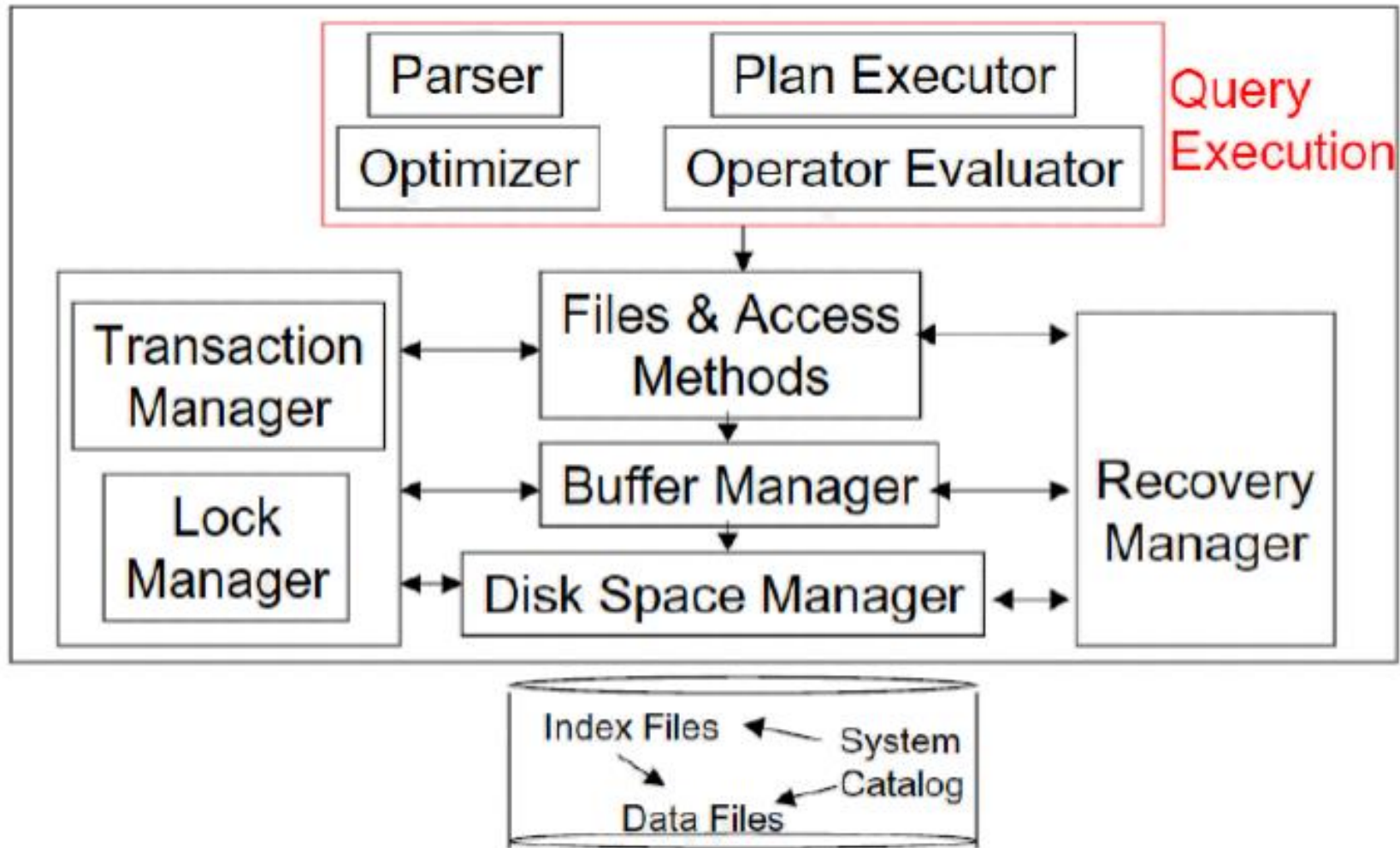


Physische Datenorganisation

Detaillierte Struktur eines DBMS



Speichermedien

- Man unterscheidet meist drei Stufen von Speichermedien:
 - **Primärspeicher/Hauptspeicher:** sehr teuer, sehr schnell, eher klein (im Vergleich zur benötigten Datenmenge)
 - Es ist möglich auf beliebige Adressen direkt zuzugreifen → die Granularität des Hauptspeichers ist sehr fein
 - **Sekundärspeicher:** Festplatte – Zugriff zu Daten ist langsamer, bietet mehr Platz, günstiger
 - Ein direkter Zugriff ist möglich, aber mit einer gröberen Granularität
 - Die kleinste Einheit auf eine Festplatte ist ein **Block**
 - In DBMS wird meistens als kleinste Einheit eine **Seite** verwendet
 - **Archivspeicher/Tertiärspeicher**

