2. Externspeicherverwaltung

Dieses Kapitel umfaßt die Aufgaben und Abbildungsverfahren der beiden untersten Schichten des Schichtenmodells (Speicherzuordnungs- und Seitenzuordnungsstrukturen)

- **Abbildung von Dateien und Blöcken**
  - Zweistufige Speicherhierarchie
  - Allgemeine Aufgaben der Externspeicherverwaltung

- **Dateisystem**
  - Operationen, Adressierung und Abbildung
  - Verfahren der Blockzuordnung
    - statisch
    - dynamische Extent-Zuordnung
    - dynamische Block-Zuordnung

- **Maßnahmen zur Fehlertoleranz**
  - Lesen und Schreiben von Blöcken
  - Nutzung von Speicherredundanz

- **Abbildung von Segmenten und Seiten**
  - Segmentkonzept
  - verzögerte Einbringstrategien
    - Schattenspeicherkonzept (shadow page mechanism)
    - Zusatzdatei (differential file)
  - Bewertung der Seitenzuordnungsverfahren

Abbildung von Dateien und Blöcken – Speicherzuordnungsstrukturen

- **Beispiel: Aufruf einer Operation an der oberen Schnittstelle**

```
          PAM       UPAMDAT, RD, FECB=ESTBLOCK, HP=1
            ⋮
UPAMDAT   FCB       FCBTYPE=PAM, LINK=EIN, IOAREA1=EINBER
```

- **Abbildungsfunktion**

Blocknummer ——— cyl-#, track-#, rec-#

- **Durchführung der E/A:**

$$\left.\begin{array}{ll} \text{Armpositionierung:} & \text{EXCP} \rightarrow \text{CCW} \\ \text{Spurauswahl:} & \text{EXCP} \rightarrow \text{CCW} \\ \text{Satzübertragung:} & \text{EXCP} \rightarrow \text{CCW} \end{array}\right\} \text{I/O-Prozessor}$$

- **Eigenschaften der oberen Schnittstelle**
  - Menge durchnumerierter Blöcke innerhalb von Dateien
  - Lese- und Schreibzugriff über die Blocknummer
  - Blockzugriff kann auf Lesefehler führen, Blöcke können alte oder ungültige Daten enthalten

Abbildung von Dateien und Blöcken

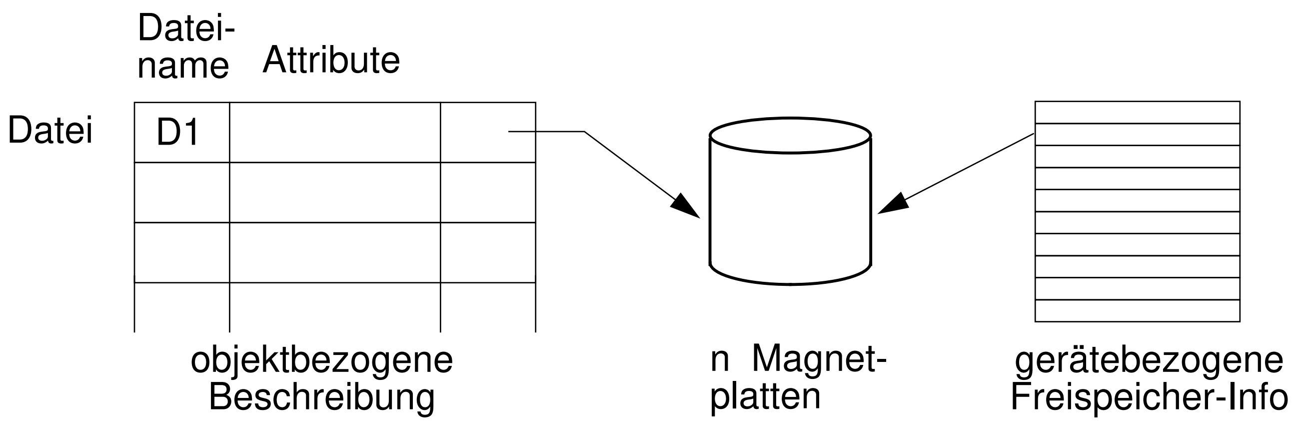
- **Anforderungen**
  - Verwaltung externer Speichermedien
  - Verbergen von Geräteeigenschaften
  - Abbildung von physischen Blöcken
  - Kontrolle des Datentransports von/zum Hauptspeicher
  - ggf. Realisierung einer mehrstufigen Speicherhierarchie
  - Fehlertoleranzmaßnahmen
    (stabiler Speicher, Spiegelplatten etc.)

- **Gründe für ein Dateikonzept**
  - selektive Aktivierung von Dateien: on/offline-Problem
  - dynamische Definition
  - temporäre Dateien
  - Einsatz unterschiedlich schneller Speichermedien
  - kürzere Adreßlängen

$$\text{DB-Speicher} \;\hat{=}\; \text{Menge von Dateien}$$

Realisierung eines Dateisystems

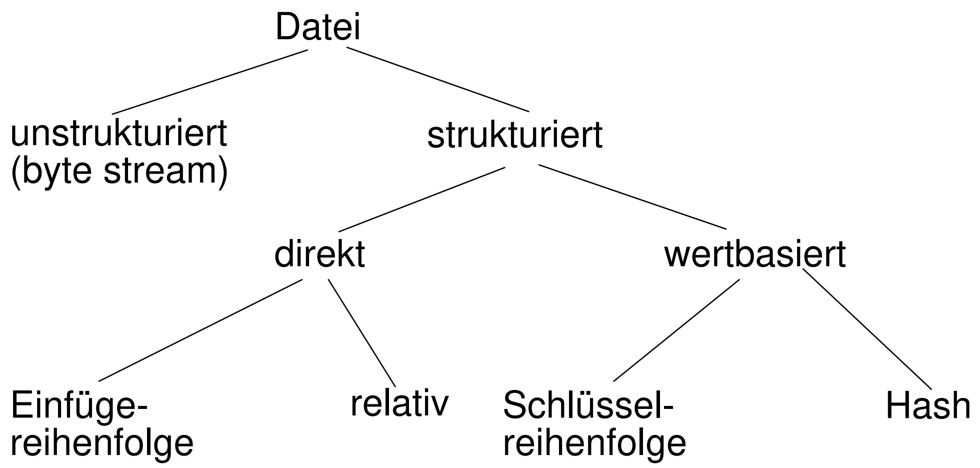
- **Dateikatalog für alle Dateien (Urdatei)**
  - feste Position
  - effiziente Implementierung/Verarbeitung
- Beispiel: **Dateikatalog**



- **Objektbezogene Beschreibung durch Dateideskriptor**
  - Dateiname, OwnerID, . . .
  - Zugriffskontrolliste
  - Zeitinformation über Erzeugung, letzter Zugriff, letzte Archivierung, . . .
  - Dateigröße, Externspeicherzuordnung, . . .
- **Freispeicherverwaltung für Externspeicher**
  - formatierte Bitlisten, hierarchische Struktur
- **Anlegen/Reservieren von Speicherbereichen**
  - Erstzuweisung
  - Erweitern
- **Einheit des physischen Zugriffs: Block**
  - Blöcke variabler Länge?
  - feste Blocklänge pro Datei
  - chained I/O wichtig für hohe E/A-Leistung

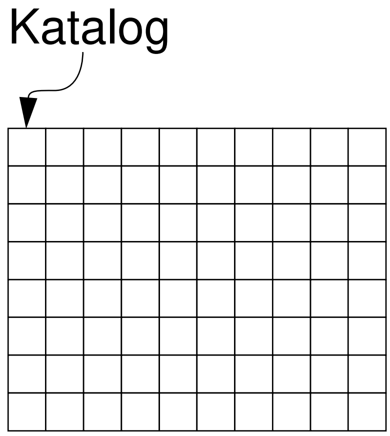
Dateiorganisationsformen

- **Dateisystem**
  - verwaltet die erzeugten Dateien und führt die Lese-/Schreibzugriffe durch
  - hält einen Deskriptor für jede Datei
- **Organisationsformen**



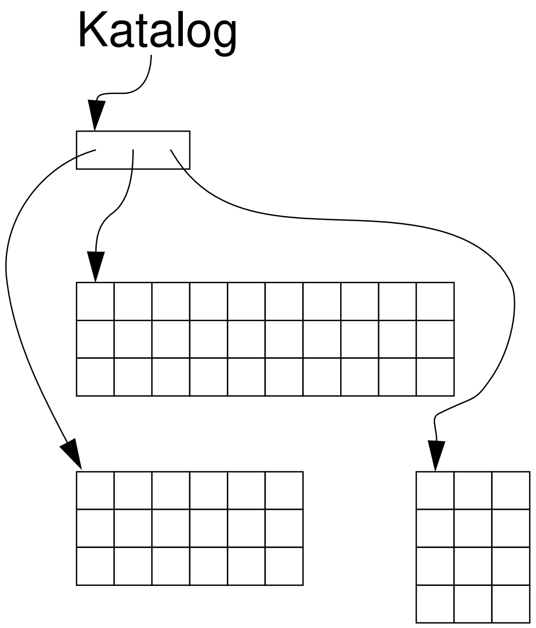
  - Datei in Einfügereihenfolge (*entry-sequenced*):
    Einfügen an Dateiende, Satzadresse ist RBA (relative Byteadresse), sequentieller Zugriff oder direkter Zugriff über RBA
  - relative Datei:
    Organisationsform ist ARRAY OF RECORDS
  - wertbasierte Dateien erlauben Zugriff über Satzschlüssel
  - Datei in Schlüsselreihenfolge (*key-sequenced*):
    Speicherung der Sätze in Schlüsselreihenfolge, direkter Zugriff über Schlüssel und sortiert-sequentieller Zugriff
  - Hash-Datei:
    direkter Zugriff über Schlüsseltransformation.
- **Bezeichnungen** in existierenden Systemen (logische Zugriffsmethoden)
  - SAM, ISAM/VSAM, . . .
  - BDAM/PAM, . . .

