ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Architektur und Implementierung von Datenbanksystemen WS 2005

Dr. Jens Dittrich

jens.dittrich@inf

www.inf.ethz.ch/~jensdi

Institut für Informationssysteme

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Speichersystem

2. Dateien und Blöcke

Aufgaben der Externspeicherverwaltung

- Verwaltung externer Speichermedien
- Verbergen von Geräteeigenschaften
- Abbildung von physischen Blöcken auf Dateien
- Kontrolle des Datentransfers von/zum DB-Puffer
- ggf. Blockchiffrierung
- ggf. Realisierung einer mehrstufigen Speicherhierarchie
- ggf. Fehlertoleranzmassnahmen (falls kein RAID verwendet wird)

Kontrolle der E/A-Operationen

- **Szenario**: Festplatte geht im laufenden Betrieb kaputt
- Frage: Wurde Block 42 auf die Festplatte geschrieben?
  - Ja
  - Nein
  - partiell (autsch!)
- Wie stellt man dies fest?

Kontrolle der E/A-Operationen

- Liegt ein Block komplett in einem Sektor, kann die Festplatte den Zustand des Blocks durch Parity-Informationen herausfinden.
- Bei einem Head-Crash muss das aber nicht funktionieren...
- Zur Sicherheit
  - Parity-Bit am Anfang und Ende des Blocks reservieren
    - Beide Bits auf 0
    - Invertieren der Bits bei jedem Schreibvorgang
    - Unterscheiden sich die Bits --> Block wurde partiell geschrieben
  - Schreiben mit Logging (logged write):
    1. Schreibe alten Inhalt des Blocks an sicheren Platz
    2. Schreibe neuen Block an alten Platz

Bemekung: Jedes “anständige” DBMS loggt Daten und Operationen (dazu später mehr)

Motivation für Dateikonzept

- selektive Aktivierung von Dateien
- temporäre Dateien
- Einsatz unterschiedlicher Speichermedien
- kurze Adresslängen

DB-Speicher = Menge von Dateien

Realisierung eines Dateisystems

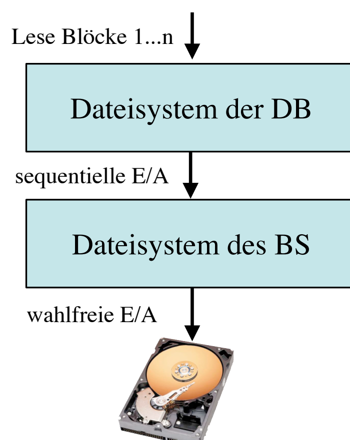
- Dateisystem enthält Katalog für alle Dateien
- Dateideskriptor: Name, Owner, Zugriffskontrollliste (ACL), Grösse, Zeitstempel, etc.
- Freispeicherverwaltung für Externspeicher (Bitlisten)
- Einheit des Zugriffs: Block
  - feste Blocklänge pro Datei
  - Unterstützung für sequentielle E/A-Vorgänge

Das alles leistet in der Regel das Betriebssystem!

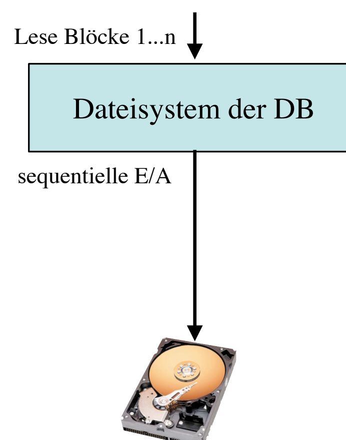
raw-devices

- Betriebssystem realisiert bereits ein Dateisystem (DS)
- realisiert die DB hierauf ein weiteres Dateisystem kommt es zu unangenehmen Effekten:

Dateisystem über Dateisystem



Dateisystem über raw-device



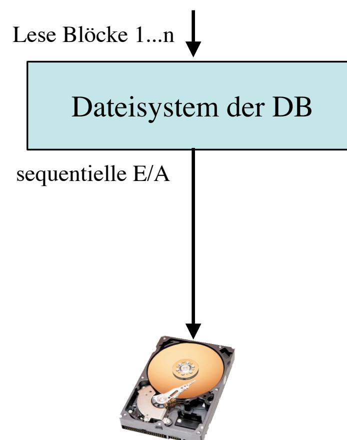
Warum nicht einfach das Dateisystem des Betriebssystems übernehmen?

