# DB IIb, Implementierung von Datenbanken

Alexander Hinneburg

WS 2008/2009

## Inhaltsverzeichnis

# 1 Organisation

### Organisation der Vorlesung

- Die Vorlesung findet Do. 12:15-13:45, Raum 3.31 und die Übung Mi. 14:15-15:45, Raum 1.03 statt. Der Stoff aus Vorlesung und Übung ist prüfungsrelevant.
- Die Vorlesung hat 15 Wochen und ist in zwei Teile gegliedert
  - Teil 1 geht von der ersten bis zur 6. Woche
  - Teil 2 geht von der 8. bis zur 14. Woche
- In der 7. und 15. Woche werden die Klausuren zur Vorlesungszeit (jeweils 90 min) geschrieben.
- Es gibt keine Voraussetzungen, um an den Klausuren teilnehmen zu können. Es wird empfohlen jeweils 50% der Übungspunkte in beiden Vorlesungsteilen zu erreichen.
- Für die Wirtschaftsinformatiker werden beide Klausuren mit jeweils 50 Fachpunkten bewertet. Bekanntgabe der Ergebnisse sind jeweils 2 Wochen nach der Klausur.
- Für WI-Inf ist das eine studienbegleitende Prüfung mit 5 LP für Vorlesung und Übung für mindestens 50 Fachpunkte erbracht werden müssen.

### Organisation der Übung

- Die Übungsblätter werden immer am Mittwoch zur Übungszeit ins Netz gestellt.
- Die Übungen sind eine Woche später bis Mittwoch 10.00 Uhr elektronisch mittels Subversion (SVN) abzugeben.
- Übungsgruppen von maximal zwei Personen sind zulässig.
- Zum Vorstellen der Übungsaufgaben muss eine kleine Präsentation in PDF vorbereitet werden. Wer seine Lösung für eine Übung nicht vorstellen kann, muss mit Punktabzug rechnen.

### Bücher

- Theo Härder, Erhard Rahm: Datenbanksysteme – Konzepte und Techniken der Implementierung. 2. Auflage, 2001, 39,95 Euro, (ist zusammen mit Skript prüfungsrelevant).
- Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems. Mcgraw-Hill Higher Education, 3. Auflage, 2002, EUR 142,13
- Gunter Saake, Andreas Heuer, Kai-Uwe Sattler: Datenbanken: Implementierungstechniken. Mitp-Verlag, 2. Auflage, 2005, EUR 49,95
- Ramez A. Elmasri, Shamkant B. Navathe: Grundlagen von Datenbanksystemen. Pearson Studium, 2002
- Gottfried Vossen: Datenbankmodelle, Datenbanksprachen und Datenbankmanagementsysteme. Oldenbourg, 4. Auflage, 2000

# 2 Architektur eines DBS

## 2.1 Anforderungen an DBS

### Anforderungen an DBS

- Repräsentation von Ausschnitten der Realität durch ein Modell
- Erfassung von relevanten Zuständen des Modells
- Reale Vorgänge werden als Zustandsübergang nachgebildet
- Möglichst gute Übereinstimmung von Realität und Modell
- Zustandsänderungen hinsichtlich technischer Fehler ununterbrechbar ⇒ Transaktionsmodell zur Qualitätssicherung

### Entwicklung von Anwendungsprogrammen 1/3

- Anforderung an Anwendungs- und Informationssysteme: Gewährleistung von
  1. aktuellen
  2. konsistenten und
  3. persistenten Daten

### Entwicklung von Anwendungsprogrammen 2/3

- Nachteile der Realisierung mittels Dateien
  - Anwendungsbreich mit Sachverhalten, Abhängigkeiten und Beziehungen ist nur grob modellierbar
  - Das Speichern und Aktualisieren von Daten ohne zentralisierte Kontrolle erschwert das Aufdecken von fehlerhaften, widersprüchlichen oder unvollständigen Informationen.
  - Dateien sind für konkrete Anwendungen und Speicherstrukturen konzipiert und auf spezielle Verarbeitungsanforderungen optimiert. Anwendungsbindung der Dateien erzeugt
    * hohe Datenabhängigkeit
    * schränkt flexible Nutzung von Dateien für andere Anwendungsprogramme (AP) ein
    * ist Ursache für mangelnde Offenheit und Erweiterbarkeit
  - Verwendung von Dateien führt oft zu redundanten Daten, was
    * zeitgerechte und
    * alle Kopien umfassendende Änderungen der Daten verhindert
  - Zusätzlicher neuer Informationsbedarf erzwingt die Evolution des Anwendungssystems; eingeforene Datenstrukturen- und Beziehungen führen zu weiterer Datenredundanz.

### Entwicklung von Anwendungsprogrammen 3/3

- Schlußfolgerung: mit isolierten Dateien lassen die Entwurfsziele
  - flexible Anpassung und
  - Aktualität der Daten und
  - deren konsistente Speicherung nicht erreichen.

- DBS für alle Aufgaben der Datenhaltung
  - integrierte Sicht auf alle Daten,
  - Konsistenzerhaltung,
  - Systemevolution ,
  - Anpassung an geänderte Umweltbedingungen

  **in Unabhängigkeit von AP**.

### Entwurfziele

- Breiter Einsatz von DBS führt zu vielfältigem Spektrum von Anforderungen
- Vor allem folgende Entwurfsziele sind zu realisieren
  - Integration der Daten und Unabhängigkeit der Datenverwaltung
  - Datenunabhängikeit & Anwendungsneutralität
  - Anwendungsprogrammierschnittstellen
  - Zentrale Integritätskontrolle
  - Transaktionsschutz
  - Effizienz und Parallelität
  - Hohe Verfügbarkeit und Fehlertoleranz
  - Skalierbarkeit

### Integration der Daten und Unabhängigkeit der Datenverwaltung

- Herauslösen aller Aufgaben der Datenverwaltung und Konsistenzkontrolle aus dem AP.
- Standardisierung und Übernahme in ein zentralisiertes System, um
  - Zuverlässigkeit,
  - Widerspruchsfreiheit und
  - Vollständigkeit der Operationen auf allen Daten zu gewährleisten.
- Konsistenz und Langfristigkeit durch eine Menge von AP zu gewährleisten geht nicht, deshalb Zentralisierung im DBS

### Datenunabhängikeit & Anwendungsneutralität

- AP nutzen Daten aus DBS ohne systemtechnische Details zu kennen
- Kapselung wird durch logisches Datenmodell und deklarative Anfragesprache erreicht.
- Datenunabhängigkeit gewährleistet Langfristigkeit ⇒ wechselseitige Änderungsimmunität zwischen AP und Daten.
- Anwendungsneutralität: keine Bevorzugung bestimmter Anwendungen, offen für neue Anwendungen

### Anwendungsprogrammierschnittstellen (API)

- „hohe“, abstrakte API für logisches Datenmodell und deklarative Anfragesprache
- Einbettung in Wirtssprachen
- Typkonvertierung

### Zentrale Integritätskontrolle

- wichtige nur zentral zu gewährleistende Aufgaben
  1. zuverlässiger Datenschutz und Rechtekontrolle
  2. Konsistenzsicherung und Einhalten von Integritätsbedingungen
  3. Logging und Recovery Maßnahmen
  4. Synchronisation von parallelen Benutzeroperationen

### Transaktionsschutz

- Transaktionsschutz mit ACID-Eigenschaften garantiert Korrektheit und Ablauf der gesamten DB-Verarbeitung, besonders bei
  - konkurrierender Benutzung und
  - im Fehlerfall.
- Transaktionen bieten
  - Atomarität (atomicity) für alle DB-Operationen,
  - Konsistenz (consistency) der geänderten DB,
  - isolierte Ausführung (isolation) im Mehrbenutzerbetrieb,
  - Dauerhaftigkeit (durability) der eingebrachten Änderungen.
- „Alles-oder-Nichts"-Eigenschaft (Atomarität) ist ein einfaches, leicht verständliches Fehlermodell (Vertragsrecht zwischen DBS und AP).
- Hauptvorteile des Transaktionskonzeptes: (a) (b)
  - fehlerfreie Sicht des AP auf die Daten und
  - Isolierung vom Mehrbenutzerbetrieb.
- Transaktionen werden auch in anderen Systemen genutzt, z.B. Betriebssysteme.

### Effizienz und Parallelität

- Notwendige Voraussetzung für Einsatz in der Praxis.
- Zugriffsmethoden müssen sehr schnell sein, Anzahl der Externspeicherzugriffe darf höchstens sublinear (z.B. logarithmisch) in Größe des Datenvolumens wachsen.
- Neue Auswertungsmethoden müssen durch neue Algorithmen, verbesserte Vorplanung, vermehrten Hauptspeichereinsatz zur Datenpufferung und Nutzung inhärenter Zugriffsparallelität unterstützt werden.

### Hohe Verfügbarkeit und Fehlertoleranz

- Anwendungen in der Wirtschaft erlauben keine separate Off-Line Zeit zur Reorganisation, Backup, usw.
- Speicherstrukturen und Sicherungsmaßnahmen müssen dynamisch und inkrementell arbeiten.
- Fehlertoleranz verlangt redundante Auslegung von Hard- und Software-Einheiten, um Fehler lokal zu isolieren und den Betrieb aufrecht zu erhalten.

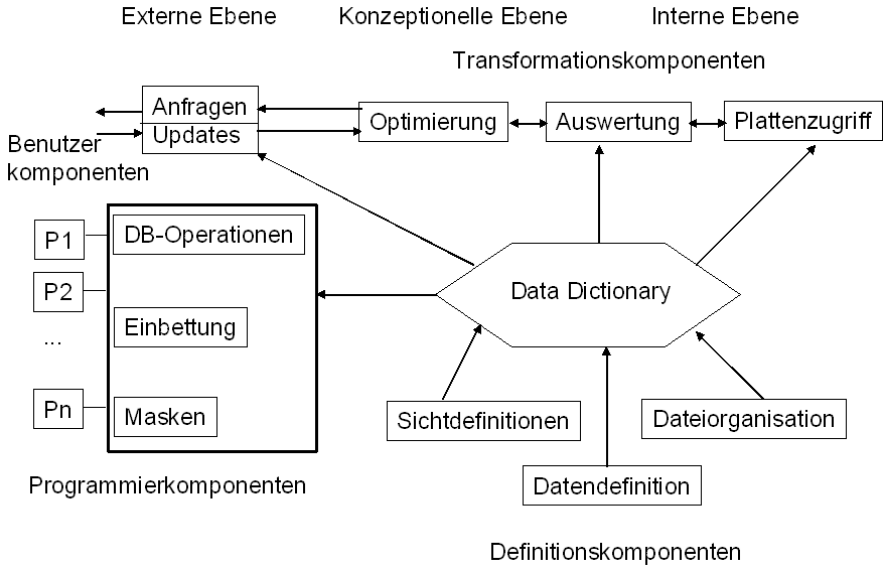



Abbildung 1: Schemaarchitektur

#### Skalierbarkeit

- Skalierbarkeit bei wachsender Transaktionslast und Datenvolumina
- Durch mehr CPUs und schnellere Festplatten und soll (idealerweise) ein linearer Leistungsanstieg erreicht werden, d.h. (a) (b)
  - mehr Transaktionen bei gleicher Antwortzeit oder
  - gleicher Transaktionsdurchsatz bei geringerer mittlerer Antwortzeit.
- DBS soll mit den Datenvolumina und Anwendungslasten ausschließlich durch vermehrten Hardware-Einsatz wachsen können, ohne seine Leistungsfähigkeit einzubüßen.
- Skalierbarkeit ist Voraussetzung für betrieblichen Einsatz von DBS.

## 2.2 DB-Schemaarchitektur nach ANSI/SPARC

#### Architektur eines DBS

- ANSI/X3/Sparc Studie

## 2.3 Schichtenmodell-Architektur

#### Architekturprinzipien

- Empfohlene SW-Konzepte
  - Geheimnisprinzip (Information Hiding)
  - Hierarchische Strukturierung
- Schicht $i+1$ benutzt Operatoren und Datenobjekte die Schicht $i$ realisiert.
- Aufbauprinzip einer Schicht:

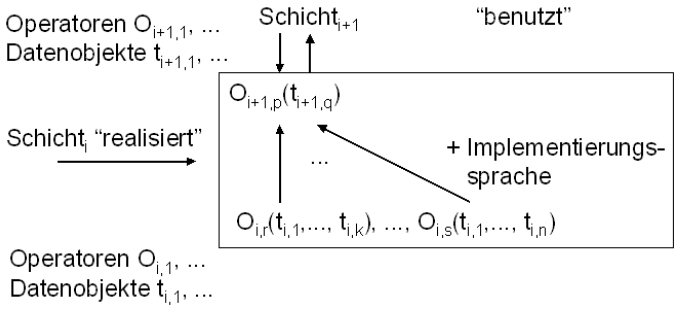



Abbildung 2: Schichtaufbau

### Architekturprinzipien

- Vorteile der Schichtenbildung
  - höhere Ebenen werden einfacher, weil sie tiefere Ebenen benutzen können,
  - Änderungen auf höheren Ebenen bleiben ohne Einfluß auf tiefere Ebenen,
  - höhere Ebenen lassen sich abtrennen, tiefere Ebenen funktionieren trotzdem und
  - tiefere Ebenen können separat getestet werden.
- Schichtenbildung ist eine Möglichkeit, um die bei der Software-Entwicklung geforderten Ziele (Wartbarkeit, Erweiterbarkeit, etc.) zu erreichen.
- „Veredelungshierarchie“: Entlang der Schichten wird nach oben hin „abstrahiert“.
- Offene Frage: Anzahl der Schichten
  - $n = 1$: Monolythisches System $\Rightarrow$ keine Vorteile der Schichtenbildung
  - $n$ sehr gross: $\Rightarrow$ hoher Koordinierungsaufwand
  - Daumenregel: $n$ meist zwischen 3 und 10

### Einfaches Schichtenmodell

- Speichersystem:
  - bildet Seiten und Segmente auf Blöcke und Dateien und
  - dient zur Satzadressierung, stellt Einbringstrategien bereit und verwaltet den Systempuffer.
- Zugriffssystem:
  - umfaßt eindimensionale Zugriffspfade (B*-Baum, Hashing, Invertierung, ....) und
  - mehrdimensionale Zugriffspfade (Quad-Tree, K-d-Baum, Multi-Key Hashing, R*-Baum...) und
  - unterstützt so das effiziente Abbilden von Sätzen auf Seiten.
- Datensystem umfaßt die Methoden zur Anfrageverarbeitung, d.h. das relationales Datenmodell und die relationale Algebra, es organisiert die Phasen der Anfrageverarbeitung und optimiert Strategien zur Anfrageverarbeitung.

### Vereinfachtes Schichtenmodell

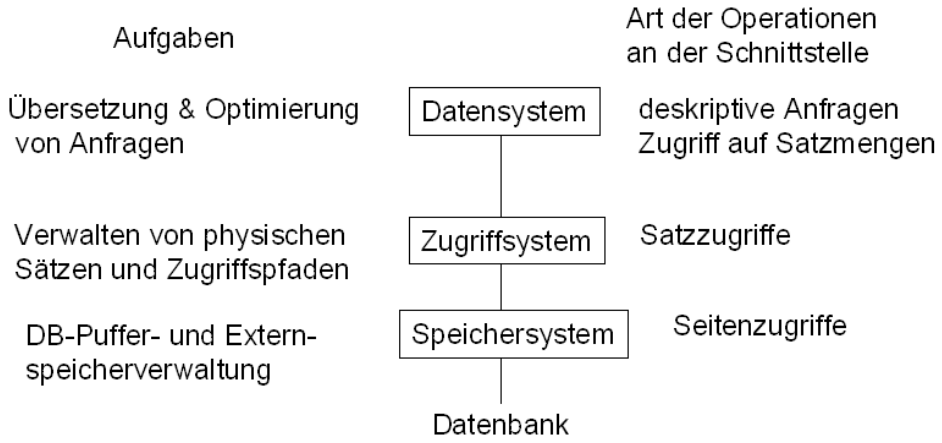



Abbildung 3: Vereinfachtes Schichtenmodell

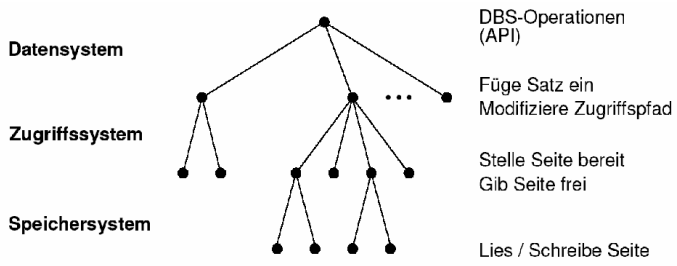



Abbildung 4: Kontrollfluß durch Schichtenmodell

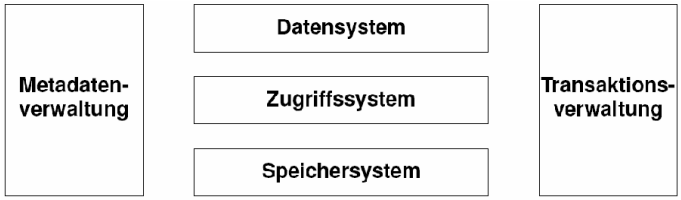



Abbildung 5: Metadaten und Transaktionsverwaltung

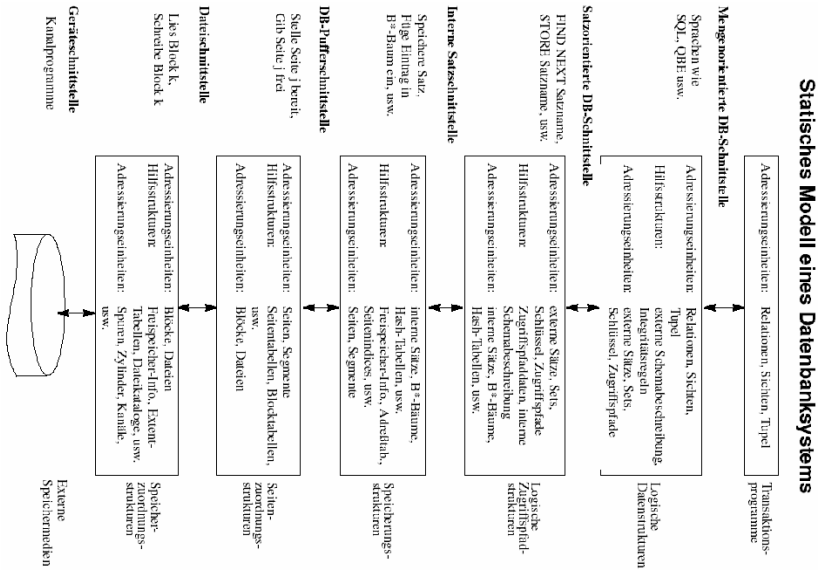



Abbildung 6: Schichtenmodell nach Härder

### Vereinfachtes Schichtenmodell

- Dynamischer Kontrollfluß durch das Schichtenmodell
- Eine mengenorientierte DB-Operation wird durch satzorientierte Operationen des Zugriffssystems abgewickelt.
- Einige Seiten sind bereits im DB-Puffer.
- Ablauf kann man sich als (left-most, depth-first) Tiefensuche vorstellen.
- Teilbäume weisen auf Nebenläufigkeit und Parallelisierungsmöglichkeiten hin.

### Vereinfachtes Schichtenmodell

- Metadaten und Transaktionsverwaltung
- Konfiguration des generischen Systems mit Beschreibungsinformation eines realen Systems ⇒ Data Dictionary
- DBS stellt Transaktionsverwaltung bereit.

### Verfeinertes Schichtenmodell nach Härder

### Verfeinertes Schichtenmodell nach Härder

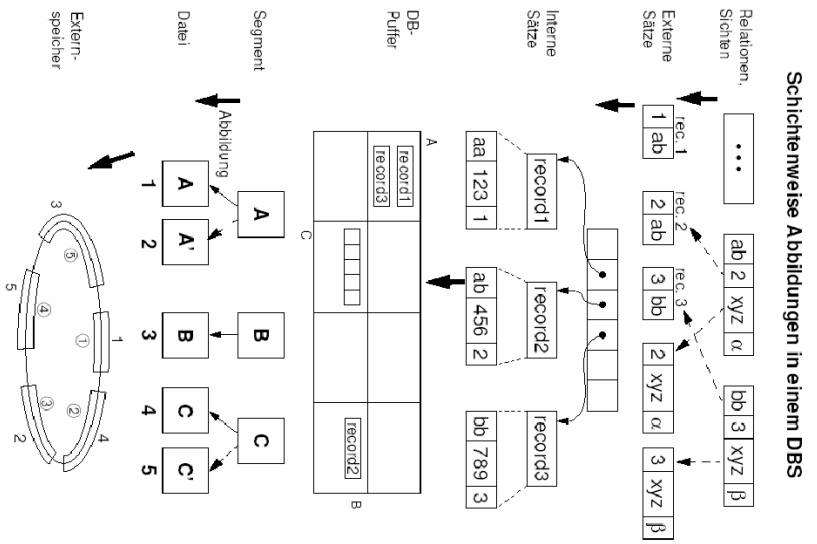



Abbildung 7: Schichtenmodell nach Härder

| Ebene | Was wird verborgen? |
|---|---|
| Logische Datenstrukturen | Positionsanzeiger und explizite Beziehungskonstrukte im Schema |
| Logische Zugriffspfade | Zahl und Art der physischen Zugriffspfade; interne Satzdarstellung |
| Speicherungsstrukturen | Pufferverwaltung; Recovery-Vorkehrungen |
| Seitenzuordnungsstrukturen | Dateiabbildung, Recovery-Unterstützung durch das BS |
| Speicherzuordnungsstrukturen | Technische Eigenschaften und Betriebsdetails der externen Speichermedien |

Abbildung 8: Schichtenmodell nach Härder

### Verfeinertes Schichtenmodell nach Härder

- Optimierungen: Übergang zwischen Schichten kostet Laufzeit.
- In unteren beiden Schichten bietet ein großer DB-Puffer die wirksamste Verbesserung.

### Zusammenfassung DBS-Architektur

- Schichtenansatz „scheint“ geeignet zur Strukturierung großer Anwendungssysteme
- Aufbau mehrschichter Software-Systeme Datenbanksysteme
  - Speichersystem,
  - Zugriffssystem,
  - Datensystem
- Optional: 5-schichtiger Architekturansatz; basierend auf Idee von Senko 1973; Weiterentwicklung von Härder 1987

- Umsetzung im Rahmen des IBM-Prototyps System R
- Genaue Beschreibung der Transformationskomponenten (Dienste und Schnittstellen)

# 3 Konzepte und Komponenten der E/A Architektur

### Konzepte & Komponenten der E/A-Architektur 1/2

- DBS werden
  - für die Verwaltung von sehr großen Datenmengen entwickelt,
  - um Anwendungen abstrakte Sichten auf die Daten bereitzustellen und
  - standardisierte komplexe Operationen effizient und zuverlässig auszuführen.
- Ideal wäre ein nichtflüchtiger Speicher mit nahezu unbegrenzter Speicherkapazität, kurzer Zugriffszeit bei wahlfreiem Zugriff, hohen Zugriffsraten und geringen Kosten.
- Hätte dieser Speicher noch die Eigenschaft, daß die CPU direkt auf ihm arbeiten kann, dann wären Datenbanken sehr einfach zu implementieren.
- Da es diesen idealen Speicher nicht gibt, wird eine komplexe Hardware-Architektur bestehend aus CPU, Hauptspeicher und E/A-Subsystem genutzt, um das gewünschte Systemverhalten in wesentlichen Eigenschaften anzunähern.

### Konzepte & Komponenten der E/A-Architektur 2/2

- Engpässe bei dieser grob skizzierten Architektur sind die Übergänge zwischen den einzelnen Komponenten.
- Die Anbindung des Prozessors an den Hauptspeicher (von-Neumann-Flaschenhals) müssen alle verarbeiteten Operationen und Daten passieren. Die Geschwindigkeitsdifferenz ist etwa Faktor $10^1$–$10^2$.
- Wichtig für DBS ist auch die Anbindung des Externspeichers an den Hauptspeicher, wo eine Geschwindigkeitsdifferenz von etwa $10^5$ (Zugriffslücke) besteht.
- Technologische Fortschritte führen zu enormen Steigerungsraten bei Kapazität und Zugriffszeit des einzelnen Elemente, was eine ständige Anpassung der Komponenten erzwingt.