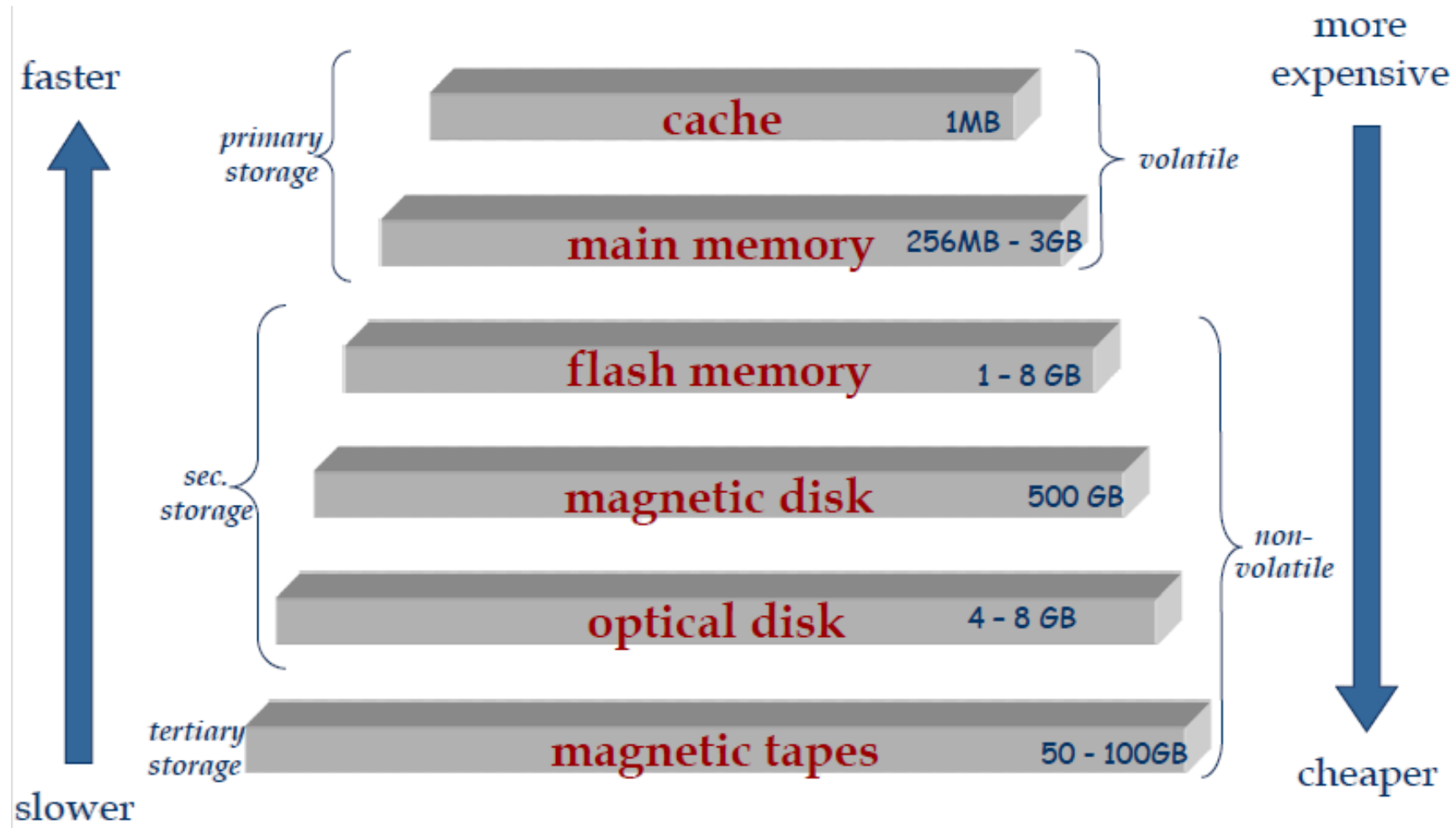
Platten und Dateien

- DBMS speichert Informationen auf externen Datenträgern: Magnetplatten (Hard Disks), Bänder (tapes)
- Zwei Hauptoperationen:
 - READ: Datentransfer von Platte (Sekundärspeicher) in den Hauptspeicher (RAM)
 - WRITE: Datentransfer vom RAM auf Platte
- Beides sind Operationen mit hohen Kosten im Vergleich zu in-memory Operationen und müssen deshalb sorgfältig geplant werden
 - Bedeutende Auswirkungen auf den Entwurf eines DBMS

Warum nicht alles im Primärspeicher behalten?

- Problem:
 - zu hohe Kosten
 - Primärspeicher (RAM) ist flüchtig/volatile - wir möchten Daten zwischen einzelnen Programmläufen speichern → wir brauchen persistente Daten
- Flüchtig/volatile = der Inhalt des Speichers steht nur zur Laufzeit des Systems zur Verfügung
- Typische Speicherhierarchie:
 - Primärspeicher – für die Daten, die an einem bestimmten Moment benutzt werden
 - **Cache** für sehr schnellen Datenzugriff
 - **Main Memory (RAM)** für häufig genutzte Daten
 - Sekundärspeicher – Disks für die aktuelle Datenbank
 - Flash Memory (SSD)
 - Festplatte – auf Magnetscheiben gespeichert
 - Optische Datenträger (CD-ROM, DVD, Blu-ray)
 - Archivspeicher/Band – für die Archivierung älterer Versionen der Datenbank oder Sicherheitskopien für Recovery-Zwecke

Speicherhierarchie

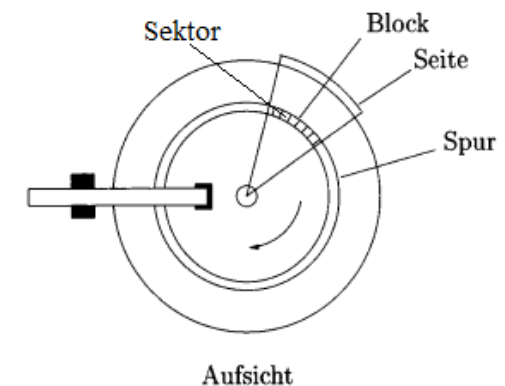
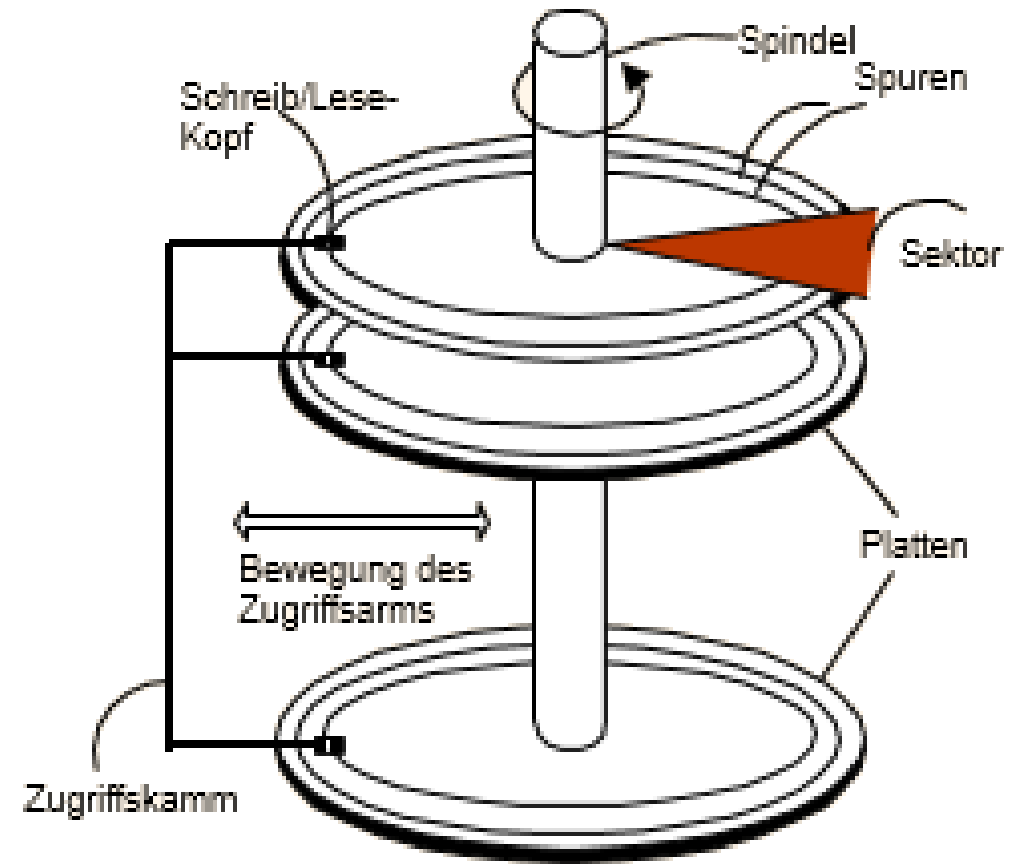


