Datenbanken 2

Einführung, Physische Datenorganisation

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg

database
research group

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 9. Oktober 2018

Inhalt

Inhalt

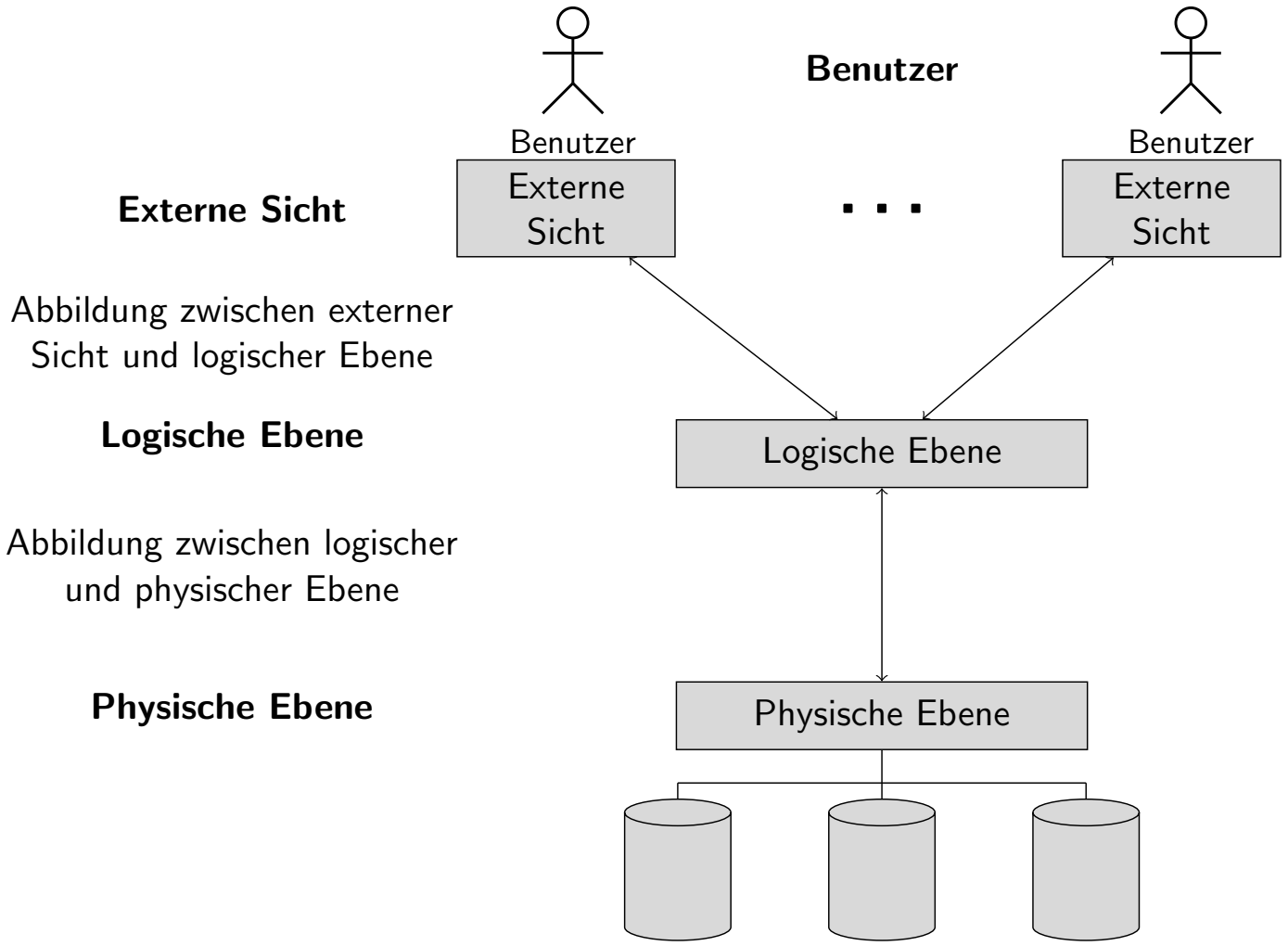
Alle Infos zu Vorlesung und Proseminar:

http://dbresearch.uni-salzburg.at/teaching/2018ws/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 **Effizienz** (schneller ist besser)

Die ANSI/SPARC Drei-Ebenen Architektur



Inhaltsübersicht Datenbanksysteme

1. **Physische Datenorganisation**
   - Speichermedien, Dateiorganisation
   - Kapitel 7 in Kemper und Eickler
   - Chapter 10 in Silberschatz et al.
2. **Indexstrukturen**
   - Sequentielle Dateien, B^+ Baum, Statisches Hashing, Dynamisches Hashing, Mehrere Suchschlüssel, Indizes 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
   - Kapitel 8 in Kemper und Eickler
   - Chapter 13 in Silberschatz et al.

Inhalt

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 ns $= 10^{-9}$s)
  - meist zu klein (oder zu teuer) um gesamte Datenbank zu speichern
    - mehrere GB weit verbreitet
    - Preise derzeit ca. 7.5 EUR/GB (DRAM)

Speichermedien/3

- Flash memory (SSD)
  - persistent
  - lesen ist sehr schnell (x0 bis x00 μs; 1 μs $= 10^{-6}$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. 0.2 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 ms $= 10^{-3}$s)
  - sequentielles Lesen: 25–100 MB/s
  - billig: Preise teils unter 30 EUR/TB
  - sehr viel mehr Platz als im Hauptspeicher; derzeit x00 GB - 14 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 bis 17 GB), Blu-ray (25 bis 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

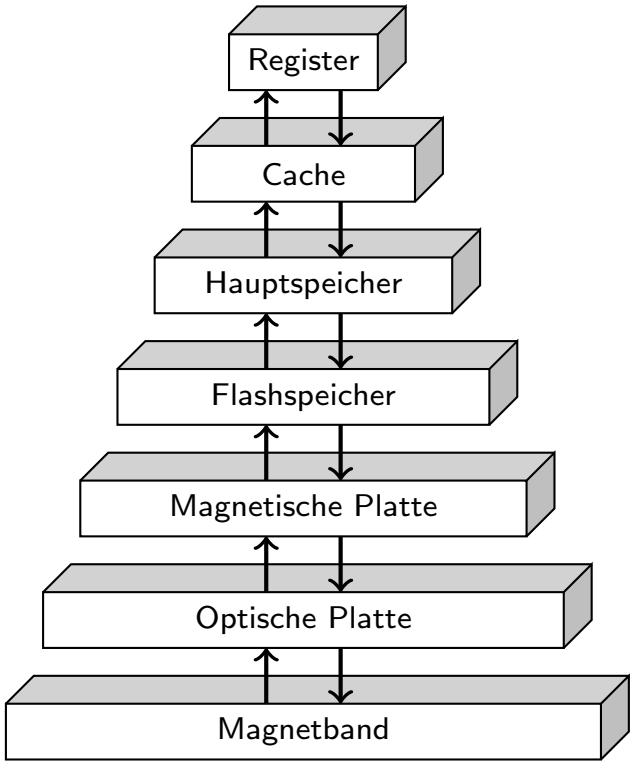
Speichermedien/6

- 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:

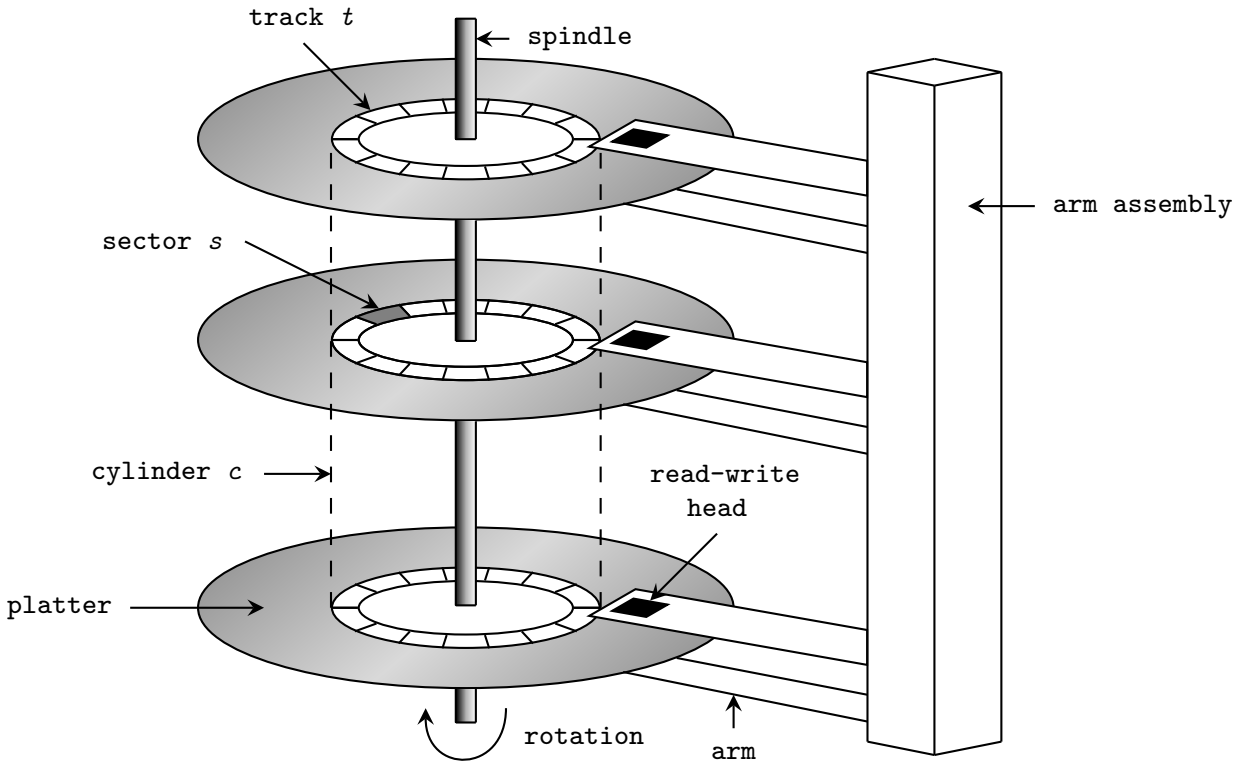
- Primärspeicher: flüchtig, schnell, teuer
  - z. B. Cache, Hauptspeicher
- Sekundärspeicher: persistent, langsamer und günstiger als Primärspeicher
  - z. B. Magnetplatten, Flash Speicher
  - auch Online-Speicher genannt
- Tertiärspeicher: persistent, sehr langsam, sehr günstig
  - z. B. Magnetbänder,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



- Datenbank muss mit Speichermedien auf allen Ebenen umgehen

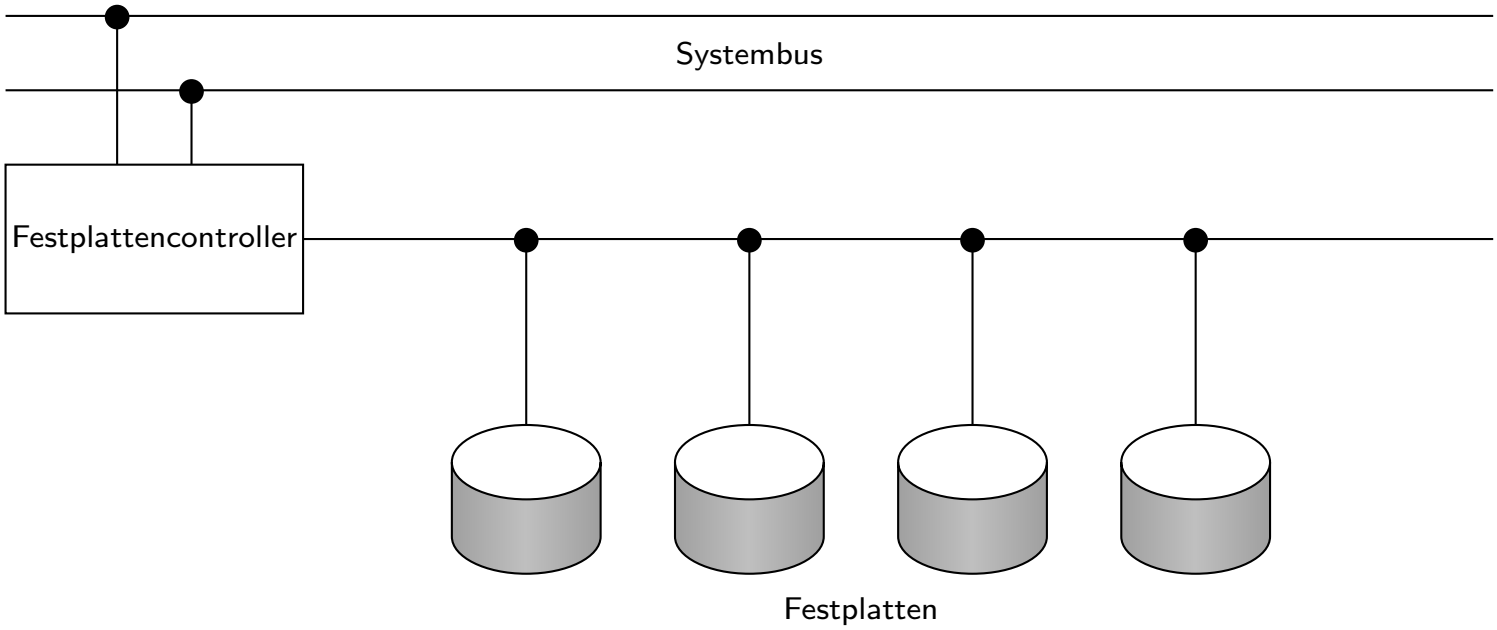
Festplatten/1

- 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 / 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

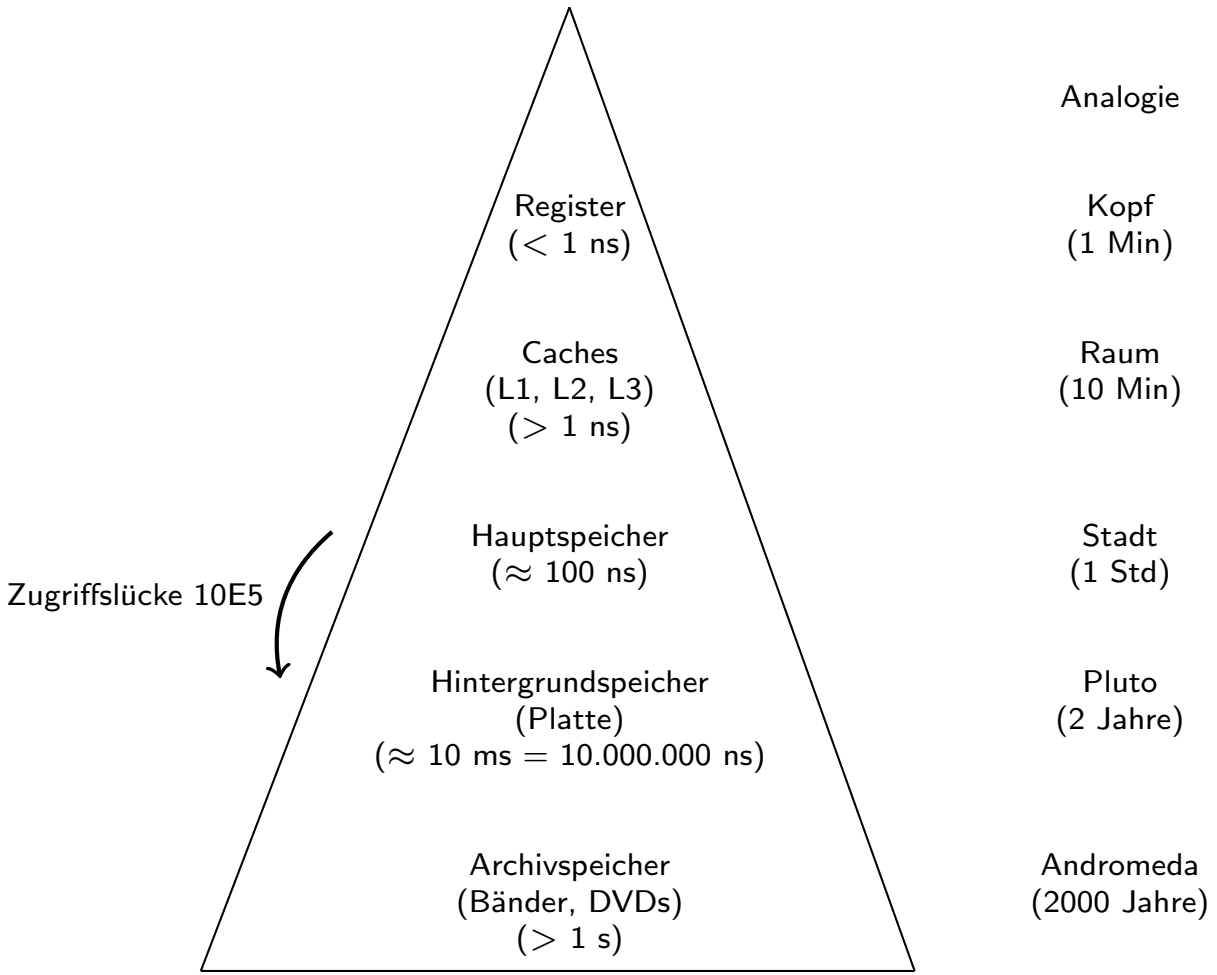
Betrachte folgende Festplatte: Sektor-Größe $B = 512$ Bytes, Sektoren/Spur $S = 20$, Spuren pro Scheibenseite $T = 400$, Anzahl der beidseitig beschriebenen Scheiben $D = 15$, mittlerer Spurwechsel $sp = 30ms$, Drehzahl $dz = 2400rpm$ (Interblock Gaps werden vernachlässigt).

Bestimme die folgenden Werte:

a) Kapazität der Festplatte
b) mittlere Zugriffszeit (1 Sektor lesen)

Inhalt

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:
  1. Disk Arm Scheduling
  2. Geeignete Dateistrukturen
  3. Schreib-Puffer und Log Disk

Block Speicher Zugriff/1

- 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/2

- 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 Zugriffsarms
  - manche Systeme erlauben das Defragmentieren des Dateisystems

Block Speicher Zugriff/3

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 benutzten um zukünfiges 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

- Pinned block: Darf nicht aus dem Puffer entfernt werden.
  - z.B. der R-Block, bevor alle Tupel von S bearbeitet sind
- Toss Immediate Strategy: Block wird sofort rausgeworfen, wenn das letzte Tupel bearbeitet wurde.
  - z.B. der R Block sobald das letzte Tupel von S bearbeitet wurde
- Gemischte Strategie mit Tipps vom Anfrageoptimierer ist am erfolgreichsten.

Ersetzstrategien für Pufferseiten/3

- Beispiel: Berechne Join mit Nested Loops

```
für jedes Tupel tr von R:
  für jedes Tupel ts von S:
    wenn ts und tr das Join-Prädikat erfüllen, dann ...
```

- Verschiedene Zugriffsmuster für R und S
  - ein R-Block wird nicht mehr benötigt, sobald das letzte Tupel 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 einen 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 + Pinned Block (für aktuellen Block von R)
- MRU + Pinned Block (für aktuellen Block von R) + Toss Immediate (für abgearbeiteten Block von R)

Welche Strategie eignet sich besser?

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
  - Heuristik: Verzeichnis im Hauptspeicher halten

Inhalt

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
  - variabel: komplizierter, flexibel, 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:

  (a) verschiebe Datensätze $i + 1, ..., n$ nach $i, ..., n - 1$

  (b) verschiebe letzten Datensatz im Block nach i

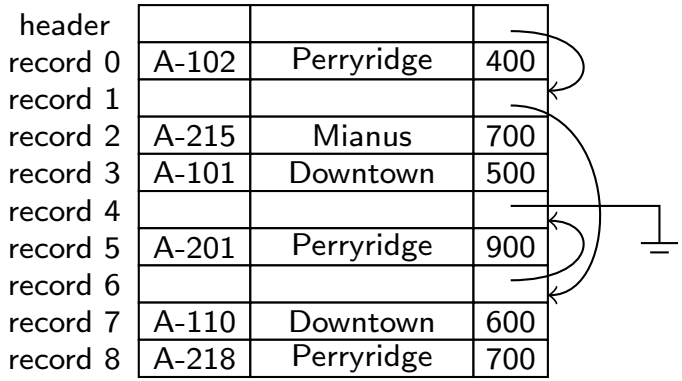
  (c) nicht verschieben, sondern "Free List" verwalten

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

  → der Speicherbereich des gelöschten Datensatzes wird für Free List Pointer verwendet
- Beispiel: Free List nach löschen der Datensätze 4, 6, 1 (in dieser Reihenfolge)

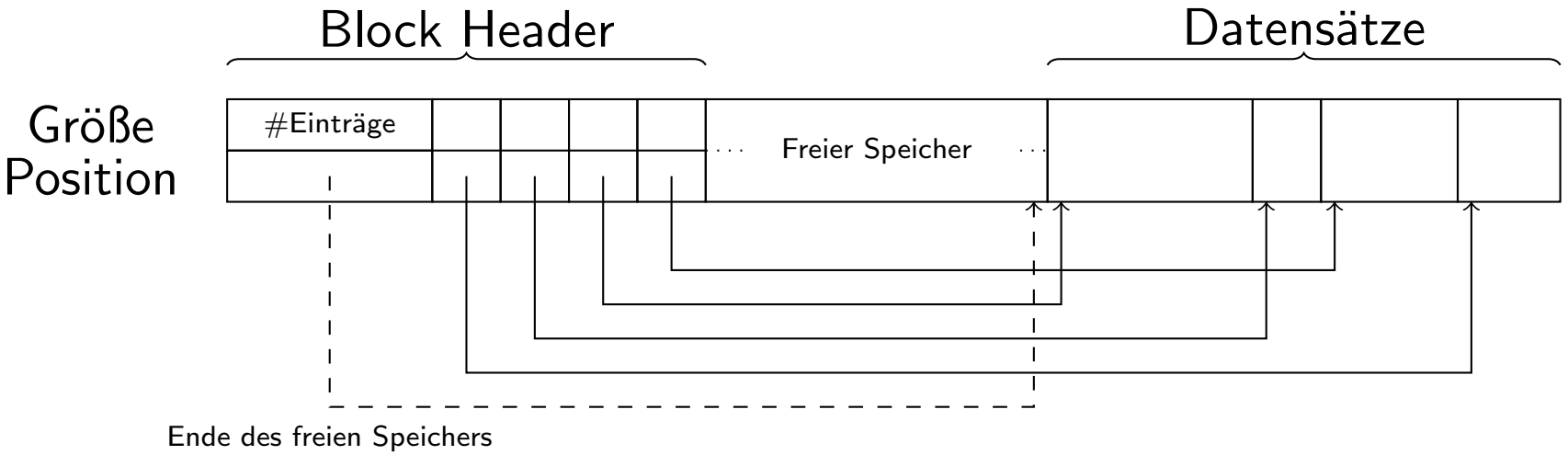


Variable Datensatzlänge/1

- 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)

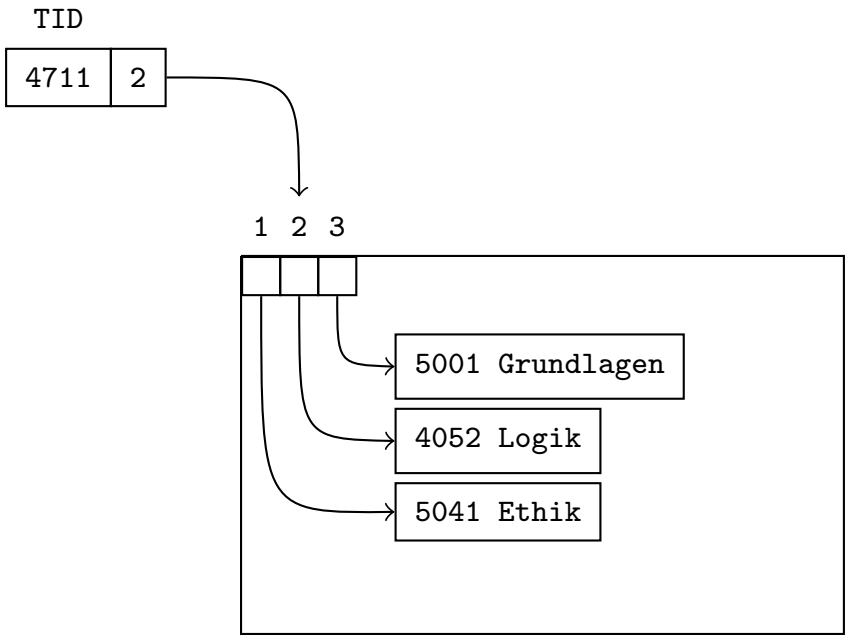
Slotted Pages/1

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



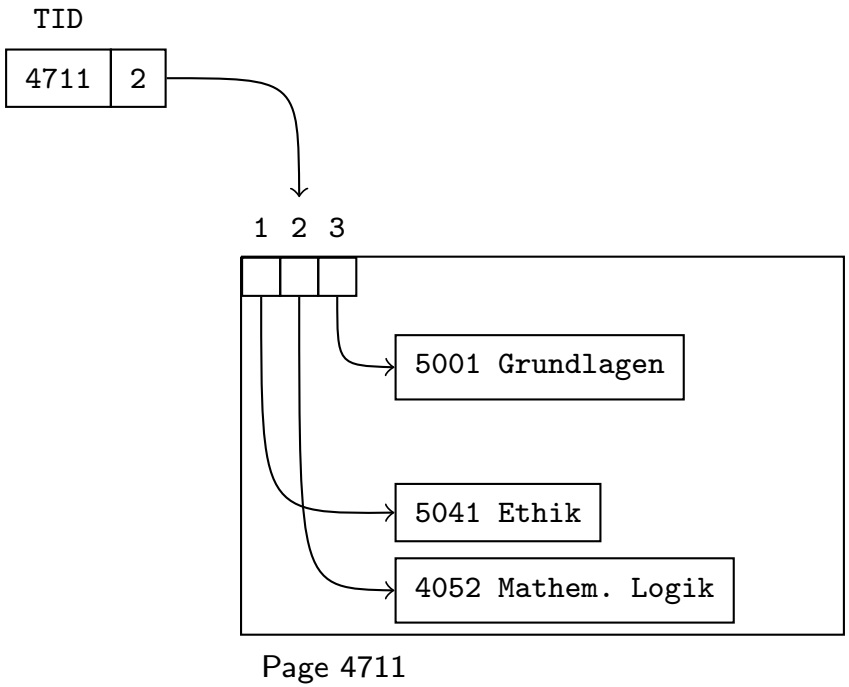
Slotted Pages/2

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



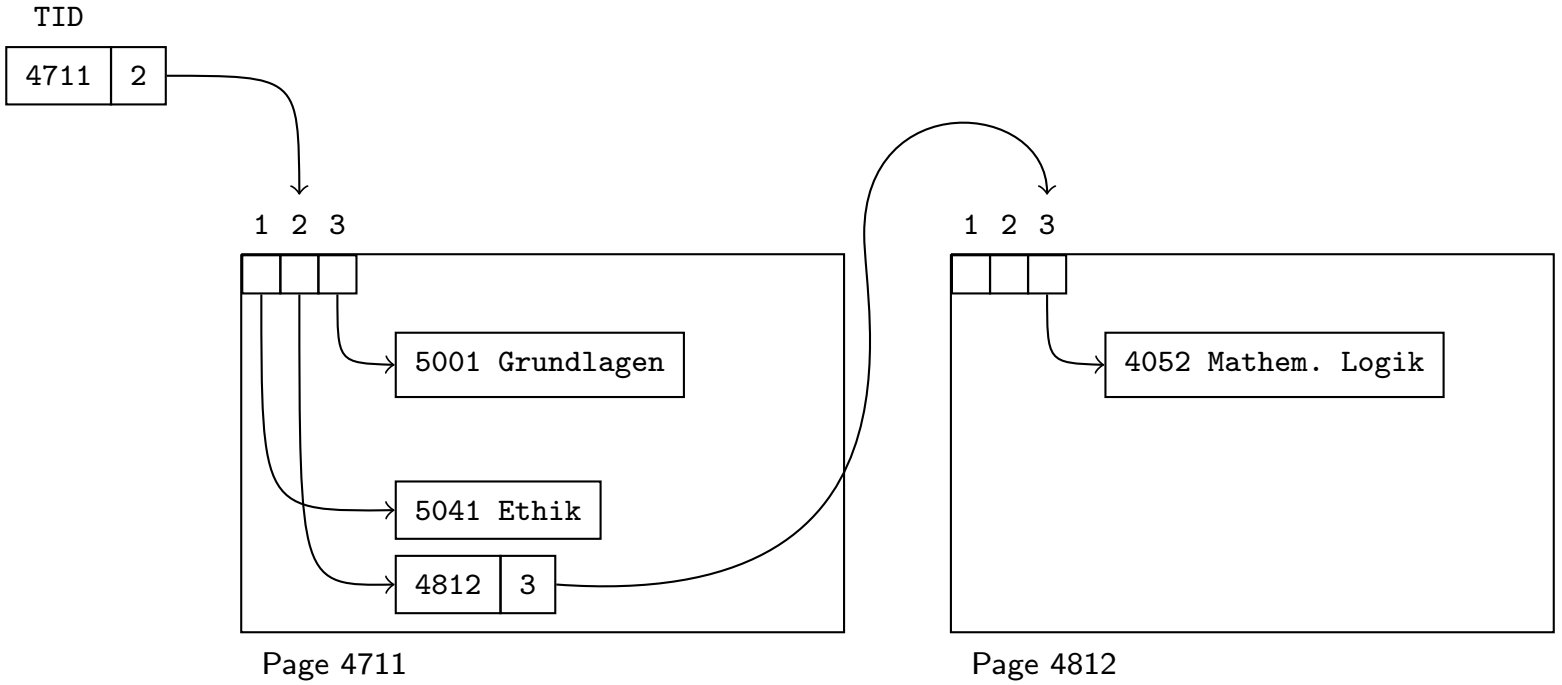
Slotted Pages/3

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



Slotted Pages/4

- Verschieben zwischen Blöcken:
  - Datensatz wird ersetzt durch Referenz auf neuen Block, welche nur intern genutzt wird
  - Zugriff auf Datensatz erfordert das Lesen von zwei Blöcken
  - TID des Datensatzes ändert sich nicht
  - weitere Verschiebungen modifizieren stets die Referenz im ursprünglichen Block (d.h. es entsteht keine verkettete Liste)



