

Datenbanken 2

Einführung, Physische Datenorganisation

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

FB Computerwissenschaften
Universität Salzburg

Wintersemester 2015/16

Inhalt

- 1 Einführung
- 2 Speichermedien
- 3 Speicherzugriff
- 4 Datei Organisation

Inhalt

- 1 Einführung
- 2 Speichermedien
- 3 Speicherzugriff
- 4 Datei Organisation

Alle Infos zu Vorlesung und Proseminar:

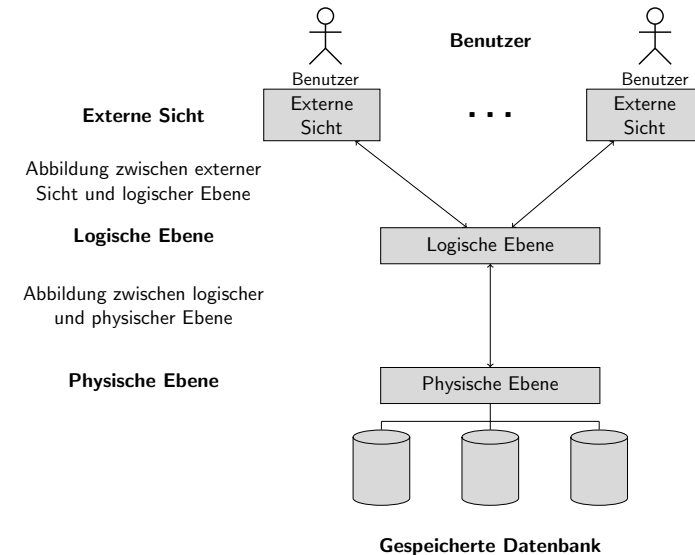
<http://dbresearch.uni-salzburg.at/teaching/2015ws/db2/>



Was erwartet Sie inhaltlich?

- **Datenbanken 1: Logische Ebene**
 - Konzeptioneller Entwurf (ER)
 - Relationale Algebra
 - SQL
 - Relationale Entwurfstheorie
- **Datenbanken 2: Physische Ebene**
 - Wie baue (programmiere) ich ein Datenbanksystem?
 - Daten müssen physisch gespeichert werden
 - **Datenstrukturen** und Zugriffs-**Algorithmen** müssen gefunden werden
 - SQL-Anfragen müssen in ausführbare Programme umgesetzt werden
 - Es geht um die **Effizienz** (schneller ist besser)

Die ANSI/SPARC Drei-Ebenen Architektur



Inhaltsübersicht Datenbanksysteme

1. **Physische Datenorganisation**
 - Speichermedien, Dateionisation
 - Kapitel 7 in Kemper und Eickler
 - Chapter 10 in Silberschatz et al.
2. **Indexstrukturen**
 - Sequentielle Dateien, B+ Baum, Statisches Hashing, Dynamisches Hashing, Mehrere Suchschlüssel, Indices in SQL
 - Kapitel 7 in Kemper und Eickler
 - Chapter 11 in Silberschatz et al.
3. **Anfragebearbeitung**
 - Effiziente Implementierung der (relationalen) Operatoren
 - Kapitel 8 in Kemper und Eickler
 - Chapter 12 in Silberschatz et al.
4. **Anfrageoptimierung**
 - Äquivalenzregeln und Äquivalenzumformungen, Join Ordnungen
 - Kapiteln 8 in Kemper und Eickler
 - Chapter 13 in Silberschatz et al.

Inhalt

- 1 Einführung
- 2 Speichermedien
- 3 Speicherzugriff
- 4 Datei Organisation

Speichermedien/1

- **Verschiedene Arten** von Speichermedien sind für Datenbanksysteme relevant.
- Speichermedien lassen sich in **Speicherhierarchie** anordnen.
- **Klassifizierung** der Speichermedien nach:
 - Zugriffsgeschwindigkeit
 - Kosten pro Dateneinheit
 - Verlässlichkeit
 - Datenverlust durch Stromausfall oder Systemabsturz
 - Physische Fehler des Speichermediums
 - Flüchtige vs. persistente Speicher
 - Flüchtig (volatile): Inhalt geht nach Ausschalten verloren
 - Persistent (non-volatile): Inhalt bleibt auch nach Ausschalten

