
Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Datenbanksysteme II

# Physische Speicherstrukturen

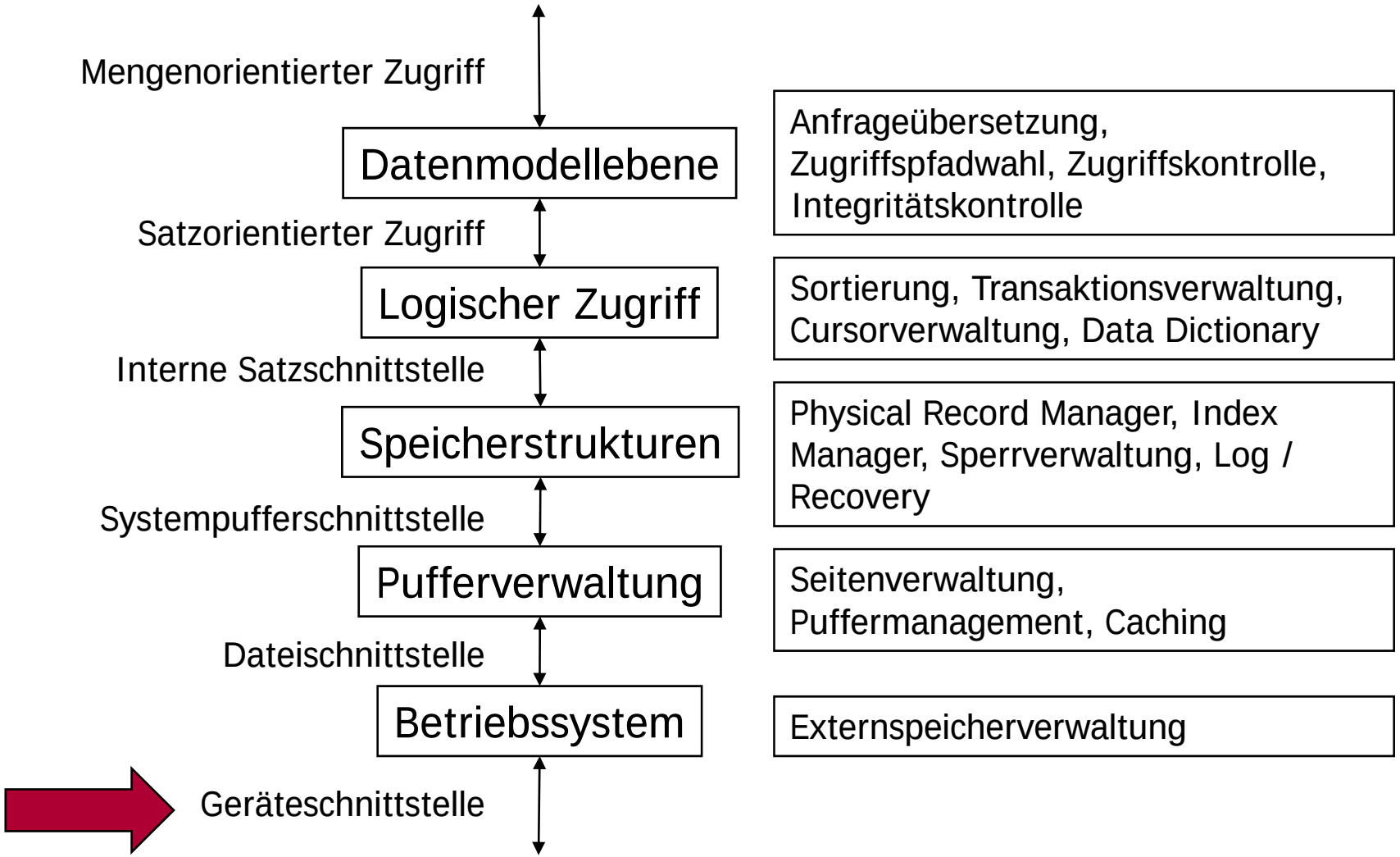
(Kapitel 11)