Organisation von Datensätzen in Dateien/1

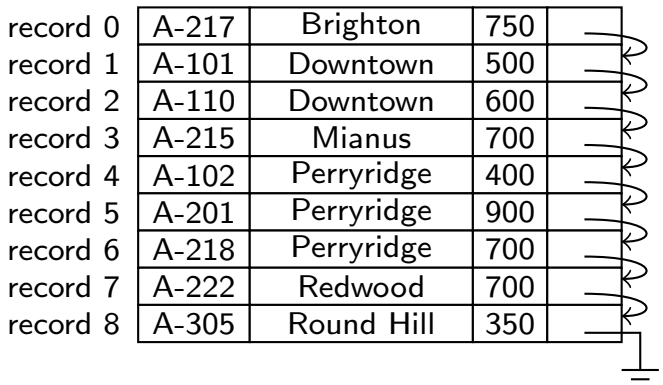
Verschiedene Ansätze, um Datensätze in Dateien logisch anzuordnen (primary file organisation):

- 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)



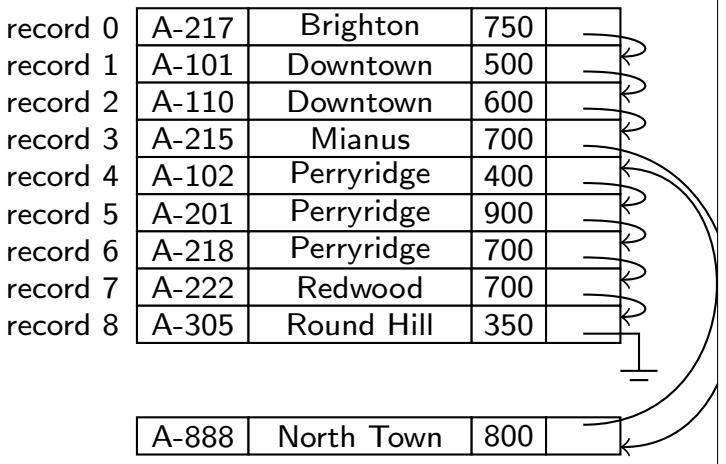
Organisation von Datensätzen in Dateien/3

- 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

  → 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`

Datenbanken 2

Indexstrukturen

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at

FB Computerwissenschaften

Universität Salzburg

database

research group

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 20. November 2018

Inhalt

Literatur und Quellen

Lektüre zum Thema "Indexstrukturen":

- Kapitel 7 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 11 in Silberschatz, Korth, and Sudarashan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Inhalt

Grundlagen/1

- Index beschleunigt Zugriff, z.B.:
  - Autorenkatalog in Bibliothek
  - Index in einem Buch
- Index-Datei besteht aus Datensätzen: den Index-Einträgen
- Index-Eintrag hat die Form

  (Suchschlüssel, Pointer)
  - *Suchschlüssel*: Attribut(liste) nach der Daten gesucht werden
  - *Pointer*: Pointer auf einen Datensatz (TID)
- Suchschlüssel darf mehrfach vorkommen

  (im Gegensatz zu Schlüsseln von Relationen)
- Index-Datei meist viel kleiner als die indizierte Daten-Datei

Grundlagen/2

- Merkmale des Index sind:
  - Zugriffszeit
  - Zeit für Einfügen
  - Zeit für Löschen
  - Speicherbedarf
  - effizient unterstützte Zugriffsarten
- Wichtigste Zugriffsarten sind:
  - Punktanfragen: z.B. Person mit SVN=1983-3920
  - Mehrpunktanfragen: z.B. Personen, die 1980 geboren wurden
  - Bereichsanfragen: z.B. Personen die mehr als 100.000 EUR verdienen

Grundlagen/3

Indextypen werden nach folgenden Kriterien unterschieden:

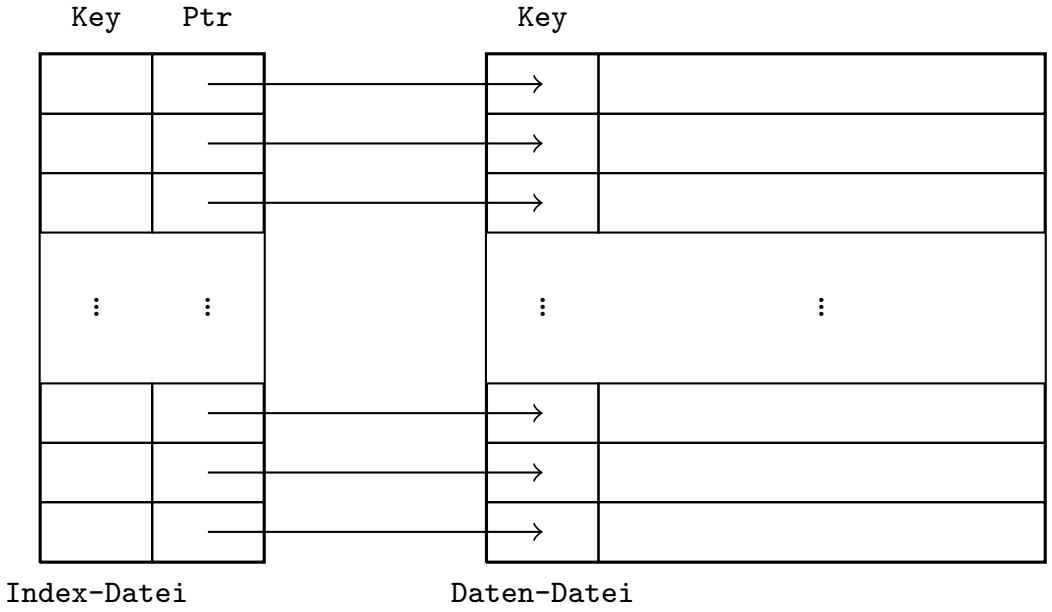
- Ordnung der Daten- und Index-Datei:
  - Primärindex
  - Clustered Index
  - Sekundärindex
- Art der Index-Einträgen:
  - sparse Index
  - dense Index

Nicht alle Kombinationen üblich/möglich:

- Primärindex ist oft sparse
- Sekundärindex ist immer dense

Primärindex/1

- Primärindex:
  - Datensätze in der Daten-Datei sind nach Suchschlüssel sortiert
  - Suchschlüssel ist eindeutig, d.h., Suche nach 1 Schlüssel ergibt (höchstens) 1 Tupel

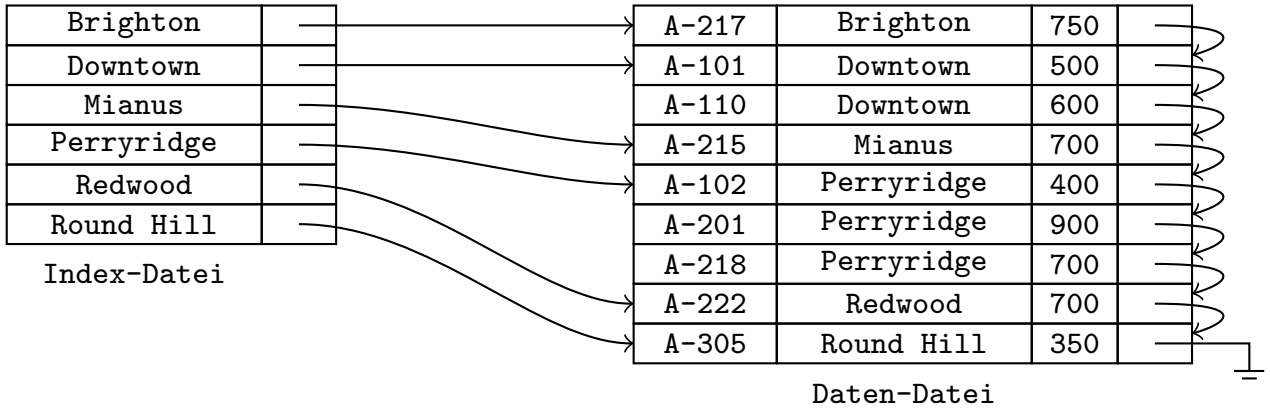


Primärindex/2

- Index-Datei:
  - sequentiell geordnet nach Suchschlüssel
- Daten-Datei:
  - sequentiell geordnet nach Suchschlüssel
  - jeder Suchschlüssel kommt nur 1 mal vor
- Effiziente Zugriffsarten:
  - Punkt- und Bereichsanfragen
  - nicht-sequentieller Zugriff (random access)
  - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

Clustered Index

- Index-Datei:
  - sequentiell geordnet nach Suchschlüssel
- Daten-Datei:
  - sequentiell geordnet nach Suchschlüssel
  - Suchschlüssel kann *mehrfach* vorkommen
- Effiziente Zugriffsarten:
  - Punkt-, Mehrpunkt-, und Bereichsanfragen
  - nicht-sequentieller Zugriff (random access)
  - sequentieller Zugriff nach Suchschlüssel sortiert (sequential access)

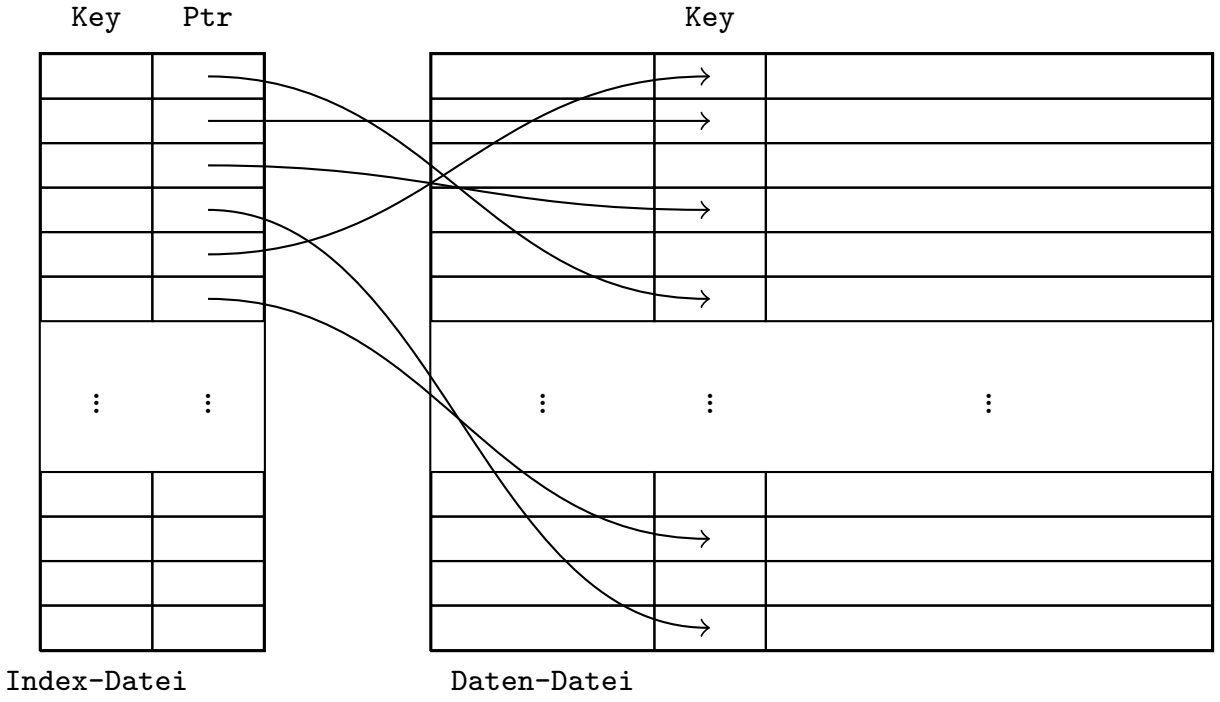


Sekundärindex/1

- Primär- vs. Sekundärindex:
  - nur 1 Primärindex (bzw. Clustered Index) möglich
  - beliebig viele Sekundärindizes
  - Sekundärindex für schnellen Zugriff auf alle Felder, die nicht Suchschlüssel des Primärindex sind
- Beispiel: Konten mit Primärindex auf Kontonummer
  - Finde alle Konten einer bestimmten Filiale.
  - Finde alle Konten mit 1000 bis 1500 EUR Guthaben.
- Ohne Index können diese Anfragen nur durch sequentielles Lesen aller Knoten beantwortet werden – sehr langsam
- Sekundärindex für schnellen Zugriff erforderlich

Sekundärindex/2

- Index-Datei:
  - sequentiell nach Suchschlüssel geordnet
- Daten-Datei:
  - Suchschlüssel kann *mehrfach* vorkommen
  - *nicht* nach Suchschlüssel geordnet



Sekundärindex/3

- Effiziente Zugriffsarten:
  - sehr schnell für Punktanfragen
  - Mehrpunkt- und Bereichsanfragen: gut wenn nur kleiner Teil der Tabelle zurückgeliefert wird (wenige %)
  - besonders für nicht-sequentiellen Zugriff (random access) geeignet

Duplikate/1

Umgang mit mehrfachen Suchschlüsseln:

(a) Doppelte Indexeinträge:

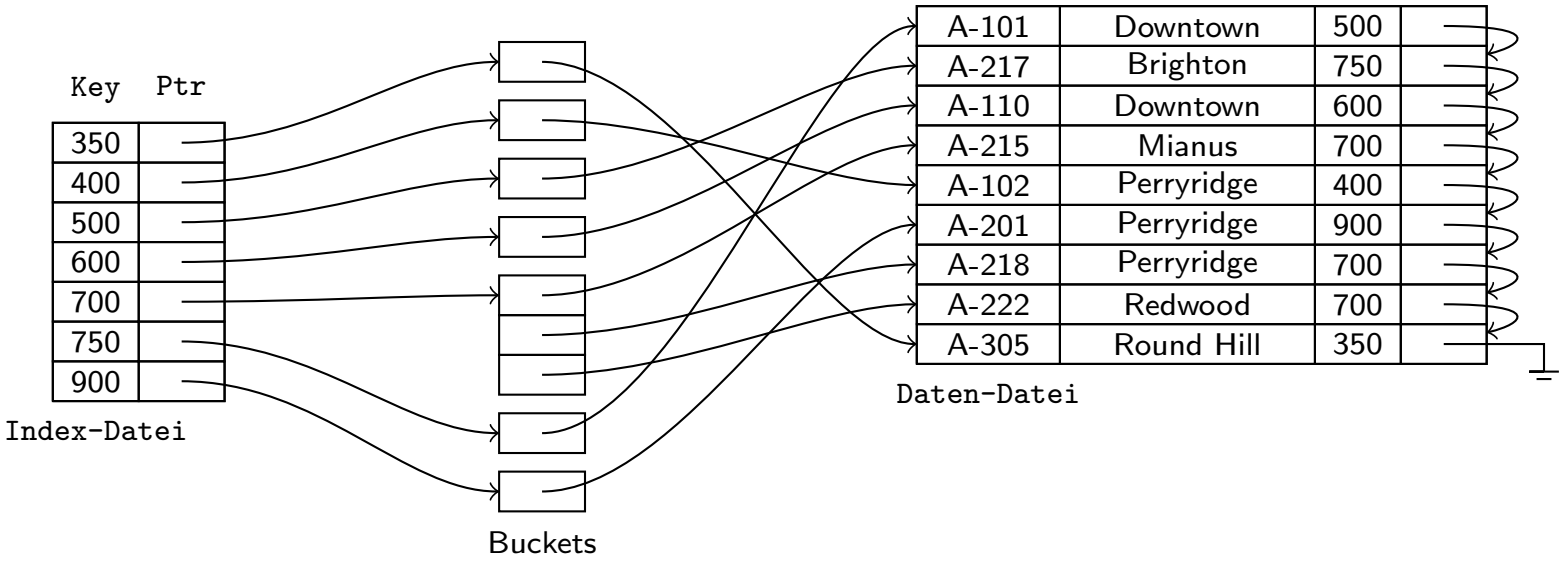
- ein Indexeintrag für jeden Datensatz

→ schwierig zu handhaben, z.B. in B^+-Baum Index

(b) Buckets:

- nur einen Indexeintrag pro Suchschlüssel
- Index-Eintrag zeigt auf ein Bucket
- Bucket zeigt auf alle Datensätze zum entsprechenden Suchschlüssel

→ zusätzlicher Block (Bucket) muss gelesen werden



Duplikate/2

Umgang mit mehrfachen Suchschlüsseln:

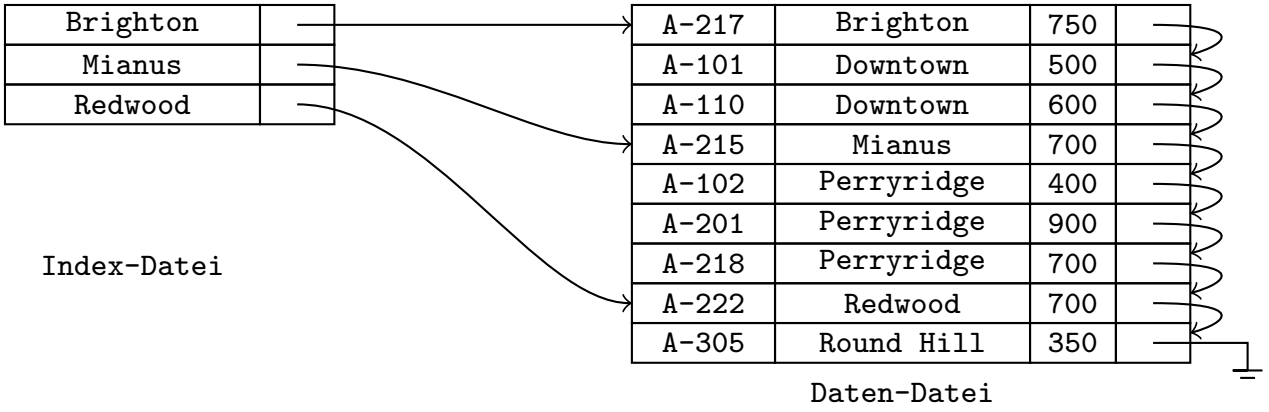
(c) Suchschlüssel eindeutig machen:

- Einfügen: TID wird an Suchschlüssel angehängt (sodass dieser eindeutig wird)
- Löschen: Suchschlüssel und TID werden benötigt (ergibt genau 1 Index-Eintrag)
- Suche: nur Suchschlüssel wird benötigt (ergibt mehrere Index-Einträge)

$\rightarrow$ wird in der Praxis verwendet

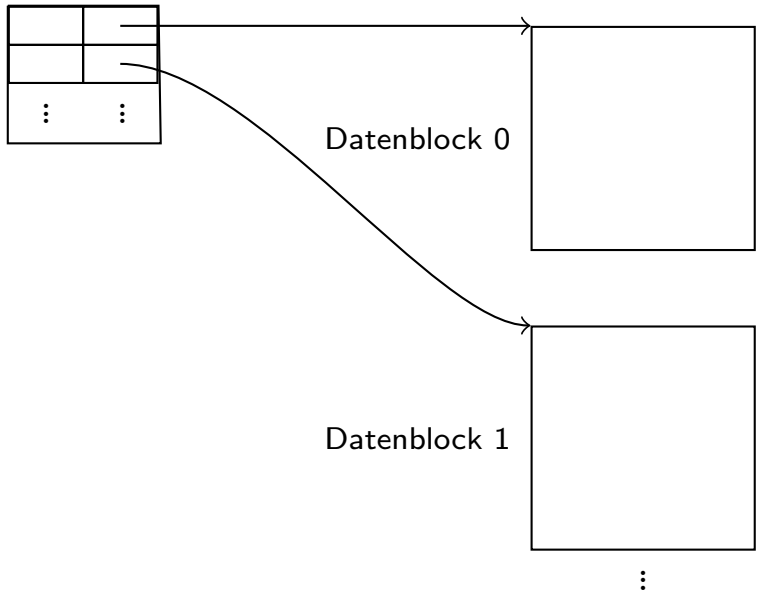
Sparse Index/1

- Sparse Index
  - ein Index-Eintrag für mehrere Datensätze
  - kleiner Index: weniger Index-Einträge als Datensätze
  - nur möglich wenn Datensätze nach Suchschlüssel geordnet sind (d.h. Primärindex oder Clustered Index)



Sparse Index/2

- Oft enthält ein sparse Index einen Eintrag pro Block.
- Der Suchschlüssel, der im Index für eine Block gespeichert wird, ist der kleinste Schlüssel in diesem Block.



Dense Index/1

- Dense Index:
  - Index-Eintrag (bzw. Pointer in Bucket) für jeden Datensatz in der Daten-Datei
  - dense Index kann groß werden (aber normalerweise kleiner als Daten)
  - Handhabung einfacher, da ein Pointer pro Datensatz
- Sekundärindex ist immer dense

Gegenüberstellung von Index-Typen

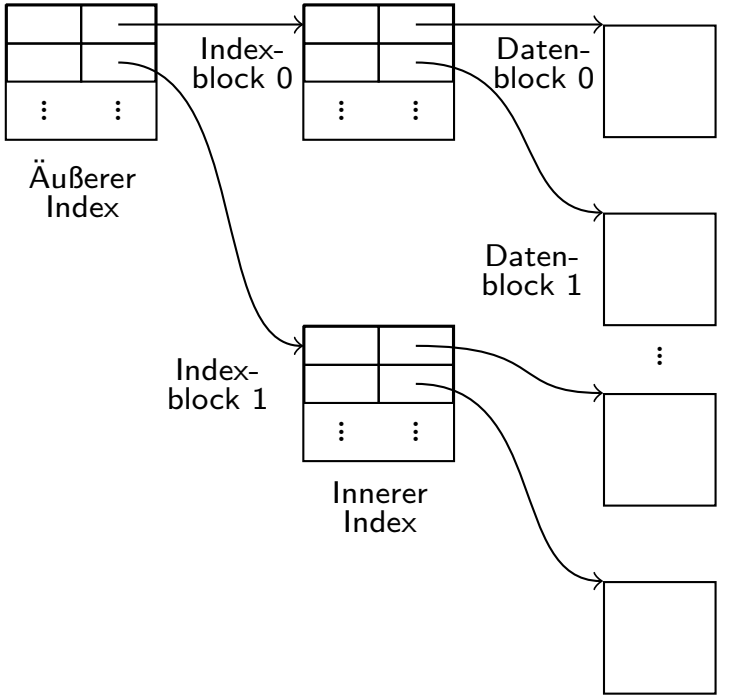
- Alle Index-Typen machen Punkt-Anfragen erheblich schneller.
- Index erzeugt Kosten bei Updates: Index muss auch aktualisiert werden.
- Dense/Sparse und Primär/Sekundär:
  - Primärindex kann dense oder sparse sein
  - Sekundärindex ist immer dense
- Sortiert lesen (=sequentielles Lesen nach Suchschlüssel-Ordnung):
  - mit Primärindex schnell
  - mit Sekundärindex teuer, da sich aufeinander folgende Datensätze auf unterschiedlichen Blöcken befinden (können)
- Dense vs. Sparse:
  - sparse Index braucht weniger Platz
  - sparse Index hat geringere Kosten beim Aktualisieren
  - dense Index erlaubt bestimmte Anfragen zu beantworten, ohne dass Datensätze gelesen werden müssen ("covering index")

Mehrstufiger Index/1

- Großer Index wird teuer:
  - Index passt nicht mehr in Hauptspeicher und mehrere Block-Lese-Operationen werden erforderlich
  - binäre Suche: $\lfloor log_2(B) \rfloor + 1$ Block-Lese-Operationen (Index mit B Blöcken)
  - eventuelle Overflow Blöcke müssen sequentiell gelesen werden
- Lösung: Mehrstufiger Index
  - Index wird selbst wieder indiziert
  - dabei wird der Index als sequentielle Daten-Datei behandelt

Mehrstufiger Index/2

- Mehrstufiger Index:
  - Innerer Index: Index auf Daten-Datei
  - Äußerer Index: Index auf Index-Datei
- Falls äußerer Index zu groß wird, kann eine weitere Index-Ebene eingefügt werden.



- Diese Art von (ein- oder mehrstufigem) Index wird auch als ISAM (Index Sequential Access Method) oder index-sequentielle Datei bezeichnet.

Mehrstufiger Index/3

- Index Suche
  - beginne beim Root-Knoten
  - finde alle passenden Einträge und verfolge die entsprechenden Pointer
  - wiederhole bis Pointer auf Datensatz zeigt (Blatt-Ebene)
- Index Update: Löschen und Einfügen
  - Indizes aller Ebenen müssen nachgeführt werden
  - Update startet beim innersten Index
  - Erweiterungen der Algorithmen für einstufige Indizes

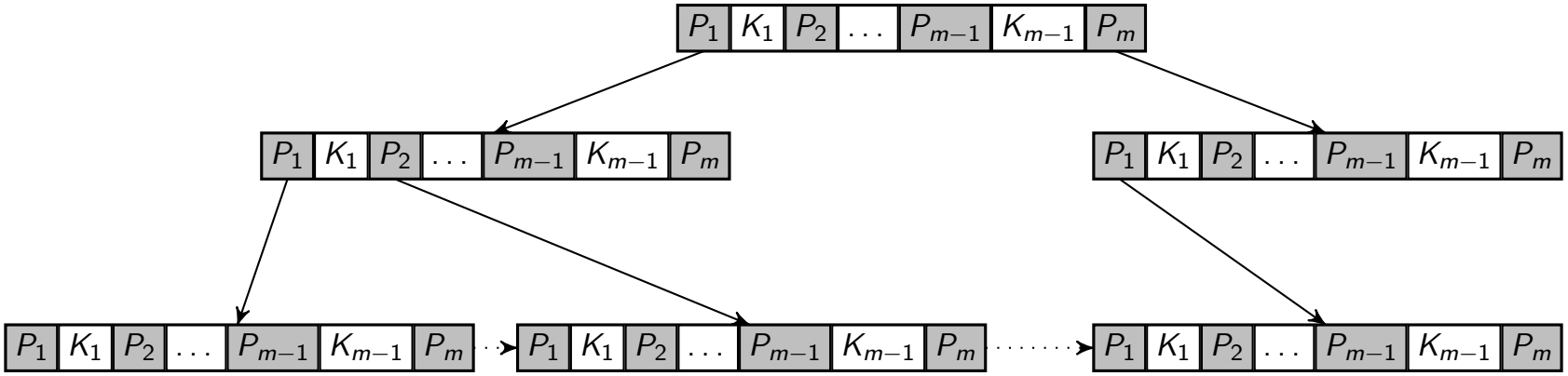
Inhalt

B^+-Baum/1

B^+-**Baum**: Alternative zu index-sequentiellen Dateien:

- Vorteile von B^+-Bäumen:
  - Anzahl der Ebenen wird automatisch angepasst
  - reorganisiert sich selbst nach Einfüge- oder Lösch-Operationen durch kleine lokale Änderungen
  - reorganisieren des gesamten Indexes ist nie erforderlich
- Nachteile von B^+-Bäumen:
  - evtl. Zusatzaufwand bei Einfügen und Löschen
  - etwas höherer Speicherbedarf
  - komplexer zu implementieren
- Vorteile wiegen Nachteile in den meisten Anwendungen bei weitem auf, deshalb sind B^+-Bäume die meist-verbreitete Index-Struktur

B^+-Baum/2



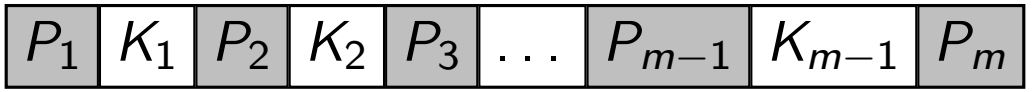
- Knoten mit Grad m: enthält bis zu $m-1$ Suchschlüssel und m Pointer
  - Knotengrad $m > 2$ entspricht der maximalen Anzahl der Pointer
  - Suchschlüssel im Knoten sind sortiert
  - Knoten (außer Wurzel) sind mindestens halb voll
- Wurzelknoten:
  - als Blattknoten: 0 bis $m-1$ Suchschlüssel
  - als Nicht-Blattknoten: mindestens 2 Kinder
- Innerer Knoten: $\lceil m/2 \rceil$ bis m Kinder (=Anzahl Pointer)
- Blattknoten: $\lceil (m-1)/2 \rceil$ bis $m-1$ Suchschlüssel bzw. Daten-Pointer
- balancierter Baum: alle Pfade von der Wurzel zu den Blättern sind gleich lang (maximal $\lceil log_{\lceil m/2 \rceil}(L) \rceil$ Kanten für L Blattknoten)

Terminologie und Notation

- Ein Paar (P_i, K_i) ist ein Eintrag
- $L[i] = (P_i, K_i)$ bezeichnet den i-ten Eintrag von Knoten L
- Daten-Pointer: Pointer zu Datensätzen sind nur in den Blättern gespeichert
- Verbindung zwischen Blättern: der letzte Pointer im Blatt, P_m, zeigt auf das nächste Blatt

Anmerkung: Es gibt viele Varianten des B^+-Baumes, die sich leicht unterscheiden. Auch in Lehrbüchern werden unterschiedliche Varianten vorgestellt. Für diese Lehrveranstaltung gilt der B^+-Baum, wie er hier präsentiert wird.