## 3.1 Einflussfaktoren

### Große Einflußfaktoren

- Rechengeschwindigkeit, Bill Joy (1985): „Rechenleistung verdoppelt sich jedes Jahr"

  $$SunMips(Jahr) = 2^{Jahr-1984}, Jahr \in [1984, \ldots, 2000]$$

  ist heute zu optimistisch, es gibt nur einen Zuwachs von 60% pro Jahr.
- Speicherchipkapazität, Moore's Law:

  $$Kapazitaet(Jahr) = 4^{\frac{Jahr-1970}{3}} \text{KB/Chip}, Jahr \in [1970, \ldots, 2000]$$

  Speicherkapazität verdoppelt sich alle 18 (24) Monate. Festplattenkapazität nimmt etwa in derselben Geschwindigkeit zu.
- Zugriffsraten für Speicherchips und Festplatten steigen nur um etwa 7-10% pro Jahr.
- Diese Leistungsbremse nimmt immer stärker zu. Deshalb sind Methoden zur Externspeicheranbindung für DBS besonders wichtig.

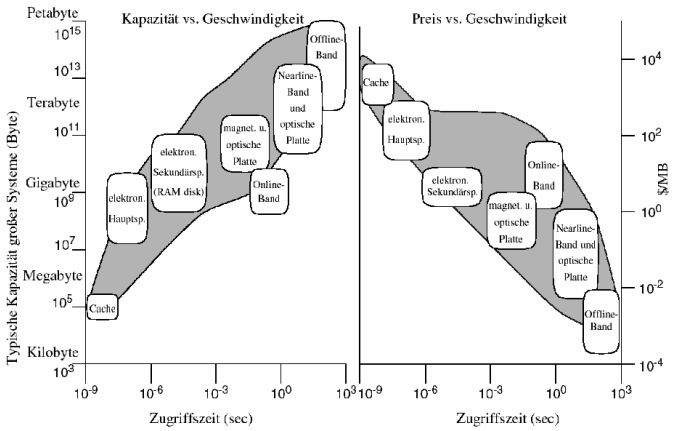



Abbildung 9: Speicherhierarchie

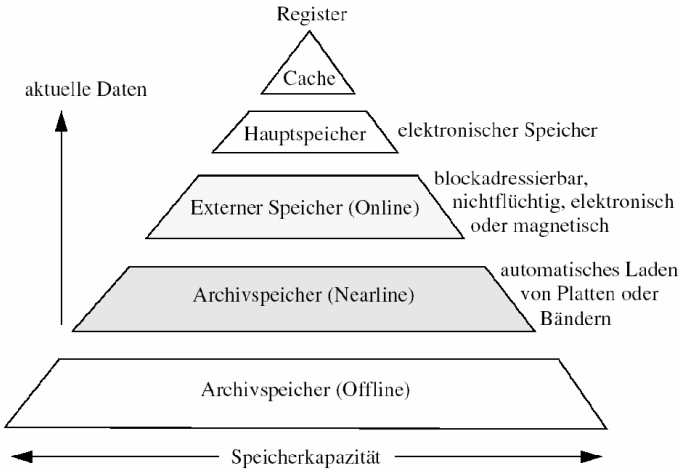



Abbildung 10: Aufbau einer Speicherhierarchie

## 3.2 Speicherhierarchien

### Speicherhierarchien

- Der ideale Speicher hält die vom Prozessor gebrauchten Daten schon bereit. Die Verhältnisse zwischen Kapazität und Zugriffszeit, bzw. Preis zeigen, das dies durch reale Speicher nicht zu bewerkstelligen ist.
- Z.B. ist Cache-Speicher klein und teuer.

### Aufbau einer Speicherhierarchie

- Mit steigender Hierarchie schrumpfen Kapazität und Zugriffsbreite, Geschwindigkeit und Kosten nehmen zu. Kleine, teure Speicher werden genutzt, um Daten in großen, langsamen Speichern zwischenzuspeichern.
- Speicherhierarchien funktionieren nur wegen der Lokalität der Datenzugriffe.
- Referenzlokalität und geeignete Ersetzungsverfahren in Datenpuffern erzielen hohe Trefferraten. Diese können durch größeren Pufferspeicher sowie durch Clusterbildung bei den referenzierten Daten gesteigert werden.

#### 5-Minuten-Regel

- Anzahl der Plattenzugriffe pro Sekunde kann durch Anzahl der Platten variiert werden
- Statt in Platten zu investieren, kann auch Speicher gekauft werden, der Daten zwischenspeichert und somit Plattenzugriffen einspart.
- Je länger Daten im Speicher unbenutzt liegen, desto weniger rentiert sich die Investition in Speicher.
- 5-Minuten-Regel: Halte ein Datenobjekt im Hauptspeicher, wenn es innerhalb von 5 min. erneut referenziert wird.

$$BreakEvenRefInterval(sec) = \frac{PagesPerMBofRAM}{AccessPerSecPerDisk} \cdot \frac{PricePerDisk}{PricePerMBofRAM}$$

- Aktuelle Kosten ergeben Interval von 1-10 min.

#### 5-Minuten-Regel, Herleitung

- Ein Festplatte braucht 10ms um auf eine 64KB Seite wahlfrei zuzugreifen. Die Platte kostet 200€, d.h. ein Zugriff pro Sekunde kostet $^{200€}/_{\frac{1000\text{ms}}{10\text{ms}}} = 2€$.
- Hauptspeicher kostet 100€ für 1 GB.
- Wenn 64KB im Hauptspeicher mit den Daten aus einer Seite belegt werden, um einen Plattenzugriff pro Sekunde zu vermeiden, spart man dessen Kosten von 2€ für den Preis von $6 \cdot 10^{-3}€$ für die 64KB.
- Um einen Plattenzugriff alle 10 Sekunden zu vermeiden, spart man 0.2€ für den Preis von $6 \cdot 10^{-3}€$.
- Das Verhältnis kippt, wenn man nur einen Plattenzugriff rund alle 300 Sekunden sparen will. 300 Sekunden sind 5 min.

Hausaufgabe: Lesen Sie die relevanten Teile der Originalartikel und leiten Sie die Fünf-Minuten-Regel formal her.

### 3.3 Magnetplatten

#### Aufbau von Festplatten

- Aufbau
- Platten befinden sich in ständiger Rotation, $5 - 15 \cdot 10^3$ U/min.
- Blöcke sind eine Menge von aufeinander folgender Sektoren.

#### Positionierungskomponente

- Adressierung auf einer Festplatte erfolgt in einem 3D-Adressraum (Zylinder, Spur, Sektor).
- Das Positionierungssystem muß bei einem E/A-Auftrag den Zugriffskamm so schnell wie möglich zum adressierten Zylinder bewegen und warten bis der gewünschte Sektor auftaucht.
- Zugriffszeit für einen Block

$$t = t_{sio} + t_w + t_s + t_{act} + t_r + t_{recon} + t_{tr} + t_c$$

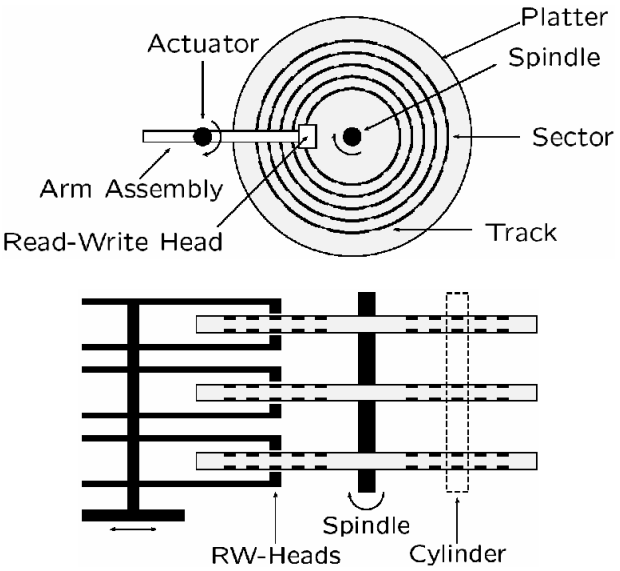



Abbildung 11: Festplattenaufbau

| Merkmal \ Magnetplattentyp | Spitzen-werte 2001 | typische Werte 1998 | IBM 3390 (1990) | IBM 3380 (1985) | IBM 3330 (1970) |
|---|---|---|---|---|---|
| $t_{smin}$ = Zugr.beweg.(Min.) | 0.65 ms | 1 ms | k. A. | 2 ms | 10 ms |
| $t_{sav}$ = " (Mittel) | 4.1 ms | 8 ms | 12.5 ms | 16 ms | 30 ms |
| $t_{smax}$ = " (Max.) | 8.5 ms | 16 ms | k. A. | 29 ms | 55 ms |
| $t_{rev}$ = Umdrehungszeit | 4 ms | 6 ms | 14.1 ms | 16.7 ms | 16.7 ms |
| $T_{cap}$ = Spurkapazität | 178 KB | 100 KB | 56 KB | 47 KB | 13 KB |
| $T_{cyl}$ = #Spuren pro Zyl. | 11 | 20 | 15 | 15 | 19 |
| $N_{dev}$ = #Zylinder | 18700 | 5000 | 2226 | 2655 | 411 |
| u = Transferrate | 45 MB/s | 15 MB/s | 4.2 MB/s | 3 MB/s | 806 KB/s |
| Nettokapazität | 36.7 GB | 10 GB | 1.89 GB | 1.89 GB | 93.7 MB |

Abbildung 12: Plattenparameter

- Lastunabhängiges vereinfachtes Modell

$$t = t_s + t_r + t_{tr}$$

- Mittlere Zugriffszeit

$$\bar{t} = \bar{t}_s + \bar{t}_r + \bar{t}_{tr} = t_{sav} + t_{rev}/2 + B_L/u$$

### Typische Plattenparameter

### Sequentieller vs. wahlfreier Zugriff

- Beim sequentiellen Lesen fallen nur zum erstmaligen Positionieren eine Zugriffsbewegung und eine Umdrehungswartezeit an. Die Daten einer Spur werden mit der vollen Transferrate übertragen. Zum Aufsuchen des nächsten Zylinders ist nur $t_{smin}$ nötig.

$$\bar{t} = t_{sav} + t_{rev}/2 + k \cdot t_{smin} + B_A \cdot B_L/u$$

RAID 0

| Disk 0 | Disk 1 | Disk 2 | Disk 4 | Disk 4 |
|---|---|---|---|---|
| Block 01 | Block 02 | Block 03 | Block 04 | Block 05 |
| Block 06 | Block 07 | Block 08 | Block 09 | Block 10 |
| Block 11 | Block 12 | Block 13 | Block 14 | Block 15 |
| Block 16 | Block 17 | Block 18 | Block 19 | Block 20 |
| Block 21 | Block 22 | Block 23 | Block 24 | Block 25 |

RAID 1

| Disk 0 | Disk 1 |
|---|---|
| Block 01 | Block 01 |
| Block 02 | Block 02 |
| Block 03 | Block 03 |
| Block 04 | Block 04 |
| Block 05 | Block 05 |

RAID 5

| Disk 0 | Disk 1 | Disk 2 | Disk 4 | Disk 4 |
|---|---|---|---|---|
| Block 01 | Block 02 | Block 03 | Block 04 | Parity 01-04 |
| Block 06 | Block 07 | Block 08 | Parity 05-08 | Block 05 |
| Block 11 | Block 12 | Parity 09-12 | Block 09 | Block 10 |
| Block 16 | Parity 13-16 | Block 13 | Block 14 | Block 15 |
| Parity 17-20 | Block 17 | Block 18 | Block 19 | Block 20 |

Abbildung 13: Raid-Systeme, Beispiel

Parity i-j = $\text{Block}_i$ XOR $\text{Block}_{i+1}$ XOR ... XOR $\text{Block}_j$

- Gesamtzeit für wahlfreien Zugriff

$$\bar{t} = B_A(t_{sav} + t_{rev}/2 + B_L/u)$$

- Beispiel:

## 3.4 RAID-Systeme

### Überblick RAID Systeme

- Redundant Array of Inexpensive Disks, erhöhter Durchsatz durch mehrere Disks im Parallelbetrieb ⇒ Performanzsteigerung
- Erhöhte Fehlertoleranz durch redundante Daten auf mehreren Disks ⇒ Redundanz
- Entwurfsziele bei der Benutzung
  - Maximierung der Anzahl von Disks, die parallel benutzt werden,
  - Minimierung der redundanten Datenmenge und
  - Minimierung des Overheads
- Unterschiedliche Stufen (Auswahl)
  - RAID 0: nicht-redundante Verteilung,
  - RAID 1: Spiegelung, keine Performanzverbesserung,
  - RAID 5: block-interleaved distributed parity

### Raid-Systeme, Beispiel

### Überblick RAID Systeme

### RAID-Systeme aus Sicht des DBS

- RAIDs realisieren eine virtuellen Disk, das DBS hat keine Kenntnis/Kontrolle über Verteilungsstrategie
- Gefahr: gegenläufige Optimierungskriterien, Performanz-/Redundanzverhältnis beachten

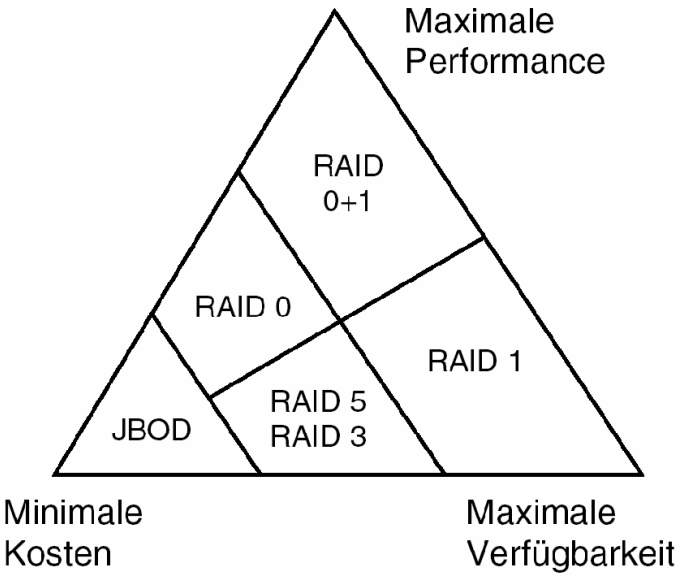



Abbildung 14: Raid-Systeme Leistungsaspekte

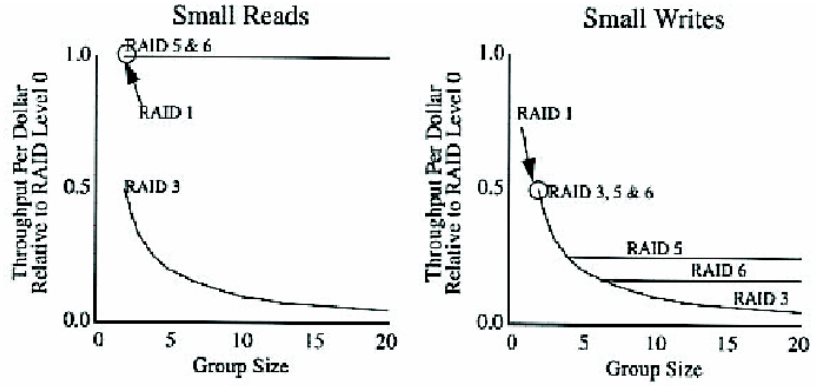



Abbildung 15: Kosten von RAID-Systemen

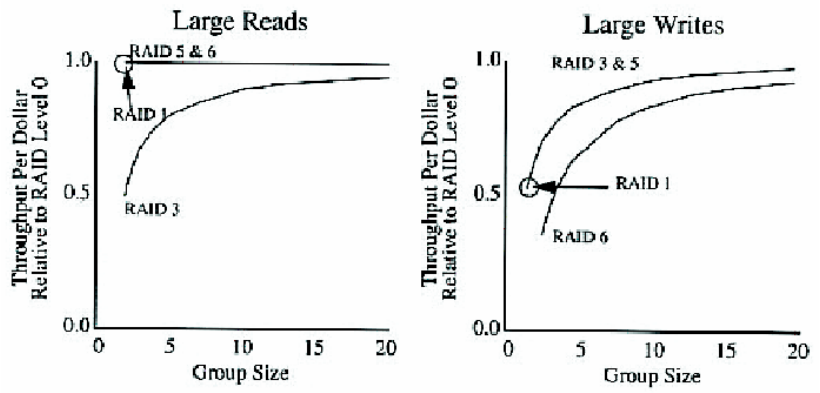



Abbildung 16: Kosten von RAID-Systemen

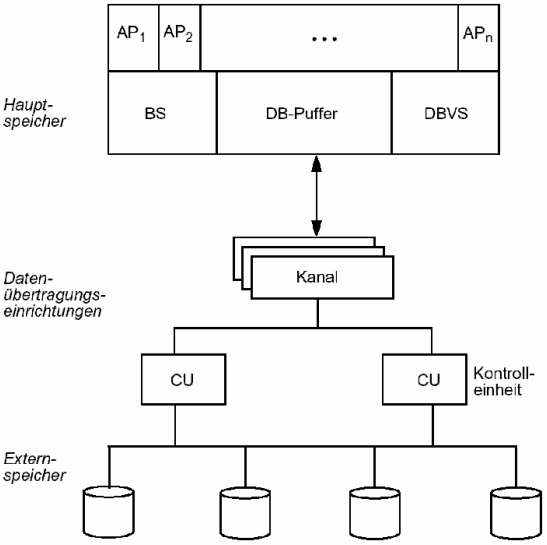



Abbildung 17: Modell einer zweistufigen Speicherhierarchie

### RAID-Systeme aus Sicht des DBS

### Zusammenfassung E/A-Komponenten

- Implementierung eines DBS wird durch verfügbare Speicher bestimmt.
- Speicherhierarchie: funktionieren wegen der Lokalität der Datenzugriffe.
- Fünf-Minuten-Regel für Datenpuffer zwischen den Speicherebenen.
- Aufbau von Festplatten: sequentieller Zugriff ist deutlich schneller als wahlfreier Zugriff
- RAID: Kompromiß zwischen parallelem Zugriff, Fehlertoleranz und Redundanz. RAID-Systeme sind kein Ersatz für Sicherungskopien.

# 4 Aufbau des Speichersystems

### Vereinfachtes Modell einer zweistufigen Speicherhierarchie

## 4.1 Dateien und Blöcke

### Abbildung von Dateien und Blöcken 1/2

- Anforderungen
  - Verwaltung externer Speichermedien
  - Verbergen von Geräteeigenschaften
  - Abbildung von physischen Blöcken
  - Kontrolle des Datentransports von / zum Hauptspeicher
  - ggf. Realisierung einer mehrstufigen Speicherhierarchie
  - Fehlertoleranzmaßnahmen (stabiler Speicher, Spiegelplatten etc.)
- Gründe für ein Dateikonzept
  - selektive Aktivierung von Dateien: on/offline-Problem
  - dynamische Definition
  - temporäre Dateien
  - Einsatz unterschiedlich schneller Speichermedien
  - kürzere Adreßlängen

### Abbildung von Dateien und Blöcken 2/2

- DB-Speicher entspricht Menge von Dateien
- Abbildungsfunktion: Blocknummer → {Zylinder, Spur, Sektor}
- Eigenschaften der oberen Schnittstelle
  - Menge durchnummerierter Blöcke innerhalb von Dateien
  - Lese- und Schreibzugriff über die Blocknummer
  - Blockzugriff kann auf Lesefehler führen, Blöcke können alte oder ungültige Daten enthalten
- Dateien repräsentieren externe Speichermedien für AP in einer geräteunabhängigen Weise. Das Dateisystem verdeckt die Eigenschaften physischer Speichergeräte und bietet als abstrakte Sicht Dateien, die als lange Byte-Vektoren behandelt werden können.

### Dateisystem

- Dateisystem
  - erlaubt eine Reihe von Operationen (vereinfachte Definition)
  - ist als lokales Dateisystem oder als eigenständiger Datei-Server realisiert
  - verwaltet typischerweise einen hierarchischen Namensraum
  - Namensgebung
- Operationen auf Dateien
  - create/delete: Anlegen und Löschen einer Datei Welche Parameter sind für create erforderlich?
  - Verarbeiten: open/close Welche Aktionen sind bei open erforderlich?
  - Lesen/Schreiben STATUS read (FILE, file_address, memory_address, length); STATUS write (FILE, file_address, memory_address, length);

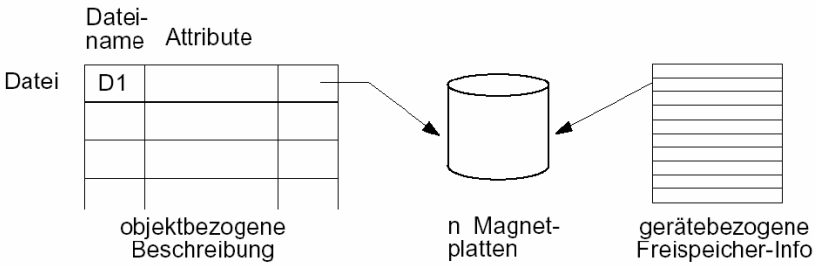



Abbildung 18: Dateikatalog

## 4.2 Realisierung eines Dateisystems

### Realisierung eines Dateisystem 1/2

- Dateikatalog für alle Dateien (Urdatei)
- Objekt-bezogene Beschreibung durch Dateideskriptor
  - Dateiname, OwnerID, ...
  - Zugriffskontrolliste
  - eitinfo. über Erzeugung, Zugriff, Archivierung, ...
  - Dateigröße, Externspeicherzuordnung, ...

### Realisierung eines Dateisystem 2/2

- Freispeicherverwaltung für Externspeicher
  - formatierte Bitlisten, hierarchische Struktur
- Anlegen/Reservieren von Speicherbereichen
  - Erstzuweisung und Erweitern
- Einheit des physischen Zugriffs: Block
  - Blöcke variabler Länge?
  - feste Blocklänge pro Datei
  - verkettete Ein/Ausgabe, wichtig für hohe E/A-Leistung

### Dateiorganisationsformen 1/2

- Dateisystem
  - verwaltet erzeugte Dateien, Lese-/Schreibzugriff
  - hält Deskriptor für jede Datei

### Dateiorganisationsformen 2/2

- Organisationsformen
  - Datei in Einfügereihenfolge (entry-sequenced)
    - Einfügen an Dateiende, Satzadresse ist relative Byte-Adresse (RBA)
    - sequentieller Zugriff oder direkter Zugriff über RBA

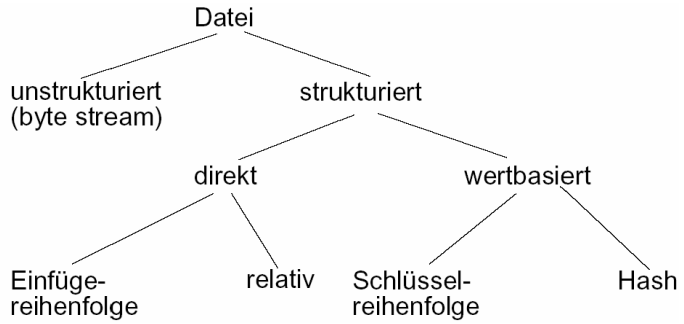



Abbildung 19: Dateiorganisationsformen

  - relative Datei: Organisation als ARRAY OF RECORDS
  - wertbasierte Dateien erlauben Zugriff über Satzschlüssel
  - Datei in Schlüsselreihenfolge (key-sequenced)
    * Speicherung der Sätze in Schlüsselreihenfolge
    * direkter Zugriff über Schlüssel und sortiert-sequentieller Zugriff
  - Hash-Datei: direkter Zugriff über Schlüsseltransformation

## 4.3 Blockzuordnung bei Magnetplatten

### Blockzuordnungsverfahren

- Statische Datei-Zuordnung
  - direkte Adressierung
  - minimale Zugriffskosten
  - keine Flexibilität
- Dynamische Extent-Zuordnung
  - Adressierung über eine kleine Tabelle
  - geringe Zugriffskosten
  - mäßige Flexibilität
- Dynamische Block-Zuordnung
  - Adressierung über eine große Tabelle
  - hohe Zugriffskosten
  - maximale Flexibilität

### Blockzuordnung durch Extent-Tabellen

### Existierende Dateisysteme

- Unix-Dateisystem, (ext2, ext3, ext4 unter Linux)
- Reiser-FS
- IBM Journaling File System JFS

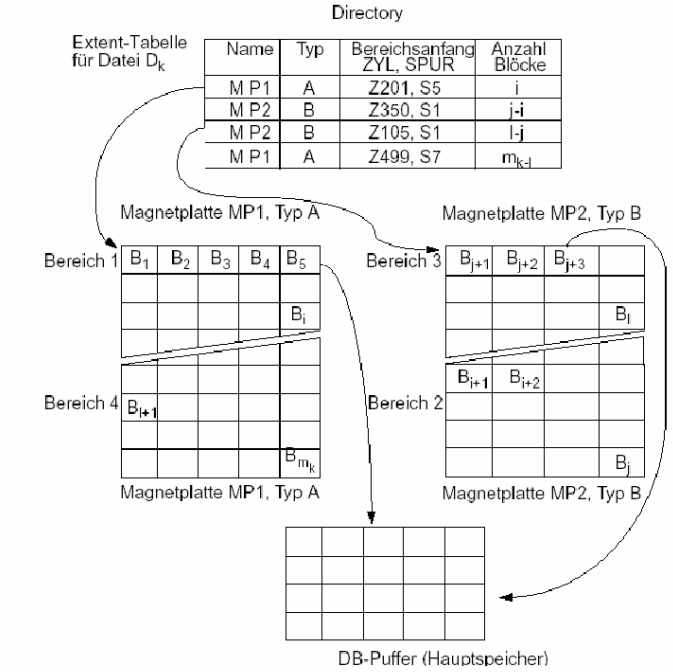



Abbildung 20: Blockzuordnung durch Extent-Tabellen

- SGIs Journaling File System XFS
- Solaris ZFS
- Oracle Cluster File System OCFS
- MS Windows: FAT32, NTFS

## 4.4 Kontrolle der E/A Operationen

### Fehlertoleranz beim Lesen-/Schreiben von Blöcken 1/3

- Einfaches Lesen
  - Lesen kann durch transiente Fehler gestört werden
  - Zusatzmaßnahme: erfolgloses Lesen wird n-mal wiederholt (sicheres Lesen)
- Einfaches Schreiben
  - Schreiben des Blocks als atomare Aktion (ganz oder überhaupt nicht) kann nicht garantiert werden
  - Schreiben kann durch transiente und dauerhafte Fehler zu falschen Resultaten führen ⇒ mögliche Schreibfehler im Katalog
- Sicheres Schreiben (read-after-write)
  - Nach einfachem Schreiben wird Block sofort wieder gelesen und Originalblock verglichen.
  - Wiederholen bis Block erfolgreich geschrieben
  - Schreiben gesichert gegen transiente Fehler: dauerhafte Fehler können jedoch zu falschen Resultaten führen.