Blockzuordnungsverfahren



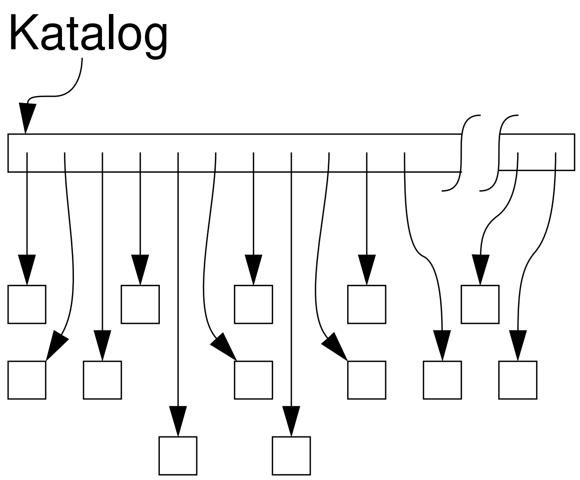
Statische Datei-Zuordnung

- direkte Adressierung
- minimale Zugriffskosten
- keine Flexibilität



Dynamische Extent-Zuordnung

- Adressierung über eine kleine Tabelle
- geringe Zugriffskosten
- mäßige Flexibilität



Dynamische Block-Zuordnung

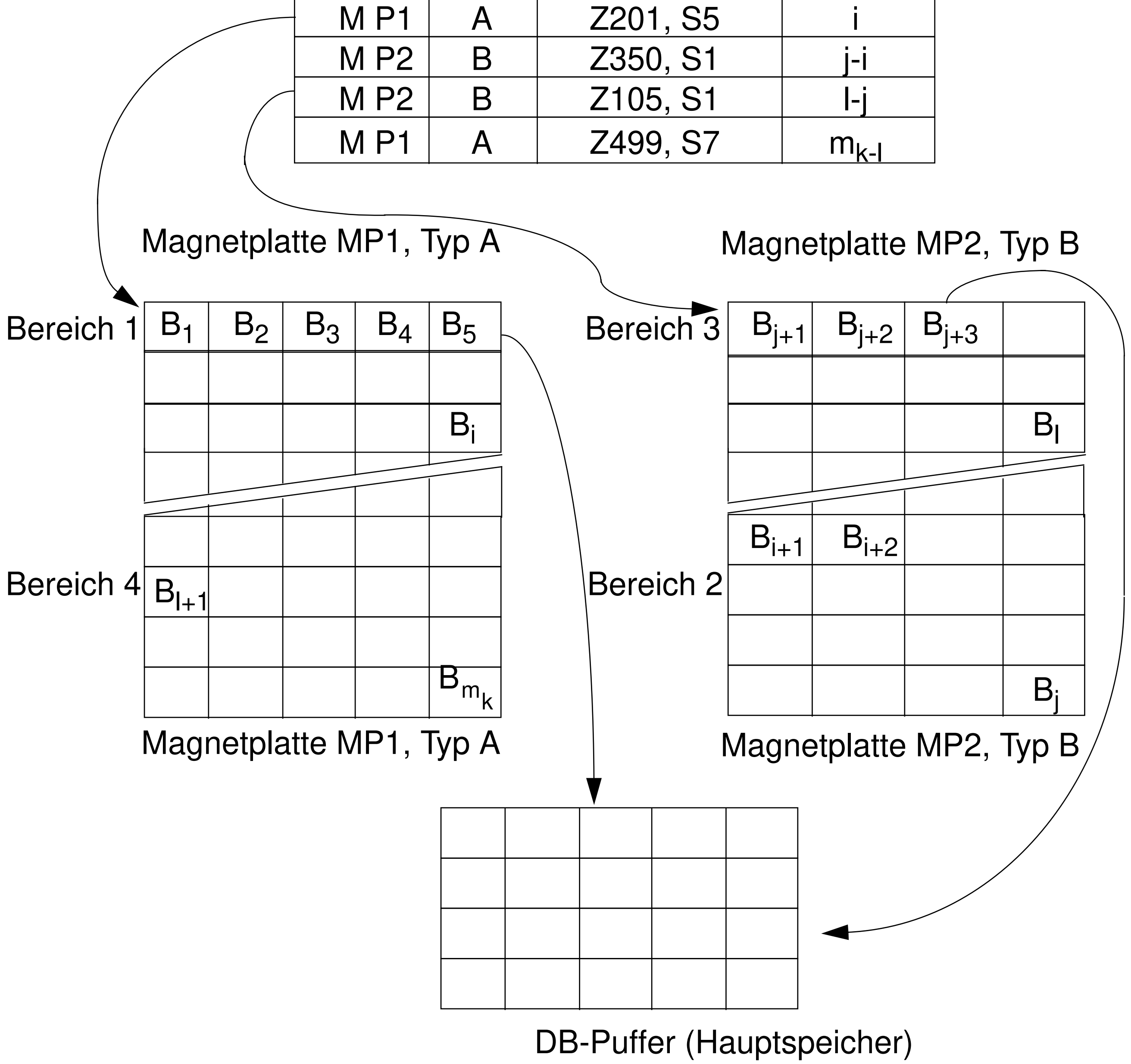
- Adressierung über eine große Tabelle
- hohe Zugriffskosten
- maximale Flexibilität

Blockzuordnung durch Extent-Tabellen

Directory

Extent-Tabelle für Datei D_k

Name	Typ	Bereichsanfang ZYL, SPUR	Anzahl Blöcke
M P1	A	Z201, S5	i
M P2	B	Z350, S1	j-i
M P2	B	Z105, S1	l-j
M P1	A	Z499, S7	m_{k-l}



Dargestellte Aktionen der Zugriffsprimitive:
- Hole Block B_5
- Hole Block B_{j+3}

Falls ein Block zum Schreiben in den DB-Puffer geholt wird, ist er nach Freigabe wieder zurückzuschreiben

In typischen Implementierungen dieses Verfahrens kann ein Primärbereich (*primary extent*) angelegt werden, der um bis zu 15 Sekundärbereiche (*secondary extents*, mit jeweils gleicher Größe) erweitert werden kann.

Maßnahmen zur Fehlertoleranz

- **MTTF von verschiedenen Plattenfehlern** [1]

Tpye of Error	MTTF	Recovery	Consequences
Soft data read error	1 hour	Retry or ECC	None
Recoverable seek error	6 hours	Retry	None
Maskable hard data read error	3 days	ECC	Remap to new sector and rewrite good data
Unrecoverable data read error	1 year	None	Remap to new sector, old data lost
Device needs repair	5 years	Repair	Data unavailable
Miscorrected data read error	10^7 years	None	Read wrong data

- **Einfaches Lesen**
  - Lesen kann durch transiente Fehler gestört werden
  - Zusatzmaßnahme: erfolgloses Lesen wird n-mal wiederholt (sicheres Lesen)

- **Einfaches Schreiben**
  - Schreiben des Blocks als atomare Aktion (ganz oder überhaupt nicht) kann nicht garantiert werden
  - Schreiben kann durch transiente und dauerhafte Fehler zu falschen Resultaten führen

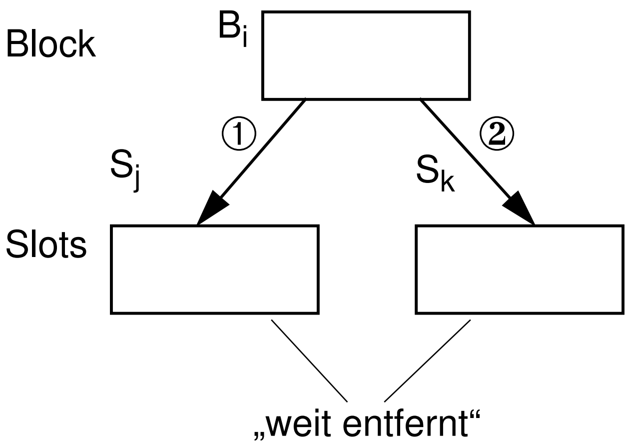
  ➥ Schreibfehler im Katalog?

- **Sicheres Schreiben (*read-after-write*)**
  - Nach einem einfachen Schreiben wird der Block sofort wieder gelesen und mit dem Originalblock verglichen.
  - Operationsfolge wird wiederholt, bis Block erfolgreich geschrieben ist
  - Schreiben ist gesichert gegen transiente Fehler.
    Dauerhafte Fehler können jedoch zu falschen Resultaten führen.