Platten

- Bevorzugtes Speichermedium für Sekundärspeicher
- Hauptvorteil gegenüber Bändern: Direktzugriff (random access)
- Daten werden gespeichert und gelesen in Einheiten, genannt Blöcke oder Seiten/Pages
- Blöcke werden in konzentrischen Ringen angeordnet = Spuren (Tracks)
- Spuren können auf beiden Seiten der Platten angelegt werden
- Im Gegensatz zum RAM hängt die Zeit zum Auffinden eines Blocks vom Speicherort auf der Platte ab
 - die relative Platzierung von Blöcke auf der Platte hat einen großen Einfluss auf die Leistung des DBMS

Komponenten einer Platte

- Plattenumdrehung (ca. 90/Sek)
- Zugriffskamm wird hinein- oder hinausbewegt, um den Schreib/Lesekopf auf die gewünschte Spur zu positionieren
- Zylinder = alle übereinander angeordneten Spuren von Plattenoberflächen
- Nur ein Schreib/Lesekopf arbeitet zu einem bestimmten Zeitpunkt
- Blockgröße ist das Vielfache einer Sektorgröße (welche fest ist)



Zugriff auf eine Seite

- Addressierung von Blöcken über Zylinder-, Spur- und Sektor-Nr
- Zugriffskamm muss über entsprechenden Zylinder geführt werden, auf den Sektor warten, der irgendwann vorbeirotiert und schließlich den Blockinhalt lesen und übertragen
- Zugriffszeit (read/write) auf einen Block:
 - Zugriffsbewegungszeit (seek time) – Bewegung der Zugriffsarme, um den Schreib/Lesekopf auf der richtigen Spur zu positionieren
 - Umdrehungswartezeit (latency time, rotational delay) – Warten bis der gesuchte Block am Kopf vorbeirotiert
 - Übertragungszeit (transfer time) – tatsächlicher Transport der Daten von/zu der Plattenoberfläche

Zugriff auf eine Seite

- Seek time und Umdrehungswartezeit dominieren
 - Seek time: zwischen 1 und 20 ms
 - Umdrehungswartezeit: zwischen 0 und 10 ms
 - Übertragungszeit – 1 ms/4 KB Seite
- Reduzierung der I/O Kosten heißt Seek time und Umdrehungswartezeit verringern!
- Hardware- vs. Software-Lösungen

Anordnung von Seiten auf einer Platte

- „Next“ Block – Konzept
 - Blöcke auf der gleichen Spur, gefolgt von
 - Blöcken auf dem gleichen Zylinder, gefolgt von
 - Blöcken auf benachbartem Zylinder
- Blöcke sollten in einer Datei sequentiell angeordnet sein (mit „next“), um die Zugriffsbewegungszeit und die Umdrehungswartezeit zu minimieren
- Prefetching – derzeit noch nicht (aber vielleicht in Kürze) benötigte Seiten werden gleich mit gelesen
- Es gilt: Sequentiellen Lese (Scan) ist immer viel schneller als wahlfreier Zugriff (Random Access)

RAID (Redundant Array of Independent Disks)

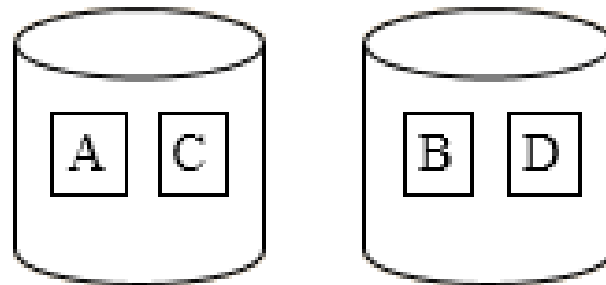
- Disk Array: Anordnung mehrerer Platten, die in Abstraktion als einzige große Platte angesehen werden können (Steuerung erfolgt durch Controller) – es ist billiger mehrere kleinere Disks zu benutzen
- Ziele:
 - Steigerung der Performanz durch Parallelität des Zugriffs
 - Verfügbarkeit (Ausfallsicherheit, Zuverlässigkeit)
- Zwei Haupttechniken:
 - Data Striping
 - Redundanz

RAID

- Data Striping:
 - Daten werden partitioniert
 - Die Größe einer Partition heißt Striping Unit
 - Verteilung der Daten: bitweise, byteweise oder blockweise
 - Partitionen werden über mehrere Platten verteilt (meist nach Round-Robin-Algorithmus)
- Redundanz: Mehrere Platten → mehrere Fehler
 - Redundante Informationen erlauben Wiederherstellung der Daten im Fehlerfall
 - Entweder Duplikate halten oder nur Kontrollinfos (z.B. Paritätsbits)

RAID Levels – Level 0

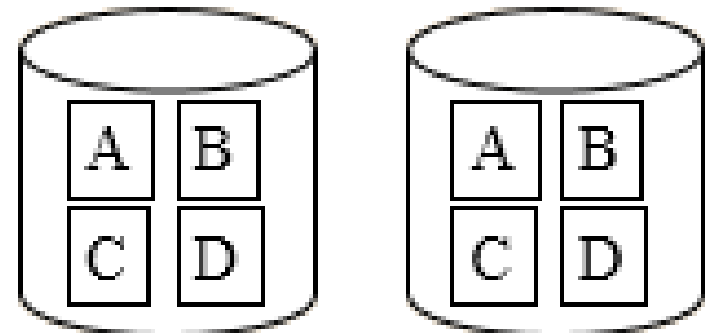
- Level 0 – keine Redundanz
 - Datenblöcke werden im Rotationsprinzip auf physische Platten verteilt
 - Striping mit **Block-Granularität**
 - Beste Write-Performance aller RAID-Levels
 - Zuverlässigkeit sinkt linear mit der Anzahl der Platten



