10. E/A-Architektur und Zugriff

Vorlesung "Informationssysteme"

Sommersemester 2017

Technische Universität
Kaiserslautern

Gliederung

- Aufbau des DB-Servers
  - Vereinfachtes Schichtenmodell
  - Verarbeitung von Anfragen
- Einsatz einer Speicherhierarchie
- E/A-Architektur von Datenbanksystemen
  - Sekundärspeicher - Zugriffsarten und Kostenmodell
  - Abbildung von Relationen auf Sekundärspeicher
  - Datenbankpuffer
    - Funktionsweise
    - Ersetzungsstrategien

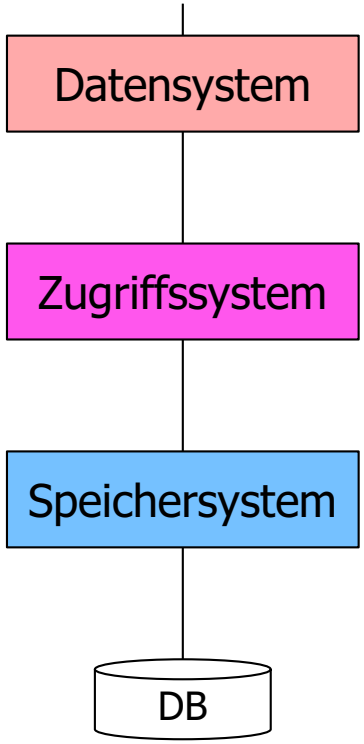
Vereinfachtes Schichtenmodell

Aufgaben der Systemschicht

Übersetzung und Optimierung von Anfragen

Verwaltung von physischen Sätzen und Zugriffspfaden

DB-Puffer- und Externspeicherverwaltung



Art der Operationen an Der Schnittstelle

Deskriptive Anfragen, Zugriff auf Satzmengen
SELECT ... FROM ... WHERE
INSERT INTO ...

Satzzugriffe
Lese Satz
Füge Satz ein, füge Eintrag in B^+-Baum ein

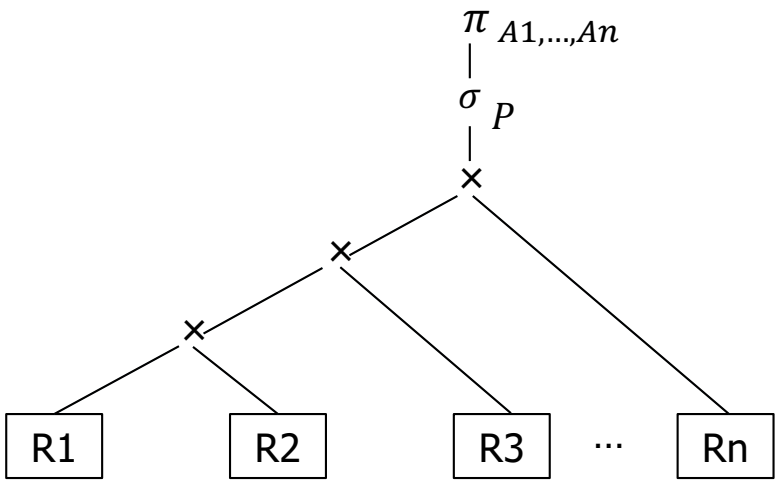
Seitenzugriffe
Stelle Seite bereit,
Gib Seite frei

Übersicht Anfrageverarbeitung

- Eingabe: Anfrage als Text (String)
- Kompilierzeitsystem (compile time system) übersetzt und optimiert die Anfrage
- Zwischenprodukt: Anfrage als Anfrageplan (query plan)
- Laufzeitsystem (runtime system) führt die Anfrage aus
- Ausgabe: Ergebnisse der Anfrage

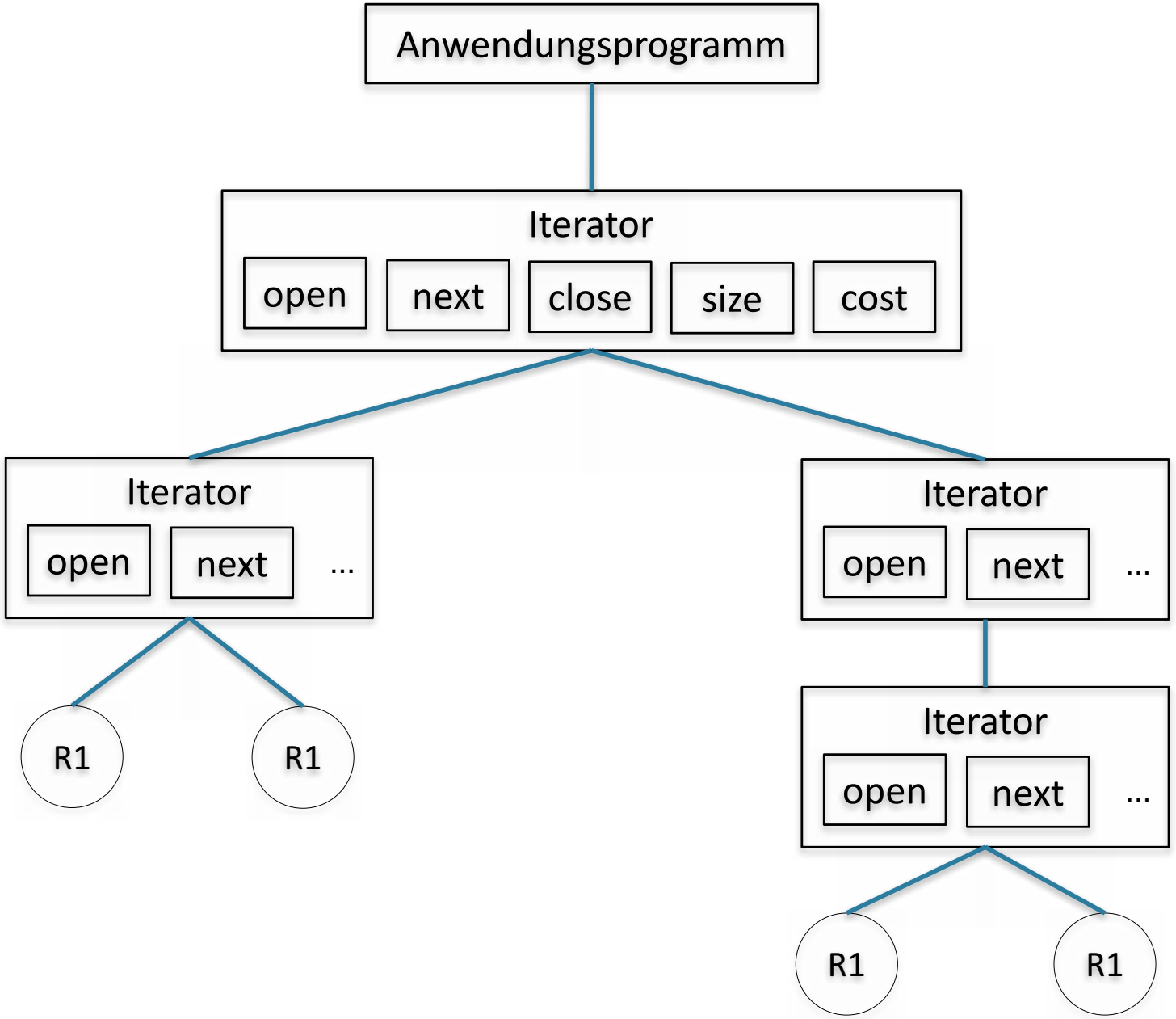
Übersetzung von SQL in Baum aus Operatoren

SELECT A1, ... An
FROM R1, ..., Rn
WHERE P



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

Iteratorkonzept



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 73-169.