1. Gray, J., Reuter, A. : Transaction Processing – Concepts and Techniques, Morgan Kaufmann, 1993.

Maßnahmen zur Fehlertoleranz (2)

- **Stabiles Schreiben (*duplexed write*)**
  - Jeder Block hat eine Versionsnummer, die bei jedem stabilen Schreiben erhöht wird.
  - Jeder Block wird in festgelegter Reihenfolge in zwei verschiedene Slots S_j und S_k geschrieben
  - Prinzip: Stabiler Speicher



- einfaches Schreiben:
  ① dann ② (synchron)
- Atomarität für <u>einen</u> Block
- verschiedene Platten, Kontroller, E/A-Pfade

- Annahme: ein Block wird nicht in einen falschen Slot geschrieben, sonst ist read-after-write erforderlich.
- Lesen von B_i erfolgt erst von Slot S_j. Falls es erfolgreich ist, wird angenommen, daß es sich um die jüngste gültige Version von B_i handelt.
- Falls das Lesen von Slot S_j scheitert, wird Slot S_k gelesen.
- Da ein Systemausfall nur einen Schreibvorgang unterbrechen kann, ist bei Wiederanlauf stets eine Version des Blockes verfügbar.

➥ **Schreiben ist somit gesichert gegen dauerhafte Fehler.**
Daß beide Versionen nicht lesbar sind, gilt als **unerwartet.**

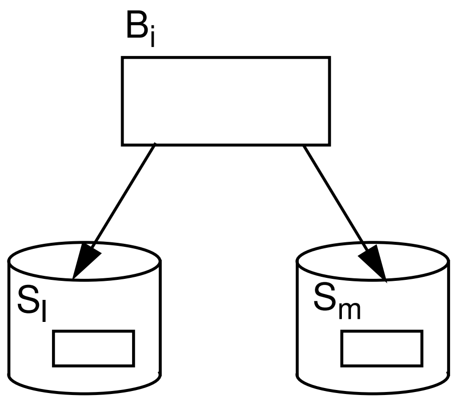
Weitere Prinzipien der Speicherredundanz

- **Einbringeinheit: 1 Block**

  ➥ Fehlertoleranzmaßnahme für <u>einen</u> Block
  gegen Schreibfehler, Systemausfall, Block- (Spur-, Geräte-) Zerstörung

- **Schreiben mit Logging (*logged write*)**
  - Nach dem Lesen wird ein Block B_i mit seinem alten Inhalt auf einen sicheren Platz L geschrieben.
  - Nach Aktualisierung wird B_i mit einem einfachen Schreiben auf seinen alten Platz zurückgeschrieben (ggf. *read-after-write*).
  - Falls das Schreiben erfolgreich war, wird die Kopie von B_i auf L nicht mehr gebraucht.

- **Prinzip: Spiegelplatten**



- auf Platten- oder Dateibasis
- synchron oder asynchron
- gleichzeitig oder zeitlich versetzt (Schattenprozeß)
- keine Atomarität automatisch durch Algorithmus

Weitere Prinzipien der Speicherredundanz (2)

- **Erkennung von fehlerhaften Blöcken**
  - Schutz vor gewissen Verfälschungen:
    - Parity-Bits
    - Prüfsummen
  - Platten-Hardware kann durch Parity-Bits eigenständig herausfinden, ob ein Sektor vollständig geschrieben wurde oder nicht

    ➥ ausreichend, wenn ein Block ganz in einem Sektor (hier gleich Slot) gespeichert werden kann

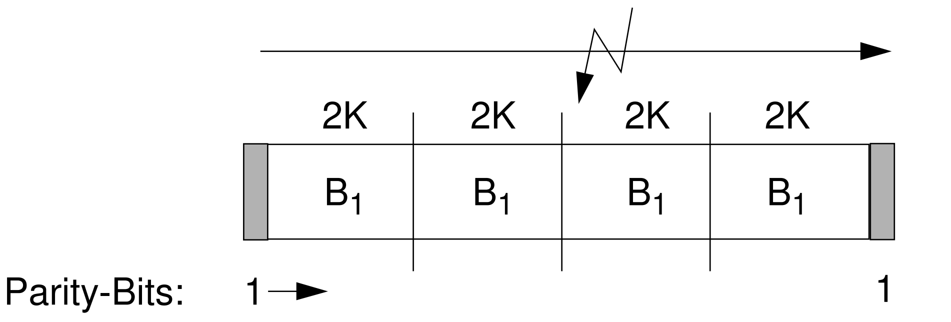
- **Slot = Sektor**
  - Benutzung von jeweils einem Bit im ersten und letzten Byte eines Blockes
  - Beiden Konsistenzbits werden jeweils identische Werte zugewiesen

- **Multi-Sektor-Slots**
  - Fehlerhafter Block wird nur erkannt, wenn beim Schreiben der Sektoren die durch den Block vorgegebene Reihenfolge eingehalten wird
  - SCSI-Laufwerke nehmen selbständig zur Leistungsoptimierung eine Umordnung der Schreibvorgänge auf den Sektoren vor
  - Prüfsumme über den gesamten Block reduziert die Wahrscheinlichkeit erheblich, ein partielles Schreiben zu übersehen, kann dies jedoch nicht ausschließen (enorm teuer)
  - Entsprechend den n Sektoren wird ein Block (logisch) unterteilt; aus jedem dieser Teile wird ein Bit genommen und als Prüfbit betrachtet. Diese n Bits werden mit identischen Werten belegt und bei jedem Schreiben invertiert.

    ➥ Wie läßt sich erreichen, daß diese n Bits nicht die Benutzerdaten im Block verfälschen?

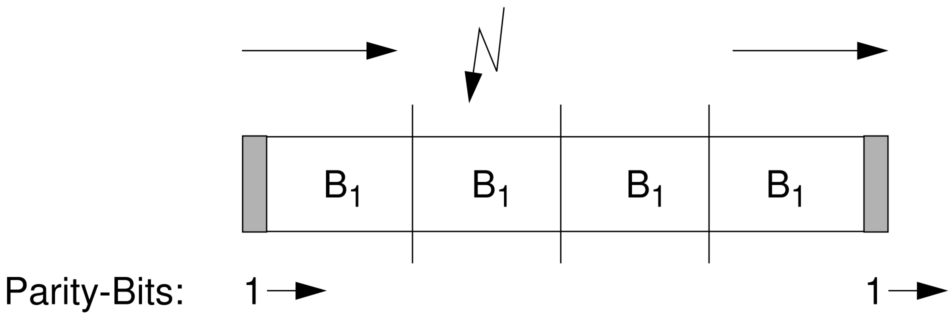
Erkennung unvollständig geschriebener Blöcke

1. **Sequentielles Schreiben**



für sequentielles Schreiben ok !

2. **Problem: Schreiben „außer der Reihe"**



3. **Lösung: Schreiben von Prüf-Bits in jeden Sektor**

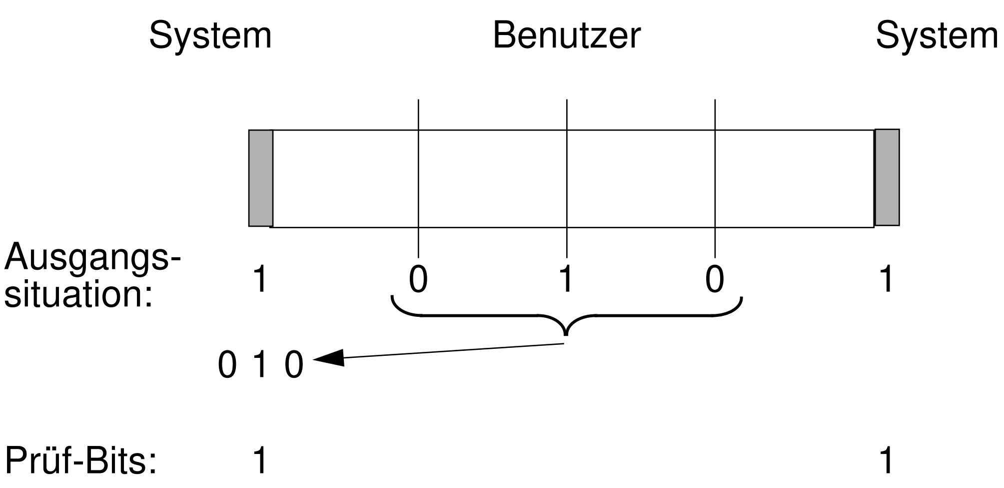


Abbildung von Segmenten und Seiten

- **Realisierung eines Segmentkonzeptes**[1]
  - Ermöglichung verzögerter Einbringstrategien
  - selektive Einführung von zusätzlichen Attributen, z. B. zur Erhöhung der Fehlertoleranz
  - Segmente als Einheiten des Sperrens, der Recovery und der Zugriffskontrolle
  - bei geeigneter Abbildung auf Dateien bleiben Vorteile des Dateikonzeptes erhalten

- **DB-Pufferverwaltung**
  (siehe Kapitel 3)
  - Bereitstellen und Freigeben von Seiten im DB-Puffer
  - Vorbereitung von E/A-Anforderungen an die Dateiverwaltung
  - Optimierung von Ersetzungsstrategien
  - Unterstützung von Segmenten verschiedenen Typs
  - neue Aufgaben:
    Verwaltung von Seiten variabler Länge und von langen Objekten

➥ **Aufteilung des logischen DB-Adreßraumes in Segmente mit sichtbaren Seitengrenzen**

---

1. In verschiedenen DBS werden bestimmte Segmenttypen auch als Tablespaces bezeichnet. Sie dienen der Speicherung von Tabellen (Relationen) sowie ggf. Indexstrukturen. In manchen DBS werden Indexstrukturen in eigenen Segmenttypen (sog. Indexspaces) verwaltet. Die Abbildung von Tablespaces auf mehrere Dateien ist dabei möglich.