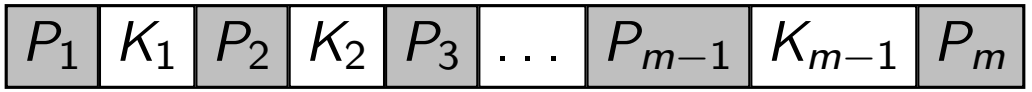
B^+-Baum Knotenstruktur/1



Blatt-Knoten:

- $K_1, \ldots, K_{m-1}$ sind Suchschlüssel
- $P_1, ..., P_{m-1}$ sind Daten-Pointer
- Suchschlüssel sind sortiert: $K_1 < K_2 < K_3 < \ldots < K_{m-1}$
- Daten-Pointer P_i, $1 \leq i \leq m-1$, zeigt auf
  - einen Datensatz mit Suchschlüssel K_i, oder
  - auf ein Bucket mit Pointern zu Datensätzen mit Suchschlüssel K_i
- P_m zeigt auf das nächste Blatt in Suchschlüssel-Ordnung

B^+-Baum Knotenstruktur/2

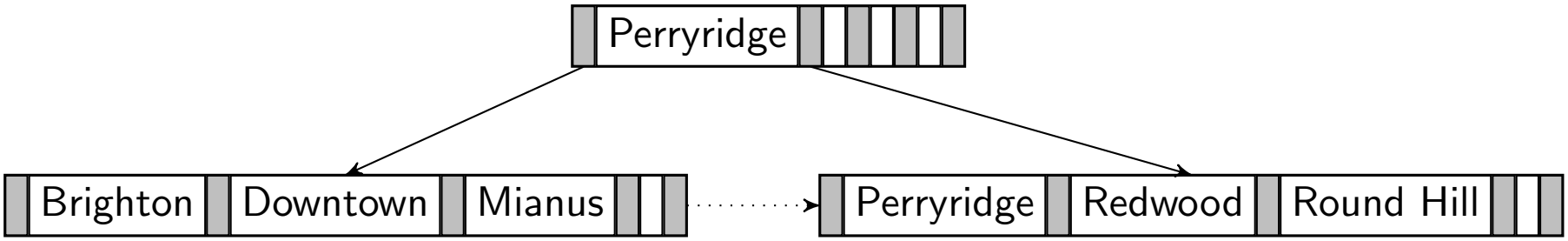


Innere Knoten:

- Stellen einen mehrstufigen sparse Index auf die Blattknoten dar
- Suchschlüssel im Knoten sind eindeutig
- $P_1, ..., P_m$ sind Pointer zu Kind-Knoten, d.h., zu Teilbäumen
- Alle Suchschlüssel k im Teilbaum von P_i haben folgende Eigenschaften:
  - $i = 1$: $k < K_1$
  - $1 < i < m$: $K_{i-1} \leq k < K_i$
  - $i = m$: $k \geq K_{m-1}$

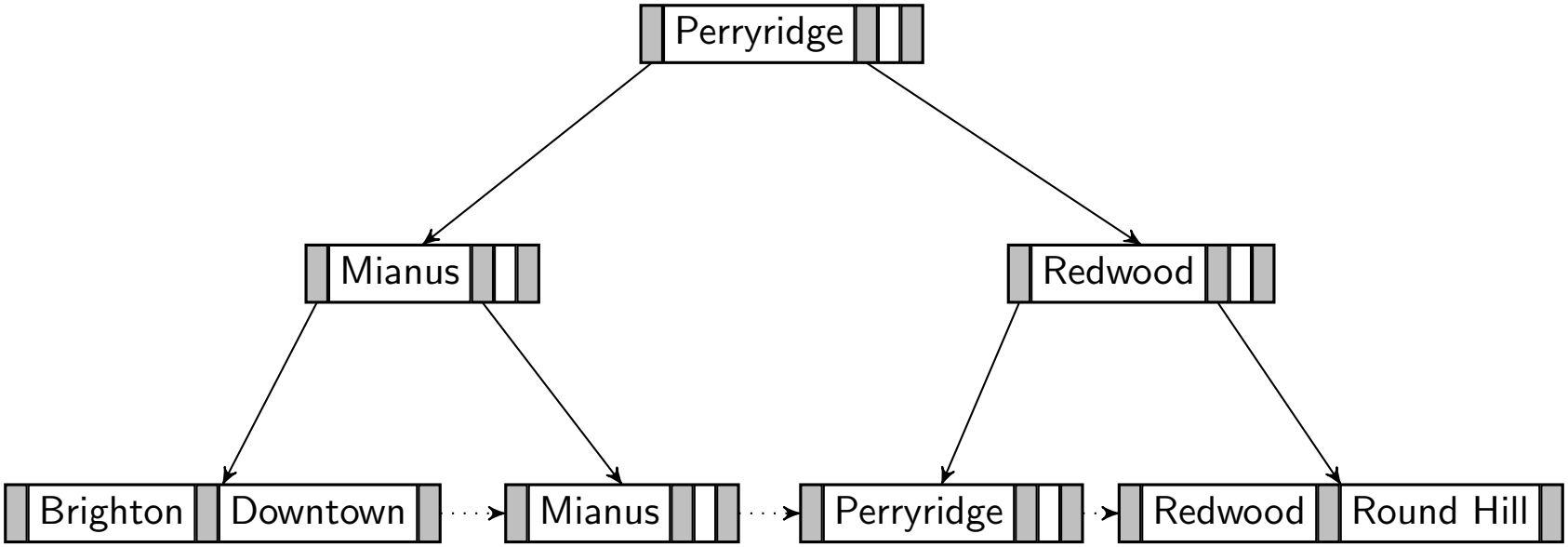
Beispiel: B^+-Baum/1

- Index auf Konto-Relation mit Suchschlüssel Filiale
- B^+-Baum mit Knotengrad $m = 5$:
  - Wurzel: mindestens 2 Pointer zu Kind-Knoten
  - Innere Knoten: $\lceil m/2 \rceil = 3$ bis $m = 5$ Pointer zu Kind-Knoten
  - Blätter: $\lceil (m-1)/2 \rceil = 2$ bis $m - 1 = 4$ Suchschlüssel



Beispiel: B^+-Baum/2

- B^+-Baum für Konto-Relation (Knotengrad $m = 3$)
  - Wurzel: mindestens 2 Pointer zu Kind-Knoten
  - Innere Knoten: $\lceil m/2 \rceil = 2$ bis $m = 3$ Pointer zu Kind-Knoten
  - Blätter: $\lceil (m-1)/2 \rceil = 1$ bis $m - 1 = 2$ Suchschlüssel

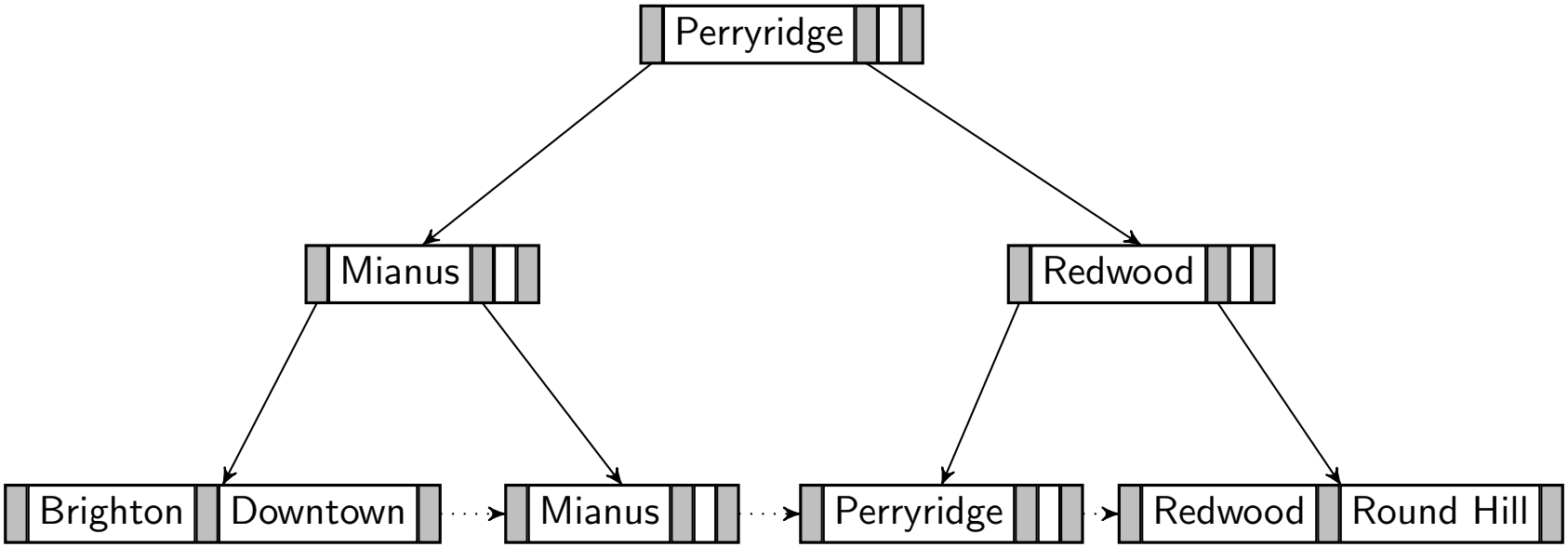


Suche im B^+-Baum/1

- Algorithmus: Suche alle Datensätze mit Suchschlüssel k (Annahme: dense B^+-Baum Index):
  1. $C \leftarrow$ Wurzelknoten
  2. **while** C keine Blattknoten **do**
     suche im Knoten C nach dem größten Schlüssel $K_i \leq k$
     if ein Schlüssel $K_i \leq k$ existiert
     then $C \leftarrow$ Knoten auf den P_{i+1} zeigt
     else $C \leftarrow$ Knoten auf den P_1 zeigt
  3. **if** es gibt einen Schlüssel K_i in C sodass $K_i = k$
     then folge Pointer P_i zum gesuchten Datensatz (oder Bucket)
     else kein Datensatz mit Suchschlüssel k existiert

Suche im B^+-Baum/2

- Beispiel: Finde alle Datensätze mit Suchschlüssel $k =$*Mianus*
  - Beginne mit dem Wurzelknoten
  - Kein Schlüssel $K_i \leq$*Mianus* existiert, also folge P_1
  - $K_1 =$*Mianus* ist der größte Suchschlüssel $K_i \leq$*Mianus*, also folge P_2
  - Suchschlüssel *Mianus* existiert, also folge dem ersten Datensatz-Pointer P_1 um zum Datensatz zu gelangen



Suche im B^+-Baum/3

- Suche durchläuft Pfad von Wurzel bis Blatt:
  - Länge des Pfads höchstens $\lceil log_{\lceil m/2 \rceil}(L) \rceil$ für L Blattknoten
    $\Rightarrow \lceil log_{\lceil m/2 \rceil}(L) \rceil + 1$ Blöcke[1] müssen gelesen werden
  - sind die Blattknoten nur minimal voll ($\lceil (m-1)/2 \rceil$), ergibt sich die maximale Anzahl der Blattknoten: $L = \left\lceil \frac{K}{\lceil (m-1)/2 \rceil} \right\rceil$
  - Wurzelknoten bleibt im Hauptspeicher, oft auch dessen Kinder, dadurch werden 1–2 Block-Zugriffe pro Suche gespart

- Suche effizienter als in sequentiellem Index:
  - bis zu $\lfloor log_2(B) \rfloor + 1$ Blöcke[1] lesen im einstufigen sequentiellen Index (binäre Suche, Index mit B Blöcken, $B = \lceil K/(m-1) \rceil$)

[1] nur Index Blöcke werden gezählt, Datenzugriff hier nicht berücksichtigt

Integrierte Übung 2.1

Es soll ein Index mit 10^6 verschiedenen Suchschlüsseln erstellt werden. Ein Knoten kann maximal 200 Schlüssel mit den entsprechenden Pointern speichern. Es soll nach einem bestimmten Suchschlüssel k gesucht werden.

a) Wie viele Block-Zugriffe erfordert ein B^+-Baum Index maximal, wenn kein Block im Hauptspeicher ist?

b) Wie viele Block-Zugriffe erfordert ein einstufiger, sequentieller Index mit binärer Suche?

Einfügen in B^+-Baum/1

- Datensatz mit Suchschlüssel k einfügen:
  1. füge Datensatz in Daten-Datei ein (ergibt Pointer)
  2. finde Blattknoten für Suchschlüssel k
  3. **falls** im Blatt noch Platz ist **dann**:
     - füge (Pointer, Suchschlüssel)-Paar so in Blatt ein, dass Ordnung der Suchschlüssel erhalten bleibt
  4. **sonst** (Blatt ist voll) teile Blatt-Knoten:
     a) sortiere alle Suchschlüssel (einschließlich k)
     b) die Hälfte der Suchschlüssel bleiben im alten Knoten
     c) die andere Hälfte der Suchschlüssel kommt in einen neuen Knoten
     d) füge den kleinsten Eintrag des neuen Knotens in den Eltern-Knoten des geteilten Knotens ein
     e) **falls** Eltern-Knoten voll ist **dann**:
        teile den Knoten und propagiere Teilung nach oben, sofern nötig

Einfügen in B^+-Baum/2

- Aufteilvorgang:
  - falls nach einer Teilung der neue Schlüssel im Elternknoten nicht Platz hat wird auch dieser geteilt
  - im schlimmsten Fall wird der Wurzelknoten geteilt und der B^+-Baum wird um eine Ebene tiefer

Algorithmus: Einfügen in B^+-Baum/1

$\rightarrow$ Knoten L, Suchschlüssel k, Pointer p (zu Datensatz oder Knoten)

Algorithm 1: B+TreeInsert(L, k, p)

if *L has less than $m-1$ key values* **then**
  insert(k, p) into L

else // Knoten teilen
  $T \leftarrow L \cup (k, p)$; // temporärer Speicher
  create new node L';
  $L'.p_m \leftarrow L.p_m$;
  $L \leftarrow \emptyset$;
  $L.p_m \leftarrow L'$;
  copy $T.p_1$ through $T.k_{\lceil m/2 \rceil}$ into L;
  copy $T.p_{\lceil m/2 \rceil+1}$ through $T.k_m$ into L';
  $k' \leftarrow T.k_{\lceil m/2 \rceil+1}$;
  B+TreeInsertInParent(L, k', L');

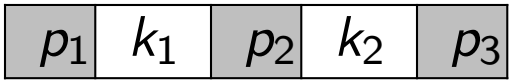
Algorithmus: Einfügen in B^+-Baum/2

Algorithm 2: B+TreeInsertInParent(L, k, L')

if *L is root* **then**
  create new root with children L, L' and value k;
  return;
$P \leftarrow$ parent(L);
if *P has less than m pointers* **then**
  insert(k, L') into P;
else // Knoten teilen
  $T \leftarrow P \cup (k, L')$;
  erase all entries from P;
  create new node P';
  copy $T.p_1$ through $T.p_{\lceil m/2 \rceil}$ into P;
  copy $T.p_{\lceil m/2 \rceil + 1}$ through $T.p_{m+1}$ into P';
  $k' \leftarrow T.k_{\lceil m/2 \rceil}$;
  B+TreeInsertInParent(P, k', P');

Blatt teilen/1

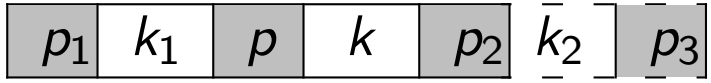
Kopiere L nach T und füge (k, p) ein:



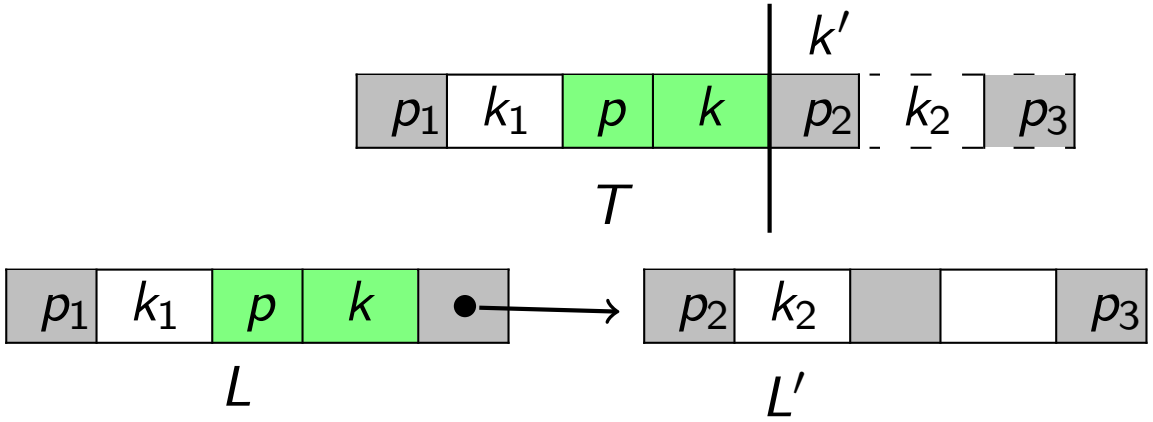
$m = 3$

1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

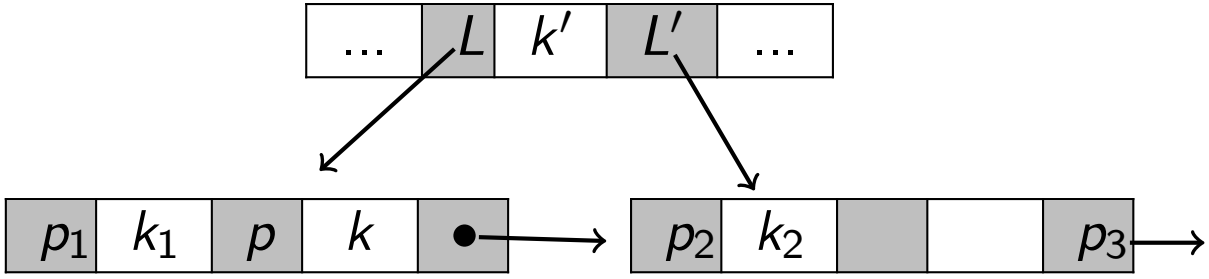
   T



2. Teilen ($k' = T.k_{\lceil m/2 \rceil + 1} = T.k_3$)



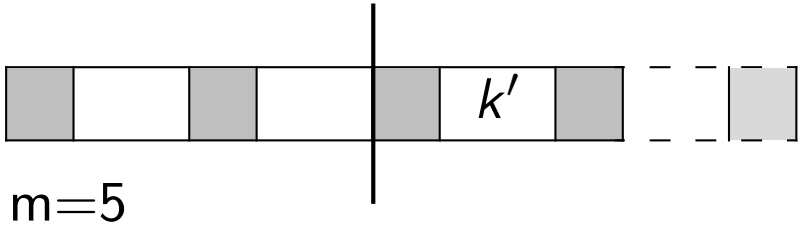
3. (k', L') in Elternknoten von L einfügen



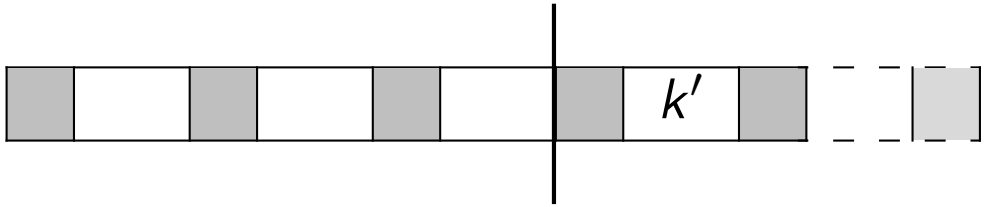
Blatt teilen/2

$k' = T.k_{\lceil m/2 \rceil + 1}$

- m gerade, z.B.: m=4



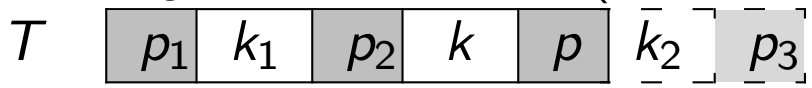
- m ungerade, z.B.: m=5



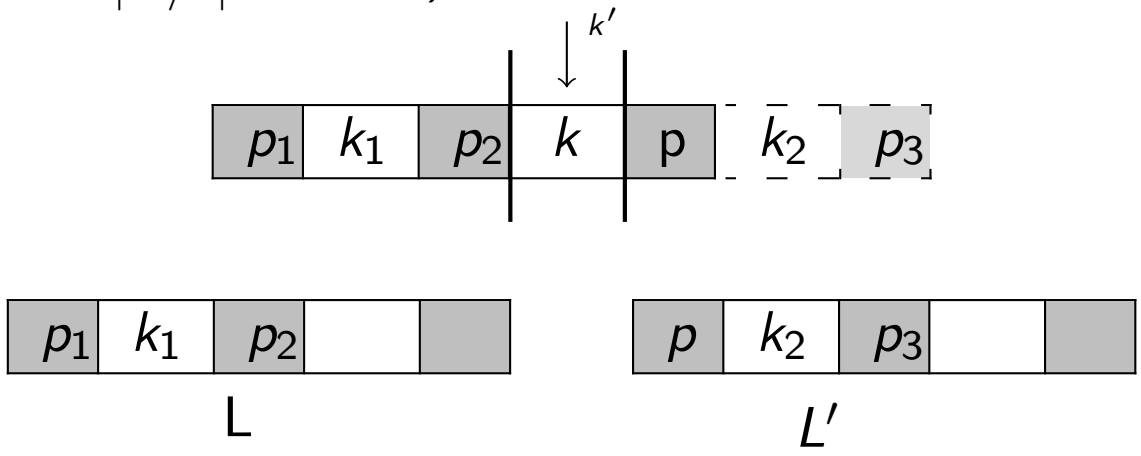
Innere Knoten teilen/1

Kopiere P nach T und füge (k, p) ein:

1. Anhängen und sortieren (z.B.: $k_1 < k < k_2$)

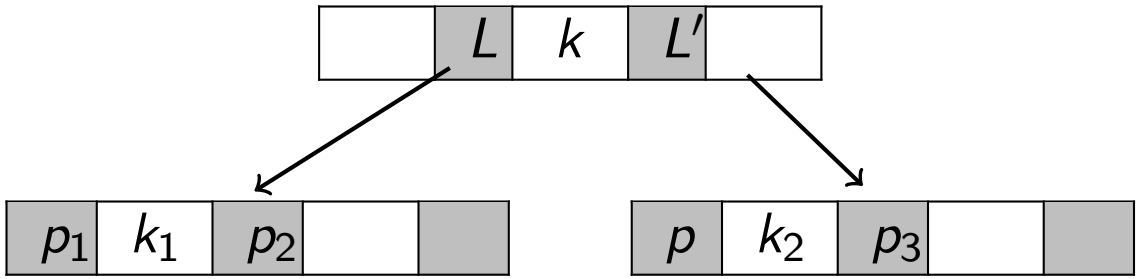


2. Teilen ($k' = T.k_{\lceil m/2 \rceil} = T.k_2$)



Innere Knoten teilen/2

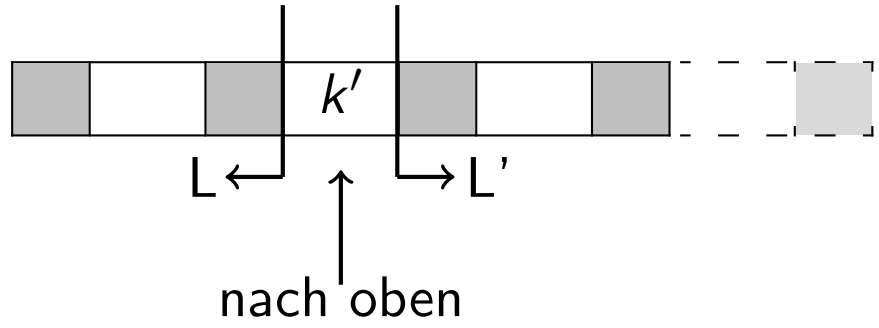
3. (k', L') in Elternknoten von L einfügen



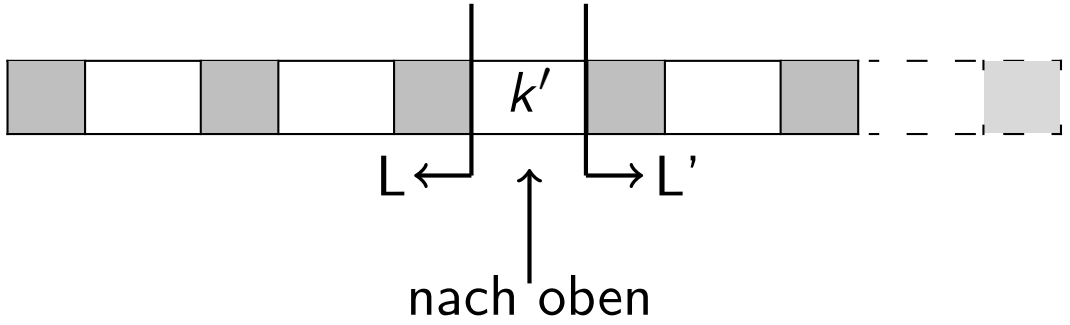
Innere Knoten teilen/3

$k' = T.k_{\lceil m/2 \rceil}$

- m gerade, z.B.: m=4

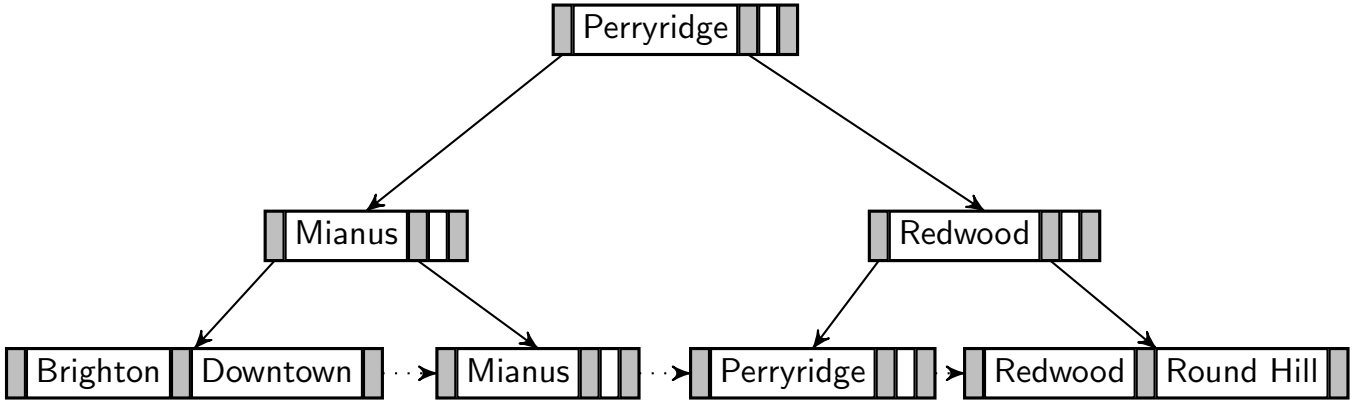


- m ungerade, z.B.: m=5

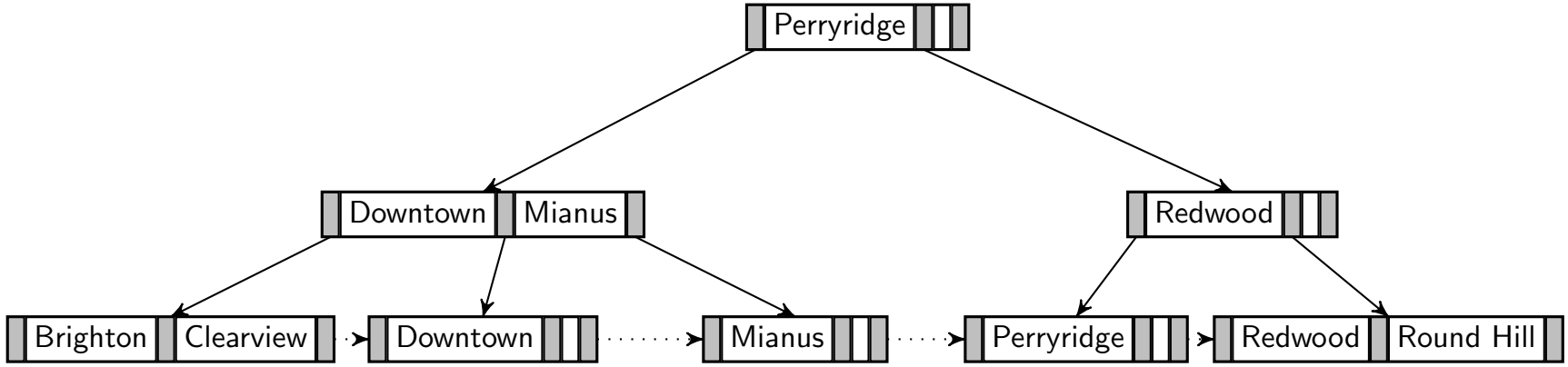


Beispiel: Einfügen in B^+-Baum/1

- B^+-Baum vor Einfügen von *Clearview*

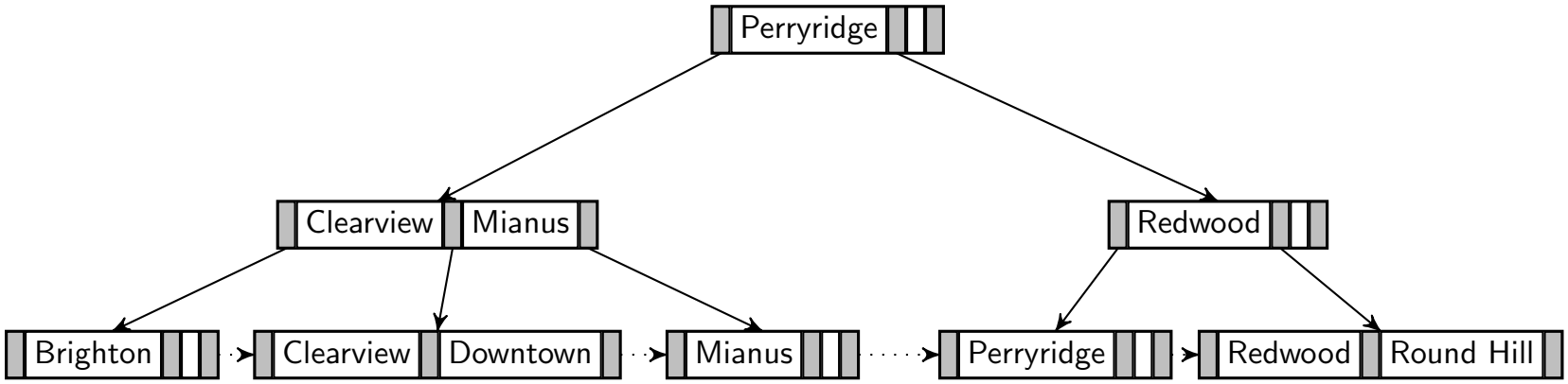


- B^+-Baum nach Einfügen von *Clearview*

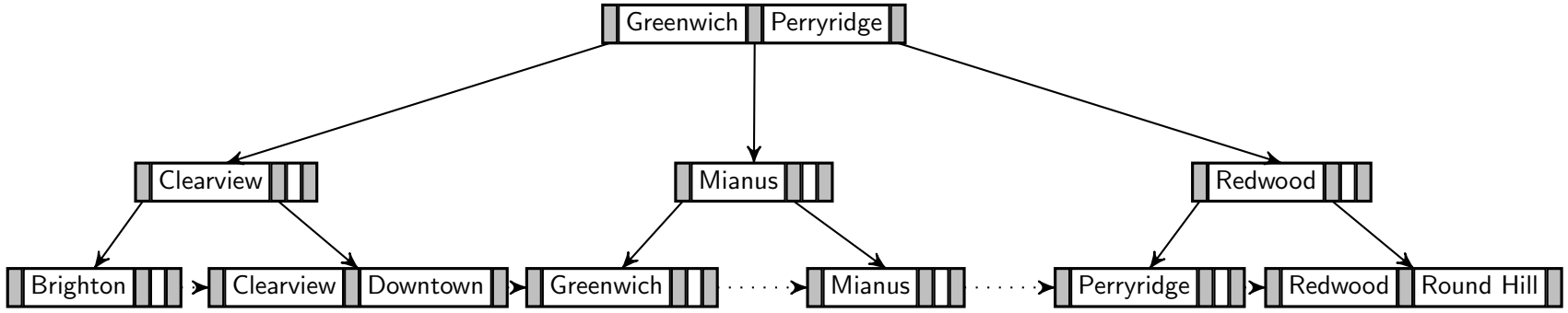


Beispiel: Einfügen in B^+-Baum/2

- B^+-Baum vor Einfügen von *Greenwich*



- B^+-Baum nach Einfügen von *Greenwich*



Löschen von B^+-Baum/1

Datensatz mit Suchschlüssel k löschen:

1. finde Blattknoten mit Suchschlüssel k
2. lösche k von Knoten
3. **falls** Knoten durch Löschen von k zu wenige Einträge hat:
   a. Einträge im Knoten und einem Geschwisterknoten passen in 1 Knoten **dann**:
      - **vereinige** die beiden Knoten in einen einzigen Knoten (den linken, falls er existiert; ansonsten den rechten) und lösche den anderen Knoten
      - lösche den Eintrag im Elternknoten der zwischen den beiden Knoten ist und wende Löschen rekursiv an
   b. Einträge im Knoten und einem Geschwisterknoten passen *nicht* in 1 Knoten **dann**:
      - **verteile** die Einträge zwischen den beiden Knoten sodass beide die minimale Anzahl von Einträgen haben
      - aktualisiere den entsprechenden Suchschlüssel im Eltern-Knoten

Löschen von B^+-Baum/2

- Vereinigung:
  - Vereinigung zweier Knoten propagiert im Baum nach oben bis ein Knoten mit mehr als $\lceil m/2 \rceil$ Kindern gefunden wird
  - falls die Wurzel nach dem Löschen nur mehr ein Kind hat, wird sie gelöscht und der Kind-Knoten wird zur neuen Wurzel

Algorithmus: Löschen im B^+-Baum

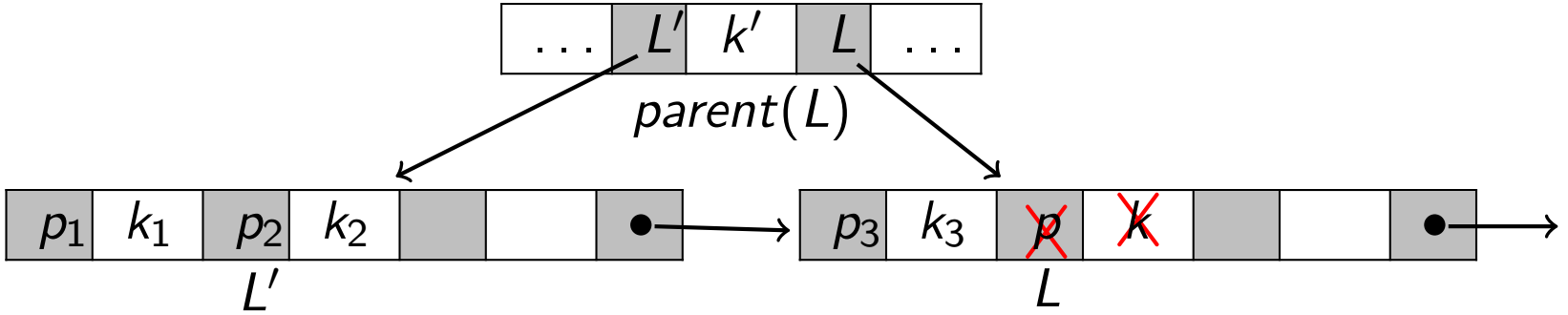
Algorithm 3: B+TreeDelete(L, k, p)

delete(k, p) from L
if *L is root* **and** *has only one remaining child* **then**
  make the child the new root and delete L
else if *L has too few values/pointers* **then**
  $L' \leftarrow$ previous sibling of L [next, if there is no previous];
  $k' \leftarrow$ value between L and L' in parent(L);
  if *entries in L and L' can fit in a single node* **then** `// vereinigen`
    if *L is a predecessor of L'* **then** swap L with L';
    if *L is not a leaf* **then** $L' \leftarrow L' \cup k'$ and all (k_i, p_i) from L;
    else $L' \leftarrow L' \cup$ all (k_i, p_i) from L;
    B+TreeDelete(parent(L), k', L);
  else `// verteilen`
    if *L' is a predecessor of L* **then**
      if *L is a nonleaf node* **then**
        remove the last (k, p) of L';
        insert the former last p of L' and k' as the first pointer and value in L;
      else move the last (p, k) of L' as the first pointer and value to L;
      replace k' in parent(L) by the former last k of L';
    else symmetric to the then case (switch first $\leftrightarrow$ last,...);

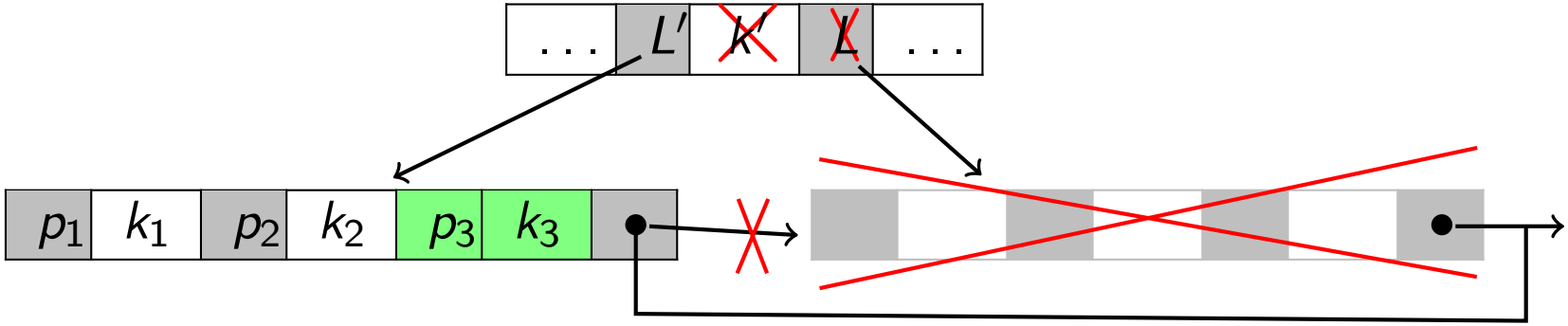
Löschen aus Blatt/1

(k, p) wird aus L gelöscht:

1. Vereinigen $(m = 4)$
   Vorher:



   Nachher:

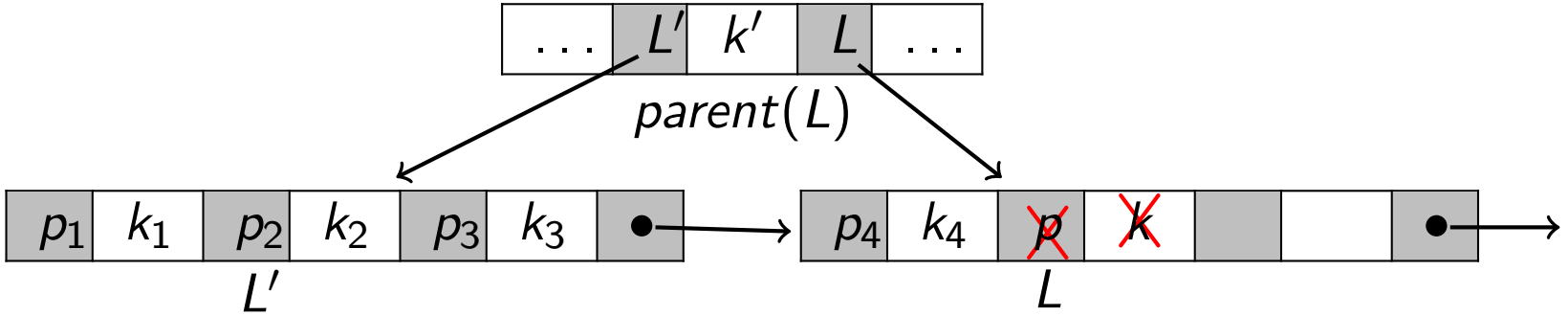


Löschen aus Blatt/2

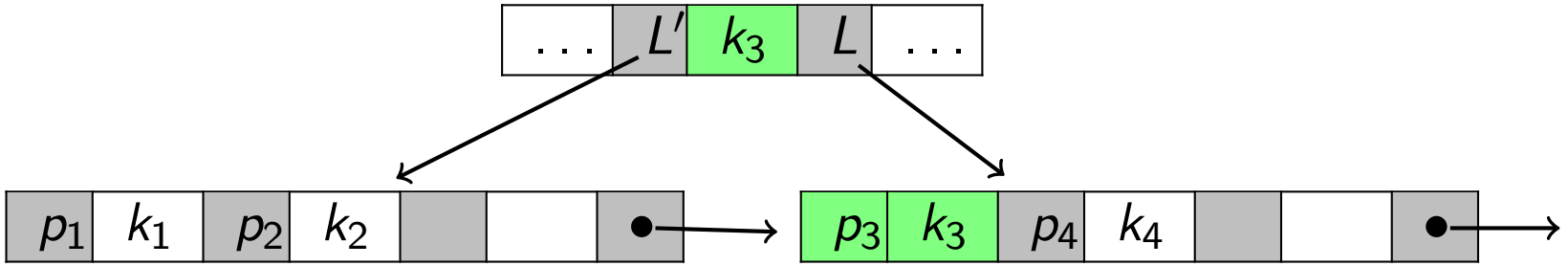
(k, p) wird aus L gelöscht:

2. Verteilen $(m = 4)$

   Vorher:



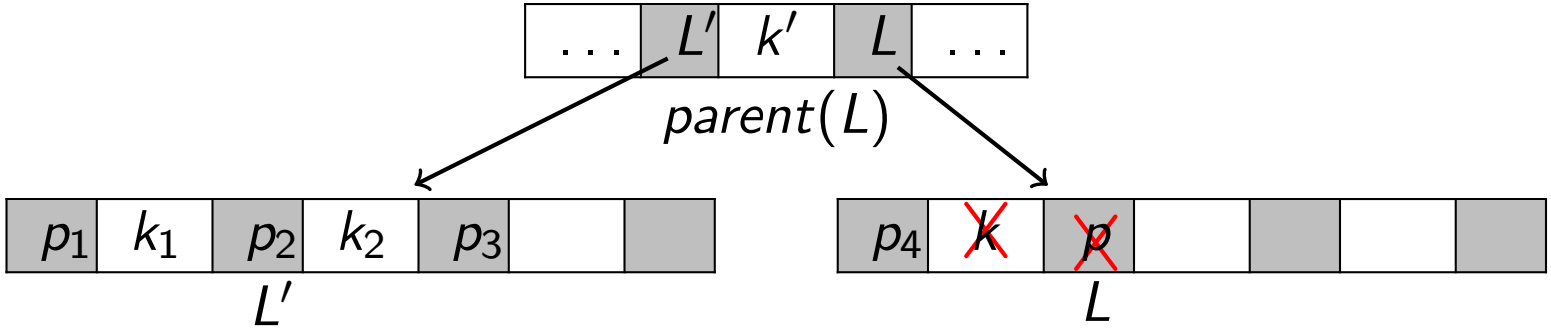
   Nachher:



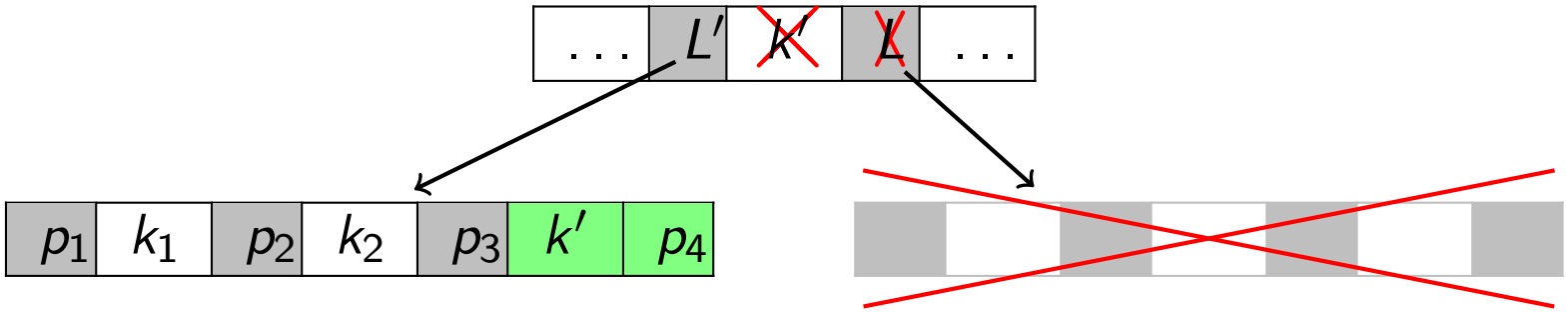
Löschen aus innerem Knoten/1

(k, p) wird aus L gelöscht:

1. Vereinigen $(m = 4)$
   Vorher:



   Nachher:

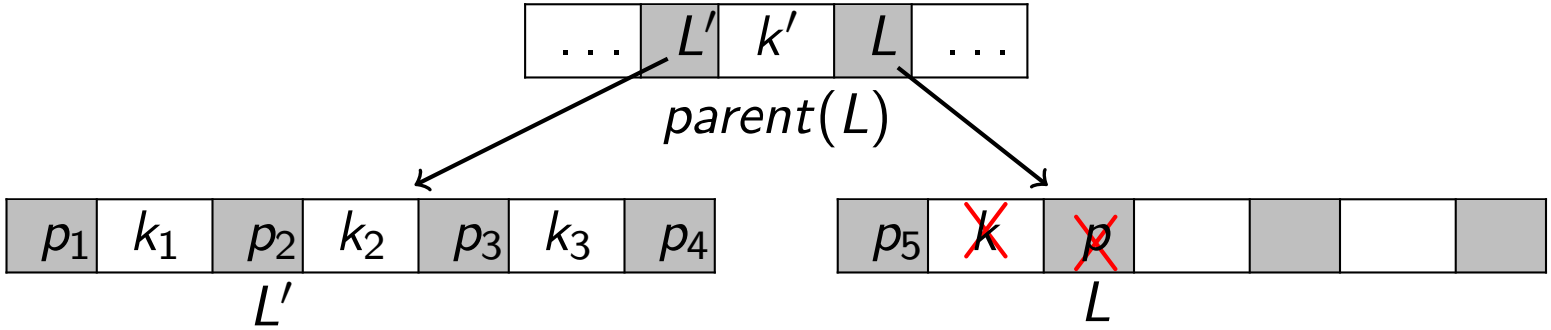


Löschen aus innerem Knoten/2

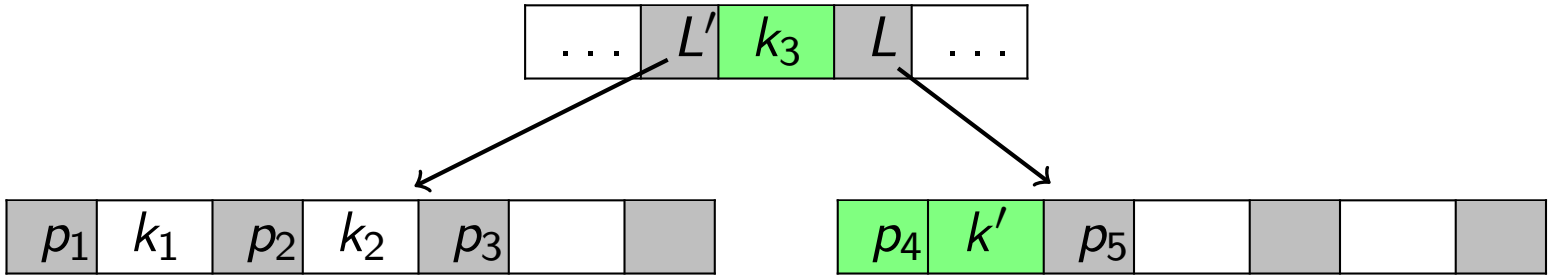
(k, p) wird aus L gelöscht:

2. Verteilen $(m = 4)$

   Vorher:

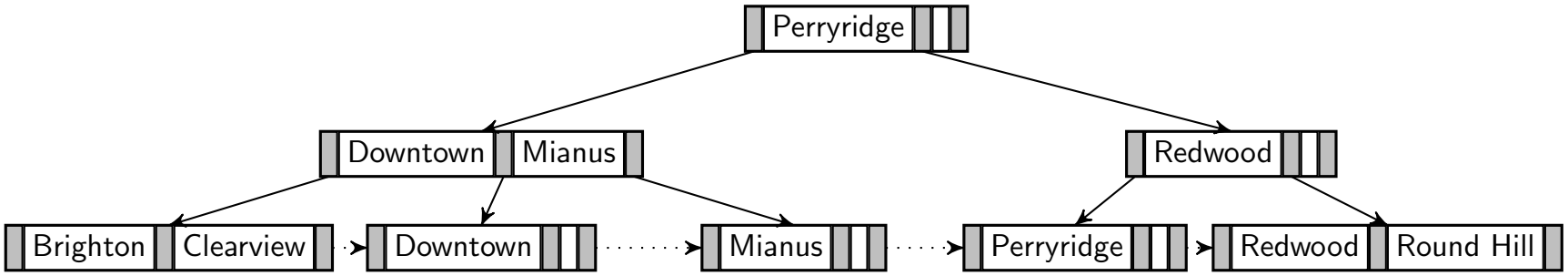


   Nachher:

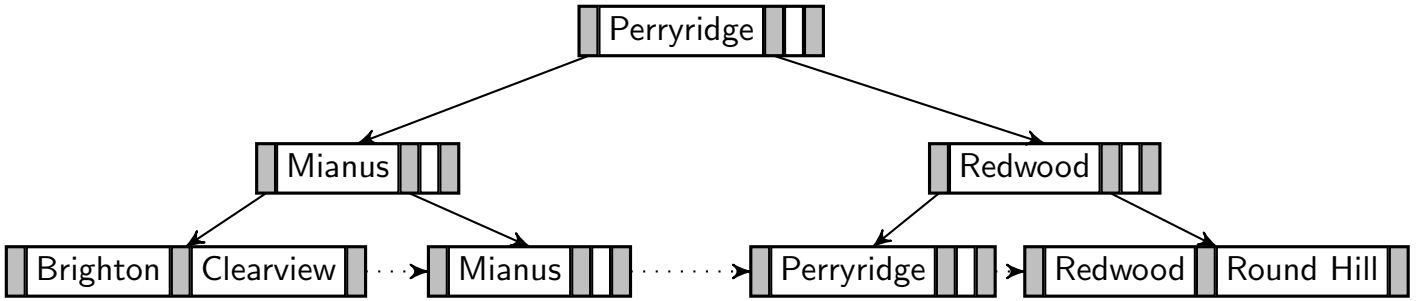


Beispiel: Löschen von B^+-Baum/1

- Vor Löschen von *Downtown*:



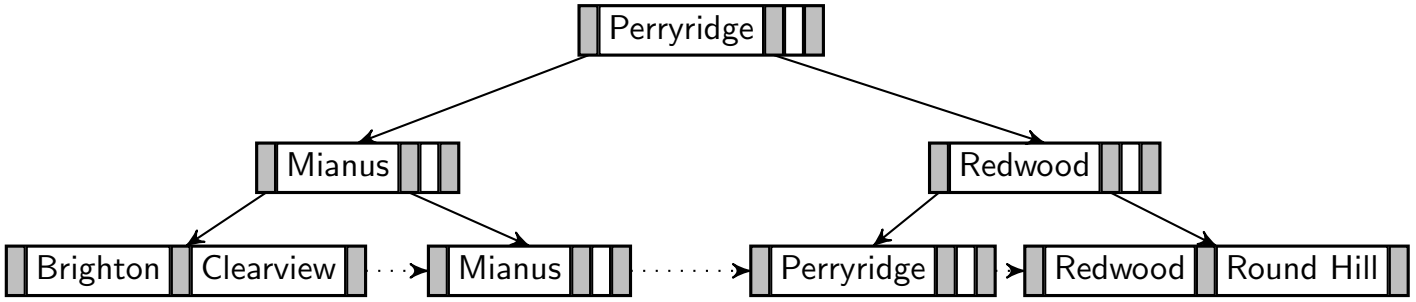
- Nach Löschen von *Downtown*:



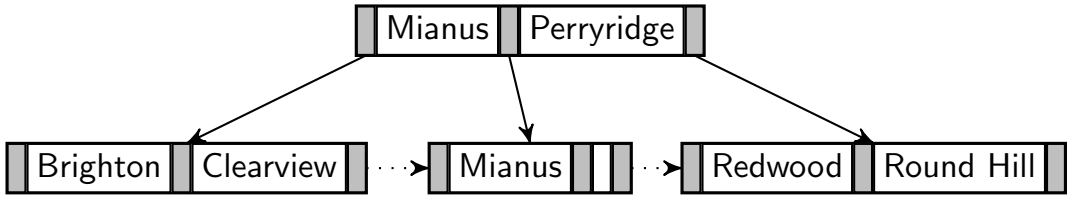
- Nach Löschen des Blattes mit *Downtown* hat der Elternknoten noch genug Pointer.
- Somit propagiert Löschen nicht weiter nach oben.

Beispiel: Löschen von B^+-Baum/2

- Vor Löschen von *Perryridge*:



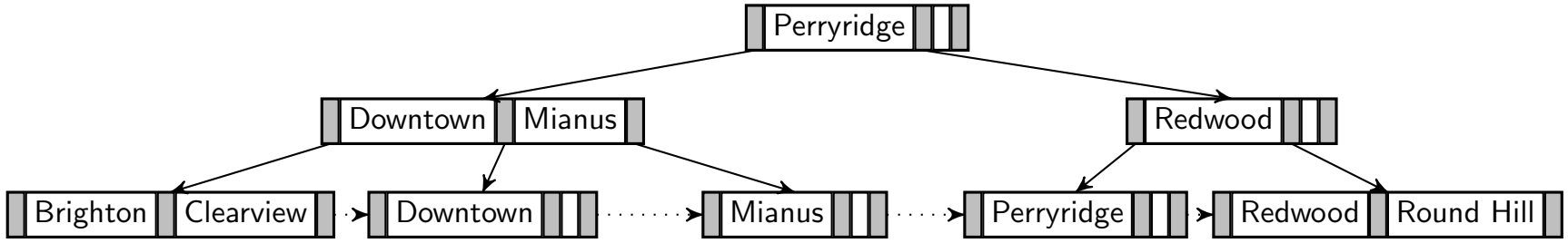
- Nach Löschen von *Perryridge*:



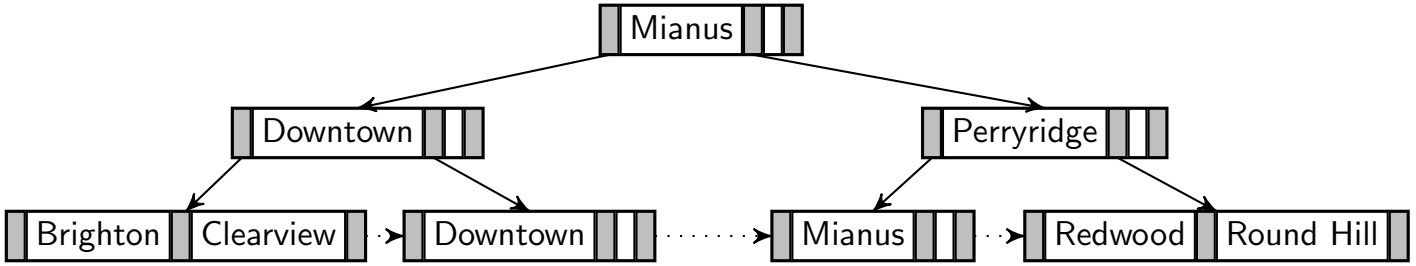
- Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und wird mit dem (rechten) Nachbarknoten vereinigt.
- Dadurch hat der Elternknoten zu wenig Pointer und wird mit seinem (linken) Nachbarknoten vereinigt (und ein Eintrag wird vom gemeinsamen Elternknoten gelöscht).
- Die Wurzel hat jetzt nur noch 1 Kind und wird gelöscht.

Beispiel: Löschen von B^+-Baum/3

- Vor Löschen von *Perryridge*:



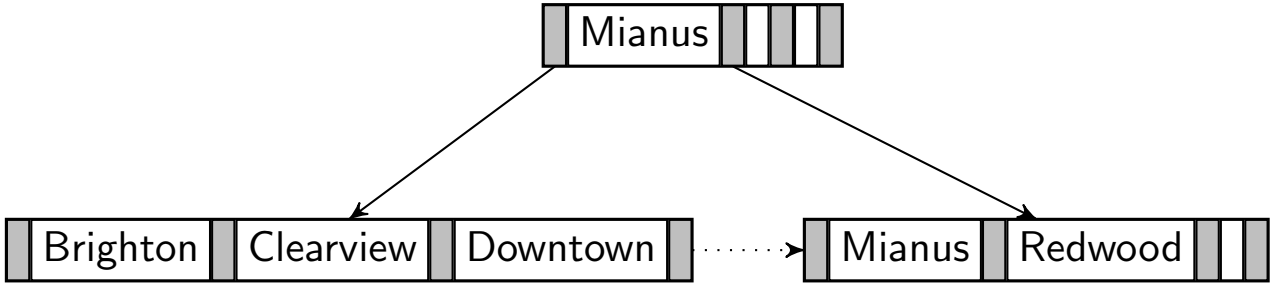
- Nach Löschen von *Perryridge*:



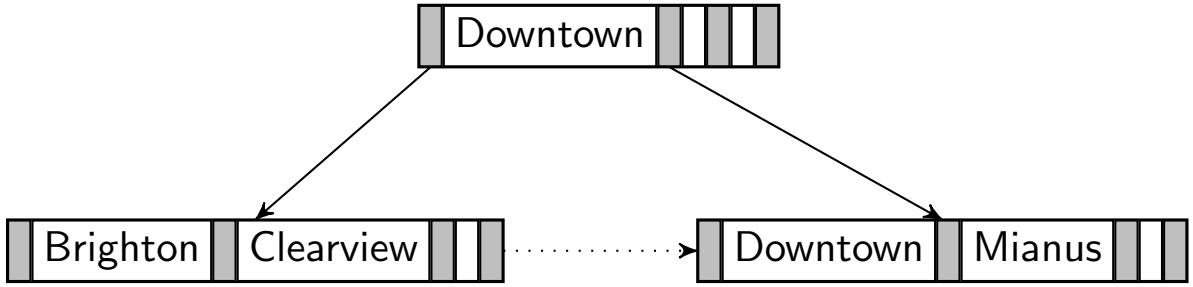
- Elternknoten von Blatt mit *Perryridge* hat durch Löschen zu wenig Einträge und erhält einen Pointer vom linken Nachbarn (Verteilung von Einträgen).
- Schlüssel im Elternknoten des Elternknotens (Wurzel in diesem Fall) ändert sich ebenfalls.

Beispiel: Löschen von B^+-Baum/4

- Vor Löschen von *Redwood*:



- Nach Löschen von *Redwood*:



- Knoten von Blatt mit *Redwood* hat durch Löschen zu wenig Einträge und erhält einen Eintrag vom linken Nachbarn (Verteilung von Einträgen).
- Schlüssel im Elternknoten (Wurzel in diesem Fall) ändert sich ebenfalls.

Zusammenfassung B^+-Baum

- Knoten mit Pointern verknüpft:
  - logisch nahe Knoten müssen nicht physisch nahe gespeichert sein
  - erlaubt mehr Flexibilität
  - erhöht die Anzahl der nicht-sequentiellen Zugriffe
- B^+-Bäume sind flach:
  - maximale Tiefe $\lceil log_{\lceil m/2 \rceil}(L) \rceil$ für L Blattknoten
  - m ist groß in der Praxis (z.B. $m = 200$)
- Suchschlüssel als "Wegweiser":
  - einige Suchschlüssel kommen als Wegweiser in einem oder mehreren inneren Knoten vor
  - zu einem Wegweiser gibt es nicht immer einen Suchschlüssel in einem Blattknoten (z.B. weil der entsprechende Datensatz gelöscht wurde)
- Einfügen und Löschen sind effizient:
  - nur $O(log(K))$ viele Knoten müssen geändert werden
  - Index degeneriert nicht, d.h. Index muss nie von Grund auf rekonstruiert werden

Inhalt

Statisches Hashing

- Nachteile von ISAM und B^+-Baum Indizes:
  - B^+-Baum: Suche muss Indexstruktur durchlaufen
  - ISAM: binäre Suche in großen Dateien
  - das erfordert zusätzliche Zugriffe auf Plattenblöcke
- Hashing:
  - erlaubt es auf Daten direkt und ohne Indexstrukturen zuzugreifen
  - kann auch zum Bauen eines Index verwendet werden

Hash Datei Organisation

- Statisches Hashing ist eine Form der Dateiorganisation:
  - Datensätze werden in Buckets gespeichert
  - Zugriff erfolgt über eine Hashfunktion
  - Eigenschaften: konstante Zugriffszeit, kein Index erforderlich
- Bucket: Speichereinheit die ein oder mehrere Datensätze enthält
  - ein Block oder mehrere benachbarte Blöcke auf der Platte
  - alle Datensätze mit bestimmtem Suchschlüssel sind im selben Bucket
  - Datensätze im Bucket können verschiedene Suchschlüssel haben
- Hash Funktion h: bildet Menge der Suchschlüssel K auf Menge der Bucket Adressen B ab
  - wird in konstanter Zeit (in der Anzahl der Datensätze) berechnet
  - mehrere Suchschlüssel können auf dasselbe Bucket abbilden
- Suchen eines Datensatzes mit Suchschlüssel:
  - verwende Hash Funktion um Bucket Adresse aufgrund des Suchschlüssels zu bestimmen
  - durchsuche Bucket nach Datensätzen mit Suchschlüssel

Beispiel: Hash Datei Organisation

- Beispiel: Organisation der Konto-Relation als Hash Datei mit Filialname als Suchschlüssel.
- 10 Buckets
- Numerischer Code des i-ten Zeichens im 26-Buchstaben-Alphabet wird als i angenommen, z.B., code(B)=2.
- Hash Funktion h
  - Summe der Codes aller Zeichen modulo 10:
  - h(Perryridge) = 125 mod 10 = 5
  - h(Round Hill) = 113 mod 10 = 3 (code(' ')=0)
  - h(Brighton) = 93 mod 10 = 3

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryrdige	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Funktionen/1

- Die Worst Case Hash Funktion bildet alle Suchschlüssel auf das gleiche Bucket ab.
  - Zugriffszeit wird linear in der Anzahl der Suchschlüssel.
- Die Ideale Hash Funktion hat folgende Eigenschaften:
  - Die Verteilung ist uniform (gleichverteilt), d.h. jedes Bucket ist der gleichen Anzahl von Suchschlüsseln aus der Menge aller Suchschlüssel zugewiesen.
  - Die Verteilung ist random (zufällig), d.h. im Mittel erhält jedes Bucket gleich viele Suchschlüssel unabhängig von der Verteilung der Suchschlüssel.