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
  - 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?
- ...

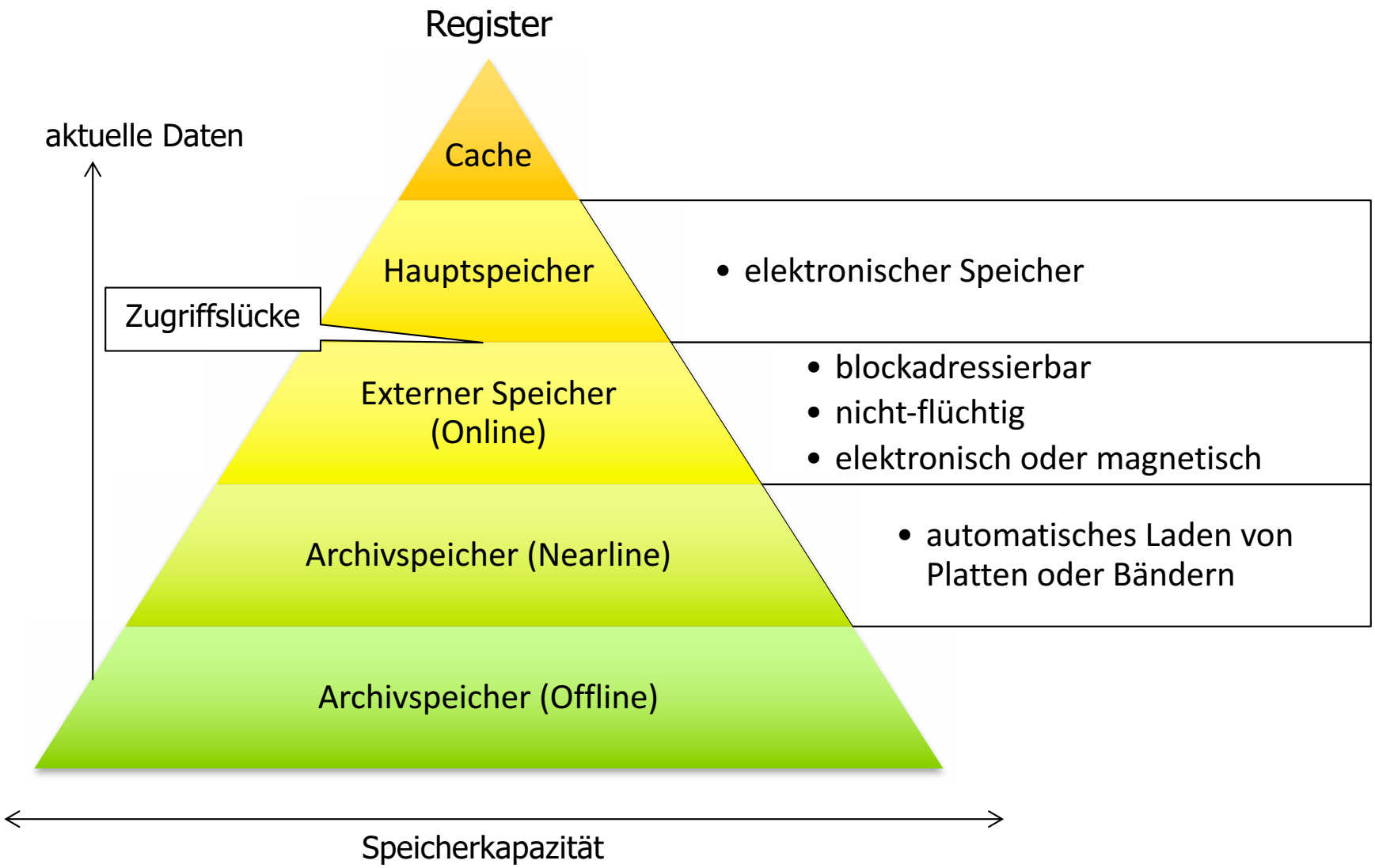
Implementierung von Operatoren wird in der VL Datenbanksysteme besprochen.

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 data center 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 → Netherlands → CA 150 000 000 ns

Numbers by Jeff Dean (Google)

Speicherhierarchie



Charakteristische Merkmale

	Ebene	Kapazität	Zugriffs-zeit	Transfer: Kontrolle	Transfer: Einheit
6	Register	Bytes	< 1 ns	Programm/ Compiler	1-8 Bytes
5	Cache	K - M Bytes	1-20 ns	Cache-Controller	8-128 Bytes
4	Hauptspeicher	M- G Bytes	50-100 ns	Betriebs-system	1-16 KBytes
3	Plattenspeicher	G – T Bytes	~ 10 ms	Benutzer/ Operator	MBytes (Dateien)
2	Bandspeicher	T Bytes	> 1 sec	Benutzer/ Operator	MBytes (Dateien)
1	Archivspeicher	T – P Bytes	sec-min		

Zugriffslücke

Relative Zugriffszeiten in einer Speicherhierarchie

- Wie weit sind die Daten entfernt?
- Welche Konsequenzen ergeben sich daraus?
  - Caching
  - Replikation
  - Prefetching