### Fehlertoleranz beim Lesen-/Schreiben von Blöcken 2/3

- Stabiles Schreiben (duplexed write)
  - Jeder Block hat eine Versionsnummer, die bei jedem stabilen Schreiben erhöht wird.
  - Jeder Block wird in festgelegter Reihenfolge in zwei verschiedene Slots $S_j, S_k$ geschrieben
  - Prinzip: Stabiler Speicher
    - Annahme: ein Block wird nicht in einen falschen Slot geschrieben, sonst ist read-after-write erforderlich.
  - Lesen von Block $B_i$ erfolgt erst von Slot $S_j$. Falls es erfolgreich ist, wird angenommen, daß es sich um die jüngste gültige Version von $B_i$ handelt.
  - Falls das Lesen von Slot $S_j$ scheitert, wird Slot $S_k$ gelesen.
  - Da ein Systemausfall nur einen Schreibvorgang unterbrechen kann, ist bei Wiederanlauf stets eine Version des Blockes verfügbar. ⇒ Schreiben ist somit gesichert gegen dauerhafte Fehler. Daß beide Versionen nicht lesbar sind, gilt als unerwartet.

### Fehlertoleranz beim Lesen-/Schreiben von Blöcken 3/3

- Schreiben mit Logging (logged write)
  - Nach dem Lesen wird ein Block $B_i$ mit seinem alten Inhalt auf einen sicheren Platz $L$ geschrieben.
  - Nach Aktualisierung wird $B_i$ mit einem einfachen Schreiben auf seinen alten Platz zurückgeschrieben (ggf. read-after-write).
  - Falls das Schreiben erfolgreich war, wird die Kopie von $B_i$ auf $L$ nicht mehr gebraucht.

## 4.5 Segmente und Seiten

### Abbildung von Segmenten und Seiten

- Realisierung eines Segmentkonzeptes
  - Ermöglichung verzögerter Einbringstrategien
  - selektive Einführung von zusätzlichen Attributen, z. B. zur Erhöhung der Fehlertoleranz
  - Segmente als Einheiten des Sperrens, der Recovery und der Zugriffskontrolle
  - bei geeigneter Abbildung auf Dateien bleiben Vorteile des Dateikonzeptes erhalten
- DB-Pufferverwaltung
  - Bereitstellen und Freigeben von Seiten im DB-Puffer
  - Vorbereitung von E/A-Anforderungen an die Dateiverwaltung
  - Optimierung von Ersetzungsstrategien
  - Unterstützung von Segmenten verschiedenen Typs
  - neue Aufgaben: Verwaltung von Seiten variabler Länge und von langen Objekten
- ⇒ Aufteilung des logischen DB-Adreßraumes in Segmente mit sichtbaren Seitengrenzen

| Eigenschaften \ Segment-Typen | Segment-Typ 1 | Segment-Typ 2 | Segment-Typ 3 | Segment-Typ 4 | Segment-Typ 5 |
|---|---|---|---|---|---|
| Benutzung | öffentlich | privat | privat | privat | privat |
| Lebensdauer | permanent | permanent | permanent | permanent | temporär in Transaktion |
| Öffnen und Schließen | automatisch durch System | | explizit durch Benutzer | | |
| Wiederherstellung im Fehlerfall | automatisch durch System | | explizit durch Benutzer | kein Wiederherstellungsmechanismus | |

Abbildung 21: Klassifikation von Segmenttypen

### Abbildung von Segmenten und Seiten

- Operationen an der oberen Schnittstelle
  - Fix $P_i$: reserviert Seite im Hauptspeicher mit logischer Blocknummer
  - Unfix $P_i$: gibt Seite wieder frei, was nicht automatische heißt das sie aus dem Hauptspeicher verdrängt wird.
- Abbildungsfunktion
  - Seitennummer $\leftrightarrow$ Blocknummer
  - Hauptspeicher $\leftrightarrow$ Externspeicher
- Aufgaben
  - Lies den Block mit Seite $P_i$
  - Ersetzen eines alten Blocks
- Eigenschaften der oberen Schnittstelle
  - Linearer Adreßraum, aufgeteilt in Seiten fester Länge
  - Innerhalb eines Segmentes können die Module der nächsthöheren Schicht frei adressieren wie in einem virtuellen Adreßraum
  - Ein DB-Segment ist nicht-flüchtig (wenn nicht explizit anders vereinbart)

### Segmenttypen in einem DBMS 1/2

### Segmenttypen in einem DBMS 2/2

Beispiele für Verwendung der Segmenttypen

- Typ 1: Katalog, Schema-Information, Log alle gemeinsam benutzbaren DB-Teile
- Typ 2: Teile der DB, die für bestimmte Benutzer bzw. Benutzergruppen reserviert sind
- Typ 3: Lokale Kopien von Teilen der DB (Sichten) für einzelne Benutzer (Snapshots)
- Typ 4: Hilfsdateien für Benutzerprogramme
- Typ 5: Temporärer Speicher z. B. für Sortierprogramme

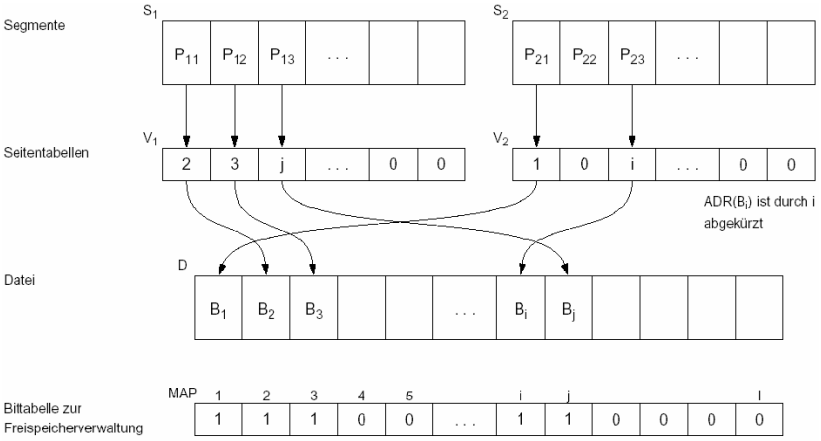



Abbildung 22: Direktes Einbringen von Änderungen

**Warum sichtbare Seiten und Segmente?**

- Explizite Abbildung auf Blöcke und Dateien erhöht Fehlertoleranz

- DB-Puffer mit Satzschnittstelle?
  - lineare Hauptspeicherabbildung: Probleme unsichtbarer Seitengrenzen
  - spanned record facility
  - benachbarte Seiten müssen im DB-Puffer benachbart angeordnet sein

  ⇒ erhebliche Abbildungs- und Ersetzungsprobleme

- Sichtbare Seitengrenzen an der DB-Puffer-Schnittstelle

## 4.6 Seitenabbildung

**Indirekte Seitenzuordnung**

**Indirekte Seitenzuordnung**

## 4.7 Einbringstrategien für Änderungen

**Speicherverwaltung beim Schattenspeicher-Verfahren**
⇒ Abbildung von Seitennr. → Blocknr. mit Hilfe der Seitentabelle

**Speicherverwaltung beim Schattenspeicher-Verfahren**
⇒ Auch die Seitentabelle muß im linearen Adreßraum untergebracht und auf Blöcke abgebildet werden.

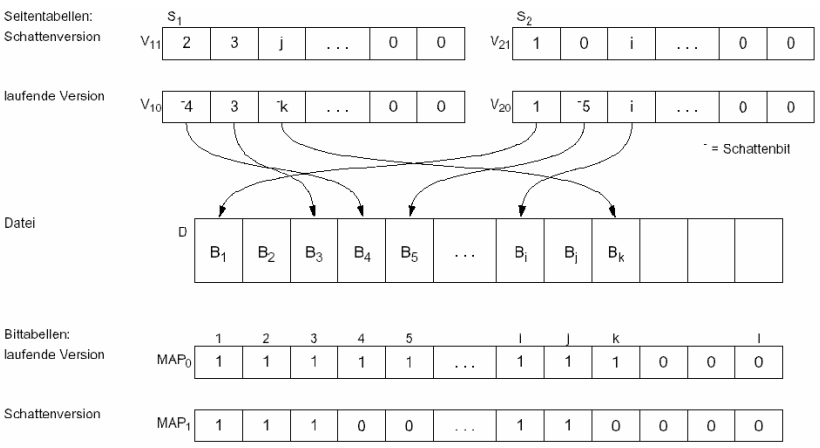



Abbildung 23: Schattenspeicher-Verfahren, Verzögertes Einbringen von Änderungen

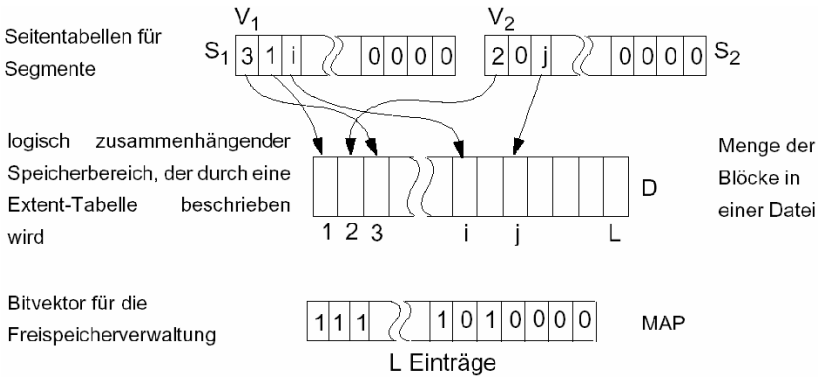



Abbildung 24: Prinzip der indirekten Seitenzuordnung

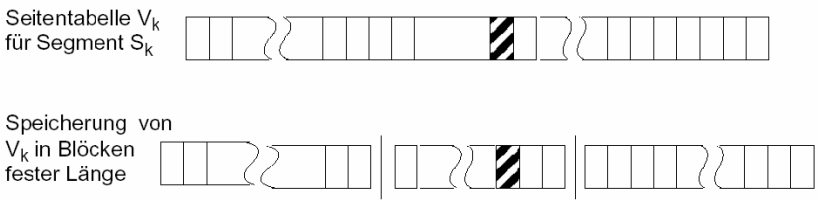



Abbildung 25: Zerlegung von Seitentabelle $V_k$ in einzelne Blöcke

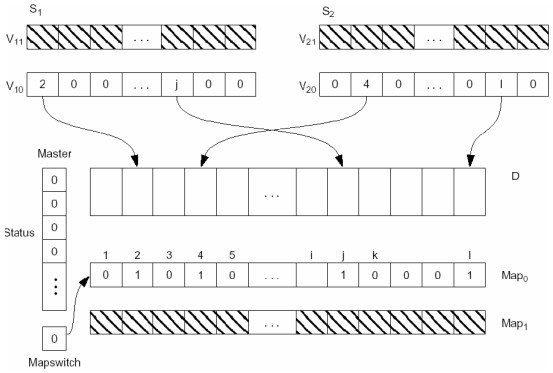



Abbildung 26: Schattenspeicher-Verfahren: Ausgangssituation

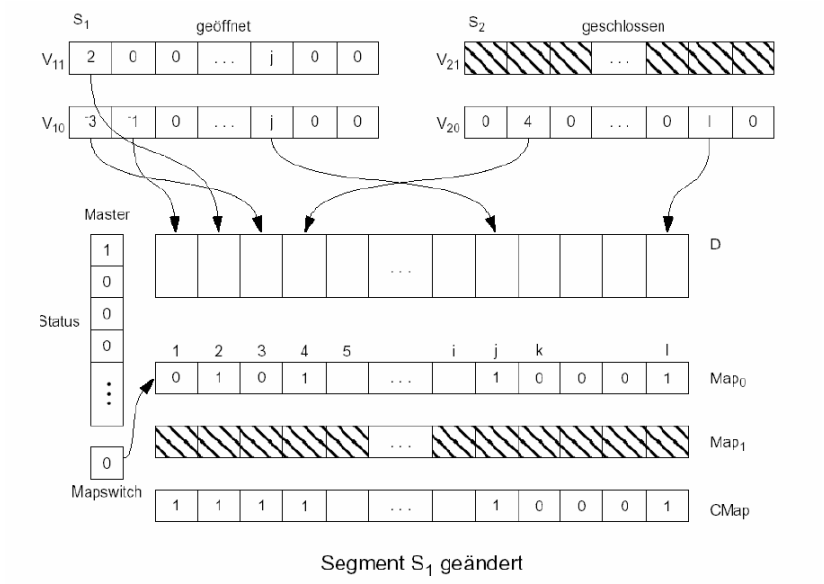



Abbildung 27: Schattenspeicher-Verfahren: Änderungsbetrieb

### Schattenspeicher-Verfahren: Ausgangssituation 1/2

- Implementierung fehlertoleranter Systeme: Erforderliche Speicherbereiche & Datenstrukturen
- Schraffierte Strukturen sind in Ausgangssituation nicht benutzte Speicherbereiche, die erst nach Eröffnung für Änderungsbetrieb erforderlich werden.

### Schattenspeicher-Verfahren: Ausgangssituation 2/2

- Status (i) enthält Eröffnungszustand für Segment i (hier: alle Segmente geschlossen).
- MAPSWITCH zeigt, welche der beiden (gleichberechtigten) Tabellen Map0 und Map1 das aktuelle Verzeichnis belegter Blöcke enthält.
- Wenn ein Segment geschlossen ist, erfüllt sein Inhalt bestimmte, von höheren Schichten kontrollierte Konsistenzbedingungen.

### Schattenspeicher-Verfahren: Änderungsbetrieb 1/2

### Schattenspeicher-Verfahren: Änderungsbetrieb 2/2

- Auf einer Blockmenge D können gleichzeitig mehrere Segmente für den Änderungsbetrieb geöffnet sein.
- Map0, Map1 und CMAP beziehen sich auf die Blockmenge. CMAP enthält für alle Segmente die mit Schatten- bzw. aktuellen Seiten belegten Blöcke.
- Einer Seite wird nur bei der erstmaligen Änderung nach Eröffnen des Segmentes ein neuer Block zugewiesen.

### Schattenspeicher-Verfahren: Funktion 1/2

- Wenn Segment $k$ für Änderungen geöffnet werden soll, sind folgende Schritte auszuführen
  - kopiere $V_{k0}$ nach $V_{k1}$
  - STATUS($k$) := 1
  - Schreibe MASTER in einer ununterbrechbaren Operation aus
  - Lege im Hauptspeicher eine Arbeitskopie CMAP von MAP0 an.
- Wenn eine Seite $P_i$ erstmalig seit Eröffnen des Segments geändert werden soll, sind folgende Aktionen auszuführen
  - Lies Seite $P_i$ aus Block $j = V_{k0}(i)$
  - Finde einen freien Block $j$ in CMAP
  - $V_{k0}(i) = j$
  - Markiere Seite $P_i$ in $V_{k0}(i)$ als geändert
  - Bei weiteren Änderungen von $P_i$ wird Block $j$ verwendet.

### Schattenspeicher-Verfahren: Funktion 2/2

- Beenden eines Änderungsintervalls
  - Erzeuge die Bitliste mit der aktuellen Speicherbelegung in MAP1 (neue Blöcke belegt, alte freigegeben)
  - Schreibe MAP1 (kein Überschreiben von MAP0)
  - Schreibe $V_{k0}$
  - Schreibe alle geänderten Blöcke
  - STATUS(k) := 0, MAPSWITCH = 1 (MAP1 ist aktuell)
  - Schreibe MASTER in einer ununterbrechbaren Operation aus.
- Zum Zurücksetzen geöffneter Segmente muß lediglich $V_{k1}$ in $V_{k0}$ kopiert und STATUS(k) auf 0 gesetzt werden.

### Erhaltung der physischen Clusterbildung 1/2

- Wegen der dynamischen Neuzuordnung von Blöcken können beim Schattenspeicherverfahren physische Nachbarschaften von logisch zusammengehörigen Seiten bei Änderungen i.a. nicht aufrechterhalten werden.
- Bsp.: Die Seiten A, B, C, D mögen in allen Transaktionen in eben dieser Reihenfolge berührt werden
- Die Änderung von A → A' und C → C' führt zu

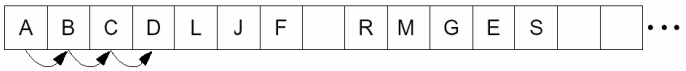



Abbildung 28: Ausgangssituation

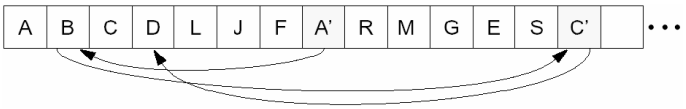



Abbildung 29: Änderung

### Schattenspeicher-Verfahren: Erhaltung der physischen Clusterbildung 2/2

- Eine Verbesserung kann durch Einteilung der Datei in physische Cluster der Größe $p$ und der Segmente in logische Cluster der Größe $l$ sowie deren gegenseitige Zuordnung erreicht werden. Beispiel mit $p = 6, l = 4$
- Bei Plattenspeichern wird ein physischer Cluster im allgemeinen ein Zylinder sein.

### Bewertung des Schattenspeicher-Verfahrens 1/2

- Vorteile
  - Rücksetzen auf letzten konsistenten Zustand einfach
  - flexiblere Schreibprotokolle für Log-Daten: Pufferung bis Umschalten auf neuen Zustand möglich
  - logisches Logging möglich, da stets operationskonsistenter Zustand verfügbar
  - bei katastrophalem Fehler ist Wahrscheinlichkeit höher, einen brauchbaren Zustand der DB zu rekonstruieren

### Bewertung des Schattenspeicher-Verfahrens 2/2

- Nachteile
  - Hilfsstrukturen (MAP und Seitentabellen Vi) werden so groß, daß Blockzerlegung notwendig wird
  - Seitentabellen $V_i$ belegen 0.1-0.2% der DB-Größe, was bei großen DB's ($\geq n$ GB) zu hohem Anteil von Seitenfehlern im DB-Puffer für Zugriffe auf $V_i$ führen kann
  - periodische Sicherungspunkte erzwingen Ausschreiben des gesamten DB-Puffers
  - physische Clusterbildung logisch zusammengehöriger Seiten wird beeinträchtigt bzw. zerstört
  - zusätzlicher Speicherplatz für Doppelbelegung; lange Batch-(Änderungs-)Programme werden dadurch schlecht unterstützt

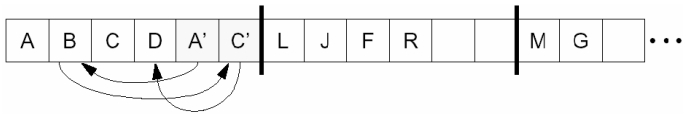



Abbildung 30: Verbesserung

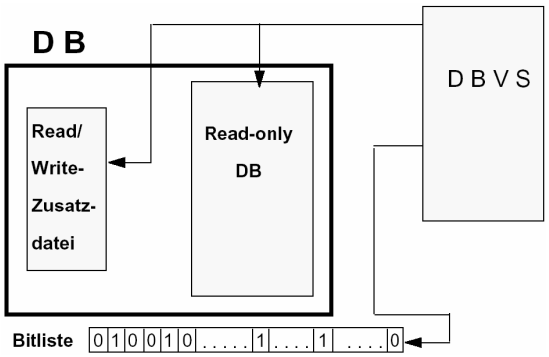



Abbildung 31: Zusatzdatei-Verfahren: Prinzip

**Zusatzdatei-Verfahren: Idee**

- Wenn eine Datei für Änderungsbetrieb geöffnet wird, wird eine zusätzliche, temporäre Datei angelegt, in welche während des Änderungsintervalles alle modifizierten Blöcke geschrieben werden. Die eigentliche Datei wird dabei nicht verändert. Am Ende des Änderungsintervalles werden alle veränderten Blöcke aus der temporären in die permanente Datei kopiert. Das Einbringen der Änderungen erfolgt also verzögert.
- Das wesentliche Problem besteht darin, während des Änderungsbetriebes für eine gegebene Seiten-Nr. zu entscheiden, ob die aktuelle Version in der temporären oder permanenten Datei steht.

**Zusatzdatei-Verfahren: Prinzip**

- Nutzung einer Bitliste, die anzeigt, ob eine Seite möglicherweise geändert wurde
- Hashabbildung erlaubt Begrenzung des Speicherplatzbedarfs

**Zusatzdatei-Verfahren: Bloom-Filter**

- Arbeitssweise
  - Bitstring B der Länge M wird im Hauptspeicher gehalten
  - Bei Änderung eines Satzes wird der Satzschlüssel $S_i$ über Hash-Funktionen $H()$ auf Bitstring der Länge M abgebildet
  - Die Hash-Funktionen zur Satzabbildung setzen einige Bits auf 1, die dann auch in B auf 1 gesetzt werden $B \leftarrow B \vee H(S_i)$
- Aufsuchen von Satz $S_i$
  - Erzeugen des charakteristischen Bitstrings in temporärem Bitstring $T \leftarrow H(S_i)$
  - AND-Operation von T und B in $Erg$ speichern, d.h. $Erg \leftarrow T \wedge B$
  - Auswertung
    - wenn alle Bits aus $T$ in $Erg$ gesetzt sind, d.h. $T = Erg \Rightarrow$ VIELLEICHT ist Satz $S_i$ in Zusatz-Datei,
    - ansonsten, d.h. $T \neq Erg \Rightarrow$ NEIN, Satz $S_i$ ist nicht in Zusatz-Datei

**Zusatzdatei-Verfahren: Beispiel**

$\Rightarrow$ Wenn $(B \wedge T) = T$, dann Antwort VIELLEICHT !

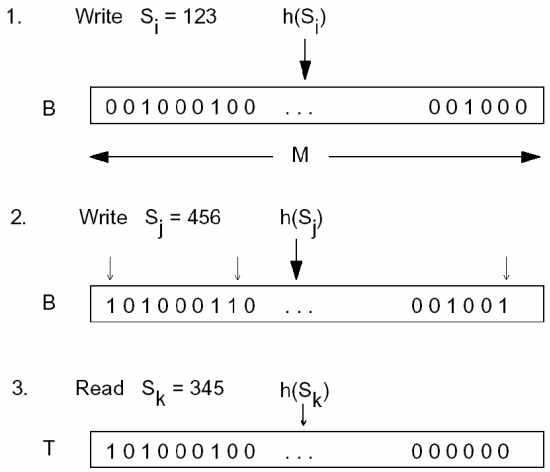



Abbildung 32: Zusatzdatei-Verfahren: Beispiel

## Zusammenfassung 1/2

- Speicherzuordnungsstrukturen erfordern effizientes Dateikonzept
  - viele Dateien variierender, nicht statisch festgelegter Größe
  - Wachstum und Schrumpfung erforderlich
  - permanente und temporäre Dateien
- empfohlene Datei-Eigenschaften
  - direkter und sequentieller Blockzugriff
  - Blockgröße pro Datei definierbar
  - Blockzuordnung über dynamische Extents ⇒ Blockzuordnungsverfahren von UNIX untauglich für große Dateien
- Segmentkonzept
  - erlaubt die Realisierung zusätzlicher Attribute für die DB-Verarbeitung (Recovery, Clusterbildung für Relationen usw.)

