Datenbanksysteme

Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Realisierung von Operatoren und Physische Optimierung

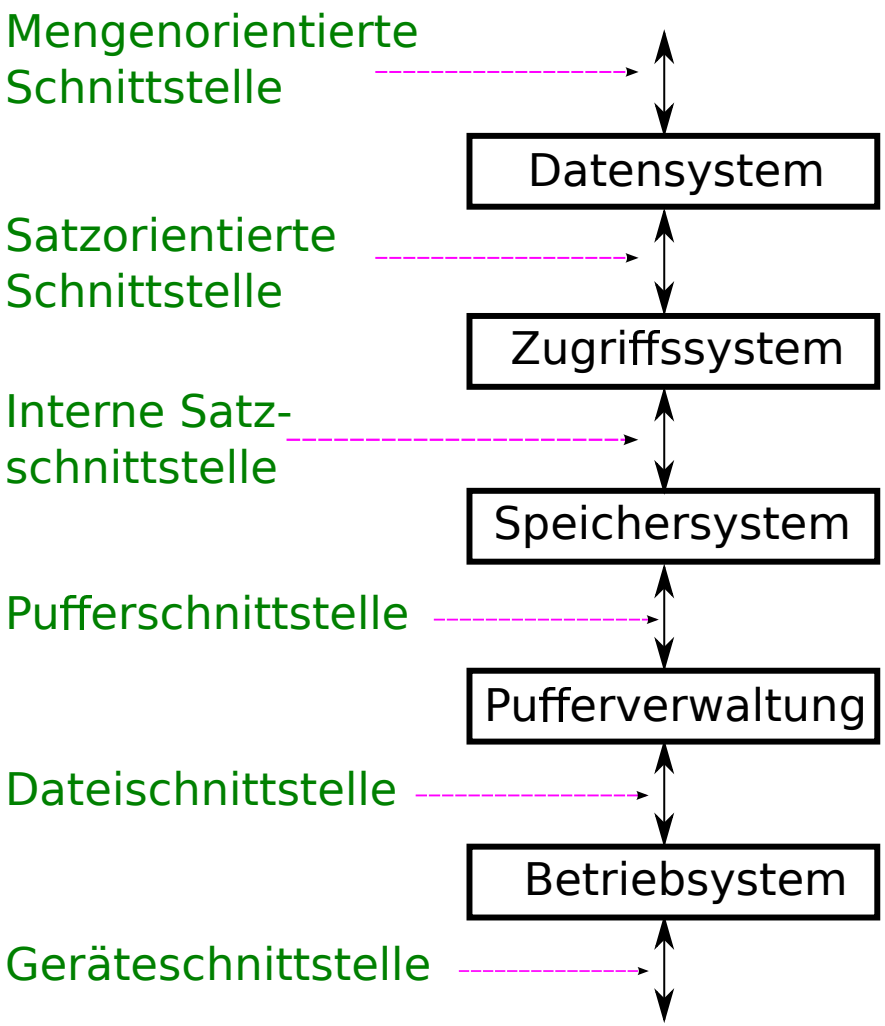
Bislang

- Logische Algebraoperatoren
- Beschreiben was berechnet wird (z.B. Join), aber nicht wie (Algorithmus).

Jetzt: Physische Optimierung

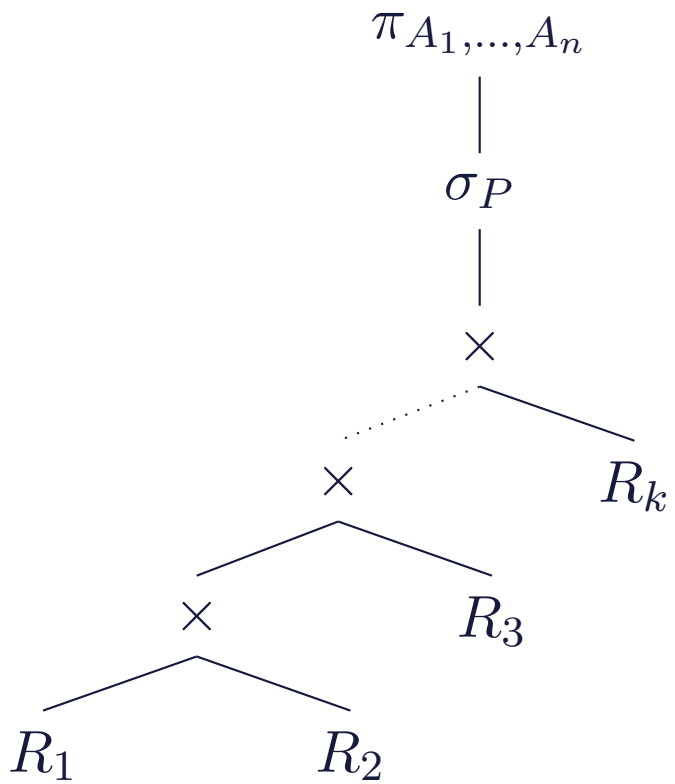
- Welche Kosten fallen an?
- Wie kann auf Daten zugegriffen werden?
- Welche Implementierungen gibt es für verschiedene Operatoren?
- Iteratorkonzept (Open, Next, Close, ...)

5 Schichtenmodell

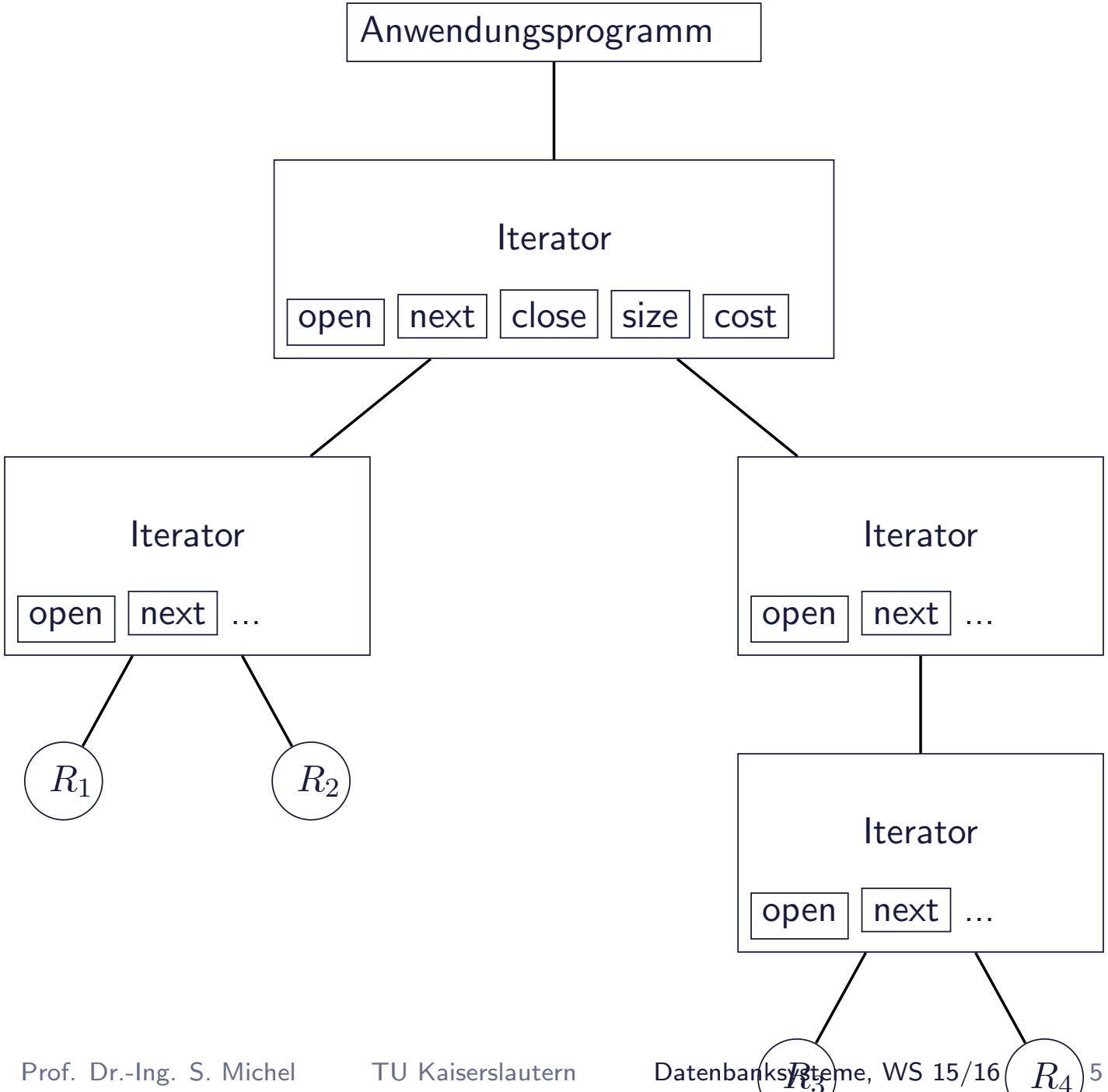


Anfrageverarbeitung: Übersetzung von SQL in Baum aus Operatoren

select A_1, ..., A_n
from R_1, ..., R_k
where P;



Was sind Operatoren, wie läuft Verarbeitung ab und wie werden Operatoren Implementiert?


Anwendungsprogramm
Iterator
open
next
close
size
cost
Iterator
open
next
...
Iterator
open
next
...
R_1
R_2
Iterator
open
next
...
R_3
R_4

Iterator

- **Open:** Konstruktor, initialisiert, öffnet die Eingabe
- **Next:** Liefert das nächste Ergebnis
- **Close:** Schließt die Eingabe
- **Cost und Size:** Geben Informationen über die geschätzten (!) Kosten

Empfehlenswert: Übersichtsartikel von G. Graefe: Query Evaluation Techniques for Large Databases, ACM Computing Surveys, 1993, volume 25, number 2, pages 73-170.

Pull-basierte Verarbeitung

- **Operatoren werden in Form von Iteratoren implementiert**
- **Datenfluss hierbei ist "von unten nach oben"**
- **Konsument-Produzent Verhältnis**
- Der konsumierende Iterator bezieht Tupel von seinen Eingaben, die ebenfalls Iteratoren sind, durch deren `next()` Schnittstelle