Speichermedien/2

- **Cache**
 - flüchtig
 - am schnellsten und am teuersten
 - von System Hardware verwaltet
- **Hauptspeicher (RAM)**
 - flüchtig
 - schneller Zugriff (x0 bis x00 ns; $1 \text{ ns} = 10^{-9} \text{ s}$)
 - meist zu klein (oder zu teuer) um gesamte Datenbank zu speichern
 - mehrere GB weit verbreitet
 - Preise derzeit ca. 5 EUR/GB (DRAM)
 - Kapazitäten steigen ständig und Preis/Byte sinkt (ca. Faktor 2 alle 2-3 Jahre)

Speichermedien/3

- **Flash memory (SSD)**
 - persistent
 - lesen ist sehr schnell (x0 bis x00 μs ; $1 \mu\text{s} = 10^{-6} \text{ s}$)
 - hohe sequentielle Datentransferrate (bis 500 MB/s)
 - nicht-sequentieller Zugriff nur ca. 25% langsamer
 - Schreibzugriff langsamer und komplizierter
 - Daten können nicht überschrieben werden, sondern müssen zuerst gelöscht werden
 - nur beschränkte Anzahl von Schreib/Lösch-Zyklen sind möglich
 - Preise derzeit ca. 1 EUR/GB
 - Speichermedien: NAND Flash Technologie (Firmware: auch NOR)
 - weit verbreitet in Embedded Devices (z.B. Digitalkamera)
 - auch als EEPROM bekannt (Electrically Erasable Programmable Read-Only Memory)

Speichermedien/4

- **Festplatte**
 - persistent
 - Daten sind auf Magnetscheiben gespeichert, mechanische Drehung
 - sehr viel langsamer als RAM (Zugriff im ms-Bereich; $1 \text{ ms} = 10^{-3} \text{ s}$)
 - sequentielles Lesen: 25–100 MB/s
 - billig: Preise teils unter 0.1 EUR/GB
 - sehr viel mehr Platz als im Hauptspeicher; derzeit x00 GB - 4 TB
 - Kapazitäten stark ansteigend (Faktor 2 bis 3 alle 2 Jahre)
 - Hauptmedium für Langzeitspeicher: speichert gesamte Datenbank
 - für den Zugriff müssen Daten von der Platte in den Hauptspeicher geladen werden
 - direkter Zugriff, d.h., Daten können in beliebiger Reihenfolge gelesen werden
 - Diskette vs. Festplatte

Speichermedien/5

- **Optische Datenträger**

- persistent
- Daten werden optisch via Laser von einer drehenden Platte gelesen
- lesen und schreiben langsamer als auf magnetischen Platten
- sequentielles Lesen: 1 Mbit/s (CD) bis 400 Mbit/s (Blu-ray)
- verschiedene Typen:
 - CD-ROM (640 MB), DVD (4.7 to 17 GB), Blu-ray (25 to 129 GB)
 - write-once, read-many (WORM) als Archivspeicher verwendet
 - mehrfach schreibbare Typen vorhanden (CD-RW, DVD-RW, DVD-RAM)
- Jukebox-System mit austauschbaren Platten und mehreren Laufwerken sowie einem automatischen Mechanismus zum Platten wechseln – "CD-Wechsler" mit hunderten CD, DVD, oder Blu-ray disks

Speichermedien/6

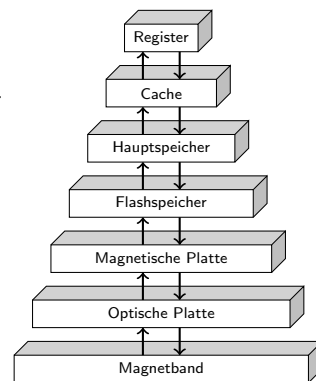
- **Band**

- persistent
- Zugriff sehr langsam, da sequentieller Zugriff
- Datentransfer jedoch z.T. wie Festplatte (z.B. 120 MB/s, komprimiert 240MB/s)
- sehr hohe Kapazität (mehrere TB)
- sehr billig (ab 10 EUR/TB)
- hauptsächlich für Backups genutzt
- Band kann aus dem Laufwerk genommen werden
- Band Jukebox für sehr große Datenmengen
 - x00 TB (1 terabyte = 10^{12} bytes) bis Petabyte (1 petabyte = 10^{15} bytes)

