
5049	Mäeutik	2	2125
4052	Logik	4	2125
2126	Russel	C4	232
5043	Erkenntnistheorie	3	2126
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
			Seite P_{i+1}
2133	Popper	C3	52
5259	Der Wiener Kreis	2	2133
2134	Augustinus	C3	309
5022	Glaube und Wissen	2	2134

Zusammenfassung

- TIDs ist geschickte Abbildung der Relationen auf Seiten im Hintergrundspeicher
- Indexe beschleunigen Zugriffe auf Datenelemente (auf Kosten der Update-Operationen)
- B^+-Bäume sind die Standardindexstrukturen in relationalen DBMS, sowohl für Punkt- als auch für Bereichsanfragen geeignet