Vorlesung Implementierung von Datenbanksystemen

3. Sätze

Prof. Dr. Klaus Meyer-Wegener

Wintersemester 2019/20

Lehrstuhl für Informatik 6 (Datenmanagement)
Friedrich-Alexander-Universität Erlangen-Nürnberg

3. Satzstrukturierte Dateien

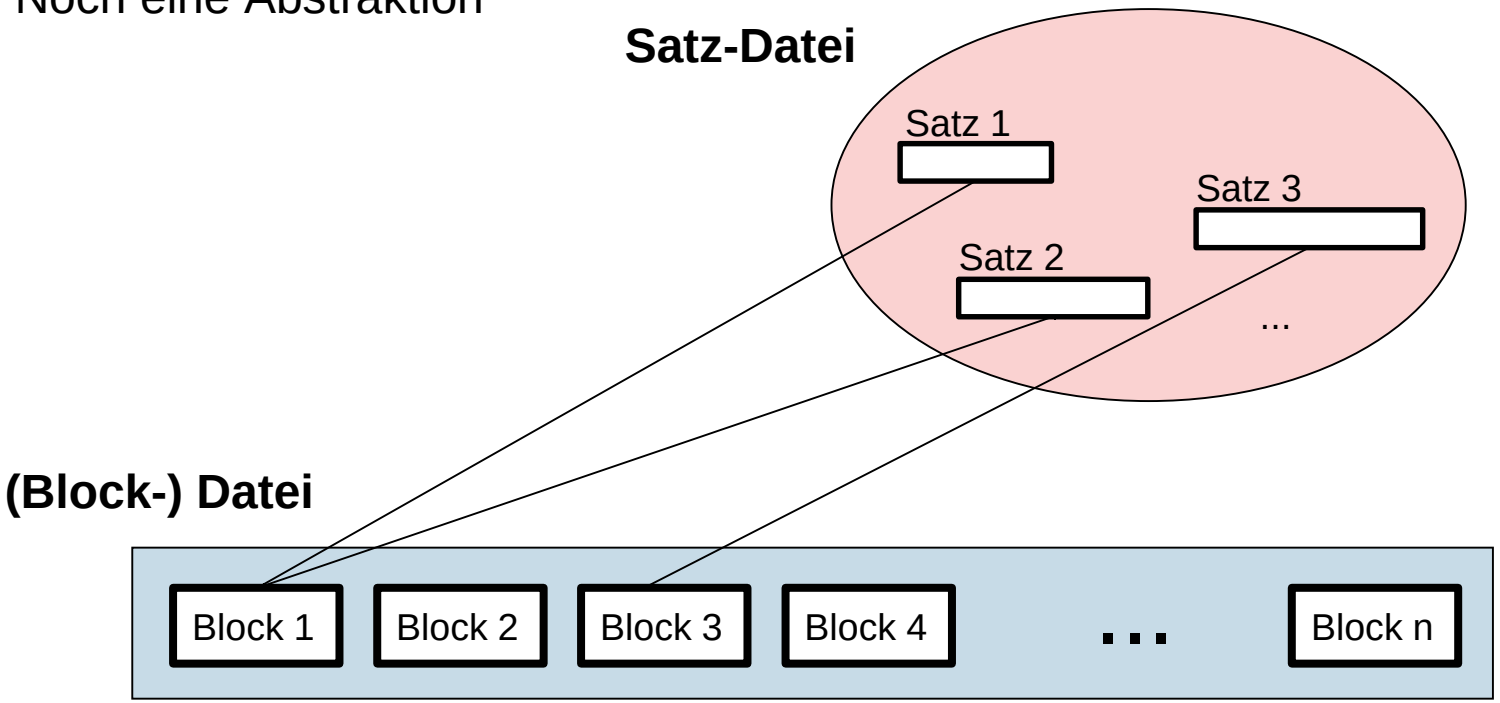
- **Block**
  - Einheit des Transports zwischen Hintergrund- und Hauptspeicher
- **Größe der Blöcke**
  - ist vom System vorgegeben (Formatierung des Hintergrundspeichers)
  - Typisch: 512 Bytes, 2K, 4K; manchmal in Grenzen wählbar
    - Werden tendenziell größer – dazu später mehr
  - **Kleine Blöcke**: viele E/A-Operationen
  - **Große Blöcke**: Mitlesen bzw. Mitschreiben uninteressanter Daten (die im selben Block abgelegt sind)
    oder Platzverschwendung (wenn man den Rest leer lässt)
- **Deshalb neue Abstraktion: Satz**
  - Neue Schicht oberhalb der Block-Dateien

Satz

- **Genau die Daten, die zu einem Gegenstand der Anwendung gehören**
  - Beispiele: Personenstammsatz, Teilestammsatz, Auftragssatz
- **Entscheidend: Größe (Länge) von der Anwendung bestimmt**
  - Und nicht (wie bei den Blöcken) vom System!
- **Insbesondere: variable Länge**
  - Von Satz zu Satz, und auch bei einem einzelnen Satz im Laufe der Zeit
  - Feste Länge als Spezialfall natürlich auch noch möglich
- **"Record" oder "Struktur" oder "Objekt" oder …**
  - In vielen Programmiersprachen
  - Dann aus Feldern zusammengesetzt (Bestandteile des Satzes); für Dateiverwaltung auf dieser Stufe aber noch irrelevant
- **Satz (wie Block) hier nur Folge von Bytes**
  - Keine weitere innere Struktur