Speichermedien/7

- Speichermedien können hierarchisch nach Geschwindigkeit und Kosten geordnet werden:

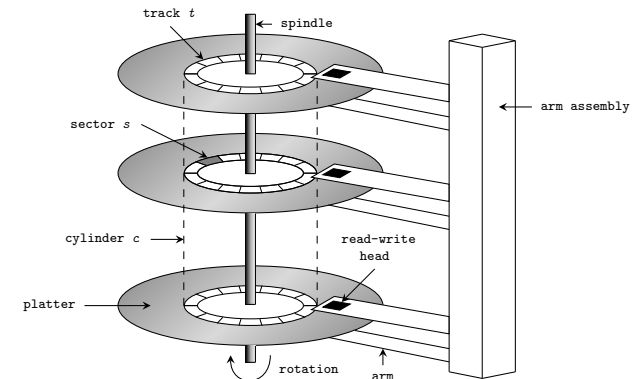
- **Primärspeicher:** flüchtig, schnell, teuer
 - e.g., Cache, Hauptspeicher
- **Sekundärspeicher:** persistent, langsamer und günstiger als Primärspeicher
 - e.g., Magnetplatten, Flash Speicher
 - auch Online-Speicher genannt
- **Tertiärspeicher:** persistent, sehr langsam, sehr günstig
 - e.g., Magnetbänder, optischer Speicher
 - auch Offline-Speicher genannt



- Datenbank muss mit Speichermedien auf allen Ebenen umgehen

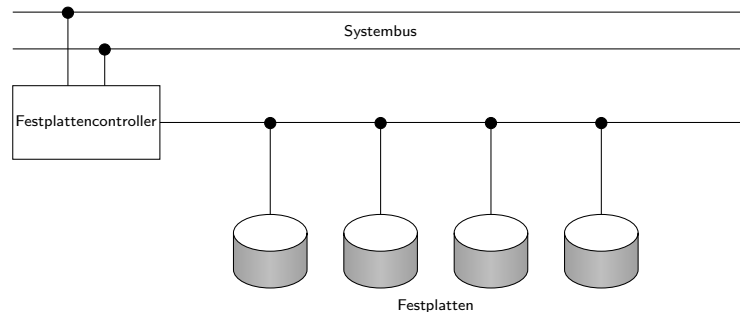
Festplatten/1

- Meist sind Datenbanken auf magnetischen Platten gespeichert, weil:
 - die Datenbank zu groß für den Hauptspeicher ist
 - der Plattenspeicher persistent ist
 - Plattenspeicher billiger als Hauptspeicher ist
- Schematischer Aufbau einer Festplatte:



Festplatten/2

- **Controller:** Schnittstelle zwischen Computersystem und Festplatten:
 - übersetzt high-level Befehle (z.B. bestimmten Sektor lesen) in Hardware Aktivitäten (z.B. Disk Arm bewegen und Sektor lesen)
 - für jeden Sektor wird Checksum geschrieben
 - beim Lesen wird Checksum überprüft



Festplatten/3

Drei Arbeitsvorgänge für Zugriff auf Festplatte:

- **Spurwechsel** (seek time): Schreib-/Lesekopf auf richtige Spur bewegen
- **Latenz** (rotational latency): Warten, bis sich der erste gesuchte Sektor unter dem Kopf vorbeibewegt.
- **Lesezeit:** Sektoren lesen/schreiben, hängt mit Datenrate (data transfer rate) zusammen

$$\text{Zugriffszeit} = \text{Spurwechsel} + \text{Latenz} + \text{Lesezeit}$$

Festplatten/4

Performance Parameter von Festplatten

- **Spurwechsel:** gerechnet wird mit mittlerer Seek Time (=1/2 worst case seek time, typisch 2-10ms)
- **Latenz:**
 - errechnet sich aus Drehzahl (5400rpm-15000rpm)
 - rpm = revolutions per minute
 - Latenz [s] = $60 / \text{Drehzahl [rpm]}$
 - mittlere Latenz: 1/2 worst case (2ms-5.5ms)
- **Datenrate:** Rate mit der Daten gelesen/geschrieben werden können (z.B. 25-100 MB/s)
- **Mean time to failure (MTTF):** mittlere Laufzeit bis zum ersten Mal ein Hardware-Fehler auftritt
 - typisch: mehrere Jahre
 - keine Garantie, nur statistische Wahrscheinlichkeit