24.04.2008

Felix Naumann

# Zoom in die interne Ebene: Die 5-Schichten Architektur





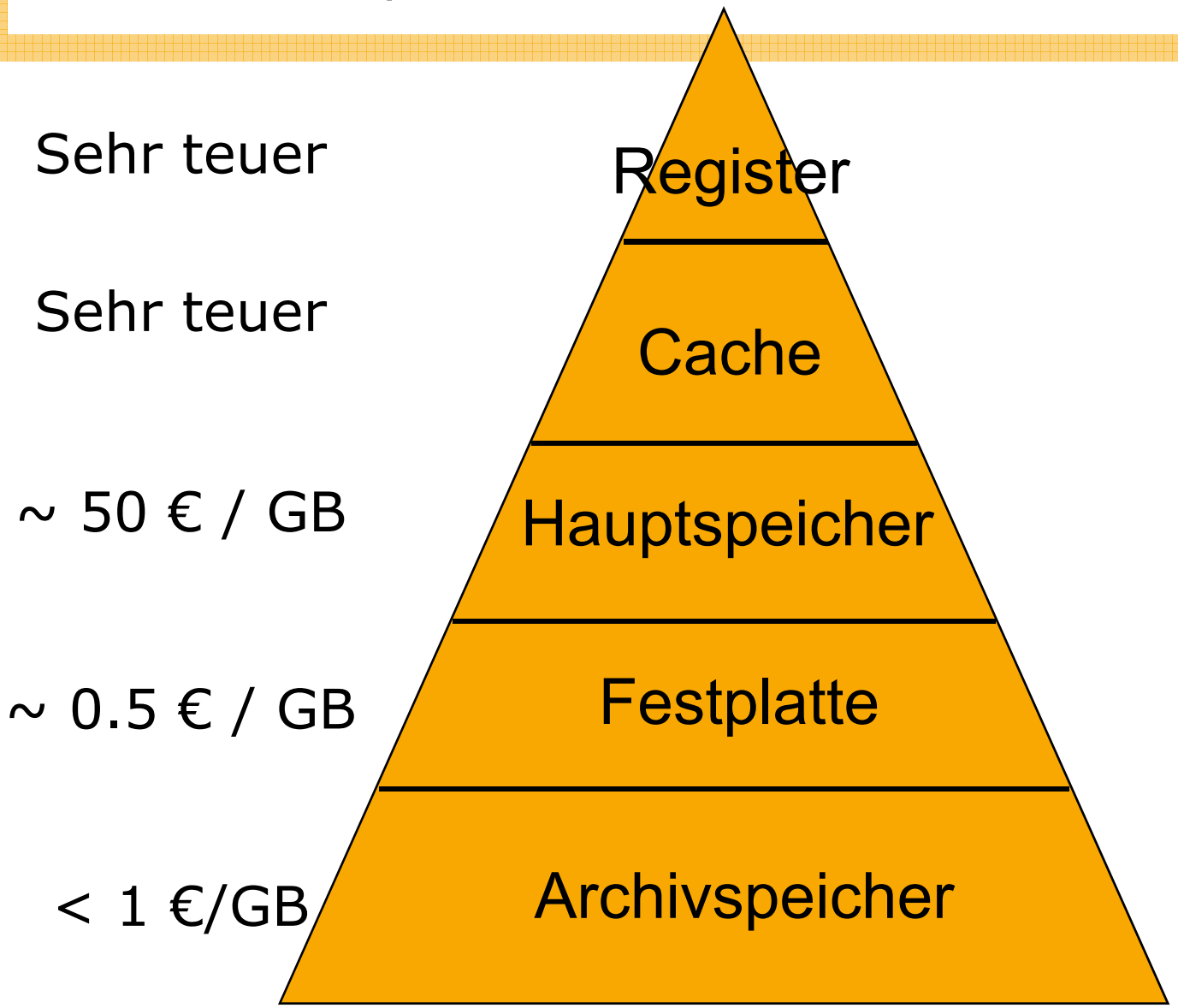
# Übersicht



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

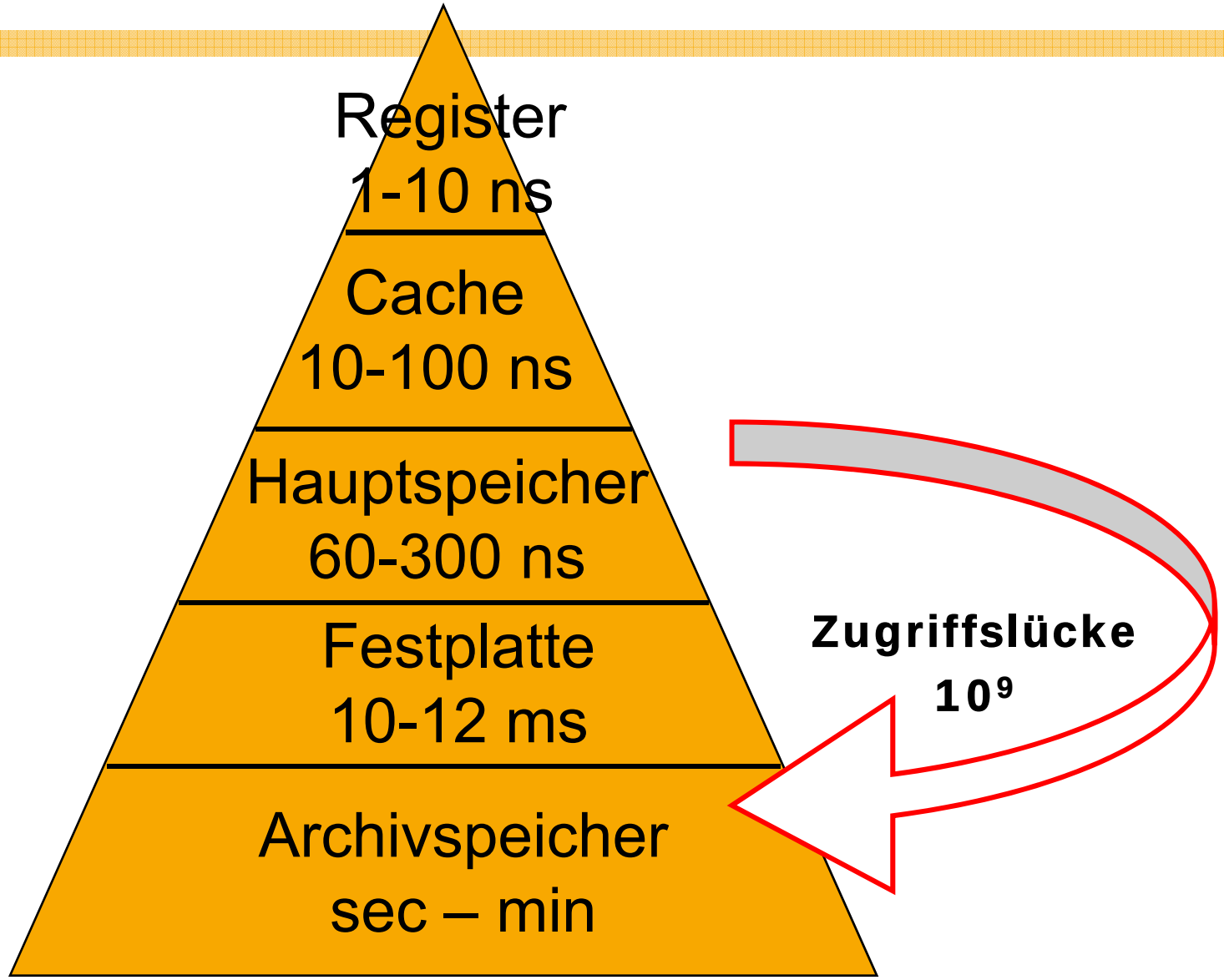
# Überblick: Speicherhierarchie





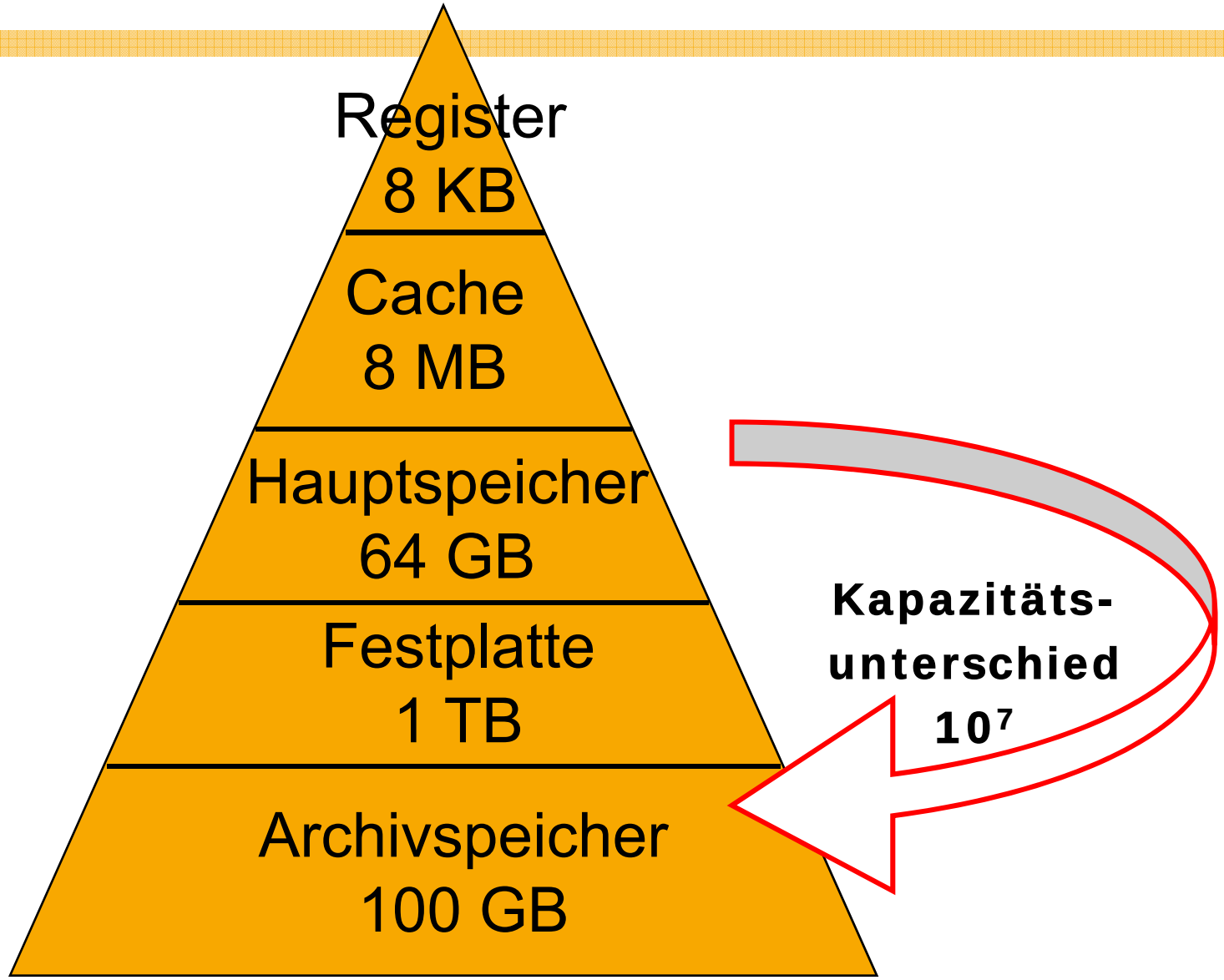
# Überblick: Speicherhierarchie





# Überblick: Speicherhierarchie





# Cache



**Allgemein:** Cache-Speicher vermindern die Zugriffslücke zwischen Medien, indem sie Daten des Mediums puffern, das in der Speicherhierarchie weiter unten liegt.

Beispiel: Prozessor-Cache

- Medien: Register (t_access: 1 ns) und Hauptspeicher (100 ns)
- enthält Kopien bestimmter Speicherorte des Hauptspeichers
- On-board Cache: direkt auf Prozessorchip
- Level-2 Cache: auf eigenem Chip
- endliche Kapazität → Verdrängungsstrategie (z. B. LRU, FIFO)
  - erst bei Verdrängung ggf. Daten im Hauptspeicher aktualisieren (write-back)
- Write-through bei Mehrprozessormaschinen
  - Jeder Prozessor hat eigenen Cache
  - Shared Memory: Prozessoren nutzen Hauptspeicher gemeinsam

# Cache – Lokalität des Zugriffs

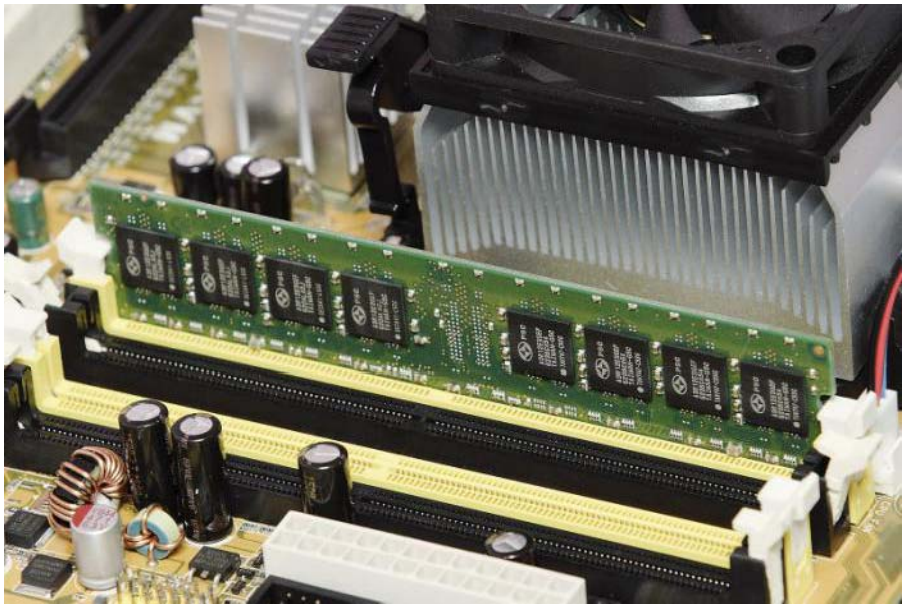


- Annahme: Typische Anwendungen können einen Großteil der Zugriffe (> 90%) mit Daten aus Cache beantworten
- Ohne Lokalität ist Cache nutzlos
- *cache hit*: angefordertes Datum im Cache
- *cache miss*: angefordertes Datum nicht im Cache
- *Cache-Hit-Ratio* = #hits / (#hits + #misses) = #hits / #requests

# Hauptspeicher



- Wahlfreier Zugriff (*random access*)
  - Zugriffszeit auf jedes Datum gleich
  - typische Zugriffszeiten: 10 bis 100 ns



# Virtueller Speicher



Jede Anwendung verwaltet einen **virtuellen Adressraum**

- Kann größer als tatsächlich verfügbarer Hauptspeicher sein
- 32-bit Adressraum $\rightarrow 2^{32}$ unterschiedliche Adressen darstellbar
- Jedes Byte hat eigene Adresse = max. Hauptspeichergröße 4 GB
- Aber: meist weniger als 4 GB Hauptspeicher vorhanden
- Abhilfe: Daten werden auf Disk ausgelagert
  - Lesen und Schreiben von ganzen Blöcken zwischen Hauptspeicher und Festplatte (Blockgröße 4 – 56 KB) $\rightarrow$ *Seiten des virtuellen Speichers*
  - Verwaltet durch Betriebssystem
  - **Datenbanken verwalten Daten auf Festplatte selbst!**
  - Für *main-memory DBMS* jedoch relevant
    - Wachsender Nischen-Markt
    - Sinnvoll bei kleinen, spezialisierten Datenbanken (die vollständig in Hauptspeicher geladen werden können)

# Sekundärspeicher: Festplatten

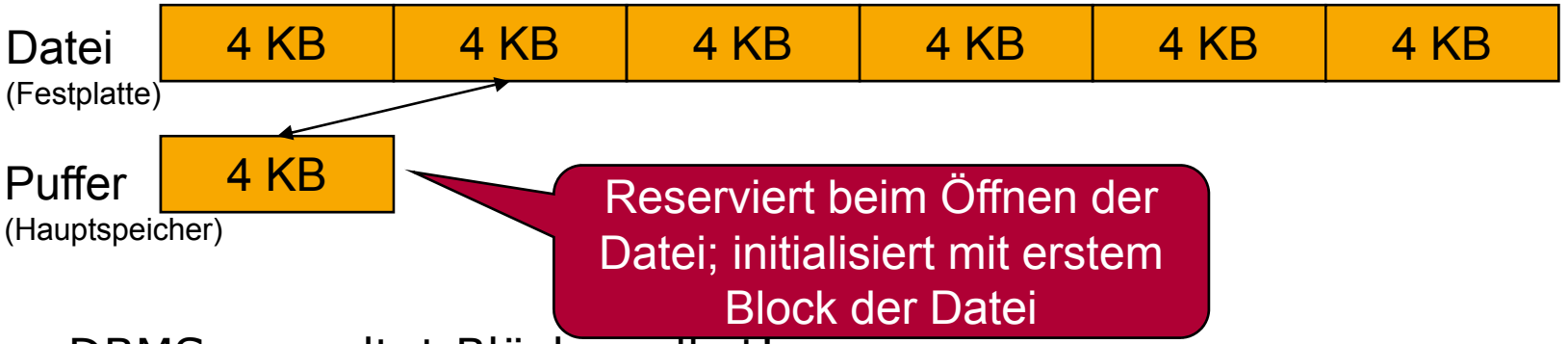


- Nicht nur (magnetische) Festplatten; auch optische (*read-only*) Speicher
- Im Wesentlichen wahlfreier Zugriff (*random access*)
  - Zugriff auf jedes Datum kostet gleich viel
  - Aber: Erst einmal hinkommen!
- Halten Daten aus Cache / Seiten des virtuellen Speichers von Anwendungsprogrammen
- Halten Daten aus Dateisystem
- Operationen
  - Disk-read (Kopieren eines Blocks in Hauptspeicher)
  - Disk-write (Kopieren eines Blocks aus dem Hauptspeicher)
  - Beides: Disk-I/O

# Festplatten - Puffer



- Hauptspeicher puffert Teile von Dateien
  - In Blockgröße (z.B. 4 KB)




- DBMS verwaltet Blöcke selbst!
- Dauer für Schreiben oder Lesen eines Blocks: 10 – 30 ms
  - in dieser Zeit viele Millionen Prozessoranweisungen ausführbar
  - I/O-Zeit dominiert Gesamtkosten
  - Deshalb am besten: Block sollte bereits im Hauptspeicher sein!

# Tertiärspeicher: Magnetbänder



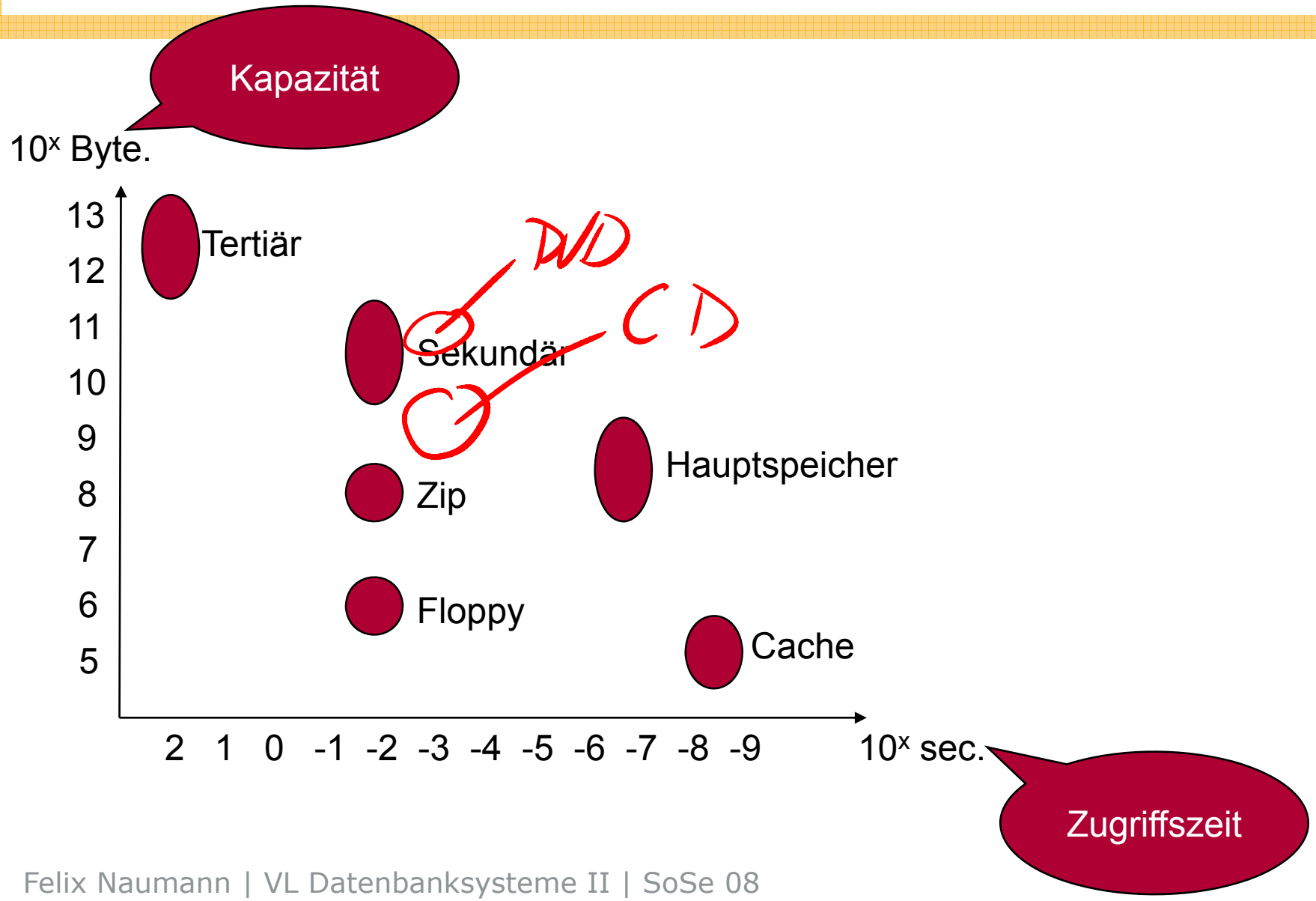
- Viele Terabyte ($10^{12}$ Bytes) Verkaufsdaten
- Viele Petabyte ($10^{15}$ Bytes) Satellitenbeobachtungsdaten
- Festplatten ungeeignet
  - Zu teuer (Wartung, Strom, Maschinen)
- Vergleich Tertiärspeicher – Sekundärspeicher
  - I/O-Zeiten wesentlich höher
  - Kapazitäten wesentlich höher
  - Kosten pro Byte geringer
- Kein wahlfreier Zugriff (*random access*)
  - Zugriffszeiten hängen stark von der Position des jeweiligen Datensatzes ab (in Bezug auf die aktuelle Position des Schreib-/Lesekopfes)