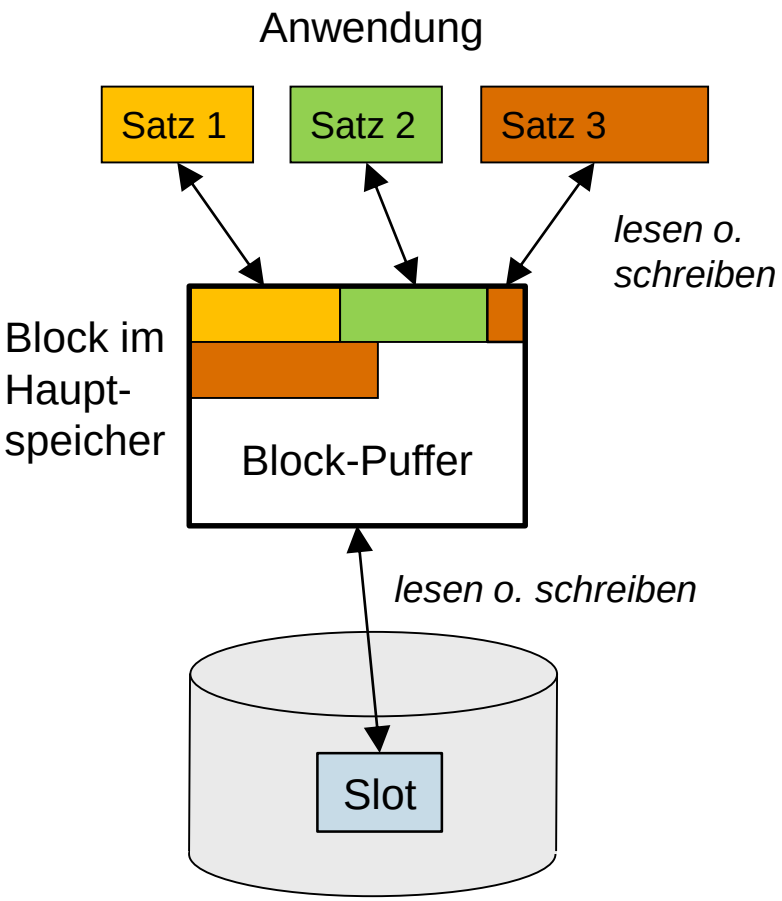
Satz-Datei

- **Menge (!) von Sätzen variabler Länge**
  - D.h. Reihenfolge unbestimmt
  - Noch eine Abstraktion



Sätze in Blöcken: Blockung

- **Typischerweise passen mehrere Sätze in einen Block.**
  - Satzlängen 100 – 1000 Bytes
  - Variable Anzahl von Sätzen pro Block
- **Vereinfachende Annahme:**
  - Keine blockübergreifenden Sätze ("spanned records")
  - D.h. jeder Satz vollständig in einem Block abgelegt
    - Ergänzung für längere Sätze aber kein großes Problem – siehe Übungen

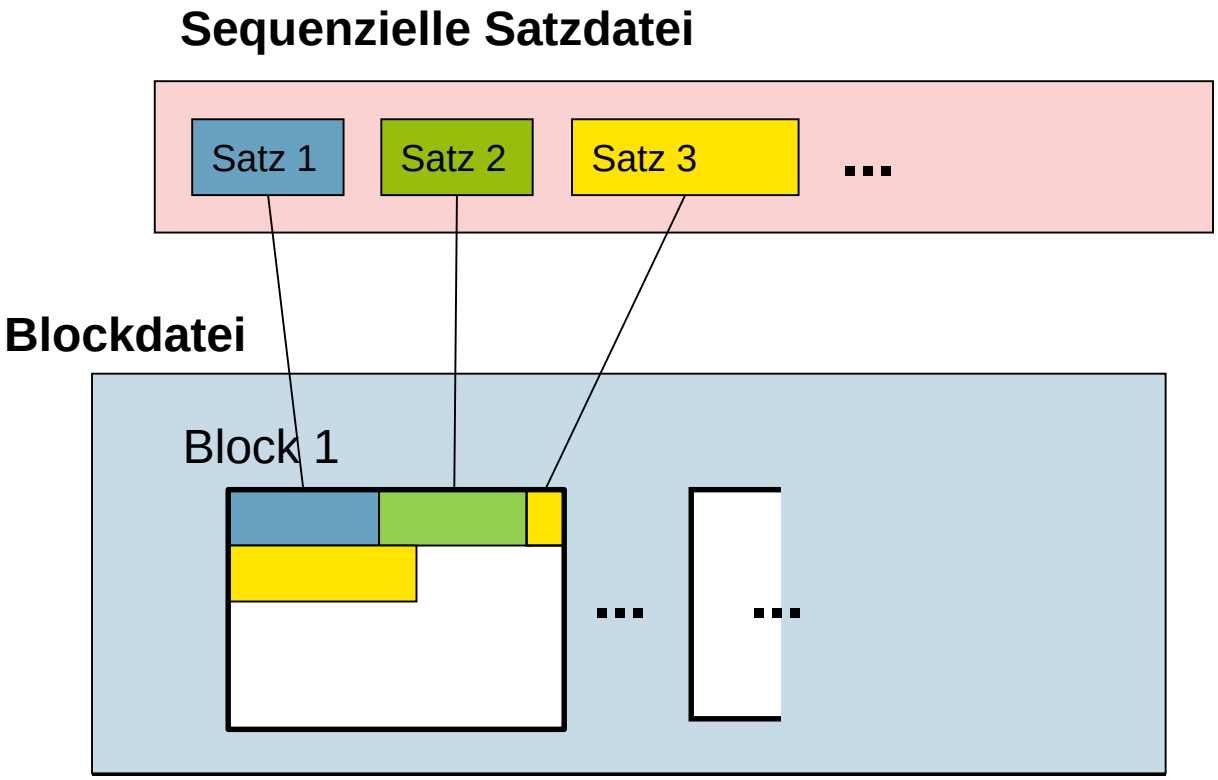


Blockung (2)

- **Feste Satzlänge:**
  - Als Sonderfall möglich – dann auch ausnutzen (kein Längenfeld)
  - Feste Zahl von Sätzen pro Block
    - Aus Blockgröße und Satzlänge errechenbar
  - Meist ungenutzter Speicherplatz am Ende eines Blocks
- **Variable Satzlänge:**
  - Zahl der Sätze ändert sich von Block zu Block
  - Erinnerung: Reihenfolge der Sätze beliebig (Menge!)
    - Man darf also z.B. umsortieren, bis nur noch wenig ungenutzter Speicherplatz am Ende eines Blocks übrig bleibt.
- **Was hier gewählt wurde ist dann eine Eigenschaft der Datei.**

3.1 Sequenzielle Satzdatei

- **Sonderfall: Folge (!) von Sätzen fester oder variabler Länge**



  - Wird nur sequenziell geschrieben und gelesen – einfacher

Sequenzielle Satzdatei (2)

- **Gibt es tatsächlich**
  - Z.B. als Zugriffsmethode SAM (Sequential Access Method)
  - oder ESDS (Entry-Sequenced Data Set) in der Zugriffsmethode VSAM (Virtual Sequential Access Method) der IBM (z/OS)
- **Schreibreihenfolge**
  - = Abspeicherungsreihenfolge
  - = Lesereihenfolge
  - Vom Benutzer definiert
    - Das System erhält sie, interpretiert sie aber nicht.
      - (Und kontrolliert sie deshalb auch nicht, z.B. Sortierung)