- Leistungsverhalten nicht auf DB zugeschnitten
- ungenügende Unterstützung für Recovery
- müsste auf jeder Plattform separat für DB-Bedürfnisse erweitert werden
- Dateisystem-Implementierungen proprietär

Vorteile von raw-device Variante

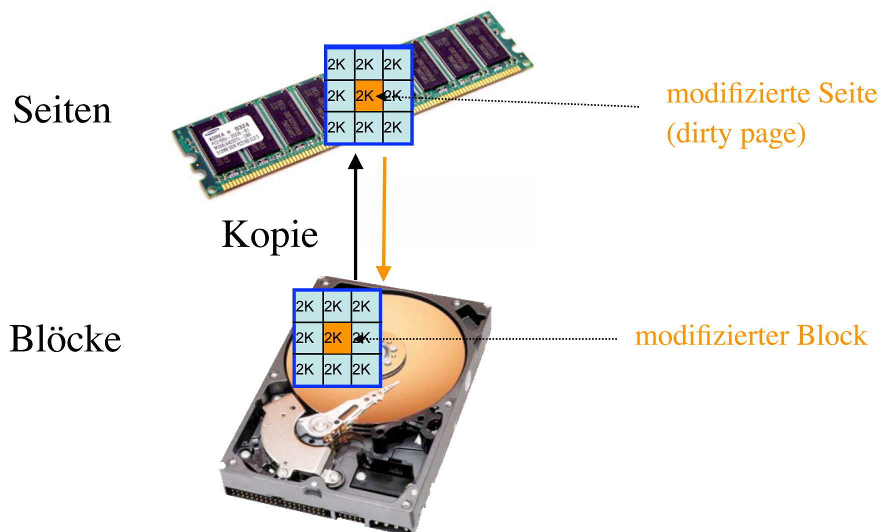
- DB-Hersteller hat volle Kontrolle über Dateisystem
- Eine Implementierung für alle Plattformen
- bessere Kontrolle:
  seq. vs. wahlfreie E/A
  (Controller kann natürlich Blöcke ummappen)
- weitere Vorteile (siehe DB-Puffer)

Dateisystem über raw-device



Seiten und Blöcke

- Seite = Repräsentation eines Blocks im Hauptspeicher



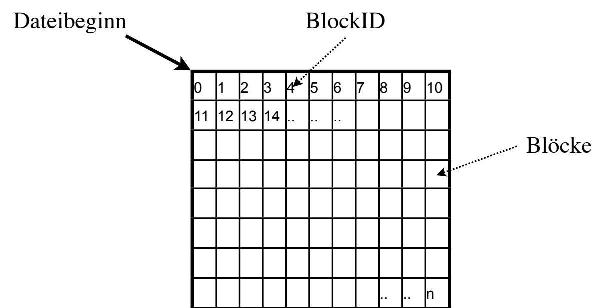
Blockzuordnung bei Magnetplatten

- realisiert Abbildung

  <BlockID> ⟶ <file,offset>

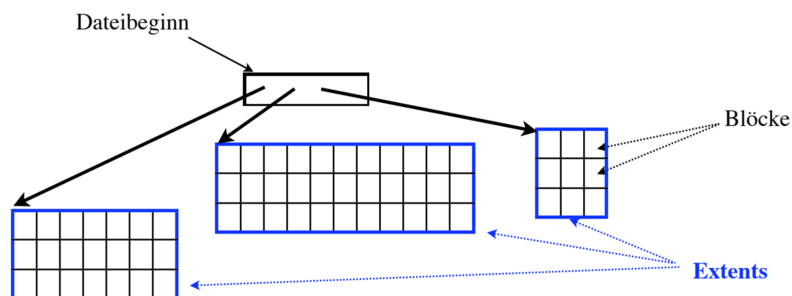
- beeinflusst in hohem Masse E/A-Performanz
- beeinflusst Flexibilität des Dateikonzeptes (Anwachsen von Dateien, Verwaltung leerer Blöcke)
- 3 wesentliche Ansätze
  - Statische Dateizuordnung
  - Dynamische Extent-Zuordnung
  - Dynamische Blockzuordnung
- Praxisbeispiel: Unix