# Tertiärspeicher



- Ad-hoc Speicherung auf Magnetbändern
  - Magnetbandspulen
  - Kassetten
  - Von Menschenhand ins Regal
  - Gut beschriften!
- Magnetbandroboter (Silo)
  - Roboter bedient Magnetbänder (Kassetten)
  - 10 mal schneller als Mensch
- CD / DVD - Juke-Boxes
  - Roboterarm extrahiert jeweiliges Medium (CD oder DVD)
  - Hohe Lebensdauer (30 Jahre)
    - Wahrscheinlicher, dass kein Lesegerät mehr existiert

# Vergleich (Stand 2001)

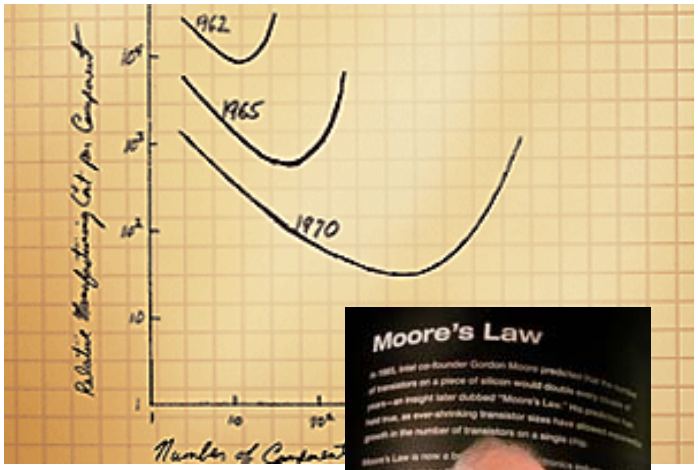


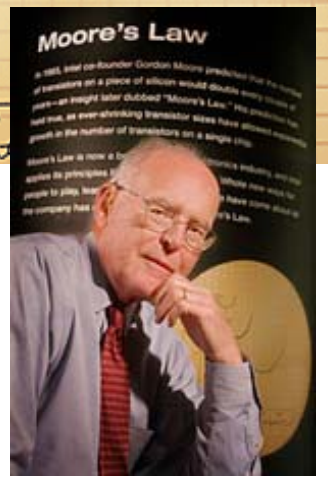
# Moore's Law (Gordon Moore, 1965)



Exponentielles Wachstum vieler Parameter

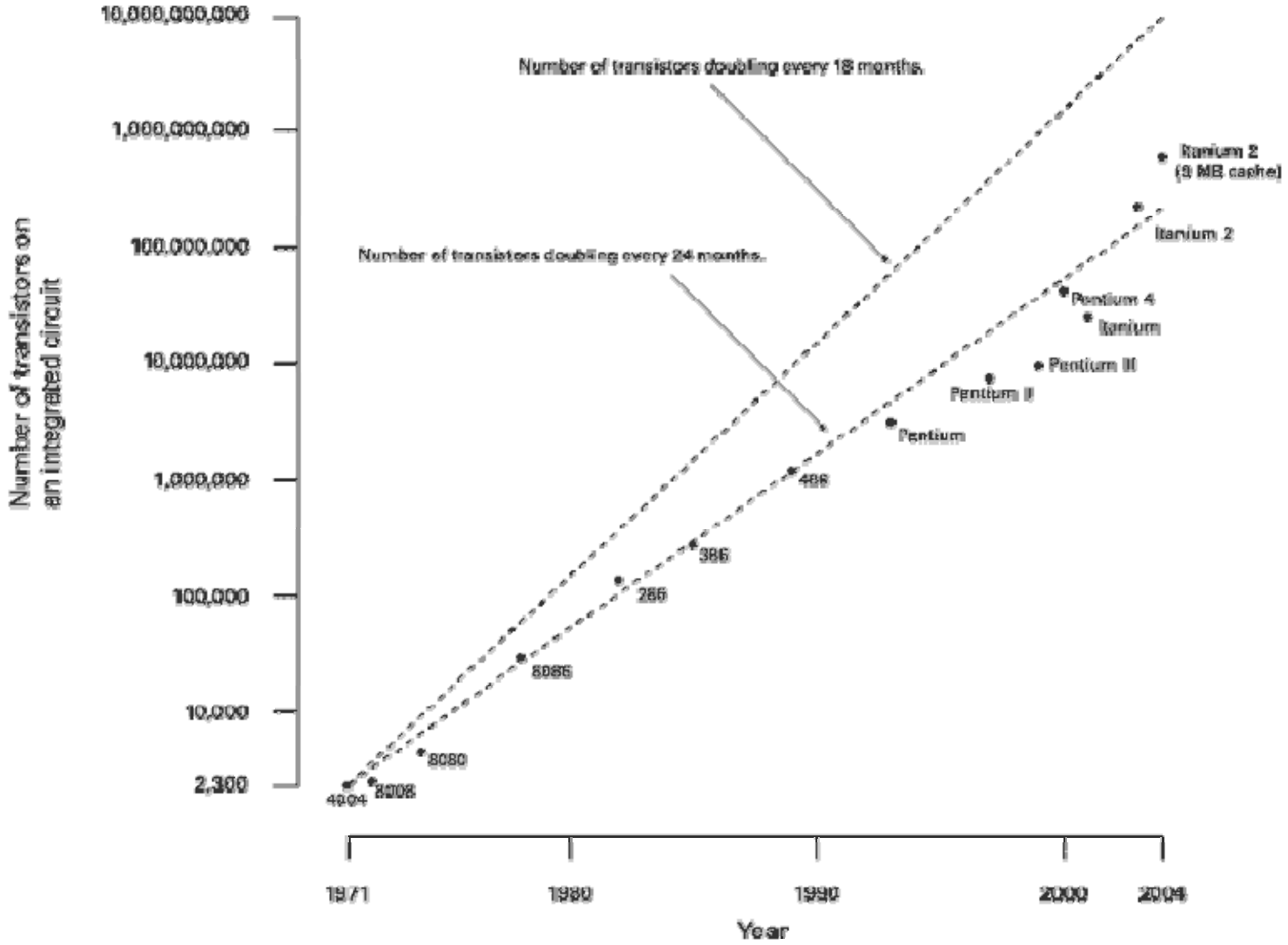
- Verdopplung alle 18 Monate
  - Prozessorgeschwindigkeit (# instr. per sec.)
  - Hauptspeicherkosten pro Bit
  - Anzahl Bits pro $cm^2$ Chipfläche
  - Diskkosten pro Bit
  - Kapazität der größten Disks
- Aber: Sehr langsames Wachstum von
  - Zugriffsgeschwindigkeit im Hauptspeicher
  - Rotationsgeschwindigkeit von Festplatten
- Folge: Latenz wächst
  - Bewegung von Daten innerhalb der Speicherhierarchie erscheint immer langsamer (im Vergleich zum Prozessorgeschwindigkeit)



# Moore's Law




10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
Number of transistors on an integrated circuit
Number of transistors doubling every 18 months.
Number of transistors doubling every 24 months.
Itanium 2 (9 MB cache)
Itanium 2
Pentium 4
Itanium
Pentium III
Pentium II
Pentium
486
386
286
8086
8080
4004
8008
1971
1980
1990
2000
2004
Year

# Übersicht

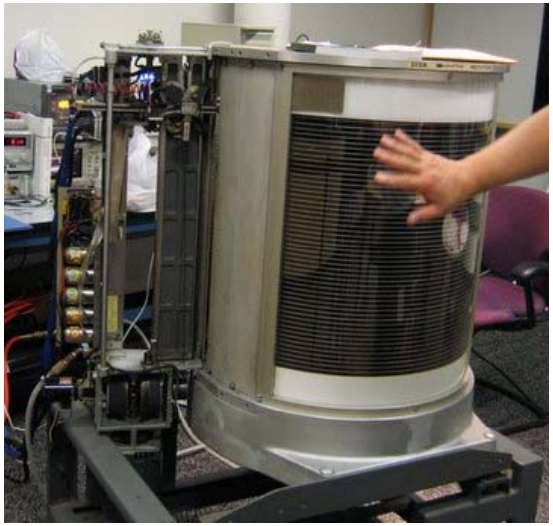


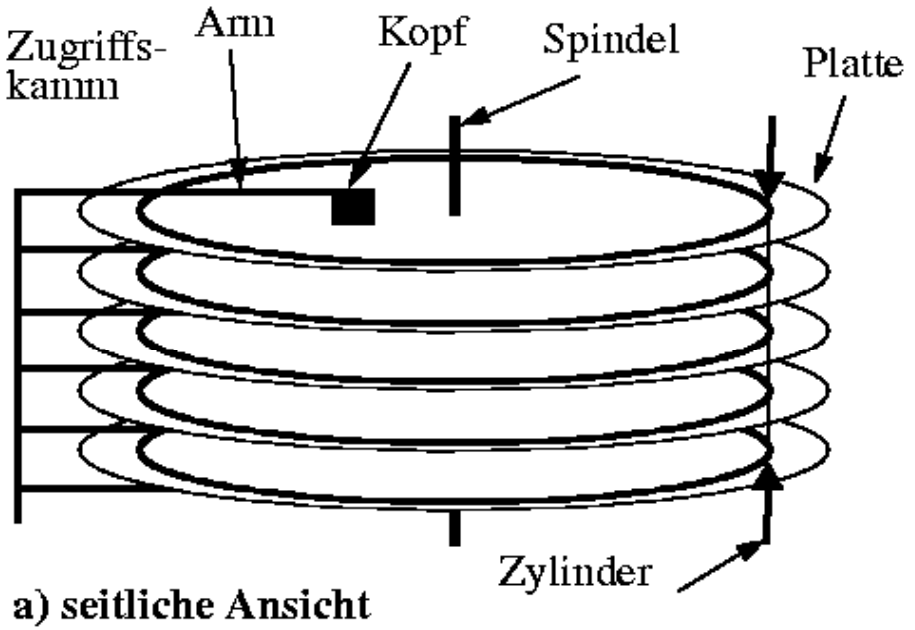
- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

# Aufbau



- Mehrere (5-10) gleichförmig rotierende **Platten** (z. B. 3.5“ Durchmesser)
- Für jede Plattenoberfläche (10-20) ein Schreib-/Lese-Kopf
  - Gleichförmige Bewegung
- magnetische Plattenoberfläche ist in **Spuren** eingeteilt
- Spuren sind als **Sektoren** fester Größe formatiert
  - Anzahl Sektoren pro Spur kann sich unterscheiden
- Übereinandergeordnete Spuren sind ein **Zylinder**