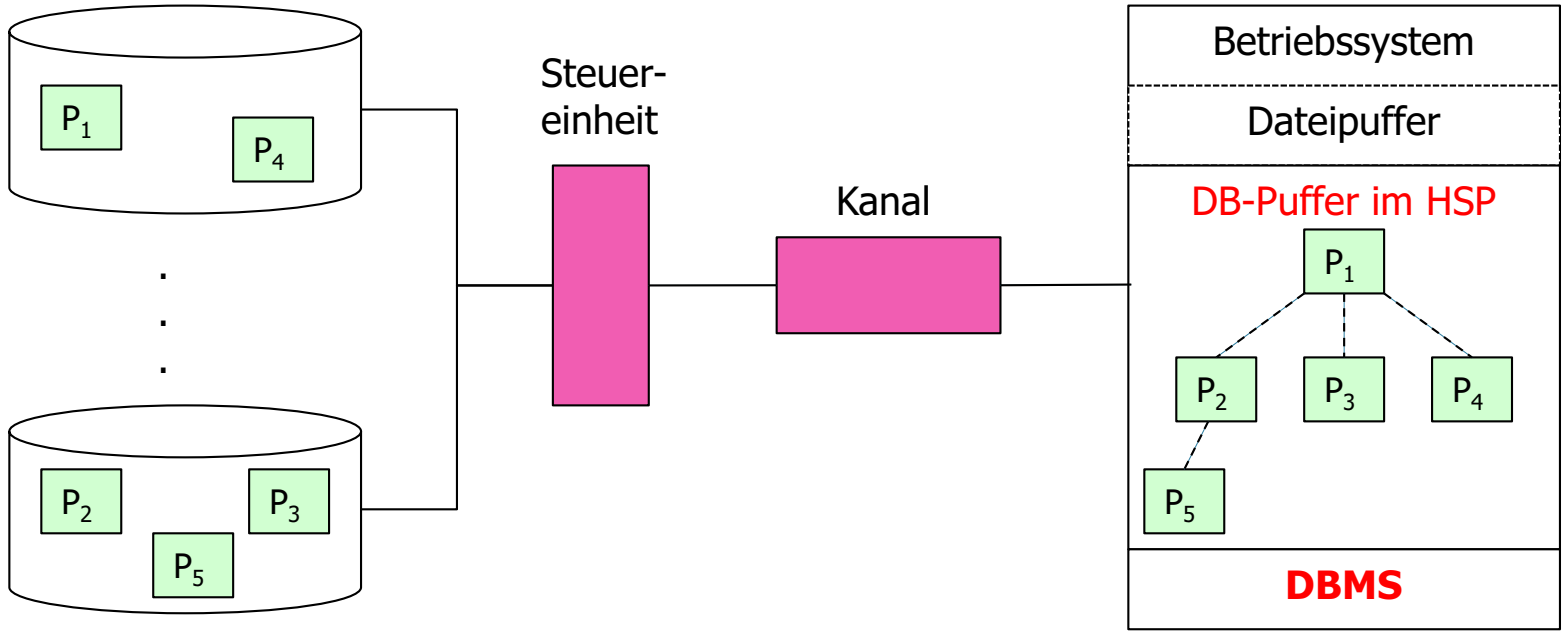
Speicherhierarchie		**Was bedeutet das in „unseren Dimensionen“?**	
Speichertyp	Rel. Zugriffszeit	Zugriffsort	Zugriffszeit
Register	1	mein Kopf	1 min
Cache (on chip)	2	dieser Raum	2 min
Cache (on board)	10	dieses Gebäude	10 min
Hauptspeicher	100	Frankfurt (mit Auto)	100 min
Magnetplatte	10^6–10^7	Pluto (5910 * 10^6 km)	> 2 Jahre
Magnetband/Opt. Speicher (automat. Laden)	10^9–10^{10}	Andromeda	> 2000 Jahre

Problemstellung

- Langfristige Speicherung und Organisation der Daten im Hauptspeicher?
  - Speicher ist (noch) flüchtig!
  - Speicher ist (noch) zu teuer und zu klein
  - Kostenverhältnis Festplatte vs. DRAM
- Mindestens zweistufige Organisation der Daten erforderlich
  - langfristige Speicherung auf großen, billigen und nicht-flüchtigen Speichern (Externspeicher)
  - Verarbeitung erfordert Transport zum/vom Hauptspeicher

Zweistufige Speicherhierarchie

- Vereinfachte E/A-Architektur:
  - Modell für Externspeicheranbindung
  - Vor Bearbeitung sind die Seiten in den DB-Puffer zu kopieren
  - Geänderte Seiten müssen zurückgeschrieben werden