Statische Dateizuordnung



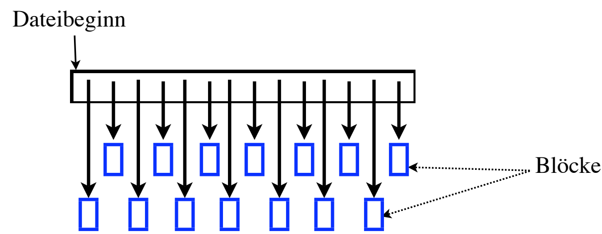
- Vorteile
  - direkte Adressierung
  - Blöcke physisch benachbart
  - sehr gute Performanz für sequentielle E/A
- Nachteile
  - Datei muss komplett reserviert werden
  - kein dynamisches Wachstum

Dynamische Extent-Zuordnung



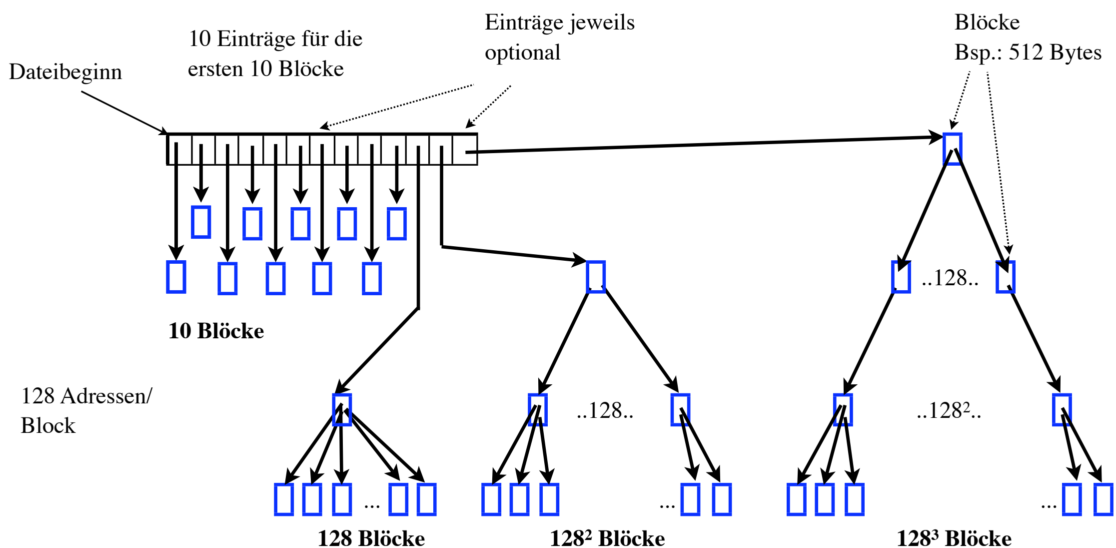
- Vorteile
  - Erlaubt Allokation von Extents auf unterschiedlichen Speichermedien
  - akzeptable Performanz für sequentielle E/A
    (innerhalb eines Extents sowieso)
- Nachteile
  - Verschnitt
  - Fragmentierung

Dynamische Blockzuordnung



- Vorteile
  - maximale Flexibilität
  - kein Verschnitt
- Nachteile
  - schlechte Performanz für sequentielle E/A

Wie macht es UNIX?



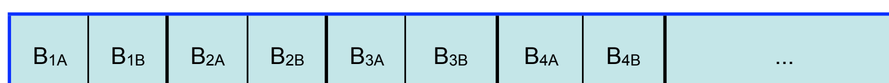
$$10 + 128 + 128^2 + 128^3 = 2{,}113{,}674 \text{ Datenblöcke}$$

Einbringstrategien für Änderungen

- Wie wird eine geänderte Seite auf den Block auf der Festplatte zurückgeschrieben?
  - Direkte Einbringstrategien
    - Seite wird an existierende Position des zugehörigen Blocks auf der Festplatte kopiert
    - Was passiert, wenn hierbei die Festplatte kaputt geht?
    - Antwort: ...
  - Indirekte Einbringstrategien
    - Alte Version des Blocks wird aufgehoben, bis die Transaktion auf der Seite abgeschlossen ist
    - Vereinfacht Rücksetzen der Transaktion
    - Bsp.: Twin Block, Schattenspeicherkonzept, Zusatzdateikonzept

Twin-Block-Verfahren

- Ziel: atomares Einbringen von Änderungen zu einem Sicherungspunkt
- Idee: halte 2 Versionen von jeder Seite
  - alte, konsistente Version
  - neue, inkonsistene Version
- Geht beim Ändern etwas schief, hat man immer noch die alte Version der Seiten
- Umschaltung erfolgt durch einfaches globales (materialisiertes) Bit
- Nachteil: Verdopplung des Speicherbedarfs