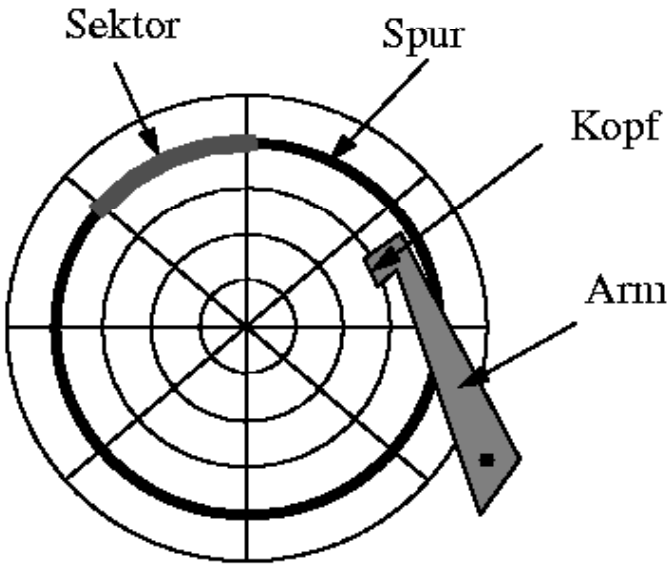



a) seitliche Ansicht

# Aufbau



- Sektoren (1-8 KB) kleinste **physische** Leseeinheit
  - Größe ist von Hersteller festgelegt
  - Mehr Sektoren auf äußeren Spuren
- Lücken zwischen Sektoren nehmen ca. 10% der Spur ein.
  - nicht magnetisiert
  - dienen zum Auffinden der Sektoranfänge
- Blöcke sind die **logische** Übertragungseinhe
  - Können aus einem oder mehreren Sektoren bestehen




hier: jede Spur hat gleiche Anzahl an Sektoren

# Disk Controller



- Kontrolliert eine oder mehrere Disks
- Kontrolliert Bewegung der Schreib-/Lese-Köpfe
  - Spuren, die zu einem Zeitpunkt unter den Schreib-/Lese-Köpfen sind, bilden **Zylinder**
- Wählt Plattenoberfläche, auf die zugegriffen werden muss
- Wählt Sektor innerhalb der Spur, die sich aktuell unter dem Schreib-/Lese-Kopf befindet
  - Kontrolliert Start und Ende eines Sektors
- Überträgt Bits zwischen Disk und Hauptspeicher bzw. umgekehrt

# Disk Eigenschaften



| Eigenschaft | 2004 | 1998 | 1990 | 1985 | 1970 |
|---|---|---|---|---|---|
| Mittlere Zugriffs-bewegungszeit (ms) | 5 | 8 | 12 | 16 | 30 |
| Umdrehungszeit (ms) | 6 | 6 | 14 | 16,7 | 16,7 |
| Spurkapazität (KB) | 420 | 100 | 56 | 47 | 13 |
| Transferrate (MB/s) | 30-40 | 15 | 4,2 | 3 | 0,8 |
| Zylinder (#) | 90.000 | 5.000 | 2226 | 2655 | 411 |
| Kapazität (GB) | 150 | 10 | 1,89 | 1,89 (?) | 0,094 |

10000 U / min

# Datenbankgrößen



- 0,5 KB = Buchseite
- 1KB = 1000 Byte
- 30KB = gescannte, komprimierte Buchseite
- 1MB = 1000000 Byte
- 5MB = Bibel als Text
- 20MB = Gescanntes Buch
- 500MB = CD-ROM; Oxford English Dictionary
- 1GB = 1000000000 Byte
- 4,7 GB = DVD
- 10 GB = komprimierter Spielfilm
- 100GB = ein Stockwerk einer Bibliothek
- 200GB = Kapazität eines VHS Bandes
- 1TB = 1000000000000 Byte
- 1TB = Bibliothek mit 1 Mill. Bände
- 20TB = Library of Congress als Textdatei
- 1PB = 1000000000000000 Byte
- 1PB = eingescannte Bände einer Nationalbibliothek
- 15PB = weltweite Plattenproduktion 1996
- 200 PB = weltweite Magnetbandproduktion in 1996

# Beispiel – Megatron 747 disk



- Eigenschaften
  - 8 Platten mit 16 Plattenoberflächen (Durchmesser: 3,5“)
  - $2^{14}$ = 16 384 Spuren pro Oberfläche
  - Durchschnittlich $2^7$ = 128 Sektoren pro Spur
  - $2^{12}$ = 4 096 Byte pro Sektor
- Gesamtkapazität?
  - 16 x 16 384 x 128 x 4 096 = $2^{37}$ Byte = 128 GB
- Blocks z.B. $2^{14}$ Byte (= 16 KB)
  - → 4 Sektoren pro Block ($2^{14}$ / $2^{12}$)
  - → 32 Blöcke pro Spur ($2^7$ / $2^2$)
- Bitdichte (äußerste Spur)
  - Bits pro Spur: $2^7$ x $2^{12}$ Byte = $2^{19}$ = 512 KB = 4 MBit
  - Spurlänge (äußerste Spur): 3,5“ · π ≈ 11“
  - ca. 10% Lücken → Spurlänge von 9,9“ hält 4 MBits
  - → 420 000 Bits pro Zoll

# Disk-Zugriffseigenschaften



- Voraussetzungen für Zugriff auf einen Block (lesend oder schreibend)
  - S-/L-Kopf ist bei Zylinder positioniert, der die Spur mit dem Block enthält.
  - Disk rotiert so, dass Sektoren, die der Block enthält, unter den S-/L-Kopf gelangen.
- **Latenzzeit**
  - Zeit zwischen Anweisung, einen Block zu lesen, bis zum Eintreffen des Blocks im Hauptspeicher

# Latenzzeit



Latenzzeit ist Summe aus vier Komponenten:

1. **Kommunikationszeit** zwischen Prozessor und Disk Controller
   - Bruchteil einer Millisekunde → ignorieren
   - Annahme hier: Keine Konkurrenz
2. **Seektime** (Suchzeit) zur Positionierung des Kopfes unter richtigem Zylinder
   - Zwischen 0 und 42 ms → Zeit proportional zum zurückgelegten Weg
   - Startzeit (1 ms), Bewegungszeit (10 – 40 ms), Stopzeit (1 ms)
3. **Rotationslatenzzeit** zur Drehung der Disk bis erster Sektor des Blocks unter S-/L-Kopf
   - Durchschnittlich ½ Umdrehung (5 ms)
   - Optimierung durch Spur-Cache im Disk-Controller möglich
4. **Transferzeit** zur Drehung der Disk bis alle Sektoren und die Lücken des Blocks unter S-/L-Kopf passiert sind
   - Ca. 16 KB-Block in ½ ms

# Schreiben und Ändern von Blöcken



- Schreiben von Blöcken
  - Vorgehen und Zeit: Analog zum Lesen
  - Zur Prüfung, ob Schreiboperation erfolgreich war, muss eine Rotation gewartet werden (Nutzung von Checksums (später))
- Ändern von Blöcken
  - Nicht direkt möglich
  1. Lesen des Blocks in Hauptspeicher
  2. Ändern der Daten
  3. Zurückschreiben auf Festplatte
  4. evtl. Korrektheit der Schreiboperation überprüfen
  - Zeit: t_read + t_write
  - aber: mit Glück ist Kopf noch in der Nähe (t_write ist billiger)

# Beispiel – Megatron 747 Disk



Wie lange dauert es, einen Block (16 KB = 16 284 Byte) zu lesen?

- Umdrehungsgeschwindigkeit 7200 U · $min^{-1}$
  - → eine Umdrehung in 8,33 ms
- Seektime
  - Start und Stopp zusammen 1ms
  - 1ms pro 1000 Zylinder, die überbrückt werden
    - Minimum (1 Zylinder): 1,001 ms
    - Maximum (16,383 Zylinder): 17,38 ms
- Minimale Zeit, um den Block zu lesen?
  - S-/L-Kopf steht über richtiger Spur und Platte ist schon richtig rotiert
  - 4 Sektoren und 3 Lücken sind zu lesen
  - 128 Lücken und 128 Sektoren pro Spur
  - Lücken bedecken 36° (10%), Sektoren bedecken 324° des Kreises (360°)
  - 324° x 4 / 128 + 36° x 3 / 128 = 10,97° des Kreises durch Block bedeckt
  - (10,97° / 360°) · 8,33 ms = 0,253 ms
- Maximale Zeit: Hausaufgabe (26 ms)
- Durchschnittliche Zeit: Hausaufgabe (11 ms)

# Übersicht



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

# Algorithmen vs. DBMS



- Annahme bei Algorithmen:
  - *RAM-Berechnungsmodell*
    - gesamte Daten passen in Hauptspeicher
    - Daten befinden sich schon im Hauptspeicher
- Annahme bei Implementierung von DBMS
  - *I/O-Modell*: gesamte Daten passen **nicht** in Hauptspeicher
- Externspeicher-Algorithmen funktionieren oft anders
  - guter ESA muss nicht bester Algorithmus lt. RAM-Modell sein
  - Entwurfsidee: I/O vermeiden
- Gleiches kann auch für Hauptspeicher-Algorithmen gelten
  - Ausnutzen des Caches → Cachegröße berücksichtigen
  - Lokalität beachten („maximiere" Anzahl der cache hits)

# I/O-Modell



- Beispiel: Einfaches DBMS
  - Zu groß für Hauptspeicher
  - Eine Disk, ein Prozessor, viele konkurrierende Nutzer / Anfragen
- Disk-Controller hält Warteschlange mit Zugriffsaufforderungen
  - Abarbeitungsprinzip: First-come-first-served
  - Jede Aufforderung erscheint zufällig
    - Kopf ist also an zufälliger Position

**Definition (Dominanz der I/O-Kosten)**

- Kosten der Bewegung eines Block zwischen Disk und Hauptspeicher sind wesentlich größer als Kosten der Operationen auf den Daten im Hauptspeicher.

→ Anzahl der Blockzugriffe (lesend und schreibend) ist gute Approximation der Gesamtkosten und sollten minimiert werden.

# Beispiel für das I/O-Modell (1): Indizes



- Relation *R*
- Anfrage sucht nach dem Tupel *t* mit Schlüsselwert *k*
- *Index* auf Schlüsselattribut
  - Datenstruktur, die schnellen Zugriff auf Block ermöglicht, der *t* enthält
  - Variante A des Index sagt nur, in welchem Block *t* liegt.
  - Variante B sagt zusätzlich, an welcher Stelle innerhalb des Blocks *t* liegt.
- **Frage:** Welcher Indexvariante ist besser geeignet?

- durchschnittlich 11 ms um 16 KB-Block zu lesen
  - in dieser Zeit: viele Millionen Prozessoranweisungen möglich
- Suche nach *k* auf dem Block kostet höchstens Tausende Prozessoranweisungen – selbst mit linearer Suche
- Aber: Zusätzliche Informationen in Variante B nehmen Platz ein (höhere I/O-Kosten).

# Beispiel für das I/O-Modell (2): Sortierung



- Relation *R*
  - 10 Millionen Tupel
  - Verschiedene Attribute, eines davon ist Sortierschlüssel
    - Nicht unbedingt eindeutig (kein Primärschlüssel)
    - hier vereinfachende Annahme: Sortierschlüssel ist eindeutig
  - Gespeichert auf Diskblöcken der Größe 16 384 = $2^{14}$ Byte
  - Annahme: 100 Tupel passen in einen Block
    → Tupelgröße ca. 160 Byte
    → R belegt 100 000 Blöcke (1,64 Mrd. Bytes) auf Festplatte
- verwendete Festplatte: 1 x Megatron 747
- verfügbarer Hauptspeicherpuffer: 100 MB (= $100 \cdot 2^{20}$)
  → $100 \cdot 2^{20} / 2^{14} = 6400$ Blöcke von *R* passen in Hauptspeicher
- **Ziel:** Sortierung soll Anzahl der Lese- und Schreiboperationen minimieren
  - Wenig „Durchläufe" durch die Daten

# Merge Sort



- Hauptspeicher-Algorithmus (Divide-and-Conquer Algorithmus)
- Idee: Mische $l \geq 2$ sortierte Listen zu einer größeren sortierten Liste.
  - Wähle aus den sortierten Listen stets das kleinste Element und füge es der großen Liste hinzu.

| | **Liste 1** | **Liste 2** | **Outputliste** |
|---|---|---|---|
| 1. | 1,3,4,9 | 2,5,7,8 | - |
| 2. | 3,4,9 | 2,5,7,8 | 1 |
| 3. | 3,4,9 | 5,7,8 | 1,2 |
| 4. | 4,9 | 5,7,8 | 1,2,3 |
| 5. | 9 | 5,7,8 | 1,2,3,4 |
| 6. | 9 | 7,8 | 1,2,3,4,5 |
| 7. | 9 | 8 | 1,2,3,4,5,7 |
| 8. | 9 | - | 1,2,3,4,5,7,8 |
| 9. | - | - | 1,2,3,4,5,7,8,9 |