## Zusammenfassung 2/2

- zweistufige Abbildung
  - von Segment/Seite auf Datei/Block und diese auf Slots der Magnetplatte erlaubt Einführung von Abbildungsredundanz durch verzögertes Einbringen
- Verzögerte Einbringstrategien
  - sind teurer als direkte, besitzen jedoch implizite Fehlertoleranz
  - sie belasten den Normalbetrieb zugunsten der Recovery
- Direkte Einbringstrategie (update-in-place)
  - einfach zu implementieren
  - keine Zusatzkosten zur Ausführungszeit für die Seitenzuordnung
  - Fehlertoleranz nur durch explizite Logging- und Recovery-Funktionen

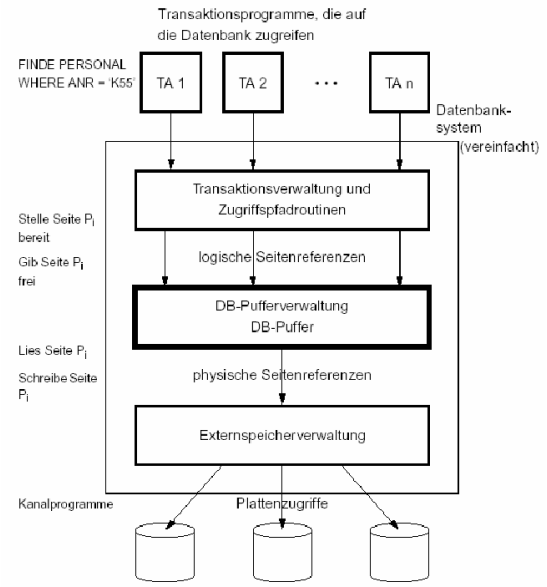



Abbildung 33: DB-Pufferverwaltung in DBS

# 5 Pufferverwaltung

## 5.1 Rolle der DB-Pufferverwaltung

### Rolle der DB-Pufferverwaltung in DBS

### Vergleich mit Betriebssystemfunktionen

Können Dateipuffer des Betriebssystems als DB-Puffer eingesetzt werden?

1. Zugriff auf Dateipuffer ist teuer (SVC: supervisor call)
2. DB-spezifische Referenzmuster können nicht mehr gezielt genutzt werden BS-Ersetzungsverfahren sind z. B. nicht auf zyklisch sequentielle oder baumartige Zugriffsfolgen abgestimmt
3. Normale Dateisysteme (DVS) bieten keine geeignete Schnittstelle für Prefetching. In DBVS ist aufgrund von Seiteninhalten oder Referenzmustern eine Voraussage des Referenzverhaltens möglich; durch Prefetching läßt sich in solchen Fällen eine Leistungssteigerung erzielen
4. Selektives Ausschreiben von Seiten zu bestimmten Zeitpunkten (z. B. für Logging) nicht immer möglich in existierenden Dateisystemen

⇒ DBVS muß eigene Pufferverwaltung realisieren

## 5.2 Eigenschaften von DB-Referenzstrings

### Eigenschaften von DB-Referenzstrings

Typische Referenzmuster in DBS

1. Sequentielle Suche

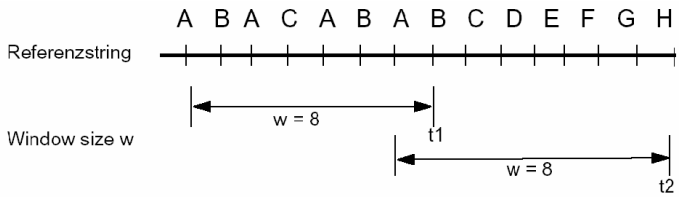



Abbildung 34: Working-Set-Modell

2. Hierarchische Pfade

3. Zyklische Pfade

### Lokalität

- erhöhte Wiederbenutzungswahrscheinlichkeit für gerade referenzierte Seiten (gradueller Begriff)
- grundlegende Voraussetzung für
  - effektive DB-Pufferverwaltung (Seitenersetzung)
  - Einsatz von Speicherhierarchien
- Wie kann man Lokalität messen? → Working-Set-Modell
  - Working Set Size: $W(t1, w = 8) = 3$, W(t2,w=8)=8
  - Aktuelle Lokalität: $AL(t, w) = \frac{W(t,w)}{w}$
  - Durchschnittliche Lokalität: $L(w) = \frac{\sum_{t=1}^{n} AL(t,w)}{n}$ (n=Länge des Referenzstrings)

### LRU-Stacktiefenverteilung 1/2

- Maß für die Lokalität (präziser als Working-Set-Ansatz)
- LRU-Stack enthält alle bereits referenzierten Seiten in der Reihenfolge ihres Zugriffsalters
- Bestimmung der Stacktiefenverteilung
  - pro Stackposition wird Zähler geführt
  - Rereferenz einer Seite führt zur Zählererhöhung für die jeweilige Stackposition

  ⇒ Zählerwerte entsprechen der Wiederbenutzungshäufigkeit

### LRU-Stacktiefenverteilung 2/2

- Beispiel
- Für LRU-Seitenersetzung kann aus der Stacktiefenverteilung für eine bestimmte Puffergröße unmittelbar die Trefferrate (bzw. Fehlseitenrate) bestimmt werden

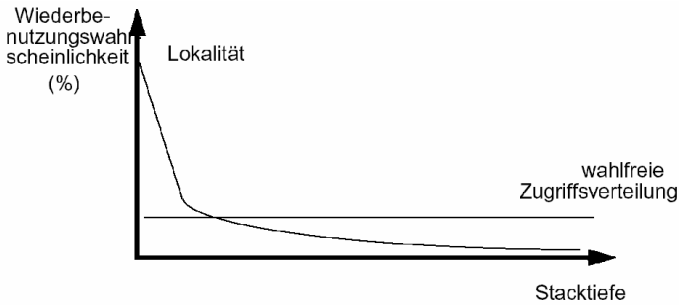



Abbildung 35: LRU-Stacktiefenverteilung

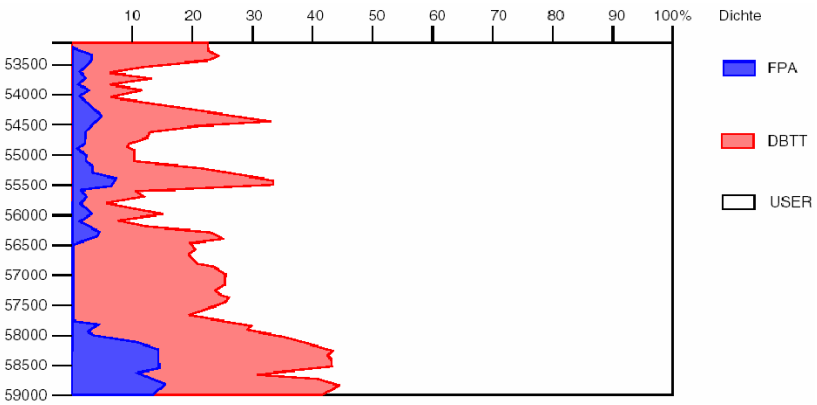



Abbildung 36: Referenzdichte-Kurven

**Beispiel: Ermittlung der Stacktiefenverteilung**

**Referenzdichte-Kurven**

- Anteil der Seitentypen
  - Freispeicherverwaltung (FPA) = 0,1%
  - Adressumsetzung (DBTT) = 6,1%
  - Daten (USER) = 93,8%

## 5.3 Speicherzuteilung im DB-Puffer

**Speicherzuteilung im DB-Puffer**

- Partitionierungsmöglichkeiten
  - eigener Pufferbereich pro Transaktion
  - TA-Typ-bezogene Pufferbereiche
  - Seitentyp-bezogene Pufferbereiche
  - DB-(Partitions)spezifische Pufferbereiche

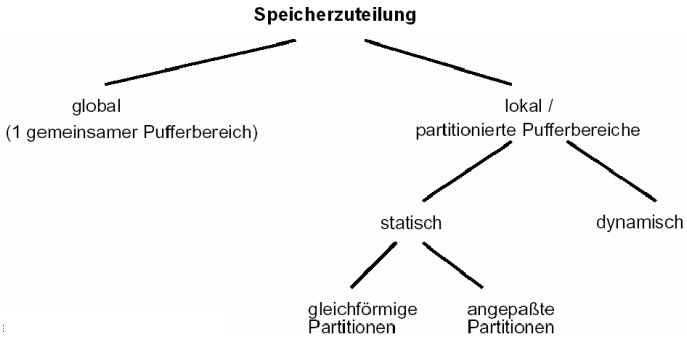



Abbildung 37: Speicherzuteilung im DB-Puffer

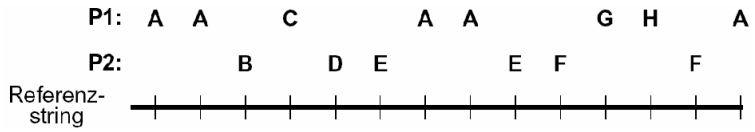



Abbildung 38: Working-Set-Ansatz

### Dynamische Pufferallokation – Working-Set-Ansatz

- pro Pufferpartition P soll Working-Set im Puffer bleiben
- Seiten, die nicht zum Working-Set gehören, können ersetzt werden
- bei Fehlseitenbedingung muß Working-Set bekannt sein, um Ersetzungskandidat zu bestimmen
  - Fenstergröße pro Partition: $w(P)$
  - Referenzzähler pro Partition: $RZ(P)$
  - letzter Referenzzeitpunkt für Seite $i$: $LRZ(P, i)$
  - ersetzbar sind solche Seiten, für die $RZ(P) - LRZ(P, i) \geq w(P)$
- Fenstergröße kritischer Parameter → Thrashing

## 5.4 Suche im DB-Puffer

### Suche im DB-Puffer

- Sequentielles Durchsuchen der Pufferrahmen
  - sehr hoher Suchaufwand
  - Gefahr vieler Paging-Fehler bei virtuellen Speichern
- Nutzung von Hilfsstrukturen (Eintrag pro Pufferrahmen)
  1. unsortierte oder sortierte Tabelle
  2. Tabelle mit verketteten Einträgen
     - geringere Änderungskosten
     - Anordnung in LRU-Reihenfolge möglich
  3. Suchbäume (z. B. AVL-, m-Weg-Bäume)

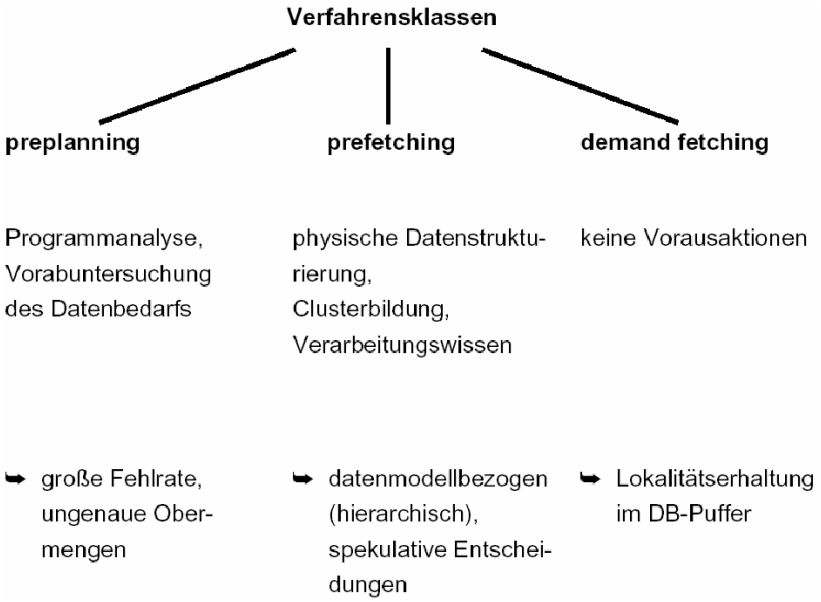



Abbildung 39: Klassifikation von Seitenersetzungsverfahren

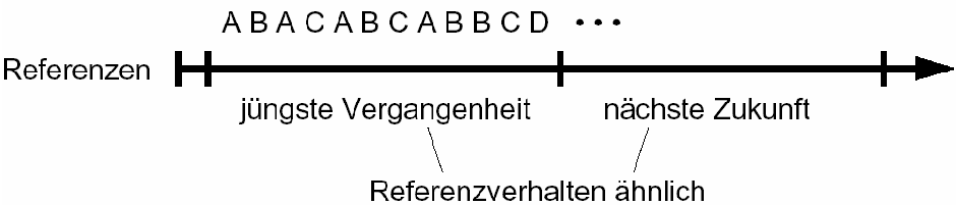



Abbildung 40: Grundannahme bei Ersetzungsverfahren

4. Hash-Tabelle mit Überlaufketten
   - beste Lösung

## 5.5 Seitenersetzungsverfahren

### Seitenersetzungsverfahren: Klassifikation

### Seitenersetzungsverfahren: Grundannahme

### Referenzverhalten und Ersetzungsverfahren

- typischerweise hohe Lokalität: Optimierung durch Ersetzungsverfahren
- manchmal Sequentialität oder zufällige Arbeitslast (RANDOM-Referenzen)
- Prinzipielle Zusammenhänge, die die Fehlseitenrate bestimmen

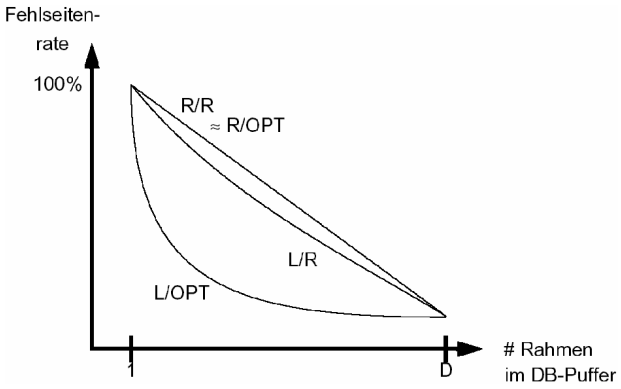



Abbildung 41: Fehlseitenrate versus Puffergröße

### Behandlung geänderter Seiten im DB-Puffer 1/2

- Ersetzung einer geänderten Seite erfordert ihr vorheriges (synchrones) Zurückschreiben in die DB ⇒ Antwortzeitverschlechterung
- Abhängigkeit zur Ausschreibstrategie
- FORCE: alle Änderungen einer Transaktion werden spätestens beim EOT in die DB zurückgeschrieben (write-through)
  - + stets ungeänderte Seiten zur Ersetzung vorhanden
  - + vereinfachte Recovery (nach Rechnerausfall sind alle Änderungen beendeter TA bereits in der DB )
  - - hoher E/A-Overhead
  - - starke Antwortzeiterhöhung für Änderungstransaktionen

### Behandlung geänderter Seiten im DB-Puffer 2/2

- NOFORCE: kein Durchschreiben der Änderungen bei EOT (verzögertes Ausschreiben, deferred write-back)
  - Seite kann mehrfach geändert werden, bevor ein Ausschreiben erfolgt
    - geringerer E/A-Overhead
    - bessere Antwortzeiten
- vorausschauendes (asynchrones) Ausschreiben geänderter Seiten: erlaubt auch bei NOFORCE, vorwiegend ungeänderte Seiten zu ersetzen
  - synchrone Schreibvorgänge in die DB lassen sich weitgehend vermeiden

### Kriterien für die Auswahl der zu ersetzenden Pufferseite

### Least-Frequently-Used (LFU)

- Führen eines Referenzzählers pro Seite im DB-Puffer
- Ersetzung der Seite mit der geringsten Referenzhäufigkeit
- Alter einer Seite wird nicht berücksichtigt

| Verfahren | Kriterien | | | |
|---|---|---|---|---|
| | Alter | letzte Referenz | Referenz-häufigkeit | andere Kriterien |
| OPT | - | - | - | Vorauswissen |
| RANDOM | - | - | - | --- |
| LFU | | | | |
| FIFO | | | | |
| LRU | | | | |
| CLOCK | | | | |
| GCLOCK | | | | |
| LRD(V1) | | | | |
| LRD(V2) | | | | |
| LRU-K | | | | |

Abbildung 42: Kriterien für Ersetzungsstrategien

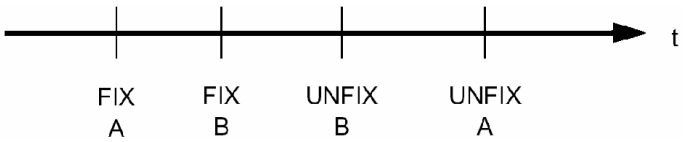



Abbildung 43: Least-Recently-Referenced und Least-Recently-Unfixed

### FIFO (First-In First-Out)

- die älteste Seite im DB-Puffer wird ersetzt
- Referenzierungsverhalten während Pufferaufenthaltes wird nicht berücksichtigt ⇒ nur für strikt sequentielles Referenzverhalten geeignet

### Least-Recently-Used (LRU)

- Beispiel (Puffergröße 4)
  1. Referenz der Seite C
  2. Referenz der Seite E
- Unterscheidung zwischen Least-Recently-Referenced und Least-Recently-Unfixed

### CLOCK (Second Chance)

- Erweiterung von FIFO
- Referenzbit pro Seite, das bei Zugriff gesetzt wird
- Ersetzung erfolgt nur bei zurückgesetztem Bit (sonst erfolgt Zurücksetzen des Bits)
- annähernde Berücksichtigung des letzten Referenzierungszeitpunktes

### GCLOCK (Generalized CLOCK)

- pro Seite wird Referenzzähler geführt (statt Bit)
- Ersetzung nur von Seiten mit Zählerwert 0 (sonst erfolgt Dekrementierung des Zählers und Betrachtung der nächsten Seite)
- Verfahrensparameter
  - Initialwerte für Referenzzähler
  - Wahl des Dekrementes
  - Zählerinkrementierung bei erneuter Referenz
  - Vergabe von seitentyp- oder seitenspezifischen Gewichten

### Least-Reference-Density (LRD V1)

- Referenzdichte: Referenzhäufigkeit während eines bestimmten Referenzintervalls
- Variante 1: Referenzintervall entspricht Alter einer Seite Berechnung der Referenzdichte
  - Globaler Zähler GZ: Gesamtanzahl aller Referenzen
  - Einlagerungszeitpunkt EZ: GZ-Wert bei Einlesen der Seite
  - Referenzzähler RZ
  - Referenzdichte $RD(j) = \frac{RZ(j)}{GZ - EZ(j)}$

### Least-Reference-Density (LRD V2)

- Variante 2: konstante Intervallgröße
  - periodisches Reduzieren der Referenzzähler, um Gewicht früher Referenzen zu reduzieren
  - Reduzierung von RZ durch Division oder Subtraktion
    - $RZ(i) = {}^{RZ(i)}/_{K1}$, falls $K1 > 1$ oder
    - $RZ(i) = \begin{cases} RZ(i) - K2 & \text{, falls } RZ(i) - K2 \geq K3 \\ K3 & \text{sonst } (K2 > 0, K3 \geq 0) \end{cases}$

### LRU-K 1/2

- Aufzeichnung der K letzten Referenzzeitpunkte (pro Seite im DB-Puffer)
  - Aufwendigere Aufzeichnung gewährleistet aktuelle Ersetzungsinformation
  - Methode benötigt kein explizites Altern über Tuning-Parameter wie LRD-V2
- Gegeben sei bis zum Betrachtungszeitpunkt t der Referenzstring $r_1, r_2, \ldots, r_t$.
- Rückwärtige K-Distanz $b_t(P, K)$ ist die in Referenzen gemessene Distanz rückwärts bis zur $K$-jüngsten Referenz auf Seite $P$
  - $b_t(P, K) = x$, wenn $r_t - x$ den Wert $P$ besitzt und es genau $K - 1$ andere Werte $i$ mit $t - x < i \leq t$ und $r_i = P$ gab.
  - $b_t(P, K) = \infty$, wenn $P$ nicht wenigstens $K$ mal in $r_1, r_2, \ldots, r_t$ referenziert wurde

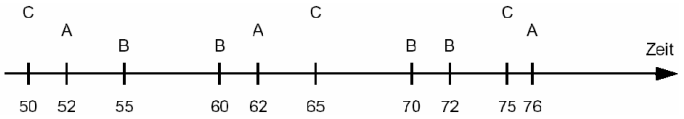



Abbildung 44: LRU-K, Beispiel ($K = 4$)

### LRU-K 2/2

- Beispiel ($K = 4$)
- Zur Ersetzung genügt es, die $b_t(P_i, K)$ der Pufferseiten zu berücksichtigen!
  - Sonderbehandlung für Seiten mit weniger als $K$ Referenzen erforderlich
  - Wie hängt LRU-K mit LRD zusammen? Approximation der Referenzdichte?
- LRU-2 (d.h. $K = 2$) stellt i. allg. beste Lösung dar
  - ähnlich gute Ergebnisse wie für $K > 2$, jedoch einfachere Realisierung
  - Verfahren reagiert schneller auf Referenzschwankungen als bei größeren $K$

## 5.6 Ersetzungsverfahren - Einbezug von Kontextwissen

### Hot-Set-Modell 1/4

- Ausnutzung von Kontextwissen bei mengenorientierten Anforderungen ⇒ Verbesserung in relationalen DBS möglich
- Zugriffspläne durch Optimizer
  - Zugriffscharakteristik/Menge der referenzierten Seiten kann bei der Erstellung von Plänen vorausgesagt/abgeschätzt werden
  - Zugriffsmuster enthält immer Zyklen/Loops (mindestens Kontrollseite Datenseite, nested loop join etc.)
  - Kostenvoranschläge für Zugriffspläne können verfügbare Rahmen berücksichtigen
  - Bei Ausführung wird die Mindestrahmenzahl der Pufferverwaltung mitgeteilt

### Hot-Set-Modell 2/4

- Hot-Set: Menge der Seiten im Referenzzyklus
  - Prinzipieller Verlauf der Fehlseitenrate (FSR) bei speziellen Operationen

### Hot-Set-Modell 3/4

- Hot Point
  - abrupte Veränderung in der FSR, z. B. verursacht durch Schleife beim Verbund
- Hot Set Size (HSS)
  - größter Hot Point kleiner als der verfügbare DB-Puffer
  - Optimizer berechnet HSS für die verschiedenen Zugriffspläne (Abschätzung der #Rahmen)
- Beispiel 2

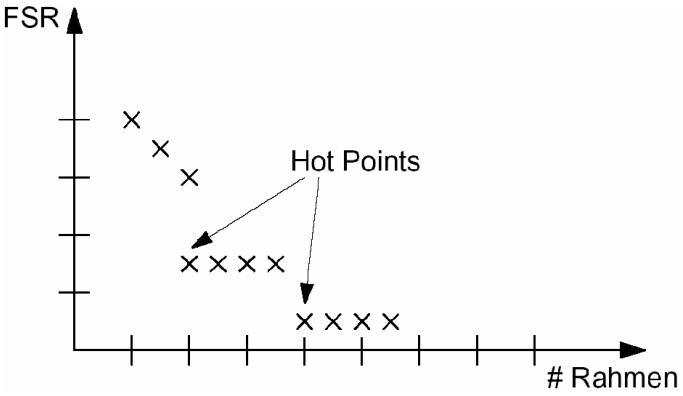



Abbildung 45: Beispiel Hot-Set-Modell

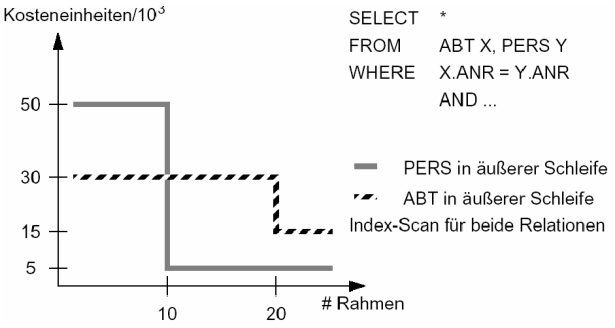



Abbildung 46: Beispiel 2 Hot-Set-Modell

### Hot-Set-Modell 4/4

- Anwendungscharakteristika
  - Berücksichtigung der HSS in Gesamtkosten
  - Auswahl abhängig von verfügbarer DB-Puffergröße
  - Bindung zur Laufzeit möglich