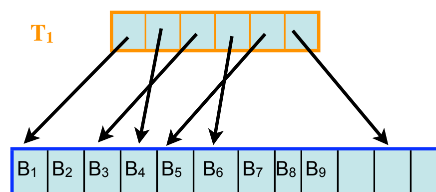
Schattenspeicherkonzept

- Ziel: atomares Einbringen von Änderungen zu einem Sicherungspunkt
- Idee: Halte 2 Versionen für jede **geänderte** Seite
  - alte, konsistente Version
  - neue, inkonsistente Version
- Alter, konsistenter Zustand wird in “Schattenseiten” gespeichert

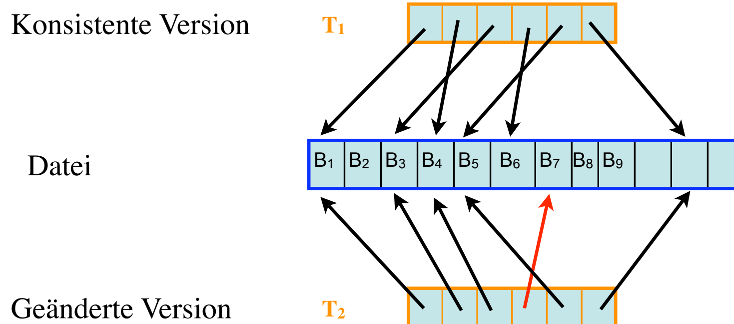
Schattenspeicherkonzept

Konsistente Version

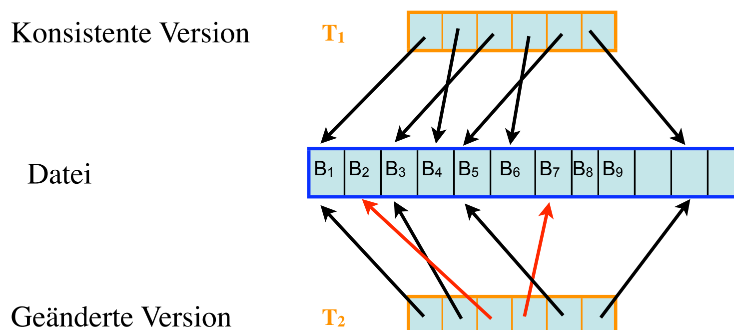
Datei



Schattenspeicherkonzept: Insert&Update

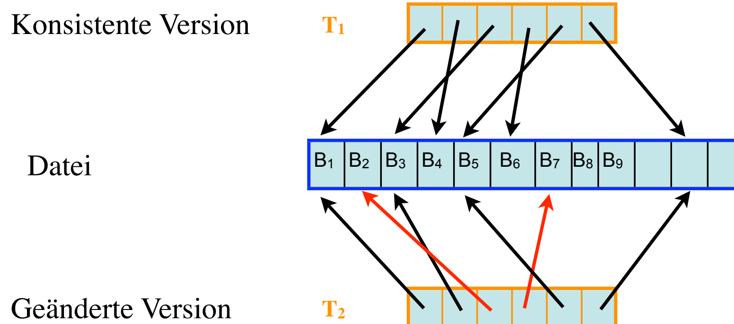


Schattenspeicherkonzept: Insert&Update



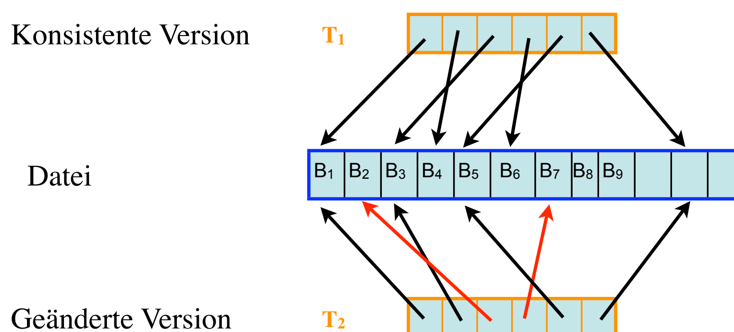
Geänderte Blöcke werden an freien Platz in der Datei geschrieben
D.h. nur von geänderten Blöcken werden Kopien erzeugt