Festplatten/5

- **Block:** (auch "Seite") zusammenhängende Reihe von Sektoren auf einer bestimmten Spur
- **Interblock Gaps:** ungenützter Speicherplatz zwischen Sektoren
- ein Block ist eine **logische Einheit** für den Zugriff auf Daten.
 - Daten zwischen Platte und Hauptspeicher werden in Blocks übertragen
 - Datenbank-Dateien sind in Blocks unterteilt
 - Block Größen: 4-16 kB
 - kleine Blocks: mehr Zugriffe erforderlich
 - große Blocks: Ineffizienz durch nur teilweise gefüllte Blocks

Integrierte Übung 1.1

Betrachte folgende Festplatte: Sektor-Größe $B = 512$ Bytes, Interblock Gap Size $G = 128$ Bytes, Sektoren/Spur $S = 20$, Spuren pro Scheibenseite $T = 400$, Anzahl der beidseitig beschriebenen Scheiben $D = 15$, mittlerer Spurwechsel $sp = 30ms$, Drehzahl $dz = 2400rpm$.

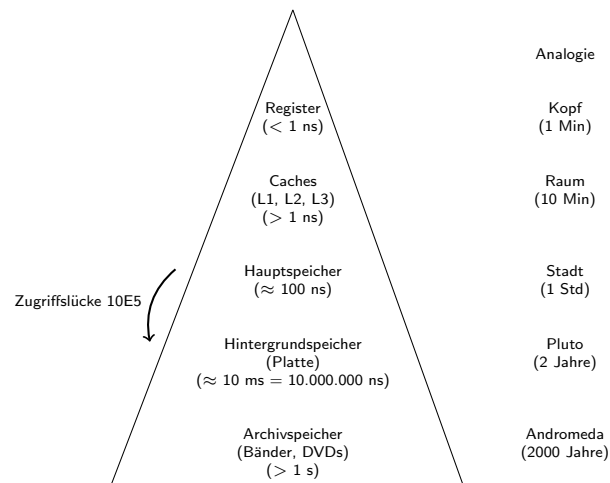
Bestimme die folgenden Werte:

- gesamte Kapazität der Festplatte
- nutzbare Kapazität der Festplatte
- mittlere Zugriffszeit (1 Sektor lesen)

Inhalt

- Einführung
- Speichermedien
- Speicherzugriff**
- Datei Organisation

Speicherhierarchie



Platten Zugriff Optimieren

- Wichtiges Ziel von DBMSs: Transfer von Daten zwischen Platten und Hauptspeicher möglichst effizient gestalten.
 - optimieren/minimieren der Anzahl der Zugriffe
 - minimieren der Anzahl der Blöcke
 - so viel Blöcke als möglich im Hauptspeicher halten (\rightarrow Puffer Manager)
- Techniken zur Optimierung des Block Speicher Zugriffs:
 - Disk Arm Scheduling
 - Geeignete Dateistrukturen
 - Schreib-Puffer und Log Disk

Block Speicher Zugriff/3

- **Disk Arm Scheduling:** Zugriffe so ordnen, dass Bewegung des Arms minimiert wird.
- **Elevator Algorithm** (Aufzug-Algorithmus):
 - Disk Controller ordnet die Anfragen nach Spur (von innen nach außen oder umgekehrt)
 - Bewege Arm in eine Richtung und erledige alle Zugriffe unterwegs bis keine Zugriffe mehr in diese Richtung vorhanden sind
 - Richtung umkehren und die letzten beiden Schritte wiederholen

Block Speicher Zugriff/4

- **Datei Organization:** Daten so in Blöcken speichern, wie sie später zugegriffen werden.
 - z.B. verwandte Informationen auf benachbarten Blöcken speichern
- **Fragmentierung:** Blöcke einer Datei sind nicht hintereinander auf der Platte abgespeichert
 - Gründe für Fragmentierung sind z.B.
 - Daten werden eingefügt oder gelöscht
 - die freien Blöcke auf der Platte sind verstreut, d.h., auch neue Dateien sind schon zerstückelt
 - sequentieller Zugriff auf fragmentierte Dateien erfordert erhöhte Bewegung des Zugriffsarm
 - manche Systeme erlauben das Defragmentieren des Dateisystems