### Zusammenfassung 1/2

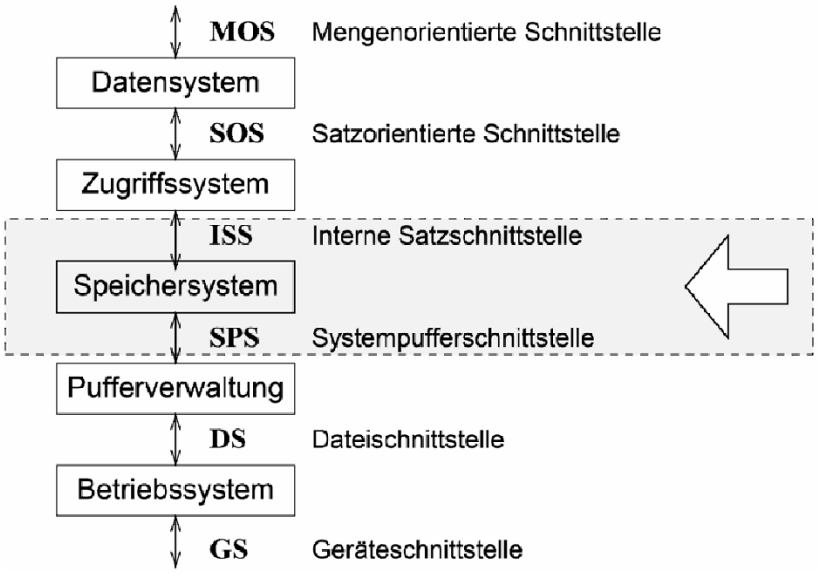
- Referenzmuster in DBS sind Mischformen
  - sequentielle, zyklische, wahlfreie Zugriff
  - Lokalität innerhalb und zwischen Transaktionen
  - bekannte Seiten mit hoher Referenzdichte
- Ohne Lokalität ist jede Optimierung der Seitenersetzung sinnlos (RANDOM)
- Suche im Puffer durch Hash-Verfahren
- Speicherzuteilung
  - global → alle Pufferrahmen für alle Transaktionen (Einfachheit, Stabilität ...)
  - lokal → Sonderbehandlung bestimmter Transaktionen/Anfragen/ DB-Bereiche
- Behandlung geänderter Seiten: NOFORCE, asynchrones Ausschreiben

### Zusammenfassung 2/2

- Seitenersetzungsverfahren
  - zu genaue Verfahren sind schwierig einzustellen (instabil)
  - Nutzung mehrerer Kriterien: Alter, letzte Referenz, Referenzhäufigkeit
  - CLOCK → LRU, aber einfachere Implementierung
  - GCLOCK, LRD, LRU-K relativ komplex
  - LRU-2 guter Kompromiss; vorletzter Referenzzeitpunkt bestimmt Opfer
- Erweiterte Ersetzungsverfahren
  - Nutzung von Zugriffsinformationen des Query-Optimierers (Hot-Set-Modell)
  - Berücksichtigung von Prioritäten

# 6 Speicherungsstrukturen und Satzverwaltung

**Satzverwaltung**

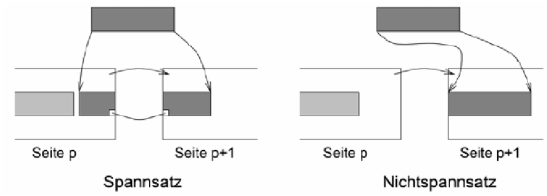



## 6.1 Abbildung von Sätzen auf Seiten

**Abbildung von Sätzen auf Seiten**

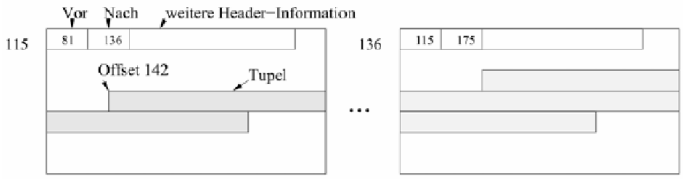
- wichtig
  - bisher: Seite fester Länge als Verarbeitungseinheit
  - jetzt: Datensatz beliebiger Länge als Verarbeitungseinheit
- Entkopplung von
  - systemvorgegebenen Verarbeitungseinheiten (Seiten), (physischer Satz)
  - Datenstrukturen einer Anwendung (Sätze), (logischer Satz)
- Abbildungsfunktion:

- Erinnerung
  - Spannsatz
  - Nicht-Spannsatz

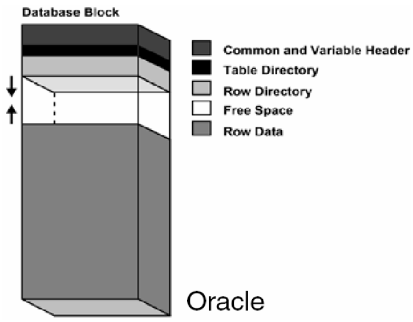


### Seitenorganisation

- Verkettung
  - Seiten sind durch doppelt verkettete Listen verbunden
  - Aufzeichnung freier Seiten: Freispeicherverwaltung




- Seiten-Header
  - Infos über Vorgänger- und Nachfolger-Seite
  - eventuell auch Nummer der Seite
  - Infos über Typ der Sätze (Table Directory)
  - freier Platz
- Row Directory
  - TID-Verweise




## 6.2 Aufbau und Speicherungsstrukturen für Sätze

### Die Verarbeitungseinheit Satz

- Definition Satz
  - Zusammenfassung von Daten, die zu einem Gegenstand, einer Person, einem Sachverhalt usw. einer Anwendung gehören und Eigenschaften des Gegenstands wiedergeben.
  - Sätze sind aus Feldern zusammengesetzt.
- Merke:
  - Die Strukturierung eines Satzes ist für die Speicherverwaltung auf dieser Ebene irrelevant.
  - In dieser Schicht ist ein Satz nur eine Bytefolge, deren Länge aber nicht mehr vom System, sondern von der Anwendung bestimmt wird!
  - Eine Datei ist auf dieser Abstraktionsebene eine (lineare) Folge von Sätzen fester oder variabler Länge.
- Aufgabe des Record-Managers
  - physische Abspeicherung / Organisation von Sätzen in Seiten
  - Operationen: Lesen, Einfügen, Modifizieren, Löschen

### Format eines Datensatzes

- Anforderungen für Verarbeitungseffizienz und -flexibilität
  - Jeder Satz wird durch ein Satzkennzeichen (SKZ oder OID) identifiziert
  - möglichst platzsparende Speicherung (Speicherökonomie)
  - Erweiterbarkeit des Satztyps muss im laufenden Betrieb möglich sein
  - einfache Berechung der satzinternen Adresse des n-ten Feldes (bei Zugriff auf nur einen Teilaspekt der Sätze)
- Satzbeschreibung
  - Satz- und Zugriffspfadbeschreibung im Katalog
  - besondere Methoden der Speicherung
    - Blank-/Nullunterdrückung, Zeichenverdichtung, kryptographische Verschlüsselung, Symbol für undefinierte Werte
  - Organisation
    - n Satztypen pro Segment
    - m Sätze verschiedenen Typs pro Seite
    - Satzlänge $<$ Seitenlänge

### Format eines Feldes

- Attribut- /Feldbeschreibung
  - Name (meist Unterschied zwischen internem Feldnamen und externem Attributnamen)
  - Charakteristik (fest, variabel, multipel)
  - Länge
  - Typ (alpha-numerisch, numerisch, gepackt)
  - besondere Methoden bei der Speicherung (z.B. Nullenunterdrückung, Zeichenverdichtung, Kryptographie etc.)
  - ggf. Symbol für den undefinierten Wert (falls nicht als Segment- oder Systemkonstante global definiert).
- Die Formatbeschreibung steuert alle Operationen auf Sätzen
- Satztypen
  - Typischerweise gibt es viele Sätze mit gleichem Aufbau, d.h. die gleichen Felder haben eine einmalige Beschreibung im Datenwörterbuch
  - Satztyp: Menge von Sätzen mit gleicher Struktur bekommt einen Namen
  - Jeder Satz muss beim Abspeichern einem Satztyp zugeordnet werden (Sätze ohne Typ sind nicht erlaubt).

### Satztypen

- Länge der Sätze eines Satztyps
  - fest, wenn alle Felder feste Länge haben oder bei Feldern variabler Länge immer die Maximallänge reserviert wird.
  - variabel sonst

- Problem
  - In welcher Seite wird ein Satz abgelegt, und wie kann anschließend dieser Satz wieder gefunden werden, auch wenn zwischenzeitlich etliche andere Sätze gelöscht und eingefügt wurden?
  - siehe Satzadressierung !!!
- Annahmen
  - variable Satzlänge (allgemeinerer Fall)
  - Reihenfolge der Abspeicherung muss nicht Reihenfolge des Einfügens sein
  - direkter Zugriff auf einzelne Sätze über ihre Satzadresse
  - Ein Satz sollte in einer Seite ablegbar sein: $L_R \leq L_S - L_{SK}$ (Standard)
  - Mehrere Satztypen pro Seite sollen möglich sein.

**Speicherungsstrukturen für Sätze**

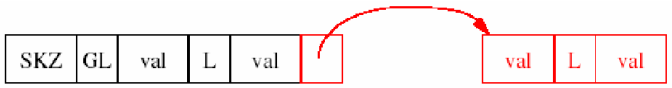
- Konkatenation von Feldern fester Länge
  - speicheraufwendig
  - unflexibel
- Zeiger im Vorspann
  - unflexibel

**Speicherungsstrukturen für Sätze**

- eingebettete Längenfelder
  - dynamische Erweiterung möglich
  - aber: zusätzliches Wissen notwendig: $f5|v|v|f6|f2|v|$
- eingebettete Längenfelder mit Zeigern
  - Adresse des n-ten Attributes wird berechnet

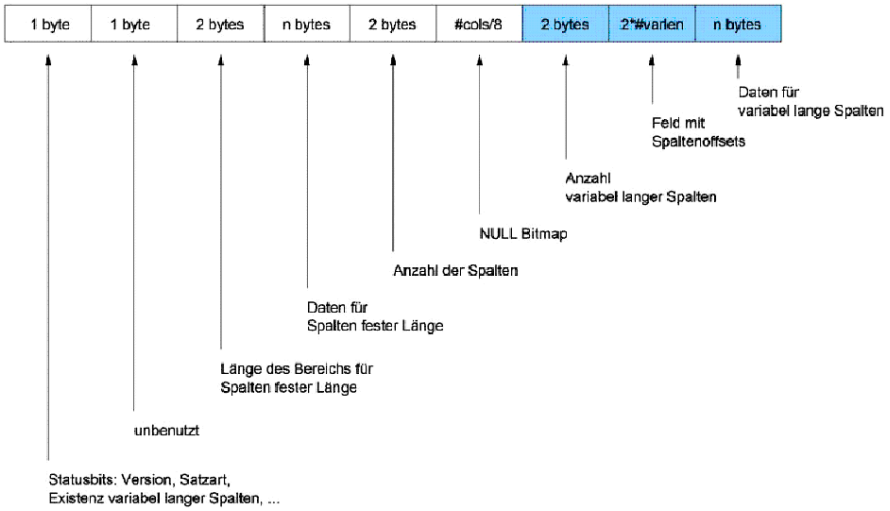
### Speicherungsstrukturen bei variabler Länge

- Speicherungsstrukturen bei variabler Länge
  - dynamisches Wachstum/variable Länge
    - Ausdehnung und Schrumpfung in einer Seite
    - Überlaufschemata
    - Garbage Collection
  - strikt zusammenhängende Speicherung von Sätzen
    - evtl. häufige Umlagerung bei hoher Änderungsfrequenz
    - Vorteile für indirekte Adressierungsschemata
- Aufspaltung des Satzes

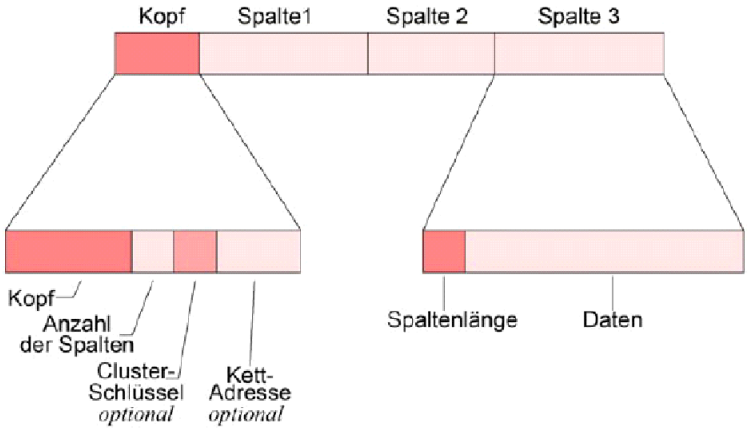



  - Ordnung nach Referenzhäufigkeiten
  - Verbesserung der Clusterbildung
  - Wiederholter Überlauf möglich
  - wird unvermeidlich bei der Einbeziehung von Attributen vom Typ TEXT oder BILD

### Beispiel: SQL Server - Aufbau von Datensätzen




### Beispiel: Oracle - Aufbau von Datensätzen



- Kettadresse für Row-Chaining
  - Verteilung von Verkettung zu großer Datensätze (> 255 Spalten) über mehrere Blöcke
  - row id = (data object identifier, data file identifier, block identifier, row identifier)

#### Oracle Datendefinition

- Syntax für Tabellendefinition

```
create table tabelle ( ...)
pctfree 10 pctUSED 40
storage ( initial 10MB, next 2MB, minextents 1,
maxextents 20, pctincrease 0, freelists 3 )
tablespace USER_TBLSPACE;
```

  - initial, next: Größe des ersten bzw. der weiteren Extents (Default: 5 Blöcke)
  - minextents, maxextents: Anzahl der min. bzw. max. zu allokierenden Extents
  - pctincrease: prozentuale Vergrößerung der nachfolgenden Extents (0: gleich große Extents)
  - freelists: Anzahl der Freispeicherlisten (insb. für paralleles Einfügen)
  - tablespace: Zuordnung zum Tablespace
  - pctfree: Datenblockanteil, der nicht für insert-Operationen genutzt werden soll (Reservebereich für update); Default 10
  - pctused: Grenze, bei der ein zuvor bis zu pctfree gefüllter Block wieder für insert genutzt werden darf; Default 40

#### Blockung

- Typischerweise passen mehrere Sätze in eine Seite (Satzlängen 100 – 1000 Bytes)
- Blockungsfaktor: Anzahl der Sätze pro Seite
- Annahme: keine blockübergreifenden Sätze (spanned records), d.h. jeder Satz wird vollständig in einer Seite abgelegt
  - feste Satzlänge
    - Blockungsfaktor aus Seitengröße und Satzlänge berechenbar
    - meist ungenutzter Speicherplatz am Ende einer Seite
  - variable Satzlänge
    - Blockungsfaktor ändert sich von Seite zu Seite

### 6.3 Satzadressierung

#### Satzadressierung

- Problem
  - langfristige Speicherung der Datensätze
  - Vermeiden von Technologieabhängigkeiten
  - Unterstützung von Migration u. a.
- Satzadressen

  - Satzadressen werden beim Einfügen von Sätzen vergeben und können später zum Zugriff auf die Sätze verwendet werden.

- Ziele der Adressierungstechnik
  - schneller, möglichst direkter Satzzugriff
  - hinreichend stabil gegen geringfügige Verschiebungen (Verschiebungen innerhalb einer Seite ohne Auswirkungen)
  - seltene oder keine Reorganisationen

- Allgemeine Form einer Satzadresse
  - DBID, SID, TID und ggf. Relationenkennzeichnung (RID)
  - Relation vollständig in einem Segment gespeichert: TID DBID, SID im DB-Katalog
  - Relation in mehreren Segmenten: SID, TID

#### Überblick über Adressierungsverfahren

#### Verfahren externspeicherbasierter Satzadressierung

- Laufende Nummer des Satzes
  - Instabil! Die laufende Nummer, und somit die Satzadresse ändert sich bei Einfügungen und Löschvorgängen, sowie bei Änderungen in der Abspeicherungsreihenfolge.

- Blocknummer und Byte-Position innerhalb des Blocks
  - Instabil! Ändert ein Satz innerhalb des Blocks seine Länge, müssen i.allg. die anderen Sätze verschoben werden.
  - Wird der Satz selbst zu lang für den Block, so muß er in einen anderen Block verlegt werden; dann ändert sich auch die Blocknummer.

- Adressierung in Segmenten
  - logisch zusammenhängender Adressraum
  - direkte Adressierung (logische Byte-Adresse)
  - $\Rightarrow$ instabil bei Verschiebungen
  - $\Rightarrow$ deshalb indirekte Adressierung

### 6.4 Zuordnungstabelle

#### Zuordnungstabelle 1/2

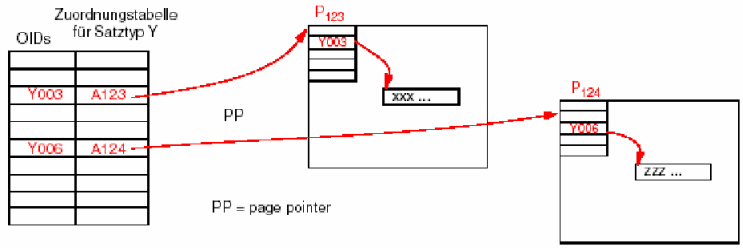
- Satzadressierung über Zuordnungstabelle (vollständige Indirektion)

- Verwaltung eines Felds in aufeinanderfolgenden Seiten des Segments, das zu jeder Satznr. die Seitennr. angibt.
  - Einfügen eines Satzes: es wird grundsätzlich eine neue Satznummer durch das DBS vergeben
  - Löschen eines Satzes: der Eintrag der entsprechenden Satznummer wird als ungültig gekennzeichnet
  - Zugriff auf Satz: erfordert zwei Seitenzugriffe: einen für das Feld, einen für die Seite mit dem Satz selbst

- Verlagerung eines Satzes
- in andere Seite: nur Eintrag des Satzes wird geändert; Satznummer bleibt unverändert - Satz ist also über Satznummer weiterhin auffindbar
- innerhalb einer Seite: Eintrag im Feld muß auch geändert und wieder auf Platte geschrieben werden (zusätzliche E/A-Operation)

- die Zuordnungstabelle kostet selbst einigen Speicherplatz

### Zuordnungstabelle 2/2

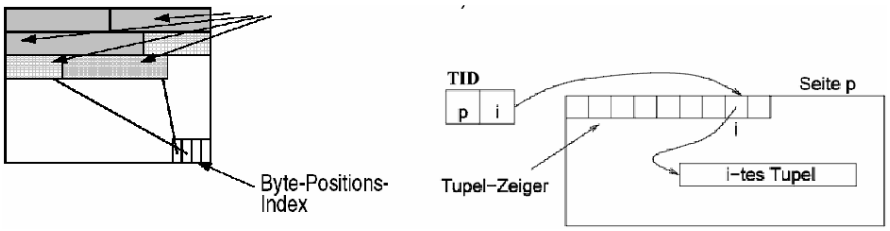



- Merke: der DBK ist eine nicht-sprechende Adresse (a la Telefonnummer)
  - DataBase-Key wird gebildet aus einer Satztypbezeichnung r und einer Folgenummer $f$. Beide, $r$ und $f$, identifizieren den Satz während seiner Lebenszeit in der DB.
  - Es wird auf die Seite $P_k$ im Segment $S_i$ verwiesen.
- Problem: Wo wird die Zuordnungstabelle abgespeichert?
  - am Anfang ⇒ Wie erweitern?
  - am Ende? ⇒ Wie den Datenbereich erweitern?
  - in einem eigenen Segment?

## 6.5 TID (ROWID)-Konzept

### TID (ROWID)-Konzept 1/2

- Satzadressierung über Indirektion innerhalb eines Blocks
  - Array mit Byte-Positionen der Sätze in diesem Block
  - Adresse ist das Paar bestehend aus Blocknummer und Index in diesem Array (TID = Tuple IDentifier)
  - Für den Zugriff auf einen Satz wird nur ein Blockzugriff benötigt.
  - Struktur eines Blocks: einzelne Datensätze (Belegung erfolgt vom Ende des Blocks aus)

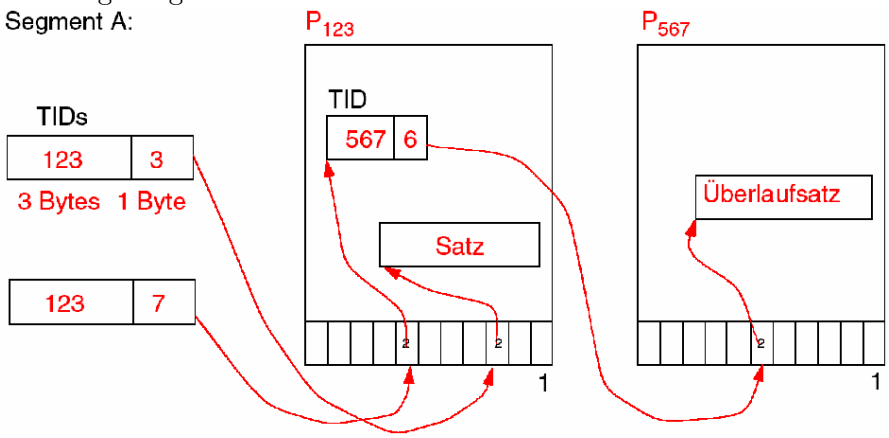


## TID (ROWID)-Konzept 2/2

- Löschen eines Satzes
  - der entsprechende Eintrag des Block-Arrays wird als ungültig gekennzeichnet
  - Alle anderen Sätze im selben Block können verschoben werden, um den freien Platz zu maximieren es ändern sich nur ihre Anfangsadressen im Block-Array.
  - Alle Satzadressen bleiben stabil.
- Update-Operation auf einen Datensatz
  - es kann sich die Länge eines Datensatzes verändern !!!
  - Datensatz schrumpft oder wird größer (ohne Überlauf):
    - Alle Sätze werden innerhalb des Blocks verschoben und der Byte-Positionsindex wird angepaßt.
  - Datensatz wird größer und der freie Platz im Block reicht nicht mehr für die Speicherung des jetzt größeren Datensatzes (Überlauf)
    - Verschiebung des Datensatzes in einen anderen Block!

## Überlaufbehandlung beim TID-Konzept 1/2

Verlagerung eines Satzes




## Überlaufbehandlung beim TID-Konzept 2/2

- Vorgehen
  - Im alten Block verbleibt an der Stelle des Originalsatzes eine neue Satzadresse, die auf den neuen Block verweist.
  - In diesem (seltenen) Fall müssen also zwei Blöcke gelesen werden.
  - Wird der Satz ein weiteres Mal verlagert, so wird die Satzadresse im ersten Block verändert. Dadurch bleibt es bei maximal einer Indirektion.
  - Die Länge der Überlaufkette ist immer kleiner oder gleich 1, d.h. ein Überlaufsatz darf nicht weiter überlaufen, sondern muss von seiner Hausadresse neu plaziert werden.
- Vorteile
  - Keine Zuordnungstabelle (Umsetztabelle)
  - Ein Satz kann innerhalb einer Seite und über Seitengrenzen hinweg verschoben werden, ohne dass der TID sich ändert.

## 6.6 Freispeicherverwaltung

### Freispeicherverwaltung 1/3

- Situation
  - Beispiel: Wo findet sich ausreichend Platz, um einen neuen Datensatz aufzunehmen?
- Freispeicherverwaltung (FPA, Free Place Administration)
  - In einer Tabelle $F_i$ zum Segment $S_i$ wird für jede Seite $s_k$ angegeben, wieviele Bytes in ihr noch frei sind.
  - $F_i(k) = n \Leftrightarrow$ in Seite $s_k$ des Segmentes $S_i$ sind $n$ Bytes frei.
- Problem
  - Wie groß wird die FPA-Tabelle?
  - Wo (in welchen Seiten eines Segmentes) wird die FPA-Tabelle abgespeichert?