Magnetplattenspeicher
ca. 10 ms

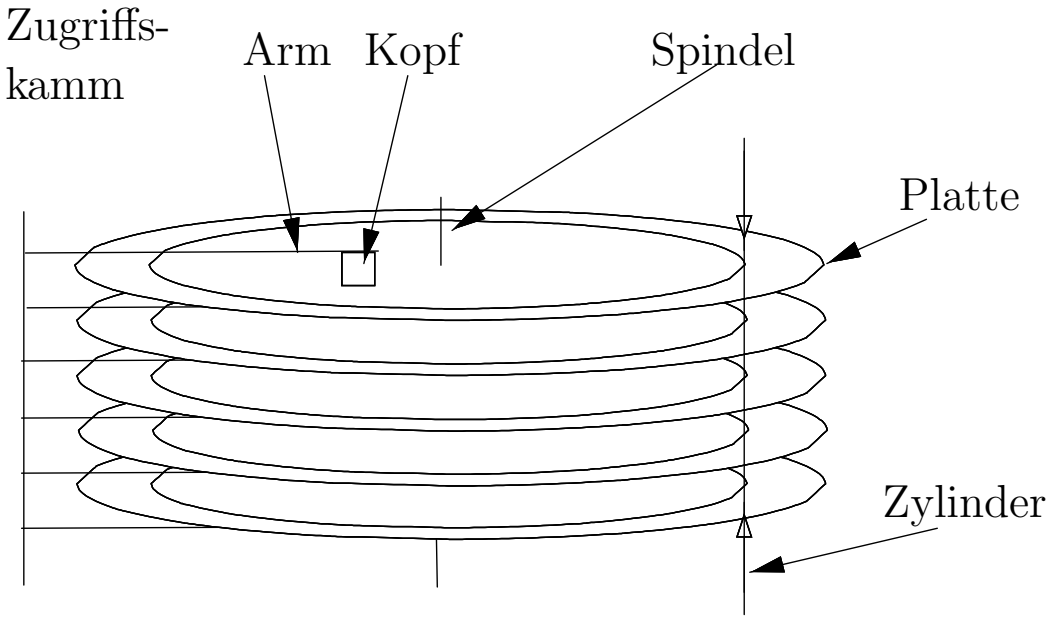
Zugriffslücke
$>2*10^5$

Hauptspeicher
< 50 ns

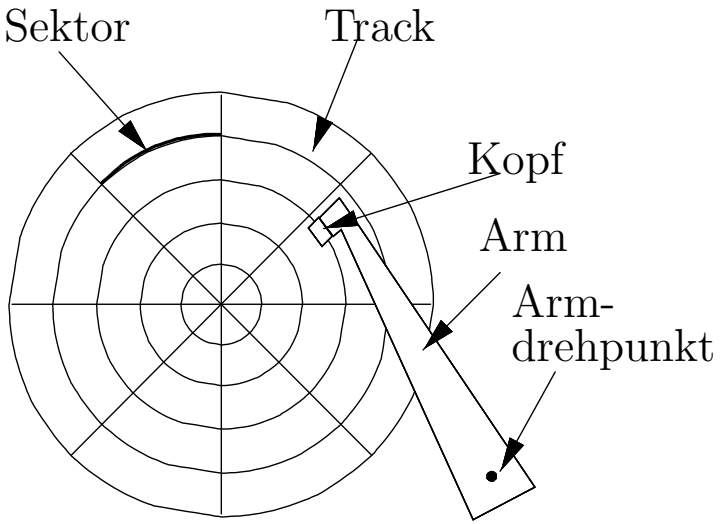
Faktor 100 - 1000 ←

→ *Faktor 100*

Aufbau einer (klassischen) Festplatte



a. Seitenansicht



b. Draufsicht

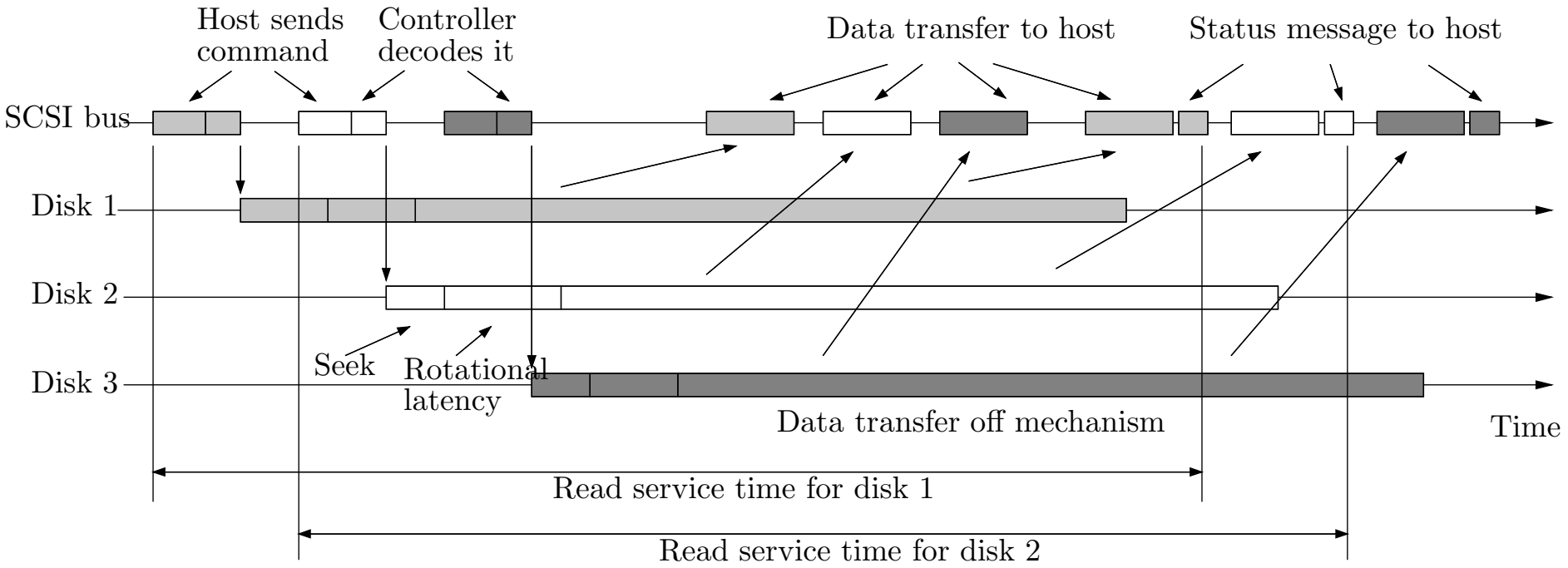
Aufbau Festplatte (Tracks, Sektoren, Zonen)

- 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.

Lesen/Schreiben eines Blocks



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
- Performance Parameter für eine 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}

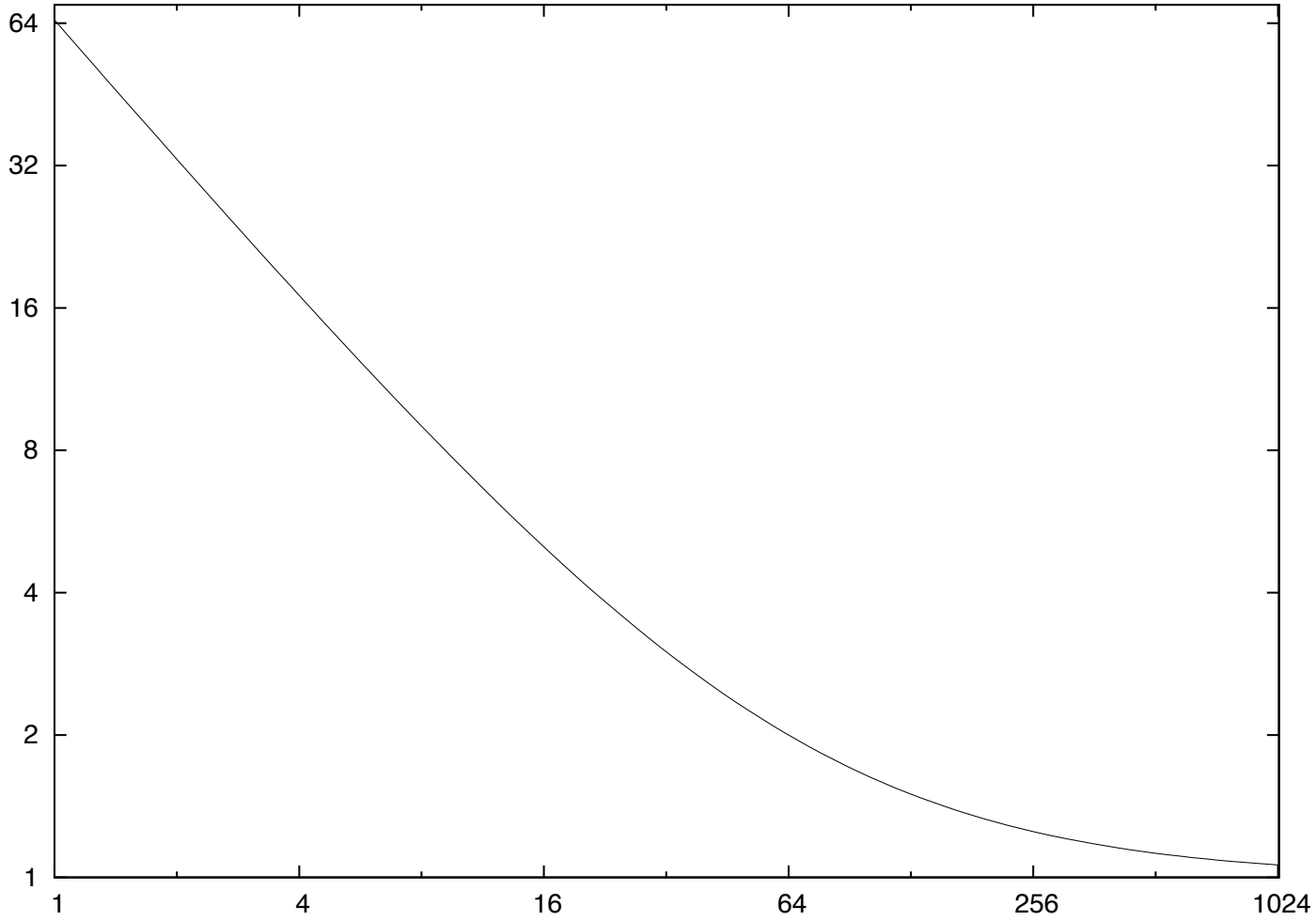
Dann: Zeit um n Bytes zu lesen ist geschätzt als $D_{lat} + n/D_{srr}$.