Hash Funktionen/2

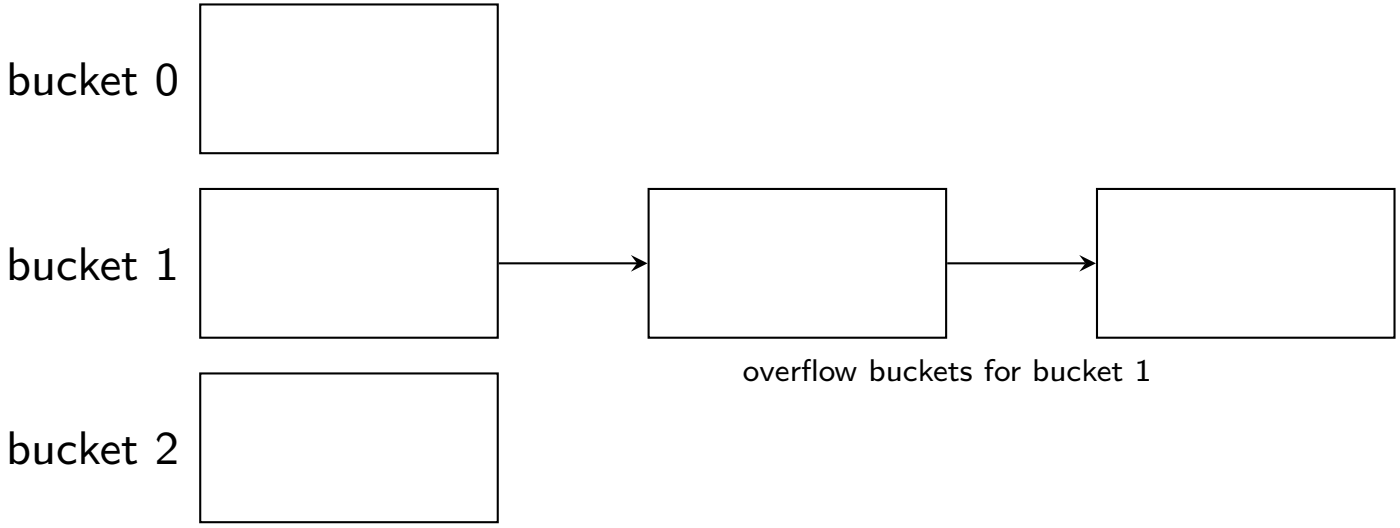
- Beispiel: 26 Buckets und eine Hash Funktion welche Filialnamen die mit dem i-ten Buchstaben beginnen dem Bucket i zuordnet.
  - keine Gleichverteilung, da es in der Domäne der Filialnamen (Menge aller möglichen Filialnamen) vermutlich mehr Filialen gibt die mit B beginnen als mit X.
- Beispiel: Hash Funktion die Kontostand nach gleich breiten Intervallen aufteilt: 1 - 10000 $\rightarrow$ 0, 10001 - 20000 $\rightarrow$ 1, usw.
  - uniform, da es für jedes Bucket gleich viele mögliche Werte von Kontostand gibt
  - nicht random, da Kontostände in bestimmten Intervallen häufiger sind, aber jedem Intervall 1 Bucket zugeordnet ist
- Typsiche Hash Funktion: Berechnung auf interner Binärdarstellung des Suchschlüssels, z.B. für String s mit n Zeichen, b Buckets:
  - $(s[0] + s[1] + \ldots + s[n-1]) \bmod b$, oder
  - $(31^{n-1}s[0] + 31^{n-2}s[1] + \ldots + s[n-1]) \bmod b$

Bucket Overflow/1

- Bucket Overflow: Wenn in einem Bucket nicht genug Platz für alle zugehörigen Datensätze ist, entsteht ein Bucket Overflow. Das kann aus zwei Gründen geschehen:
  - zu wenig Buckets
  - Skew: ungleichmäßige Verteilung der Hashwerte
- Zu wenig Buckets: die Anzahl n_B der Buckets muss größer gewählt werden als die Anzahl der Datensätze n geteilt durch die Anzahl der Datensätze pro Bucket f: $n_B > n/f$
- Skew: Ein Bucket ist überfüllt obwohl andere Buckets noch Platz haben. Zwei Gründe:
  - viele Datensätze haben gleichen Suchschlüssel (ungleichmäßige Verteiltung der Suchschlüssel)
  - Hash Funktion erzeugt ungleichmäßige Verteiltung
- Obwohl die Wahrscheinlichkeit für Overflows reduziert werden kann, können Overflows nicht gänzlich vermieden werden.
  - Overflows müssen behandelt werden
  - Behandlung durch Overflow Chaining

Bucket Overflow/2

- Overflow Chaining (closed addressing)
  - falls ein Datensatz in Bucket b eingefügt wird und b schon voll ist, wird ein Overflow Bucket b' erzeugt, in das der Datensatz gespeichert wird
  - die Overflow Buckets für Bucket b werden in einer Liste verkettet
  - für einen Suchschlüssel in Bucket b müssen auch alle Overflow Buckets von b durchsucht werden



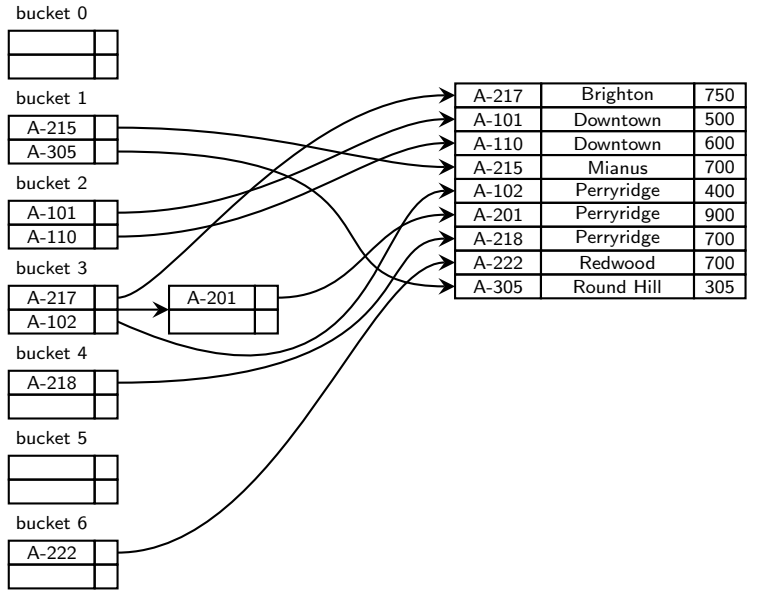
Bucket Overflow/3

- Open Addressing: Die Menge der Buckets ist fix und es gibt keine Overflow Buckets.
  - überzählige Datensätze werden in ein anderes (bereits vorhandenes) Bucket gegeben, z.B. das nächste das noch Platz hat (linear probing)
  - wird z.B. für Symboltabellen in Compilern verwendet, hat aber wenig Bedeutung in Datenbanken, da Löschen schwieriger ist

Hash Index

- Hash Index: organisiert (Suchschlüssel,Pointer) Paare als Hash Datei
  - Pointer zeigt auf Datensatz
  - Suchschlüssel kann mehrfach vorkommen

- Beispiel: Index auf Konto-Relation
  - Hash Funktion h: Quersumme der Kontonummer modulo 7
  - Beachte: Konto-Relation ist nach Filialnamen geordnet



- Hash Index ist immer Sekundärindex:
  - ist deshalb immer "dense"
  - Primär- bzw. Clustered Hash Index entspricht einer Hash Datei Organisation (zusätzliche Index-Ebene überflüssig)

Inhalt

Probleme mit Statischem Hashing

- Richtige Anzahl von Buckets ist kritisch für Performance:
  - zu wenig Buckets: Overflows reduzieren Performance
  - zu viele Buckets: Speicherplatz wird verschwendet (leere oder unterbesetzte Buckets)
- Datenbank wächst oder schrumpft mit der Zeit:
  - großzügige Schätzung: Performance leidet zu Beginn
  - knappe Schätzung: Performance leidet später
- Reorganisation des Index als einziger Ausweg:
  - Index mit neuer Hash Funktion neu aufbauen
  - sehr teuer, während der Reorganisation darf niemand auf die Daten schreiben
- Alternative: Anzahl der Buckets dynamisch anpassen

Dynamisches Hashing

- Dynamisches Hashing (dynamic hashing): Hash Funktion wird dynamisch angepasst.

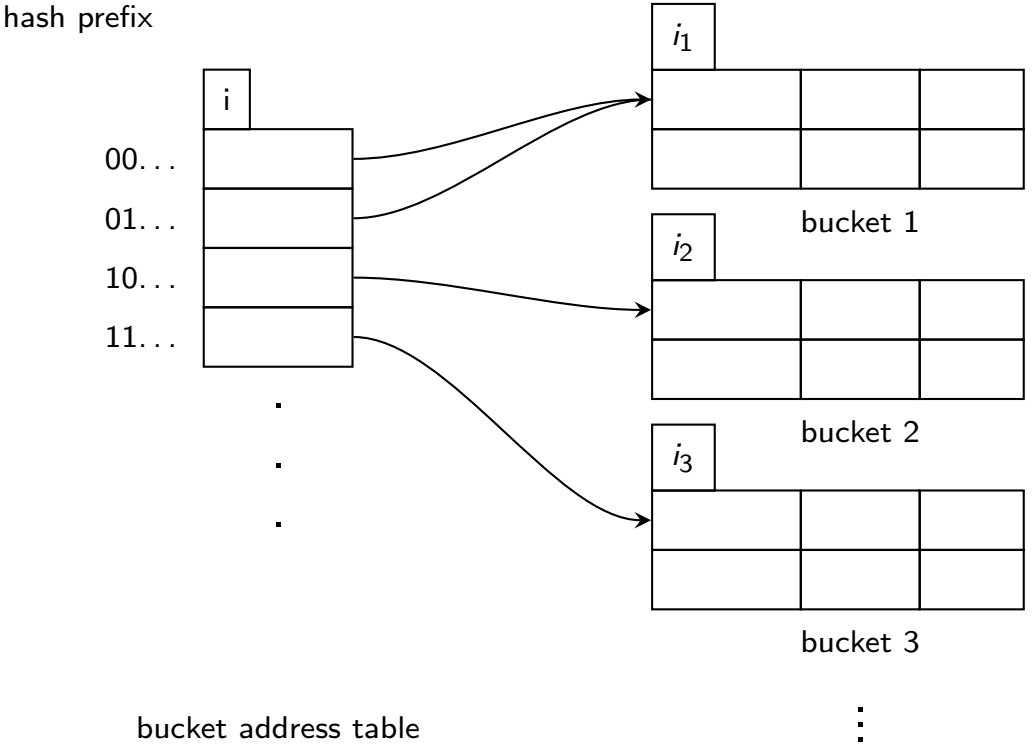
- Erweiterbares Hashing (extendible hashing): Eine Form des dynamischen Hashing.

Erweiterbares Hashing

- Hash Funktion h berechnet Hash Wert für sehr viele Buckets:
  - eine b-Bit Integer Zahl
  - typisch $b = 32$, also ~ 4 Milliarden (mögliche) Buckets
- Hash-Prefix:
  - nur die i höchstwertigen Bits (MSB) des Hash-Wertes werden verwendet
  - $0 \leq i \leq b$ ist die *globale Tiefe*
  - i wächst oder schrumpft mit Datenmenge, anfangs $i = 0$
- Verzeichnis: (directory, bucket address table)
  - Hauptspeicherstruktur: Array mit 2^i Einträgen
  - Hash-Prefix indiziert einen Eintrag im Verzeichnis
  - jeder Eintrag verweist auf ein Bucket
  - mehrere aufeinanderfolgende Einträge im Verzeichnis können auf dasselbe Bucket zeigen

Erweiterbares Hashing

- Buckets:
  - Anzahl der Buckets $\leq 2^i$
  - jedes Bucket j hat eine *lokale Tiefe* i_j
  - falls mehrere Verzeichnis-Pointer auf dasselbe Bucket j zeigen, haben die ensprechenden Hash Werte dasselbe i_j-Prefix.
- Beispiel: $i = 2$, $i_1 = 1$, $i_2 = i_3 = 2$,



Erweiterbares Hashing: Suche

- Suche: finde Bucket für Suchschlüssel K
  1. berechne Hash Wert $h(K) = X$
  2. verwende die i höchstwertigen Bits (Hash Prefix) von X als Adresse ins Verzeichnis
  3. folge dem Pointer zum entsprechenden Bucket

Erweiterbares Hashing: Einfügen

- Einfügen: füge Datensatz mit Suchschlüssel K ein
  1. verwende Suche um richtiges Bucket j zu finden
  2. **If** genug freier Platz in Bucket j **then**
     - füge Datensatz in Bucket j ein
  3. **else**
     - teile Bucket und versuche erneut

Erweiterbares Hashing: Bucket teilen

- Bucket j teilen um Suchschlüssel K einzufügen

  If $i > i_j$ (mehrere Pointer zu Bucket j) **then**
  - lege neues Bucket z an und setze i_z und i_j auf das alte $i_j + 1$
  - aktualisiere die Pointer die auf j zeigen (die Hälfte zeigt nun auf z)
  - lösche alle Datensätze von Bucket j und füge sie neu ein (sie verteilen sich auf Buckets j und z)
  - versuche K erneut einzufügen

  Else if $i = i_j$ (nur 1 Pointer zu Bucket j) **then**
  - erhöhe i und verdopple die Größe des Verzeichnisses
  - ersetze jeden alten Eintrag durch zwei neue Einträge die auf dasselbe Bucket zeigen
  - versuche K erneut einzufügen

- Overflow Buckets müssen nur erzeugt werden, wenn das Bucket voll ist und die Hashwerte aller Suchschlüssel im Bucket identisch sind (d.h., teilen würde nichts nützen)

Integrierte Übung 2.2

Betrachten Sie die folgende Hashfunktion:

Schlüssel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Nehmen Sie Buckets der Größe 2 an und erweiterbares Hashing mit einem anfangs leeren Verzeichnis. Zeigen Sie die Hashtabelle nach folgenden Operationen:

- füge 1 Brighton und 2 Downtown Datensätze ein
- füge 1 Mianus Datensatz ein
- füge 1 Redwood Datensatz ein
- füge 3 Perryridge Datensätze ein

Erweiterbares Hashing: Löschen

- Löschen eines Suchschlüssels K
  1. suche Bucket j für Suchschlüssel K
  2. entferne alle Datensätze mit Suchschlüssel K
  3. Bucket j kann mit Nachbarbucket(s) verschmelzen falls
     - alle Suchschlüssel in einem Bucket Platz finden
     - die Buckets dieselbe lokale Tiefe i_j haben
     - die $i_j - 1$ Prefixe der entsprechenden Hash-Werte identisch sind
  4. Verzeichnis kann verkleinert werden, wenn $i_j < i$ für alle Buckets j

Integrierte Übung 2.3

Betrachten Sie die folgende Hashfunktion:

Schlüssel	Hashwert
Brighton	0010
Downtown	1010
Mianus	1100
Perryridge	1111
Redwood	0011

Gehen Sie vom Ergebnis der vorigen Übung aus und führen Sie folgende Operationen durch:

- 1 Brighton und 1 Downtown löschen
- 1 Redwood löschen
- 2 Perryridge löschen

Erweiterbares Hashing: Pro und Kontra

- Vorteile von erweiterbarem Hashing
  - bleibt effizient auch wenn Datei wächst
  - Overhead für Verzeichnis ist normalerweise klein im Vergleich zu den Einsparungen an Buckets
  - keine Buckets für zukünftiges Wachstum müssen reserviert werden
- Nachteile von erweiterbarem Hashing
  - zusätzliche Ebene der Indirektion – macht sich bemerkbar, wenn Verzeichnis zu groß für den Hauptspeicher wird
  - Verzeichnis vergrößern oder verkleinern ist relativ teuer

B^+-Baum vs. Hash Index

- Hash Index degeneriert wenn es sehr viele identische (Hashwerte für) Suchschlüssel gibt – Overflows!
- Im Average Case für Punktanfragen in n Datensätzen:
  - Hash index: $O(1)$ (sehr gut)
  - B^+-Baum: $O(\log n)$
- Worst Case für Punktanfragen in n Datensätzen:
  - Hash index: $O(n)$ (sehr schlecht)
  - B^+-Baum: $O(\log n)$
- Anfragetypen:
  - Punktanfragen: Hash und B^+-Baum
  - Mehrpunktanfragen: Hash und B^+-Baum
  - Bereichsanfragen: Hash Index nicht brauchbar

Inhalt

Zugriffe über mehrere Suchschlüssel/1

- Wie kann Index verwendet werden, um folgende Anfrage zu beantworten?

  select *AccNr*
  from *account*
  where *BranchName* = *"Perryridge"* **and** *Balance* = *1000*

- Strategien mit mehreren Indizes (jeweils 1 Suchschlüssel):
  a) *BranchName* = *"Perryridge"* mit Index auf *BranchName* auswerten; auf Ergebnis-Datensätzen *Balance* = *1000* testen.
  b) *Balance* = *1000* mit Index auf *Balance* auswerten; auf Ergebnis-Datensätzen *BranchName* = *"Perryridge"* testen.
  c) Verwende *BranchName* Index um Pointer zu Datensätzen mit *BranchName* = *"Perryridge"* zu erhalten; verwende *Balance* Index für Pointer zu Datensätzen mit *Balance* = *1000*; berechne die Schnittmenge der beiden Pointer-Mengen.

Zugriffe über mehrere Suchschlüssel/2

- Nur die dritte Strategie nützt das Vorhandensein mehrerer Indizes.
- Auch diese Strategie kann eine schlechte Wahl sein:
  - es gibt viele Konten in der "Perryridge" Filiale
  - es gibt viele Konten mit Kontostand 1000
  - es gibt nur wenige Konten die beide Bedingungen erfüllen
- Effizientere Indexstrukturen müssen verwendet werden:
  - (traditionelle) Indizes auf kombinierten Schlüsseln
  - spezielle mehrdimensionale Indexstrukturen, z.B., Grid Files, Quad-Trees, Bitmap Indizes.

Zugriffe über mehrere Suchschlüssel/3

- Annahme: Geordneter Index mit kombiniertem Suchschlüssel *(BranchName, Balance)*
- Kombinierte Suchschlüssel haben eine Ordnung (*BranchName* ist das erstes Attribut, *Balance* ist das zweite Attribut)
  - Folgende Bedingung wird effizient behandelt (alle Attribute):
    where *BranchName* = *"Perryridge"* **and** *Balance* = *1000*
  - Folgende Bedingung wird effizient behandelt (Prefix):
    where *BranchName* = *"Perryridge"*
  - Folgende Bedingung ist ineffizient (kein Prefix der Attribute):
    where *Balance* = *1000*

Inhalt

Index Definition in SQL

- SQL-92 definiert keine Syntax für Indizes da diese nicht Teil des logischen Datenmodells sind.
- Jedoch alle Datenbanksysteme stellen Indizes zur Verfügung.
- Index erzeugen:

  create index <IdxName> **on** <RelName> (<AttrList>)

  z.B. **create index** BrNaIdx **on** branch (branch-name)
- **Create unique index** erzwingt eindeutige Suchschlüssel und definiert indirekt ein Schlüsselattribut.
- Primärschlüssel (**primary key**) und Kandidatenschlüssel (**unique**) werden in SQL bei der Tabellendefinition spezifiziert.
- Index löschen:

  drop index <index-name>

  z.B. **drop index** BrNaIdx

Beispiel: Indizes in PostgreSQL

- **CREATE [UNIQUE] INDEX** name **ON** table_name
  "(" col [**DESC**] { "," col [**DESC**] } ")" [...]

- Beispiele:
  - **CREATE INDEX** MajIdx **ON** Enroll (Major);
  - **CREATE INDEX** MajIdx **ON** Enroll **USING HASH** (Major);
  - **CREATE INDEX** MajMinIdx **ON** Enroll (Major, Minor);

Indexes in Oracle

- B^+-Baum Index in Oracle:

  CREATE [UNIQUE] INDEX name **ON** table_name
  "(" col [**DESC**] { "," col [**DESC**] } ")" [**pctfree** n] [...]

- Anmerkungen:
  - pct_free gibt an, wieviel Prozent der Knoten anfangs frei sein sollen.
  - UNIQUE sollte nicht verwendet werden, da es ein logisches Konzept ist.
  - Oracle erstellt einen B^+-Baum Index für jede **unique** oder **primary key** definition bei der Erstellung der Tabelle.

- Beispiele:

  CREATE TABLE BOOK (
  ISBN **INTEGER**, Author **VARCHAR2 (30)** , ...);
  CREATE INDEX book_auth **ON** book(Author);

Anmerkungen zu Indizes in Datenbanksystemen

- Indizes werden automatisch nachgeführt wenn Tupel eingefügt, geändert oder gelöscht werden.
- Indizes verlangsamen deshalb Änderungsoperationen.
- Einen Index zu erzeugen kann lange dauern.
- Bulk Load: Es ist (viel) effizienter, zuerst die Daten in die Tabelle einzufügen und nachher alle Indizes zu erstellen als umgekehrt.

Zusammenfassung

- Index Typen:
  - Primary, Clustering und Sekundär
  - Dense oder Sparse
- B^+-Baum:
  - universelle Indexstruktur, auch für Bereichsanfragen
  - Garantien zu Tiefe, Füllgrad und Effizienz
  - Einfügen und Löschen
- Hash Index:
  - statisches und erweiterbares Hashing
  - kein Index für Primärschlüssel nötig
  - gut für Prädikate mit "="
- Mehrschlüssel Indizes: schwieriger, da es keine totale Ordnung in mehreren Dimensionen gibt
- Indizes in SQL

Datenbanken 2

Anfragebearbeitung

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 8. Januar 2019

Inhalt

Literatur und Quellen

Lektüre zum Thema "Anfragebearbeitung":

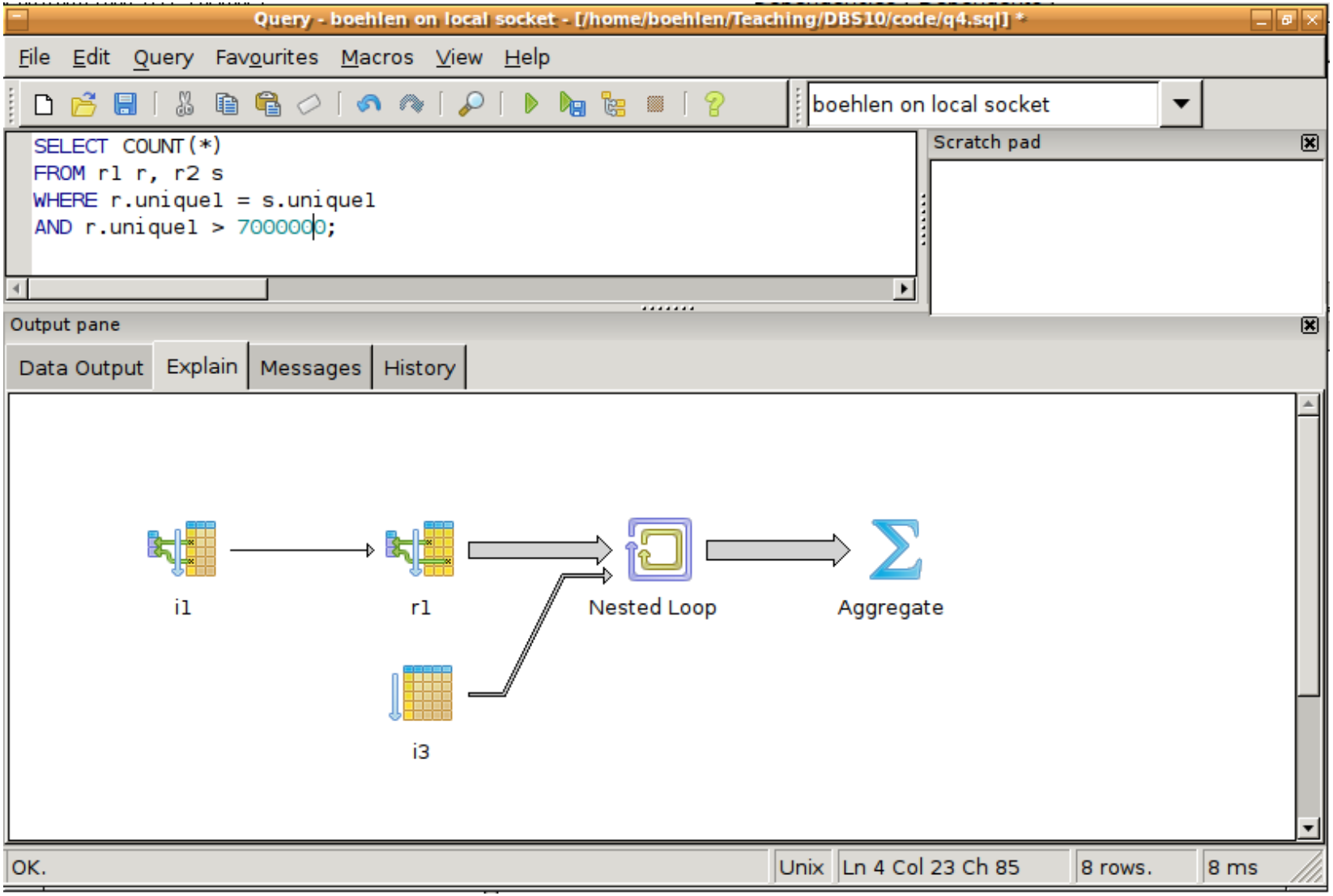
- Kapitel 8 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 12 in Silberschatz, Korth, and Sudarashan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

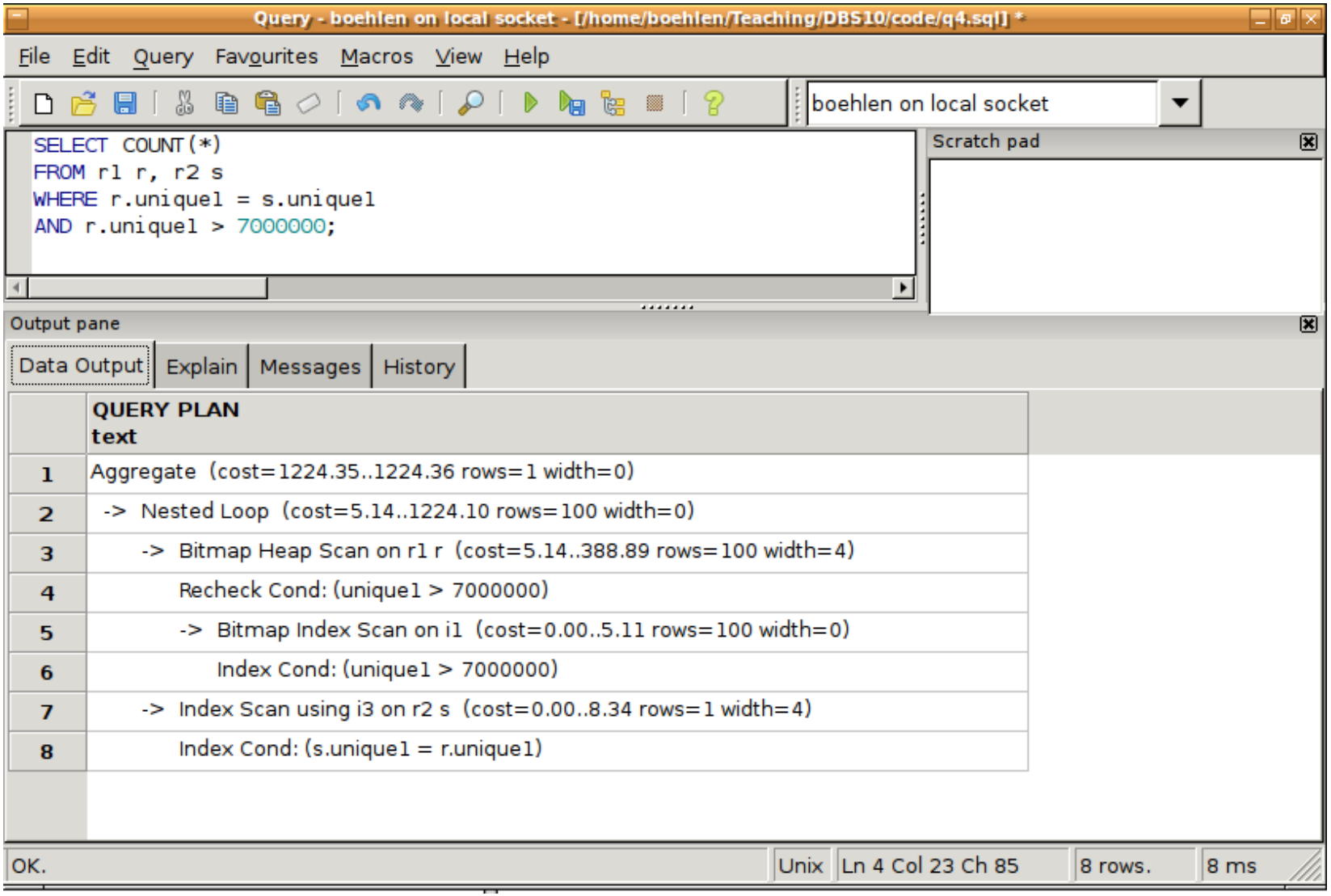
- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Inhalt