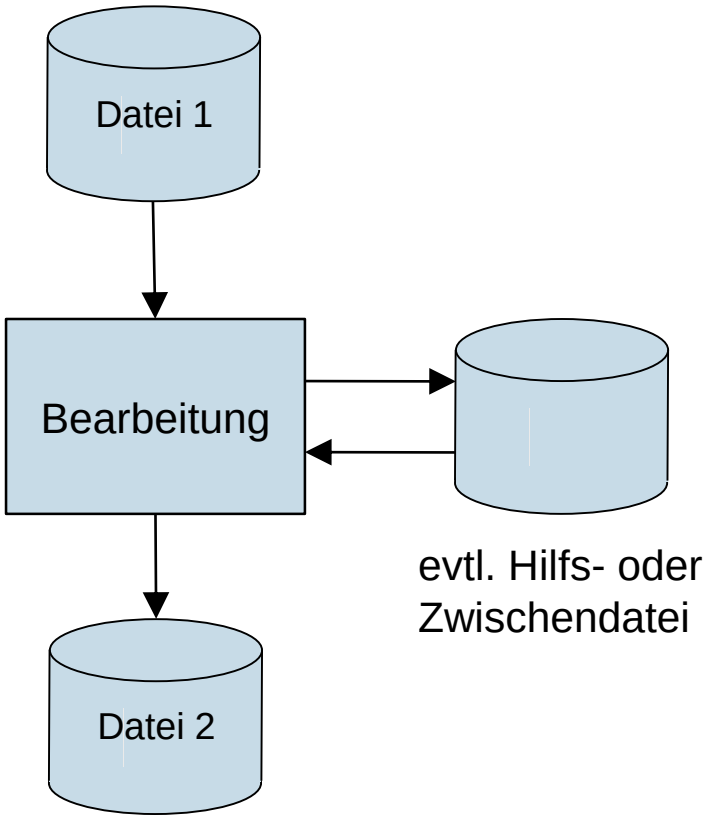
Sequenzielle Satzdatei (3)

- **Primäre Zugriffsarten:**
  - Sequenzielles Lesen (der ganzen Datei)
  - Sequenzielles (Über-) Schreiben der ganzen Datei
  - Sequenzielles Anhängen weiterer Sätze an eine Datei

- **Insbesondere werden *nicht* unterstützt:**
  - Wahlfreies Lesen eines bestimmten Satzes
  - Einfügen eines Satzes
  - Ändern eines Satzes
    - I.Allg. mit Längenänderung verbunden

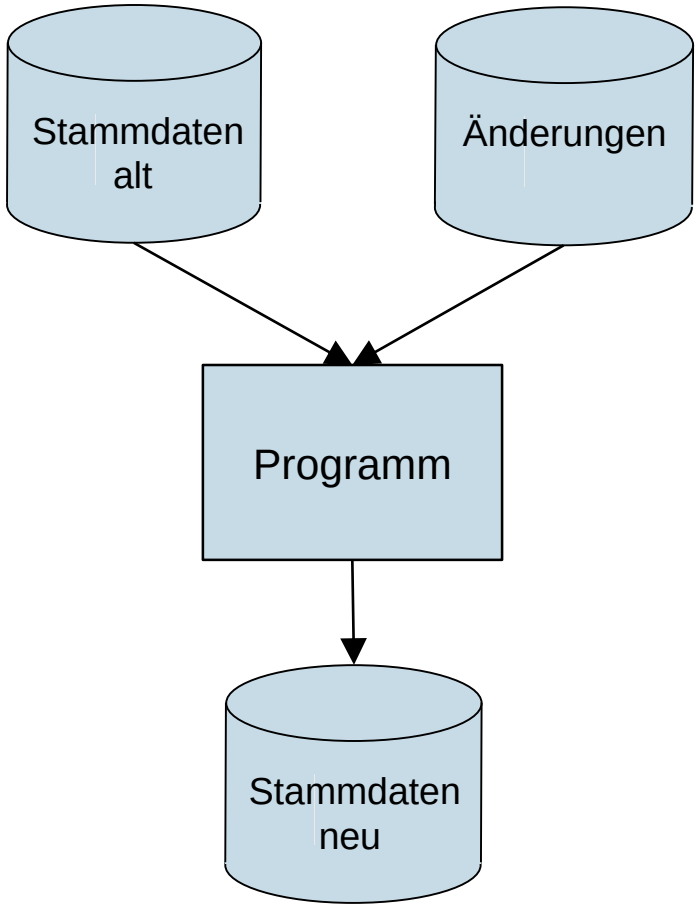
Stapelverarbeitung (Batch Processing)

- **Typische Bearbeitungsform für sequenzielle Dateien**
  - Insbesondere: **Sortierung**



Stapelverarbeitung (2)

z.B. Kontostand



z.B. Zu- und Abbuchungen

Zugriffsoperationen (Dienste)

- **Datei öffnen:**

```
SeqRecordFile::SeqRecordFile (
   char *FileName,
   char Mode,
   int *RecLength )
```

  - Dateiname je nach Betriebssystem
  - Modus gibt an, wie die Datei geöffnet werden soll
    - Zulässige Werte: nur Lesen ("r"), Schreiben ("w"), Anhängen ("a")
  - Satzlänge
    - beim lesenden Zugriff Ausgabeparameter
    - beim schreibenden Zugriff Eingabeparameter,
      0 heißt: variable Satzlänge

Zugriffsoperationen (2)

- **Nächsten Satz lesen:**

```
void SeqRecordFile::read-next (
   int *RecordLength,
   char *RecordBuffer )
```

  - Satzlänge als Eingabeparameter: maximale Länge,
    als Ausgabeparameter: tatsächliche Länge
    - Verhindert unbeabsichtigtes Überschreiben von Teilen des Arbeitsspeichers, wenn erwartete Länge < tatsächliche Länge
      - "Buffer Overflow"
  - Statt Satzpuffer kann auch **Zeiger** auf Satzanfang zurückgeliefert werden
    - Vorteil: kein Kopieren
    - Nachteil: Zeiger nur bis zum nächsten **read-next** benutzbar
  - Dateikontrollblock enthält auch **Leseposition** in der Datei
    - Wird jedes Mal weitergerückt, so dass nachfolgendes **read-next** den nächsten Satz liest (Dienstprotokoll)

Zugriffsoperationen (3)

- **Nächsten Satz schreiben:**

```
void SeqRecordFile::write-next (
   char *RecordBuffer,
   int RecordLength )
```

  - Nur beim Öffnungsmodus "Schreiben" oder "Anhängen" erlaubt
  - Schreibt den Satz an das Ende der Datei
  - Wurde die Datei mit fester Satzlänge geöffnet, so wird die Angabe einer Satzlänge ignoriert
  - Mögliche Fehler:
    - Satz zu lang
    - Datei war nur zum Lesen geöffnet
    - Kein Platz mehr auf dem Hintergrundspeicher
- **Schließen der Datei:**

```
SeqRecordFile::~SeqRecordFile ()
```