Abbildung von Segmenten und Seiten – Seitenzuordnungsstrukturen

- **Beispiel: Operationen an der oberen Schnittstelle**

FIX $\mathbf{P_i}$

⋮ logische Seitennummer

UNFIX $\mathbf{P_i}$

- **Abbildungsfunktion:**

Seitennummer	⟷	Blocknummer
Hauptspeicher	⟷	Externspeicher

- **Aufgaben:**

Ersetzen eines alten Blocks	PAM UPAMDAT, WR,...
Lies den Block mit Seite P_i	PAM UPAMDAT, RD,...

- **Eigenschaften der oberen Schnittstelle**
  - Linearer Adreßraum, aufgeteilt in Seiten fester Länge
  - Innerhalb eines Segmentes können die Moduln der nächsthöheren Schicht frei adressieren wie in einem virtuellen Adreßraum
  - Ein DB-Segment ist nicht-flüchtig (wenn nicht explizit anders vereinbart)

Segmenttypen in einem DBMS

- **Klassifikation von Segmenttypen**

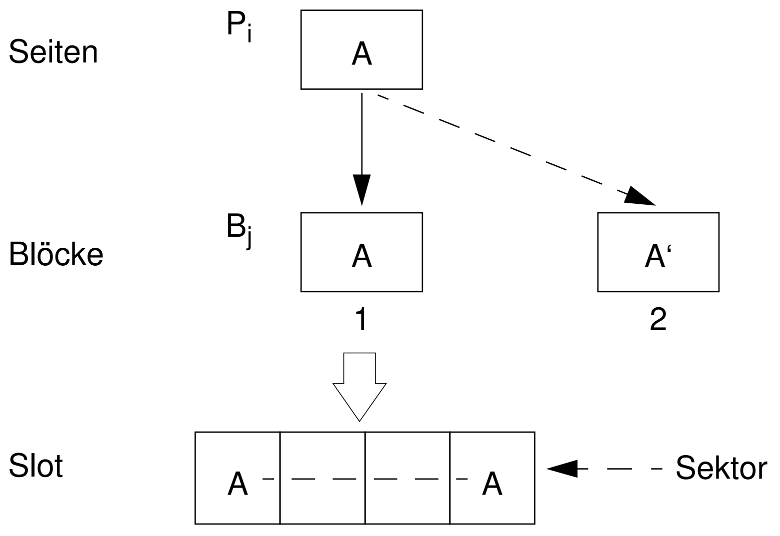
Eigenschaften \ Segment-Typen	Segment-Typ 1	Segment-Typ 2	Segment-Typ 3	Segment-Typ 4	Segment-Typ 5
Benutzung	öffentlich	privat	privat	privat	privat
Lebensdauer	permanent	permanent	permanent	permanent	temporär in Transaktion
Öffnen und Schließen	automatisch durch System		explizit durch Benutzer		
Wiederherstellung im Fehlerfall	automatisch durch System		explizit durch Benutzer	kein Wiederherstellungsmechanismus	

- **Beispiele für die Verwendung der Segmenttypen**

  - Typ 1: Katalog, Schema-Information, Log,
    alle gemeinsam benutzbaren DB-Teile
  - Typ 2: Teile der DB, die für bestimmte Benutzer
    bzw. Benutzergruppen reserviert sind
  - Typ 3: Lokale Kopien von Teilen der DB (Sichten)
    für einzelne Benutzer (*Snapshots*)
  - Typ 4: Hilfsdateien für Benutzerprogramme
  - Typ 5: Temporärer Speicher z. B. für Sortierprogramme

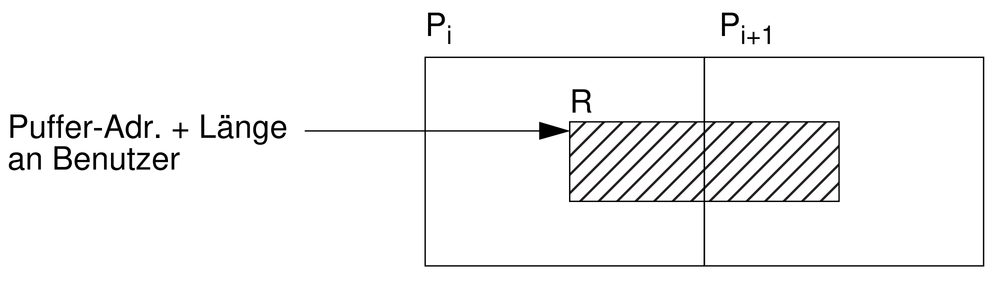
Warum sichtbare Seiten und Segmente?

- **Explizite Abbildung auf Blöcke und Dateien erhöht Fehlertoleranz**

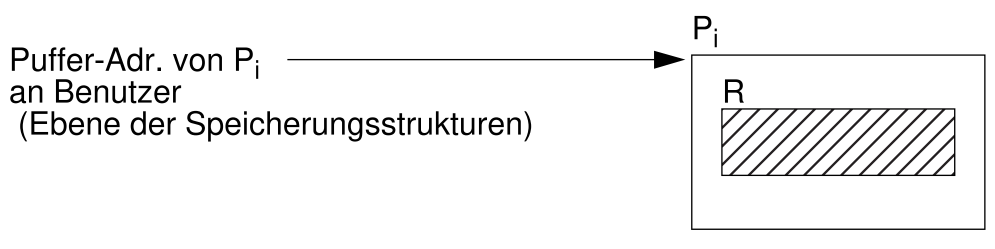


- **DB-Puffer mit Satzschnittstelle?**
  - „lineare“ Hauptspeicherabbildung: Probleme unsichbarer Seitengrenzen
  - „spanned record facility“
  - benachbarte Seiten müssen im DB-Puffer benachbart angeordnet sein