Blöcke: A, B, C, D
2 Platten

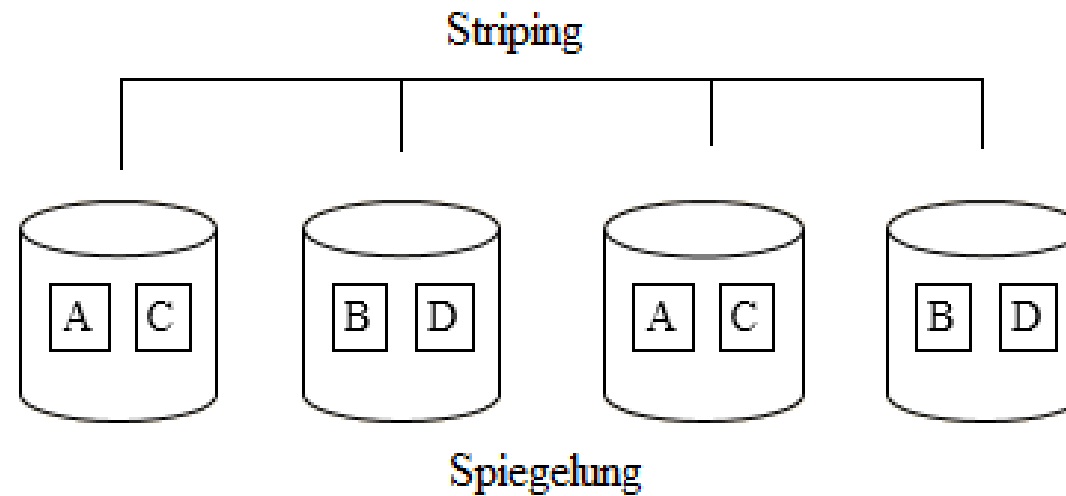
RAID Levels – Level 1

- Level 1 – Spiegelung
 - Zwei identische Kopien
 - Jede Platte hat eine Spiegelplatte (Mirror image)
 - Teuerste Lösung: jedes Write muss auf beiden Platten erfolgen
 - Parallele Reads auf verschiedenen Platten möglich
 - Maximale Datentransfer-Rate = Transfer-Rate einer Platte
 - 50%-ige Nutzung des Plattenspeichers



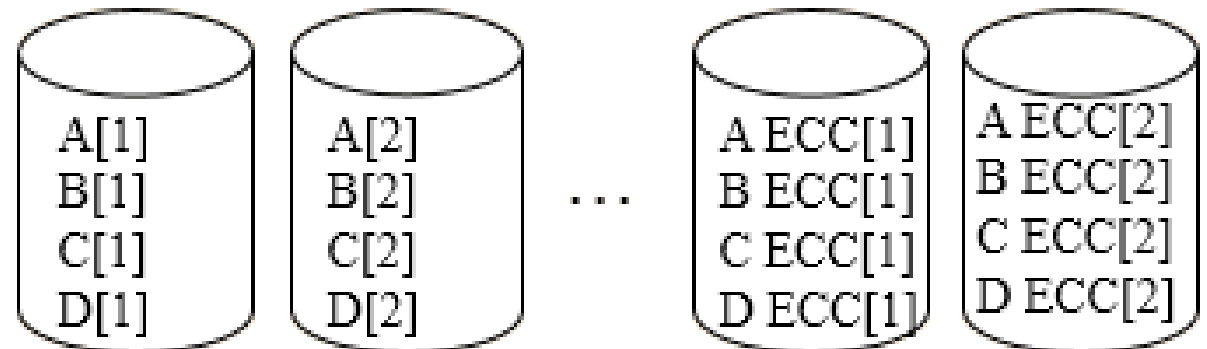
RAID Levels – Level 0+1

- Level 0+1 – Striping und Spiegelung
 - Kombination aus Level 0 (Striping) und Level 1 (Spiegelung)
 - Parallele Read möglich
 - Write-Kosten analog zu Level 1 (beide Platten involviert)
 - Maximale Transfer-Rate = Aggregierte Bandbreite aller Platten



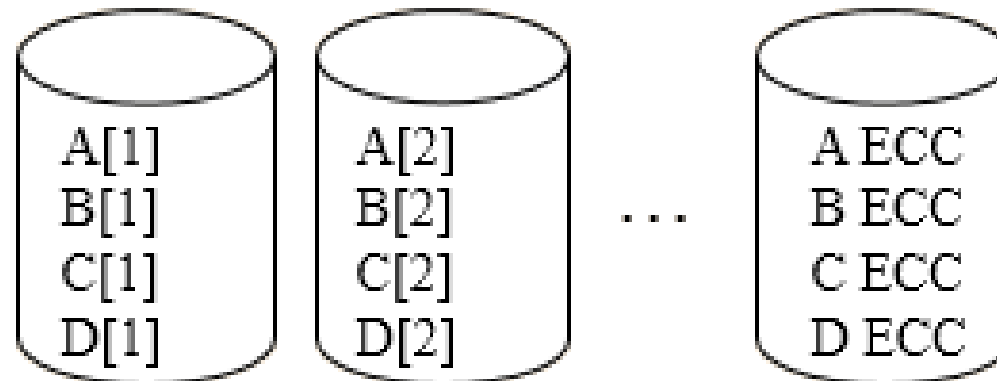
RAID Levels – Level 2

- Level 2 – Error-Correcting Codes
 - Striping auf **Bitebene**
 - Speicherung zusätzlicher Paritätsbits oder Error Correcting Codes (ECC) auf zusätzlichen Platten
 - Regel: 1.Bit auf 1.Platte, ..., i. Bit auf Platte $(i \bmod n)$; Kontrollinfos auf zusätzlichen Platten
 - Read erfordert 8 parallele Lese-Operationen (für 1 Byte)
 - Zugriff auf einzelne Daten nicht schneller, aber höherer Gesamtdurchsatz (high data transfer rates)
 - Wird in der Praxis nicht benutzt



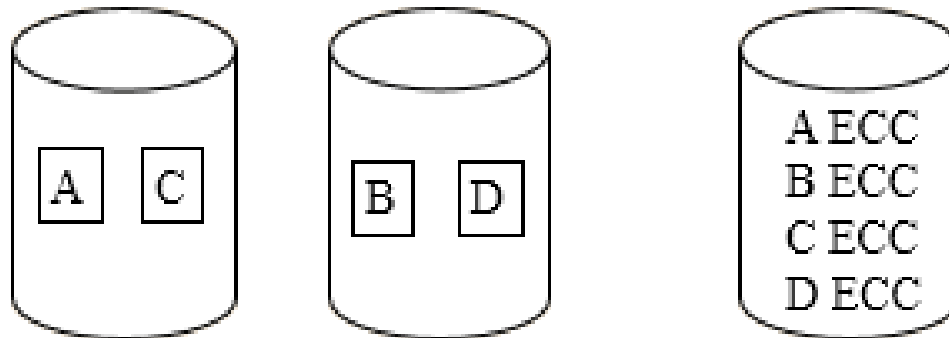
RAID Levels – Level 3

- Level 3 – Bit-Interleaved Parity
 - Striping auf **Bitebene**
 - Nur **eine Check Disk** (ein Paritätsbit ausreichend) → platzsparender als Level2
 - Read und Write erfordern alle Platten
 - Disk Array kann einen Request gleichzeitig verarbeiten
 - Zugriff auf einzelne Daten nicht schneller, aber höher Gesamtdurchsatz