### Freispeicherverwaltung 2/3

- Speicheraufwand für Freispeicherverwaltung mit
  - $L_S$ = Seitenlänge
  - $L_{SK}$ = Länge Seitenkopf (page header) für die beschreibenden Informationen einer Seite
  - $L_F$ = Länge eines Eintrags (im allgemeinen 2 Byte)
- ergibt sich
  - $k = \lfloor (L_S - L_{SK}) / L_F \rfloor$ = Anzahl der Einträge pro Seite
  - $s$ = Anzahl der Seiten im Segment
  - $n = \lceil s/k \rceil$ = belegte Seiten im Segment

### Freispeicherverwaltung 3/3

- Lokation für Freispeicherverwaltung
  - äquidistante Verteilung der Tabellenseiten gemäß $i \cdot k + 1$ mit $i = 0, 1, 2, \ldots, n-1$ d.h. eine Tabellenseite steht vor den $k$ Seiten, für die sie die Freispeicherinformation enthält.
    - Vorteil: Segment kann problemlos erweitert werden.
    - Nachteil: Suche nach freiem Speicher hüpft durch das Segment.
  - Bei direkter Seitenadressierung werden deshalb für FPA-Tabelle üblicherweise die ersten n Seiten eines Segments belegt.
    - Nachteil: Erweiterung des Segments
  - Bei indirekter Seitenadressierung befindet sich die Freispeicherinformation mit in der Seitentabelle

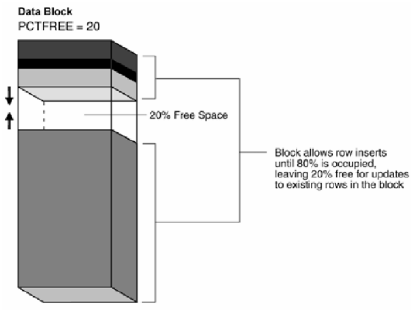
| Seite | 1 | 2 | 3 | . . . | s |
|---|---|---|---|---|---|
| Block | i | j | k | . . . | r |
| Freier Platz | F(1) | F(2) | F(3) | . . . | F(s) |

## 6.7 Belegungsfaktoren (Oracle)

### Belegungsfaktoren (Oracle)

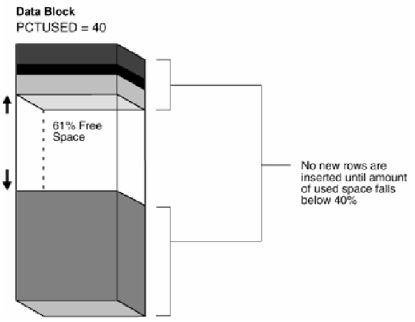
PCTFREE

- Anteil am Block, der für Updates an existierenden Datensätzen freigehalten wird
- erlaubt neue Datensätze bis Füllgrad > 1-PCTFREE

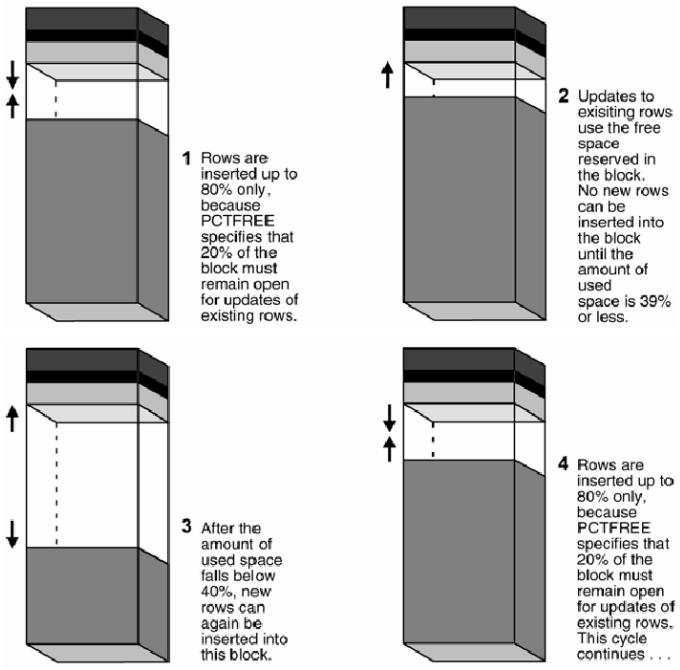



PCTUSED

- Füllgrad eines Blocks, ab dem neue Sätze in den Block wieder eingefügt werden dürfen
- keine neuen Datensätze, falls Füllgrad > PCTUSED




### Zusammenspiel der Belegungskontrollfaktoren



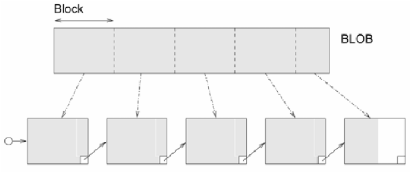
## 6.8 Realisierung langer Felder

### Darstellung und Handhabung langer Felder 1/2

- Anforderungen
  - idealerweise keine Größenbeschränkung
  - Verkürzen, Verlängern und Kopieren
  - Suche nach vorgegebenem Muster, Längenbestimmung, ...
- Erweiterte Anforderungen
  - Effiziente Speicherallokation und -freigabe für Feldgrößen von bis zu 100MB
  - bis 2GB (Sprache, Bild, Musik oder Video)
  - hohe E/A-Leistung: Schreib- und Lese-Operationen sollen E/A-Raten nahe der Übertragungsgeschwindigkeit der Magnetplatte erreichen
- Verarbeitungsprobleme
  - Ist Objektgröße vorab bekannt?
  - Gibt es während der Lebenszeit des Objektes viele Änderungen?
  - Ist schneller sequentieller Zugriff erforderlich?

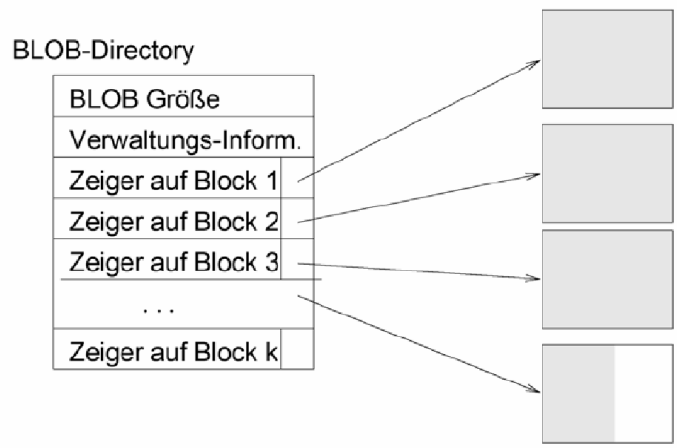
### Darstellung und Handhabung langer Felder 2/2

- Darstellung großer Speicherobjekte
  - besteht potentiell aus vielen Seiten oder Segmenten
  - ist eine uninterpretierte Bytefolge
    - Adresse (OID, object identifier) zeigt auf Objektkopf (header)
  - OID ist Stellvertreter im Satz, zu dem das lange Feld gehört
  - geforderte Verarbeitungsflexibilität bestimmt Zugriffs- und Speicherungsstruktur
- Abbildung auf Externspeicher
  - seitenbasiert (Seite als Einheit)
  - verstreute Sammlung von Seiten
  - segmentbasiert (mehrere Seiten)
  - Segmente fester Größe (EXODUS)
  - Segmente mit einem festen Wachstumsmuster (STARBURST)
  - Segmente variabler Größe (EOS)



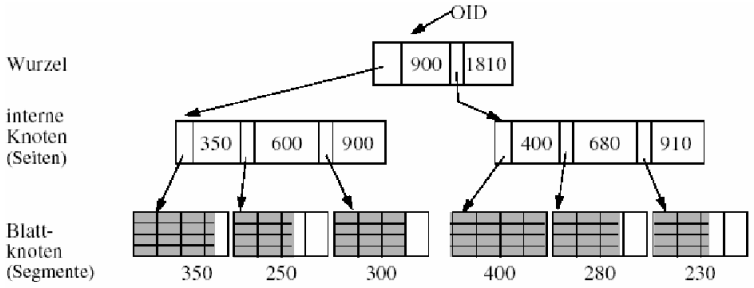
### Lange Felder mit BLOB-Verzeichnis

- Speicherverfahren
  - zentrales BLOB-Verzeichnis mit Zeiger auf die einzelnen Blöcke
  - kann in ursprünglichen Satz eingebettet werden




### Lange Felder als Segmente fester Größe

- Speicherverfahren
  - Daten werden in Seiten / (kleinen) Segmenten fester Größe abgelegt




- Nutzung
  - Baumorganisierte Zugriffsstruktur
    - B*-Baum o.ä: siehe später

### Lange Felder als Segmente fester Größe

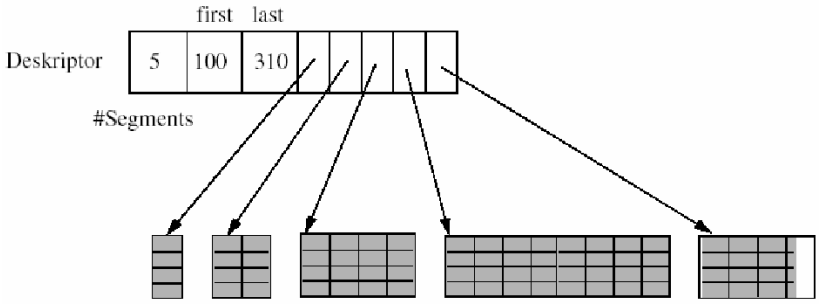
- Baumstruktur
  - Blätter sind Segmente fester Größe (hier 4 Seiten a 100 Bytes)
  - interne Knoten und Wurzel sind Index für Bytepositionen
  - interne Knoten und Wurzel speichern für jeden Kind-Knoten Einträge der Form (Zähler, Seitennummer)
  - Zähler enthält die maximale Bytenummer des jeweiligen Teilbaums (links stehende Seiteneinträge zählen zum Teilbaum)
  - Zähler im weitesten rechts stehenden Eintrag der Wurzel enthält Länge des Objektes
  - Repräsentation sehr langer dynamischer Objekte

  - bis zu 1GB mit drei Baumebenen (selbst bei kleinen Segmenten)
  - Speicherplatznutzung typischerweise 80%

- Bewertung
  - bei bekannter Verarbeitungscharakteristik Wahl geeigneter Segmentgrößen möglich
  - Einfügen von Bytefolgen einfach und überall möglich
  - schlechteres Verhalten bei sequentiellem Zugriff

### Lange Felder als Segmente variabler Größe

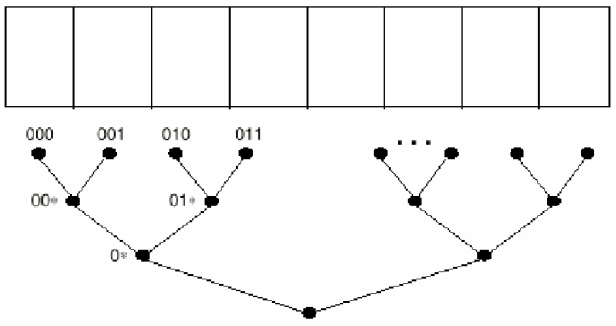
- Prinzipielle Repräsentation
  - Deskriptor mit Liste der Segmentbeschreibungen
  - Langes Feld besteht aus einem oder mehreren Segmenten
  - Segmente, auch als Buddy-Segmente bezeichnet, werden nach dem Buddy- Verfahren in großen vordefinierten Bereichen fester Länge auf Externspeicher angelegt




### Segmentallokation

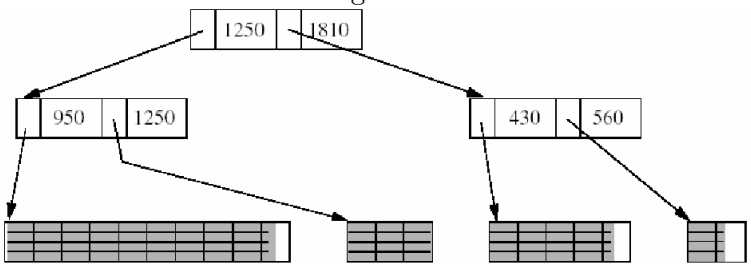
- bei vorab bekannter Objektgröße $G$
  - $G \leq MaxSeg$: es wird ein einzelnes Segment angelegt
  - $G > MaxSeg$: es wird eine Folge maximaler Segmente angelegt; das letzte Segment wird auf verbleibende Objektgröße gekürzt

- bei unbekannter Objektgröße: Allokation von Buddy-Segmenten
  - Wachstumsmuster der Segmentgrößen gemäß: $1, 2, 4, \ldots, 2^n$
  - Seiten werden jeweils zu einem Buddy-Segment zusammengefaßt



### Baumstrukturierte Speicherallokation (variable Segmente)

- Repräsentation
  - Objekt ist gespeichert in einer Folge von Segmenten variabler Größe
  - Segment besteht aus Seiten, die physisch zusammenhängend auf Externspeichern angeordnet sind
  - nur die letzte Seite eines Segmentes kann freien Platz aufweisen



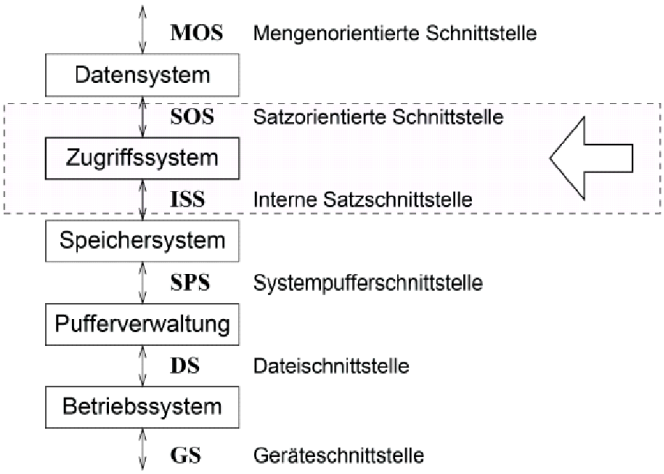

  - ererbt die guten operationalen Eigenschaften der beiden Vorgängeransätze

### Zusammenfassung

- Abbildung von Sätzen
  - Speicherung variabel langer Felder
  - dynamische Erweiterungsmöglichkeiten
  - Berechnung von Feldadressen
- Ziele bei externspeicherbasierter Adressierung
  - Kombination der Geschwindigkeit direkten Zugriffs mit Flexibilität einer Indirektion
  - Satzverschiebungen in einer Seite ohne Auswirkungen
- Alternativen
  - TID-Konzept
  - Zuordnungstabelle

# 7 Eindimensionale Zugriffspfade

### Einordnung in Schichtenarchitektur

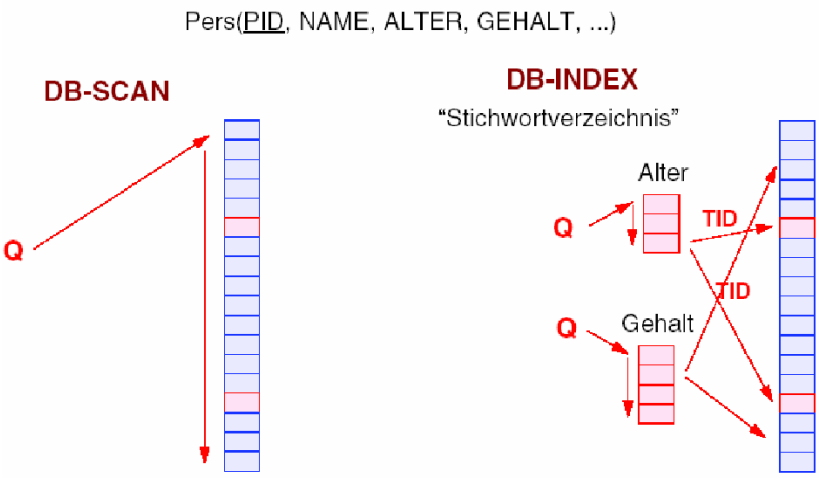



### Motivation von Zugriffspfaden

- Arten von Zugriffen
  - Sequentieller Zugriff auf alle Sätze eines Satztyps (Scan)
  - Sequentieller Zugriff in Sortierreihenfolge eines Attributes
  - Direkter Zugriff über den Primärschlüssel (z.B.: Kennzeichen = HAL-EK 2332)
  - Direkter Zugriff über einen Sekundärschlüssel (z.B. Farbe = silber und Automarke = VW)
  - Direkter Zugriff über zusammengesetzte Schlüssel und komplexe Suchausdrücke (Wertintervalle, ...)
  - Navigierender Zugriff von einem Satz zu einer dazugehörigen Satzmenge desselben oder eines anderen Satztyps
- Anforderungen an Zugriffspfade
  - effizientes (direktes) Auffinden von Datensätzen bzgl. inhaltlichen Kriterien
  - Vermeiden von sequentiellem Durchsuchen aller Datensätze
  - Erleichterung von Zugriffskontrollen durch vorgegebene Zugriffspfade (constraints)
  - Erhaltung topologischer Beziehungen

### Prinzipielles Vorgehen bei der Nutzung eines Index

Einführung eines Zwischenschrittes




### DB-Scan versus Index-Nutzung

- DB-Scan
  - Alle Blöcke müssen gelesen und alle Sätzen in den eingelesenen Seiten müssen hinsichtlich des Suchkriterium untersucht werden
  - Wird von allen DBMS unterstützt
  - Ist ausreichend / effizient bei
    - Kleinen Satztypen (z. B. $< 6$ Seiten)
    - Anfragen mit großen Treffermengen (z. B. $> 1 \ldots 3\%$)
    - DBMS kann Prefetching zur Scan-Optimierung nutzen
- Index
  - Zwei Klassen von Indexstrukturen
    1. Schlüsselwerte werden transformiert um die betreffenden Seiten/Blöcke zu ermitteln
    2. Schlüsselwerte werden redundant in einer eigenen Struktur gehalten und mit dem Suchkriterium verglichen
  - Wenn kein geeigneter Zugriffspfad vorhanden (oder dessen Nutzung nicht ökonomischer) ist, müssen alle Zugriffsarten durch einen SCAN abgewickelt werden

### Allgemeine Beschreibungselemente

- Bestandteile einer Indexstruktur
  - Name des Zugriffspfades
  - Typ des Zugriffspfades
    - Primärschlüssel-Index (Garantie der Eindeutigkeit)
    - Sekundärschlüssel-Index (mehrere Tupel für einen Schlüsselwert)
  - Liste der betreffenden Attributnamen plus potentiell weitere Attribute
  - optional: Sortierung
- Schlüsselzugriff/Schlüsseltransformation
  - Schlüsselzugriff: Zuordnung von Primär- oder Sekundärschlüsselwerten zu Adressen in Hilfsstruktur wie Indexdatei

- Beispiel: indexsequentielle Organisation, B-Baum, KdB-Baum, ...
- Schlüsseltransformation: berechnet Tupeladresse durch Formel aus Primär- oder Sekundärschlüsselwerten (statt Indexeinträgen nur Berechnungsvorschrift gespeichert)
- Beispiel: Hash-Verfahren

### Statische versus Dynamische Strukturen

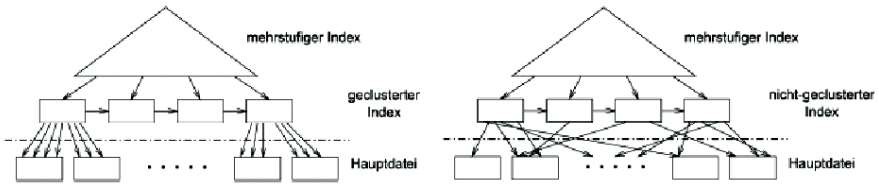
- Statische Zugriffstruktur
  - optimal nur bei bestimmter (fester) Anzahl von verwaltenden Datensätzen
  - Beispiel: Adresstransformation für Personalausweisnummer p von Personen mit p mod 5
  - 5 Seiten, Seitengröße 1 KB, durchschnittliche Satzlänge 200 Bytes, Gleichverteilung der Personalausweisnummern für 25 Personen optimal, für 10.000 Personen nicht mehr ausreichend
  - unterschiedliche Verfahren: Heap, indexsequentiell, indiziert-nichtsequentiell
  - oft grundlegende Speichertechnik in DBMS für direkte Organisation
    - Vorteil: keine Hilfsstruktur, keine Adressberechnung
- Dynamische Zugriffstruktur
  - unabhängig von der Anzahl der Datensätze optimal
    - dynamische Adresstransformationsverfahren:
      $\Rightarrow$ dynamische Anpassung des Bildbereichs der Transformation
    - dynamische Indexverfahren: dynamische Anpassung der Anzahl der Indexstufen

### Primär- versus Sekundärindex 1/2

Klassifikation

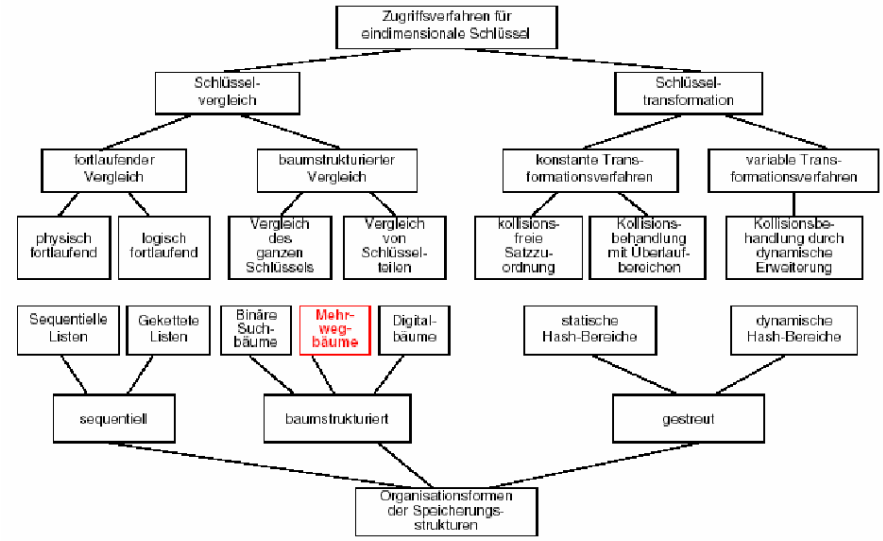
- (Primär-)Index: bestimmt Dateiorganisationsform
  - unsortierte Speicherung von Tupeln: Heap-Organisation
  - sortierte Speicherung von internen Tupeln: sequentielle Organisation
  - gestreute Speicherung von internen Tupeln: Hash-Organisation
  - Speicherung in mehrdimensionalen Räumen: mehrdimensionale Dateiorganisationsformen
  - Normalfall: Primärschlüssel über Primärindex / geclusterter Index
- Sekundärindex
  - redundante Zugriffsmöglichkeit, zusätzlicher Zugriffspfad

### Primär- versus Sekundärindex 2/2



## 7.1 Klassifikation der Verfahren

### Klassifikation der Verfahren

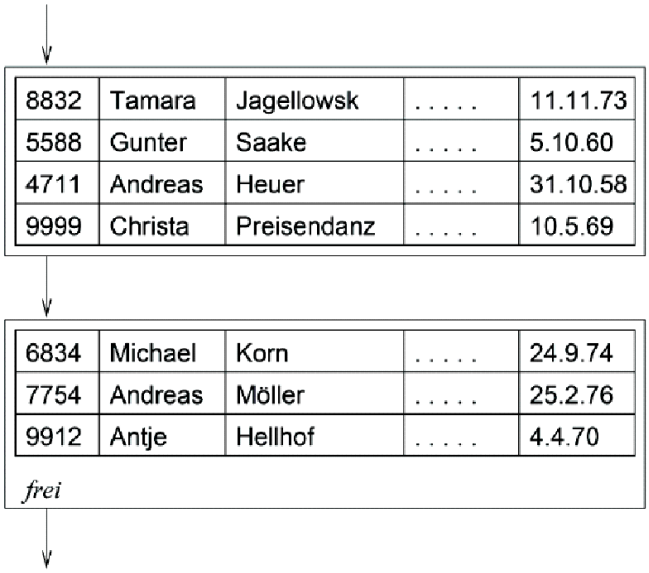



## 7.2 Dateiorganisation

### Physische Dateiorganisation

- Heapfile-Organisation
  - völlig unsortierte Speicherung
  - physische Reihenfolge der Datensätze entspricht der zeitlichen Reihenfolge der Aufnahme von Datensätzen
- Insert-Operation
  - Zugriff auf letzte Seite der Datei
  - Falls genügend freier Platz ⇒ Satz anhängen
  - Ansonsten nächste freie Seite holen
- Delete-Operation
  - lookup, dann Löschbit setzen
- Lookup-Operation
  - sequenzielles Durchsuchen der Gesamtdatei
  - maximaler Aufwand (Heap-Datei meist zusammen mit Sekundärindex eingesetzt)
- Komplexitätsbetrachtung: Neuaufnahme von Daten O(1), Suchen O(n)