PostgreSQL Beispiel/1



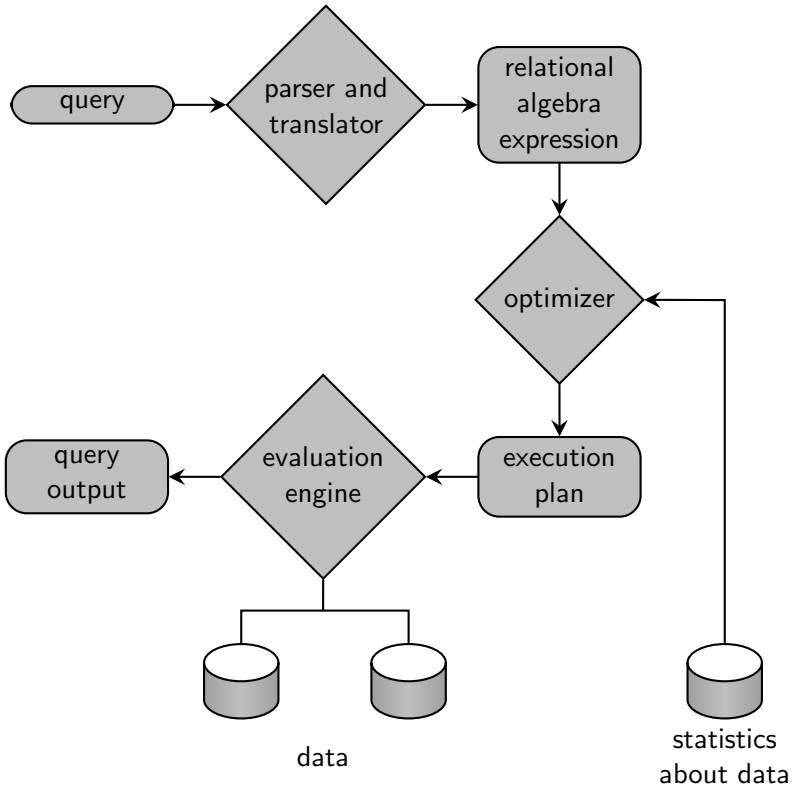
PostgreSQL Beispiel/2



Anfragebearbeitung

- Effizienter Auswertungsplan gehört zu den wichtigsten Aufgaben eines DBMS.
- Selektion und Join sind dabei besonders wichtig.
- 3 Schritte der Anfragebearbeitung:

1. **Parsen und übersetzen**
   (von SQL in Rel. Alg.)
2. **Optimieren**
   (Auswertungsplan erstellen)
3. **Auswerten**
   (Auswertungsplan ausführen)



Inhalt

Anfragekosten/1

- Anfragekosten werden als gesamte benötigte Zeit verstanden.
- Mehrere Faktoren tragen zu den Anfragekosten bei:
  - CPU
  - Netzwerk Kommunikation
  - Plattenzugriff
    - sequentielles I/O
    - random I/O
  - Puffergröße
- Puffergröße:
  - mehr Puffer-Speicher (RAM) reduziert Anzahl der Plattenzugriffe
  - verfügbarer Puffer-Speicher hängt von anderen OS Prozessen ab und ist schwierig von vornherein festzulegen
  - wir verwenden oft worst-case Anschätzung mit der Annahme, dass nur der mindest nötige Speicher vorhanden ist

Anfragekosten/2

- Plattenzugriff macht größten Teil der Kosten einer Anfrage aus.
- Kosten für Plattenzugriff relativ einfach abzuschätzen als Summe von:
  - Anzahl der Spurwechsel * mittlere Spurwechselzeit (avg. seek time)
  - Anzahl der Block-Lese-Operationen * mittlere Block-lese-Zeit
  - Anzahl der Block-Schreib-Operationen * mittlere Block-schreib-Zeit
    - → Block schreiben ist teuerer als lesen, weil geschriebener Block zur Kontrolle nochmal gelesen wird.
- Zur Vereinfachung
  - zählen wir nur die Anzahl der Schreib-/Lese-Operationen
  - berücksichtigen wir nicht die Kosten zum Schreiben des Ergebnisses auf die Platte

Inhalt

Sorting

- Sortieren ist eine wichtige Operation:
  - SQL-Anfragen können explizit eine sortierte Ausgabe verlangen
  - mehrere Operatoren (z.B. Joins) können effizient implementiert werden, wenn die Relationen sortiert sind
  - oft ist Sortierung der entscheidende erste Schritt für einen effizienten Algorithmus
- Sekundärindex für Sortierung verwenden?
  - Index sortiert Datensätze nur logisch, nicht physisch.
  - Datensätze müssen über Pointer im Index zugegriffen werden.
  - Für jeden Pointer (Datensatz) muss möglicherweise ein eigener Block von der Platte gelesen werden.
- Algorithmen je nach verfügbarer Puffergröße:
  - Relation kleiner als Puffer: Hauptspeicher-Algorithmen wie **Quicksort**
  - Relation größer als Puffer: Platten-Algorithmen wie **Mergesort**

Externes Merge-Sort/1

- Grundidee:
  - **teile** Relation in Stücke (Läufe, *runs*) die in den Puffer passen
  - **sortiere** jeden Lauf im Puffer und schreibe ihn auf die Platte
  - **mische** sortierte Läufe so lange, bis nur mehr ein Lauf übrig ist
- Notation:
  - b: Anzahl der Plattenblöcke der Relation
  - M: Anzahl der Blöcke im Puffer (Hauptspeicher)
  - $N = \lceil b/M \rceil$: Anzahl der Läufe

Externes Merge-Sort/2

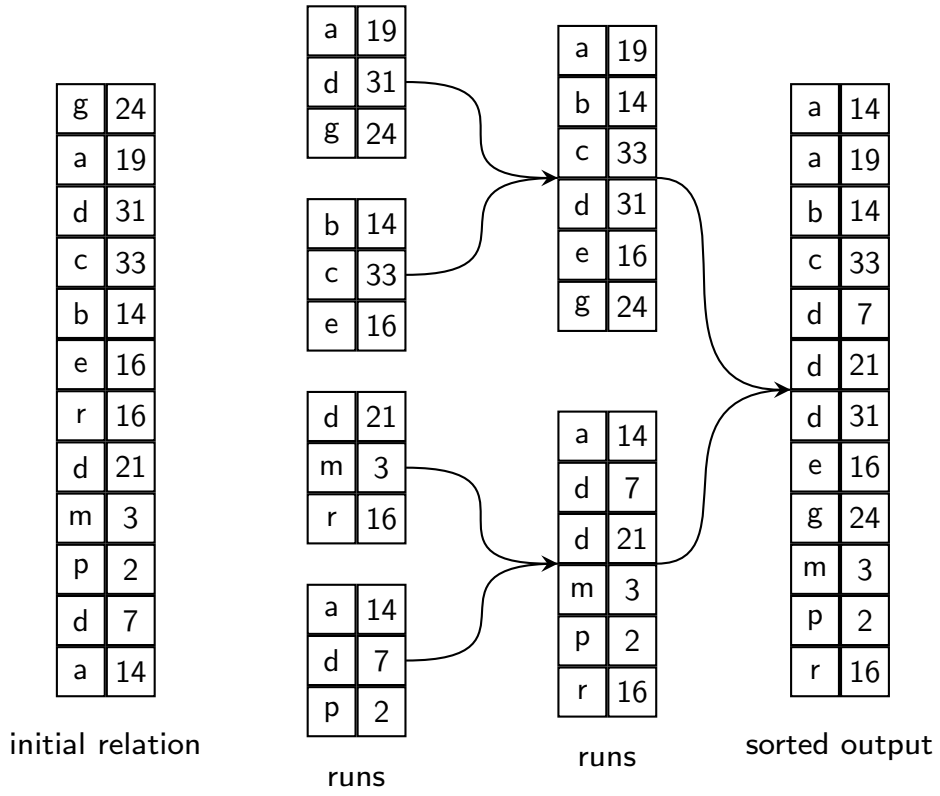
- Schritt 1: erzeuge N Läufe
  1. starte mit $i = 0$
  2. wiederhole folgende Schritte bis Relation leer ist:
     a. lies M Blöcke der Relation (oder den Rest) in Puffer
     b. sortiere Tupel im Puffer
     c. schreibe sortierte Daten in Lauf-Datei L_i
     d. erhöhe i
- Schritt 2: mische Läufe (N-Wege-Mischen) (Annahme $N < M$)
  (N Blöcke im Puffer für Input, 1 Block für Output)
  1. lies ersten Block jeden Laufs L_i in Puffer Input Block i
  2. wiederhole bis alle Input Blöcke im Puffer leer sind:
     a. wähle erstes Tupel in Sortierordnung aus allen nicht-leeren Input Blöcken
     b. schreibe Tupel auf Output Block; falls der Block voll ist, schreibe ihn auf die Platte
     c. lösche Tupel vom Input Block
     d. falls Block i nun leer ist, lies nächsten Block des Laufs L_i

Externes Merge-Sort/3

- Falls $N \geq M$, werden mehrere Misch-Schritte (Schritt 2) benötigt.
- Pro Durchlauf...
  - werden jeweils $M - 1$ Läufe gemischt
  - wird die Anzahl der Läufe um Faktor $M - 1$ reduziert
  - werden die Läufe um den Faktor $M - 1$ größer
- Durchläufe werden wiederholt bis nur mehr ein Lauf übrig ist.
- Beispiel: Puffergröße $M = 11$, Anzahl Blocks $b = 1100$
  - $N = \lceil b/M \rceil = 100$ Läufe à 11 Blocks werden erzeugt
  - nach erstem Durchlauf: 10 Läufe à 110 Blocks
  - nach zweitem Durchlauf: 1 Lauf à 1100 Blocks

Externes Merge-Sort/4

- Beispiel: M = 3, 1 Block = 1 Tupel



Externes Merge-Sort/5

- Kostenanalyse:
  - b: Anzahl der Blocks in Relation R
  - anfängliche Anzahl der Läufe: b/M
  - gesamte Anzahl der Misch-Durchläufe: $\lceil log_{M-1}(b/M) \rceil$
    - $\rightarrow$ die Anzahl der Läufe sinkt um den Faktor $M-1$ pro Misch-Durchlauf
  - Plattenzugriffe für Erzeugen der Läufe und für jeden Durchlauf: $2 * b$
    - $\rightarrow$ Ausnahme: letzter Lauf hat keine Schreibkosten
- Kosten für externes Merge-Sort: Anzahl der gelesenen oder geschriebenen Blöcke

$$Kosten = b(2\lceil log_{M-1}(b/M) \rceil + 1)$$

- Beispiel: Kostenanalyse für voriges Beispiel:
  - $M = 3, b = 12$
  - $12 * (2 * \lceil \log_2(12/3) \rceil + 1) = 60$ Schreib-/Lese-/Operationen

Inhalt

Auswertung der Selektion/1

- Der Selektionsoperator:
  - **select** * **from** R **where** θ
  - $\sigma_\theta(R)$

  berechnet die Tupel von R welche das Selektionsprädikat (=Selektionsbedingung) θ erfüllen.
- Selektionsprädikat θ ist aus folgenden Elementen aufgebaut:
  - Attributnamen der Argumentrelation R oder Konstanten als Operanden
  - arithmetische Vergleichsoperatoren ($=$, $<$, $>$)
  - logische Operatoren: $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
- Strategie zur Auswertung der Selektion hängt ab
  - von der Art des Selektionsprädikats
  - von den verfügbaren Indexstrukturen

Auswertung der Selektion/2

Grundstrategien für die Auswertung der Selektion:

- Sequentielles Lesen der Datei (file scan):
  - Klasse von Algorithmen welche eine Datei Tupel für Tupel lesen um jene Tupel zu finden, welche die Selektionsbedingung erfüllen
  - grundlegendste Art der Selektion
- Index Suche (index scan):
  - Klasse von Algorithmen welche einen Index benutzen
  - Index wird benutzt um eine Vorauswahl von Tupeln zu treffen
  - Beispiel: B^+-Baum Index auf A und Gleichheitsbedingung: $\sigma_{A=5}(R)$

Auswertung der Selektion/3

Arten von Prädikaten:

- Gleichheitsanfrage: $\sigma_{A=V}(R)$
- Bereichsanfrage: $\sigma_{A<V}(R)$ oder $\sigma_{A>V}(R)$
- Konjunktive Selektion: $\sigma_{\theta_1 \wedge \theta_2 \cdots \wedge \theta_n}(\mathrm{R})$
- Disjunktive Selektion: $\sigma_{\theta_1 \vee \theta_2 \cdots \vee \theta_n}(\mathrm{R})$

Auswertung der Selektion/4

A1 Lineare Suche: Lies jeden einzelnen Block der Datei und überprüfe jeden Datensatz ob er die Selektionsbedingung erfüllt.

- Ziemlich teuer, aber immer anwendbar, unabhängig von:
  - (nicht) vorhandenen Indexstrukturen
  - Sortierung der Daten
  - Art der Selektionsbedingung
- Hintereinanderliegende Blöcke lesen wurde von den Plattenherstellern optimiert und ist schnell hinsichtlich Spurwechsel und Latenz (pre-fetching)
- Kostenabschätzung (b = Anzahl der Blöcke in der Datei):
  - Worst case: $Cost = b$
  - Selektion auf Kandidatenschlüssel: $Mittlere\ Kosten = b/2$
    (Suche beenden, sobald erster Datensatz gefunden wurde)

Auswertung der Selektion/5

A2 Binäre Suche: verwende binäre Suche auf Blöcken um Tupel zu finden, welche Bedingung erfüllen.

- Anwendbar falls
  - die Datensätze der Tabelle physisch sortiert sind
  - die Selektionsbedingung auf dem Sortierschlüssel formuliert ist
- Kostenabschätzung für $\sigma_{A=C}(R)$:
  - $\lfloor log_2(b) \rfloor + 1$: Kosten zum Auffinden des ersten Tupels
  - plus Anzahl der weiteren Blöcke mit Datensätzen, welche Bedingung erfüllen (diese liegen alle nebeneinander in der Datei)

Auswertung der Selektion/6

Annahme: Index ist B^+-Baum mit H Ebenen[1]

A3 Primärindex + Gleichheitsbedingung auf Suchschlüssel

- gibt einen einzigen Datensatz zurück
- *Kosten* $= H + 1$ (Knoten im B^+-Baum + 1 Datenblock)

A3 Clustered Index + Gleichheitsbedingung auf Suchschlüssel

- gibt mehrere Datensätze zurück
- alle Ergebnisdatensätze liegen hintereinander in der Datei
- *Kosten* $= H +$ *# Blöcke mit Ergebnisdatensätzen*

[1] $H \leq \lceil \log_{\lceil m/2 \rceil}(K) \rceil + 1$ für B$^+$-Baum mit K Suchschlüsseln, Knotengrad m

Auswertung der Selektion/7

A5 Sekundärindex + Gleichheitsbedingung auf Suchschlüssel

- Suchschlüssel ist Kandidatenschlüssel
  - gibt einen einzigen Datensatz zurück
  - $\mathit{Kosten} = H + 1$
- Suchschlüssel ist nicht Kandidatenschlüssel[2]
  - mehrere Datensätze werden zurückgeliefert
  - $\mathit{Kosten} =$
    $(H - 1) + \#\ \mathit{Blattknoten\ mit\ Suchschlüssel} + \#\ \mathit{Ergebnisdatensätze}$
  - kann sehr teuer sein, da jeder Ergebnisdatensatz möglicherweise auf einem anderen Block liegt
  - sequentielles Lesen der gesamten Datei möglicherweise billiger

[2] Annahme: TIDs werden an Suchschlüssel angehängt, um diese im B^+-Baum eindeutig zu machen; die Erweiterung um TIDs ist für Benutzer nicht sichtbar.

Auswertung der Selektion/8

A6 Primärindex auf A + Bereichsanfrage

- $\sigma_{A>V}(R)$: verwende Index um ersten Datensatz $> V$ zu finden, dann sequentielles Lesen
- $\sigma_{A<V}(R)$: lies sequentiell bis erstes Tupel $\geq V$ gefunden; Index wird nicht verwendet

A7 Sekundärindex auf A + Bereichsanfrage

- $\sigma_{A>V}(R)$: finde ersten Datensatz $> V$ mit Index; Index sequentiell lesen um alle Pointer zu den entsprechenden Datensätzen zu finden; Pointer verfolgen und Datensätze holen
- $\sigma_{A<V}(R)$: Blätter des Index sequentiell lesen und Pointer verfolgen bis Suchschlüssel $\geq V$
- Pointer verfolgen braucht im schlimmsten Fall eine Lese-/Schreib-Operation pro Datensatz; sequentielles Lesen der gesamten Datei möglicherweise schneller

Auswertung der Selektion/9

- Pointer verfolgen in Sekundärindex:
  - jeder Datensatz liegt möglicherweise auf einem anderen Block
  - Pointer sind nicht nach Block-Nummern sortiert
  - das führt zu Random-Zugriffen quer durch die Datei
  - derselbe Block wird möglicherweise sogar öfters gelesen
  - falls Anzahl der Ergebnisdatensätze $\geq b$, dann wird im Worst Case jeder Block der Relation gelesen
- Bitmap Index Scan: hilft bei großer Anzahl von Pointern
  - Block i wird durch i-tes Bit in Bit Array der Länge b repräsentiert
  - statt Pointer im Index zu verfolgen, wird nur das Bit des entsprechenden Blocks gesetzt
  - dann werden alle Blöcke gelesen, deren Bit gesetzt ist
  - ermöglicht teilweise sequentielles Lesen
  - gut geeignet, falls Suchschlüssel kein Kandidatenschlüssel ist

Integrierte Übung 3.1

Was ist die beste Auswertungsstrategie für folgende Selektion, wenn es einen B^+-Baum Sekundärindex auf $(BrName, BrCity)$ auf der Relation $Branch(\underline{BrName}, \underline{BrCity}, Assets)$ gibt?

$$\sigma_{BrCity<'Brighton' \ \wedge \ Assets<5000 \ \wedge \ BrName='Downtown'}(Branch)$$

Inhalt

Join Operator/1

- Theta-Join: $r \bowtie_\theta s$
  - für jedes Paar von Tupeln $t_r \in r$, $t_s \in s$ wird Join-Prädikat θ überprüft
  - falls Prädikat erfüllt, ist $t_r \circ t_s$ im Join-Ergebnis
  - Beispiel: Relationen $r(a, b, c)$, $s(d, e, f)$
    Join-Prädikat: $(a < d) \wedge (b = d)$
    Schema des Join-Ergebnisses: (a, b, c, d, e, f)

- Equi-Join: Prädikat enthält "=" als einzigen Operator

- Natürlicher Join: $r \bowtie s$
  - Equi-Join, bei dem alle Attribute gleichgesetzt werden die gleich heißen
  - im Ergebnis kommt jedes Attribut nur einmal vor
  - Beispiel: Relationen $r(a, b, c)$, $s(c, d, e)$
    Natürlicher Join $r \bowtie s$ entspricht θ-Equi-Join $\pi_{a,b,c,d,e}(r \bowtie_{r.c=s.c} s)$
    Schema des Ergebnisses: (a, b, c, d, e)

Join Operator/2

- Join ist kommutativ (bis auf Ordnung der Attribute):

$$r \bowtie s = \pi(s \bowtie r)$$

  - Ordnung der Attribute wird durch (logisches) Vertauschen der Spalten (Projektion π) wiederhergestellt und ist praktisch kostenlos
- Join ist assoziativ:

$$(r \bowtie s) \bowtie t = r \bowtie (s \bowtie t)$$

- Effizienz der Auswertung:
  - vertauschen der Join-Reihenfolge ändert zwar das Join-Ergebnis nicht
  - die Effizienz kann jedoch massiv beeinflusst werden!
- Benennung der Relationen: $r \bowtie s$
  - r die äußere Relation
  - s die innere Relation

Join Selektivität

- Kardinalität: absolute Größe des Join Ergebnisses $r \bowtie_\theta s$

$$|r \bowtie_\theta s|$$

- Selektivität: relative Größe des Join Ergebnisses $r \bowtie_\theta s$

$$sel_\theta = \frac{|r \bowtie_\theta s|}{|r \times s|}$$

- schwache Selektivität: Werte nahe bei 1 (viele Tupel im Ergebnis)
- starke Selektivität: Werte nahe bei 0 (wenig Tupel im Ergebnis)

Integrierte Übung 3.2

Gegeben Relationen $R1(\underline{A}, B, C)$, $R2(\underline{C}, D, E)$, $R3(\underline{E}, F)$, Schlüssel unterstrichen, mit Kardinalitäten $|R1| = 1000$, $|R2| = 1500$, $|R3| = 750$.

- Schätzen Sie die Kardinalität des Joins $R1 \bowtie R2 \bowtie R3$ ab (die Relationen enthalten keine Nullwerte).
- Geben Sie eine Join-Reihenfolge an, welche möglichst kleine Joins erfordert.
- Wie könnte der Join effizient berechnet werden?

Join Operator/3

- Es gibt verschiedene Algorithmen um einen Join auszuwerten:
  - Nested Loop Join
  - Block Nested Loop Join
  - Indexed Nested Loop Join
  - Merge Join
  - Hash Join
- Auswahl aufgrund einer Kostenschätzung.
- Wir verwenden folgende Relationen in den Beispielen:
  - Anleger = (AName, Stadt, Strasse)
    - Anzahl der Datensätze: $n_a = 10'000$
    - Anzahl der Blöcke: $b_a = 400$
  - Konten = (AName, KontoNummer, Kontostand)
    - Anzahl der Datensätze: $n_k = 5'000$
    - Anzahl der Blöcke: $b_k = 100$

Nested Loop Join/1

- Nested Loop Join Algorithms: berechne Theta-Join $r \bowtie_\theta s$

```
for each tuple t_r in r do
  for each tuple t_s in s do
    if (t_r, t_s) erfüllt Join-Bedingung θ then
      gib t_r ∘ t_s aus
  end
end
```

- Immer anwendbar:
  - für jede Art von Join-Bedingung θ anwendbar
  - kein Index erforderlich
- Teuer da jedes Tupel des Kreuzproduktes ausgewertet wird

Nested Loop Join/2

- Ordnung der Join Argumente relevant:
  - r wird 1x gelesen, s wird bis zu n_r mal gelesen
- Worst case: $M = 2$, nur 1 Block von jeder Relation passt in Puffer
  $Kosten = b_r + n_r * b_s$
- Best case: $M > b_s$, innere Relation passt vollständig in Puffer (+1 Block der äußeren Relation)
  $Kosten = b_r + b_s$
- Beispiel:
  - Konten $\bowtie$ Anleger: $M = 2$
    $b_k + n_k * b_a = 100 + 5'000 * 400 = 2'000'100$ Block Zugriffe
  - Anleger $\bowtie$ Konten: $M = 2$
    $b_a + n_a * b_k = 400 + 10'000 * 100 = 1'000'400$ Block Zugriffe
  - Kleinere Relation *(Konten)* passt in Puffer: $M > b_k$
    $b_a + b_k = 400 + 100 = 500$ Block Zugriffe
- Einfacher Nested Loop Algorithms wird nicht verwendet da er nicht Block-basiert arbeitet.

Block Nested Loop Join/1

- Block Nested Loop Join vergleicht jeden Block von r mit jedem Block von s.
- Algorithmus für $r \bowtie_\theta s$

for each Block B_r **of** r **do**
  for each Block B_s **of** s **do**
    for each Tuple t_r **in** B_r **do**
      for each Tuple t_s **in** B_s **do**
        if (t_r, t_s) erfüllt Join-Bedingung θ **then**
          gib $t_r \circ t_s$ aus

Block Nested Loop Join/2

- Worst case: $M = 2$, $Kosten = b_r + b_r * b_s$
  - Jeder Block der inneren Relation s wird für jeden Block der äußeren Relation einmal gelesen (statt für jedes Tupel der äußeren Relation)
- Best case: $M > b_s$, $Kosten = b_r + b_s$

- Beispiel:
  - Konten $\bowtie$ Anleger: $M = 2$
    $b_k + b_k * b_a = 100 + 100 * 400 = 40'100$ Block Zugriffe
  - Anleger $\bowtie$ Konten: $M = 2$
    $b_a + b_a * b_k = 400 + 400 * 100 = 40'400$ Block Zugriffe
  - Kleinere Relation *(Konten)* passt in Puffer: $M > b_k$
    $b_a + b_k = 400 + 100 = 500$ Block Zugriffe

Block Nested Loop Join/3

- Zick-Zack Modus: $R \bowtie_\theta S$
  - reserviere $M - k$ Blöcke für R und k Blöcke für S
  - innere Relation wird abwechselnd vorwärts und rückwärts durchlaufen
  - dadurch sind die letzten k Seiten schon im Puffer (LRU Puffer Strategie) und müssen nicht erneut gelesen werden
- Kosten: $k \leq b_s, 0 < k < M$

$$b_r + k + \lceil b_r/(M-k) \rceil (b_s - k)$$

  - r muss einmal vollständig gelesen werden
  - innere Schleife wird $\lceil b_r/(M-k) \rceil$ mal durchlaufen
  - erster Durchlauf erfordert b_s Block Zugriffe
  - jeder weitere Durchlauf erfordert $b_s - k$ Block Zugriffe
- Optimale Ausnutzung des Puffers:
  - $b_r \leq b_s$: kleinere Relation außen (Heuristik)
  - $k = 1$: $M - 1$ Blöcke für äußere Relation, 1 Block für innere

Integrierte Übung 3.3

Berechne die Anzahl der Block Zugriffe für folgende Join Alternativen, jeweils mit Block Nested Loop Join, Puffergröße $M = 20$.
Konto: $n_k = 5'000$, $b_k = 100$. Anleger: $n_a = 10'000$, $b_a = 400$

- Konto $\bowtie$ Anleger, $k = 19$
- Konto $\bowtie$ Anleger, $k = 10$
- Konto $\bowtie$ Anleger, $k = 1$
- Anleger $\bowtie$ Konto, $k = 1$

Indexed Nested Loop Join/1

- Index Suche kann Scannen der inneren Relation ersetzen
  - auf innerer Relation muss Index verfügbar sein
  - Index muss für Join-Prädikat geeignet sein (z.B. Equi-Join)
- Algorithmus: Für jedes Tupel t_r der äußeren Relation r verwende den Index um die Tupel der inneren Relation zu finden, welche die Bedingung θ erfüllen.
- Worst case: für jedes Tupel der äußeren Relation wird eine Index Suche auf die innere Relation gemacht.
  $Kosten = b_r + n_r * c$
  - c sind die Kosten, den Index zu durchlaufen und alle passenden Datensätze aus der Relation s zu lesen
  - c kann durch die Kosten einer einzelnen Selektion mithilfe des Index abgeschätzt werden
- Index auf beiden Relationen: kleinere Relation außen

Indexed Nested Loop Join/2

- Beispiel: Berechne Konten $\bowtie$ Anleger (Konten als äußere Relation), B^+-Baum mit $m = 20$ auf Relation Anleger.

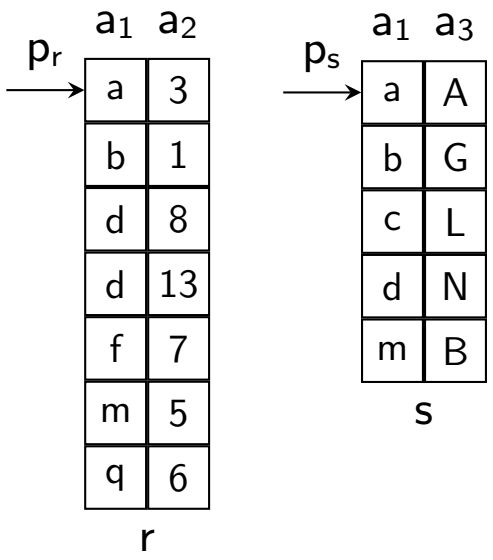
- Lösung:
  - Anleger hat $n_a = 10'000$ Datensätze.
  - B^+-Baum hat maximal $L = \lceil n_a / \lceil (m-1)/2 \rceil \rceil = 1000$ Blätter (Blattknoten minimal befüllt)
  - Kosten für 1 Datensatz von Relation Anleger mit Index lesen:

$$c = \lceil \log_{\lceil m/2 \rceil}(L) \rceil + 2 = \lceil \log_{10}(1'000) \rceil + 2 = 5$$

    $\rightarrow$ B^+-Baum durchlaufen: maximale Pfadlänge + 1
    $\rightarrow$ 1 Zugriff auf Datensatz (Schlüssel)
  - Konten hat $n_k = 5'000$ Datensätze und $b_k = 100$ Blöcke.
  - Indexed Nested Loop Join:
    Kosten $= b_k + n_k * c = 100 + 5'000 * 5 = 25'100$ Blockzugriffe

Merge Join/1

- Merge Join: Verwende zwei Pointer *pr* und *ps* die zu Beginn auf den ersten Datensatz der sortierten Relationen *r* bzw. *s* zeigen und bewege die Zeiger synchron, ähnlich wie beim Mischen, nach unten.
- Algorithmus: $r \bowtie s$ (Annahme: keine Duplikate in Join-Attributen)
  1. sortiere Relationen nach Join-Attributen (falls nicht schon richtig sortiert)
  2. starte mit Pointern bei jeweils 1. Tupel
  3. aktuelles Tupel-Paar ausgeben falls es Join-Bedingung erfüllen
  4. bewege den Pointer der Relation mit dem kleineren Wert; falls die Werte gleich sind, bewege den Pointer der äußeren Relation



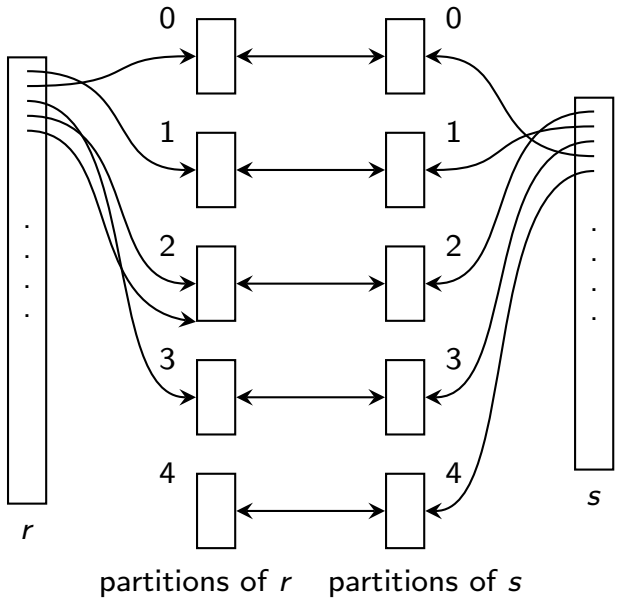
- Duplikate in den Join-Attributen: bei gleichen Werten muss jede Kopie der äußeren mit jeder Kopie der inneren Relation gepaart werden

Merge Join/2

- Anwendbar nur für Equi- und Natürliche Joins
- Kosten: Falls alle Tupel zu einem bestimmten Join-Wert im Puffer Platz haben:
  - r und s 1x sequentiell lesen
  - $Kosten = b_r + b_s$ (+ Sortierkosten, falls Relationen noch nicht sortiert)
- Andernfalls muss ein Block Nested Loop Join zwischen den Tupeln mit identischen Werten in den Join-Attributen gemacht werden.
- Sort-Merge Join: Falls Relationen noch nicht sortiert sind, muss zuerst sortiert werden.