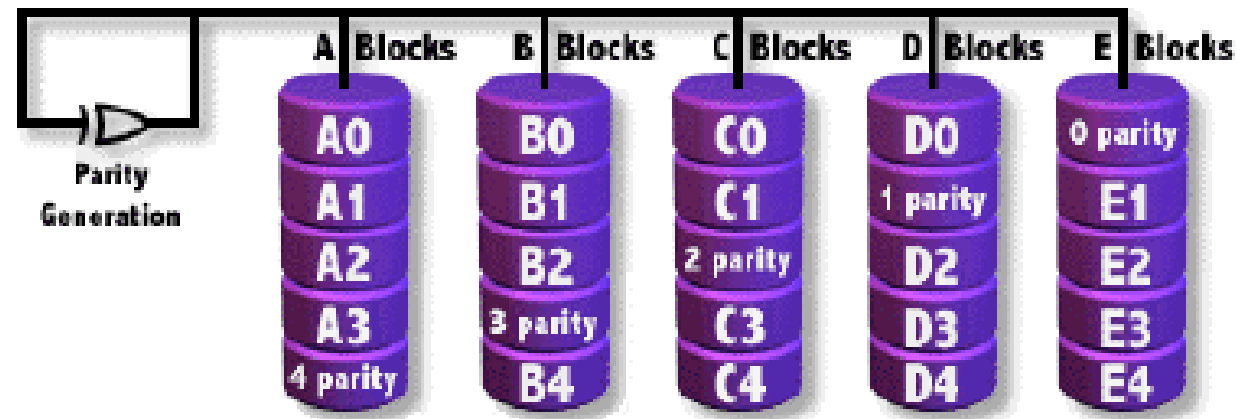
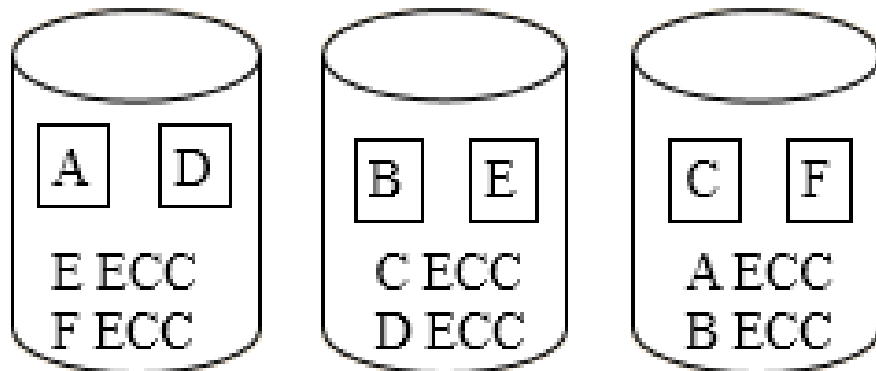
RAID Levels – Level 4

- Level 4 – Block-Interleaved Parity
 - Striping auf **Blockebene**
 - Nur **eine Check Disk** zur Aufnahme des Paritätsbits
 - Kleine Datenmengen können effizienter gelesen werden, da nur eine physische Platte betroffen ist
 - Große Read-Requests können die volle Bandbreite des Disk-Array nutzen
 - Write eines einzelnen Blocks betrifft 2 Platten: Data und Check Disk



RAID Levels – Level 5

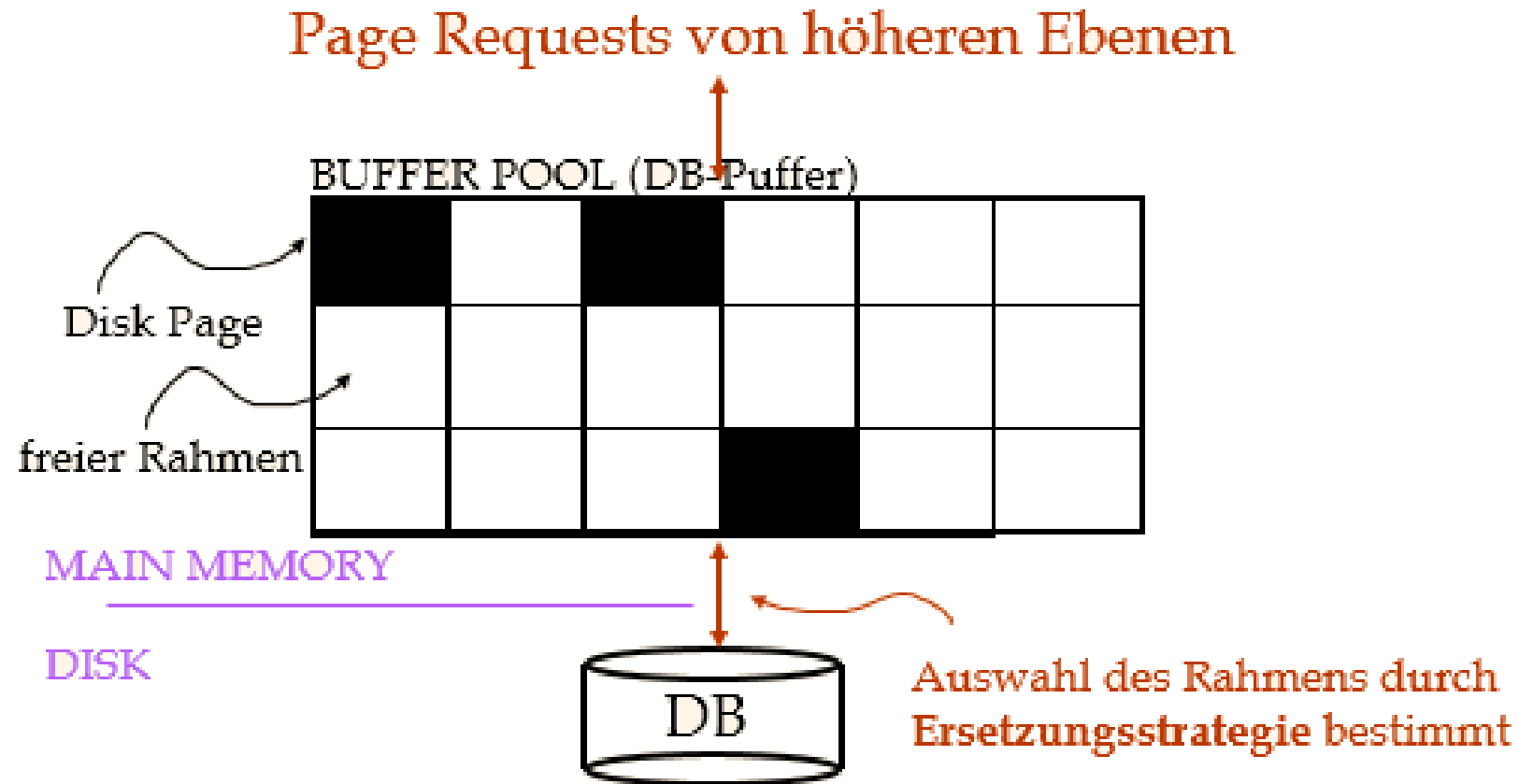
- Level 5 – Block-Interleaved Distributed Parity
 - Ähnlich wie Level 4
 - Blöcke mit den Paritätsbits sind über alle Datenplatten verteilt
 - Vorteile: Write-Requests können ausgeführt werden (kein Bottleneck mehr durch einzige Check Disk); höhere Parallelität beim Lesen (da alle Platten genutzt werden)
 - Beste Performance aller RAID Levels für kleine und große Read-Request und große Write-Requests