Sequenzielle und Wahlfreie Zugriffe

- 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 Seite von z.B. 8 KB
- Beispiel: Lese 100 MB
  - Sequenzielles Lesen: 5 ms + 1 s
  - Lesen durch wahlfreie Zugriffe (Seitengröße 8KB): 65 s

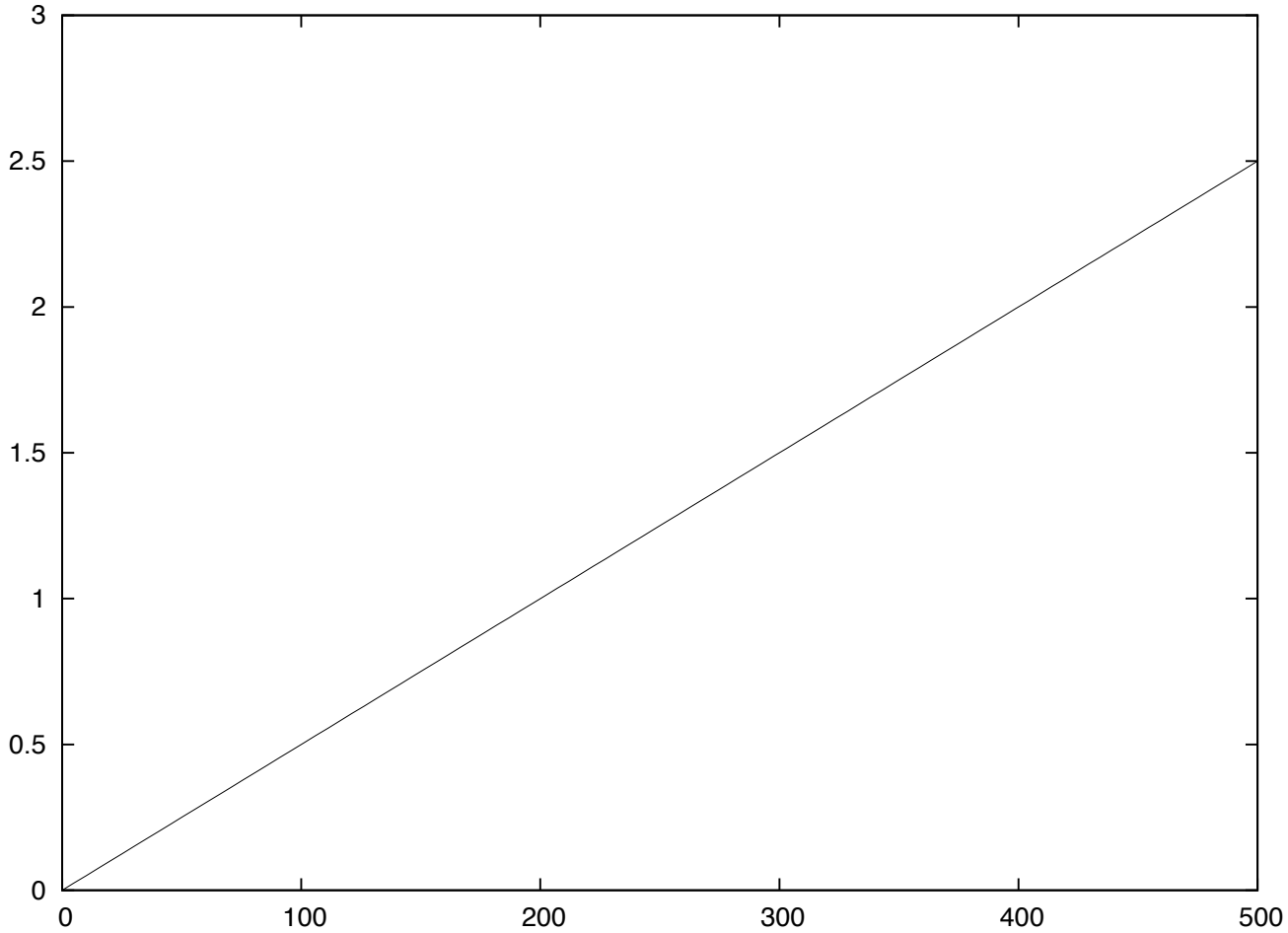
Lesen von 100MB

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



Wahlfreies Lesen von n Seiten

- 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

$$n * (a + p/s) = a + d/s$$

$$n = (a + d/s)/(a + p/s)$$

$$= (as + d)/(as + p)$$

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.

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

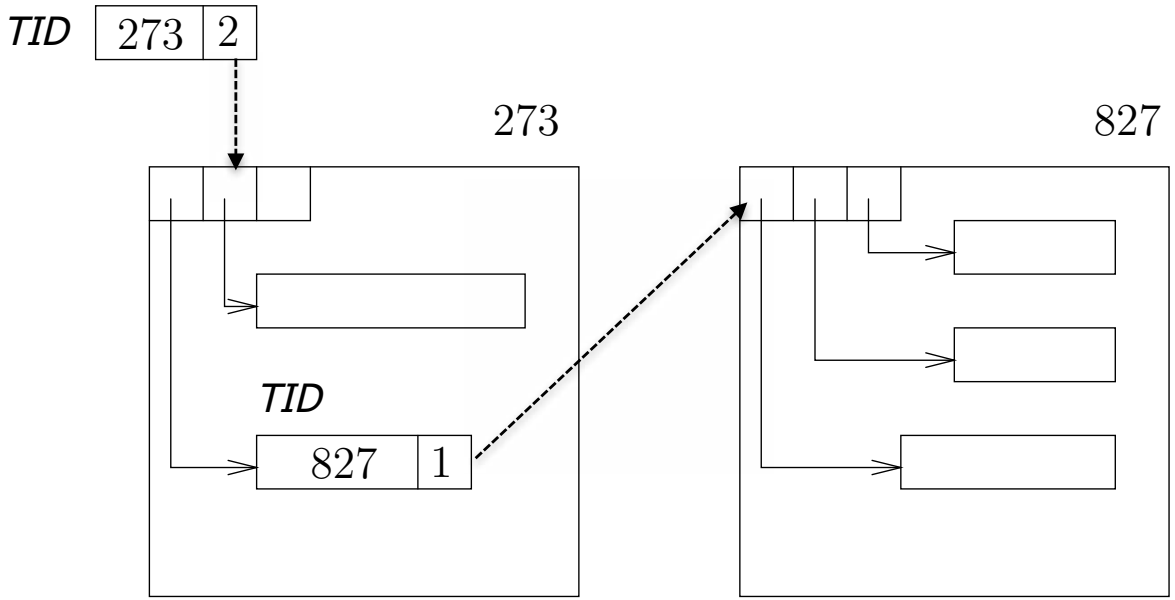
- In-Memory Datenbanken
  - Verfügbarer RAM oft ausreichend groß, um gesamte Datenbank zu halten.
  - Oft betrachtet in Zusammenhang mit Mehrkernsystemen und NUMA (Non Uniform Memory Access) Architekturen.
  - Wo ist der neue Flaschenhals für die Performance?