Schattenspeicherkonzept: Crash



1. T_2 wegwerfen, alte T_1 wird wieder aktuelle Version
2. geänderte Blöcke in der Datei freigeben

Schattenspeicherkonzept: Persistieren der Änderungen



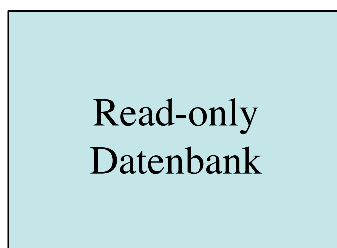
1. Alle geänderten Blöcke schreiben
2. T_2 schreiben
3. Auf T_2 atomar umschalten

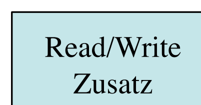
Bewertung des Schattenspeicherkonzepts

- Vorteile
  - Verdopplung des Speicherbedarfs nur für geänderte Blöcke
  - Zurücksetzen relativ einfach und schnell
  - nach katastrophalem Fehler ist die DB mit höhere Wahrscheinlichkeit in einem brauchbaren Zustand als bei einer direkten Einbringstrategie
- Nachteile
  - Hilfsstrukturen können sehr gross werden (> 1 Block)
  - Hilfsstrukturen müssen eventuell auf Festplatte ausgelagert werden
  - Sequentieller Zugriff auf Daten wird mit der Zeit zerstört
  - Konzept nicht geeignet für grosse Datenbanken

Zusatzdatei-Konzept

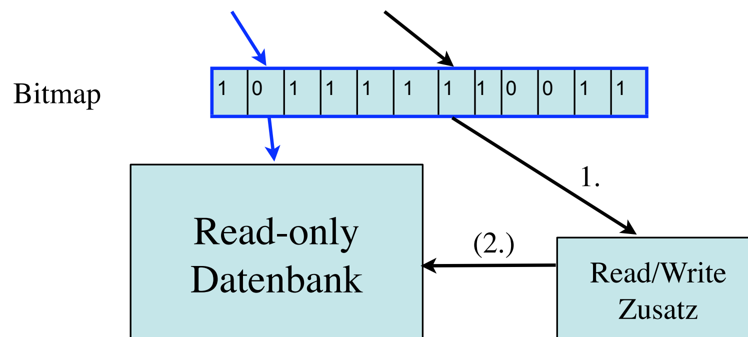
- Idee
  - 2 Versionen der Datenbank
    - Read-only: konsistenter alter Zustand
    - Read/Write: Änderungen
  - spiele Änderungen zu festen Zeitpunkten in Datenbank ein
  - Analogie: Verlag sammelt Korrekturen und erstellt hiermit neue Auflage eines Buchs





Zusatzdatei-Konzept

- In welcher Datenbank liegt die jüngste Version des Blocks?
- Algorithmus (Bloom-Filter)
  - Hash-Funktion h(B) => Bit
  - Bit = 0: Block in Read-only Datenbank
  - Bit = 1: Block vielleicht in Read/Write Zusatz



Bewertung des Zusatzdatei-Konzepts

- Vorteile
  - leicht zu implementieren
  - leicht erweiterbar um Transkationen
  - Inhalt der Zusatzdatei entspricht inkrementellem Backup für Datensicherung
  - gut geeignet für Systeme mit überwiegend Leseanforderungen (Read-mostly): Suchmaschinen, Data Warehouses
  - Merge der beiden Versionen kann jedesmal ausgenutzt werden, um Blöcke zu sequentialisieren...
- Nachteile
  - Sperren während des Merge (kann man leicht beheben)

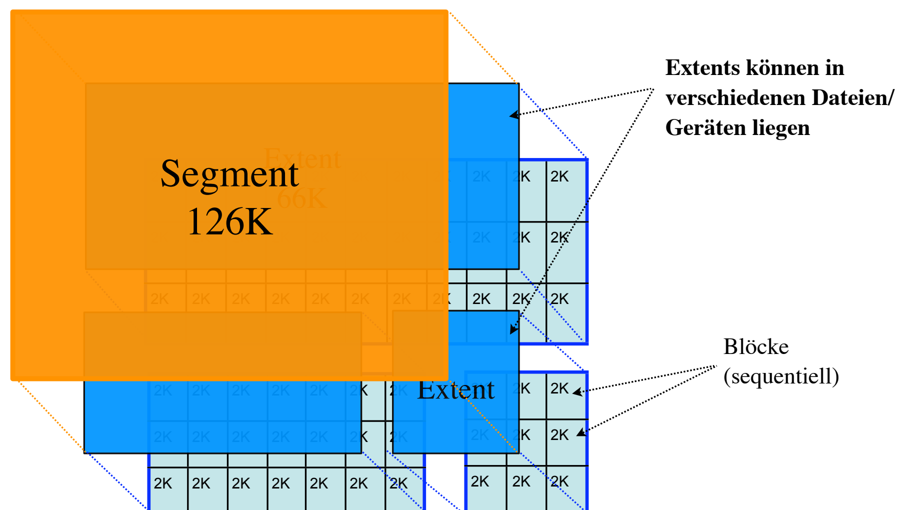
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Speichersystem