# Merge Sort



- Rekursion
  - Teile eine Liste mit mehr als einem Element beliebig in zwei gleich lange Listen $L_1$ und $L_2$ und auf.
  - Sortiere $L_1$ und $L_2$ rekursiv
  - Mische $L_1$ und $L_2$ zu einer sortierten Liste zusammen
- Aufwand (Eingabegröße $|R| = n$)
  - Mischen zweier sortierter Listen $L_1$, $L_2$: $O(|L_1| + |L_2|) = O(n)$
  - Rekursionstiefe: $\log_2 n$
    - in jedem Rekursionsschritt halbiert sich die Listenlänge
    - nach i Schritten noch $n / 2^i$ Elemente in der Liste
  - **Ergo:** $O(n \log n)$
  - trifft die untere Schranke für das vergleichsbasierte Sortieren

# Two-Phase, Multiway Merge-Sort (TPMMS)



- TPMMS wird in vielen DBMS eingesetzt.
- Besteht aus zwei Phasen
- Phase 1:
  - Lade jeweils soviele Tupel, wie in Hauptspeicher passen
  - Sortiere Teilstücke (im Hauptspeicher)
  - Schreibe sortierte Teilstücke auf Festplatte zurück
  - Ergebnis: viele sortierte Teillisten (auf Festplatte)
- Phase 2:
  - Mische alle sortierten Teillisten in einzige große Liste

# TPMMS – Phase 1



- Rekursionsanfang nun nicht nur mit einem oder zwei Elementen!
- Sortierung der Teillisten z.B. mit Quicksort (worst-case, sehr selten $O(n^2)$)

1. Fülle verfügbaren Hauptspeicher mit Diskblöcken aus Originalrelation
2. Sortiere Tupel, die sich im Hauptspeicher befinden
3. Schreibe sortierte Tupel auf neue Blöcke der Disk

→ Resultat: eine sortiere Teilliste

- Beispiel
  - 6 400 Blöcke in Hauptspeicher; 100 000 Blöcke insgesamt
  - 16 Füllungen des Hauptspeichers nötig (Letzte Füllung ist kleiner)
  - Aufwand: 200 000 I/O-Operationen
    - 100 000 Blöcke lesen
    - 100 000 Blöcke schreiben
  - Zeit: durchschnittlich 11 ms pro I/O-Operation
    → 11 ms · 200 000 = 2 200 s = **37 min**
    → Prozessorzeit für Sortieren vernachlässigbar

# TPMMS – Phase 2



- Naive Idee: paarweises mischen von *l* sortierten Teillisten
  - erfordert 2 log *l* mal Lesen und Schreiben jedes Blocks (jedes Tupels)
  - Im Beispiel: Ein Durchlauf für 16 sortierte Teillisten, einer für 8, einer für 4 und ein letzter für 2 sortierte Teillisten
    - Insgesamt ist jeder Block an 8 I/O-Operationen beteiligt
- Bessere Idee: Lesen nur ersten Block **jeder** Teilliste

1. Suche kleinsten Schlüssel unter den ersten Tupel aller Blöcke
   - Lineare Suche (lin.), Priority Queue (log.)
2. Bewege dieses Element in Output-Block (im HS)
3. Falls Output-Block voll ist, schreibe auf Disk
4. Falls ein Inputblock leer ist, lese nächsten Block aus gleicher Liste

- Aufwand: 2 I/O-Operationen pro Block (und Tupel): ebenfalls 37 min

→ Laufzeit für TPMMS insgesamt: **74 min**

**Was machen wir, wenn # Teillisten > # Blocks im Hauptspeicher?**

# Bemerkungen zur Blockgröße



- Beobachtung
  - Je größer Blockgröße, desto weniger I/O-Operationen
  - (Transferzeit erhöht sich etwas)
- Beispiel bisher
  - Blockgröße 16 KB
  - ∅ Latenzzeit 10,88 ms (davon nur 0,253 ms für Transfer)
- Beispiel neu
  - Blockgröße 512 KB (16 · 32)
  - ∅ Latenzzeit 20ms (davon 8 ms für Transfer)
  - Nur noch 12 500 I/Os für Sortierung nötig
  - Gesamtzeit: **4,16 min**
  - **17-fache Beschleunigung!**

- Nachteile der Blockvergrößerung
  - Blocks sollten sich nicht über mehrere Spuren erstrecken
  - Kleine Relationen nutzen nur Bruchteile eines Blocks
    → Speicherverschwendung
  - Viele Datenstrukturen für Externspeicher bevorzugen Aufteilung von Daten auf viele (kleine) Blöcke

# TPMMS – Grenzen



- Notation: Blockgröße *B* Bytes, Hauptspeichergröße (für Blocks) *M* Bytes, Tupelgröße *R* Bytes
- *M* / *B* Blöcke passen in Hauptspeicher
- In Phase 2 wird Platz für einen Outputblock benötigt.
- Phase 1 kann also maximal M / B - 1 sortierte Teillisten erzeugen
- Ebenso oft kann man also den Hauptspeicher mit Tupeln füllen und sortieren (in Phase 1)
  - jede Füllung enthält M / R Tupel
- maximal (M / R) * ((M / B) - 1) ≈ M² / (RB) Tupel sortierbar
- unser Beispiel:
  - M = 104 857 600 Bytes, B = 16 384 Bytes, R = 160 Bytes
  - Zusammen: maximale Eingabegröße 4,2 Mrd. Tupel (ca. 0.67 Terabyte)

# TPMMS – Grenzen



- Falls Eingaberelation noch größer ist
  - Füge dritte Phase hinzu
  - Nutze TPMMS, um sortierte Listen der Größe $M^2 / (RB)$ zu erzeugen
  - Phase 3: Mische maximal $M / B - 1$ solcher Listen zu einer sortierten Liste zusammen
- Insgesamt $M^3 / (RB^2)$ Tupel sortierbar
- Bezogen auf unser Beispiel:
  maximale Eingabegröße 27 Billionen Tupel (ca. 4.3 Petabytes)
- Idee verallgemeinerbar $\rightarrow$ Übung

# Übersicht



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

# Zugriffsbeschleunigung – Ideen



- Annahmen bisher
  - Nur eine Disk
  - Blockzugriffe zufällig (viele kleine Anfragen)
- Verschiedene Verbesserungsideen
  - Blöcke, die gemeinsam gelesen werden, auf gleichen **Zylinder** platzieren → Reduzierung der *seek time*
  - Daten auf mehrere (kleine) Disks **verteilen**
    - Unabhängige Schreib-/Leseköpfe
    - Dadurch: mehrere (unabhängige) Blockzugriffe gleichzeitig
  - **Spiegelung** von Daten auf mehrere Disks
  - Verwendung eines Disk-**Scheduling** Algorithmus
  - **Prefetching** von Blöcken
    - Ablegen von Blöcken im Hauptspeicher, die möglicherweise demnächst benötigt werden

# Daten gemäß Zylinder organisieren



- *seek time* macht ca. 50% der durchschnittlichen Blockzugriffszeit aus
  - Megatron 747: *seek time* zwischen 0 und 42 ms
- Idee: Daten, die zusammen gelesen werden, auf gleichen Zylinder platzieren
  - Z.B. Tupel einer Relation
  - Falls Zylinder nicht ausreicht: nutze mehrere nebeneinander liegende Zylinder
- Beim Lesen der Relation fällt im besten Fall nur **einmal** die *seek time* an.
  - Maximale Zugriffszeit der Disk wird erreicht: Zugriffszeit wird nur noch durch die Transferzeit bestimmt

# Zylinderorganisation – Beispiel



- Megatron 747-Festplatte
  - mittlere Transferzeit pro Block: ¼ ms
  - mittlere *seek time*: 6,46 ms
  - mittlere Rotationslatenzzeit: 4,17 ms
  - Jede der 16 Oberflächen mit 16.384 Spuren á 512 Blöcken (durchschnittlich)
- Sortierung von 10 Mio. Tupeln mittels TPMMS-Algorithmus dauerte 74 min
  - 100.000 Blöcke von *R* belegen 196 Spuren → brauchen 196 / 16 = 13 Zylinder
- **Phase 1 – Lesen der Blöcke**
  - Hauptspeicher (6.400 Blöcke) wird 16 mal gefüllt
  - Müssen Blöcke von 13 Zylindern lesen, die aber direkt nebeneinander liegen
    - nur 1 ms für Spurwechsel
  - Keine Rotationslatenzzeit: Reihenfolge beim Lesen der Tupel egal
  - Zeit pro Füllung:
    - 6,46 ms (1. seek) + 12 ms (12 mal Spurwechsel) + 1,6 s (Transfer 6400 Blöcke) = 1,6 s
  - 1,6 s x 16 Füllungen = 26 s (<< 18 min!)
- **Phase 1 – Schreiben:** analog zum Lesen → zusammen 52 s (vorher: 37 min)
- **Phase 2** wird nicht beschleunigt
  - Lesen aus verschiedenen (verteilten) Teillisten
  - Schreiben des Ausgabepuffers zwar sequentiell, aber unterbrochen von Leseoperationen

# Mehrere Disks



- **Problem:** S-/L-Köpfe einer Festplatte bewegen sich stets gemeinsam
- **Lösung:** nutze mehrere Festplatten (mit unabhängigen Köpfen)
  - Annahme: Disk-Controller, Hauptspeicher, Bus kommen mit höheren Transferraten klar
  - Resultat: Division aller Zugriffszeiten durch Festplattenanzahl
- Megatron 737 wie 747, aber nur 2 Platten → 4 Plattenoberflächen
  - Ersetzen eine Megatron 747 durch vier Megatron 737
  - Teilen *R* auf vier Festplatten auf
- **TPMMS – Phase 1**
  - Lesen: Von jeder Platte nur ¼ der Daten (1600 Blöcke)
    - Durch günstige Zylinderorganisation: *seek time* und Rotationslatenz 0
    - → 1600 Blöcke · 0,25 ms (mittlere Transferzeit) = 400 ms pro Füllung
    - → 16 Füllungen x 400 ms = 6,4 s
  - Schreiben: Jede Teilliste wird auf 4 Disks verteilt
    - Wie Lesen: 6,4 s
  - Zusammen nur 13 s (statt 52 s zuvor; bzw. statt 37 min bei zufälliger Anordnung)

# Mehrere Disks



**TPMMS – Phase 2**

- Verteilung nützt zunächst nichts
  - Immer wenn Block einer Teilliste abgearbeitet ist, wird nächster Block dieser Teilliste in Hauptspeicher geladen
  - → Erst wenn nächster Block vollständig geladen ist, kann Mischen fortgesetzt werden
- Trick für Lesen: Mischen kann fortgesetzt werden, bevor Block vollständig im Hauptspeicher geladen wurde (erstes Element genügt schon)
  - So können potenziell mehrere Blöcke parallel (jeweils einer pro Teilliste) geladen werden → Verbesserung, wenn diese auf unterschiedlichen Festplatten sind
  - Vorsicht: Sehr delikate Implementierung
- Schreiben des Outputs
  - Verwende mehrere Output-Blöcke (hier: 4)
  - Einer wird gefüllt während die anderen drei geschrieben werden (parallel, wenn Schreiben auf unterschiedliche Festplatten)
- Geschätzte Beschleunigung von Phase 2: Faktor 2 bis 3
  - Immerhin!

# Spiegelung



- Idee: Zwei oder mehr Festplatten halten identische Kopien
  - Mehr Sicherheit vor Datenverlust
  - Beschleunigter Lesezugriff (bei *n* Festplatten, bis zu *n* mal so schnell)
- TPMMS, Phase 2, Lesen: Trick bei mehreren Disks klappt nicht immer
  - Keine Verbesserung, falls Blöcke verschiedener Teillisten auf gleicher Festplatte liegen
  - Bei Spiegelung kann garantiert werden, dass immer so viele Blöcke unterschiedlicher Teillisten parallel gefüllt werden wie Spiegelungen vorhanden sind
- Weiterer Vorteil, auch ohne Parallelität (weniger als *n* Blöcke gleichzeitig)
  - Auswahl der Festplatte möglich, auf die zugegriffen wird
  - → Wähle die Festplatte, deren Kopf am dichtesten an relevanter Spur steht
- Anmerkungen
  - Teuer
  - Keine Beschleunigung des Schreibzugriffs

# Disk Scheduling



Idee: Disk-Controller entscheidet, welche Zugriffsanweisungen zuerst ausgeführt werden.

- Nützlich bei vielen kleinen Prozessen, je auf wenigen Blöcken
  - OLTP
  - Ziel: Erhöhung des Durchsatzes
- **Elevator Algorithmus**
  - Fahrstuhl fährt in Gebäude hoch und runter
    - Hält an Stockwerken an, wenn jemand ein- oder aussteigen will.
    - Dreht um, falls weiter oben/unten keiner mehr wartet.
  - Diskkopf streicht über Oberfläche einwärts und auswärts
    - Hält an Zylinder an, wenn es eine (oder mehrere) Zugriffsanweisung(en) gibt.
    - Dreht um, falls in jeweiliger Richtung keine Anweisungen mehr ausstehen.