  ➥ erhebliche Abbildungs- und Ersetzungsprobleme

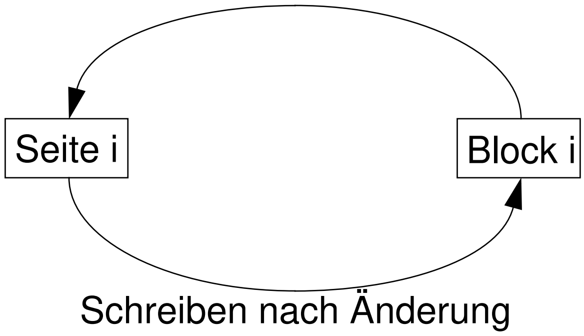


- **Sichtbare Seitengrenzen an der DB-Puffer-Schnittstelle**

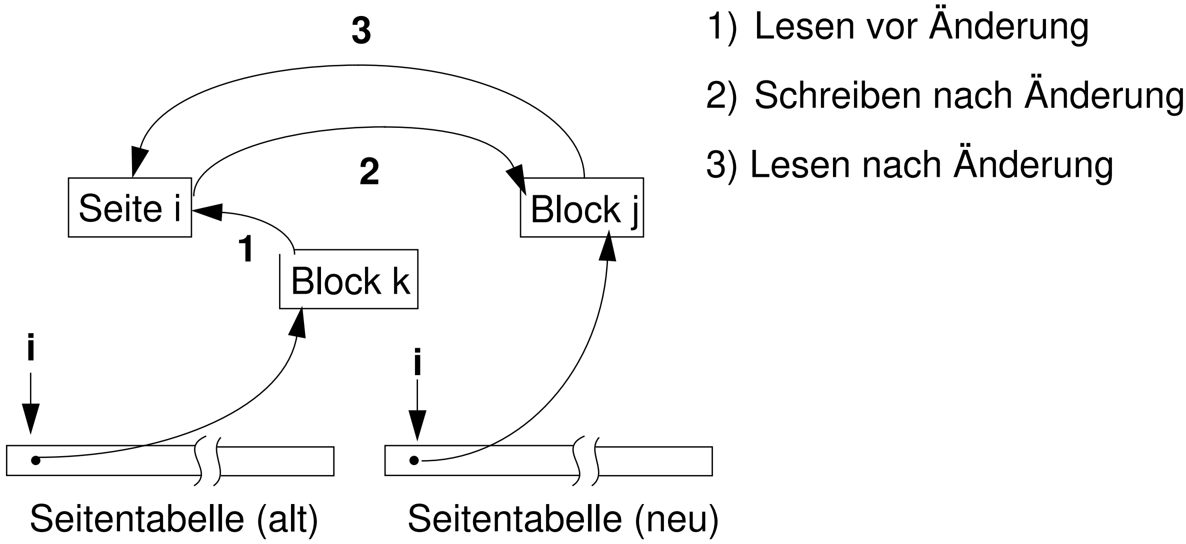


Verfahren zur Seitenzuordnung

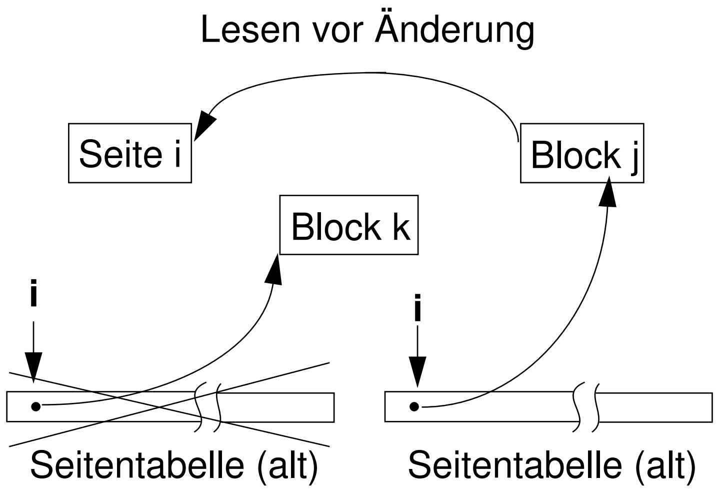
- **Direkte Seitenzuordnung (*update-in-place*)**



- **Indirekte Seitenzuordnung**



➥ Zustand nach verzögertem Einbringen (mengenorientiert)

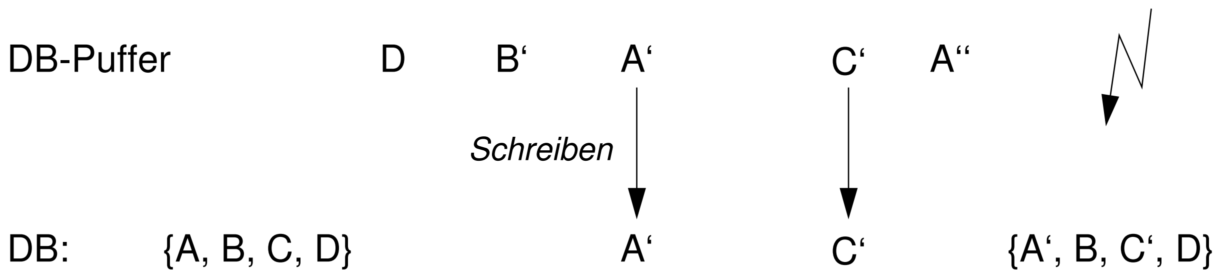


Einbringen von Seiten –
Konsistenz der DB nach Crash

- **direktes Einbringen:**

$$P_i \Rightarrow B_j$$
$$P_i' \Rightarrow B_j'$$

➥ Schreiben = Einbringen

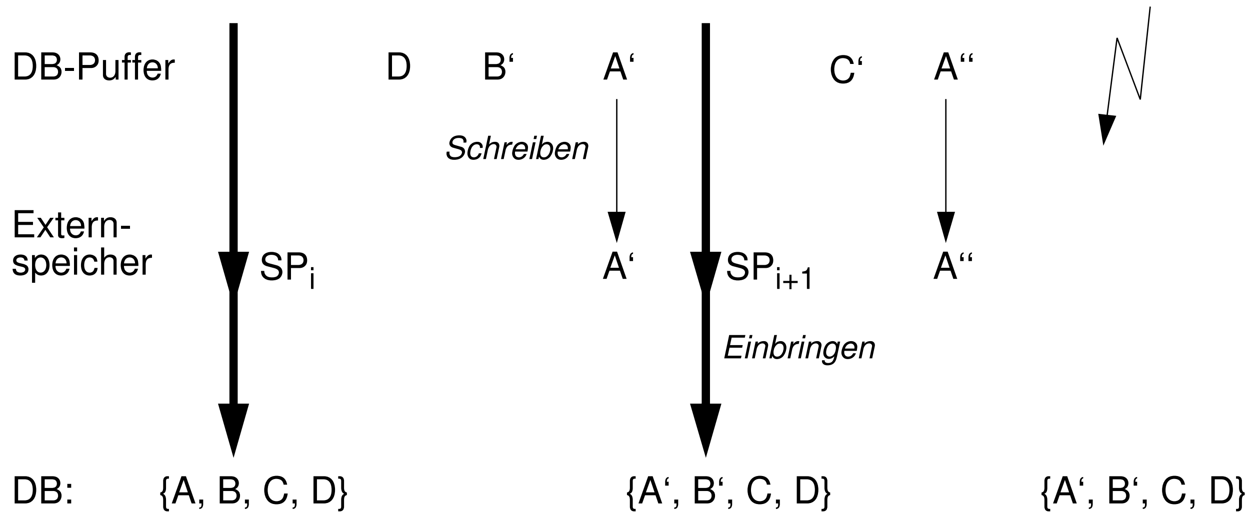


➥ DB nach Crash: ***chaos-konsistent*** !

- **verzögertes Einbringen:**

$$P_i \Rightarrow B_j$$
$$j \neq k$$
$$P_i' \Rightarrow B_k$$

➥ Schreiben ≠ Einbringen

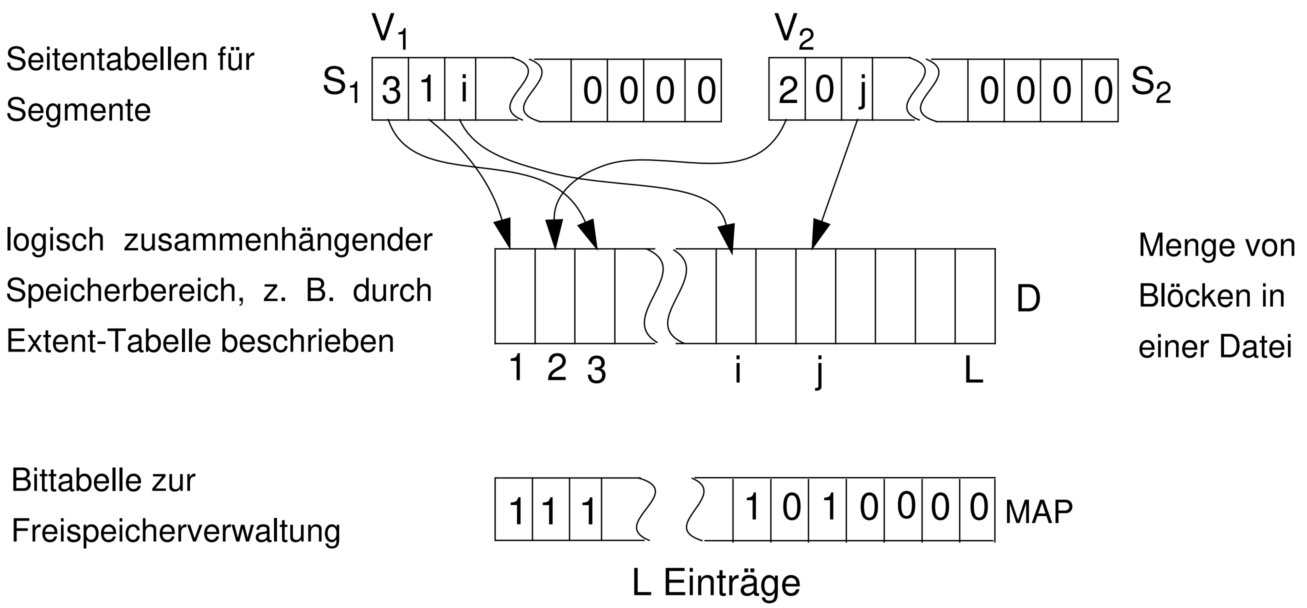


geeignete Wahl der Sicherungspunkte SP_i : Operationsgrenzen beachten !

➥ DB nach Crash: ***aktionskonsistent (operationskonsistent)*** !