- **Im Allgemeinen werden Zwischenergebnisse nicht explizit materialisiert.**

Anmerkung: Push-basierte Verarbeitung

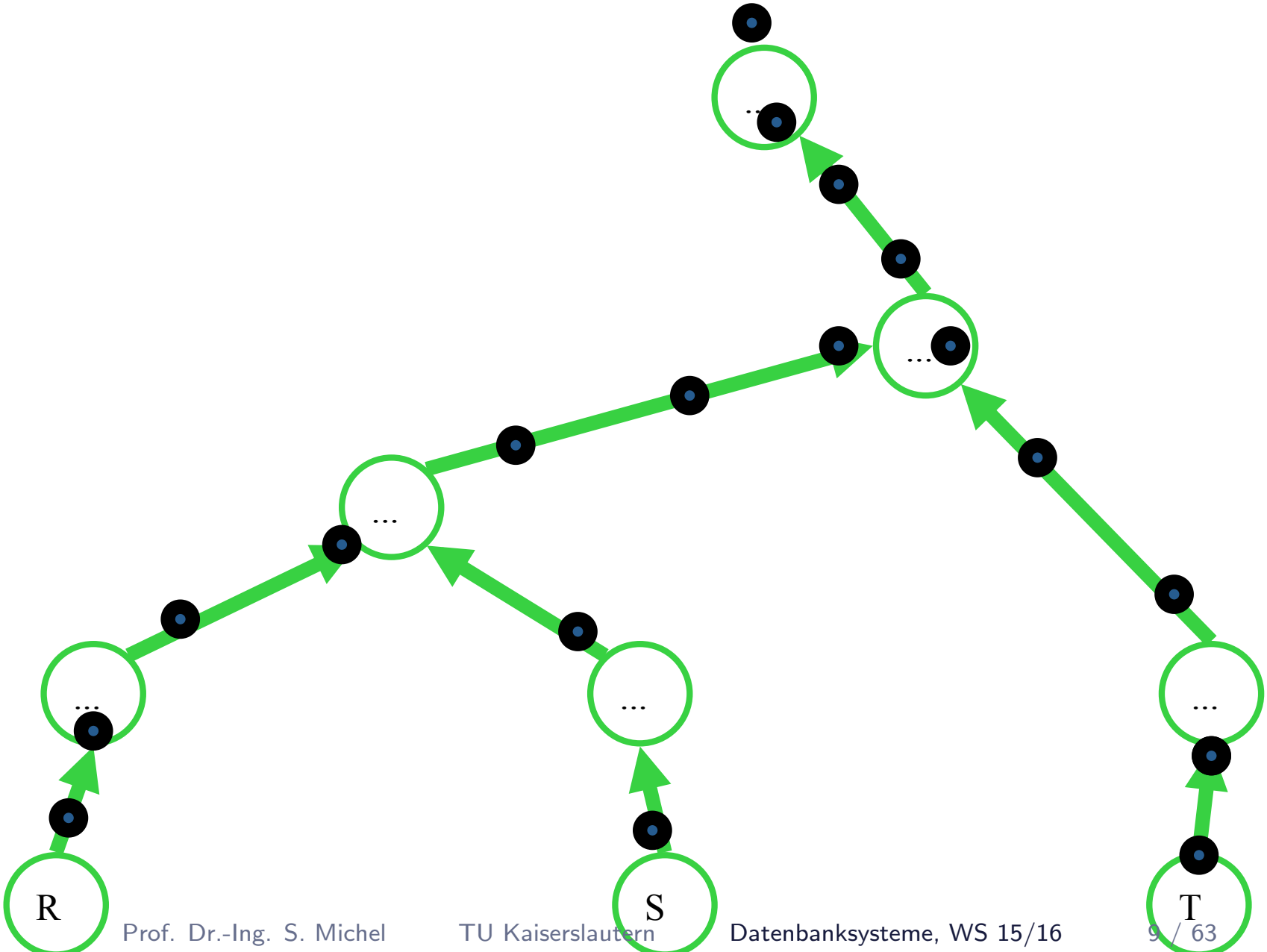
Es gibt insbesondere bei Systemen zur Datenstromverarbeitung die sogenannte Push-basierte Verarbeitung. Dort registrieren sich "Konsumenten" an Datenquellen oder anderen Operatoren. Falls diese neue Daten haben, werden die Daten an die Konsumenten weiter gereicht (aktiv).

Blockierende Operatoren

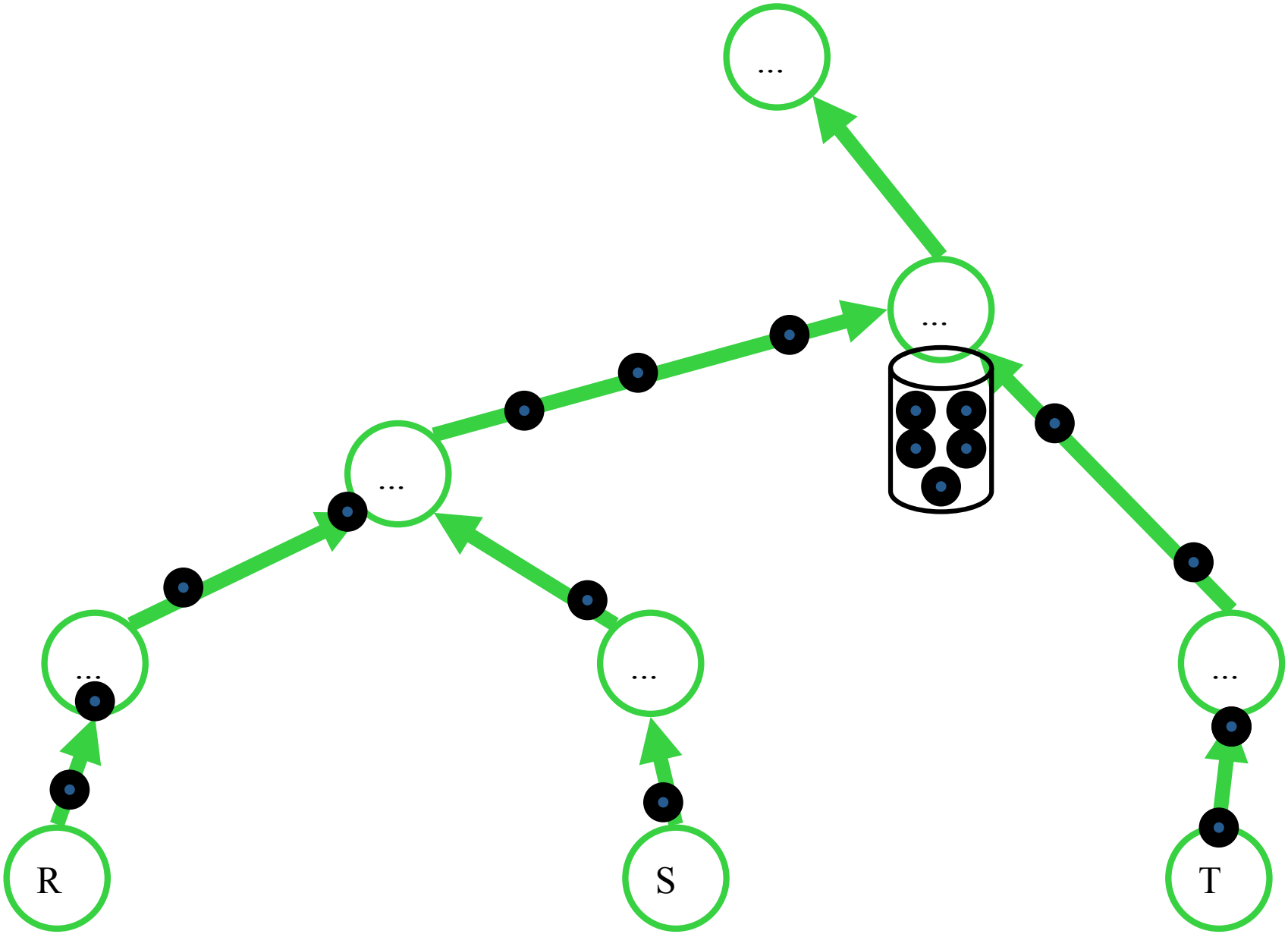
Idealerweise

- **Operatoren blockieren den Datenfluss nicht**
- D.h. beim Aufruf von `next()` werden darunter liegende Operatoren angefragt via next und das Ergebnis direkt weiter geleitet. Nur wenige Tupel werden dabei gelesen.
- **Erlaubt Pipelining**
- Im Gegensatz dazu: Operatoren, die den Datenfluss "blockieren", sogenannte Pipeline-Breaker

Pipelining vs. Pipeline-Breaker



Pipelining vs. Pipeline-Breaker



Pipeline-Breaker

- Sortieren
- Duplikate eliminieren (unique, distinct)
- Aggregation: min, max, avg, ...
- Joins (je nach Implementierung)
- Union (je nach Implementierung)

Implementierung von Operatoren

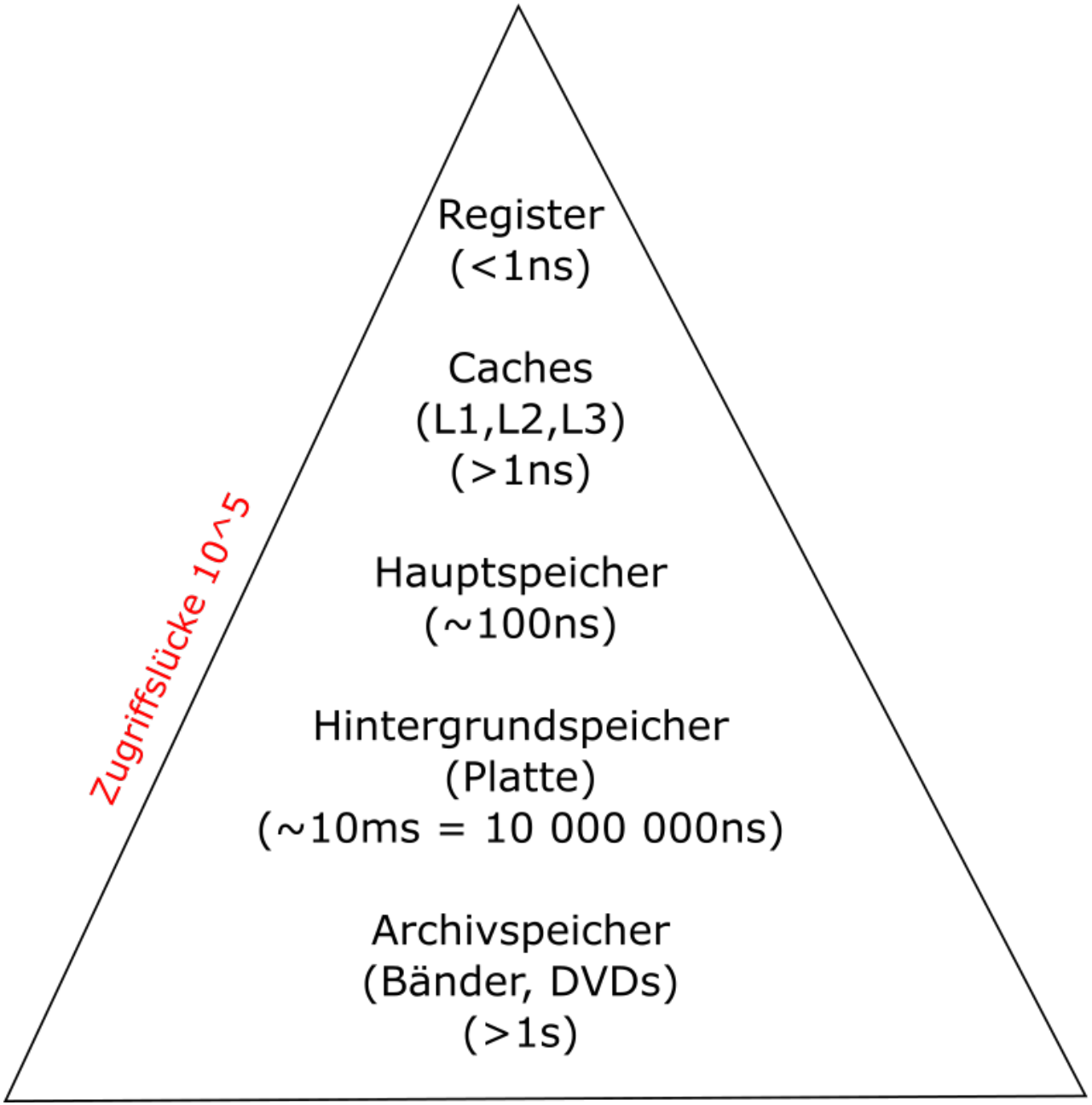
- **Bei der Erzeugung eines physischen Anfrageplans muss entschieden werden wie genau die Anfrage ausgeführt werden soll**

- Welche Implementierungen gibt es für die diversen Operatoren?
- Welche Implementierung ist effizienter?
- Und wie kann dies überhaupt berechnet/vorhergesagt werden? (Kostenmodelle)
- Gibt es Indexstrukturen, die ausgenutzt werden können?
- Falls ja, macht es auch Sinn einen Index zu benutzen?
- ...

Was kostet wieviel?