Blockunabhängigkeit

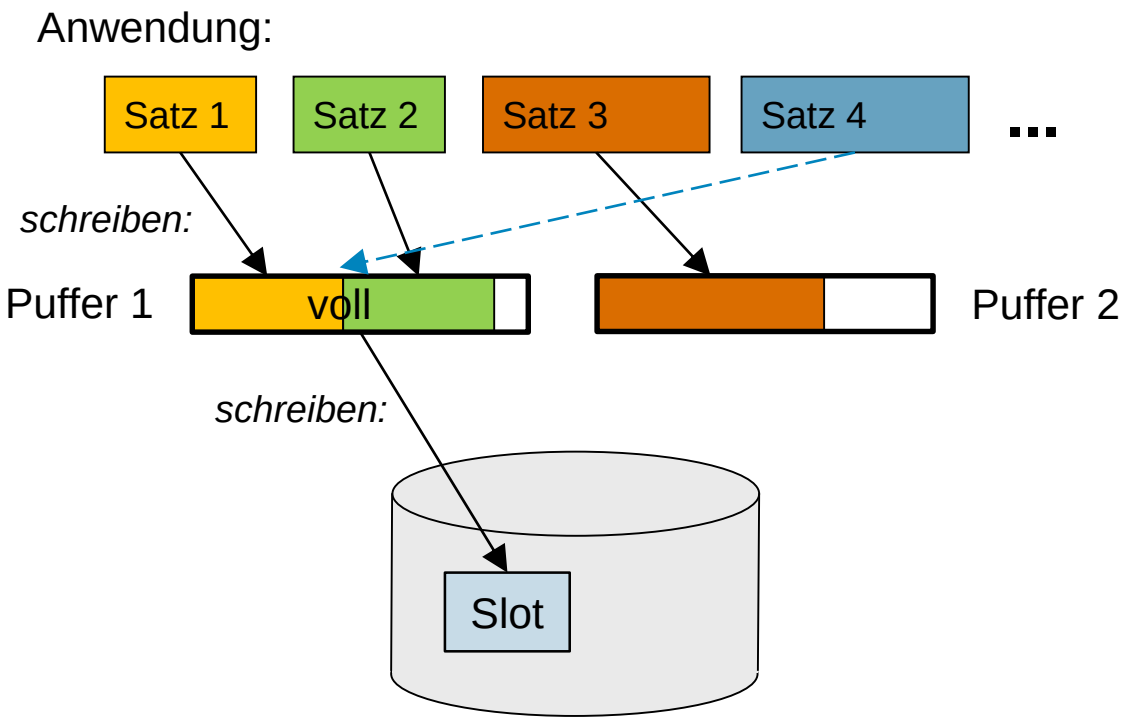
- **Anwendungsprogramm, das mit sequenzieller Datei arbeitet,**
  - kennt (und benutzt) die Einheit des physischen Transports zwischen Speichergerät und Hauptspeicher (= Block) nicht mehr.
    - Also kann diese verändert werden,
      ohne dass das Programm betroffen ist.
  - kann ebenso gut mit einem Magnetband arbeiten!
  - braucht nur zwei Operationen zu kennen
    - Deshalb z.B. beim Dateikonzept der Programmiersprache Pascal verwendet
  - kann einige Schreib- oder Leseoperationen ohne "physische" Ein-/Ausgabe ausführen (*n* – 1 bei *n* Sätzen im Block)
- **Nachteile:**
  - Kein Direktzugriff auf Sätze
  - Beim Schreiben werden immer nur einzelne Blöcke hinzugefügt (oder eine feste Zahl) – Gefahr der Zerstückelung der Datei!

Puffer

- **Block**
  - ist weiterhin die Einheit des physischen Transfers zwischen Hintergrund- und Hauptspeicher
- **Satz lesen:**
  - Immer einen ganzen Block vom Hintergrundspeicher lesen – enthält meist außer dem gewünschten noch andere Sätze
- **Satz schreiben:**
  - Block erst dann auf den Hintergrundspeicher schreiben, wenn er nach mehreren Satz-Schreiboperationen ganz mit Sätzen gefüllt ist
- **Sätze werden also gepuffert**
- **Dateipuffer**
  - Hauptspeicherbereich, der (mindestens) einen Block der Datei aufnehmen kann, damit Sätze herausgeholt oder hineingepackt werden können ("Blockpuffer")

Wechselpuffertechnik

- **Optimierungsmöglichkeit bei sequenziellen Dateien: Verwendung von *zwei* Blockpuffern**



Wechselpuffertechnik (2)

- **Beim Schreiben:**
  - Wenn Puffer 1 voll, **asynchron** auf Hintergrundspeicher schreiben und gleichzeitig Puffer 2 zur Aufnahme weiterer Sätze verwenden
  - Ist auch Puffer 2 voll, ebenfalls asynchron schreiben und mit Puffer 1 fortfahren
- **Beim Lesen analog:**
  - Während Sätze in Puffer 1 nach und nach gelesen werden, *gleichzeitig* schon nächsten Block vom Hintergrundspeicher in Puffer 2 lesen
    - Wird auch "Prefetching" genannt
  - Der ist dann (idealerweise) bereits da, wenn sein erster Satz gebraucht wird.
  - Übernächsten Block wieder in Puffer 1 lesen usw.
- **Rein interne Optimierung:**
  - *Unsichtbar* für Programmierer (bzw. höhere Schichten)
  - Möglich geworden durch "Kapselung" des Blocktransfers