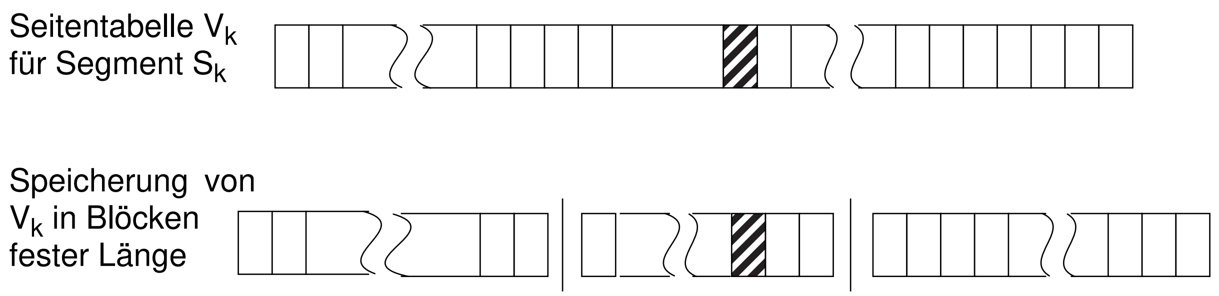
Speicherverwaltung beim Schattenspeicher-Verfahren

- **Prinzip der indirekten Seitenzuordnung**



➥ Abbildung von Seitennr. → Blocknr. mit Hilfe der Seitentabelle

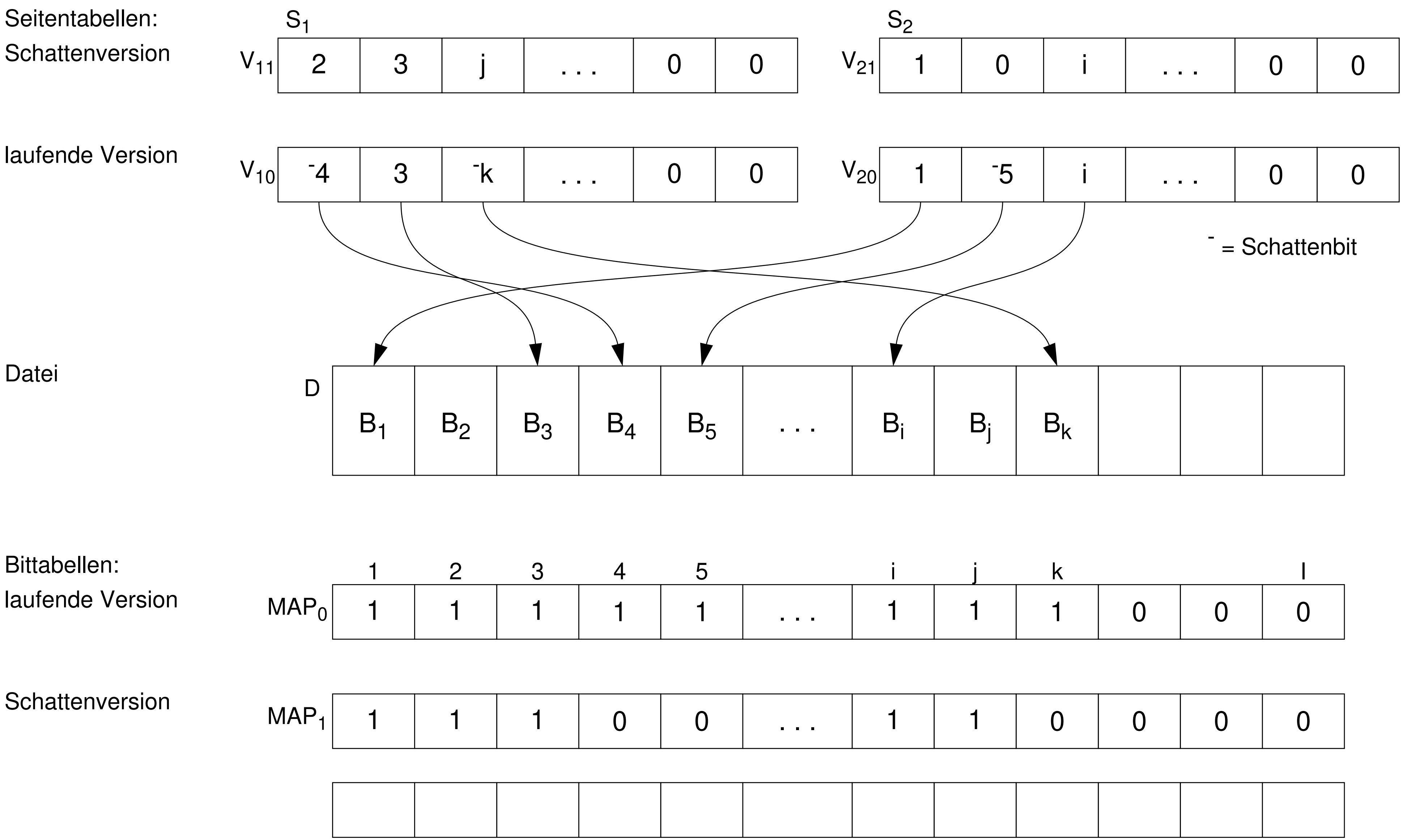
- **Zerlegung von Seitentabelle V_k in einzelne Blöcke**



➥ Auch die Seitentabelle muß im linearen Adreßraum untergebracht und auf Blöcke abgebildet werden.