- L1 cache reference 0,5 ns
- L2 cache reference 7 ns
- Main memory reference 100 ns
- Compress 1K bytes with Zippy 10 000 ns
- Send 2K bytes over 1 Gbps network 20 000 ns
- Read 1 MB sequentially from memory 250 000 ns
- Round trip within same datacenter 500 000 ns
- Disk seek 10 000 000 ns
- Read 1 MB sequentially from network 10,000,000 ns
- Read 1 MB sequentially from disk 30 000 000 ns
- Send packet CA $\rightarrow$ Netherlands $\rightarrow$ CA 150 000 000 ns

Numbers by Jeff Dean (Google)



Analogie

Kopf
(1 Min)

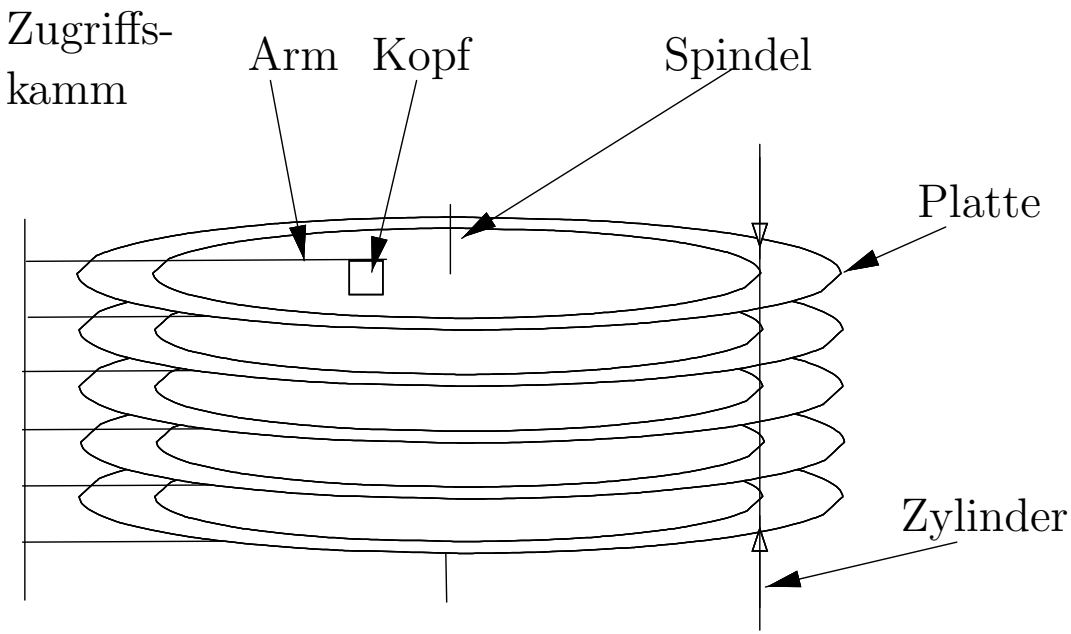
Raum
(10 Min)

Stadt
(1 Std)

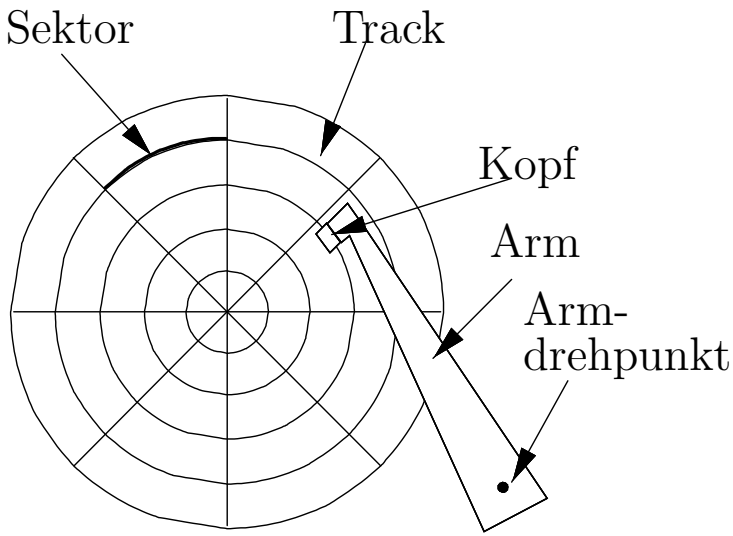
Pluto
(2 Jahre)

Andromeda
(2000 Jahre)

Aufbau einer (klassischen) Festplatte



a. Seitenansicht



b. Draufsicht

Aufbau Festplatte (Tracks, Sektoren, Zonen)

- **Track:** Teil eines Zylinders auf einer Platte
- **Sektor:** Teil eines Tracks. Anzahl Sektoren pro Track ursprünglich gleich für alle Tracks
- Aber, auf Zylinder/Tracks weiter "außen" passen mehr Sektoren. Daher, aktuelle Hardware, variable Anzahl von Tracks: äußere Tracks haben mehr Sektoren als innere Tracks; Zylinder sind in Zonen unterteilt.

Blöcke

- Aka. Sektoren, Aka. physical Record. Kleinste Transfereinheit bei Block-Storage-Devices. Seit wenigen Jahren typischerweise 4KB oder sonst (historisch) 512 Byte groß.
- **Achtung**, es gibt auch Unterschiede zu Blöcken bzw. Seiten des Dateisystems, dort kann ein Block auch mehrere Festplattenblöcke umfassen.

Beispiel

Die Festplatte SAMSUNG HD103SJ, die in meinem Desktop PC im Büro eingebaut ist hat laut (Linux tool) `hdparm -i` (oder `hdparm -I` für mehr Details):

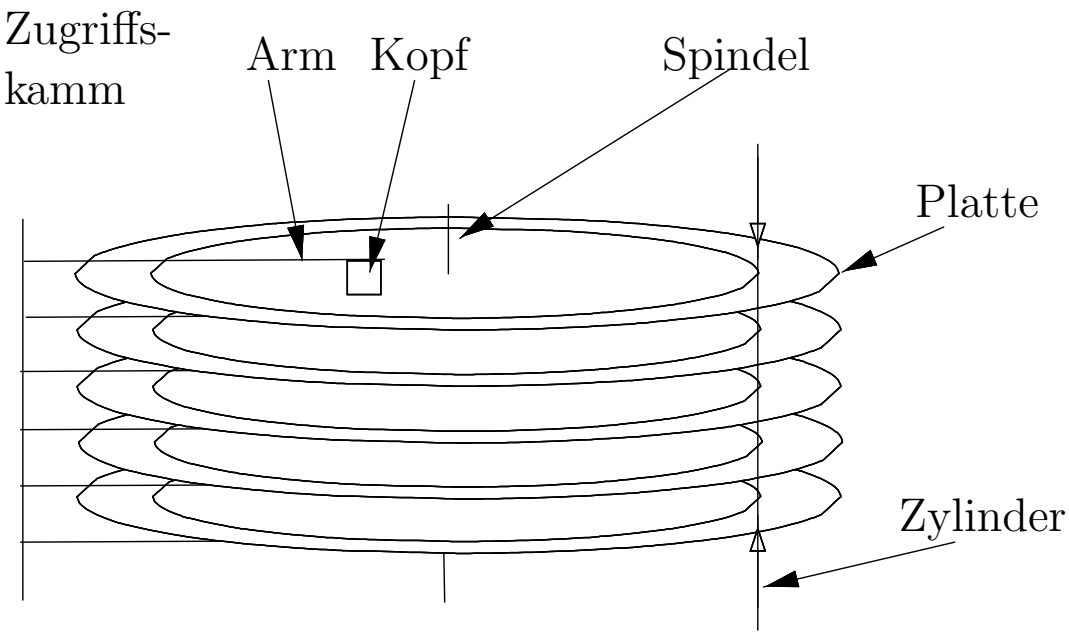
- 1953525168 Sektoren
- a 512 Bytes.

Das macht: 1 TB

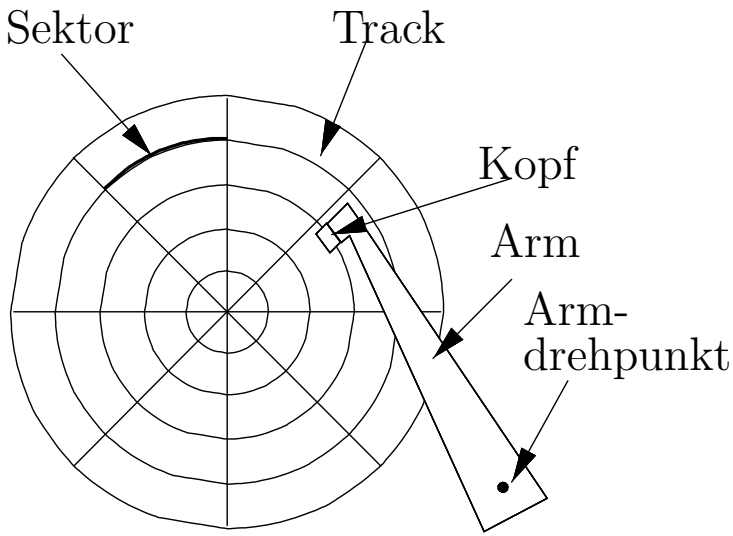
Ausgabe (Auszug) von hdparm -I /dev/sda

```
CHS current addressable sectors:   16514064
LBA    user addressable sectors:  268435455
LBA48  user addressable sectors: 1953525168
Logical  Sector size:                   512 bytes
Physical Sector size:                   512 bytes
device size with M = 1024*1024:      953869 MBytes
device size with M = 1000*1000:     1000204 MBytes (1000 GB)
```

Aufbau einer (klassischen) Festplatte



a. Seitenansicht



b. Draufsicht

Sektoren adressieren

- Physische Adresse: Zylinder Nummer (c), Kopf Nummer (h), Sektor Nummer (s)
- Logische Adresse: LBN/LBA (logical block number/address)

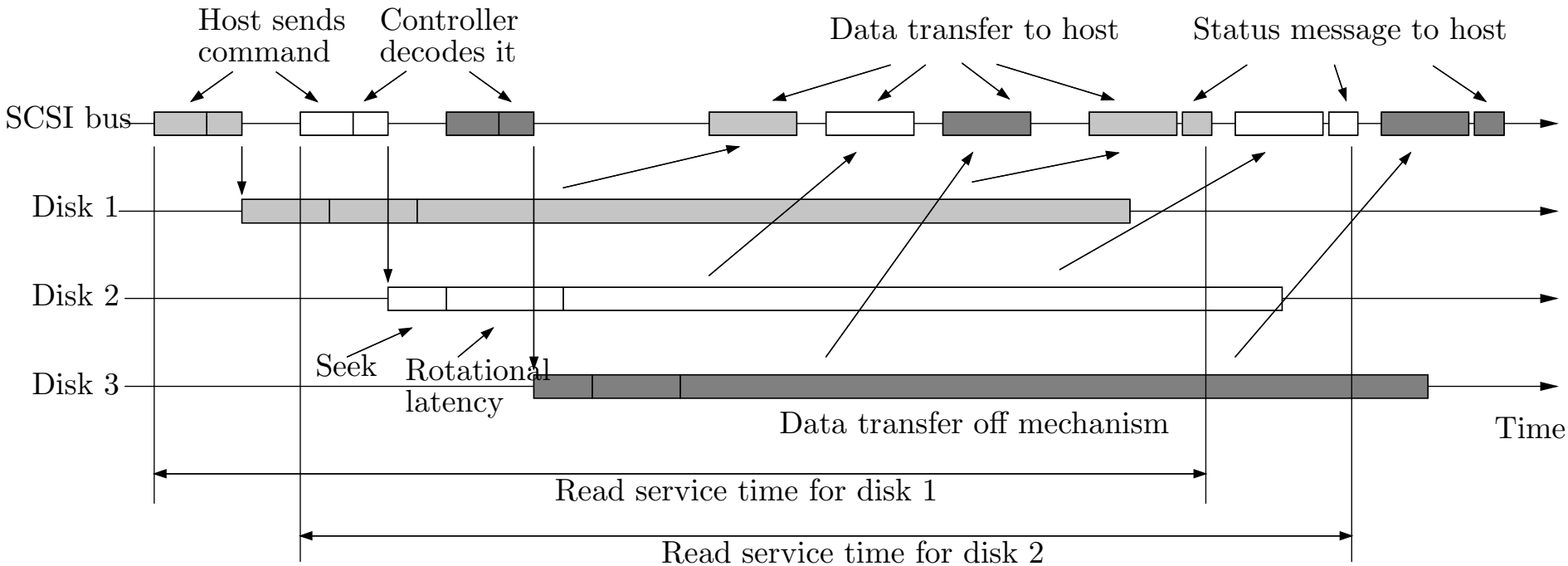
CHS vs. LBA

- **Cylinder-head-sector** (CHS Adressierung formals benutzt)
- Aktuell: LBA (**Logical Block Addressing**)
- Umrechnung: $LBA = (c * Nheads + h) * Nsectors + (s - 1)$, wobei: $Nheads =$ Anzahl der Leseköpfe, $Nsectors =$ Anzahl Sektoren.

Weiteres

- LBA: Anfangs nur 28-bit, **nun meist 48-bit.**
- Aus Gründen der Kompatibilität wird CHS noch angegeben.
- Unter Linux mit Befehl `sudo filefrag -e filename` schauen wo und wie Datei auf Festplatte abgelegt ist.