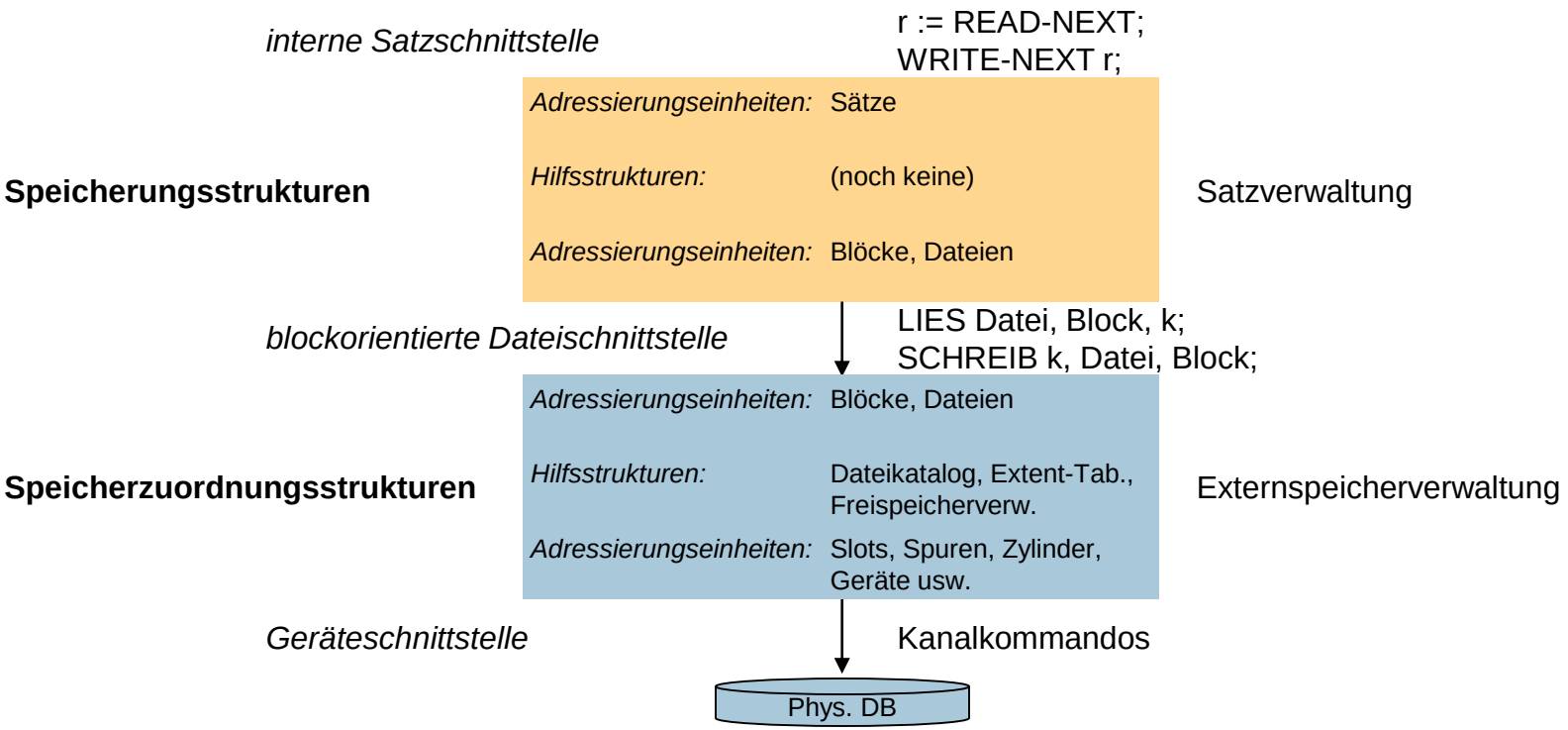
Allgemeiner Dateipuffer

- **Verallgemeinerung:**
  - Nicht nur zwei Blockpuffer, sondern beliebig viele
  - Dann auch **Pufferrahmen** – engl. buffer frames – oder **Kacheln** genannt
  - Möglich wegen zunehmend größerer Hauptspeicher
- **Beim Lesen**
  - die nächsten *n* Blöcke auf einmal lesen (Prefetching)
- **Beim Schreiben**
  - mehrere Blöcke füllen, dann auf einmal ausschreiben
- **Noch günstiger als Wechselpuffertechnik:**
  - Spart Zugriffsarmbewegungen
  - Größerer Nutzen vor allem bei nicht-sequenziellen Dateien (siehe unten)
  - Aber: Persistenz wird (zunächst) ein Problem – nach Schreiben eines Satzes steht der erst einmal nur im (flüchtigen) Puffer …

Sequenzielle Satzdatei – Fazit

- **So primitiv sie auch sein mag**
- **Zeigt sie doch schon einiges an Kapselung und Optimierungspotenzial:**
  - **Blockunabhängigkeit:**
    - Blöcke als Einheit des Transfers zwischen Hintergrund- und Hauptspeicher sind nicht mehr sichtbar;
    - könnten also durch Reorganisation (Formatierung) verändert werden.
  - **Pufferung:**
    - Mehr Speicherplatz bereitstellen für eine bessere Leistung
    - Blöcke mit Sätzen füllen oder Sätze aus Blöcken holen und *gleichzeitig* andere Blöcke auf den Hintergrundspeicher schreiben bzw. von ihm lesen
    - Völlig unsichtbar für Benutzer der Schnittstelle, rein interne Optimierung

Resultierende Schicht



3.2 Direktzugriff auf Sätze

- **Flexiblerer Ansatz als bei sequenziellen Dateien gesucht:**
  - Auch Einfügen, Löschen und Ändern *einzelner Sätze* soll möglich sein!
- **Generelle Frage:**
  - In welchem Block einen neuen Satz **ablegen**,
    und wie später diesen Satz **wiederfinden**,
    auch wenn zwischenzeitlich etliche andere Sätze
    gelöscht und eingefügt wurden?
- **Annahmen:**
  - Variable Satzlänge (allgemeiner Fall)
  - Reihenfolge der Abspeicherung
    muss nicht mehr Reihenfolge des Einfügens sein.
    - Information über Einfügereihenfolge darf verlorengehen.
    - Datei ist jetzt wirklich eine *Menge* von Sätzen.
  - Direkter Zugriff auf einzelne Sätze über ihre **Satzadresse**

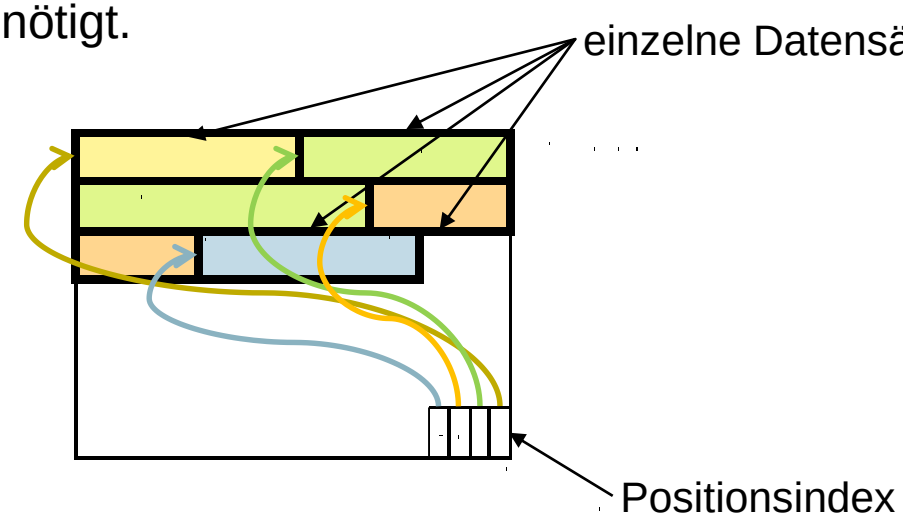
Satzadresse

- **Beim Einfügen eines Satzes vom DBVS zugeteilt ("Handle")**
- **Später zum Zugriff auf denselben Satz verwenden**