Hash Join/1

- Nur für Equi- und Natürliche Joins.
- Partitioniere Tupel von r und s mit derselben Hash Funktion h, welche die Join-Attribute ($JoinAttrs$) auf die Menge $\{0, 1, \ldots, n\}$ abbildet.
- Alle Tupel einer Relation mit demselben Hash-Wert bilden eine Partition (=Bucket):
  - Partition r_i enthält alle Tupel $t_r \in r$ mit $h(t_r[JoinAttrs]) = i$
  - Partition s_i enthält alle Tupel $t_s \in s$ mit $h(t_s[JoinAttrs]) = i$



- Partitionsweise joinen: Tupel in r_i brauchen nur mit Tupel in s_i verglichen werden
  - ein r-Tupel und ein s-Tupel welche die Join-Kondition erfüllen, haben denselben Hash-Wert i und werden in die Partitionen r_i bzw. s_i gelegt

Hash Join/2

- Algorithmus für Hash Join $r \bowtie s$.
  1. Partitioniere r und s mit derselben Hash Funktion h; jede Partition wird zusammenhängend auf die Platte geschrieben
  2. Für jedes Paar (r_i, s_i) von Partitionen:
     a. build: lade s_i in den Hauptspeicher und baue einen Hauptspeicher-Hash-Index mit neuer Hash-Funktion $h' \neq h$.
     b. probe: für jedes Tupel $t_r \in r_i$ suche zugehörige Join-Tupel $t_s \in s_i$ mit Hauptspeicher-Hash-Index.
- Relation s wird Build Input genannt; r wird Probe Input genannt.
- Kleinere Relation (in Anzahl der Blöcke) wird als Build Input verwendet, damit weniger Partitionen benötigt werden.
  - Hash-Index für jede Partition des Build Input muss in Hauptspeicher passen ($M - 1$ Blöcke für Puffergröße M)
  - von Probe Input brauchen wir jeweils nur 1 Block im Speicher

Hash Join/3

- Kosten für Hash Join:
  - Partitionieren der beiden Relationen: $2 * (b_r + b_s)$
    - $\rightarrow$ jeweils gesamte Relation einlesen und zurück auf Platte schreiben
  - Build- und Probe-Phase lesen jede Relation genau einmal: $b_r + b_s$
  - $Kosten = 3 * (b_r + b_s)$
  - Kosten von nur teilweise beschriebenen Partitionen werden nicht berücksichtigt.

Hash Join/4

Beispiel: $Konto \bowtie Anleger$ soll als Hash Join berechnet werden.
Puffergröße $M = 20$ Blöcke, $b_k = 100$, $b_a = 400$.

- Welche Relation wird als Build Input verwendet?
  Konto, da kleiner ($b_k < b_a$)
- Wieviele Partitionen müssen gebildet werden?
  $\lceil \frac{b_k}{M-1} \rceil = 6$ Partitionen, damit Partitionen von Build Input in Puffer ($M - 1 = 19$) passen. Partitionen von Probe Input müssen nicht in Puffer passen: es wird nur je ein Block eingelesen.
- Wie groß sind die Partitionen?
  Build Input: $\lceil 100/6 \rceil = 17$, Probe Input: $\lceil 400/6 \rceil = 67$
- Kosten für Join?
  $3(b_k + b_a) = 1'500$ laut Formel. Da wir aber nur ganze Blöcke schreiben können, sind die realen Kosten etwas höher:
  $b_k + b_a + 2 * (6 * 17 + 6 * 67) = 1'508$

Rekursives Partitionieren

- Eine Relation kann höchstens in $M - 1$ Partitionen zerlegt werden:
  - 1 Input-Block
  - $M - 1$ Output Blocks (1 Block pro Partition)
- Partitionen der Build-Relation (b Blöcke) müssen in Speicher passen
  - Build-Partition darf maximal $M - 1$ Blöcke haben
    $\Rightarrow$ Anzahl der Partitionen mindestens $\lceil \frac{b}{M-1} \rceil$
- Build-Relation könnte zu groß für maximale Partitionen-Anzahl sein:
  - falls $\lceil \frac{b}{M-1} \rceil > M - 1$ können nicht genug Partitionen erzeugt werden
- Rekursives Partitionieren:
  - erzeuge $M - 1$ Partitionen (r_i, s_i), $1 \leq i < M$
  - partitioniere jedes Paar (r_i, s_i) rekursiv (mit einer neuen Hash-Funktion), bis Build-Partition in Hauptspeicher passt
  - (r_i, s_i) wird also behandelt wie zwei Relationen

Overflows/1

- Overflow: Build Partition passt nicht in den Hauptspeicher
  - kann auch vorkommen, wenn es sich von der Größe der Build-Relation her ausgehen müsste (d.h. $\lceil \frac{b}{M-1} \rceil \leq M-1$)
- Overflows entstehen durch verschieden große Partitionen:
  - einige Werte kommen viel häufiger vor oder
  - die Hashfunktion ist nicht uniform und random
- Fudge Factor:
  - etwas mehr als $\lceil \frac{b}{M-1} \rceil$ Partitionen (z.B. 20% mehr) werden angelegt
  - dadurch werden kleine Unterschiede in der Partitionsgröße abgefedert
  - hilft nur bis zu einem gewissen Grad
- Lösungsansätze
  - Overflow Resolution
  - Overflow Avoidance

Overflows/2

- Overflow Resolution: während der Build-Phase
  - falls Build-Partition s_i zu groß: partitioniere Probe- und Build-Partition (r_i, s_i) erneut bis Build-Partition in Speicher passt
  - für erneutes Partitionieren muss neue Hashfunktion verwendet werden
  - selbe Technik wie rekursives Partitionieren
    (es wird jedoch aufgrund unterschiedlicher Partitionsgrößen neu partitioniert, nicht wegen der Größe der Build-Relation)
- Overflow Avoidance: während des Partitionierens
  - viele kleine Partitionen werden erzeugt
  - während der Build-Phase werden so viele Partitionen wie möglich in den Hauptspeicher geladen
  - die entsprechenden Partitionen in der anderen Relation werden für das Probing verwendet
- Wenn alle Stricke reißen...
  - wenn einzelne Werte sehr häufig vorkommen versagen beide Ansätze
  - Lösung: Block-Nested Loop Join zwischen Probe- und Build-Partition

Zusammenfassung

- Nested Loop Joins:
  - Naive NL: ignoriert Blöcke
  - Block NL: berücksichtigt Blöcke
  - Index NL: erfordert Index auf innere Relation
- Equi-Join Algorithmen:
  - Merge-Join: erfordert sortierte Relationen
  - Hash-Join: keine Voraussetzung

Datenbanken 2

Anfrageoptimierung

Nikolaus Augsten

nikolaus.augsten@sbg.ac.at
FB Computerwissenschaften
Universität Salzburg

http://dbresearch.uni-salzburg.at

WS 2018/19

Version 16. Januar 2019

Inhalt

Literatur und Quellen

Lektüre zum Thema "Anfrageoptimierung":

- Kapitel 8 aus Kemper und Eickler: Datenbanksysteme: Eine Einführung. Oldenbourg Verlag, 2013.
- Chapter 13 in Silberschatz, Korth, and Sudarashan: Database System Concepts. McGraw Hill, 2011.

Danksagung Die Vorlage zu diesen Folien wurde entwickelt von:

- Michael Böhlen, Universität Zürich, Schweiz
- Johann Gamper, Freie Universität Bozen, Italien

Inhalt

Schritte der Anfragebearbeitung

1. Parser
   - input: SQL Anfrage
   - output: Relationaler Algebra Ausdruck
2. Optimierer
   - input: Relationaler Algebra Ausdruck
   - output: Auswertungsplan
3. Execution Engine
   - input: Auswertungsplan
   - output: Ergebnis der SQL Anfrage

1. Parser

Parser:

- Input: SQL Anfrage vom Benutzer
  Beispiel:
  ```
  SELECT DISTINCT balance
  FROM account
  WHERE balance < 2500
  ```
- Output: Relationaler Algebra Ausdruck
  Beispiel: $\sigma_{balance<2500}(\pi_{balance}(account))$
- Algebra Ausdruck nicht eindeutig!
  Beispiel: folgende Ausdrück sind äquivalent
  - $\sigma_{balance<2500}(\pi_{balance}(account))$
  - $\pi_{balance}(\sigma_{balance<2500}(account))$
- Kanonische Übersetzung führt zu algebraischer Normalform (eindeutig)

Parser: Kanonische Übersetzung

- SQL Anfrage: `SELECT DISTINCT` $A_1, A_2, \ldots, A_n$
  `FROM` $R_1, R_2, \ldots, R_k$
  `WHERE` θ
- Algebraische Normalform:

$$\pi_{A_1,A_2,\ldots,A_n}(\sigma_\theta(R_1 \times R_2 \times \ldots \times R_k))$$

- Prädikat θ kann sowohl Selektions- als auch Join-Bedingungen enthalten

2. Optimierer

Optimierer:

- Input: Relationaler Algebra Ausdruck
  Beispiel: $\pi_{balance}(\sigma_{balance<2500}(account))$
- Output: Auswertungsplan
  Beispiel:

$\pi_{balance}$

| pipeline

$\sigma_{balance<2500}$

use index 1

|

account

- Auswertungsplan wird in drei Schritten konstruiert:
  A) Logische Optimierung: Äquivalenzumformungen
  B) Physische Optimierung: Annotation der relationalen Algebra Operatoren mit physischen Operatoren
  C) Kostenabschätzung für verschiedene Auswertungspläne

A) Logische Optimierung: Äquivalenzumformungen

- Äquivalenz relationaler Algebra Ausdrücke:
  - äquivalent: zwei Ausdrücke erzeugen dieselbe Menge von Tupeln auf allen legalen Datenbankinstanzen
  - legal: Datenbankinstanz erfüllt alle Integritätsbedingungen des Schemas
- Äquivalenzregeln:
  - umformen eines relationalen Ausdrucks in einen äquivalenten Ausdruck
  - analog zur Algebra auf reelle Zahlen, z.B.:
    $a + b = b + a$, $a(b + c) = ab + ac$, etc.
- Warum äquivalente Ausdrücke erzeugen?
  - äquivalente Ausdrücke erzeugen dasselbe Ergebnis
  - jedoch die Ausführungszeit unterscheidet sich signifikant

Äquivalenzregeln – Beispiele

- Selektionen sind untereinander vertauschbar:
  $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
  - E relationaler Ausdruck (im einfachsten Fall eine Relation)
  - θ_1 und θ_2 sind Prädikate auf die Attribute von E z.B. $E.salary < 2500$
  - σ_θ ergibt alle Tupel welche die Bedingung θ erfüllen

- Natürlicher Join ist assoziativ: $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  - das Join Prädikat im natürlichen Join ist "Gleichheit" auf allen Attributen zweier Ausdrücke mit gleichem Namen

    Beispiel: $R[A, B], S[B, C]$, Prädikat ist $R.B = S.B$
  - falls zwei Ausdrücke keine gemeinsamen Attribute haben, wird der natürliche Join zum Kreuzprodukt

    Beispiel: $R[A, B], S[C, D]$, $R \bowtie S = R \times S$

Äquivalenzregeln – Beispiel Anfrage

- Schemas der Beispieltabellen:

  branch(branch-name, branch-city, assets)
  account(account-number, branch-name, balance)
  depositor(customer-name, account-number)

- Fremdschlüsselbeziehungen:

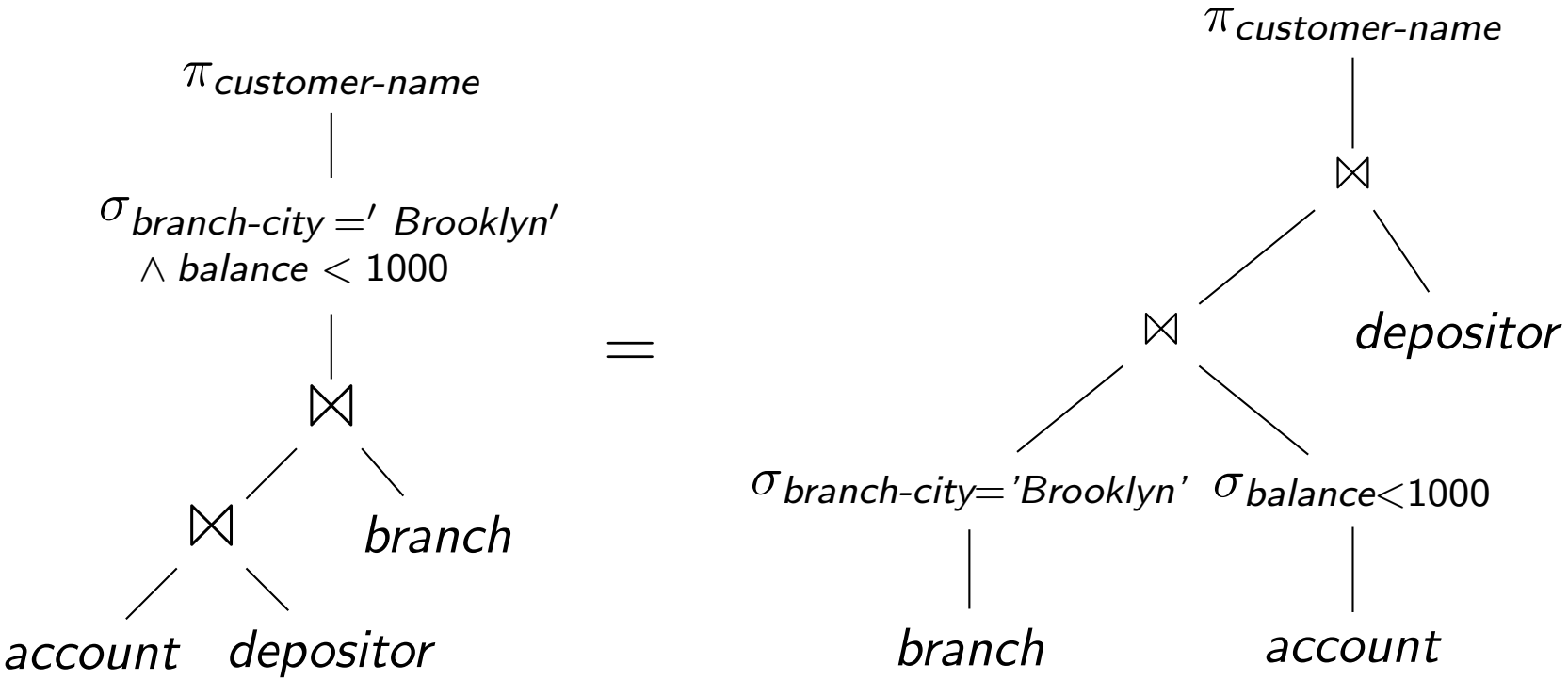
  $\pi_{\textit{branch-name}}(\textit{account}) \subseteq \pi_{\textit{branch-name}}(\textit{branch})$
  $\pi_{\textit{account-number}}(\textit{depositor}) \subseteq \pi_{\textit{account-number}}(\textit{account})$

- Anfrage:

```
SELECT customer-name
FROM branch, account, depositor
WHERE branch-city='Brooklyn' AND
   balance < 1000 AND
   branch.branch-name = account.branch-name AND
   account.account-number = depositor.account-number
```

Äquivalenzregeln – Beispiel Anfrage

- Äquivalente relationale Algebra Ausdrücke (als Operatorbäume dargestellt):

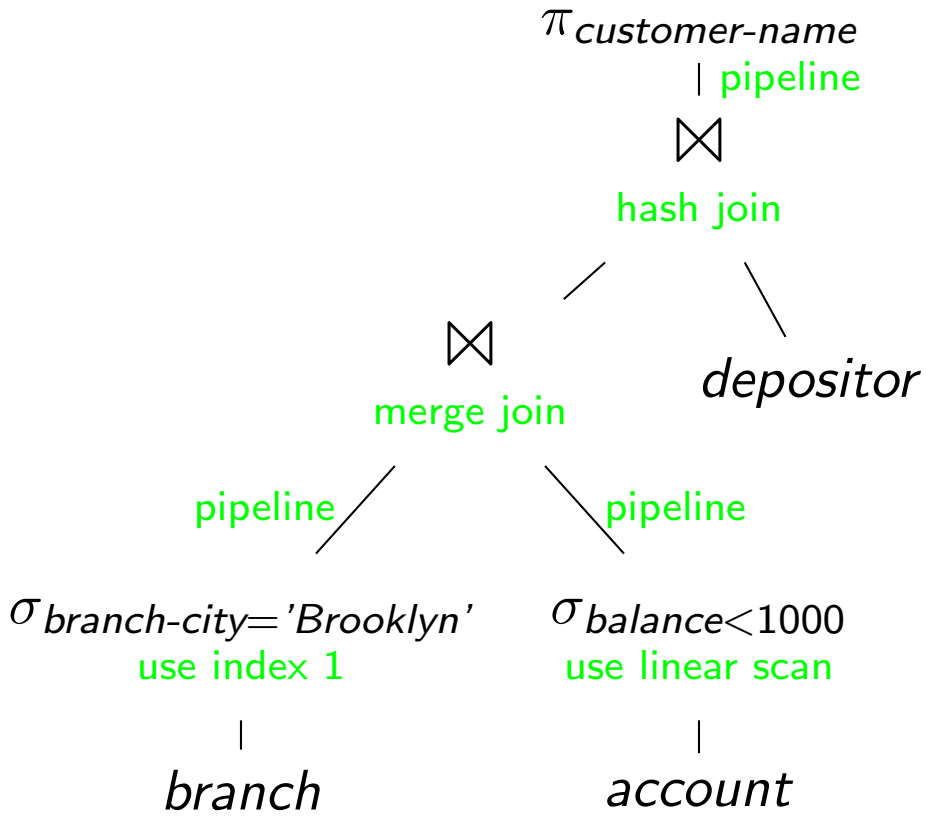


B) Annotation der relationalen Algebra Ausdrücke

- Ein Algebraausdruck ist noch kein Ausführungsplan.
- Zusätzliche Entscheidungen müssen getroffen werden:
  - welche Indizes sollen verwendet werden, z.B. für Selektion oder Join?
  - welche Algorithmen sollen verwendet werde, z.B. Nested-Loop oder Hash Join?
  - sollen Zwischenergebnisse materialisiert oder "pipelined" werden?
  - usw.
- Für jeden Algebra Ausdruck können mehrere Ausführungspläne erzeugt werden.
- Alle Pläne ergeben dieselbe Relation, unterscheiden sich jedoch in der Ausführungszeit.

Beispiel: Ausführungsplan

- Ausführungsplan für die vorige Beispielanfrage:
  - *account* ist physisch sortiert nach *branch-name*
  - index 1 ist ein B^+-Baum Index auf *(branch-city,branch-name)*



C) Kostenabschätzung

- Welches ist der beste (=schnellste) Ausführungsplan?
- Schwieriges Problem:
  - Kosten für Ausführungsplan können nur abgeschätzt werden
  - es gibt eine sehr große Zahl von möglichen Ausführungsplänen

Datenbankstatistik für Kostenabschätzung

- Katalog: Datenbanksystem pflegt Statistiken über Daten
- Beispiel Statistiken:
  - Anzahl der Tupel pro Relation
  - Anzahl der Blöcke pro Relation
  - Anzahl der unterschiedlichen Werte für ein Attribut
  - Histogramm der Attributwerte
- Statistik wird verwendet um Kosten von Operationen abzuschätzen, z.B.:
  - Kardinalität des Ergebnisses einer Selektion
  - Kosten für Nested-Loop vs. Hash-Join
  - Kosten für sequentielles Lesen der Tabelle vs. Zugriff mit Index
- Beachte: Statistik wird nicht nach jeder Änderung aktualisiert und ist deshalb möglicherweise nicht aktuell

3. Execution Engine

Die Execution Engine

- erhält den Ausführungsplan vom Optimierer
- führt den Plan aus, indem die entsprechenden Algorithmen aufgerufen werden
- liefert das Ergebnis an den Benutzer zurück

Materialisierung und Pipelining

- Materialisierung:
  - gesamter Output eines Operators (Zwischenergebnis) wird gespeichert (z.B. auf Platte)
  - nächster Operator liest Zwischenergebnis und verarbeitet es weiter
- Pipelining:
  - sobald ein Tupel erzeugt wird, wird es an den nächsten Operator weitergeleitet
  - kein Zwischenspeichern erforderlich
  - Benutzer sieht erste Ergebnisse, bevor gesamte Anfrage berechnet ist
- Blocking vs. Non-Blocking:
  - Blocking: Operator muss gesamten Input lesen, bevor erstes Output Tupel erzeugt werden kann
  - Non-Blocking: Operator liefert erstes Tupel zurück sobald ein kleiner Teil des Inputs gelesen ist

Integrierte Übung 4.1

- Welche der folgenden Operatoren sind “blocking” bzw. “non-blocking”?
  - Selektion
  - Projektion
  - Sortierung
  - Gruppierung+Aggregation
  - Block Nested-Loop Join
  - Index Nested-Loop Join
  - Hash Join
  - Merge Join, Sort-Merge Join

Iteratoren

- Demand-driven vs. Producer-driven Pipeline:
  - Demand-driven: Operator erzeugt Tupel erst wenn von Eltern-Knoten angefordert; Auswertung beginnt bei Wurzelknoten
  - Producer-driven: Operatoren produzieren Tupel und speichern sie in einen Buffer; Eltern-Knoten bedient sich aus Buffer (Producer-Consumer Modell)
- Demand-driven Pipelining: relationale Operatoren werden oft als Iteratoren mit folgenden Funktionen implementiert:
  - open(): initialisiert den Operator
    z.B. Table Scan: Datei öffnen und Cursor auf ersten Datensatz setzen
  - next(): liefert nächstes Tupel
    z.B. Table Scan: Tupel an Cursorposition lesen und Cursor weitersetzten
  - close(): abschließen
    z.B. Table Scan: Datei schließen
- Im Iteratormodell fragt der Wurzelknoten seine Kinder so lange nach Tupeln, bis keine Tupel mehr geliefert werden.

Inhalt

Überblick

- nur eine Auswahl von Äquivalenzregeln (equivalence rules, ER) wird präsentiert
- die Auswahl ist nicht minimal, d.h., einige der Regeln können aus anderen hergeleitet werden
- Notation:
  - $E, E_1, E_2 \ldots$ sind relationale Algebra Ausdrücke
  - $\theta, \theta_1, \theta_2 \ldots$ sind Prädikate (z.B. $A < B \wedge C = D$)

Definition von relationalen Algebra Ausdrücken

- Ein elementarer Ausdruck der relationalen Algebra ist
  - eine Relation in der Datenbank (z.B. Konten)
- Zusammengesetzte Ausdrücke: Falls E_1 und E_2 relationale Algebra Ausdrücke sind, dann lassen sich durch relationale Operatoren weitere Ausdrücke bilden, z.B.:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_\theta(E_1)$, θ ist ein Prädikat in E_1
  - $\pi_A(E_1)$, A ist eine Liste von Attributen aus E_1
- Geschlossenheit der relationalen Algebra: elementare und zusammengesetzte Ausdrücke können gleich behandelt werden

Äquivalenzregeln/1

Selektion und Projektion:

- ER1 Konjunktive Selektionsprädikate können in mehrere Selektionen aufgebrochen werden:

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

- ER2 Selektionen sind untereinander vertauschbar:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

- ER3 Geschachtelte Projektionen können eliminiert werden:

$$\pi_{A_1}(\pi_{A_2}(\ldots(\pi_{A_n}(E))\ldots)) = \pi_{A_1}(E)$$

(Ai sind Listen von Attributen)

- ER4 Selektion kann mit Kreuzprodukt und θ-Join kombiniert werden:

(a) $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
(b) $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Äquivalenzregeln/2

Kommutativität und Assoziativität von Joins:

- ER5 Theta-Join und natürlicher Join sind kommutativ:

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

$$E_1 \bowtie E_2 = E_2 \bowtie E_1$$

- ER6 Joins und Kreuzprodukte sind assoziativ:
  - (a) Natürliche Joins sind assoziativ:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

  - (b) Theta-Joins sind assoziativ:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

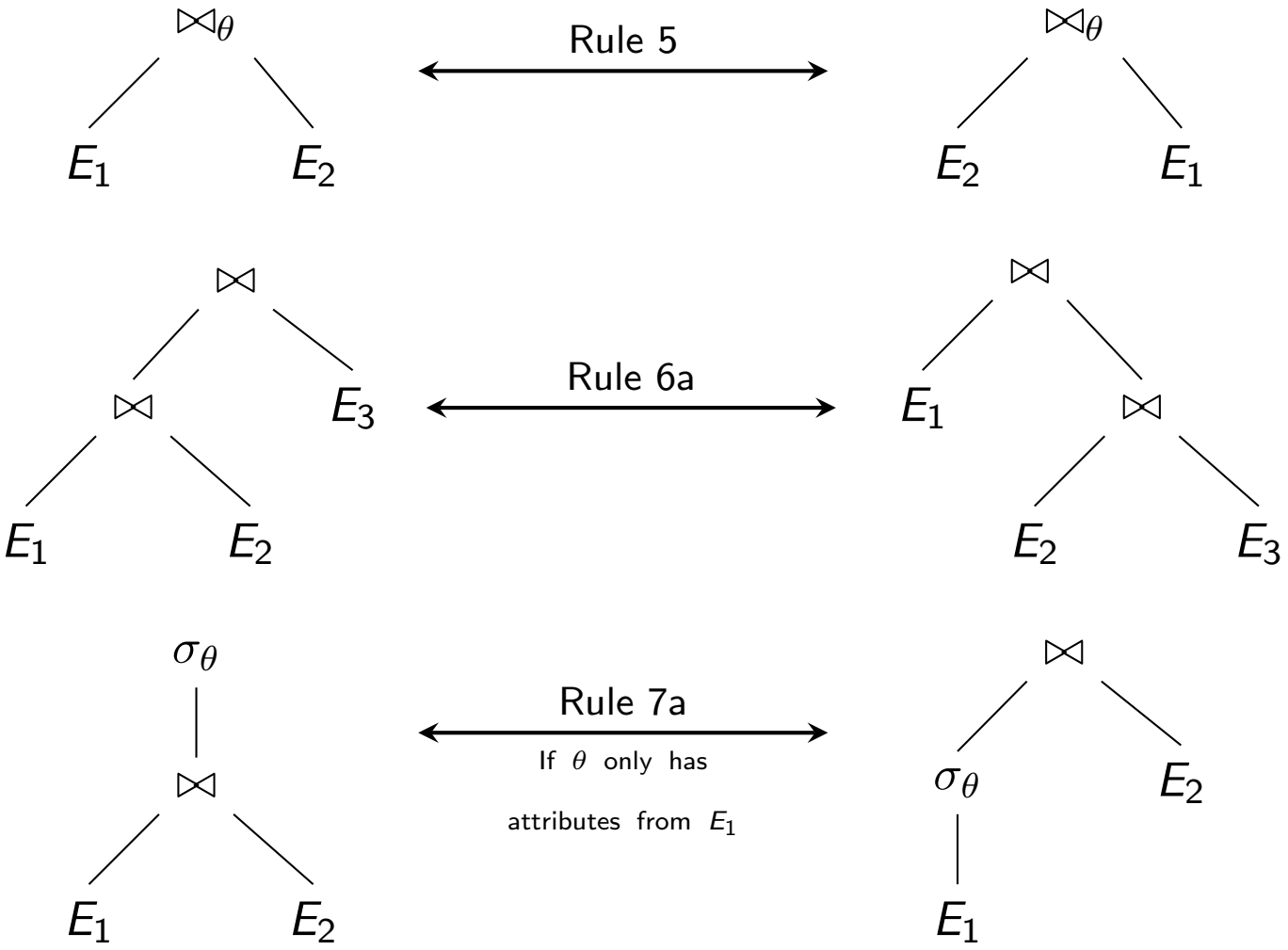
    (θ_2 enthält nur Attribute von E_2 und E_3)
  - (c) Jedes Prädikat θ_i im Theta-Join kann leer sein, also sind auch Kreuzprodukte assoziativ.

Äquivalenzregeln/3

- ER7 Selektion kann bedingt an Join vorbeigeschoben werden:
  a. θ_1 enthält nur Attribute eines Ausdrucks (E1):
  $$\sigma_{\theta_1}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie_\theta E_2$$
  b. θ_1 enthält nur Attribute von E_1 und θ_2 enthält nur Attribute von E_2:
  $$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie_\theta \sigma_{\theta_2}(E_2)$$

Beispiel: Äquivalenzregeln

- Darstellung einiger Äquivalenzregeln als Operatorbaum



Äquivalenzregeln/4

- ER8 Projektion kann an Join und Selektion vorbeigeschoben werden:
  - A_1 und A_2 sind jeweils Projektions-Attribute von E_1 und E_2.

(a) Join: θ enthält nur Attribute aus $A_1 \cup A_2$:

$$\pi_{A_1 \cup A_2}(E_1 \bowtie_\theta E_2) = \pi_{A_1}(E_1) \bowtie_\theta \pi_{A_2}(E_2)$$

(b) Join: θ enthält Attribute die nicht in $A_1 \cup A_2$ vorkommen:

- A_3 sind Attribute von E_1 die in θ vorkommen, aber nicht in $A_1 \cup A_2$
- A_4 sind Attribute von E_2 die in θ vorkommen, aber nicht in $A_1 \cup A_2$

$$\pi_{A_1 \cup A_2}(E_1 \bowtie_\theta E_2) = \pi_{A_1 \cup A_2}(\pi_{A_1 \cup A_3}(E_1) \bowtie_\theta \pi_{A_2 \cup A_4}(E_2))$$

(c) Selektion: θ enthält nur Attribute aus A_1:

$$\pi_{A_1}(\sigma_\theta(E_1)) = \sigma_\theta(\pi_{A_1}(E_1))$$

(d) Selektion: θ enthält Attribute A_3 die nicht in A_1 vorkommen:

$$\pi_{A_1}(\sigma_\theta(E_1)) = \pi_{A_1}(\sigma_\theta(\pi_{A_1 \cup A_3}(E_1)))$$

Äquivalenzregeln/5

Mengenoperationen:

- ER9 Vereinigung und Schnittmenge sind kommutativ:

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

- ER10 Vereinigung und Schnittmenge sind assoziativ.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

Äquivalenzregeln/6

- ER11 Selektion kann an $\cup, \cap$ und $-$ vorbeigeschoben werden:

$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$$
$$\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$$
$$\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$$

  Für $\cap$ und $-$ gilt außerdem:

$$\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap E_2$$
$$\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - E_2$$

- ER12 Projektion kann an Vereinigung vorbeigeschoben werden:

$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

Integrierte Übung 4.2

Stellen Sie die folgenden relationalen Algebra Ausdrücke als Operatorbäume dar:

- $RA1 = \pi_A(R1) \cup \sigma_{X>5}(R2)$
- $RA2 = \pi_A(R1 \bowtie \sigma_{X=Y}(R2 \bowtie \pi_{B,C}(R3 - R4) \bowtie R5))$
  (relationale Operatoren sind linksassoziativ)

Integrierte Übung 4.3

Folgende Äquivalenzregeln sind falsch. Zeigen Sie dies durch ein Gegenbeispiel:

1. $\pi_A(R - S) = \pi_A(R) - \pi_A(S)$
2. $R - S = S - R$
3. $(R - S) - T = R - (S - T)$
4. $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup E_2$

Inhalt

Aufzählung Äquivalenter Ausdrücke

- Optimierer verwenden die Äquivalenzregeln um systematisch äquivalente Ausdrücke zu erzeugen.