# Elevator Algorithmus



Initial: Kopf steht bei 2000

Entspricht naiver FCFS-Strategie

| Angefragter Zylinder | Eintreffen der Anweisung | Zeitpunkt der Fertigstellung |
|---|---|---|
| 2000 | 0 | 4,42 (0,25 + 4,17) |
| 6000 | 0 | 13,84 (4,42 + [5 + 4,42]) |
| 14000 | 0 | 27,26 |
| 4000 | 10 | 42,68 |
| 16000 | 20 | 60,10 |
| 10000 | 30 | 71,52 |

| Angefragter Zylinder | Zeitpunkt der Fertigstellung |
|---|---|
| 2000 | 4,42 |
| 6000 | 13,84 |
| 14000 | 27,26 |
| 16000 | 34,68 |
| 10000 | 46,10 |
| 4000 | **57,52** |

Nur kleine Verbesserung, aber auch nur Mikro-Beispiel

Zahlen sind zum selber nachrechnen

# Elevator Algorithmus



- Verbesserung steigt mit durchschnittlicher Anzahl von wartenden Anweisungen.
  - So viele wartende Zugriffsanweisungen wie Anzahl Zylinder
    → Jeder Seek geht über nur wenige Zylinder
    → durchschnittliche *seek time* (bezogen auf wartende Zugriffsanweisungen) wird minimiert
  - Mehr Zugriffsanweisungen als Zylinder
    → mehrere Zugriffsanweisungen pro Zylinder
    → Sortierung um den Zylinder herum möglich
    → dadurch: Reduzierung der Rotationslatenzzeit
- Nachteil (falls Anzahl wartender Anweisungen zu groß)?
  - Wartezeiten für einzelnen Zugriffsanweisungen können sehr groß werden!

# Prefetching



**Idee:** Wenn man voraussagen kann, welche Blöcke in naher Zukunft gebraucht werden, kann man sie früh (bzw. während man sie sowieso passiert) in den Hauptspeicher laden.

- TPMMS, Phase 2, Lesen: 16 Blöcke für die 16 Teillisten reserviert
  - Viel Hauptspeicher frei
  - Reserviere zwei Blöcke pro Teilliste
    - Fülle einen Block, während der andere abgearbeitet wird
    - Wenn einer entleert ist, wechsele zum anderen
  - Aber: Laufzeit wird nicht verbessert

**Idee:** Kombination mit guter Spur-/ oder Zylinderorganisation

- TPMMS, Phase 1, Schreiben: Schreibe Teillisten auf ganze, aufeinanderfolgende Spuren / Zylinder
- TPMMS, Phase 2, Lesen: Lese ganze Spuren / Zylinder, wenn aus einer Liste ein neuer Block benötigt wird.

**Idee für das Schreiben analog:**

- Zögere Schreiboperationen hinaus bis ganz Spur / ganzer Zylinder geschrieben werden kann
- Verwende mehrere Ausgabepuffer → während einer auf Festplatte geleert wird, in anderen schreiben

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

Fünf Tricks

- Zylinderorganisation
- Verwendung mehrerer Festplatten
- Spiegelung
- Scheduling mit Elevator Algorithmus
- Prefetching

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

**Zylinderorganisation**

- Idee: Daten, die zusammen gelesen werden, auf gleichen Zylinder platzieren.
- Vorteil: Perfekt für 1.
- Nachteil: Hilft nicht für 2.

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

**Mehrere Disks (RAID 0, Striping)**

- Idee: Mehrere Disks mit unabhängigen Köpfen beheben das Problem der gemeinsamen Kopfbewegung
- Vorteil: Erhöht Schreib-/Leserate für 1. und 2.
- Problem: Operationen auf gleicher Disk werden nicht beschleunigt. Gesamtbeschleunigung nicht *n*-fach
- Nachteil: Teurer bei gleicher Diskkapazität (viele kleine Festplatten teurer als eine große Festplatte)

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

**Spiegelung (RAID 1)**

- Idee: Identische Kopien auf mehreren Festplatten
- Vorteile: Erhöht Leserate für 1. und 2.
  Beschleunigung ist *n*-fach
  Erhöhte Fehlertoleranz
- Nachteil: Kosten verdoppelt bzw. ver-*n*-facht

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

**Scheduling und Elevator Algorithmus**

- Idee: Disk-Controller entscheidet, welche Zugriffsanweisungen zuerst ausgeführt werden.
- Vorteil: Reduziert durchschnittliche Lese/Schreibzeit für 2.
- Problem: Funktioniert am besten bei vielen wartenden Anweisungen
- In diesem Fall aber auch Erhöhung der durchschnittlichen Wartezeit

# Zusammenfassung



Zwei Arten von Anwendungen

1. Viele Blöcke werden in bekannter Folge gelesen oder geschrieben. Nur ein Prozess. (TPMMS, Phase 1)
2. Viele kleine, parallele, unvorhersagbare Prozesse (ähnlich TPMMS, Phase 2)

**Prefetching**

- Idee: Künftig benötigte Blöcke frühzeitig laden
- Vorteil: Beschleunigt 2.; es ist bekannt, welche Daten benötigt werden, es ist nur unklar wann.
- Nachteil: Benötigt zusätzlichen Hauptspeicher

# Übersicht



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

# Umfrage



Sind Ihnen schon einmal Daten abhanden gekommen?

*Head Crash*

# Fehlerklassifikation



1. Transaktionsfehler
   - Führt zu Transaktionsabbruch
   - Fehler in der Anwendung (division by zero)
   - **`abort`** Befehl
   - Abbruch durch DBMS (Deadlock)
2. Systemfehler
   - Absturz in DBMS, Betriebssystem, Hardware
   - Daten im Hauptspeicher werden zerstört
3. Medienfehler
   - *Head crash*, Controller-Fehler, Katastrophe
   - Daten auf Disk werden zerstört
   - Jetzt mehr…

# Diskfehlerarten (Medienfehler)



1. Sporadischer Fehler
   - Erfolgloser Lese- oder Schreibversuch
   - Wiederholte Versuche führen zu Erfolg
2. Medienverfall
   - Korrupte Bits auf Disk
   - Wiederholte Leseversuche führen nicht zum Erfolg
3. Schreibfehler
   - Schreibanweisung kann nicht ausgeführt werden
   - Z.B. Stromausfall
4. Diskausfall
   - Gesamte Disk kann nicht gelesen werden
   - Plötzlich und permanent

Jetzt: Prüfsummen Stable Storage

Gleich: RAID

# Sporadische Fehler



- Idee: Redundante Bits speichern Status-Informationen
  - „correct“ / „incorrect“
- Leseanweisung gibt Paar *(w, s)* zurück
  - *w*: Die Daten
  - *s*: Der Status (Lesen erfolgreich / nicht erfolgreich)
- Falls *s* „incorrect“ ist: bis zu 100 Wiederholungsversuche
- *s* kann selbst fehlerhaft sein
  - Beliebig genau durch weitere Redundanz
- Nach Schreibanweisung analog
  - *s* lesend prüfen
  - Wenn *s* „correct“, so nehmen wir erfolgreiches Schreiben an

# Prüfsummen / Paritätsbit



Idee: Jeder Sektor erhält einige Bits zur Speicherung einer Prüfsumme (*checksum*)

- Prüfsumme ist abhängig von Daten im Sektor
- Problem: Prüfsumme könnte zufällig zu inkorrekten Daten passen
  - Mehr Bits verringern diese Wahrscheinlichkeit
- Beispiel: Paritätsbit (*parity bit*) als Prüfsumme
  - Ungerade Anzahl 1en → Paritätsbit ist 1
  - Gerade Anzahl 1en → Paritätsbit ist 0

→ Anzahl 1en im gesamten Sektor ist immer gerade!

  - 01101000: parity bit 1 → 011010001
  - 11101110: parity bit 0 → 111011100
- Jeder 1-Bit Fehler kann leicht erkannt werden (Anzahl 1en ist ungerade)
- Erkennen mehrerer Fehler
  - Wahrscheinlichkeit 0.5, dass Anzahl 1en dennoch gerade und Fehler nicht erkannt wird
- Idee: 8 Paritätsbits – eines für jedes Bit der Bytes
  - Wkt. Fehler in einem Bit zu übersehen: 1/2
  - Wkt. einen Fehler zu übersehen (bezogen auf gegebene Bytes): $(1/2)^8 = 1 / 256$
- Allgemein: *n* Paritätsbits → Wahrscheinlichkeit $1/2^n$

# Stable Storage



- Problem: Prüfsummen lediglich fehlererkennend, helfen aber nicht Fehler zu **beheben**.
- Problem: Bei fehlerhaftem Schreiben überschreibt man auch alte Daten.
  - Beispiel: Kontostandveränderung
  - Nach fehlerhaften Schreiben: alten und neuen Kontostand verloren
- Idee: Sektor $X$ wird durch Paar ($X_L$, $X_R$) repräsentiert.
  - Linke Kopie $X_L$
  - Rechte Kopie $X_R$
  - Annahme: Durch Prüfsummen können Lese- und Schreibfehler in $X_L$ und $X_R$ **erkannt** werden.

# Stable Storage Protokolle



- Schreibprotokoll
  1. Schreibe Wert für $X$ auf $X_L$.
  2. Prüfe (mittels Prüfsumme) auf Korrektheit.
  3. Falls inkorrekt: Wiederhole 1. und 2. $n$-mal
  4. Falls immer noch inkorrekt: Medienfehler; neuen Sektor allozieren und wiederholen
  5. Wiederhole 1. – 4. auf $X_R$.
- Leseprotokoll
  1. Lese Wert von $X_L$.
  2. Falls inkorrekt, wiederhole 1. $n$-mal
  3. Falls immer noch inkorrekt, wiederhole 1. und 2. mit $X_R$.

# Stable Storage – Medienverfall



Szenario 1: Medienverfall

- Daten können immer vom jeweils anderen Sektor gelesen werden.
- Falls $X_R$ defekt
  - Leseprotokoll merkt nichts
  - Schreibprotokoll wird Fehler erkennen
- Falls $X_L$ defekt
  - Leseprotokoll springt zu $X_R$
  - Schreibprotokoll bemerkt Fehler ebenfalls
- Beide defekt: Unwahrscheinlich

# Stable Storage – Schreibfehler



Szenario 2: Schreibfehler

- Stromausfall während Schreiboperation
  - Wert im Hauptspeicher ist verloren
  - Wert ist erst zur Hälfte geschrieben (inkorrekt)
- Alter Wert von $X$ kann immer noch gelesen werden
  - Fehler trat beim Schreiben auf $X_L$
    - Status von $X_L$ ist inkorrekt
    - Status von $X_R$ ist korrekt
      - Kann verwendet werden, um $X_L$ zu reparieren
  - Fehler trat nach Schreiben auf $X_L$
    - $X_L$ mit Status korret
    - $X_R$ sollte mit Wert von $X_L$ repariert werden.

# Übersicht



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6

# Diskausfälle



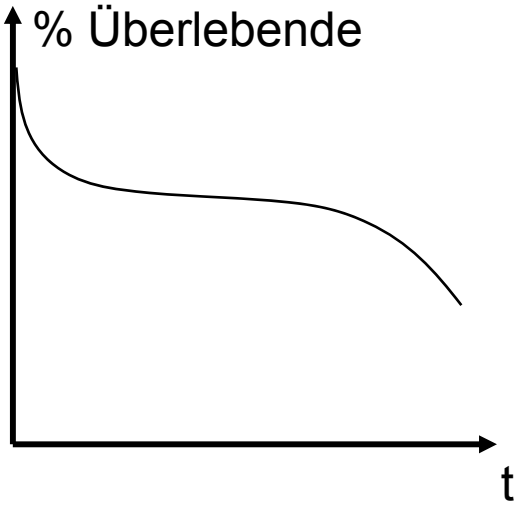
Generell: **Redundanz** zur Vermeidung von Datenverlust

- RAID: **R**edundant **A**rrays of **I**nexpensive **D**isks
  - 1988: Patterson, Gibson, Katz
- RAID: **R**edundant **A**rrays of **I**ndependent **D**isks
- „*Commodity Hardware*“ mit Software zu schnellen und fehlertoleranten Systemen zusammenbauen.