- **Zwei grundlegende Eigenschaften:**
  - **eindeutig**
  - **unveränderlich**
    - Auch bei beliebigen Einfügungen, Änderungen und Löschungen anderer Sätze
    - Auch dann, wenn der Satz verschoben werden muss, innerhalb des Blocks oder in anderen Block
      - Z.B. weil er länger geworden ist

TID-Konzept

- **Verbreitetste Realisierung einer Satzadresse**
- **Satzadressierung über Indirektion innerhalb der Blöcke**
  - Hilfsstruktur: Feld (Array) mit Anfangsadressen (Byte-Positionen) aller Sätze dieses Blocks (sog. "Positionsindex")
  - Satzadresse ist ein Paar aus Blocknummer und Index in diesem Feld
  - ("**TID** = Tuple IDentifier")
  - Für den Zugriff auf einen Satz wird nur ein Blockzugriff benötigt.
  - Struktur eines Blocks:

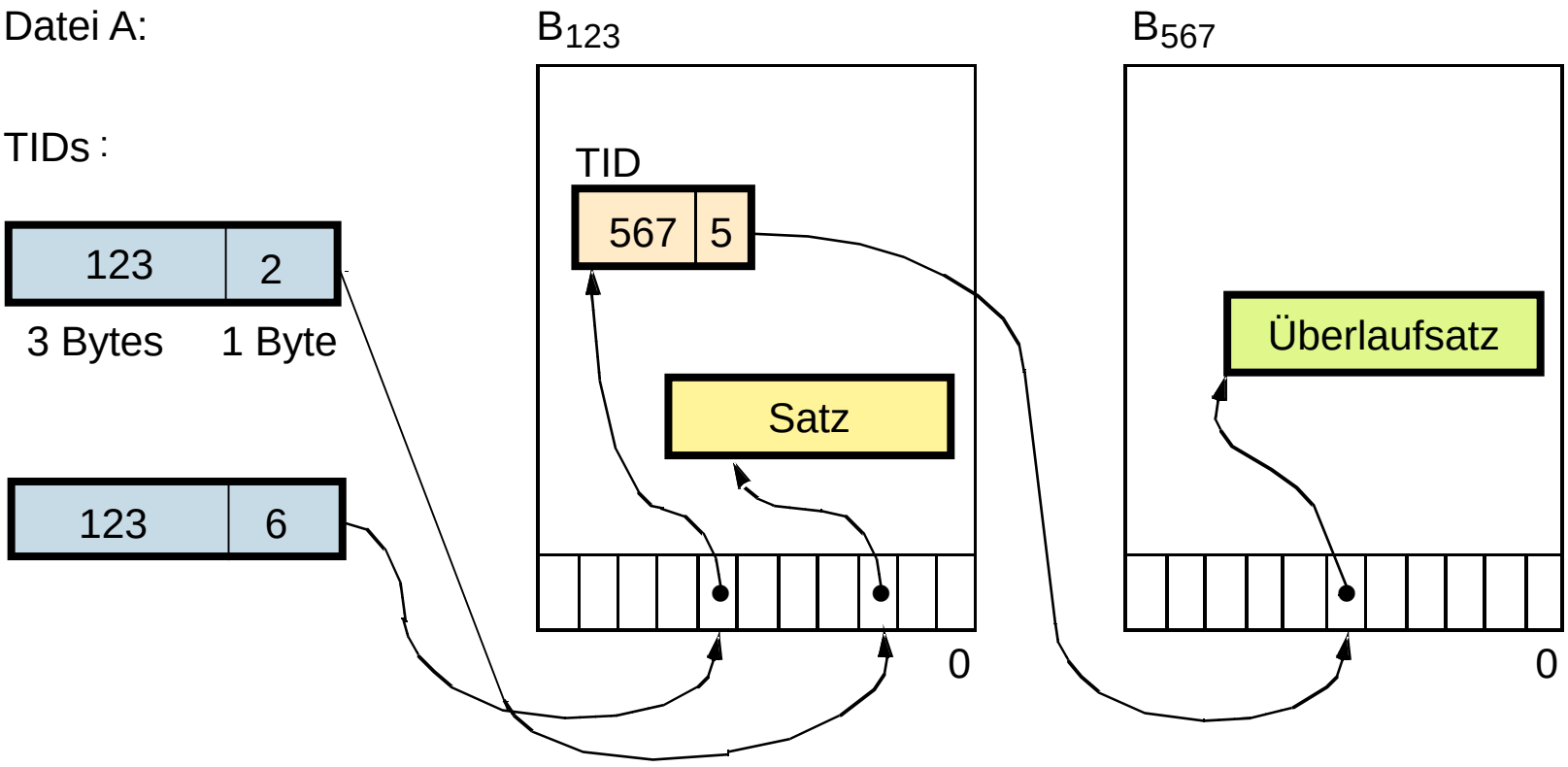


TID-Konzept (2)

- **Löschen eines Satzes**
  - Der entsprechende Eintrag im Positionsindex wird als ungültig gekennzeichnet.
  - Alle anderen Sätze im selben Block können verschoben werden, um den freien Platz zu maximieren –
    es ändern sich nur ihre Anfangsadressen im Positionsindex.
  - Alle Satzadressen bleiben stabil.
- **Änderungsoperation auf einem Satz**
  - Es kann sich die Länge des Satzes ändern !!!
  - Satz schrumpft oder wird größer (ohne Überlauf):
    - Alle Sätze werden innerhalb des Blocks verschoben und der Positionsindex wird angepasst.
  - Satz wird größer und der freie Platz im Block reicht nicht mehr für die Speicherung des jetzt größeren Satzes (Überlauf):
    - Verschieben des Satzes in einen anderen Block!

Überlaufbehandlung beim TID-Konzept

- **Verschieben eines Satzes**



Überlaufbehandlung beim TID-Konzept (2)

- **Vorgehen**
  - Im alten Block verbleibt an der Stelle des Satzes eine neue Satzadresse, die auf den neuen Block verweist.
  - In diesem (seltenen) Fall müssen also zwei Blöcke gelesen werden, um einen Satz über seine TID zu finden.
  - Wird der Satz ein weiteres Mal verlagert, so wird die (Weiterleitungs-) Satzadresse im ersten Block verändert. Dadurch bleibt es bei maximal einer Indirektion.
- **Trick**
  - Die Länge der Überlaufkette ist also immer kleiner oder gleich 1, d.h. ein Überlaufsatz darf nicht weiter "überlaufen", sondern wird von seiner Hausadresse aus neu platziert.
- **Vorteile**
  - Ein Satz kann innerhalb eines Blocks und über Blockgrenzen hinweg verschoben werden, ohne dass die TID sich ändert.