Physische Organisation einer Datenbank

- Das Datenbanksystem organisiert den physischen Speicher in verschiedene Schichten.
  - Datenbank: Menge von Dateien
  - Datei: Sequenz von Blöcken.
  - Segment: 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

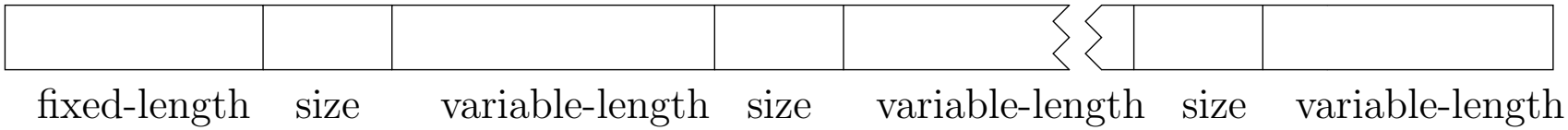
Seite und Tuple Identifier (TID)

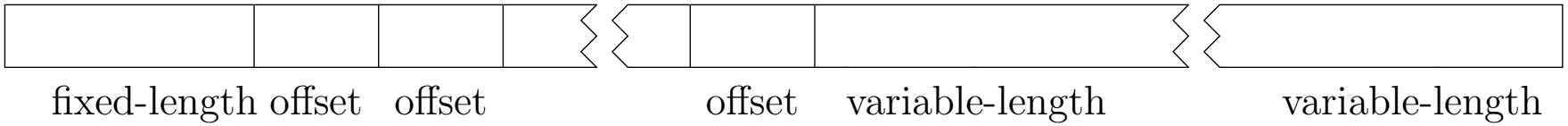
- Seite (Page) ist organisiert in Anzahl von Bereiche (Slots)
- Slots zeigen auf Daten ... oder auch auf andere Seiten
- Ein TID ist ein Paar bestehend aus
  - Seiten ID (z.B. Datei/Segment Nummer plus Nummer der Seite)
  - Slot Nummer
- TID wird manchmal auch Row Identifier (RID) genannt

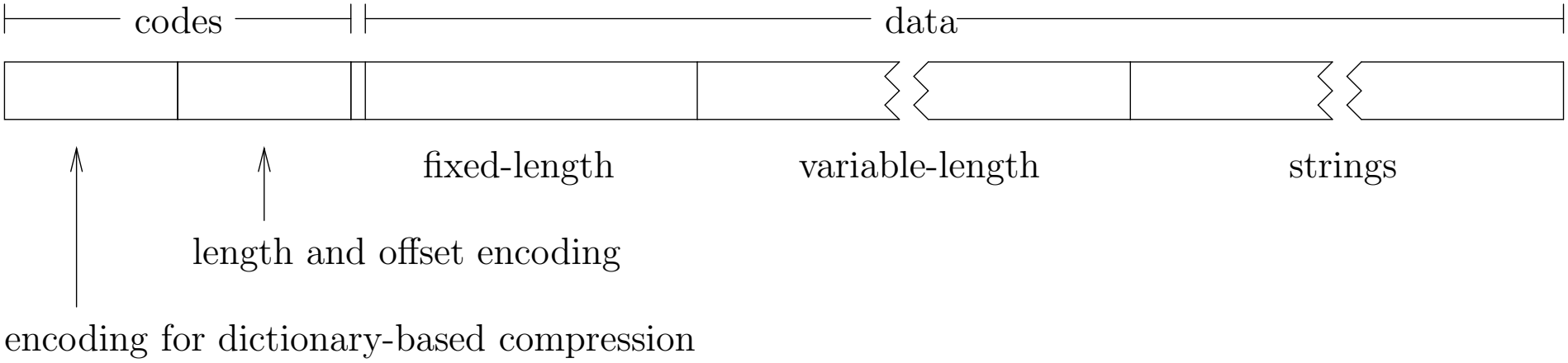


Satz-Layout

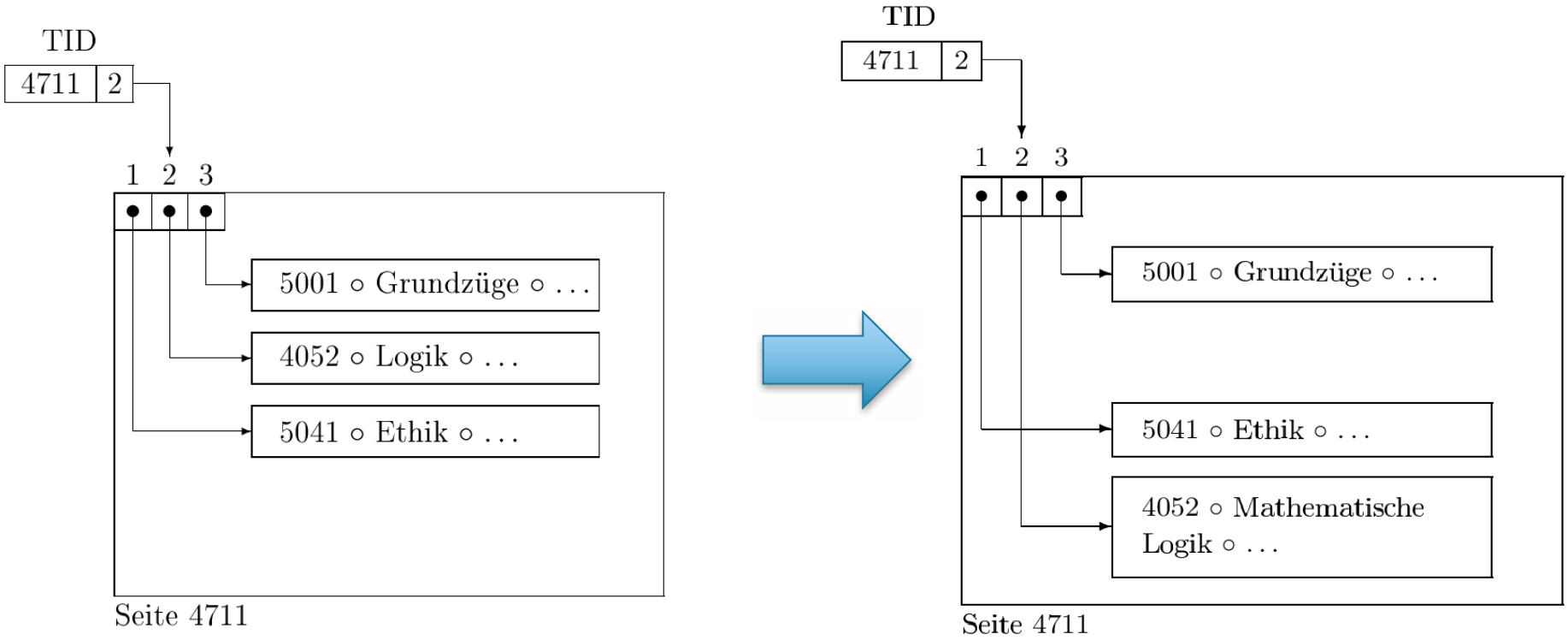
- Verschiedene mögliche Layouts:







Speicherung von Tupeln auf Seiten



- Seiteninterne Datensatztabelle
  - ermöglicht einfache interne Reorganisation

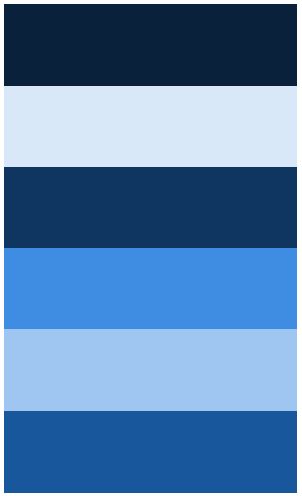
Verdrängen von Tupeln auf andere Seiten

- 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 verweist
- Was passiert bei mehrfachem Verschieben?
  - Verweis in der ursprünglichen Seite wird angepasst.