3. Blöcke, Extents, Segmente, Tablespaces

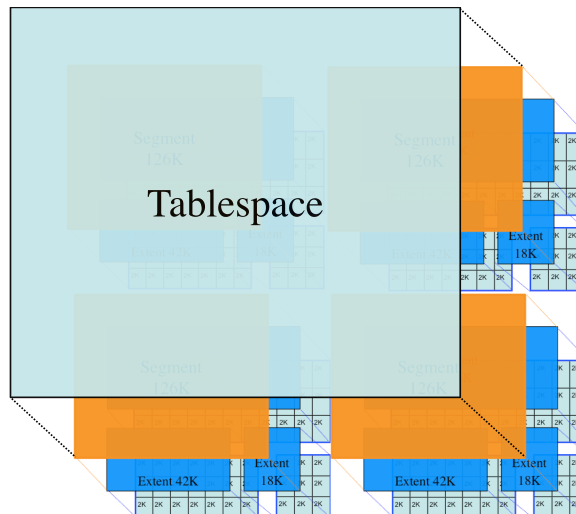
Segmente

- Segment = Menge von Extents

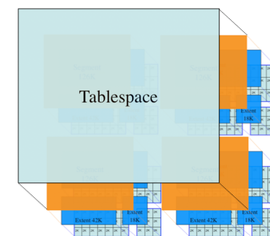


Tablespaces (Oracle)

- Tablespace = Menge von Segmenten



Tablespaces (Oracle)



- 1:n Beziehung zu Dateien
- Administrator verwaltet Tablespaces und steuert damit, welche Tabellen wo abgelegt werden
- Anlegen eines Tablespace

```
CREATE TABLESPACE mein_tablespace DATAFILE
file1.dat SIZE 40 MB, file2.dat SIZE 30 MB
```

- Erweitern eines Tablespace (erfolgt nicht unbedingt automatisch)

```
ALTER TABLESPACE mein_tablespace ADD DATAFILE
file3.dat SIZE 20 MB
```

- Zuordnung zu Relationen

```
CREATE TABLE customer(...) TABLESPACE mein_tablespace
```

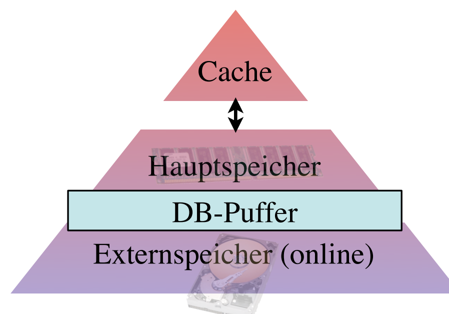
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Speichersystem

4. DB Pufferverwaltung

Aufgaben der Pufferverwaltung

- DB-Puffer sitzt zwischen Externspeicher und Hauptspeicher
- Hält Menge von *Pufferrahmen* (Speicherbereiche der Grösse: 1 Seite)
- Ziel: Verringerung der E/A-Zugriffe durch Nutzen *zeitlicher* Lokalität



- Nur eine beschränkte Menge von Seiten kann im Puffer gehalten werden

Allgemeine Arbeitsweise des DB-Puffers

- GET P_x
  - Stelle Seite P_x im Puffer bereit und gibt Referenz auf Seite zurück
- FIX P_x
  - Seite P_x darf nicht verdrängt werden
- UNFIX P_x
  - Seite P_x darf verdrängt werden
- PAGE_IN_BUFFER P_x
  - Gibt true zurück, falls Seite P_x im Puffer vorhanden ist
    Implementierung: Hash-Tabelle
- CHOOSE_PAGE
  - Wähle Seite zur Verdrängung aus
  - Gib Referenz auf Seite zurück
    Implementierung: Seitenersetzungsverfahren (s.u.)