- Aufzählung aller äquivalenten Ausdrücke von E:
  $X = \{E\}$ (X ist die Menge aller äquivalenten Ausdrücke)
  repeat
    for each $E_i \in X$:
      wende alle möglichen Äquivalenzumformungen an
      speichere erhaltene Ausdrücke in X
  until keine weiteren Ausdrücke gefunden werden können

- Sehr zeit- und speicherintensiver Ansatz.

Effiziente Aufzählungstechniken

- Speicher sparen: Ausdrücke teilen sich gemeinsame Teilausdrücke:
  - Wenn $E2$ aus $E1$ durch eine Äquivalenzumformung entsteht, bleiben die tieferliegenden Teilbäume gleich und brauchen nicht doppelt abgelegt zu werden.
- Zeit sparen: Aufgrund von Kostenabschätzungen werden einige Ausdrücke gar nie erzeugt.
  - Wenn für einen Teilausdruck E' ein äquivalenter Teilausdruck E'' gefunden wird, der schneller ist, brauchen keine Ausdrücke die E' enthalten berücksichtigt werden.
- Heuristik: Wende Heuristiken an um viel versprechende Ausdrücke zu erzeugen:
  - Selektionen möglichst weit nach unten
  - Projektionen möglichst weit nach unten
  - Joins mit kleinem zu erwartenden Ergebnis zuerst berechnen

Heuristische Optimierung/1

- Heuristische Optimierung transformiert den Operatorbaum nach einer Reihe von Heuristiken, welche die Ausführung normalerweise (jedoch nicht in allen Fällen) beschleunigen.
- Ziel der Heuristiken: Größe der Zwischenergebnisse so früh als möglich (d.h. nahe an den Blättern des Operatorbaums) klein machen.
- Einige (alte) Systeme verwenden nur heuristische Optimierung.
- Modern Systeme kombinieren Heuristiken (nur einige Ausdrücke werden betrachtet) mit kostenbasierter Optimierung (schätze die Kosten für jeden betrachteten Ausdruck ab).

Heuristische Optimierung/2

- Typischer Ansatz der heuristischen Optimierung:
  1. Transformiere alle konjunktiven Selektionen in eine Reihe verschachtelter Selektionen (ER1).
  2. Schiebe Selektionen so weit als möglich im Operatorbaum nach unten (ER2, ER7(a), ER7(b), ER11).
  3. Ersetze Kreuzprodukte, welche von einer Selektion gefolgt sind, durch Joins (ER4(a)).
  4. Führe Joins und Selektionen mit starker Selektivität zuerst aus (ER6).
  5. Schiebe Projektionen so weit nach unten als möglich und erzeuge neue Projektionen, sodass kein Attribut weitergeleitet wird, das nicht mehr gebraucht wird (ER3, ER8, ER12).
  6. Identifiziere die Teilbäume, für die Pipelining möglich ist, und führe diese mit Pipelining aus.

Äquivalenzumformung: Beispieltabellen

- Schemas der Beispieltabellen:

  branch(<u>*branch-name*</u>*, branch-city, assets)*
  account(<u>*account-number*</u>*, branch-name, balance)*
  depositor(<u>*customer-name*</u>*, account-number)*

- Fremdschlüsselbeziehungen:

  $\pi_{\textit{branch-name}}(\textit{account}) \subseteq \pi_{\textit{branch-name}}(\textit{branch})$
  $\pi_{\textit{account-number}}(\textit{depositor}) \subseteq \pi_{\textit{account-number}}(\textit{account})$

Beispiele Äquivalenzumformungen/1

- Beispiel 1: Selektion nach unten schieben.
- Anfrage: Finde die Namen aller Kunden die ein Konto in einer Filiale in Brooklyn haben.

  $\pi_{\textit{customer-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch} \bowtie (\textit{account} \bowtie \textit{depositor})))$
- Der Join wird für die Konten und Kunden aller Filialen berechnet, obwohl wir nur an den Filialen in Brooklyn interessiert sind.
- Umformung unter Verwendung von ER7(a):

  $\pi_{\textit{customer-name}}$
  $(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch}) \bowtie (\textit{account} \bowtie \textit{depositor}))$
- Die Selektion wird vorgezogen, damit sich die Größe der Relationen, auf die ein Join berechnet werden muss, reduziert.

Beispiele Äquivalenzumformungen/2

- Beispiel 2: Oft sind mehrere Umformungen notwendig.
- Anfrage: Finde die Namen aller Kunden mit einem Konto in Brooklyn, deren Kontostand kleiner als 1000 ist.

$$\pi_{\textit{customer-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'\wedge\textit{balance}<1000}(\textit{branch}\bowtie(\textit{account}\bowtie\textit{depositor})))$$

- Umformung 1: ER6(a) (Join Assoziativität):

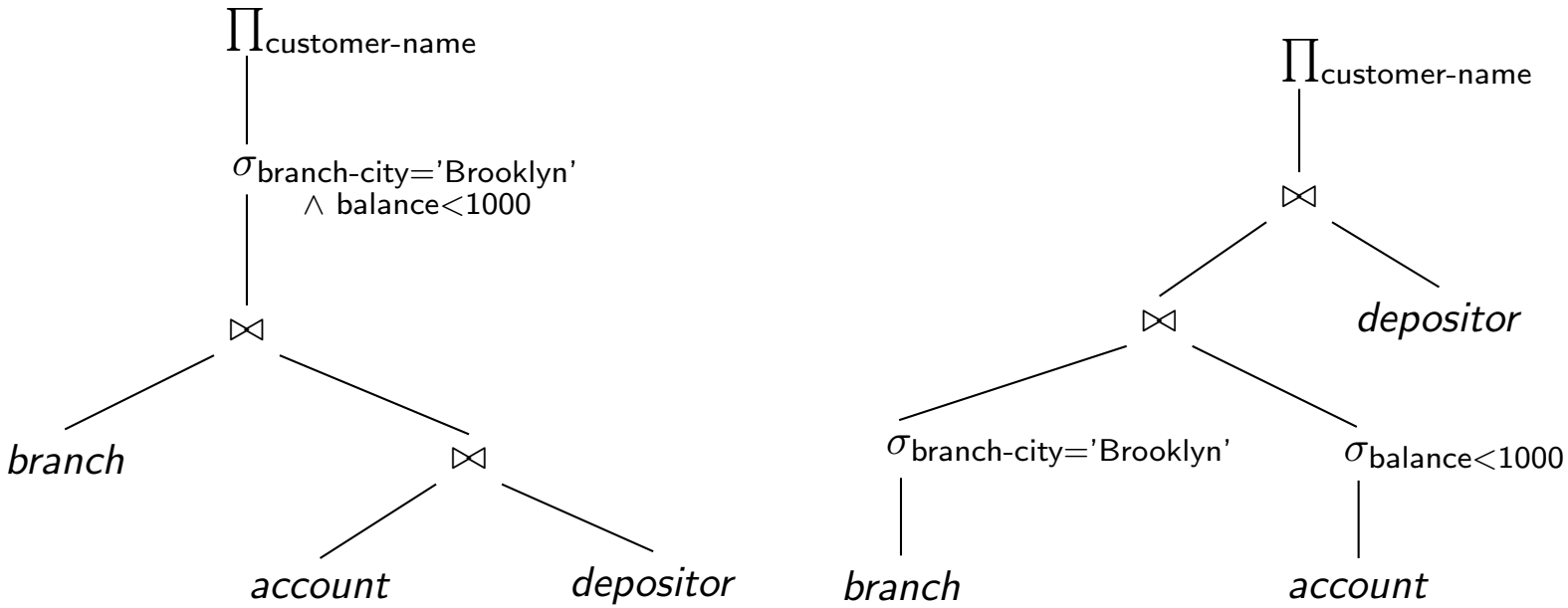
$$\pi_{\textit{customer-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'\wedge\textit{balance}<1000}((\textit{branch}\bowtie\textit{account})\bowtie\textit{depositor}))$$

- Umformung 2: ER7(a) und (b) (Selektion nach unten schieben)

$$\pi_{\textit{customer-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch})\bowtie\sigma_{\textit{balance}<1000}(\textit{account})\bowtie\textit{depositor})$$

Beispiele Äquivalenzumformungen/3

- Beispiel 2 (Fortsetzung)
  - Operatorbaum vor und nach den Umformungen.



Beispiele Äquivalenzumformungen/4

- Beispiel 3: Projektion
- Anfrage: (wie Beispiel 1)

  $\pi_{\textit{customer-name}}((\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch}) \bowtie \textit{account}) \bowtie \textit{depositor})$
- Join $\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch}) \bowtie \textit{account}$ ergibt folgendes Schema:

  (branch-name, branch-city, assets, account-number, balance)
- Nur 1 Attribut wird gebraucht: *account-number* für Join mit *depositor*.
- Umformung: ER8(b) (Projektion nach unten schieben):

  $\pi_{\textit{customer-name}}$
  $(\pi_{\textit{account-number}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch}) \bowtie \textit{account})$
  $\bowtie \textit{depositor})$

Integrierte Übung 4.4

- Verwenden Sie die Äquivalenzregeln, um die Projektionen so weit als möglich nach unten zu schieben:

$$\pi_{\textit{customer-name}}\Big(\big(\pi_{\textit{account-number}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch})) \bowtie \textit{account}\big) \bowtie \textit{depositor}\Big)$$

- Lösung:
  - Anwendung von ER8(b): $A_1 = \emptyset$, $A_2 = \{\textit{account-number}\}$, $A3 = A4 = \{\textit{branch-name}\}$

$$\pi_{\textit{customer-name}}\Big(\pi_{\textit{account-number}}\big(\pi_{\textit{branch-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\textit{branch}))\big) \bowtie \pi_{\textit{account-number},\textit{branch-name}}(\textit{account})\Big) \bowtie \textit{depositor}\Big)$$

  - Anwendung von ER8(d): $A_1 = \{\textit{branch-name}\}$, $A_3 = \{\textit{branch-city}\}$

$$\pi_{\textit{customer-name}}\Big(\pi_{\textit{account-number}}\big(\pi_{\textit{branch-name}}(\sigma_{\textit{branch-city}='\textit{Brooklyn}'}(\pi_{\textit{branch-name},\textit{branch-city}}(\textit{branch}))) \bowtie \pi_{\textit{account-number},\textit{branch-name}}(\textit{account})\big) \bowtie \textit{depositor}\Big)$$

Beispiele Äquivalenzumformungen/5

- Beispiel 4: Joinreihenfolge
- Für alle Relationen r_1, r_2, r_3 gilt (Assoziativität):
$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$
- Falls $r_2 \bowtie r_3$ groß ist und $r_1 \bowtie r_2$ klein, wählen wir die Reihenfolge
$$(r_1 \bowtie r_2) \bowtie r_3$$
sodass nur ein kleines Zwischenergebnis berechnet und evtl. zwischengespeichert werden muss.

Beispiele Äquivalenzumformungen/6

- Beispiel 5: Joinreihenfolge
- Anfrage:

  $\pi_{customer\text{-}name}(\sigma_{branch\text{-}city='Brooklyn'}(branch) \bowtie account \bowtie depositor)$

- Welcher Join soll zuerst berechnet werden?
  (a) $\sigma_{branch\text{-}city='Brooklyn'}(branch) \bowtie depositor$
  (b) $\sigma_{branch\text{-}city='Brooklyn'}(branch) \bowtie account$
  (c) $account \bowtie depositor$

- (a) ist ein Kreuzprodukt, da *branch* und *depositor* keine gemeinsamen Attribute haben
  $\rightarrow$ sollte vermieden werden
- (b) ist vermutlich kleiner als (c), da (b) nur die Konten in Brooklyn berücksichtigt, (c) jedoch alle Konten.

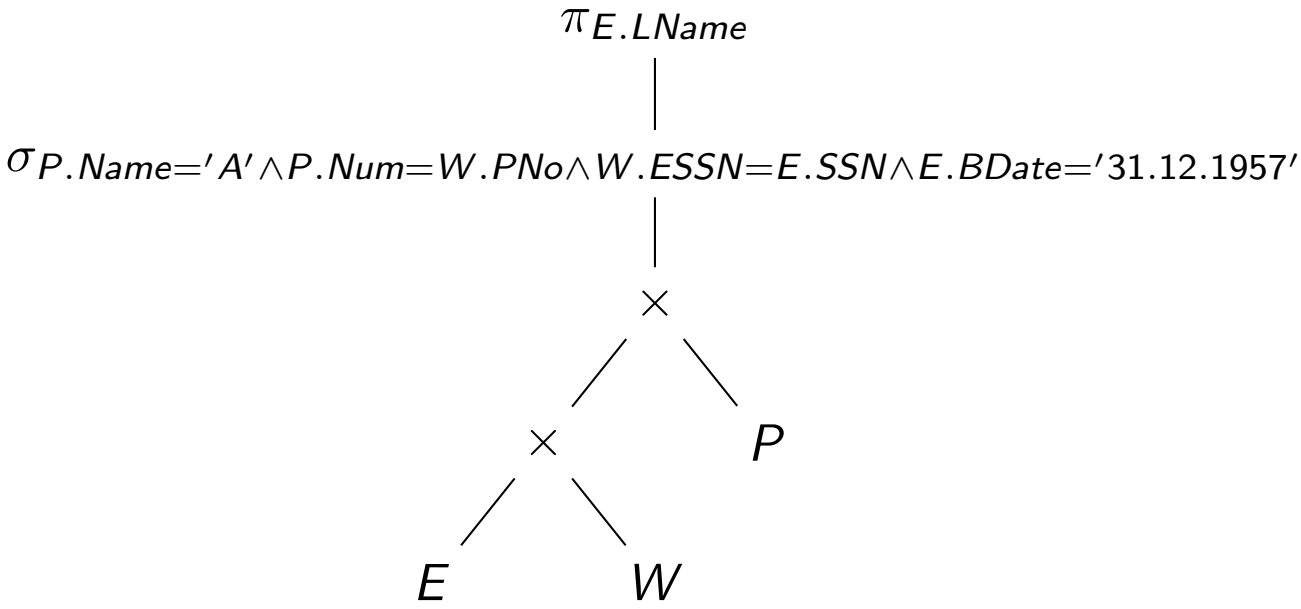
Integrierte Übung 4.5

Stellen Sie die folgende Anfrage als Operatorbaum dar und führen Sie günstige Äquivalenzumformungen durch:

```
SELECT DISTINCT E.LName
FROM Employee E, WorksOn W, Project P
WHERE P.PName = 'A'
AND P.PNum = W.PNo
AND W.ESSN = E.SSN
AND E.BDate = '31.12.1957'
```

Integrierte Übung – Lösung/1

Operatorbaum (algebraische Normalform):



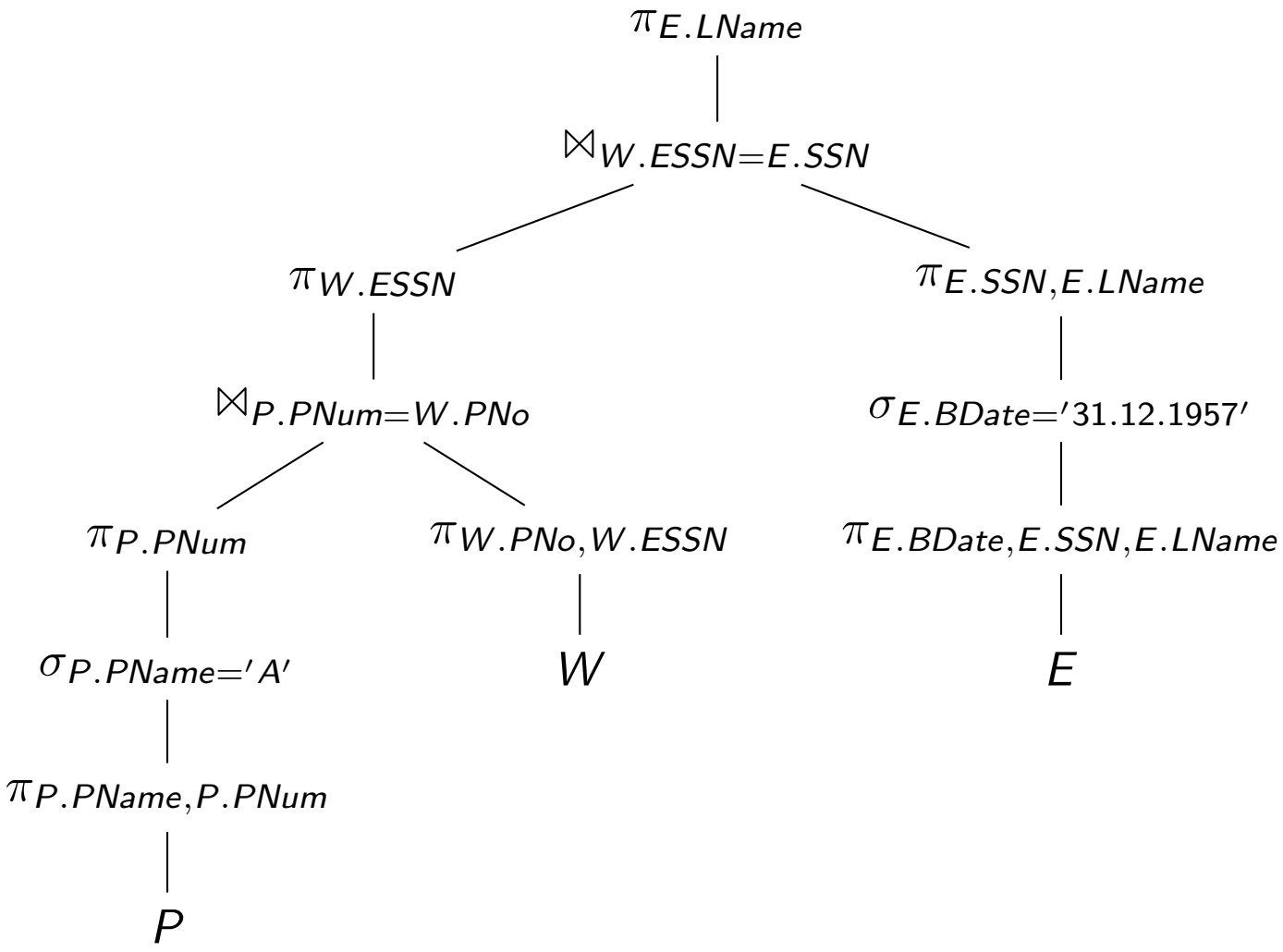
Integrierte Übung – Lösung/2

Anwendung der Äquivalenzregeln:

- Konjunktive Selektionen in verschachtelte Selektionen umwandeln
- Selektionen möglichst weit nach unten schieben
- Kreuzprodukte wenn möglich in Joins umwandeln
- Welcher Join soll als erstes ausgeführt werden?
  - $E \bowtie_\theta P$ wäre ein Kreuzprodukt (da $\theta = \emptyset$) und kommt nicht in Frage
  - beide anderen Möglichkeiten sind sinnvoll, da je eine volle Relation (W) mit einer selektierten Relation (P bzw. E) verbunden wird
  - mit der Annahme, dass es mehr Leute mit gleichem Geburtsdatum als Projekte mit gleichem Namen gibt, wurde W als erstes mit P verbunden
- Projektionen möglichst weit nach unten schieben

Integrierte Übung – Lösung/3

Operatorbaum nach Anwendung der Äquivalenzregeln:



Inhalt

Kostenbasierte Optimierung

- Kostenbasierte Optimierer schätzen die Kosten aller möglichen Anfragepläne ab und wählen den billigsten (=schnellsten).

- Kostenabschätzung erfolgt aufgrund von
  - Datenbankstatistik (im Katalog gespeichert)
  - Wissen über die Kosten der Operatoren (z.B. Hash Join braucht $3(b_r + b_s)$ Blockzugriffe für $r \bowtie s$)
  - Wissen über die Interaktion der Operatoren (z.B. sortiertes Lesen mit einem Index ermöglicht Merge Join statt Sort-Merge Join)

Kombination von Kosten mit Heuristiken

- kostenbasierte Optimierung: durchsuche alle Pläne und suche den billigsten
- heuristische Optimierung: erzeuge einen vielversprechenden Plan nach heuristischen Regeln
- Praktische Optimierer kombinieren beide Techniken:
  - erzeuge eine Menge vielversprechender Pläne
  - wähle den billigsten
  - Plan wird sofort bewertet, sobald er erzeugt wird (und evtl. verworfen)

Teilpläne bewerten

- Optimierer kann Teilpläne bewerten und langsame, äquivalente Teilpläne verwerfen.
  - Dadurch reduziert sich die Menge der Teilpläne, die betrachtet werden müssen.
  - Es reicht jedoch nicht, nur den jeweils schnellsten Teilbaum zu behalten.

- Beispiel:
  - Hash Join ist schneller als Merge Join
  - es kann dennoch besser sein, den Merge Join zu verwenden, wenn die Ausgabe sortiert sein muss
  - der Merge Join liefert ein sortiertes Ergebnis und man spart sich einen zusätzlichen Sortierschritt

Datenbankstatistik

- Katalog (Datenbankverzeichnis) speichert u.A. Informationen über die gespeicherten Daten.
- Statistik über Index: Anzahl der Ebenen in Index i
- Statistik über Tabelle $R(A_1, A_2, \ldots, A_n)$:
  - n_R: Anzahl der Tupel in R
  - b_R: Anzahl der Blöcke, auf denen R gespeichert ist
  - $V(R, A) = |\pi_A(R)|$: Anzahl der unterschiedlichen Werte von Attribut A
- Beispiel: $V(R, A_1) = 1,\ V(R, A_2) = 3,\ V(R, A_3) = 2$

A_1	A_2	A_3
a	b	c
a	x	d
a	y	c

Join Reihenfolgen/1

- Kostenbasierte Optimierung kann verwendet werden, um die beste Join Reihenfolge herauszufinden.
- Join Reihenfolgen der Relationen entstehen durch:
  - Assoziativgesetz: $(R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3)$
  - Kommutativgesetz: $R_1 \bowtie R_2 = R_2 \bowtie R_1$
- Die Join Reihenfolge hat große Auswirkung auf Effizienz:
  - Größe der Zwischenergebnisse
  - Auswahlmöglichkeit der Algorithmen (z.B. vorhandene Indizes verwenden)

Join Reihenfolgen/2

Wieviele Reihenfolgen gibt es für $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m$?

- Assoziativgesetz:
  - Operatorbaum: es gibt C_{m-1} volle binäre Bäume mit m Blättern (anders ausgedrückt: es gibt C_{m-1} Klammerungen von m Operanden)
  - dabei ist C_n die Catalan-Zahl:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \qquad n \geq 0$$

- Kommutativgesetz:
  - Blätter des Operatorbaums sind die Relationen $R_1, R_2, \ldots R_m$
  - für jeden Operatorbaum gibt es $m!$ Permutationen
- Anzahl der Join-Reihenfolgen für m Relationen:

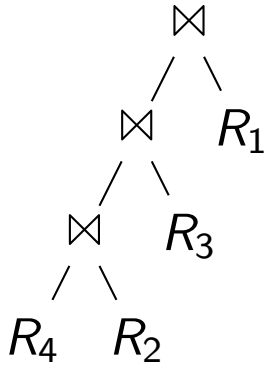
$$m!\, C_{m-1} = \frac{(2(m-1))!}{(m-1)!}$$

Join Reihenfolgen/3

- Anzahl der Join-Reihenfolgen wächst sehr schnell an:
  - $m = 3$: 12 Reihenfolgen
  - $m = 7$: 665.280 Reihenfolgen
  - $m = 10$: $>$ 17.6 Milliarden Reihenfolgen
- Dynamic Programming Ansatz:
  - Laufzeit Komplexität: $O(3^m)$
  - Speicher Komplexität: $O(2^m)$
- Beispiel: $m = 10$
  - Anzahl der Join-Reihenfolgen: 17.6×10^9
  - Dynamic Programming: $3^m = 59'049$
- Trotz Dynamic Programming bleibt Aufzählung der Join-Reihenfolgen teuer.

Join Reihenfolgen/4

- Left-deep Join Reihenfolgen
  - rechter Join-Operator ist immer eine Relation (nicht Join-Ergebnis)
  - dadurch ergeben sich sog. left-deep Operatorbäume (im Gegensatz zu "bushy", wenn alle Operatorbäume erlaubt sind)



- Anzahl der left-deep Join Reihenfolgen für m Relationen: $O(m!)$
- Dynamic Programming: Laufzeit $O(m\,2^m)$.
- Vergleich für m Relationen und Beispiel $m = 10$:

	left-deep	bushy
#Baumformen	1	C_{m-1}
#Join Reihenfolgen	$m!$	$\frac{(2(m-1))!}{(m-1)!}$
Dynamic Programming	$O(m\,2^m)$	$O(3^m)$

$m = 10$	left-deep	bushy
	1	4.862
	3.63×10^6	1.76×10^{10}
	10.240	59.049

Greedy Algorithmus für Join Reihenfolgen

- Ansatz: In jedem Schritt wird der Join mit dem kleinsten Zwischenergebnis verwendet.
- Überblick: Greedy Algorithmus für Join Reihenfolge
  - nur left-deep Join Reihenfolgen werden betrachtet
  - Relationen-Paar mit dem kleinsten Join Ergebnis kommt zuerst dran
  - in jedem weiteren Schritt wird jene Relation dazugegeben, die mit dem vorhandenen Operatorbaum das kleinste Join-Ergebnis erzeugt
- Algorithmus: Join Reihenfolge von $S = \{R_1, R_2, \ldots, R_m\}$
  1. $O \leftarrow R_i \bowtie R_j$, sodass $|R_i \bowtie R_j|$ minimal ist $(i \neq j)$
  2. $S \leftarrow S - \{R_i, R_j\}$
  3. **while** $S \neq \emptyset$ **do**
     a. wähle $R_i \in S$ sodass $|O \bowtie R_i|$ minimal ist
     b. $O \leftarrow O \bowtie R_i$
     c. $S \leftarrow S - \{R_i\}$
  4. **return** Operatorbaum O
- Laufzeit: $O(n^2)$

Abschätzung der Join Kardinalität/1

- Greedy Algorithmus benötigt Abschätzung der Join Kardinalität.
- Abschätzung erfolgt aufgrund der Anzahl der unterschiedlichen Werte für die Join Attribute, z.B. $V(R, A)$.
- Abschätzung für $|R \bowtie S|$ mit dem Join Attribut A:

$$|R \bowtie S| \approx \frac{|R| \cdot |S|}{\max(V(R, A), V(S, A))}$$

- Annahmen über die Werte der Attribute (A ist Join-Attribut):
  - *Gleichverteilung*: Jeder der Werte in $\pi_A(R)$ bzw. $\pi_A(S)$ kommt mit der gleichen Wahrscheinlichkeit vor.
  - *Teilmenge*: $V(R, A) \leq V(S, A) \Rightarrow \pi_A(R) \subseteq \pi_A(S)$
  - *Werterhaltung*: falls Attribut B in R vorkommt aber nicht in S (d.h. B ist kein Join-Attribut), dann gilt: $V(R \bowtie S, B) = V(R, B)$

Abschätzung der Join Kardinalität/2

- Beispiel: schätze $|R \bowtie S|$ ab, wobei folgende Statistik gegeben ist.

$R(A,B)$	$S(B,C)$
$n_R = 1000$	$n_S = 2000$
$V(R,B) = 20$	$V(S,B) = 500$

- Abschätzung:

$$|R \bowtie S| \approx \frac{n_R \cdot n_S}{\max(V(R,B), V(S,B))} = \frac{1000 \cdot 2000}{500} = 4000$$

Abschätzung der Join Kardinalität/3

- Bisherige Abschätzung ist limitiert auf 1 Join-Attribut zwischen 2 Relationen.
- Für den Greedy Algorithmus muss die Abschätzung verallgemeinert werden:
  - m Relationen $R_1, R_2, \ldots, R_m$
  - beliebig viele Join-Attribute (A ist Join-Attribut wenn es in mindestens zwei Relationen vorkommt)
- Verallgemeinerung der Abschätzung:
  1. starte mit der Größe des Kreuzproduktes $|R_1| \cdot |R_2| \cdot \ldots \cdot |R_m|$
  2. für jedes Join-Attribut: dividiere durch alle $V(R_i, A)$ außer durch das kleinste

Abschätzung der Join Kardinalität/4

- Beispiel: schätze $|R \bowtie S \bowtie T|$ ab, wobei folgende Statistik gegeben ist.

$R(A,B,C)$	$S(B,C,D)$	$T(B,E)$
$n_R = 1000$	$n_S = 2000$	$n_T = 5000$
$V(R,A) = 100$		
$V(R,B) = 20$	$V(S,B) = 50$	$V(T,B) = 200$
$V(R,C) = 200$	$V(S,C) = 100$	
	$V(S,D) = 400$	
		$V(T,E) = 500$

- Abschätzung:

$$|R \bowtie S \bowtie T| \approx \frac{n_R \cdot n_S \cdot n_T}{V(S,B) \cdot V(T,B) \cdot V(R,C)} = 5000$$

Integrierte Übung 4.6

Eine Datenbank mit folgenden Relationen ist gegeben:

- $|R_1(A, B, C)| = 1000, V(R_1, C) = 900$
- $|R_2(C, D, E)| = 1500, V(R_2, C) = 1100, V(R_2, D) = 50, V(R_2, E) = 50$
- $|R_3(D, E)| = 750, V(R_3, D) = 50, V(R_3, E) = 100$

Finden Sie eine effiziente Join Reihenfolge für den Join $R_1 \bowtie R_2 \bowtie R_3$ und berechnen Sie die Kardinalität des Join-Ergebnisses.