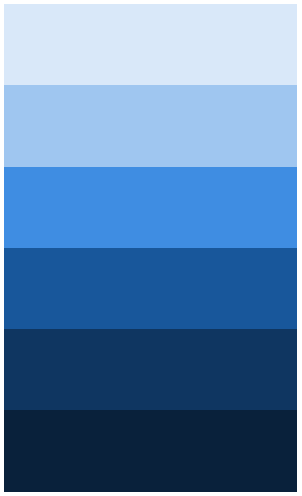
  ➔ Länge der Verweiskette auf zwei beschränkt

Organisation der Dateien

- Haufen
  - Einfachster Weg eine Datei zu organisieren ist neu ankommende Sätze in die Seite (den Block) am Ende der Datei einzufügen.
  - Einfügen ist sehr effizient.
  - Suche ist teuer.



- Sortierte Dateien
  - Gegeben ein Schlüssel anhand dessen Sätze geordnet werden können
  - Effizientere Suche (binär) - Obwohl so nicht realisiert (siehe z.B. B+ Baum)



Beispiel: Sortierte Datei (hier nach Name)

	Name	MatrNr	Semester
Block 1	Aaron	443421	10
	Adam	233499	1
	…		
	Acosta	561921	1
Block 2	Allen	581722	9
	Anderson	339163	8
	…		
	Archer	965492	1
Block 3	Arnold	672961	3
	Arnold	759311	1
	…		
	Atkins	173522	8
	.		
	.		
	.		
Block n	Wright	672963	4
	Wyatt	197646	7
	…		
	Zimmer	524145	12

Suche in Dateien

"If you don't find it in the index,
look very carefully through the entire catalog"
- Sears, Roebuck, and Co., Consumers' Guide, 1897

(aus dem Buch von Ramakrishnan & Gehrke)

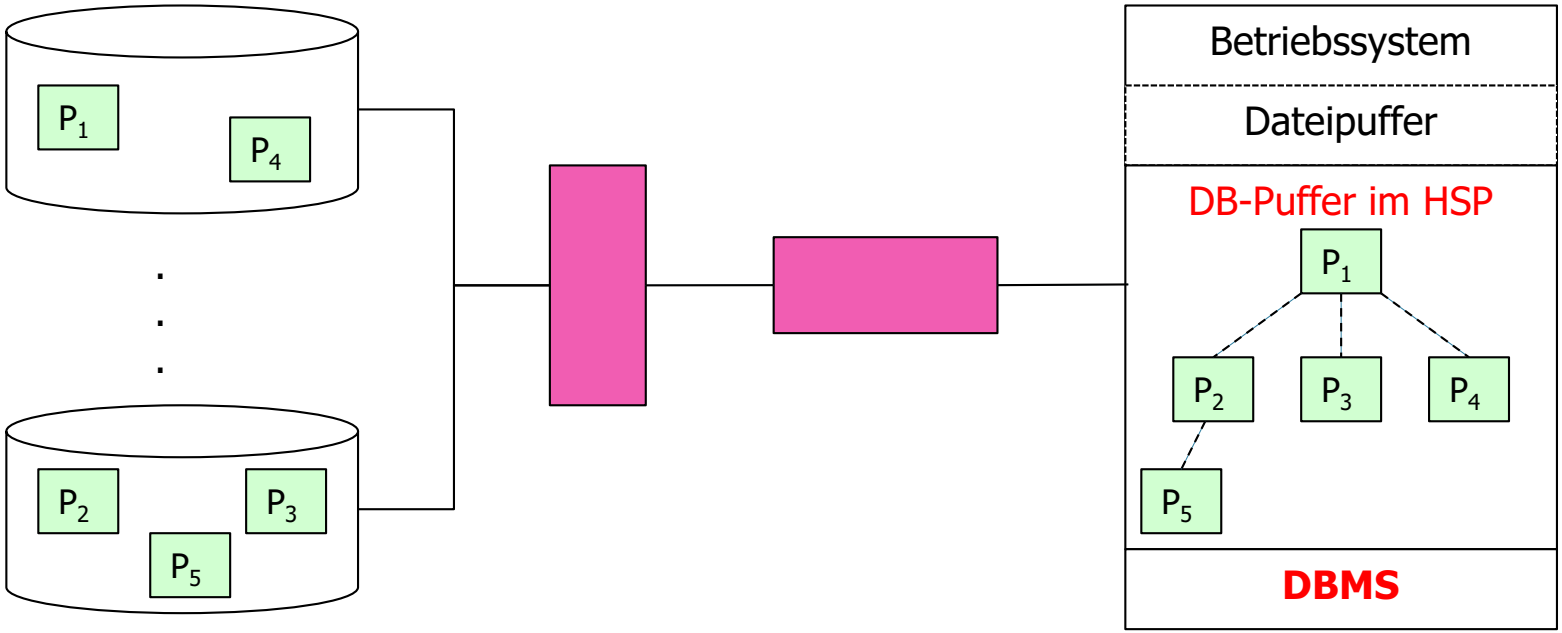
- Lineare Suche
- Binäre Suche in sortierten Dateien
- Suche via Indexen

Grundidee eines Index

- Abbildung: Schlüssel → Menge von Einträgen
- Beispiele:
  - MatrikelNummer → persönliche Daten des Studenten
  - PLZ → Name und andere Informationen einer Stadt
  - Term → alle Dokumente in denen dieser Term enthalten ist
- Natürlich möchte man bei diesen häufig auftretenden Anfragen die Antwortzeit gering halten.
- Dafür wird ein Index angelegt: Ein Index materialisiert diese Abbildung!
- Betrachten wir später in diesem Kapitel.