# Fehlertoleranz

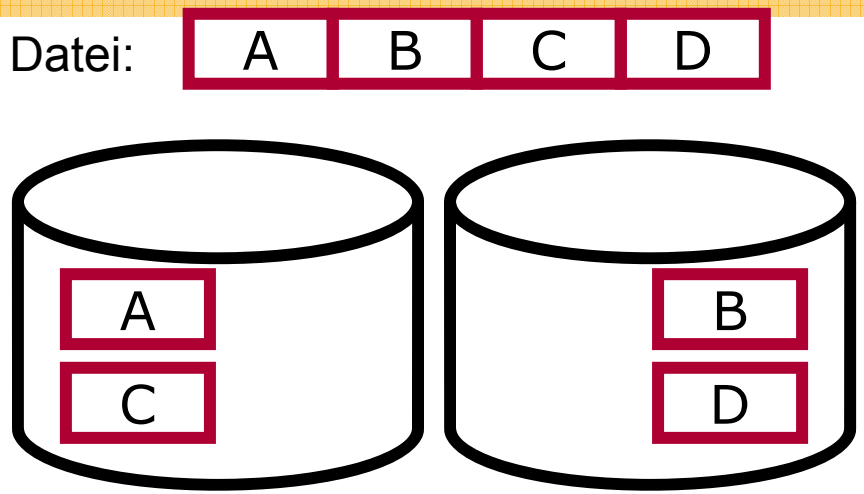


- Diskarray mit *N* Festplatten
  - Mean-time-to-failure (MTTF)
    - Moderne Festplatten: ca. 10 Jahre
  - Mean-time-to-repair (MTTR)
  - Mean-time-to-data-loss (MTTDL)
  - Ohne Fehlertoleranzmechanismen **erhöhte** Ausfallwahrscheinlichkeit
    - MTTDL = MTTF / *N*



# RAID 0 – Striping



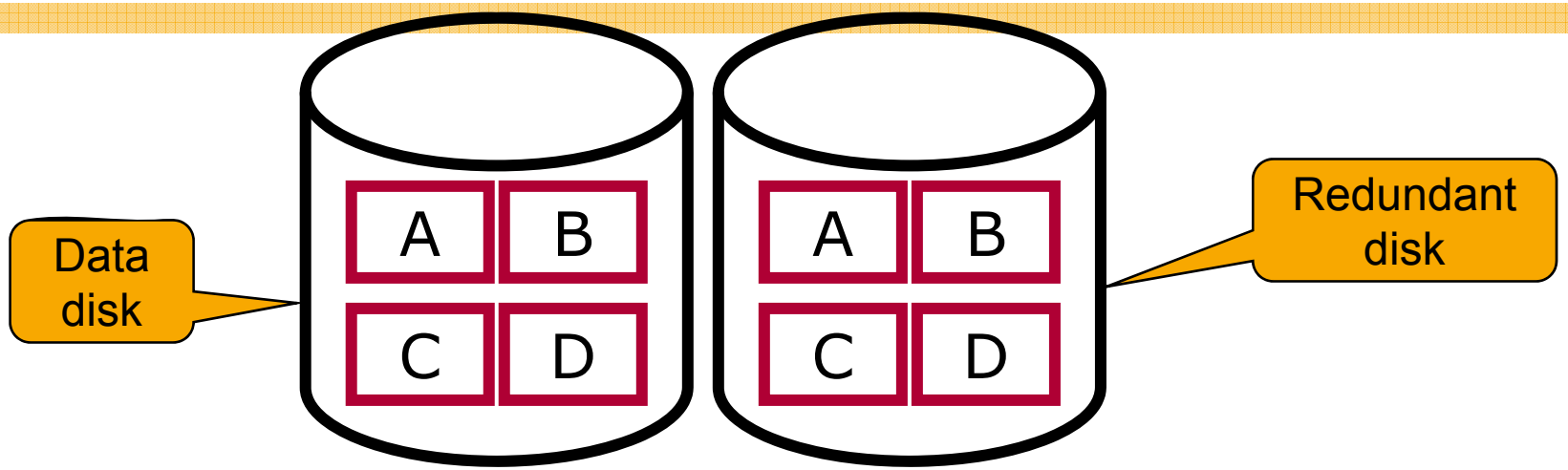



- Entspricht Beschleunigungsidee „Mehrere Disks“ (s.o.)
- Doppelte Bandbreite beim sequentiellen Lesen der Datei
  - Zyklische Verteilung der Blöcke, wenn alle Blöcke mit gleicher Häufigkeit gelesen/geschrieben werden (round robin)
  - Alternative: Verteilung nach Zugriffshäufigkeit (Optimierungsproblem)
    - Zugriffshäufigkeit aber i.d.R. nicht bekannt
- Nachteil 1: Keine Beschleunigung für Lesen eines einzelnen Blocks
- Nachteil 2: Dateiverlust wird immer wahrscheinlicher, je mehr Disks man verwendet.

Quelle: Folien Ulf Leser (HU)

# RAID 1 – Mirroring






- Entspricht Beschleunigungsidee „Mirroring“ (s.o.)
- Datensicherheit durch Redundanz aller Daten
- Aber: Doppelter Speicherbedarf
- Lastbalancierung beim Lesen (wie RAID 0)
- Außerdem „konkurrierendes“ Lesen einer 1-Block Datei möglich.
  - Der schnellere Kopf gewinnt.
- Beim Schreiben müssen beide Kopien geschrieben werden.
  - Kann parallel geschehen

Quelle: Folien Ulf Leser (HU)

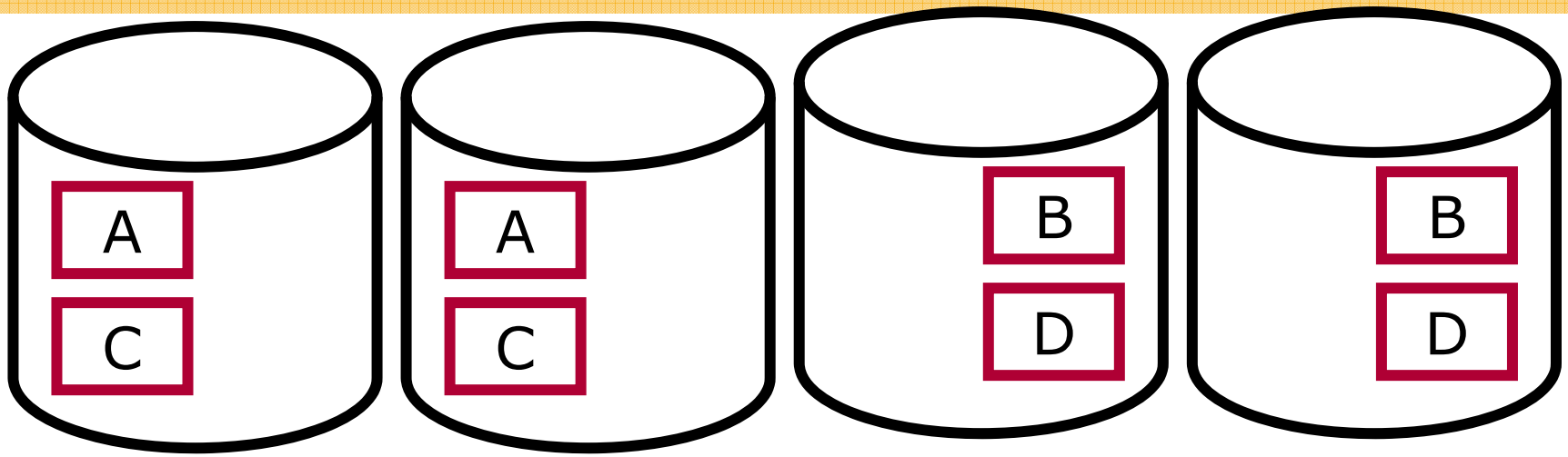
# RAID 1 – Datenverlust (Beispiel)



- 2 Disks mit MTTF = 10y
  - Wkt. 10%, dass Disk in gegebenem Jahr ausfällt
- Bei Ausfall einer Disk
  - Disk ersetzen und Dateien von redundanter Disk kopieren.
  - Mögliches Problem: Redundante Disk fällt während dieses Kopiervorgangs aus.
- Wahrscheinlichkeit für Ausfall?
  - 3h für Kopieren = 1/8 Tag = 1/2920 Jahr
  - Wkt. eines Ausfalls in einem Jahr = 1/10
  - Wkt. dass Disk während des Kopierens ausfällt = 1/29.200
  - Eine der beiden Disks fällt alle 5 Jahre aus.
    - Bei jedem 29.200ten Mal führt dies zu Speicherverlust
  - MTTDL mit RAID 1: 5 x 29.200 = 146.000 Jahre
- Annahme?
  - Unabhängigkeit der Diskfehler
  - Stimmt diese Annahme?

# RAID 0 + 1 – Mirrored Striping



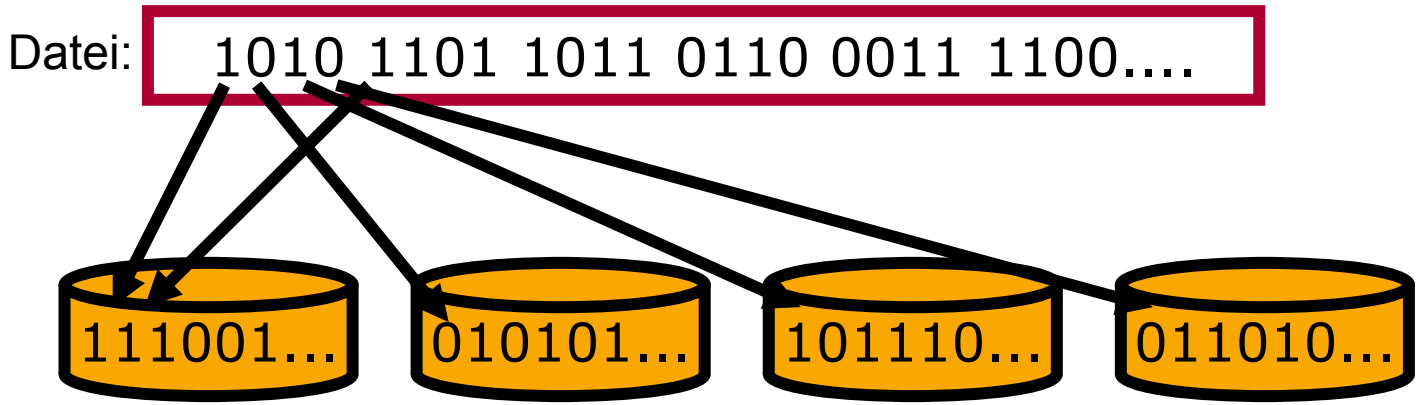



- Kombiniert RAID 0 und RAID 1
- Doppelter Speicherbedarf
- Erhöhte Ausfallsicherheit (durch RAID 1)
- Weiter erhöhte Lesegeschwindigkeit für Dateien
  - Datei kann von vier Disks parallel gelesen werden.
- Lesen eines einzelnen Blocks kaum schneller als ohne RAID
- Schreibgeschwindigkeit einer Datei verdoppelt sich.

Quelle: Folien Ulf Leser (HU)

# RAID 2 - Striping auf Bit-Ebene



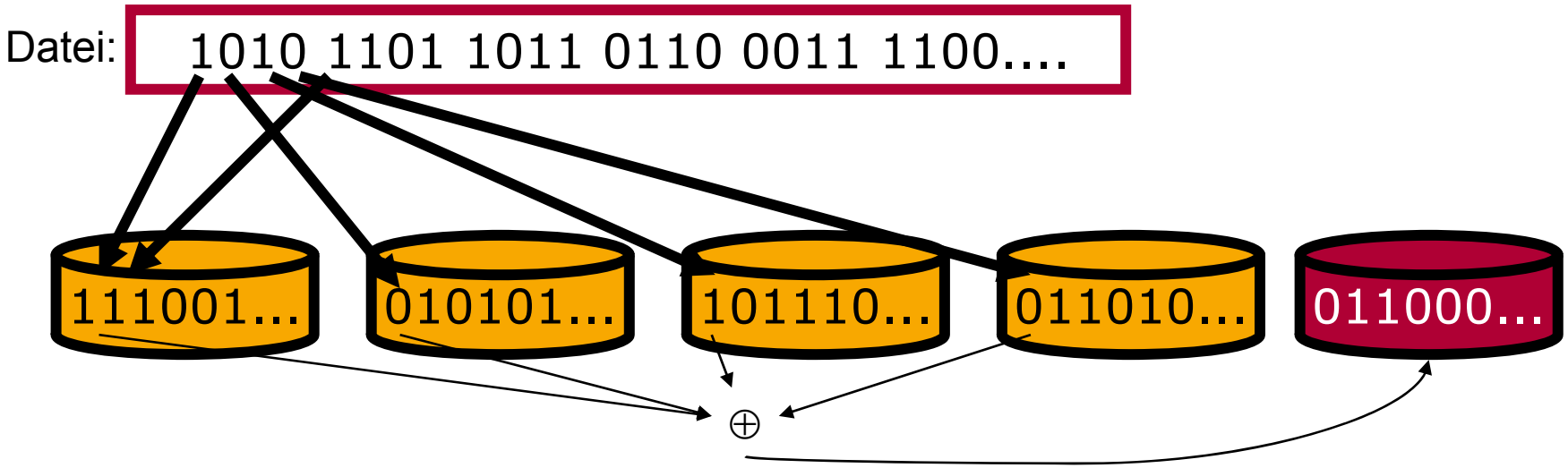



- Striping auf Bit- (oder Byte-) statt Blockebene
- Idealerweise höherer Durchsatz schon beim Lesen einzelner Blöcke
  - Aber: Lesen eines Sektors dauert i.d.R. genauso lange wie Lesen 1/8 Sektors
  - Also muss Aufteilung von Blöcken auf Sektoren beachtet werden
    - Typisch: 8-16 KB Blöcke, 512 Byte Sektoren = 16-32 Sektoren pro DB Block
- Keine Beschleunigung für mehrere parallele Leseoperationen
  - Jedes Lesen/Schreiben braucht alle Disks
- Verschlechterte Ausfallsicherheit