Schattenspeicher-Verfahren – Prinzip

Verzögertes Einbringen von Änderungen

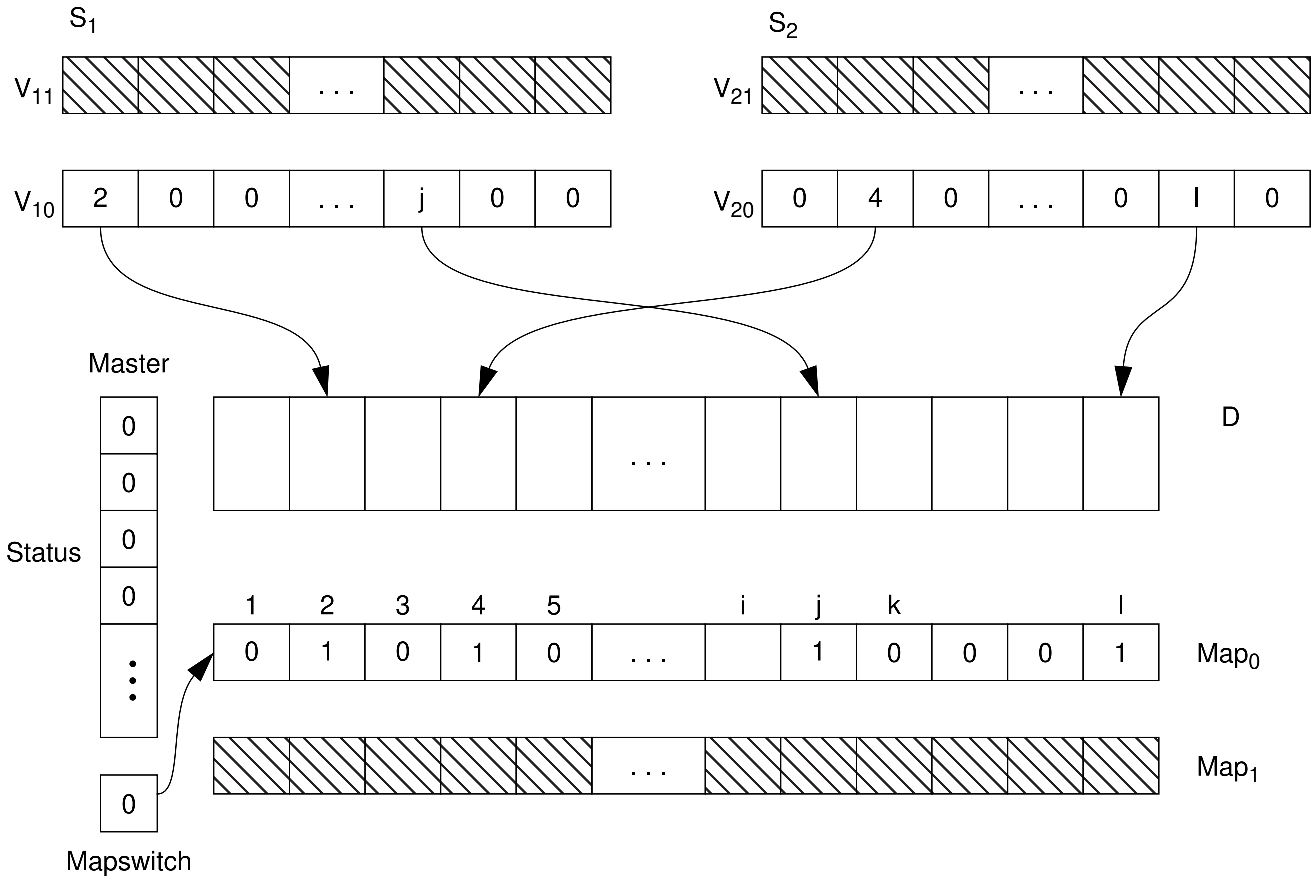


Schattenspeicher-Verfahren

- **Verwendung:**
  Implementierung fehlertoleranter Systeme

- **Erforderliche Speicherbereiche und Datenstrukturen**

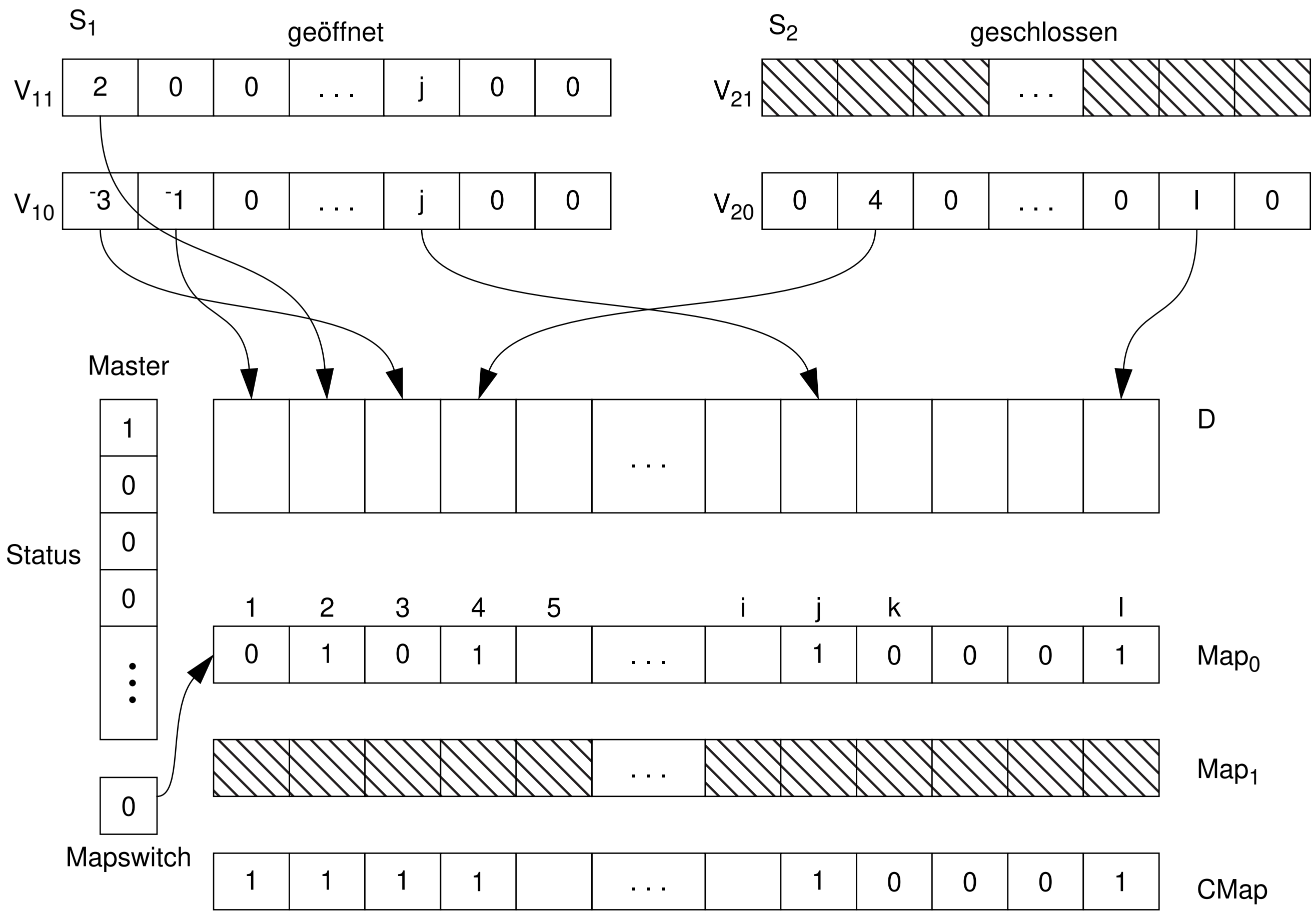
Die schraffierten Strukturen sind in der Ausgangssituation nicht benutzte Speicherbereiche, die erst nach der Eröffnung für Änderungsbetrieb erforderlich werden.



- Status (i) enthält den Eröffnungszustand für Segment i (hier: **alle Segmente geschlossen**).
- MAPSWITCH zeigt an, welcher der beiden (gleichberechtigten) Tabellen Map_0 und Map_1 das aktuelle Verzeichnis belegter Blöcke enthält.
- Wenn eine Segment geschlossen ist, erfüllt sein Inhalt bestimmte, von höheren Schichten kontrollierte Konsistenzbedingungen.

Schattenspeicher-Verfahren (2)

- **Änderungsbetrieb**



Segment S_1 geändert

- Auf einer Blockmenge D können gleichzeitig mehrere Segmente für den Änderungsbetrieb geöffnet sein.
- Map_0, Map_1 und CMAP beziehen sich auf die Blockmenge. CMAP enthält für alle Segmente die mit Schatten- bzw. aktuellen Seiten belegten Blöcke.
- Einer Seite wird nur bei der erstmaligen Änderung nach Eröffnen des Segmentes ein neuer Block zugewiesen.

Schattenspeicher-Verfahren – Funktionsprinzip

- Wenn **Segment k für Änderungen geöffnet** werden soll,
  sind folgende Schritte auszuführen:
  - kopiere V_{k0} nach V_{k1}
  - STATUS(k) := 1
  - Schreibe MASTER in einer ununterbrechbaren Operation aus
  - Lege im Hauptspeicher eine Arbeitskopie CMAP von MAP_0 an.

- Wenn eine **Seite P_i erstmalig seit Eröffnen des Segments geändert** werden soll, sind folgende Aktionen auszuführen:
  - Lies Seite P_i aus Block j = $V_{k0}(i)$
  - Finde einen freien Block j’ in CMAP
  - $V_{k0}(i)$ = j’
  - Markiere Seite P_i in $V_{k0}(i)$ als geändert
  - Bei weiteren Änderungen von P_i wird Block j’ verwendet.

- **Beenden eines Änderungsintervalls:**
  - Erzeuge die Bitliste mit der aktuellen Speicherbelegung in MAP_1
    (neue Blöcke belegt, alte freigegeben)
  - Schreibe MAP_1 (kein Überschreiben von MAP_0)
  - Schreibe V_{k0}
  - Schreibe alle geänderten Blöcke
  - STATUS(k) := 0, MAPSWITCH = 1 (MAP_1 ist aktuell)
  - Schreibe MASTER in einer ununterbrechbaren Operation aus.

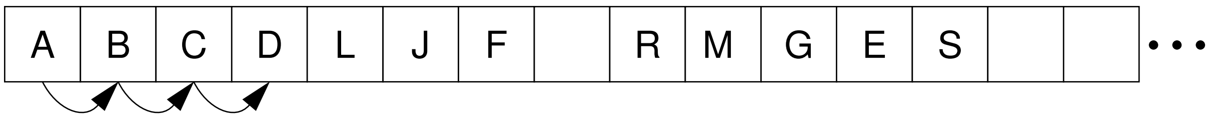
- Zum **Zurücksetzen geöffneter Segmente** muß lediglich V_{k1} in V_{k0} kopiert und STATUS(k) auf 0 gesetzt werden.

Schattenspeicher-Verfahren (3)

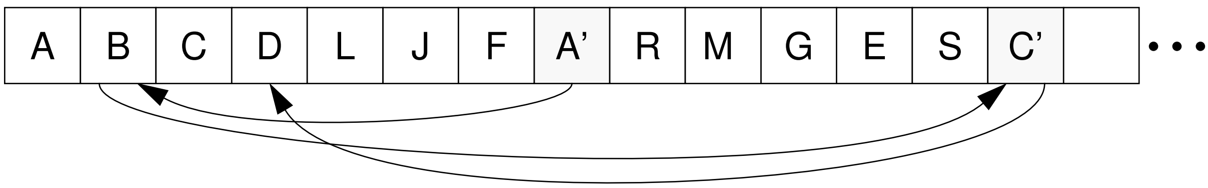
- **Erhaltung der physischen Clusterbildung**

Wegen der dynamischen Neuzuordnung von Blöcken können beim Schattenspeicherverfahren physische Nachbarschaften von logisch zusammengehörigen Seiten bei Änderungen i.a. nicht aufrechterhalten werden.

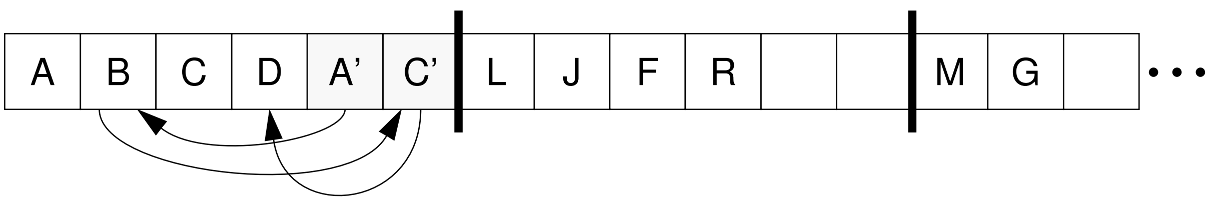
Bsp.: Die Seiten A, B, C, D mögen in allen Transaktionen in eben dieser Reihenfolge berührt werden:



Die Änderung von A → A’ und C → C’ führt zu:



Eine Verbesserung kann durch Einteilung der Datei in physische Cluster der Größe p und der Segmente in logische Cluster der Größe l sowie deren gegenseitige Zuordnung erreicht werden.
Beispiel mit p=6, l=4:



Bei Plattenspeichern wird ein physisches Cluster i. allg. ein Zylinder sein.