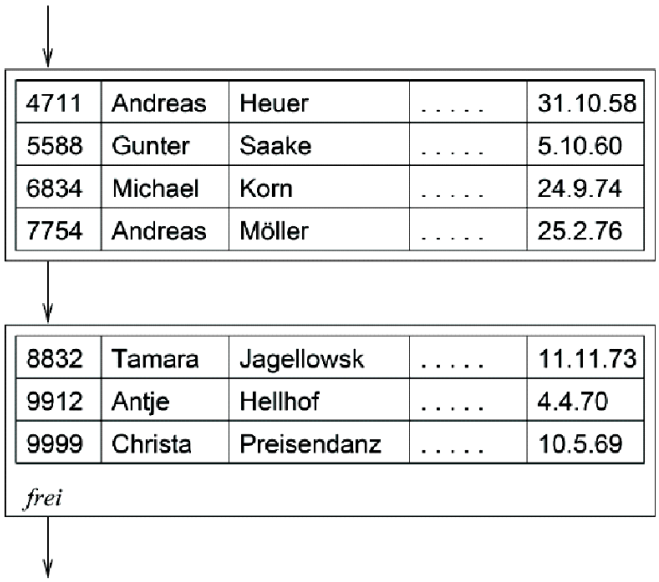
### Heapfile-Organisation




**Sequenzielle Dateiorganisation**

- Prinzip
  - Sortieres Speichern der Datensätze nach einem anwendungsseitig vorgegebenen Schlüsselkriterium
- Insert-Operation
  - Seite suchen und Datensatz einsortieren
  - Füllgrad: beim Anlegen oder sequenziellen Füllen einer Datei jede Seite nur bis zu gewissem Grad (etwa 66%) füllen
- Delete-Operation
  - lookup, dann Löschbit setzen
  - normalerweise in Verbindung mit zusätzlichem Index
    $\Rightarrow$ indexsequenzielle Dateiorganisation

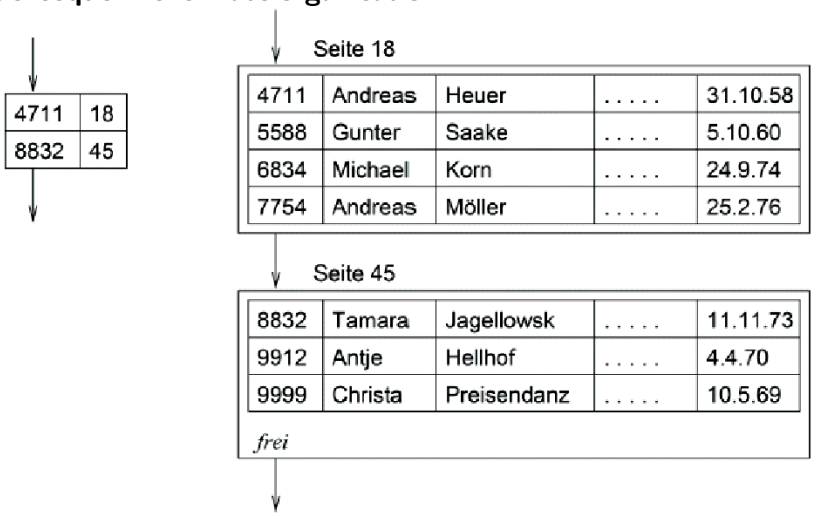
**Sequenzielle Dateiorganisation**



### Indexsequenzielle Dateiorganisation

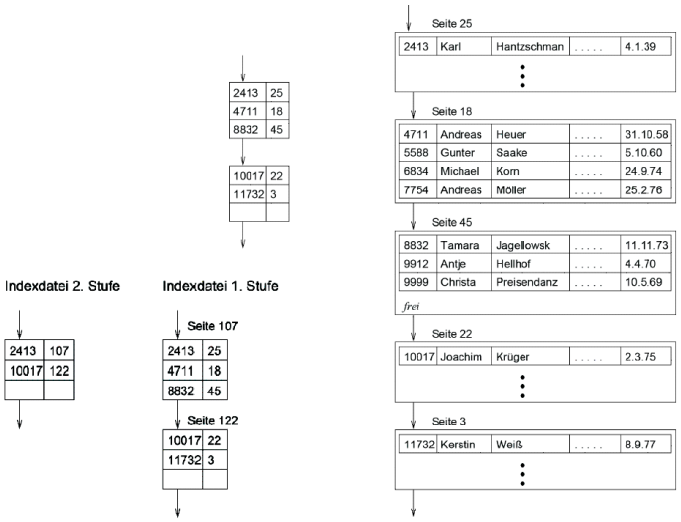
- Prinzip
  - sequenziell organisierte Hauptdatei
  - zusätzliche Indexdatei
    - schnellerer Lookup
    - mehr Platzbedarf (für Index)
    - mehr Zeitbedarf (für Insert und Delete- Operationen)
- Organisation
  - mindestens zweistufiger Baum
    - Blattebene ist Hauptdatei (Datensätze)
    - jede andere Stufe ist Indexdatei mit Einträgen: (Primärschlüsselwert, Seitennummer)
  - zu jeder Seite in der Hauptdatei genau ein Index-Datensatz in der Indexdatei
  - Zwang zu mehrstufigen Indexstrukturen, falls Seitengröße überstiegen wird

### Indexsequenzielle Dateiorganisation




### Aufbau der Indexdatei

- Indexdatei wiederum indexsequentiell verwalten
- Wurzel darf nur aus einer Seite bestehen




## Indexsequenzielle Dateiorganisation

- Lookup-Operation
  - Gesucht wird Datensatz zum Schlüsselwert $w$
  - Sequenzielles Durchlaufen der Indexdatei und Suche von $(v_1, s)$ mit $v_1 \leq w$
    - $(v_1, s)$ ist letzter Satz der Indexdatei $\Rightarrow$ Datensatz zu $w$ kann höchstens auf dieser Seite gespeichert sein (wenn er existiert)
    - nächster Satz $(v_2, s')$ im Index hat $v_2 > w \Rightarrow$ Datensatz zu $w$, wenn vorhanden, ist in Seite $s$ gespeichert
  - $(v_1, s)$ überdeckt Zugriffsattributwert $w$
- Insert-Operation
  - Seite mit Lookup-Operation finden
  - Falls Platz, Satz sortiert in gefundener Seite speichern, Index anpassen, falls neuer Satz der erste Satz in der Seite
  - Falls kein Platz, neue Seite von Freispeicherverwaltung holen, Sätze der zu vollen Seite gleichmäßig auf alte und neue Seite verteilen; für neue Seite Indexeintrag anlegen (ggf. Anlegen einer Überlaufseite)

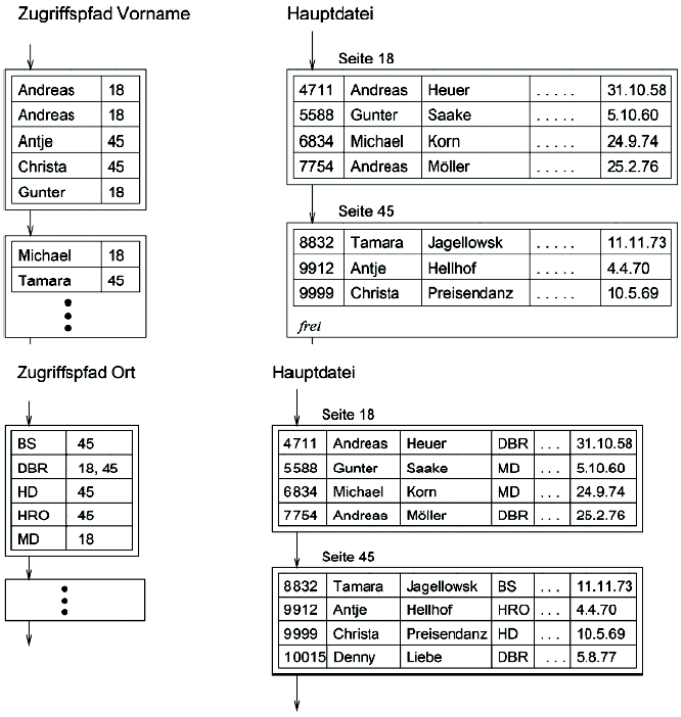
## Indexsequenzielle Dateiorganisation

- Delete-Operation
  - Seite mit Lookup-Operation finden
  - Satz auf Seite löschen (Löschbit setzen)
    - Falls erster Satz auf Seite $\Rightarrow$ Index anpassen
    - Falls Seite nach Löschen leer $\Rightarrow$ Index anpassen und Seite an Freispeicherverwaltung zurückgeben
- Bewertung
  - stark wachsende Dateien: Zahl der linear verketteten Indexseiten wächst
  - automatische Anpassung der Stufenanzahl nicht vorgesehen
  - stark schrumpfende Dateien: nur zögernde Verringerung der Index- und Hauptdatei-Seiten
  - unausgeglichene Seiten in der Hauptdatei (unnötig hoher Speicherplatzbedarf, zu lange Zugriffszeit)

### Indiziert-Nichtsequenzielle Zugriffspfade

- Idee für Organisation für einen Index
  - analog zu einem Stichwortverzeichnis in einen Buch: für jeden Schlüsselwert, die Stellen, an denen der Wert auftritt
  - Unterstützung von Sekundärschlüsseln ⇒ mehrere Zugriffspfade (Sekundärindexe) pro Relation möglich
    - zu jedem Satz der Relation existiert ein Satz (Sekundärschlüsselwert, Seite/TID) im Index
    - Nicht-Eindeutigkeit: mehrere Einträge oder (Seite/TID)
- Mehrstufige Organisation, wobei höhere Indexstufen wieder indexsequentiell organisiert sind → Baumverfahren mit dynamischer Stufenzahl
- Lookup-Operation
  - Schlüsselwert kann mehrfach auftreten
- Insert-Operation
  - Anpassung des Index-Eintrags erforderlich
- Delete-Operation
  - Eintrag aus dem Index entfernen (ggf. auch die Einträge auf höherer Ebene)

### Beispiel zu Sekundärindex

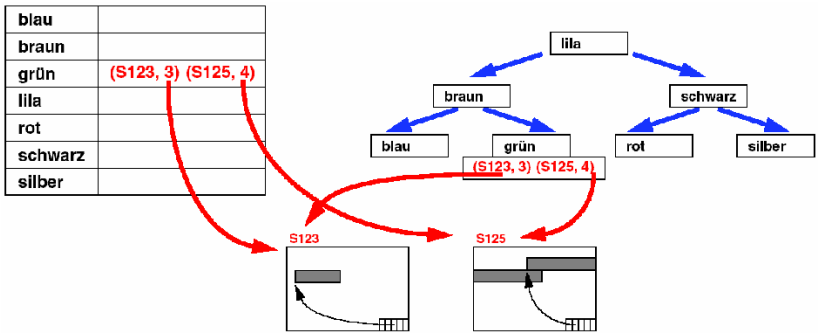



## 7.3 B-Baum

### B-Baum

- Verwaltung der Indexeinträge
  - Variante 1: Liste von Einträge
  - Variante 2: Organisation als Binärbaum / Binärer Suchbaum

* Baumstruktur mit einem linken und rechten Kind
* ausgeglichener balancierter Suchbaum



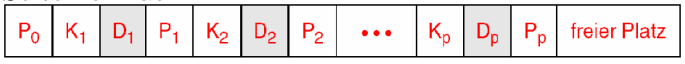

### Erweiterung des binären Suchbaumes

- Mehrwegbaum
  - Baumstruktur mit mehreren Kindern
  - Idee: Die maximale Größe eines Knotens entspricht exakt der Speicherkapazität einer Seite
- B-Baum
  - Variante eines Mehrwegbaumes zur Abbildung von Schlüsselwerten auf interne Satzadressen
  - entworfen für den Einsatz in Datenbanksystemen (Bayer, McCreight, 1972)
- Funktion
  - dynamische Reorganisation durch Splitten und Mischen von Seiten
  - direkter Schlüsselzugriff
  - sortierter sequentieller Zugriff (insbes. B*-Baum) Erweiterung des binären Suchbaumes

## 7.4 B-Baum Eigenschaften und Parameter

### B-Baum Eigenschaften

- Definition: Ein B-Baum vom Typ $(k, h)$ ist ein Baum mit folgenden drei Eigenschaften
  - Jeder Pfad von der Wurzel zum Blatt hat die gleiche Länge $h$
  - Jeder Knoten (außer Wurzel und Blätter) hat mindestens $k+1$ Nachfolger. Die Wurzel ist ein Blatt oder hat mindestens 2 Nachfolger
  - Jeder Knoten hat höchstens $2k+1$ Nachfolger
- Seitenformat




  - $(K_i, D_i, P_i)$ = Eintrag, $K_i$ = Schlüssel
  - $D_i$ = Daten des Satzes oder Verweis auf den Satz (materialisiert oder referenziert)
  - $P_i$ = Zeiger zu einer Nachfolgerseite

### B-Baum Parameter

- Bedeutung der Zeiger $K_i (i = 0, 1, \ldots, p)$
  - $P_0$ weist auf einen Teilbaum mit Schlüsseln kleiner als $K_1$
  - $P_i$ $(i = 1, 2, \ldots, l-1)$ weist auf einen Teilbaum, dessen Schlüssel zwischen $K_i$ und $K_{i+1}$ liegen
  - $P_p$ weist auf einen Teilbaum mit Schlüsseln größer als $K_p$
  - In den Blattknoten sind die Zeiger nicht definiert
- Parameter k (Ordnung des Baumes)
  - errechnet sich aus der Seitengröße
  - $k = \lceil \frac{n}{2} \rceil$, d.h. $2 \cdot k$ ist die maximale Anzahl von Einträgen pro Seite
- Parameter h (Höhe des Baumes)
  - ergibt sich aus der Anzahl der gespeicherten Datenelemente und der Einfügereihenfolge

### Berechnung der maximalen Höhe

- Maximale Höhe $h_{max}$
  - B-Baum der Ordnung $k$ mit $n$ Schlüsseln

    **Level 2** hat $\geq 2$ Knoten

    **Level 3** hat $\geq 2(k+1)$ Knoten

    **Level 4** hat $\geq 2(k+1)^2$ Knoten

    . . . .

    **Level** $h+1$ hat $n+1 \geq 2(k+1)^{h-1}$ (äußere) Knoten

    $\Rightarrow h \leq 1 + \log_{k+1} \frac{n+1}{2}$ und somit:

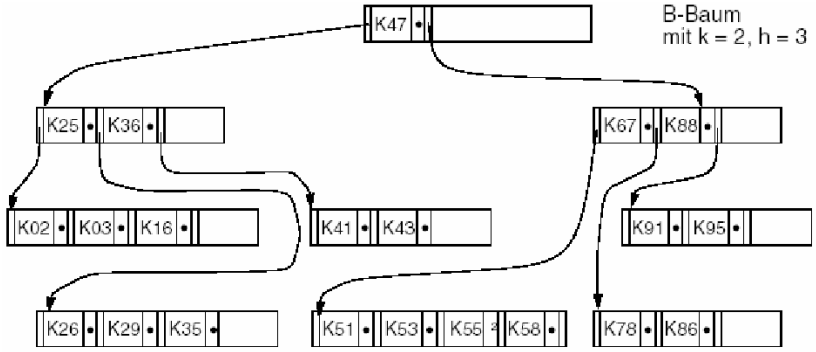
$$\lceil \log_{2k+1} n + 1 \rceil \leq h \leq \lceil \log_{k+1} \frac{n+1}{2} \rceil + 1$$

- Beobachtung
  - Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt $\Rightarrow$ Speicherplatzausnutzung $\geq 50\%$

## 7.5 Operationen auf B-Bäumen

### Beispiel eines B-Baumes

- B-Baumstruktur als Zugriffspfad für den Primärschlüssel ANR

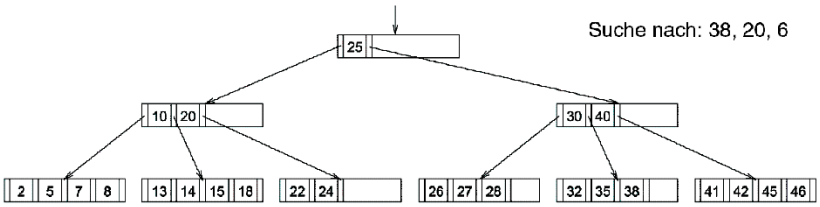



- Operationen
  - Suchen eines Datensatzes mit vorgegebenem Schlüsselwert
  - Einfügen und Löschen eines Datensatzes

### Suche im B-Baum

- Beginnend mit dem Wurzelknoten, wird ein Knoten jeweils von links nach rechts durchsucht
  1. Stimmt $K_i$ mit dem gesuchten Schlüsselwert überein, ist der Satz gefunden. (Weitere Sätze mit gleichem Schlüsselwert befinden sich ggf. in dem Teilbaum, auf den $P_{i-1}$ zeigt.
  2. Ist $K_i$ größer als der gesuchte Wert, wird die Suche in der Wurzel des von $P_{i-1}$ identifizierten Teilbaumes fortgesetzt.
  3. Ist $K_i$ kleiner als der gesuchte Wert, wird der Vergleich mit $K_{i+1}$ wiederholt.
  4. Ist auch $K_{2k}$ noch kleiner als der gesuchte Wert, wird die Suche im Teilbaum von $P_{2k}$ fortgesetzt
- Ist der weitere Abstieg in einen Teilbaum (2. oder 4.) nicht möglich (Blattknoten): Suche abbrechen, kein Satz mit gewünschtem Schlüsselwert vorhanden.

### Suche im B-Baum, Beispiel

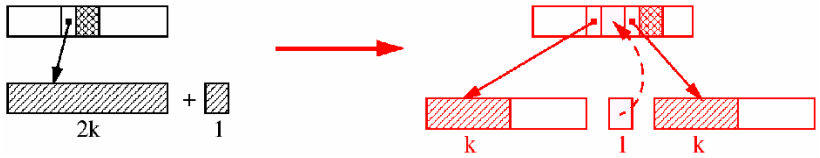



### Einfügen im B-Baum

- Regel: Eingefügt wird nur in Blattknoten!
- Vorgehen: zunächst Abstieg durch den Baum wie bei Suche
  - $S \leq K_i$: folge $P_{i-1}$
  - $S > K_i$ : prüfe $K_{i+1}$
  - $S > K_{2k}$: folge $P_{2k}$
- im so gefundenen Blattknoten
  - Satz entsprechend der Sortierreihenfolge einfügen
  - Sonderfall: Blattknoten ist schon voll (enthält $2k$ Sätze) $\Rightarrow$ Split des Blattknotens

### Split beim Einfügen eines Satzes im B-Baum 1/2

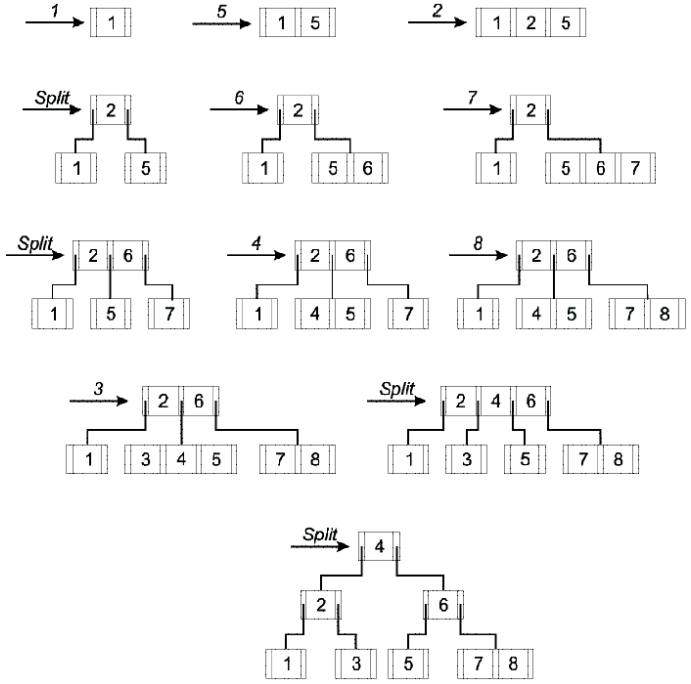
- Vorgehen beim Split
  - einen neuen Blattknoten erzeugen
  - die 2k+1 Sätze (in Sortierordnung!) halbe-halbe zwischen altem und neuem Blattknoten aufteilen
  - die ersten k Sätze in die erste (die linke) Seite
  - die letzten k Sätze in die zweite (die rechte) Seite
  - den mittleren (k+1-ten) Satz als neuen Diskriminator (als Verzweigungsinformation bei der Suche) in den eine Stufe höheren Knoten einfügen, der auf den Blattknoten verweist



### Split beim Einfügen eines Satzes im B-Baum 2/2

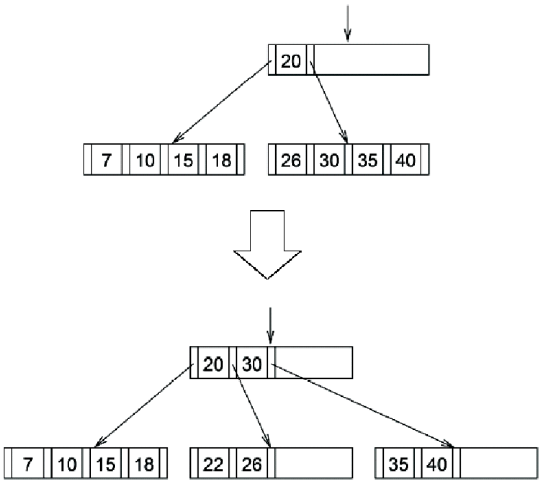
- zwei mögliche Situationen nach einem Split
  - der übergeordnete Knoten ist voll ⇒ Split auf dieser Ebene wiederholen
  - ausreichend Platz ⇒ Fertig
- Weiterer Sonderfall
  - Split des Wurzelknotens ⇒ Erzeugung
  - von zwei neuen Knoten ⇒ Neue Wurzel mit zwei Nachfolgeknoten
  - Höhe des Baums wächst um 1 (Man sagt bildlich: Der Baum reißt von unten nach oben auf.)
- Dynamische Reorganisation
  - kein Entladen und Laden erforderlich
  - Baum immer balanciert

### Einfügen im B-Baum: Beispiel

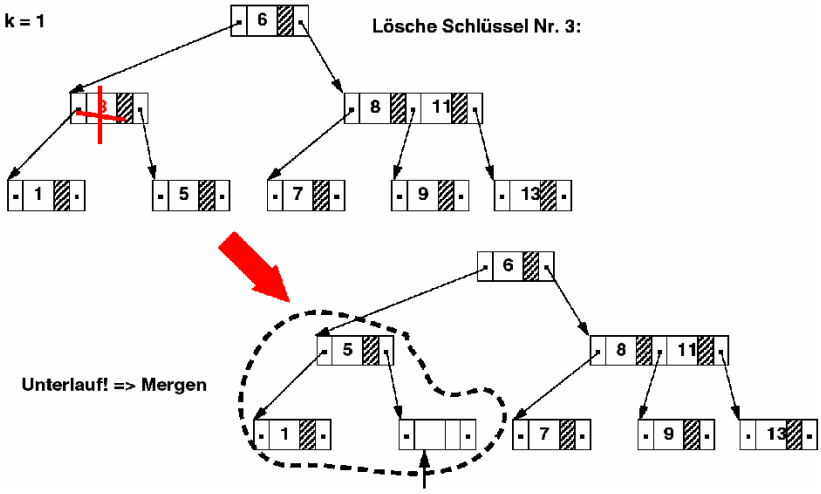



### Einfügen und Löschen im B-Baum

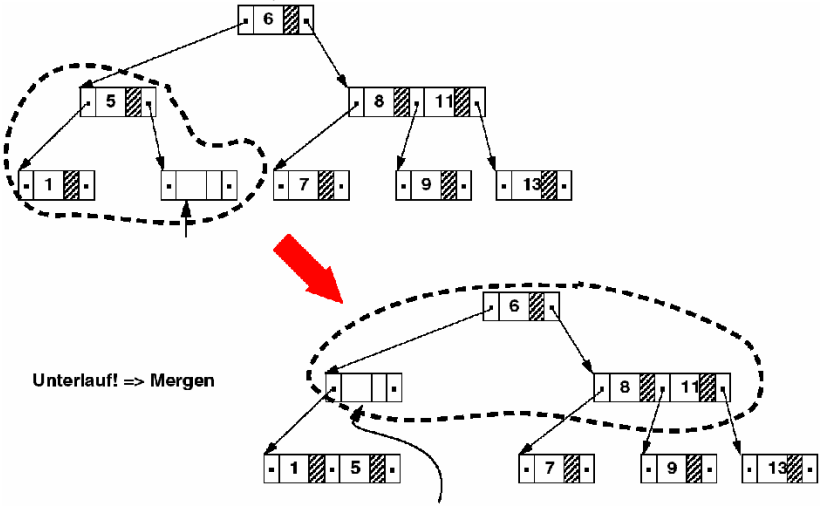
- Problem: Einfügen kann Überlauf erzeugen; Löschen kann Unterlauf und Überlauf erzeugen;
- Beispiel: Einfügen und Löschen von Schlüssel Nr. 22