Pufferverwaltung in einem DBMS (Buffer Management)

- Puffer – Hauptspeicher-Bereich für Kopien von Platten-Blöcken
 - Enthält eine feste Anzahl von Pufferrahmen (Speicherbereiche von der Größe einer Seite)
- Puffer Manager – Subsystem zur Verwaltung des Puffers
 - ähnlich der virtuellen Speicherverwaltung in Betriebssystemen
- Daten müssen im RAM sein, damit DBMS darauf operieren kann
- Programm fragt Puffer Manager an, wenn es einen Block von der Platte braucht

Pufferverwaltung in einem DBMS



- Tabelle mit <frame#, pageid> pairs wird verwaltet

Wenn eine Seite benötigt wird...

- Wenn die angeforderte Seite nicht im Puffer ist:
 - Auswahl eines Pufferrahmens zur **Ersetzung** (wenn sämtliche Pufferrahmen belegt) nach einer bestimmten Strategie
 - Ein rausgeworfener Block wird nur auf die Platte geschrieben, falls er seit dem letzten Schreiben auf die Platte geändert wurde (**dirty**); Anderenfalls kann die betroffene Seite direkt überschrieben werden
 - Einlesen der gesuchten Seite in den gewählten Rahmen
- Rückgabe der Adresse der gelesenen Seite (pin the page and return its address)
- Wenn Requests vorgesehen werden können (z.B. beim sequentiellen Lesen) ist ein Pre-Fetch von Seiten möglich

Pufferverwaltung

- **Pinned Block/Markierte Seite** – darf nicht aus dem Puffer entfernt werden
- Der Requestor unpins eine Seite und gibt an ob die Seite geändert wurde oder nicht – **dirty pin** wird dafür benutzt
- Angeforderte Seiten werden mit einem Verwendungszähler (***pin count***) verwaltet
- Concurrency control & Recovery machen zusätzliche I/O Operationen erforderlich wenn eine Seite im Puffer für Ersetzen ausgewählt wurde (z.B. Write-Ahead Logging Protokoll – vor dem Schreiben einer schmutzigen Änderung in die Datenbank muss die zugehörige Undo-Information in die Log-Datei geschrieben werden)
- Auswahl eines Rahmens zur Ersetzung durch eine Ersetzungsstrategie (replacement policy)

Ersetzungsstrategien für Pufferseiten

- LRU Strategie (least recently used): Ersetze Block, der am längsten nicht benutzt wurde
 - Idee: Zugriffsmuster der Vergangenheit benutzen, um zukünftiges Verhalten vorherzusagen
 - Anfragen in DBMSs haben wohldefinierte Zugriffsmuster (z.B. sequentielles Lesen) und das DBMS kann die Information aus den Benutzeranfragen verwenden, um zukünftig benötigte Blöcke vorherzusagen
 - erfolgreich in Betriebssystemen eingesetzt
- MRU Strategie: (most recently used): Ersetze zuletzt benutzten Block als erstes
- Toss Immediate Strategy: Block wird sofort rausgeworfen, wenn das letzte Tupel bearbeitet wurde
- Gemischte Strategien mit Tipps vom Anfrageoptimierer ist am erfolgreichsten

Ersetzungsstrategien für Pufferseiten

- Information im Anfrageplan, um zukünftige Blockanfragen vorherzusagen
- Statistik über die Wahrscheinlichkeit, dass eine Anfrage für eine bestimmte Relation kommt
- Strategie hat großen Einfluß auf die Anzahl der I/O Operationen-abhängig vom Zugriffsmuster (access pattern)
- Sequential Flooding – verursacht bei LRU + wiederholtes Scannen von Daten
 - Jeder Seiten Request kann eine I/O Operation verursachen
 - MRU ist viel besser in dieser Situation (LRU kann schlecht für bestimmte Zugriffsmuster in Datenbanken sein)