Bewertung des Schattenspeicher-Verfahrens

- **Vorteile**
  - Rücksetzen auf letzten konsistenten Zustand sehr einfach
  - flexiblere Schreibprotokolle für Log-Daten: Pufferung bis zum Umschalten auf einen neuen Zustand möglich
  - logisches Logging möglich, da stets operationskonsistenter Zustand verfügbar ist
  - bei katastrophalem Fehler ist Wahrscheinlichkeit höher, einen brauchbaren Zustand der DB zu rekonstruieren

- **Nachteile**
  - Hilfsstrukturen (MAP und Seitentabellen V_i) werden so groß, daß Blockzerlegung notwendig wird
  - Die Seitentabellen V_i belegen etwa 0.1-0.2% der DB-Größe, was bei großen DB's ($\geq$ n GB) zu einem hohen Anteil von Seitenfehlern im DB-Puffer für die Zugriffe auf V_i führen kann
  - periodische Sicherungspunkte erzwingen Ausschreiben des gesamten DB-Puffers
  - physische Clusterbildung logisch zusammengehöriger Seiten wird beeinträchtigt bzw. zerstört
  - zusätzlicher Speicherplatz für die Doppelbelegung; lange Batch-(Änderungs-)Programme werden dadurch schlecht unterstützt.

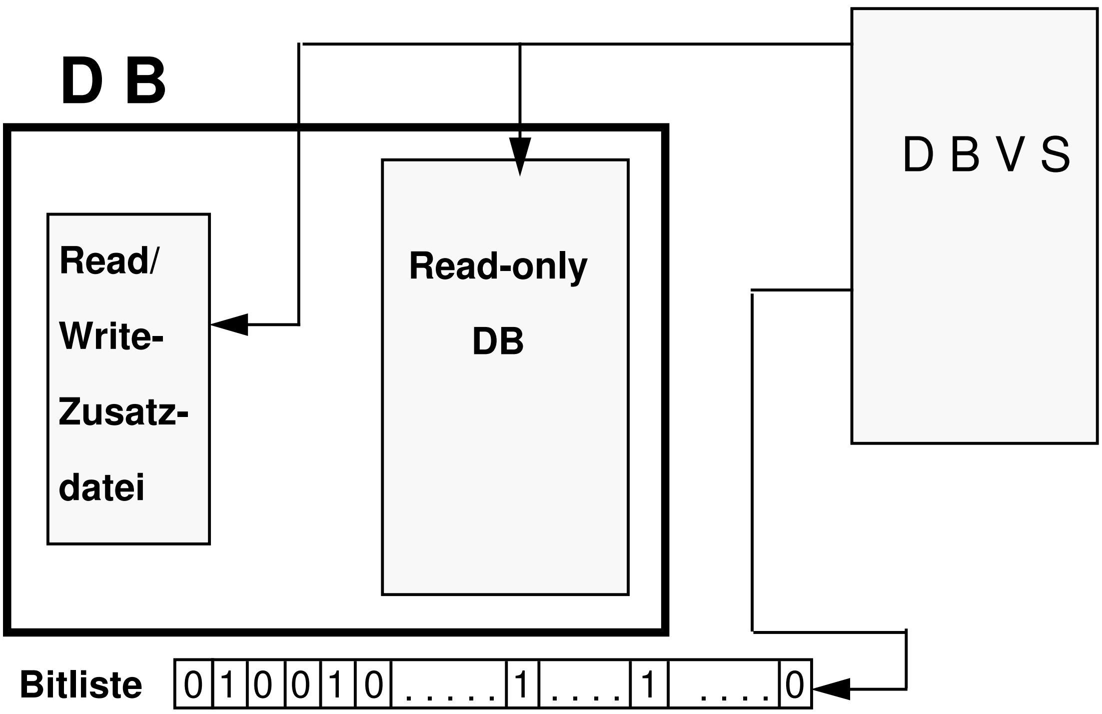
Das Zusatzdatei-Verfahren

- **Idee:**

Wenn eine Datei für Änderungsbetrieb geöffnet wird, wird eine zusätzliche, temporäre Datei angelegt, in welche während des Änderungsintervalles alle modifizierten Blöcke geschrieben werden. Die eigentliche Datei wird dabei nicht verändert. Am Ende des Änderungsintervalles werden alle veränderten Blöcke aus der temporären in die permanente Datei kopiert. Das Einbringen der Änderungen erfolgt also verzögert.

Das wesentliche Problem besteht darin, während des Änderungsbetriebes für eine gegebene Seiten-Nr. zu entscheiden, ob die aktuelle Version in der temporären oder permanenten Datei steht.

- **Prinzip:**

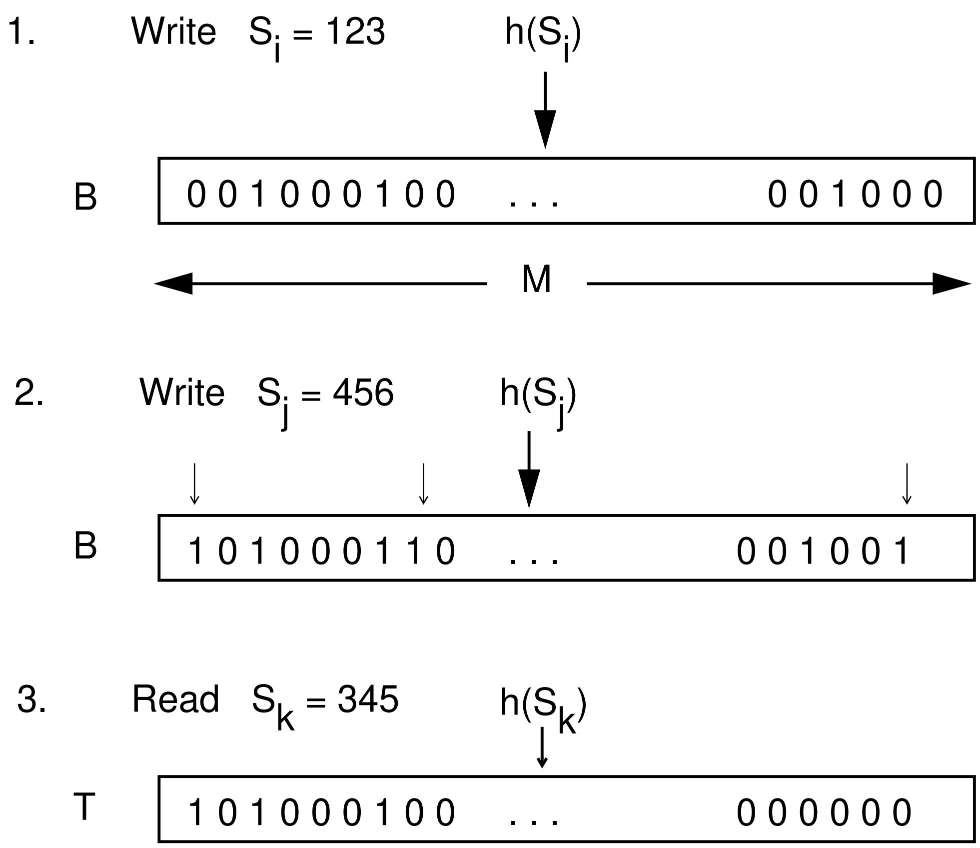


➥ Nutzung einer Bitliste, die anzeigt, ob eine Seite möglicherweise geändert wurde

➥ Hashabbildung erlaubt Begrenzung des Speicherplatzbedarfs

Bloom-Filter

- **Wirkungsweise**
  - Bitliste B der Länge M im Hauptspeicher
  - Signatur des Satzes in B bei seiner Änderung
  - Hash-Funktionen zur Satzabbildung adressieren x Bits, die in B auf 1 gesetzt werden
- **Aufsuchen von Satz S_i**
  - Erzeugen der charakteristischen x Bits in temporärer Bitliste T
  - **AND**-Operation von T und B in Erg
  - wenn alle x Bits in Erg gesetzt $\rightarrow$ VIELLEICHT
    sonst $\rightarrow$ NEIN
- **Beispiel:**



➥ Wenn (B **AND** T) = T,
dann Antwort VIELLEICHT !

Zusammenfassung

- **Speicherzuordnungsstrukturen erfordern effizientes Dateikonzept**
  - viele Dateien variierender, nicht statisch festgelegter Größe
  - Wachstum und Schrumpfung erforderlich
  - permanente und temporäre Dateien

- **Empfohlene Datei-Eigenschaften:**
  - direkter und sequentieller Blockzugriff
  - Blockgröße pro Datei definierbar
  - Blockzuordnung über dynamische Extents

  ➥ Dynamische Blockzuordnung (bei UNIX als mehrstufiger Baum) untauglich für große Dateien

- **Segmentkonzept**

  erlaubt die Realisierung zusätzlicher Attribute für die DB-Verarbeitung (Recovery, Clusterbildung für Relationen usw.)

- **Zweistufige Abbildung**
  - von Segment/Seite auf Datei/Block und diese auf Slots der Magnetplatte erlaubt Einführung von Abbildungsredundanz durch verzögertes Einbringen

- **Verzögerte Einbringstrategien**
  - sind teurer als direkte, besitzen jedoch implizite Fehlertoleranz
  - sie belasten den Normalbetrieb zugunsten der Recovery

- **Direkte Einbringstrategie (*update-in-place*)**
  - einfach zu implementieren
  - keine Zusatzkosten zur Ausführungszeit für die Seitenzuordnung
  - Fehlertoleranz nur durch explizite Logging- u. Recovery-Funktionen