3.3 Direkte Satzdateien

- **Mit Hilfe der Satzadressen (TID's)**
  sind einzelne Sätze beliebig zugreifbar, änderbar und löschbar
- **Zugriffsoperationen:**

```
DirectRecordFile::DirectRecordFile ( ... )
```

  - Wie gehabt, nur ist im Schreibmodus jetzt auch Lesen möglich

```
RecordAddress DirectRecordFile::insert
   ( char *RecordBuffer, int RecordLength )
void DirectRecordFile::read ( RecordAddress Address,
   char *RecordBuffer, int *RecordLength )
void DirectRecordFile::modify ( RecordAddress Address,
   char *RecordBuffer, int RecordLength )
void DirectRecordFile::delete ( RecordAddress Address )
```

  - (Bei NoSQL-Systemen auch "CRUD" genannt: create, read, update, delete)

- **Z.B. in jedem Block am Anfang**
  - Zahl der freien Bytes
  - Suche bei **insert** (und evtl. **modify**) dann aber zu mühsam: alle Blöcke abklappern
- **Alternative:**
  - *k* Blöcke am Anfang der Datei mit Feld "Freispeichertabelle" (neuer Blocktyp)
  - in diesen Blöcken (hinter dem Header) nur Einträge des Typs vorzeichenlose Festpunktzahl, für leere Blöcke mit (Blockgröße – Header-Größe) initialisiert
  - Eintrag *i* der Freispeichertabelle stimmt mit Eintrag "freie Bytes" im Header des *i*-ten Blocks überein (falls es den überhaupt noch gibt)
  - Suche nach Block mit freiem Platz viel schneller
  - Aber: beim Speichern und Löschen immer auch die Freispeichertabelle ändern – mindestens zwei Blöcke ändern (und zurückschreiben)
  - Erweitern der Datei um neue Blöcke schwieriger (warum?)

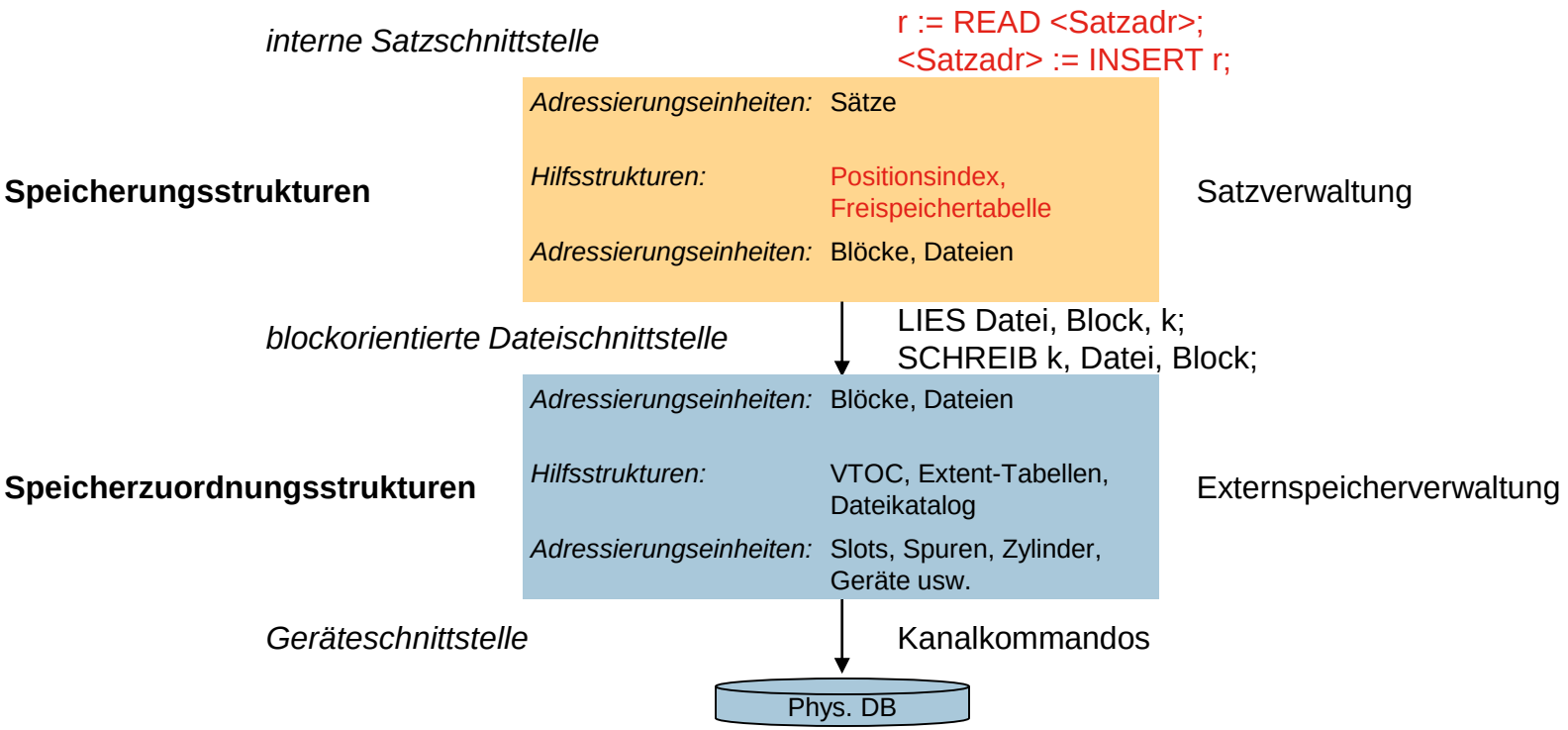
Weitere Operationen

- **Sequenzielles Lesen aller Sätze auch möglich:**

```
void DirectRecordFile::read-first
   ( char *RecordBuffer, int *RecordLength )
void DirectRecordFile::read-next
   ( char *RecordBuffer, int *RecordLength )
```

- **Aber:**
  - System entscheidet über Abspeicherungsreihenfolge, nicht Benutzer!
  - Man bekommt alle Sätze – in irgendeiner Reihenfolge (heute so, morgen evtl. anders).
  - System verwaltet die Menge der Sätze und keine Folge
- **Primär eben: Einzelsatzverarbeitung**
  - Erster Schritt von Stapelverarbeitung zu interaktiver Verarbeitung

Resultierende Schicht



Anhang

- **Nicht mehr verwendete Folien**
- **Zur Vertiefung einzelner Aspekte oder zum Nachschlagen bei Bedarf**