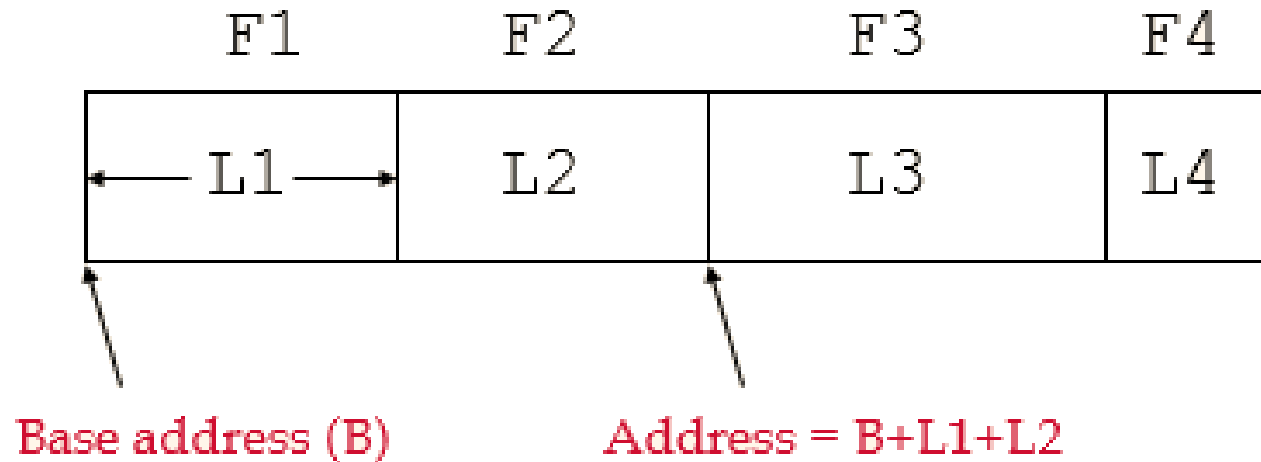
DBMS vs. Filesystem des Betriebssystems

- Betriebssystem umfasst Plattenspeicher- und Puffermanagement
→ warum nicht nutzen für DBMS?
- Es gibt Unterschiede bei den Betriebssystemen → Portabilitätsprobleme
- Beschränkungen: z.B. keine plattenübergreifenden Files
- Puffermanagement erfordert spezielle Fähigkeiten:
 - Markieren (**pin**) einer Seite im Puffer (inkrementieren **Verwendungszähler**), Zwingen einer Seite auf Platte (wichtig für Concurrency Control und Recovery)
 - Abstimmung von **Ersetzungsstrategien** und Unterstützung von **Prefetching** basierend auf Zugriffsmustern in typischen Datenbankoperationen

Datei aus Sätzen

- Für I/O Operationen genügen Seiten/Blöcke
- Höhere Schichten eines DBMS operieren auf **Sätzen (Records)** und **Dateien aus Sätzen (Files of records)**
- File/Datei:
 - Ansammlung von Seiten, die jeweils eine bestimmte Anzahl von Sätzen enthält
 - Unterstützt folgende Operationen:
 - Einfügen/Löschen/Modifizieren von Sätzen
 - Lesen eines bestimmten Satzes (spezifiziert durch eine Record-ID)
 - Scannen aller Sätze (möglich mit Einschränkungen auf eine Teilmenge von Sätzen, die zurückgegeben werden sollen)

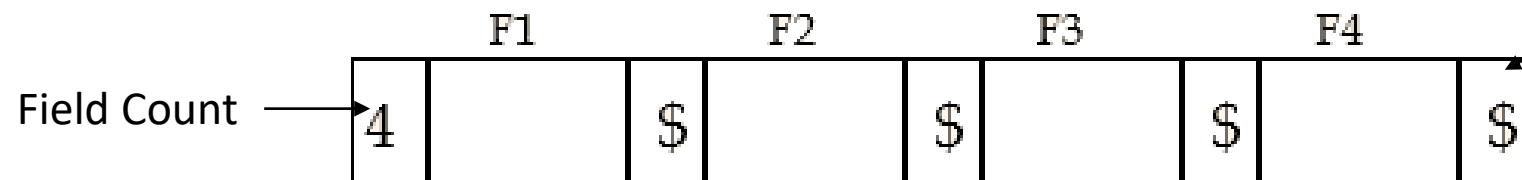
Satzformate in einem File – Feste Länge



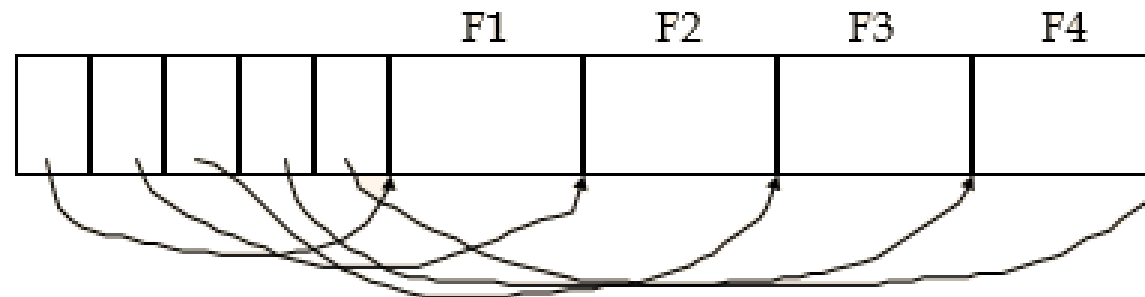
- Informationen über Feldtypen sind gleich für alle Datensätze in einem File
- Finde das i-te Feld erfordert Scannen des Datensatzes

Satzformate in einem File – Variable Länge

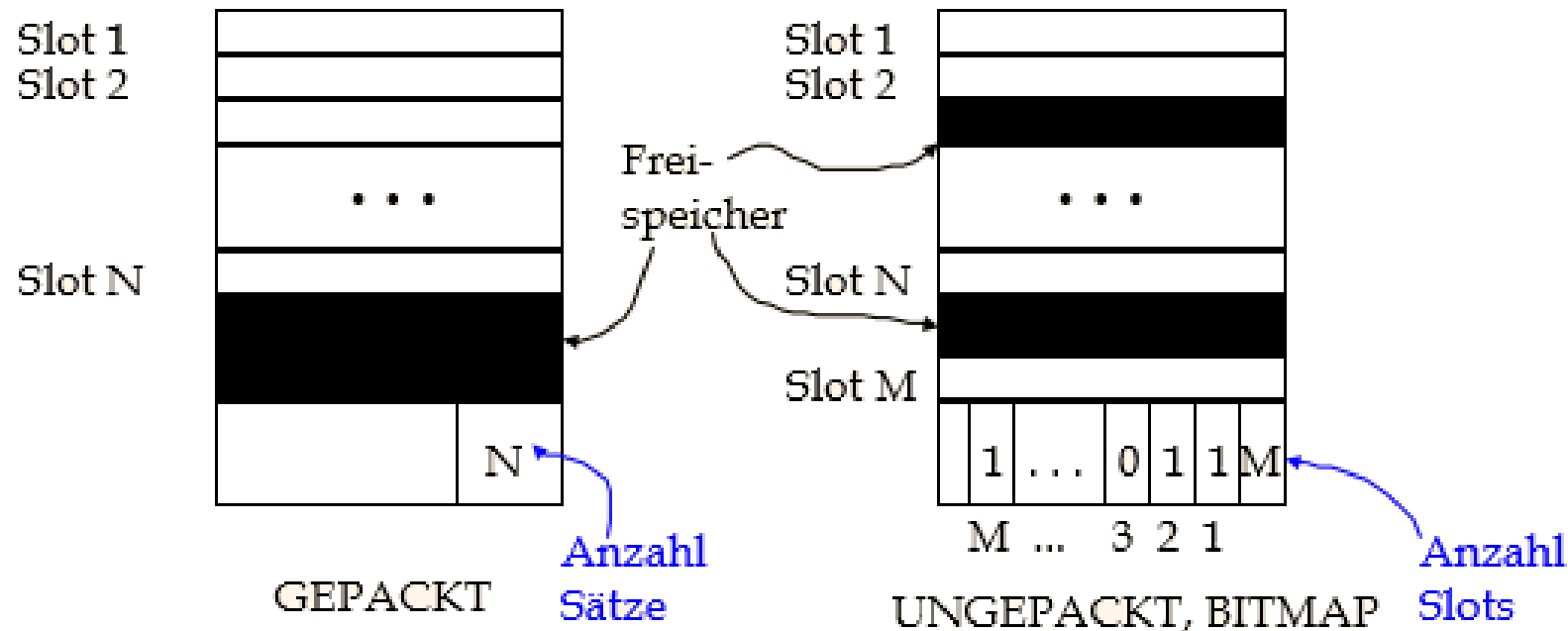
- Erstes Format (# Felder ist fix) – Felder getrennt durch spezielle Symbole