Reading/Writing a Block



Schwierig die Kosten genau zu berechnen

Kosten eines Festplattenzugriffs hängen ab von:

- der aktuellen **Position des Lesekopfs**
- der **Position (Drehung/Winkel) der Platte**

Diese Informationen sind zur Zeit der Übersetzung der Anfrage **nicht bekannt**.

Daher: Kosten über mehrere Zugriffe (Mittel), einfaches Modell.

Einfaches Kostenmodell

Parameter des Kostenmodells:

- Durchschnittliche **Latenzzeit** (average latency time):
  Durchschnittliche Zeit für Positionierung (seek+rotational delay)
  - Durchschnittliche Zugriffszeit für einen einzelnen Zugriff
- **Lese- / Schreibrate** (sustained read/write rate):
  - Nach Positionierung: Datentransferrate bei sequentiellem Zugriff

Beispiel: Performance Parameter für Beispiel-Festplatte

Modell aus 2004		
Parameter	**Wert**	**Abkürzung**
Kapazität (capacity)	180 GB	D_{cap}
Latenz (average latency time)	5 ms	D_{lat}
Leserate (sustained read rate)	100 MB/s	D_{srr}
Schreibrate (sustained write rate)	100 MB/s	D_{swr}

Dann: **Zeit um n Bytes zu lesen** ist geschätzt als $D_{\text{lat}} + n/D_{\text{srr}}$.

Sequenzielle und Wahlfreie Zugriffe (Sequential vs. Random I/O)

Es wird unterschieden zwischen zwei verschiedenen Arten von Zugriffen (I/O):

- **Sequenzielle (sequential)** I/O und
- **Wahlfreie (random)** I/O.

In unserem einfachen Kostenmodell:

- für sequenzielle Zugriffe: es gibt eine Positionierung des Lesekopfes, danach wird mit Leserate gelesen.
- für wahlfreie Zugriffe: es gibt eine Positionierung pro gelesener Einheit – typischerweise einer Seite von z.B. 8 KB

Beispielanwendung des einfachen Kostenmodells

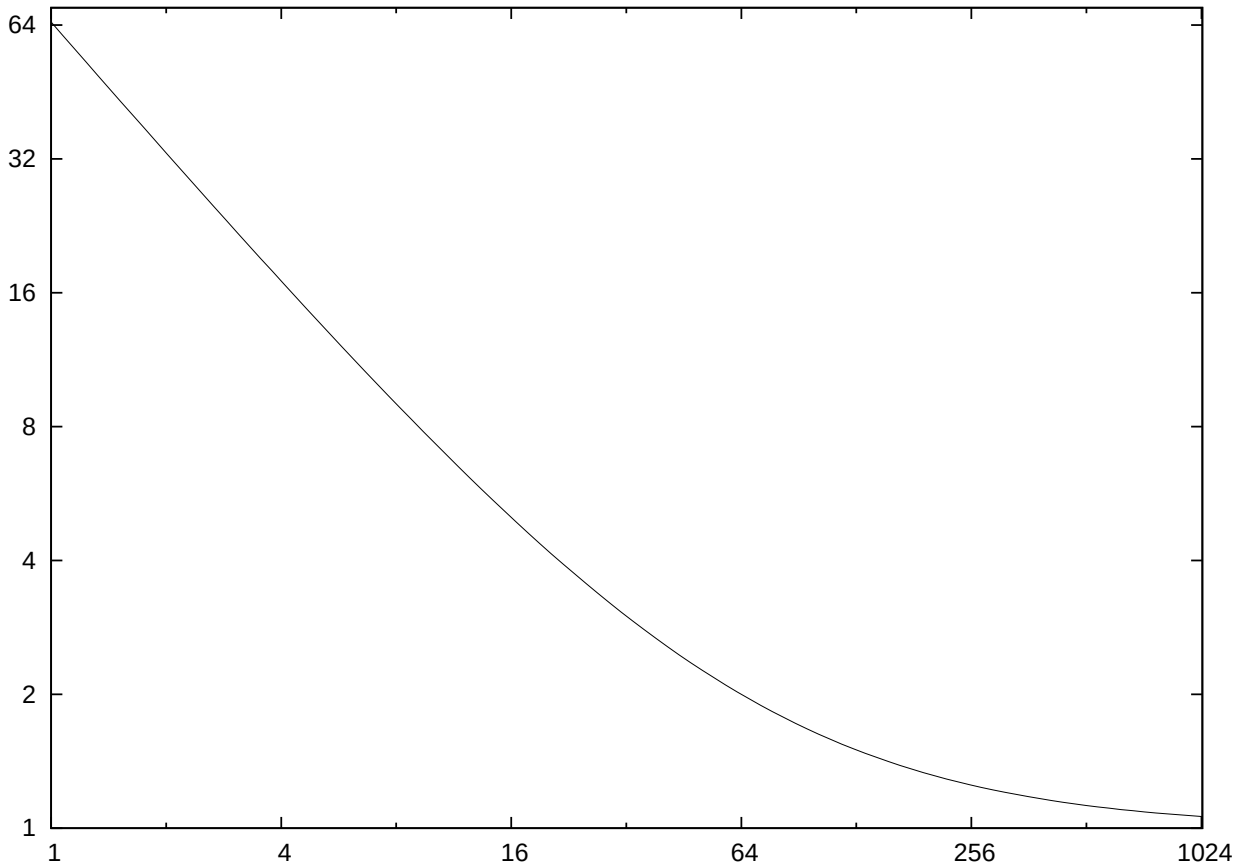
Lese 100 MB

- Sequenzielles Lesen: 5 ms + 1 s
- Lesen durch wahlfreie Zugriffe (Seitengröße 8KB): 65 s

Time to Read 100 MB

x-Achse: Größe der zu lesenden Chunks in 8 KB

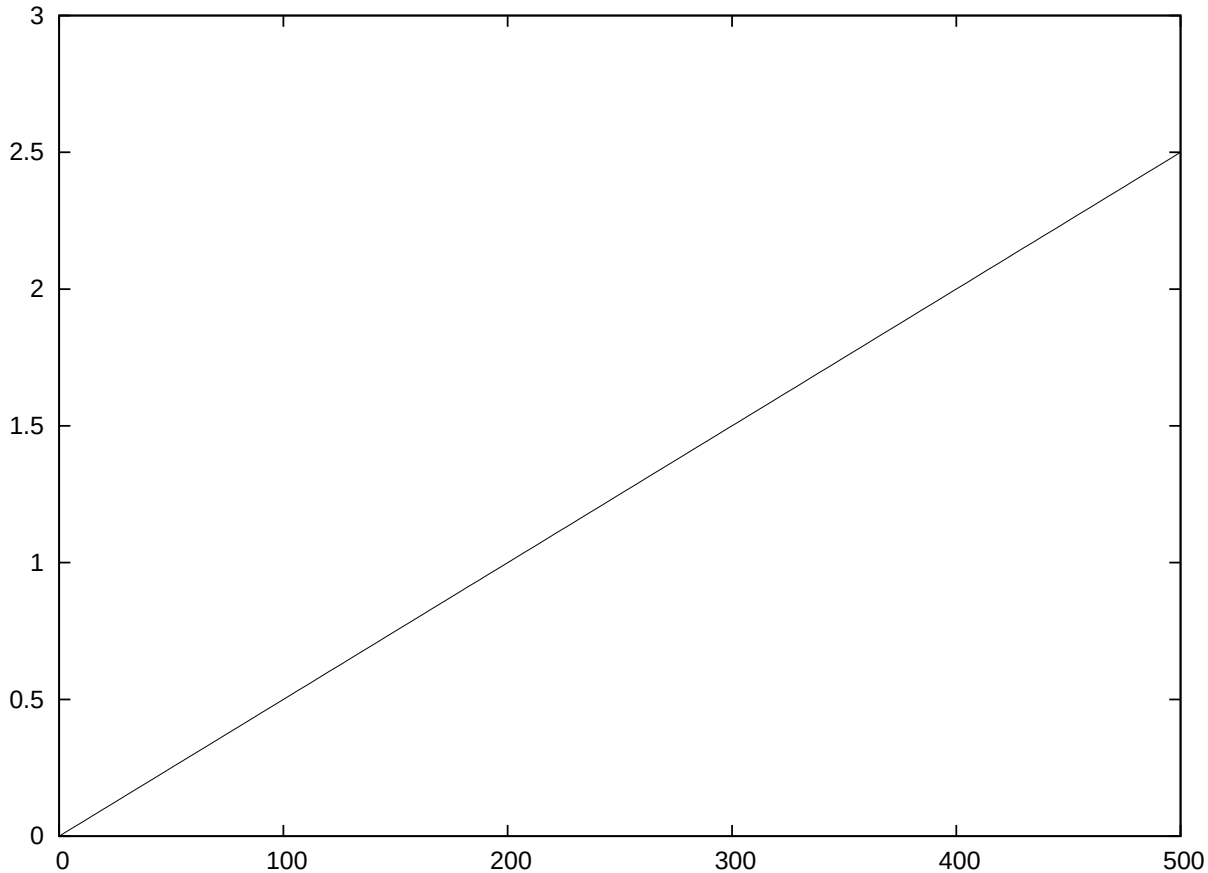
y-Achse: Sekunden



Time to Read n Random Pages

x-Achse: n

y-Achse: Sekunden



Beispiel: Berechnung Zugriffskosten

- Lesen einer Datei von 100 MB Größe, gespeichert in 12800 8 KB Seiten.
- In unserem einfachen Modell kostet das wahlfreie (random access) Lesen von 200 Seiten ungefähr genauso lange wie das Lesen der gesamten 100 MB im sequenziellen Modus.

Das heißt, das Lesen von $1/64$ einer 100 MB Datei im wahlfreien Zugriff dauert genauso lang wie das Lesen der gesamten Datei im sequenziellen Modus.

Break-Even-Point im einfachen Kostenmodell

Sei a die Positionierungs-Zeit, s die Leserate, p die Seitengröße und d die Anzahl an fortlaufend abgelegten Bytes. Dann ist der **Break-Even-Point** gegeben durch

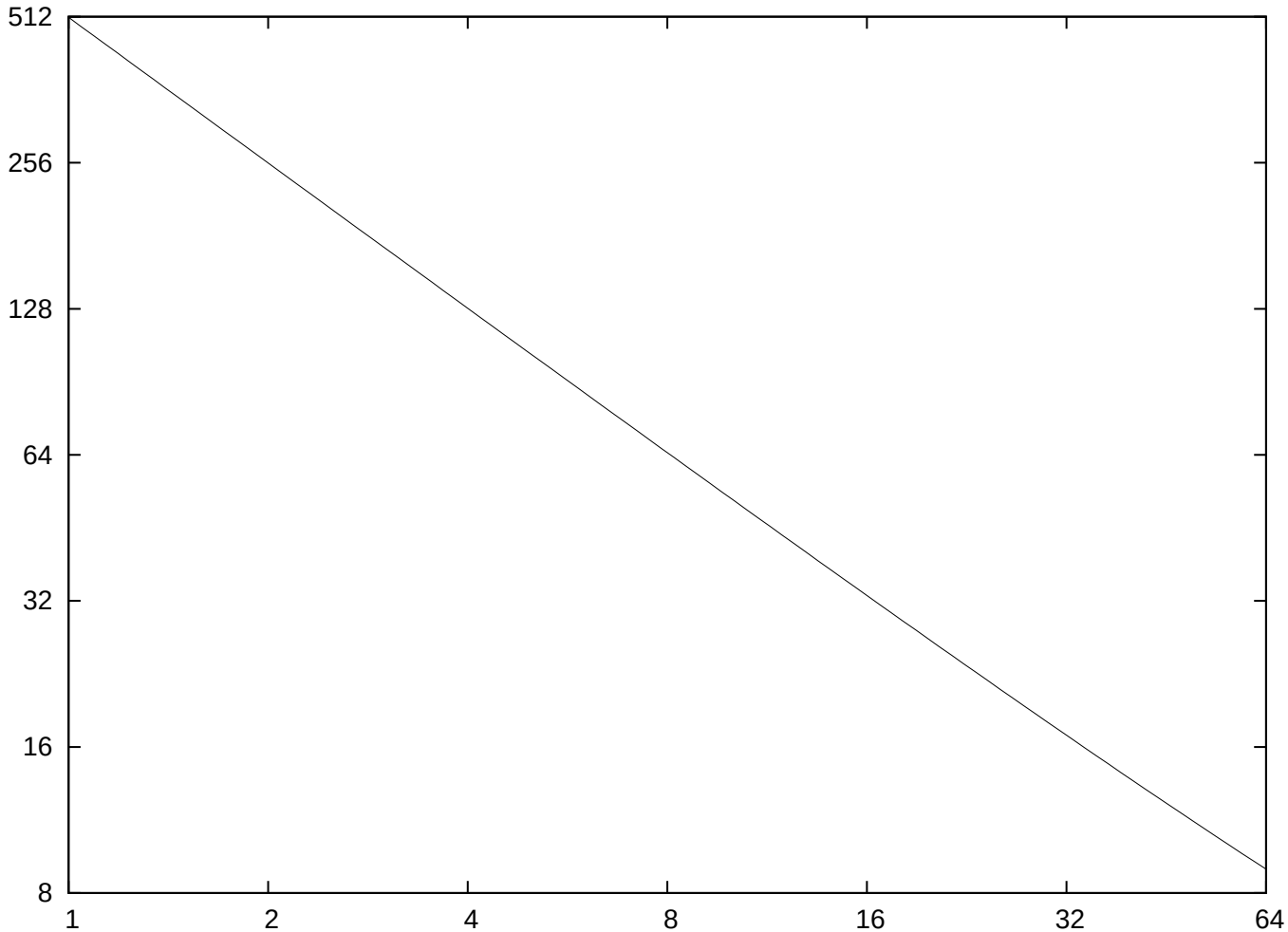
$$\begin{aligned} n * (a + p/s) &= a + d/s \\ n &= (a + d/s)/(a + p/s) \\ &= (as + d)/(as + p) \end{aligned}$$