Block Speicher Zugriff/5

Schreibzugriffe können asynchron erfolgen um Throughput (Zugriffe/Sekunde) zu erhöhen

- **Persistente Puffer:** Block wird zunächst auf persistenten RAM (RAM mit Batterie-Backup oder Flash Speicher) geschrieben; der Controller schreibt auf die Platte, wenn diese gerade nicht beschäftigt ist oder der Block zu lange im Puffer war.
 - auch bei Stromausfall sind Daten sicher
 - Schreibzugriffe können geordnet werden um Bewegung des Zugriffsarms zu minimieren
 - Datenbank Operationen, die auf sicheres Schreiben warten müssen, können fortgesetzt werden
- **Log Disk:** Eine Platte, auf die der Log aller Schreibzugriffe sequentiell geschrieben wird
 - wird gleich verwendet wie persistenter RAM
 - Log schreiben ist sehr schnell, da kaum Spurwechsel erforderlich
 - erfordert keine spezielle Hardware

Puffer Manager/1

- **Puffer:** Hauptspeicher-Bereich für Kopien von Platten-Blöcken
- **Puffer Manager:** Subsystem zur Verwaltung des Puffers
 - Anzahl der Platten-Zugriffe soll minimiert werden
 - ähnlich der virtuellen Speicherverwaltung in Betriebssystemen

Puffer Manager/2

- Programm fragt Puffer Manager an, wenn es einen Block von der Platte braucht.
- Puffer Manager **Algorithmus**:
 1. Programm fordert Plattenblock an
 2. Falls Block nicht im Puffer ist:
 - Der Puffer Manager reserviert Speicher im Puffer (wobei nötigenfalls andere Blöcke aus dem Puffer geworfen werden)
 - Ein rausgeworfener Block wird nur auf die Platte geschrieben, falls er seit dem letzten Schreiben auf die Platte geändert wurde.
 - Der Puffer Manager liest den Block von der Platte in den Puffer.
 3. Der Puffer Manager gibt dem anfordernden Programm die Hauptspeicheradresse des Blocks im Puffer zurück.
- Es gibt verschiedene Strategien zum Ersetzen von Blöcken im Puffer.

Ersetzstrategien für Pufferseiten/1

- **LRU Strategie** (least recently used): Ersetze Block der am längsten nicht benutzt wurde
 - Idee: Zugriffsmuster der Vergangenheit benutzen um zukünftiges Verhalten vorherzusagen
 - erfolgreich in Betriebssystemen eingesetzt
- **MRU Strategie** (most recently used): Ersetze zuletzt benutzten Block als erstes.
 - LRU kann schlecht für bestimmte Zugriffsmuster in Datenbanken sein, z.B. wiederholtes Scannen von Daten
- Anfragen in DBMSs haben wohldefinierte Zugriffsmuster (z.B. sequentielles Lesen) und das DBMS kann die Information aus den Benutzeranfragen verwenden, um zukünftig benötigte Blöcke vorherzusagen

Ersetzstrategien für Pufferseiten/2

- **Beispiel**: Berechne Join mit Nested Loops
 für jedes Tupel tr von R :
 für jedes Tupel ts von S :
 wenn ts und tr das Join-Prädikate erfüllen, dann ...
- Verschiedene Zugriffsmuster für R und S
 - ein R -Block wird nicht mehr benötigt, sobald das letzte Tuple des Blocks bearbeitet wurde; er sollte also sofort entfernt werden, auch wenn er gerade erst benutzt worden ist
 - ein S -Block wird nochmal benötigt, wenn alle anderen S -Blöcke abgearbeitet sind

Integrierte Übung 1.2

Zwischen R (2 Blöcke) und S (3 Blöcke) soll ein Nested Loop Join ausgeführt werden. Jeder Block enthält nur 1 Tupel.

Der Puffer fasst 3 Blöcke.