- Zweites Format – Array von Field Offsets
 - erlaubt direkten Zugriff aufs i-te Felds

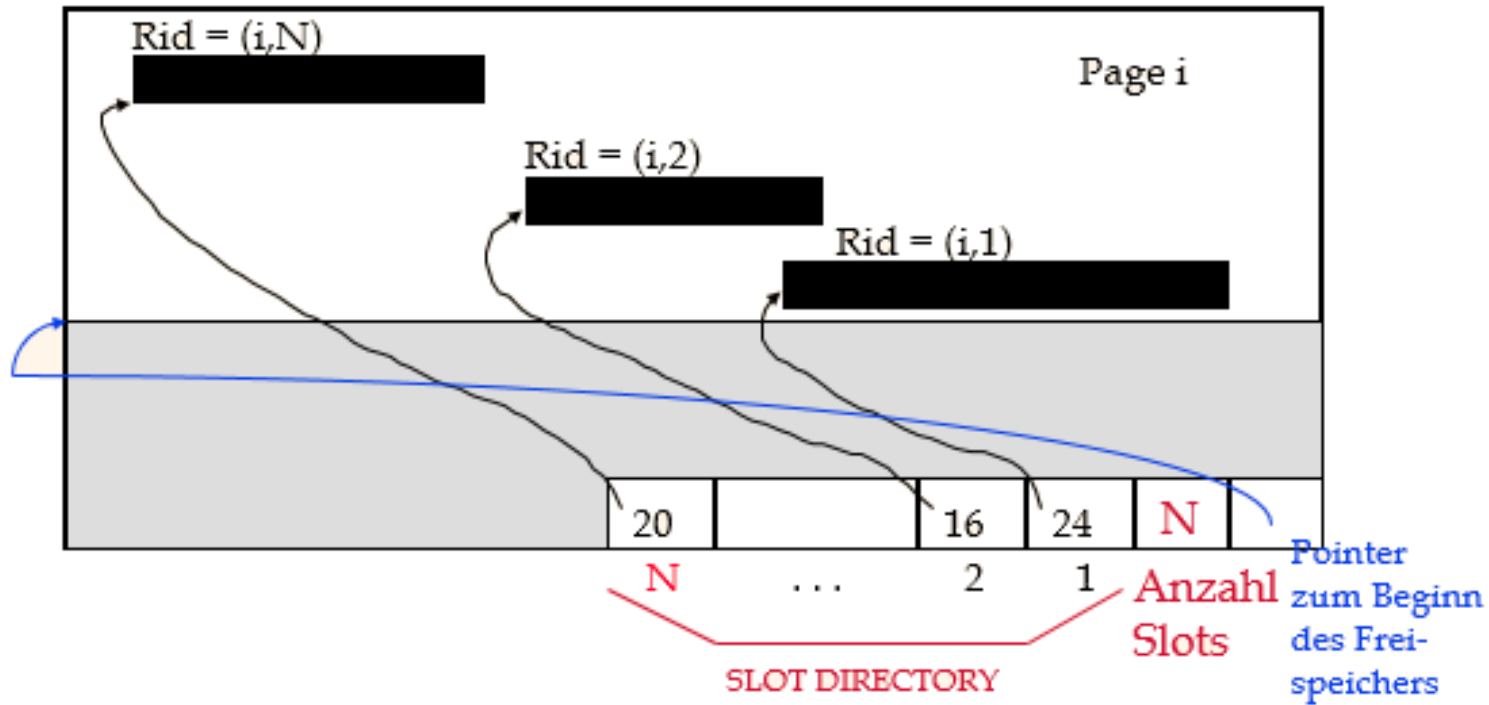


Seitenformate – Sätze fester Länge



- Record id = <page id, slot #> (Slot = Tupel-Identifikator)
- Erster Ansatz (gepackt): Verschieben von Sätzen in Freispeicher (z.B. Beim Löschen) so dass gilt: Satz i im Slot i. Das ändert jedoch die ID des Satzes (meist nicht akzeptabel)
- Bessere Alternative mit Bitmap

Seitenformate – Sätze variabler Länge



- Verschieben von Sätzen in einer Seite ohne Veränderung der ID möglich (somit auch attraktiv für Sätze fester Länge)
- Im Slot-Directory (Liste von Satz-Zeigern) stehen Offset und Länge für jeden Satz

Organisation von Datensätze

- Verschiedene Ansätze, um Datensätze in Dateien logisch anzuordnen:
 - Heap Datei – ein Datensatz kann irgendwo gespeichert werden, wo Platz frei ist, oder er wird am Ende angehängt
 - Sequentielle/Sortierte Datei – Datensätze werden nach einem bestimmten Datenfeld sortiert abgespeichert
 - Gut für Anwendungen, die sequentiellen Zugriff auf gesamte Datei brauchen
 - Hash Datei
 - Die Datei ist eine Sammlung von Blöcken (buckets)
 - der Hash-Wert für ein Datenfeld bestimmt, in welchem Block (bucket) der Datei der Datensatz gespeichert wird
 - Indexsequentielle Datei (Tree file)
 - Kombination von sequentieller Hauptdatei und Indexdatei
 - Hat die Vorteile der sortierten Dateien (die Nachteile sind nicht so stark)