Implementierung von GET

	Kosten
■ GET P_x	
▪ PAGE_IN_BUFFER(P_x):	
- FIX P_x	billig
- Gebe Referenz auf Seite zurück	
▪ Sonst	
- Falls kein freier Rahmen im Puffer vorhanden:	
• P_i = CHOOSE_PAGE	billig
• Falls Seite P_i geändert wurde (dirty):	
◦ Schreibe Seite P_i auf Externspeicher	**teuer**
- Lade Block B_x vom Externspeicher in freien Pufferrahmen	**teuer**
- FIX P_x	billig
- Gebe Referenz auf Seite zurück	

Schreibstrategien

- **FORCE** (write-through)
  - schreibt geänderte Seiten spätestens beim commit zurück
  - Nachteil: hoher E/A-Aufwand
  - hohe Antwortzeiten von ändernden Transaktionen
  - einfache Recovery
- **NO FORCE** (write-back)
  - schreibe geänderte Seite erst bei Verdrängung aus dem DB-Puffer zurück
  - Achtung: Verdrängungszeitpunkt kann weit in der Zukunft liegen
    - Was passiert, wenn eine häufig geänderte Seite nicht verdrängt wird?
    - Was steht auf der Festplatte?

  ⇨ klappt nur in Kombination mit Logging

Lesestrategien

- **Preplanning**
  - analysiere Anwenderprogramme
  - Versuche zu lesende Seiten zu bestimmen
  - Nachteile
    - ungenaue Obermengen
    - klappt nicht immer
- **Prefetching**
  - nutze Clusterbildung der Daten aus
  - überlappe CPU- und E/A-Operationen
  - Nachteile
    - falsche Entscheidungen
    - Festplatte macht bereits read-ahead
    - bring nichts bei hohen Trefferraten (zusätzlicher Aufwand)

Lesestrategien

- **Demand Fetching**
  - fordere Seite dann an, wenn die Seite gebraucht wird
  - Vorteil: kein Zusatzaufwand
  - Nachteil: hohe Antwortzeiten, Anwendung muss auf Daten warten

Referenzmuster

- Referenzmuster = Referenzierungssequenz von Seiten
- **Sequentielle Suche**
  - Beispiel: table scan von l Seiten

    $P_1, P_2, P_3, P_4, \ldots, P_l$
- **Hierarchische Pfade**
  - Beispiel: index scan

    $P_1, P_{11}, P_{42}, P_{77}, P_{34}$
- **Zufällige Zugriffe**
  - Beispiel: RID-scan

    $P_{456}, P_{1124}, P_{422}, P_7, P_{343}$
- **Zyklische Pfade**
  - Beispiel: nested-loops join, innere Schleife der Länge n

    $P_1, P_2, \ldots, P_n, P_1, P_2, \ldots, P_n, P_1, P_2, \ldots, P_n$

Seitenersetzungsverfahren (Implementierung von CHOOSE_PAGE)

- Klassische Verfahren
  - least-recently used (LRU) vs. most-recently used (MRU)
  - first-in first out (FIFO)
  - least-frequently used (LFU)
  - clock, second-chance, gclock
  - least-reference density

LRU-k

- **Idee**
  Ersetze diejenige Seite, bei der der k-letzte Zugriff am längsten zurückliegt
- LRU-1 = LRU
- Historie einer Seite muss gespeichert werden (sogar noch dann, falls die Seite ersetzt wurde!)

Algorithmus

Sei $p_1,\ldots,p_n$ eine Referenzierungssequenz von Seiten

Dann ist

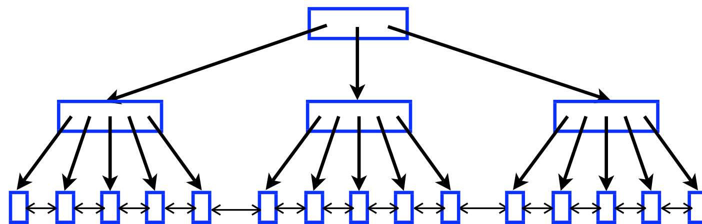
$b_t(P,k) = x$, wenn p_{t-x} den Wert P besitzt und es gibt genau k-1 Referenzen auf die Seite P in $p_{n-x+1},\ldots,p_n$