Betrachte den Puffer während des Joins und zähle die Anzahl der geladenen Blöcke für folgende Puffer-Strategien:

- LRU
- MRU

Welche Strategien eignen sich besser?

Ersetzstrategien für Pufferseiten/3

- **Pinned block:** Darf nicht aus dem Puffer entfernt werden.
 - z.B. der *R*-Block, bevor alle Tupel bearbeitet sind
- **Toss Immediate Strategy:** Block wird sofort rausgeworfen, wenn das letzte Tupel bearbeitet wurde
 - z.B. der *R* Block sobald das letzte Tupel bearbeitet wurde
- Gemischte Strategie mit Tipps vom Anfrageoptimierer ist am erfolgreichsten.
- MRU + Pinned Block ist die beste Strategie für den Nested Loop Join

Ersetzstrategien für Pufferseiten/4

Informationen für Ersatzstrategien in DBMSs:

- Zugriffspfade haben **wohldefinierte Zugriffsmuster** (z.B. sequentielles Lesen)
- **Information im Anfrageplan** um zukünftige Blockanfragen vorherzusagen
- **Statistik** über die Wahrscheinlichkeit, dass eine Anfrage für eine bestimmte Relation kommt
 - z.B. das Datenbankverzeichnis (speichert Schema) wird oft zugegriffen
 - Heuristic: Verzeichnis im Hauptspeicher halten

Inhalt

- 1 Einführung
- 2 Speichermedien
- 3 Speicherzugriff
- 4 **Datei Organisation**

Datei Organisation

- **Datei:** (file) aus logischer Sicht eine Reihe von Datensätzen
 - ein *Datensatz* (record) ist eine Reihe von Datenfeldern
 - mehrere Datensätze in einem Platten-Block
 - *Kopfteil* (header): Informationen über Datei (z.B. interne Organisation)
- **Abbildung von Datenbank in Dateien:**
 - eine Relation wird in eine Datei gespeichert
 - ein Tupel entspricht einem Datensatz in der Datei
- **Cooked vs. raw files:**
 - cooked: DBMS verwendet Dateisystem des Betriebssystems (einfacher, code reuse)
 - raw: DBMS verwaltet Plattenbereich selbst (unabhängig von Betriebssystem, bessere Performance, z.B. Oracle)
- **Fixe vs. variable Größe** von Datensätzen:
 - fix: einfach, unflexibel, Speicher-ineffizient
 - variable: komplizierter, flexible, Speicher-effizient

Fixe Datensatzlänge/1

- **Speicheradresse:** i -ter Datensatz wird ab Byte $m * (i - 1)$ gespeichert, wobei m die Größe des Datensatzes ist
- Datensätze an der **Blockgrenze:**
 - *überlappend:* Datensätze werden an Blockgrenze geteilt (zwei Blockzugriffe für geteilten Datensatz erforderlich)
 - *nicht-überlappend:* Datensätze dürfen Blockgrenze nicht überschreiten (freier Platz am Ende des Blocks bleibt ungenutzt)
- mehrere Möglichkeiten zum **Löschen des i -ten Datensatzes:**
 - verschiebe Datensätze $i + 1, \dots, n$ nach $i, \dots, n - 1$
 - verschiebe letzten Datensatz im Block nach i
 - nicht verschieben, sondern "Free List" verwalten

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Fixe Datensatzlänge/2

- **Free List:**
 - speichere Adresse des ersten freien Datensatzes im Kopfteil der Datei
 - freier Datensatz speichert Pointer zum nächsten freien Datensatz

→ eigenes Datenfeld für Free List Pointer (wie in der Abbildung dargestellt) ist nicht nötig, da der Speicherbereich des gelöschten Datensatzes verwendet wird

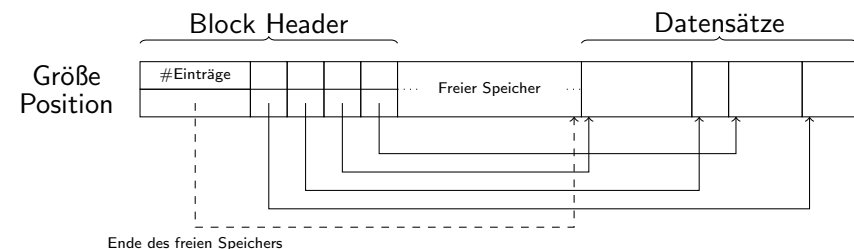
header			
record 0	A-102	Perryridge	400
record 1			
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4			
record 5	A-201	Perryridge	900
record 6			
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Variable Datensatzlänge/1