a und s sind Parameter, die durch die Festplatte gegeben sind (also unveränderlich). Für gegebenes d hängt der Break-Even-Point nur von der Seitengröße ab.

Break-Even-Point (Abhängig von Seitengröße)

x-Achse: Die Seitengröße p in Vielfachen von 1 K

y-Achse: $(d/p)/n$ für $d = 100$ MB.



Lessons Learned

- **Sequenzielles Lesen ist sehr viel schneller als wahlfreies Lesen.**
- **Das Datenbanksystem sollte dies idealerweise ausnutzen.**

Möglichkeiten

- Sorgfältig ausgewähltes physisches Layout auf der Festplatte (z.B. Zylinder- oder Track-Aligned, Clustering)
- Festplatten Scheduling, multi-page Requests
- Prefetching
- Puffer

  und nicht zu vergessen:
- Effiziente und robuste Algorithmen (Implementierungen) der algebraischen Operatoren

Neuere Entwicklungen: SSDs

- Was ändert sich bei SSDs (Solid State Disks) gegenüber traditionellen mechanischen Festplatten?
- **Sehr viel mehr IOPs** (Input/Output Operations Per Second) möglich, Unterschied in Größenordnungen: Z.B. 1000 Leseoperationen a 4KB Block pro Sekunde einer SSD gegenüber 100 pro Sekunde einer normalen Festplatte.
- Dennoch sequenzielles Lesen günstiger als wahlfreies Lesen.
- **Was ändert dies für die Anfrageoptimierung?** Siehe Papier unten.

	Disk	Flash
Model	WD VelociRaptor 10Krpm	OCZ RevoDrive
Capacity	300gb	120gb
Price	$164	$300
Random Read	10ms	90µs
Seq. Read	120mb/s	190mb/s

Pelley et al. Do Query Optimizers Need to be SSD-aware?, ADMS Workshop, 2011.

Neuere Entwicklungen: In-Memory Datenbanken

In-Memory Datenbanken

- Verfügbarer RAM oft ausreichend groß, um gesamte Datenbank zu halten.
- Oft betrachtet in Zusammenhang mit Mehrkernsystemen und NUMA (**Non Uniform Memory Access**) Architekturen.
- Wo ist der neue **Flaschenhals für die Performance**?

Vorlesung am Ende des Semesters über aktuelle Entwicklungen in der Datenbankwelt.

Physische Organisation einer Datenbank

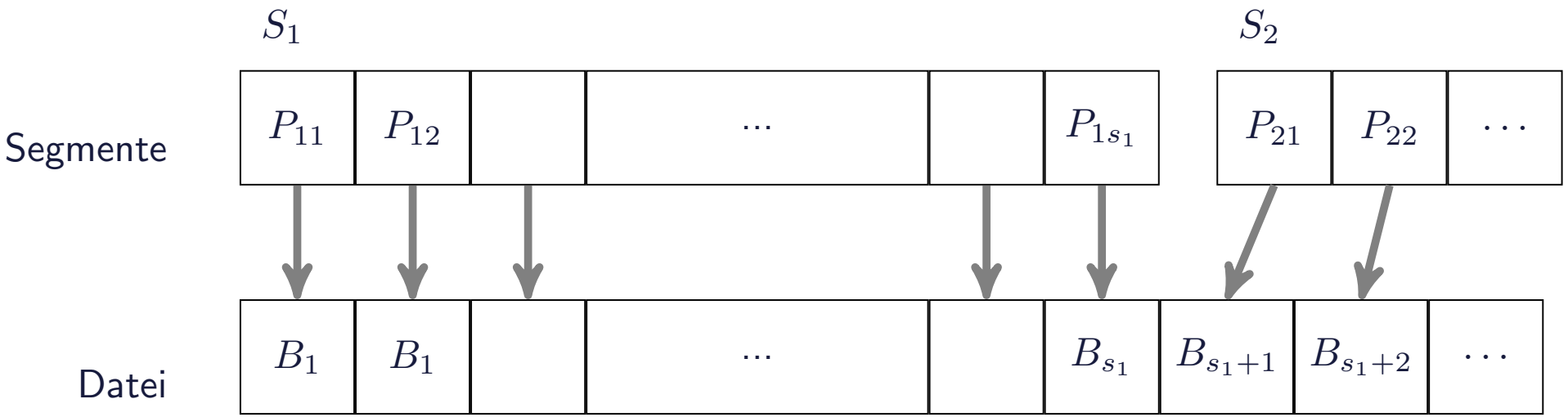
Das Datenbanksystem organisiert den physikalischen Speicher in verschiedene Schichten.

- **Datenbank**: Menge von Dateien
- **Datei**: Sequenz von Blöcken.
- **Segmente**: Organisationseinheit im DBMS (bzgl. Sperren, Rechten, etc.)

Zugriffseinheiten

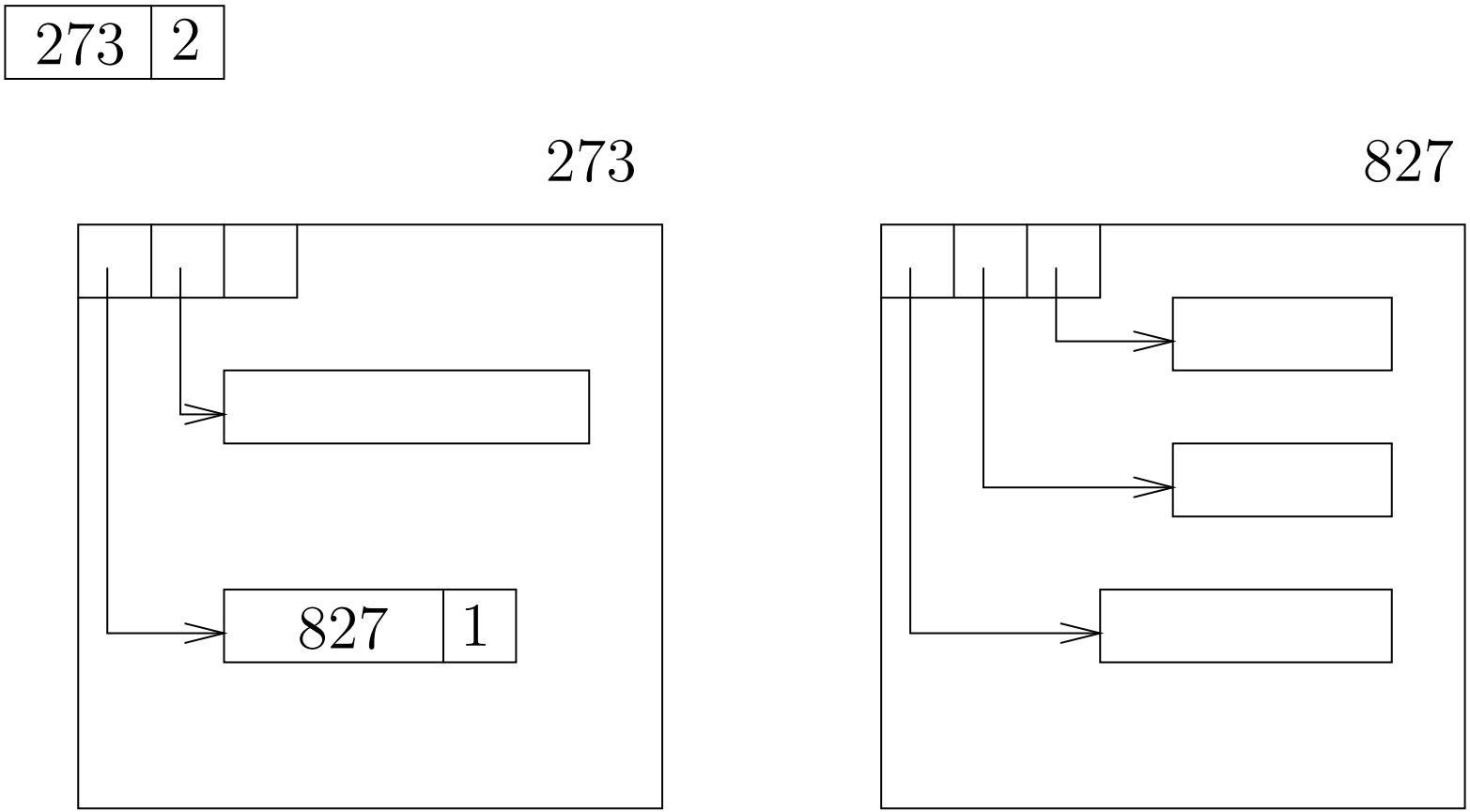
- **Segmente**
- **Seiten** werden in Segmenten gespeichert
- **Seite** enthält Sätze
- **Satz**: In einer Seite gespeicherte Sequenz von Bytes. Menge von echten Daten, verschiedene Felder.
- Bzw. man redet auch von **Tupeln** im DB (Relationen) Kontext

Seitenabbildung: Direkte Seitenadressierung



- Direkte Zuordnung zwischen Seiten eines Segments S und Blöcken einer Datei D.
- Seite P_i mit $1 \leq i \leq s$ wird in Block B_j $(1 \leq j \leq d)$ gespeichert, so dass $j = K - 1 + i$ und $d \geq K - 1 + s$.
- K bezeichnet die Nummer des ersten für S reservierten Blocks.
- **In der Regel:** 1:1 Zuordnung, d.h. $K = 1$ und $s = d$.