$b_t(P,k) =$ unendlich, sonst

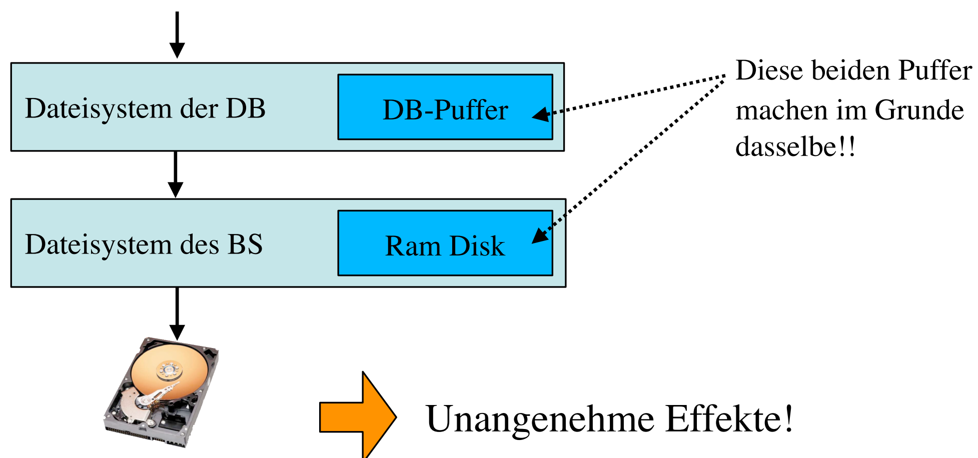
LRU-k wählt die Seite aus dem Puffer mit höchstem Wert für $b_t(P,k)$

LRU-k

- **Idee**
  Ersetze diejenige Seite, bei der der k-letzte Zugriff am längsten zurückliegt
- Beispiel
  B+-Baum, 5 Blätter pro Knoten, Zugriff auf ein Pfad + 5 Blätter
  Puffergrösse: 6 bzw. 8 Seiten
  LRU-1 vs. LRU-k?

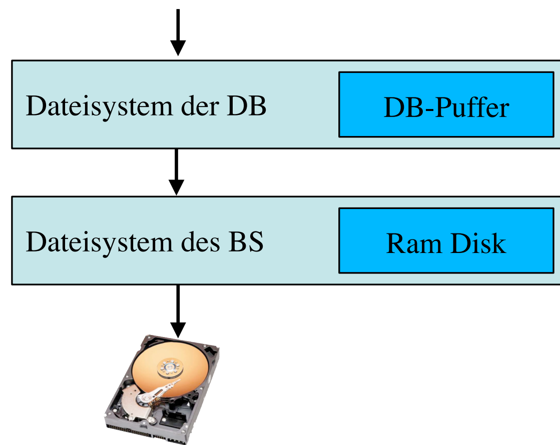


DB-Puffer und Virtueller Speicher



DB-Puffer und Virtueller Speicher

- Double Page Fault
  - die Seite P ist nicht im DB-Puffer
  - Seite P ist auch nicht in der Ram Disk
  - D.h. Seite aus dem Swap-Bereich oder von Platte lesen
- Wozu zwei Puffer?
- Woher weiss ich, dass das BS Seite P auf Platte geschrieben hat?



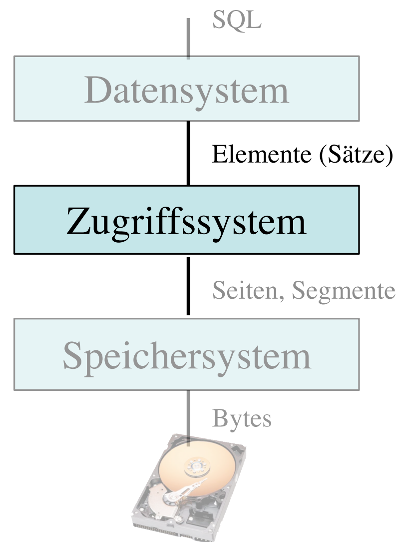
➡ raw disks!

Satz- vs. Seitenorientierte Pufferverwaltung

- Seitenorientiert
  - + geringer Verwaltungsaufwand
  - + einfach zu implementieren
  - + kein Verschnitt von Speicher
  - – Verschwendung durch Kopieren nutzloser Daten
- Satzorientiert (Warum die gesamte Seite speichern, wenn ich nur einen Satz brauche?)
  - – Verwaltungsaufwand pro Satz
  - – aufwendige Implementierung
  - – Kopieraufwand zum Isolieren der Sätze
  - – Fragmentierung des Speichers
  - – Verschnitt
  - + sehr gutes Nutzen des Speichers

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Nächste Woche



Zugriffssystem

1. Speicherungsstrukturen
2. Eindimensionale Zugriffspfade (Teil 1)