- **Warum Datensätze mit variabler Größe?**
 - Datenfelder variabler Länge (z.B., VARCHAR)
 - verschiedene Typen von Datensätzen in einer Datei
 - Platz sparen: z.B. in Tabellen mit vielen null-Werten (häufig in der Praxis)
- **Datensätze verschieben** kann erforderlich werden:
 - Datensätze können größer werden und im vorgesehenen Speicherbereich nicht mehr Platz haben
 - neue Datensätze werden zwischen existierenden Datensätzen eingefügt
 - Datensätze werden gelöscht (leere Zwischenräume verhindern)
- **Pointer soll sich nicht ändern:**
 - alle existierenden Referenzen zum Datensatz müssten geändert werden
 - das wäre kompliziert und teuer
- **Lösung:** Slotted Pages (TID-Konzept)

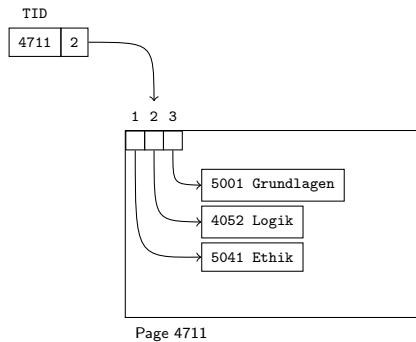
Slotted Pages/1

- **Slotted Page:**
 - Kopfteil (header)
 - freier Speicher
 - Datensätze
- **Kopfteil speichert:**
 - Anzahl der Datensätze
 - Ende des freien Speichers
 - Größe und Pointer auf Startposition jedes Datensatzes



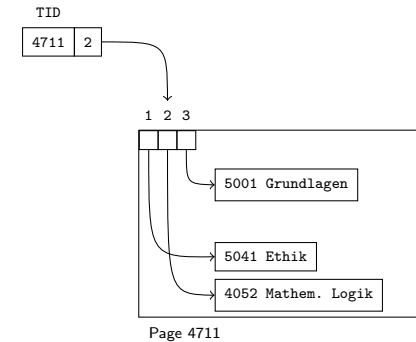
Slotted Pages/2

- TID: Tuple Identifier besteht aus
 - Nummer des Blocks (page ID)
 - Offset des Pointers zum Datensatz
- Datensätze werden **nicht direkt adressiert**, sondern über TID



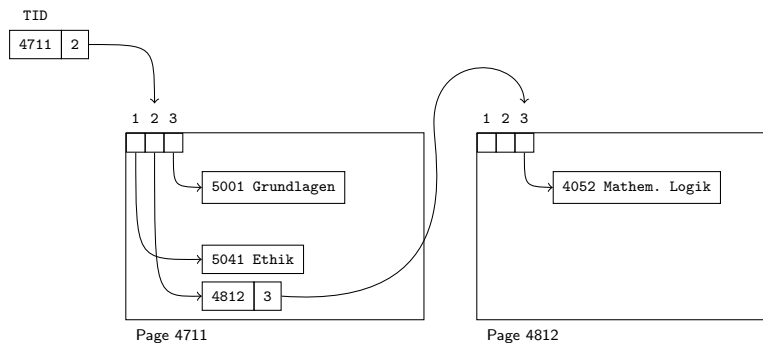
Slotted Pages/3

- Verschieben innerhalb des Blocks:
 - Pointer im Kopfteil wird geändert
 - TID ändert sich nicht



Slotted Pages/4

- Verschieben zwischen Blöcken:
 - Datensatz wird ersetzt durch TID des Datensatzes im neuen Block, welcher nur intern genutzt wird
 - Zugriff auf Datensatz erfordert das Lesen von zwei Blöcken
 - TID ändert sich nicht



Organisation von Datensätzen in Dateien/1

Verschiedene Ansätze, um Datensätze in Dateien logisch anzuordnen (primary file organisation):

- **Heap Datei:** ein Datensatz kann irgendwo gespeichert werden, wo Platz frei ist, oder er wird am Ende angehängt
- **Sequentielle Datei:** Datensätze werden nach einem bestimmten Datenfeld sortiert abgespeichert
- **Hash Datei:** der Hash-Wert für ein Datenfeld wird berechnet; der Hash-Wert bestimmt, in welchem Block der Datei der Datensatz gespeichert wird

Normalerweise wird jede Tabelle in eigener Datei gespeichert.