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

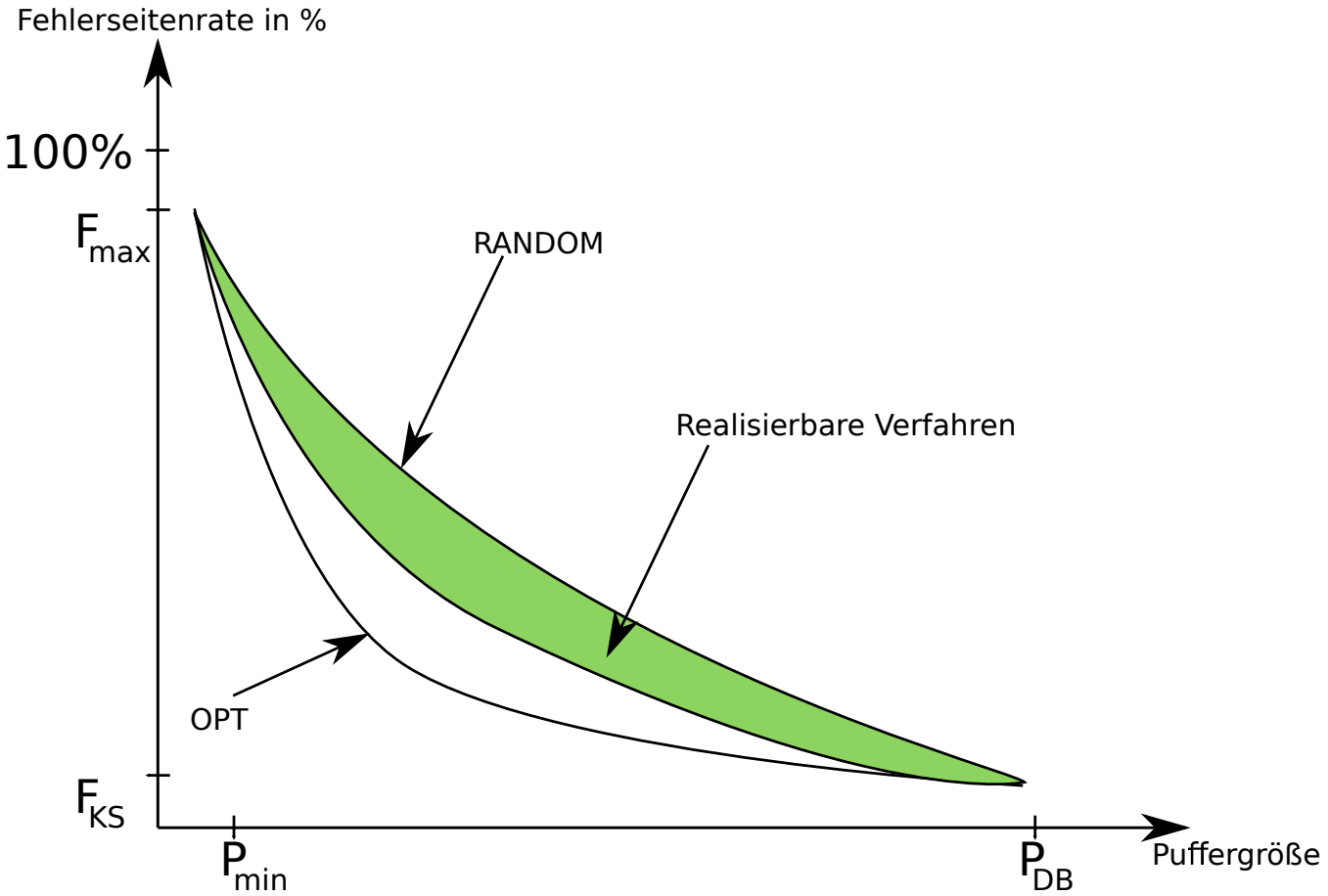
Empfehlung zum Thema Datenbank-Pufferverwaltung: Das Buch von Härder und Rahm: *"Datenbanksysteme - Konzepte und Techniken der Implementierung"* ist hier sehr ausführlich.

Ersetzungsstrategien: Konzept und Klassifizierung

- Wenn eine Seite nicht im Puffer auffindbar ist wird sie in den 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

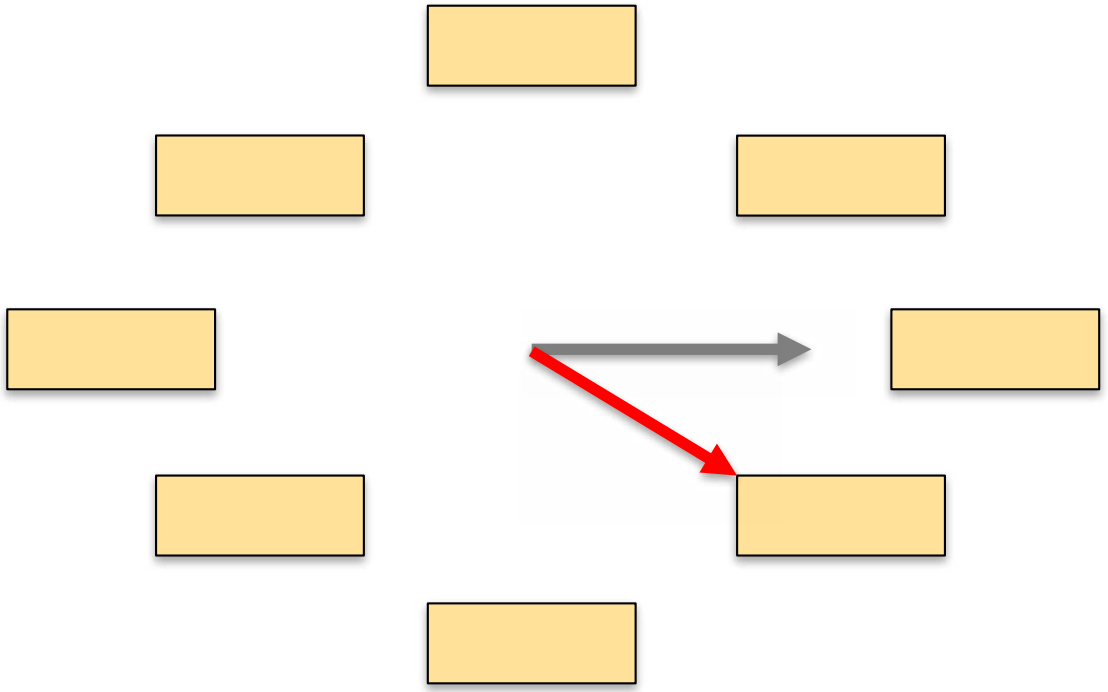


Illustration: Kreisförmige Anordnung. Zeiger verweist auf den ältesten Eintrag. Grauer Zeiger= alt. Roter Zeiger=neu.

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

  $$r_1, r_2, \dots, r_t, \dots$$

  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.

Weitere Ersetzungsstrategien werden in der VL Datenbanksysteme im Wintersemester vorgestellt.

Zusammenfassung

- Auswertung von Anfragen
  - Anfrageplan besteht aus (logischen) Operatoren
  - Verschiedene Implementierungen (physische Operatoren), basierend auf Iteratorkonzept
  - Auswahl von physischen Operatoren anhand von Kostenmodell
- Vereinfachte E/A-Architektur: Zweistufige Speicherhierarchie
  - Externspeicher (groß, billig, nicht-flüchtig) zur langfristigen Speicherung der Daten erforderlich
    - Kostenmodell für wahlfreien und sequentiellen Zugriff
    - Sequentieller Zugriff ist schneller!
  - Physische Organisation von Datenbanken
    - Speicherung von Sätzen auf Seiten
  - DB-Pufferverwaltung
    - Ersetzungsstrategien