Seite



- Seite (Page) ist organisiert in Anzahl von Bereiche (Slots)
- Slots zeigen auf Daten
- ... oder auch auf andere Seiten

Tuple Identifier (TID)

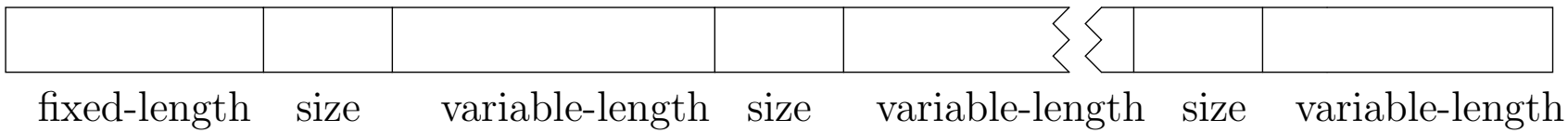
Ein TID ist eine Paar bestehend aus

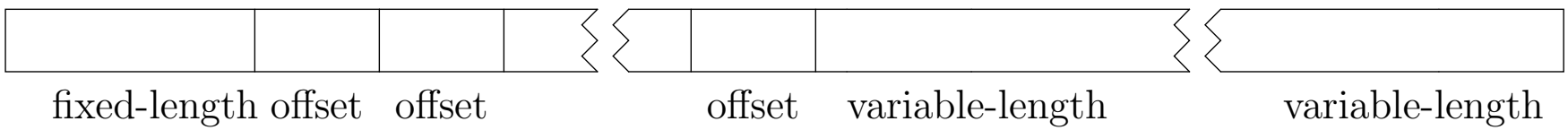
- **Seiten ID** (z.B. Datei/Segment Nummer plus Nummer der Seite)
- **Slot Nummer**

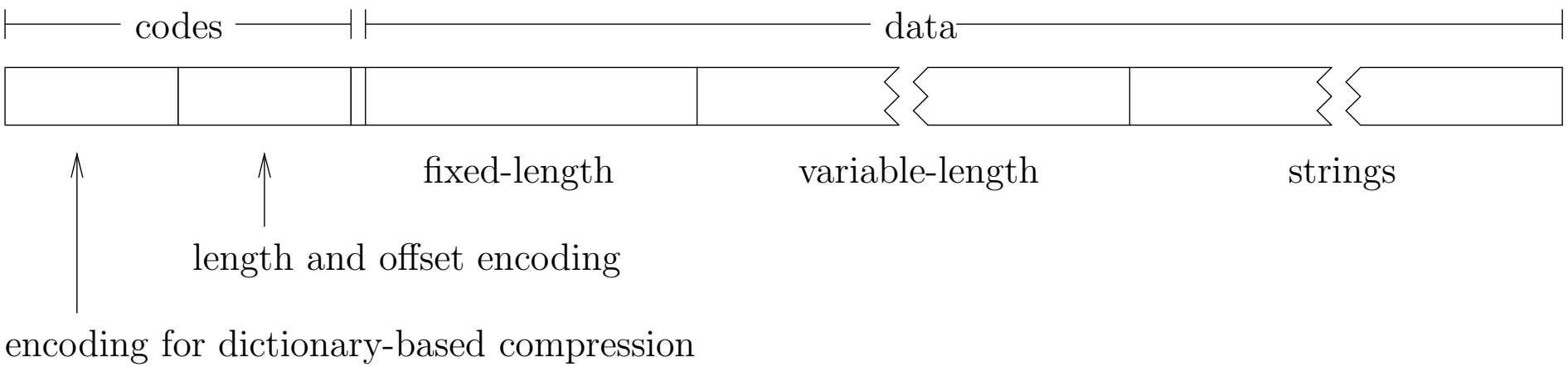
TID wird manchmal auch Row Identifier (RID) genannt

Satz Layout

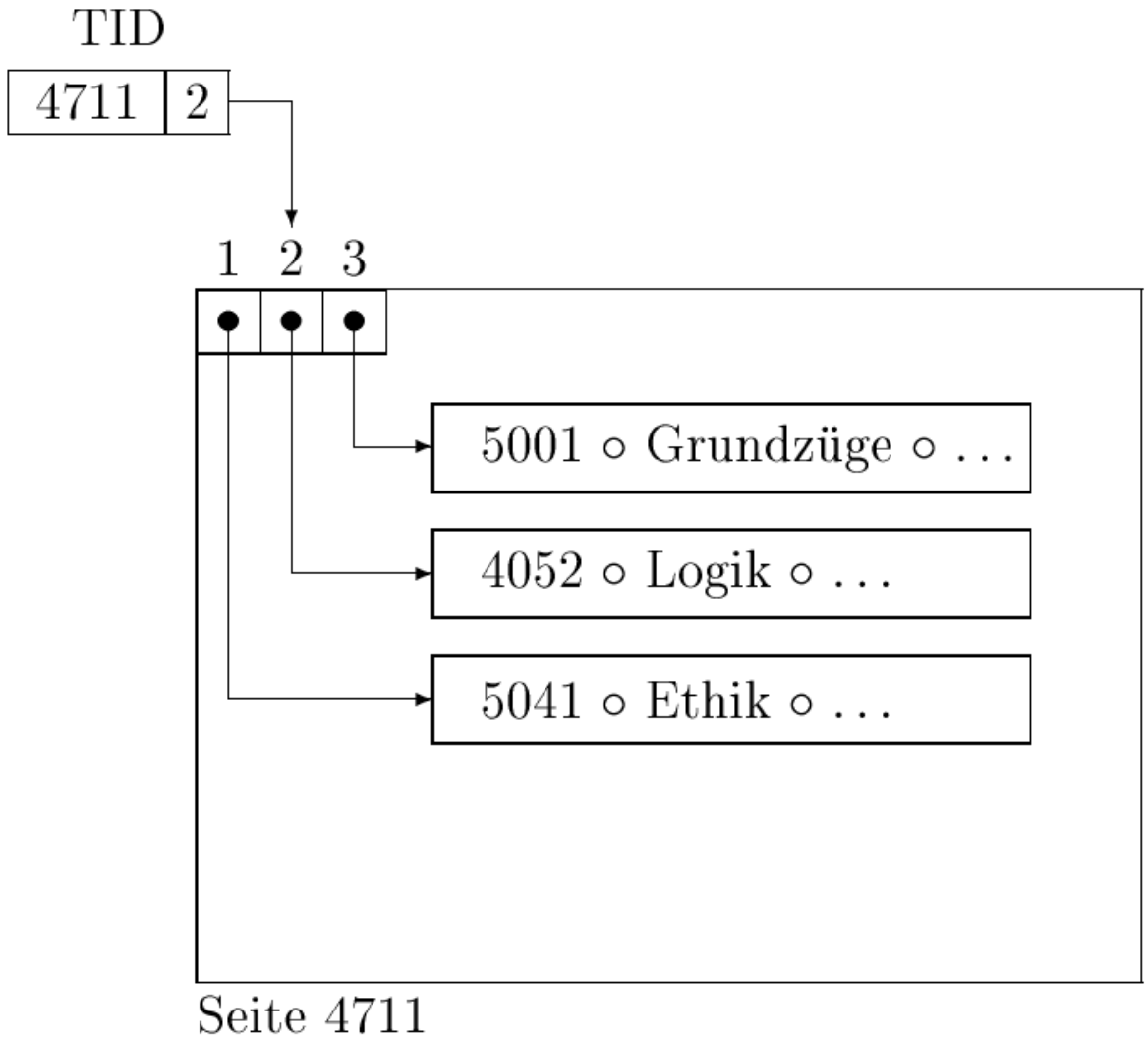
Verschiedene mögliche Layouts:



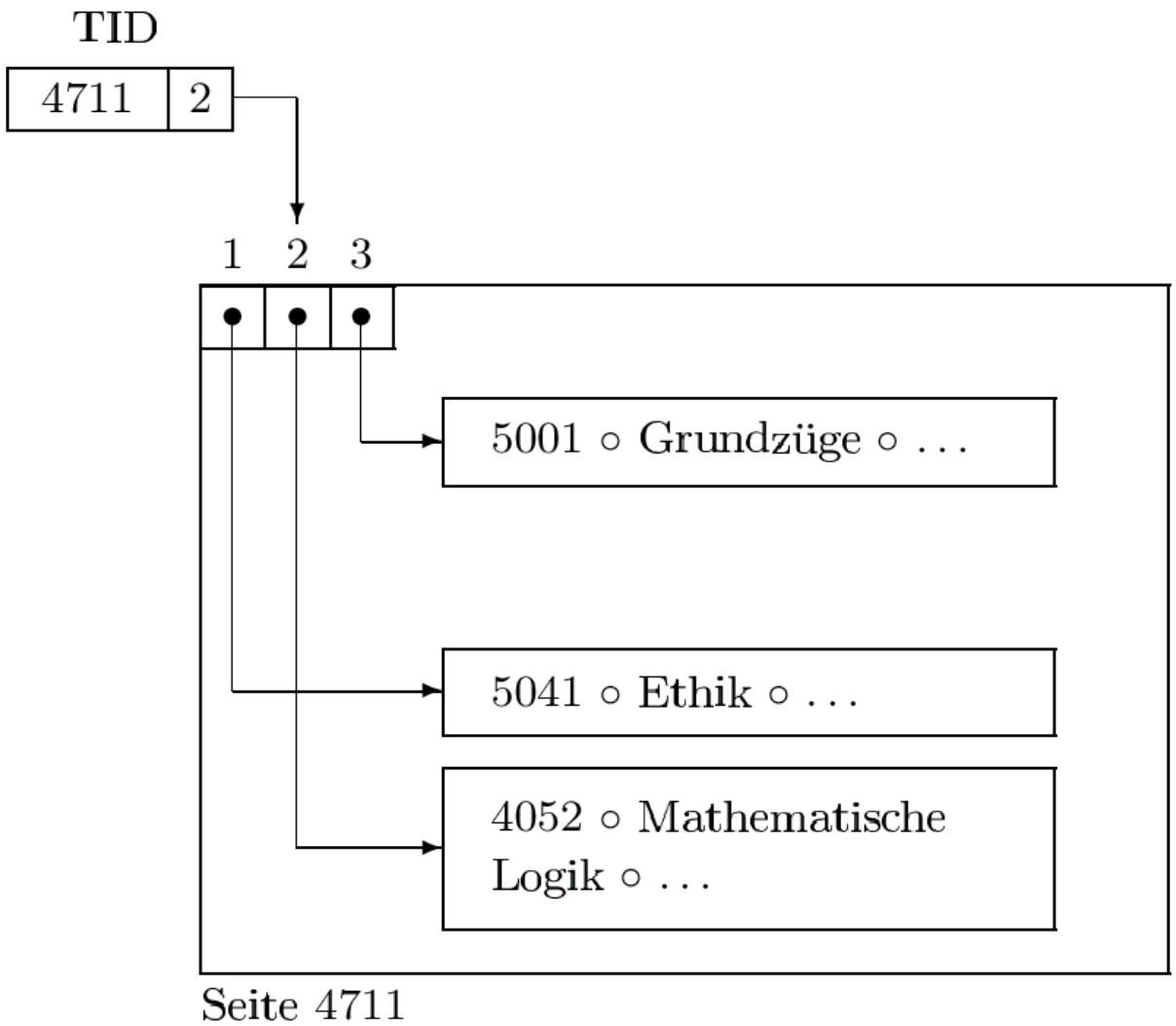




Speicherung von Tupeln auf Seiten



Verschieben von Tupeln innerhalb einer Seite



Verdrängen von Tupeln auf andere Seiten

- Falls eine Seite zu klein wird.
- Verschiebe Tupel in eine andere (z.B. neue, leere) Seite
- Füge in ursprünglicher Seite eine TID hinzu die auf den neuen Ort verweise

Was passiert bei mehrfachem Verschieben?

- Verweis in der ursprünglichen Seite wird angepasst.