Organisation von Datensätzen in Dateien/2

- **Sequentielle Datei:** Datensätze nach Suchschlüssel (ein oder mehrere Datenfelder) geordnet
 - Datensätze sind mit Pointern verkettet
 - gut für Anwendungen, die sequentiellen Zugriff auf gesamte Datei brauchen
 - Datensätze sollten – soweit möglich – nicht nur logisch, sondern auch physisch sortiert abgelegt werden
- **Beispiel:** Konto(KontoNr, **FilialName**, Kontostand)

record 0	A-217	Brighton	750	→
record 1	A-101	Downtown	500	→
record 2	A-110	Downtown	600	→
record 3	A-215	Mianus	700	→
record 4	A-102	Perryridge	400	→
record 5	A-201	Perryridge	900	→
record 6	A-218	Perryridge	700	→
record 7	A-222	Redwood	700	→
record 8	A-305	Round Hill	350	→

Organisation von Datensätzen in Dateien/3

- **Physische Ordnung erhalten** ist schwierig.
- **Löschen:**
 - Datensätze sind mit Pointern verkettet (verkettete Liste)
 - gelöschter Datensatz wird aus der verketteten Liste genommen
- leere Zwischenräume reduzieren Datendichte
- **Einfügen:**

- finde Block, in den neuer Datensatz eingefügt werden müsste
- falls freier Speicher im Block: einfügen
- falls zu wenig freier Speicher: Datensatz in Überlauf-Block (overflow block) speichern

record 0	A-217	Brighton	750	→
record 1	A-101	Downtown	500	→
record 2	A-110	Downtown	600	→
record 3	A-215	Mianus	700	→
record 4	A-102	Perryridge	400	→
record 5	A-201	Perryridge	900	→
record 6	A-218	Perryridge	700	→
record 7	A-222	Redwood	700	→
record 8	A-305	Round Hill	350	→

A-888	North Town	800	→
-------	------------	-----	---

→ Tabelle sortiert lesen erfordert nicht-sequentiellen Blockzugriff

- Datei muss **von Zeit zu Zeit reorganisiert** werden, um physische Ordnung wieder herzustellen

Datenbankverzeichnis/1

- **Datenbankverzeichnis (Katalog):** speichert Metadaten
 - Informationen über Relationen
 - Name der Relation
 - Name und Typen der Attribute jeder Relation
 - Name und Definition von Views
 - Integritätsbedingungen (z.B. Schlüssel und Fremdschlüssel)
 - Benutzerverwaltung
 - Statistische Beschreibung der Instanz
 - Anzahl der Tupel in der Relation
 - häufigste Werte
 - Physische Dateiorganisation
 - wie ist eine Relation gespeichert (sequentiell/Hash/...)
 - physischer Speicherort (z.B. Festplatte)
 - Dateiname oder Adresse des ersten Blocks auf der Festplatte
 - Information über Indexstrukturen

Datenbankverzeichnis/2

- **Physische Speicherung** des Datenbankverzeichnisses:
 - spezielle Datenstrukturen für effizienten Zugriff optimiert
 - Relationen welche bestehende Strategien für effizienten Zugriff nutzen
- **Beispiel-Relationen** in einem Verzeichnis (vereinfacht):
 - **RELATION-METADATA**(relation-name, number-of-attributes, storage-organization, location)
 - **ATTRIBUTE-METADATA**(attribute-name, relation-name, domain-type, position, length)
 - **USER-METADATA**(user-name, encrypted-password, group)
 - **INDEX-METADATA**(index-name, relation-name, index-type, index-attributes)
 - **VIEW-METADATA**(view-name, definition)
- **PostgreSQL (ver 9.3):** mehr als 70 Relationen:
<http://www.postgresql.org/docs/9.3/static/catalogs-overview.html>