3.A.1 Satzadresse

- **Erste Idee: laufende Nummer des Satzes**
  - Kein zusätzlicher Speicherplatzbedarf
  - Zugriff auf Satz *i* durch Abzählen von vorn (direkter Zugriff???)
  - *Instabil!*
    ändert sich bei Einfügungen und Löschungen
    sowie bei Änderungen in der Abspeicherungsreihenfolge
- **Zweite Idee: Blocknummer und Byte-Position (Offset) im Block**
  - Kein zusätzlicher Speicherplatzbedarf
  - Direktzugriff auf Satz
  - *Instabil!*
    ändert Satz seine Länge, müssen i.Allg. andere Sätze im Block verschoben werden; wird Satz zu lang für Block, so muss er in anderen Block verlegt werden

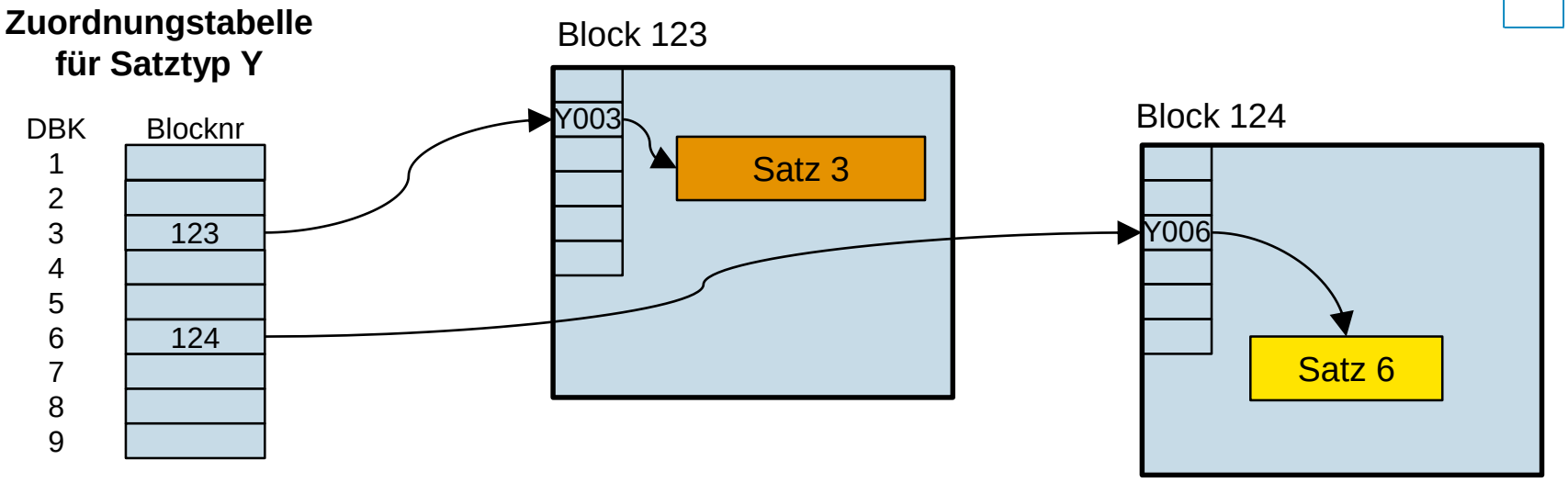
Zuordnungstabelle

- **Indirektion**
- **Verwaltung eines Feldes (in den ersten *k* Blöcken der Datei), das zu jeder Satznummer (Feldindex) Blocknummer und Byte-Position enthält**
  - Bei Einfügungen grundsätzlich neue Satznummer vergeben
  - Bei Löschungen Eintrag der entsprechenden Satznummer als ungültig kennzeichnen
  - **Stabil**: bei Verschiebungen im Block oder Verlagerungen in anderen Block nur Eintrag des Satzes im Feld ändern; Satznummer bleibt unverändert – Satz also über dieselbe Satznummer weiterhin auffindbar
  - Zugriff auf Satz erfordert nun zwei Blockzugriffe:
    einen für Feld, einen für Block mit dem Satz selbst
  - Bei Verschieben eines Satzes muss auch Eintrag im Feld geändert und wieder auf Hintergrundspeicher geschrieben werden (Persistenz) – zusätzliche E/A-Operation
  - Feld selbst beansprucht Speicherplatz
  - Auch bekannt als **Database Key Translation Table** (DBTT)

Zuordnungstabelle (2)

- **Optimierung:**
  möglichst keine Änderung der Zuordnungstabelle,
  wenn Satz innerhalb eines Blocks verschoben wird
  - Zusätzlich Speicherplatz im Block bereitstellen
    für kleine Zuordnungstabelle der Sätze in diesem Block
    - Enthält Satznummer und Anfangsadresse jedes Satzes
    - Ähnlich wie bei TID
  - In der großen Zuordnungstabelle dann nur noch Blocknummer
  - Wenn Satz innerhalb eines Blocks verschoben wird:
    - Neue Anfangsadresse in die kleine Zuordnungstabelle in diesem Block eintragen
    - Nur ein Block geändert, nur ein Block zurückzuschreiben auf Hintergrundspeicher

Zuordnungstabelle (3)



- **Database Key (DBK) ist "nicht sprechende" Adresse (wie Telefonnummer)**
  - wird gebildet aus Satztypbezeichnung r und Folgenummer f;
  - identifiziert Satz während seiner Lebenszeit in der Datei.
- **Problem: Wo wird die Zuordnungstabelle abgespeichert,**
  - Am Anfang? Wie erweitern?
  - Am Ende? Wie den Datenbereich davor erweitern?
  - In einer eigenen Datei?

3.A.2 Verkettung von Sätzen

- **Wie und wo bewahrt man Satzadressen auf?**
- **Z.B. in anderen Sätzen**
  - Satzadressen als Verweise (Referenzen, Zeiger, Pointer) auf andere Sätze verwenden
- **Einfachstes Beispiel: lineare Liste der Sätze**
  - Next-Pointer
  - Anwenderdefinierte Ordnung nach irgendeinem Kriterium – unabhängig von der Speicherungsreihenfolge
- **Aber z.B. auch: Gruppenbildung**
  - Sätze mit bestimmtem Merkmal verketten (z.B. alle Autos eines Typs)
  - Evtl. mehrdimensional: ein Satz in diversen Ketten
- **Einfügen und Löschen**
  - wie bei verketteten Listen sonst auch

Verkettung von Sätzen (2)

- **kombiniert benutzerdefinierte Ordnung (unabhängig von Speicherungsreihenfolge) mit der Möglichkeit des Direktzugriffs auf Sätze**
  - Sehr flexibler Mechanismus
  - Beliebige Beziehungen der Anwendungswelt darstellbar
  - "Navigation" in den Satzmengen
  - Auch zwischen Sätzen verschiedener Dateien

Verkettung von Sätzen (3)

- **Beispiele:**