$\Rightarrow$ Länge der Verweiskette auf zwei beschränkt

Katalog und Freispeicherverwaltung

Weitere Aspekte

- Abbildung von Datenbank-Tupeln auf Segmente/Seiten
- Puffer-Verwaltung: Organisation, Suche und Ersetzungsstrategien
- Verwaltung von freien Seiten (Freispeicherverwaltung)

Einbringsstrategien für Änderungen

Update-in-Place

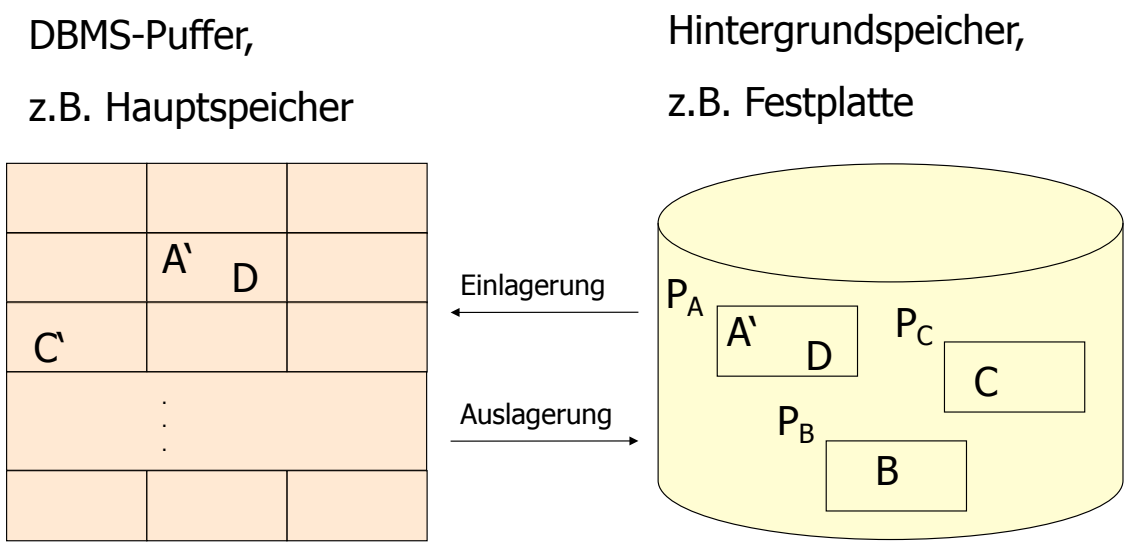
- Aka. **direkte Einbringstrategie**.
- Bei Änderung wird Seite in ihren ursprünglichen Block zurückgeschrieben.
- Was passiert wenn dies nur Teilweise geschieht, bzw. Recovery nach Wiederanlauf?
- Bzw. wenn Transaktion zurückgesetzt wird (Rollback)... $\rightarrow$ Kapitel über Recovery

Verzögerte Einbringstrategien

- Vorteile bzgl. Performance und Recovery.
- Realisierung: **Schattenspeicherkonzept.** "Switch" zwischen alter und neuer Version der Seite.

Datenbank-Pufferverwaltung

- **Motivation**: Bereits erwähnte Zugriffslücke zwischen Hauptspeicher und Festplatte (Externspeicher)
- **Idee**: Halte Seiten, auf die zugegriffen wurde, in einem Puffer im Hauptspeicher
- Dieser Puffer kann durchaus sehr groß sein (hunderte Megabyte oder viele Gigabytes). Trotzdem viel kleiner als DB selbst.



Datenbank-Pufferverwaltung (2)

5 Minuten Regel

“Pages referenced every five minutes should be memory resident.”

Siehe Papier von Jim Gray & Franco Putzolu:
`http://www.hpl.hp.com/techreports/tandem/TR-86.1.pdf`

Besonderheiten DB Puffer (vs. OS Puffer)

- **DB-spezifische Referenzmuster**
- z.B. sequentielle oder baumartige Zugriffsfolgen
- Gleiches gilt für **Prefetching**. Hier kann in DB aufgrund von Seiteninhalten oft eine Voraussage gemacht werden.

Empfehlung zum Thema Datenbank-Pufferverwaltung: Das Buch von Härder und Rahm: “Datenbanksysteme - Konzept und Techniken der Implementierung” ist hier sehr ausführlich.

Ersetzungsstrategien: Konzept und Klassifizierung

Wenn eine Seite nicht im Puffer auffindbar ist wird sie in Puffer eingetragen. Falls dieser bereits voll ist, muss eine andere Seite weichen, aber welche?

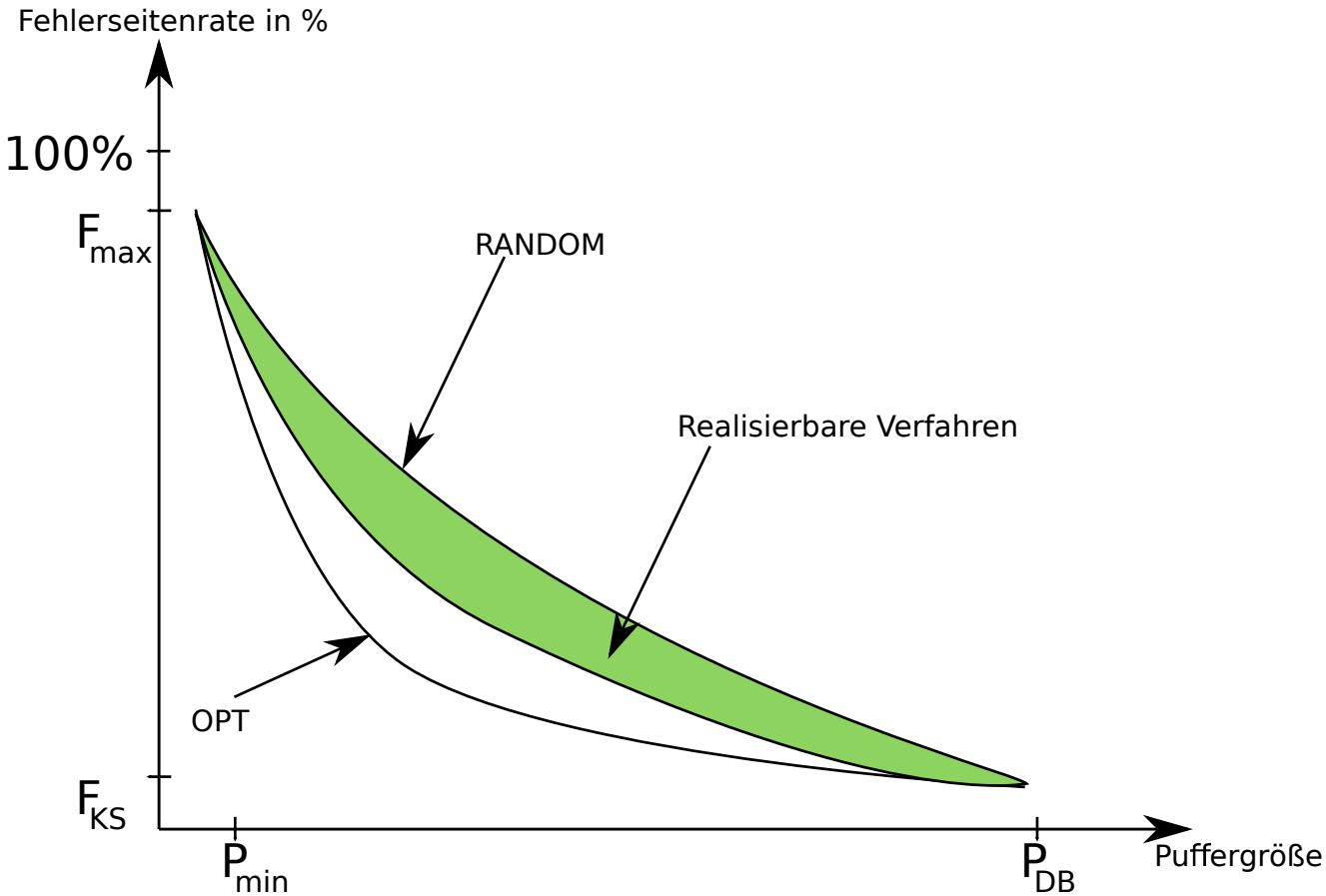
Klassifizierung von **Strategien** anhand ...

- ob das **Alter seit Einlagerung**, seit letztem Zugriff oder überhaupt nicht, und
- ob alle **Referenzen**, die letzte Referenz oder keine

bei der Auswahlentscheidung (welche Seite ersetzt werden soll) zum Tragen kommt.

Erste (triviale) Idee: Zufälliges Ersetzen von Seiten (RANDOM Strategie).

Optimales vs. Zufälliges vs. Realisierbare Verfahren



P_{min}= minimale Größe des DB-Puffers

P_{DB} = Datenbankgröße

F_{KS} = Fehlerseitenrate bei Kaltstart

FIFO

First-In, First-Out

- Ersetzt diejenige Seite, die am längsten im Puffer ist

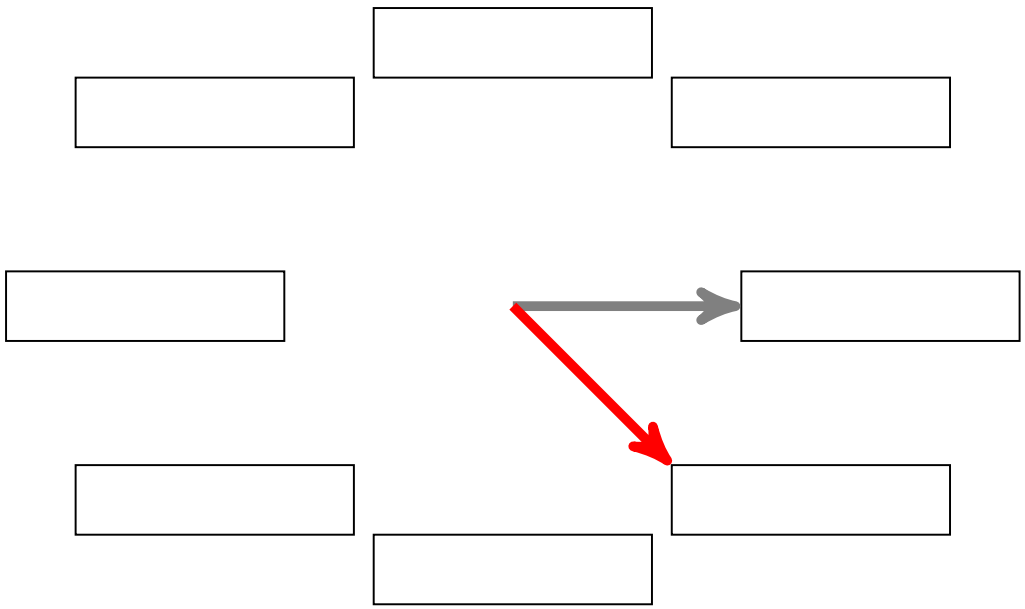


Illustration: Kreisförmige Anordnung. Zeiger verweist auf den älteste Eintrag. Grauer Zeiger= alt. Roter Zeiger=neu.

Least-Frequently-Used (LFU)

- **Ersetzt Seite mit der geringsten Referenz- (Zugriffs-) Häufigkeit**
- Für jede Seite wird ein **Zugriffszähler** (aka. Referenzzähler RZ) geführt.
- **Die Seite mit dem kleinsten Zählerstand wird ersetzt.**

Seiten, auf die kurzzeitig sehr sehr oft zugegriffen wurde bleiben sehr lange im Puffer, auch wenn nicht mehr wirklich benutzt.

Alter des Zugriffs (bzw. der letzten Zugriffe) wird nicht berücksichtigt.

- "Problem" kann durch periodisches Herabsetzen der Referenzzähler adressiert werden.

Least-Recently-Used (LRU)

- **Ersetzung basierend auf Zeit seit dem letzten Zugriff auf Seite.**
- Halte Seiten in Form eines Stacks:
  - Eine Seite kommt bei jeder Referenz auf oberste Position
  - Seite auf der untersten Position des Stacks wird bei Bedarf ersetzt

Beispiel:

Zugriff (in dieser Reihenfolge) auf Seiten:
A, B, C, D, A, C, A, B, B, B, C, D, A. Puffer hat Platz für 3 Seiten. Seite E soll eingelagert werden. Welche Seite muss weichen?

Beobachtung:

Was passiert, wenn in einer Leseoperation von der Festplatte viele neue Seiten gelesen werden?
Wieso ist das problematisch?

Theoretische Sicht

Wir betrachten einen **String von Referenzen auf Seiten**

$$r_1, r_2, \ldots, r_t, \ldots$$

wobei $r_t = p$ bedeutet, dass auf Seite p zugegriffen wurde.
Zu einem Zeitpunkt t nehmen wir an, dass jede **Seite eine gewisse Wahrscheinlichkeit b_p besitzt als nächstes Aufgerufen zu werden**, d.h. $Pr(r_{t+1} = p) = b_p$.

Interarrival Time

- **Wie viele Zugriffe liegen zwischen den Zugriffen auf eine Seite p?** Genau b_p^{-1}.