Quelle: Folien Ulf Leser (HU)

# RAID 3 – Striping mit Parity Blocks



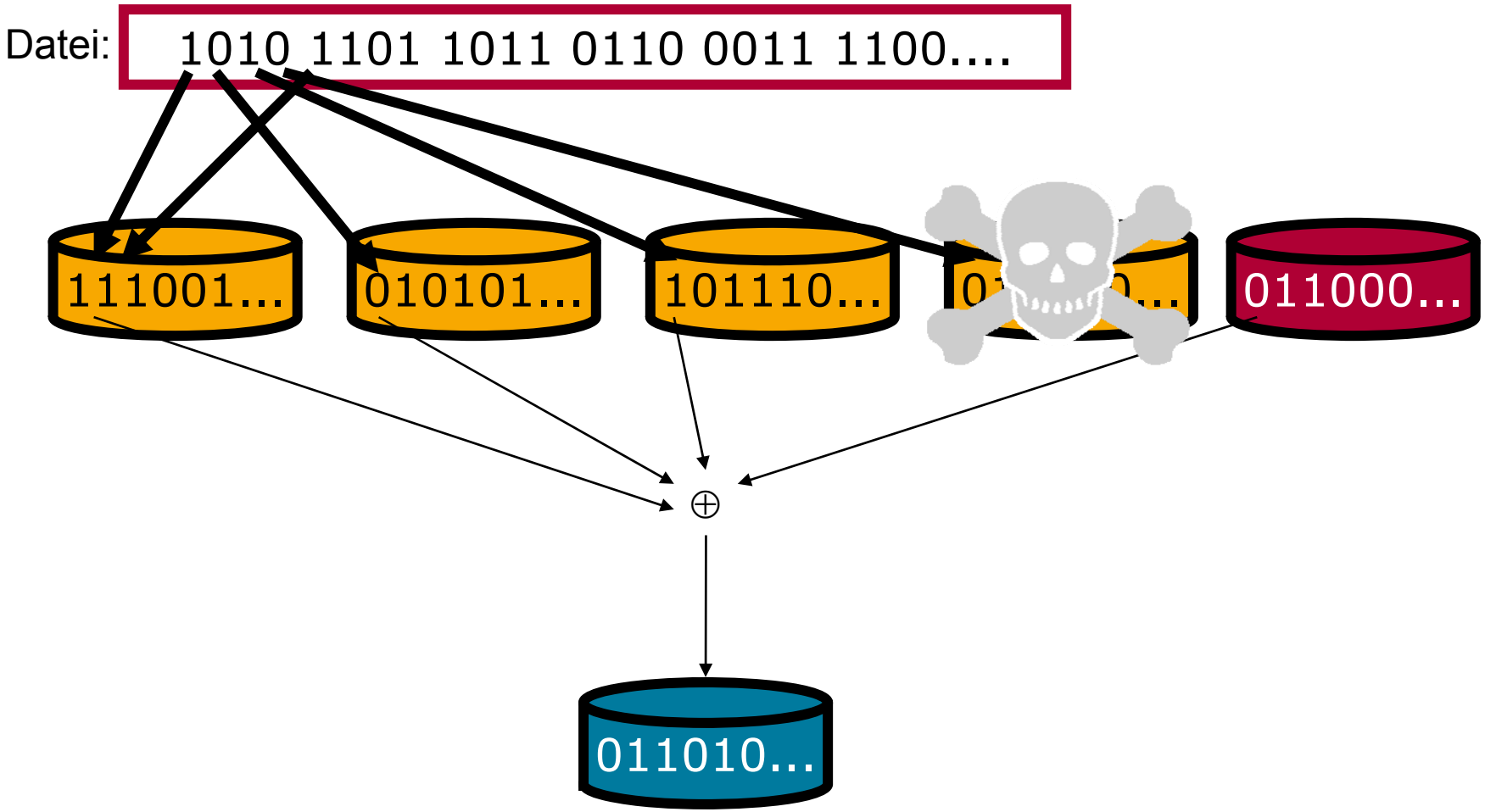



- Striping auf Bit- (oder Byte-) Ebene (wie RAID 2)
- Zusätzliche Disk für Parität der anderen Disks
  - Parität = bit-weises xor ⊕
  - Ausfall einer Disk kann kompensiert werden.
- Lesen/Schreiben eines Blocks erfordert Zugriff auf alle Disks.
- Deutlich weniger Platzbedarf als RAID 1, trotzdem Ausfallsicherheit

Quelle: Folien Ulf Leser (HU)

# RAID 3 - Diskausfall

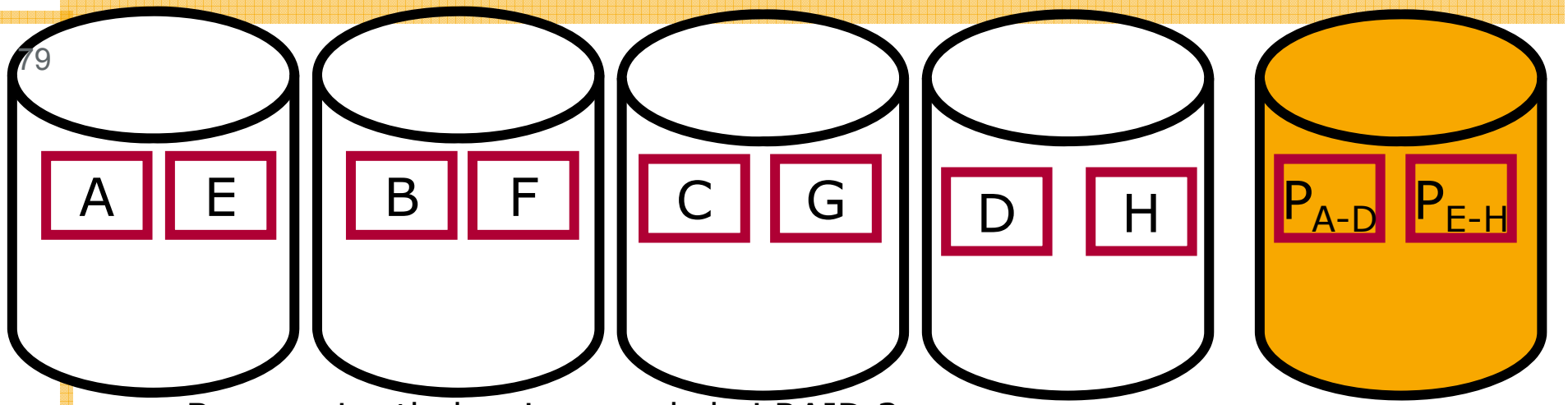





Quelle: Folien Ulf Leser (HU)

# RAID 4 – Block Striping mit Parity



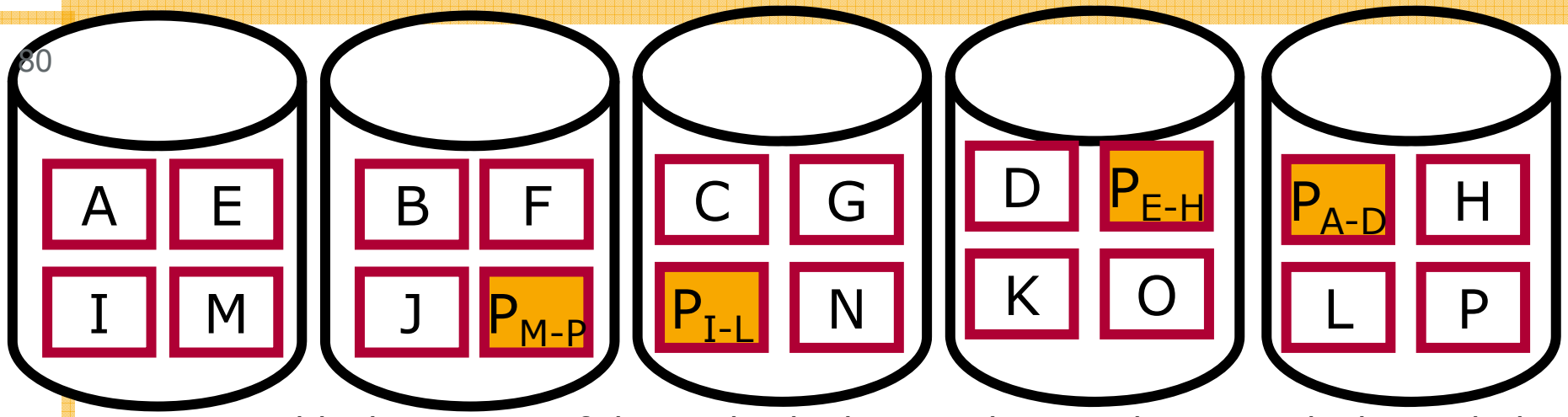



- Bessere Lastbalancierung als bei RAID 3
  - Verschiedene Prozesse können parallel von verschiedenen Disks lesen.
- Damit: Paritätsdisk ist Flaschenhals
  - Bei jedem Schreiben muss darauf zugegriffen werden.
  - Bei jedem Lesen muss u.U. darauf zugegriffen werden (zur Kontrolle).
  - Bei Modifikation von Block A zu A‘: $P'_{A-D} := P_{A-D} \oplus A \oplus A'$
- D.h. bei einer Änderung von Block A muss der alte Zustand von A und der alte Paritätsblock gelesen werden und der neue Paritätsblock und der neue Block A‘ geschrieben werden.

Quelle: Folien Ulf Leser (HU)

# RAID 5 – Block Striping mit verteilter Parity






- Parityblock immer auf der Disk, die keinen der zugehörigen Blöcke enthält
- Wesentlich bessere Lastbalancierung als bei RAID 4
  - Paritätsdisk kein Flaschenhals mehr
  - Schreiben erfordert *n*-1 Disks für die Daten und 1 Disk für Parity
- Wird in der Praxis häufig eingesetzt.
  - Typischerweise langsamer als RAID 0+1
  - Dafür deutlich Platz sparender
- Guter Ausgleich zwischen Platzbedarf und Leistungsfähigkeit

Quelle: Folien Ulf Leser (HU)

# RAID 6 – Hamming Codes



- Wie RAID 5 aber mit mehr Kontrollinformationen
- Erlaubt Wiederherstellung auch nach mehreren Diskausfällen.
- Höchste RAID-Stufe

- Nicht hier
  - Siehe z.B. http://de.wikipedia.org/wiki/Hamming_Code

- Weitere RAID Architekturen: Wikipedia

# Zusammenfassung



| | Block-Striping | Bit-Striping | Kopie | Parität | Parität, dedizierte Disk | Verteilte Parität | Erkennen mehrerer Fehler |
|---|---|---|---|---|---|---|---|
| RAID 0 | X | | | | | | |
| RAID 1 | | | X | | | | |
| RAID 0+1 | X | | X | | | | |
| RAID 2 | | X | | | | | |
| RAID 3 | | X | | X | X | | |
| RAID 4 | X | | | X | X | | |
| RAID 5 | X | | | X | | X | |
| RAID 6 | X | | | X | | | X |

- RAID 1 und vor allem RAID 5 sind weit verbreitet
  - RAID1: Einfach, schnell zu realisieren, erhöhte Ausfallsicherheit und Geschwindigkeit, aber verdoppelter Platzbedarf (billig)
  - RAID5: Relativ komplex, erhöhte Ausfallsicherheit und Geschwindigkeit, nur geringfügig erhöhter Platzbedarf; benötigt mindestens 3 Disks

# Rückblick



- Speicherhierarchie
- Disks
- Effiziente Diskoperationen
- Zugriffsbeschleunigung
- Diskausfälle
- RAID 0 – 6