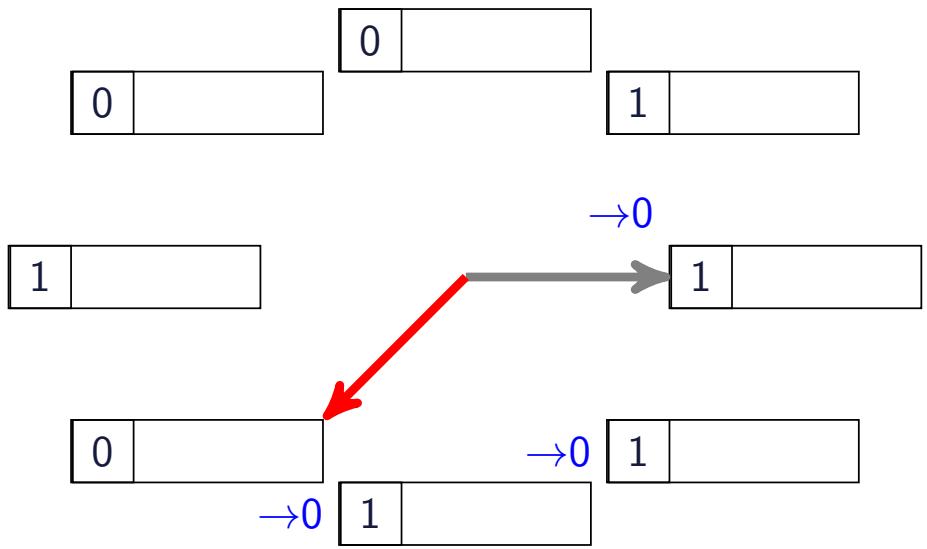
Interarrival Time Annäherung durch LRU

- Interarrival Time: Zwischen zwei Zugriffen auf Seite p liegen b_p^{-1} Zugriffe.

- **Diejenigen Seiten mit kleinster Interarrival Time bzw. größter Wahrscheinlichkeit b_p sollten im Puffer gehalten werden.**
- Dies wird **bei LRU approximiert** durch die Seiten mit dem Zeitpunkt des letzten Zugriffs auf Seite.

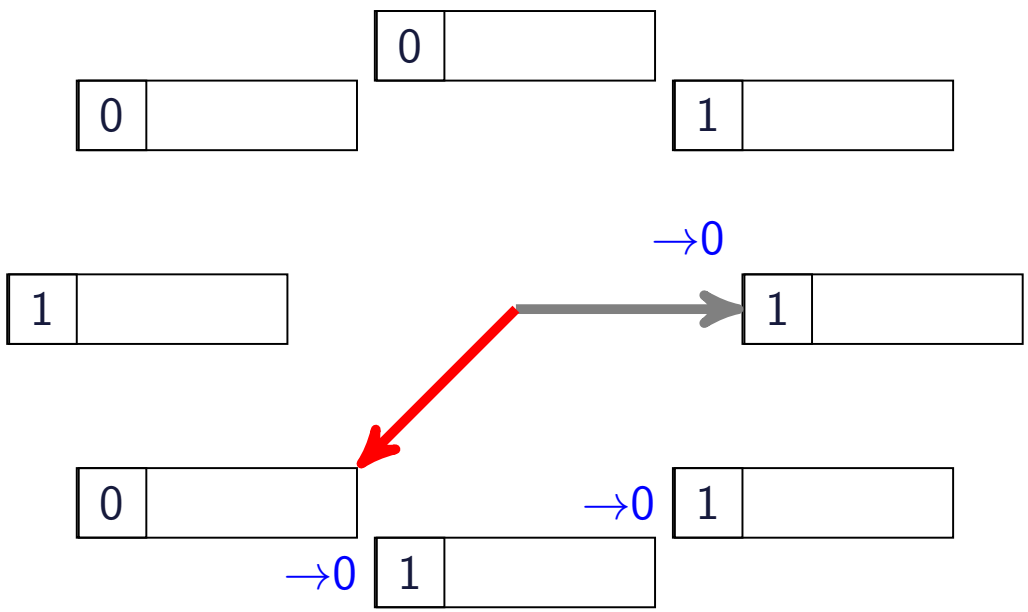
CLOCK

- Modifikation von FIFO: Jede Seite enthält ein **Benutzt-Bit, das bei Seitenreferenz auf 1 gesetzt wird.**
- Zyklische Suche: **Falls Bit auf 1, so wird es auf 0 gesetzt, nichts passiert. Zeiger wandert weiter. Falls Bit bereits auf 0, dann wird Seite entfernt.**
- Daher wird dieser Algorithmus **auch "Zweite Chance" (Second Chance) genannt.**
- **Befindet sich eine Seite bereits im Puffer, wird Bit auf 1 gesetzt, falls es 0 war, Zeiger bewegt sich nicht!**



CLOCK (2)

- In Abbildung unten: Grauer Zeiger beschreibt Stellung des Zeigers bevor die Suche nach einer zu ersetzenden Seite los geht
- Roter Zeiger beschreibt Seite die zum Ersetzen ausgewählt wurde.
- Wie wir sehen wurden die Zählerstände der vom Zeiger besuchten Seiten von 1 auf 0 gesetzt ($\rightarrow$0).



CLOCK: Beispiel

Betrachten wir die folgende Sequenz von Seiten Zugriffen: C, B, D, D, C, E für den Fall eines Puffers der Größe 3. Der initiale Puffer ist leer. Mit $\rightarrow$ ist der Zeiger auf die Stelle (Seite) im Puffer dargestellt, die bei der nächsten Ersetzung zu erst angeschaut wird.

init		
$\rightarrow$	—	0
	—	0
	—	0

1: C		
	C	1
$\rightarrow$	—	0
	—	0

2: B		
	C	1
	B	1
$\rightarrow$	—	0

3: D		
$\rightarrow$	C	1
	B	1
	D	1

4: D		
$\rightarrow$	C	1
	B	1
	D	1

5: C		
$\rightarrow$	C	1
	B	1
	D	1

6: E		
	E	1
$\rightarrow$	B	0
	D	0

Wir sehen hier auch einen **Unterschied zu LRU**: C wird durch E ersetzt, obwohl der letzte Zugriff auf C weniger lange her ist als der Zugriff auf D oder B. Bei LRU wäre B verdrängt worden.

CLOCK: Weitere Erläuterung

Schauen wir uns den Übergang von Schritt 5 nach 6 genauer an:

5: C		
→	C	1
	B	1
	D	1

6: E		
	E	1
→	B	0
	D	0

Was ist hier passiert? Da E nicht im Puffer gespeichert war wird nun laut Zeigerstand zuerst geschaut ob C ersetzt werden kann. Da C einen Zählerstand von 1 hat wird C nicht ersetzt, bekommt aber den neuen Zählerstand von 0. Der Zeiger geht weiter auf B. Auch B kann nicht ersetzt werden: also bekommt auch B einen neuen Zählerstand von 0 und der Zeiger geht weiter auf D. Auch hier kann nur der Zählerstand auf 0 gesetzt werden. Und der Zeiger geht auf C, das nun einen Zähler von 0 hat und durch E ersetzt wird. Im folgenden sind diese Schritt angegeben:

5: C		
→	C	1
	B	1
	D	1

6a: E		
	C	0
→	B	1
	D	1

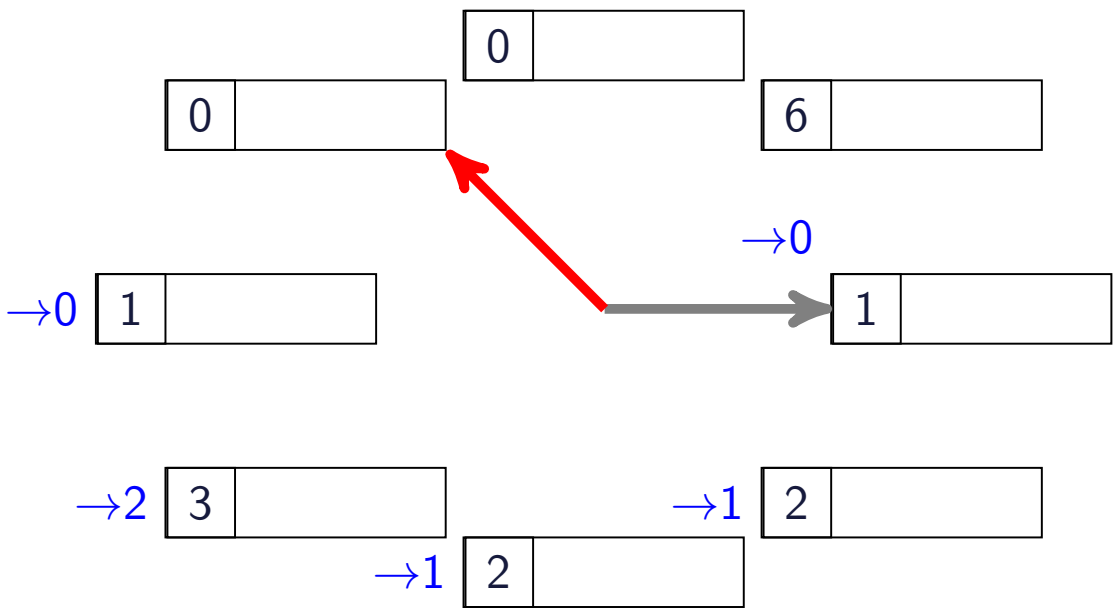
6b: E		
	C	0
	B	0
→	D	1

6c: E		
→	C	0
	B	0
	D	0

6d: E		
	E	1
→	B	0
	D	0

GCLOCK: Generalized CLOCK

- **Jede Seite bekommt Referenzzähler (RZ)**
- **Bei Referenz wird dieser erhöht**
- Wie bei CLOCK: **zyklische Suche**
  - Zähler wird herabgesetzt.
  - Null? Dann wird Seite ersetzt. Nicht Null? Dann weiter suchen.



GCLOCK: Varianten zur Berücksichtigung verschiedener Typen

Für verschiedene Typen T_i von Seite (z.B. Index-Seiten, Datenseiten) können verschiedene Werte zur Initialisierung und Inkrement von RZ angegeben werden.

Für Seite j vom Typ i kann Referenzzähler wie folgt verändert werden:

V1: -bei Erstreferenz: $RZ(j) = E_i$
-bei jeder weiteren Referenz: $RZ(j) = RZ(j) + W_i$

V2: -bei Erstreferenz: $RZ(j) = E_i$
-bei jeder weiteren Referenz: $RZ(j) = W_i$

Beobachtung: Wenn $E_i = 1$ und $W_i = 1$ (für alle i), dann geht V2 über in CLOCK, während V1 die Grundversion von GCLOCK darstellt.

LRU-k

- **Die letzten k Referenzzeitpunkte einer Seite werden berücksichtigt.**
- **Kann zwischen häufig und weniger häufig referenzierten Seiten unterscheiden**

Paper von J. O'Neill, P. O'Neill und G. Weikum: "The LRU-K Page Replacement Algorithm For Database Disk Buffering", aus dem Jahr 1993.

LRU-k: Definition

Backward k-Distance

Wir betrachten wieder einen String von Referenzen auf Seiten

$$r_1, r_2, \ldots, r_t, \ldots$$

Die **backward k-distance** $b_t(p,k)$ ist die **Distanz zur k-jüngsten Referenz** auf Seite p.

$$b_t(p,k) = \begin{cases} x & \text{falls } r_{t-x} \text{ die Seite } p \text{ referenziert und es gab genau } k-1 \text{ Auftreten einer Referenz auf } p \text{ in der Zeit zwischen } t-x \text{ und } t. \\ \infty & \text{falls } p \text{ weniger als } k \text{ mal referenziert wurde in } r_1, r_2, \ldots r_t \end{cases}$$

- **Ersetze die Seite mit der maximalen** $b_t(p,k)$.
- Es kann vorkommen, dass mehrere Seiten $b_t(p,k) = \infty$ haben, dann Ersetzung durch alternative Strategie (z.B. LRU)

Übersicht

Welche Ersetzungsstrategie passt in welche Zelle der Tabelle? (Mehrfachnennung möglich)

Berücksichtigung bei Auswahlentscheidung		Alter		
		nicht	seit letzter Ref.	seit Einlagerung
Referenzen	keine			
	letzte Referenz			
	alle Referenzen			

Puffer: Ausblick

Ausblick

- Was passiert mit Seiten im Puffer, die geändert werden? Gleich auf Festplatte schreiben oder erstmal noch im Puffer lassen?
- Was passiert mit Seiten, die von noch laufender Transaktion benutzt werden aber ersetzt werden soll? Ersetzung zulassen oder nicht?
- Welche Auswirkungen hat dieses auf Fehlerbehandlung?

  $\rightarrow$ Kapitel über Recovery