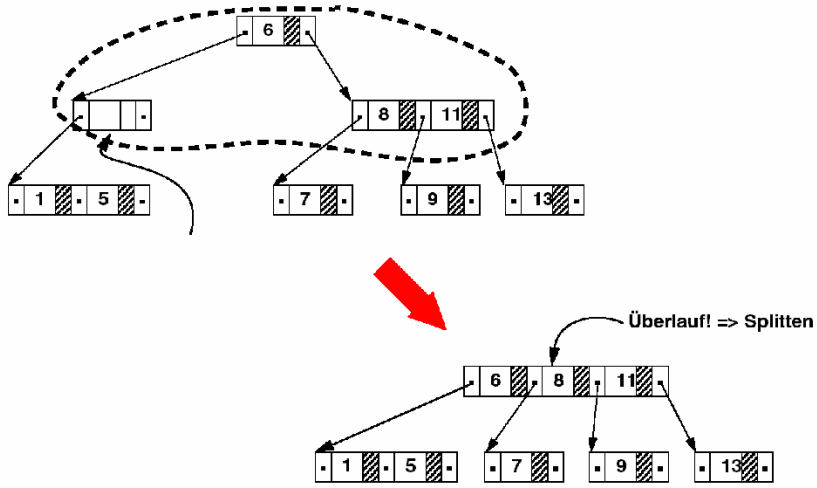

## Löschen im B-Baum 1/4



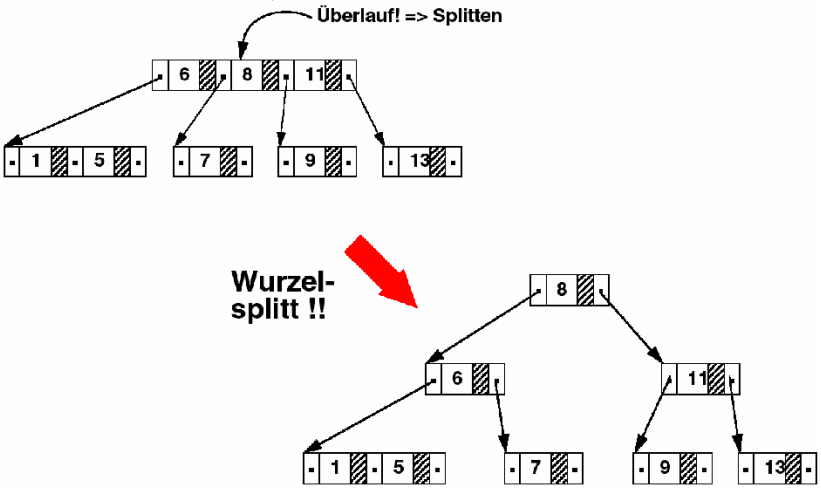

## Löschen im B-Baum 2/4




## Löschen im B-Baum 3/4




## Löschen im B-Baum 4/4




## Löschvorgang - algorithmisch

- Beispiel - es gibt verschiedene Algorithmen
  - Suche den Knoten, in dem der zu löschende Schlüssel $S$ liegt
  - Falls Schlüssel $S$ in Blattknoten, dann lösche Schlüssel in Blattknoten und behandle evtl. entstehenden Unterlauf
- Falls Schlüssel $S$ in einem inneren Knoten, dann untersuche linken und rechten Nachfolgerknoten zu dem zu löschenden Schlüssel $S$:
  - untersuche, welcher Nachfolgerknoten von $S$ mehr Elemente hat, der linke oder der rechte. Falls beide gleich viele Elemente haben, dann entscheide für einen.
  - Ersetze zu löschenden Schlüssel $S$ durch direkten Vorgänger $S'$ aus linken Nachfolgeknoten bzw. durch direkten Nachfolger $S''$ aus rechten Nachfolgeknoten.
  - Lösche $S'$ bzw. $S''$ aus dem entsprechenden Nachfolgeknoten (rekursiv)

## Anmerkungen zum Unterlauf

- ein entgültiger Unterlauf entsteht bei obigen Algorithmus erst auf Blattebene!
- Unterlaufbehandlung wird durch einen Merge des Unterlaufknotens mit seinem Nachbarknoten und dem darüberliegenden Diskriminator durchgeführt

- Wurde einmal mit dem Mergen auf Blattebene begonnen, so setzt sich dieses Mergen nach oben hin fort
- Das Mergen auf Blattebene wird solange weitergeführt, bis kein Unterlauf mehr existiert, oder die Wurzel erreicht ist
- Wird die Wurzel erreicht, kann der Baum in der Höhe um eins schrumpfen. Beim Mergen kann es auch wieder zu einem Überlauf kommen. In diesem Fall muss wieder gesplittet werden.

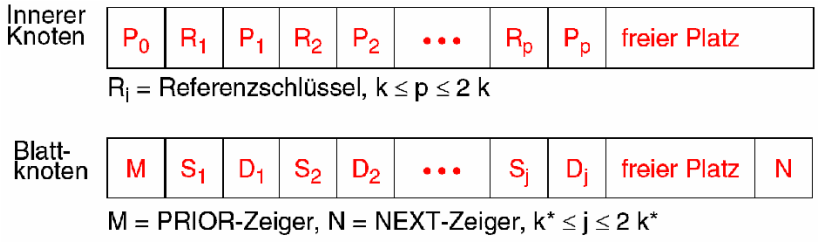
#### Komplexität der Operationen

- Aufwandsabschätzung
  - Einfügen, Suchen und Löschen: $O(\log_k n)$ Operationen
  - entspricht Höhe eines Baumes
  - Ziel: geringere Höhe → größere Breite
- Konkretes Beispiel
  - Seiten der Größe 4 KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger ⇒ zwischen 50 und 100 Indexeinträge pro Seite; Ordnung dieses B-Baumes 50
  - 1.000.000 Datensätze: $\log_{50} 1.000.000 = 4$ Seitenzugriffe im schlechtesten Fall
  - Wurzelseite jedes B-Baumes normalerweise im Puffer: drei Seitenzugriffe

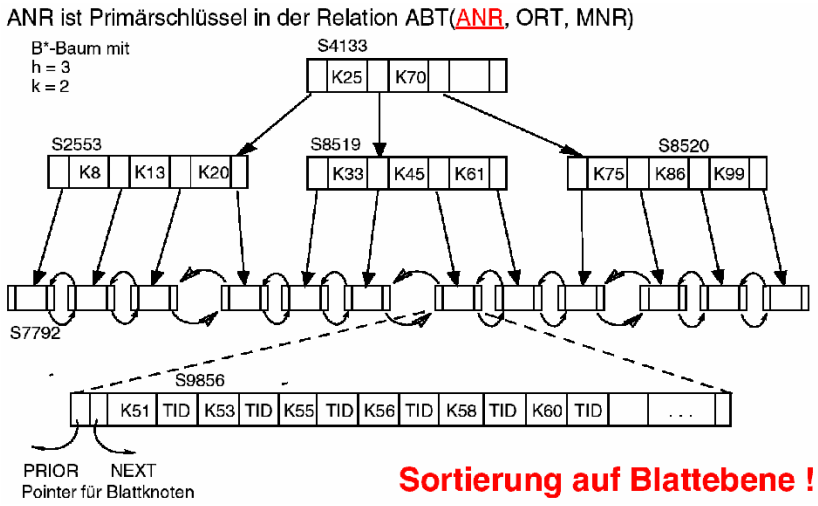
### 7.6 B*-Baum

#### B*-Baum

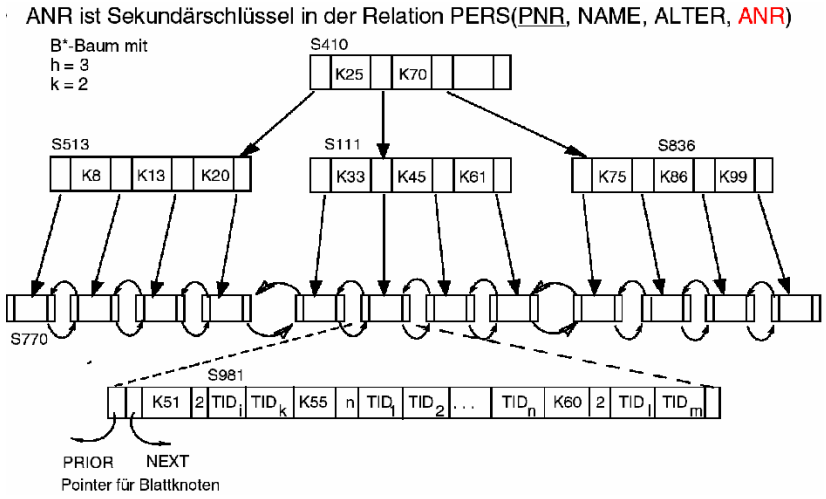
- Eigenschaften und Unterschiede zum B-Baum
  - Alle Sätze (bzw. Schlüsselwerte mit TID) werden in den Blattknoten abgelegt.
  - Innere Knoten enthalten nur Verzweigungsinformation (also u.U. auch Schlüsselwerte, die in keinem Satz vorkommen), aber keine Daten.
  - Aufbau von B*-Baum-Knoten




#### Beispiel eines B*-Baums für Primärschlüssel

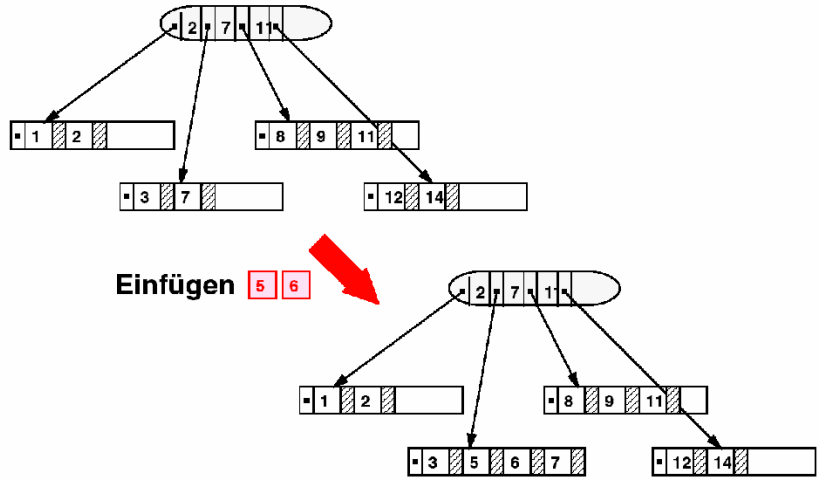



### Beispiel eines B*-Baums für Sekundärschlüssel

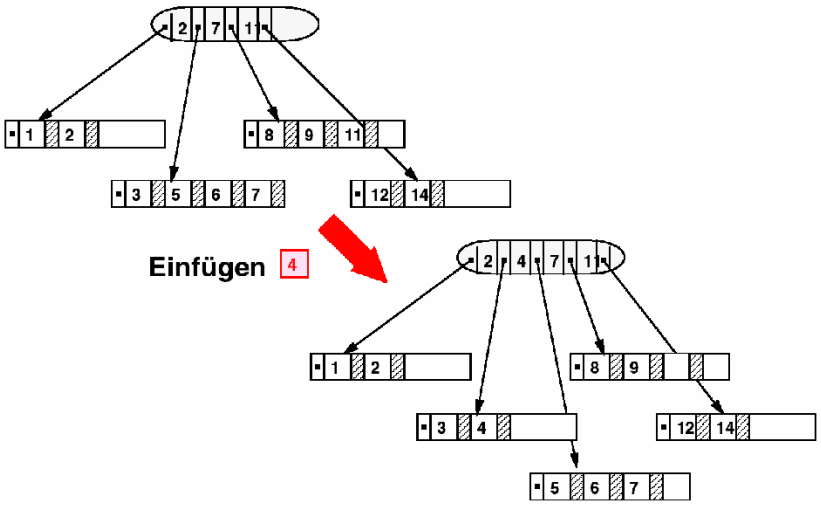



## 7.7 Operationen auf B*-Bäumen

### Einfügen im B*-Baum

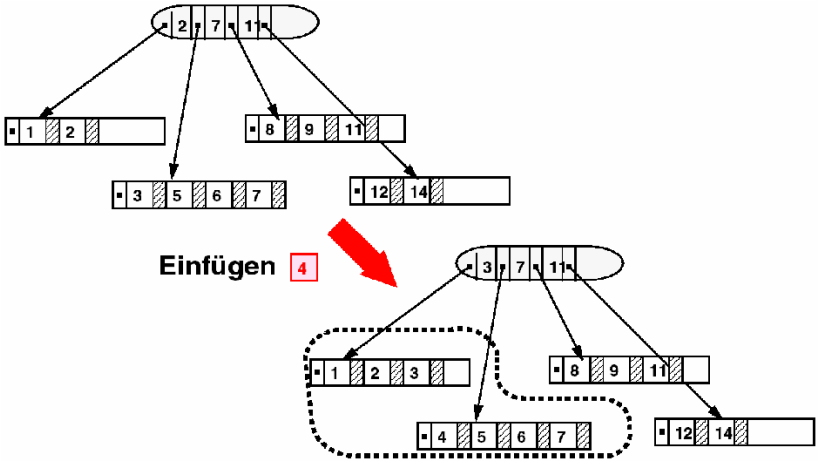



### Split im B*-Baum

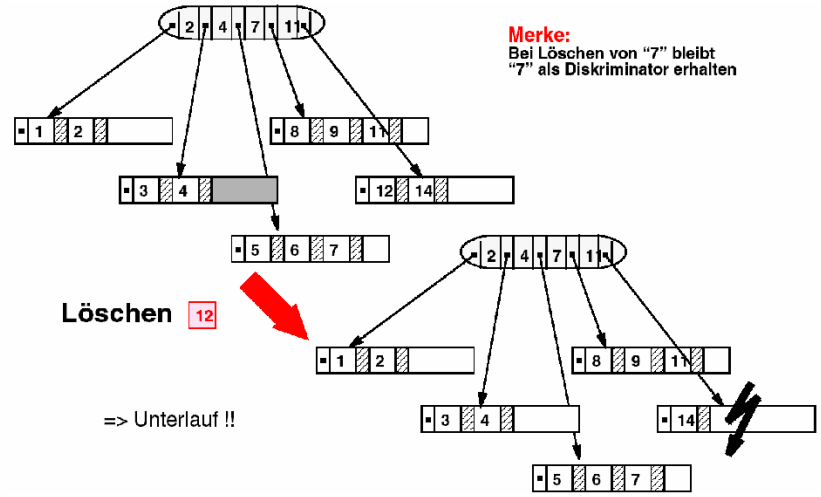



## Balancierung im B*-Baum

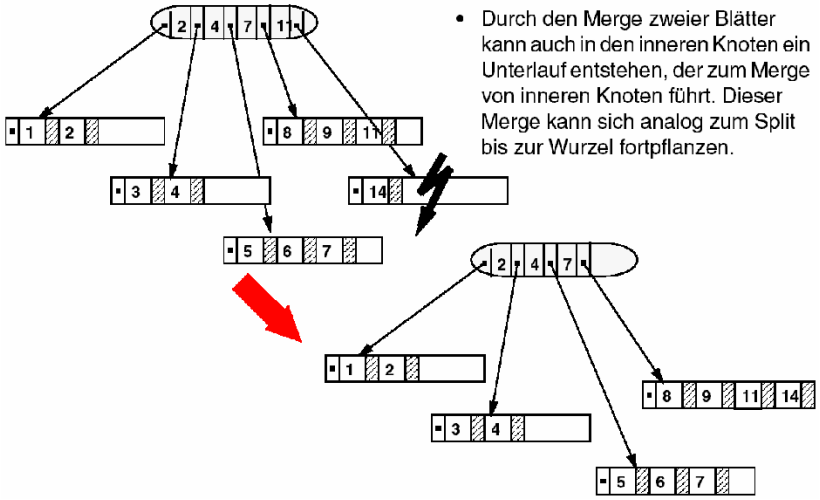
- Statt Splitt bei Überlauf, Neuverteilung der Einträge unter Berücksichtigung eines oder mehrer benachbarter Knoten




## Löschen im B*-Baum




## Löschen im B*-Baum




### Löschen von Sätzen aus B*-Baum

- Algorithmus
  1. Suche den zu löschenden Eintrag im Baum
  2. Entsteht durch das Löschen ein Unterlauf? (#Einträge $< k$?)
  - **NEIN:** Entferne den Satz aus dem Blatt (Eine Aktualisierung des Diskriminators im Vaterknoten ist nicht erforderlich!)
  - **JA:** Mische Blatt mit einem Nachbarknoten
    - Ist die Summe der Einträge in beiden Knoten größer als 2k?
    - **NEIN:** Fasse beide Blätter zu einem Blatt zusammen; falls dabei ein Unterlauf im Vaterknoten entsteht: mische die inneren Knoten analog
    - **JA:** Teile die Sätze neu auf beide Knoten auf, so daß ein Knoten jeweils die Hälfte der Sätze aufnimmt; der Diskriminator im Vaterknoten ist entsprechend zu aktualisieren
- Anmerkung
  - Vielzahl von Varianten bzgl. Aufteilung / Neuverteilung nach UDI-Operationen

## 7.8 Vergleich B- und B*-Baum

### Vergleich B- und B*-Baum

| B-Baum | B*-Baum |
|---|---|
| • keine Redundanz | • Schlüsselwerte teilweise redundant gespeichert |
| • Lesen aller Sätze sortiert nach Schlüsselwert nur mit Verwaltung eines Stacks der max. Tiefe = Baumhöhe h | • Kette der Blattknoten liefert alle Sätze nach Schlüsselwert sortiert |
| • bei Einbettung der Datensätze geringe Verzweigungszahl ("Grad" oder "fan-out"), daher größere Höhe | • hohe Verzweigung in der inneren Knoten, daher geringere Höhe |
| • einige wenige Sätze (die in der Wurzel) werden mit *einem* Seitenzugriff gefunden | • für alle Sätze müssen h Seiten gelesen werden |
| | • Schlüsselwerte in den inneren Knoten müssen nicht in den Datensätzen vorkommen (Optimierung beim Löschen von Sätzen) |

## 7.9 Erweiterungen für B- und B*-Bäumen

### Duplikatbehandlung in B- und B*-Bäumen 1/2

- Problem
  - Die einfachen Algorithmen zum Suchen, Einfügen, Splitten und Zusammenfassen nehmen an, dass alle Sätze mit dem gleichen Schlüssel in einen Knoten hineinpassen
  - Was passiert, wenn diese Annahme verletzt wird?
- Lösung 1
  - Sätze oder TIDs von Duplikaten werden in Überlaufseiten verwaltet, die den Index nicht beeinflussen.
- Lösung 2
  - Im B*-Baum wird der am weitesten links stehende Eintrag gesucht, der dem Suchschlüssel entspricht. Duplikate werden durch Verfolgen der verketteten Blätter gefunden.
  - Im B-Baum werden zusätzlich rekursiv alle Zeiger $P_{i-1}$ verfolgt, für die der Index $i$ der Bedingung $K_{i-1} = K = K_i$ gilt.
  - Problem: Beim Löschen eines Satzes müssen beim B- und B*-Baum alle Duplikate nach dem gesuchten Tupel-Identifier (RowID) durchsucht werden.

### Duplikatbehandlung in B- und B*-Bäumen 2/2

- Lösung 3
  - Schlüsselwert und Tupel-Identifier (RowID) werden zu einem Schlüssel aneinander gehängt.
  - Dadurch wird der Schlüssel eindeutig, d.h. es treten keine Duplikate mehr auf.
  - Verwendet in IBM DB2, Oracle 8, Microsoft SQL Server

### Clustered und Non-Clustered B*-Bäume

- Clustered Indizes
  - Die Blätter eines clustered B*-Baumes sind die Datenseiten der gespeicherten Tabelle
  - Die Tupel sind in den Datenseiten in sortierter Reihenfolge bezüglich des Indizierten Attributes oder der Attributkombination gespeichert.
  - Schnellerer Zugriff auch bei Bereichsanfragen
- Non-Clustered Indizes
  - Die Blätter eines non-clustered B*-Baumes enthalten nur Tupel-Identifier (RowIDs) auf die Tupel in den Datenseiten der gespeicherten Tabelle.
  - Die Tupel sind in den Datenseiten in zufälliger Reihenfolge als Heap-File gespeichert.
  - Die Anzahl der gelesenen Blöcke erhöht sich dadurch um mindesten eins (Höhe des B*-Baumes plus einen Zugriff für den TID-Verweis).

### Bulk-Loading

- Problem
  - B*-Baum soll nachträglich für eine große Menge an Daten erstellt werden
    - z.B. create index auf einer gefüllten Tabelle
  - Dynamischer Algorithmus für einzelne Inserts ist langsam
- Bulk-Loading
  - Sortiere erst die Daten nach den Index-Attribut oder der Attributkombination
  - Baum wird in einem Durchlauf über die sortierten Daten aufgebaut
  - Nur die Knoten auf der rechten Seite des Baumes ändern sich (Anzahl ist in $O(\log n)$) und werden während des Aufbau im Hauptspeicher gehalten.
- Beispiel

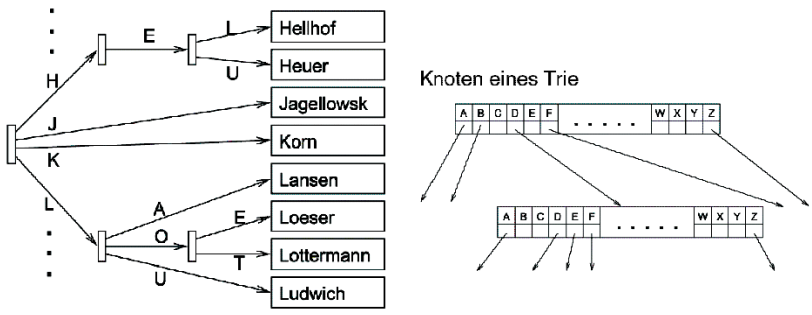
## 7.10 Weitere Baum-Indexstrukturen

### Digital- und Präfixbäume

- Indexierung von Zeichenketten
  - B-Bäume: Betrachtung als atomare Werte
  - Lösungsansatz: Digital- oder Präfixbäume aus dem Umfeld des Information Retrieval
- Digitalbäume
  - (feste) Indexierung der Buchstaben des zugrundeliegenden Alphabets
  - keine Garantie der Balancierung
  - Beispiele: Tries, Patricia-Bäume
- Präfix-Bäume
  - Indexierung über Präfixe der Menge von Zeichenketten

### Tries

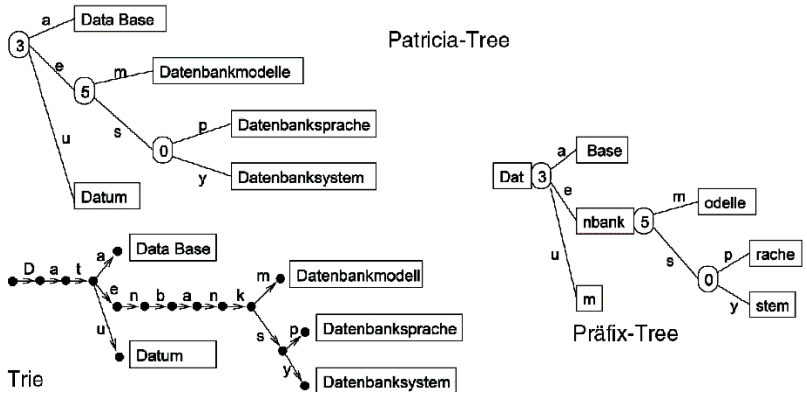
- Beispiel




- Probleme verursachen (fast) leere Knoten / sehr unausgeglichene Bäume
  - lange gemeinsame Teilworte
  - nicht vorhandene Buchstaben und Buchstabenkombinationen

### Patricia-Bäume

- Practical Algorithm To Retrieve Information Coded In Alphanumeric
  - Überspringen von Teilworten (zusätzliche Speicherung in Präfix-Bäumen)



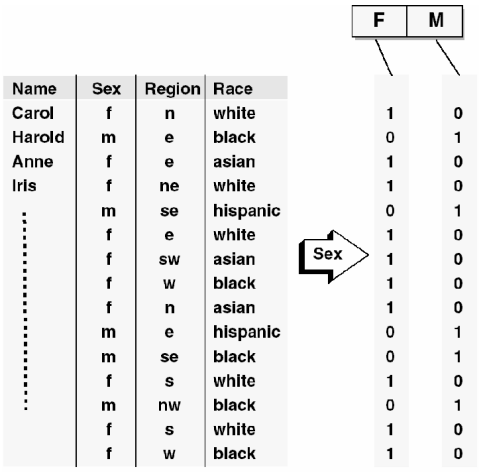

## 7.11 Bitmap-Index

### Bitmap-Index

- Problem-Beispiel: B-Baum auf Geschlecht bei Kundentabelle mit 1000.000 Tupeln resultiert in zwei Listen mit jeweils ca. 500.000 Tupeln
  - Anfrage nach allen weiblichen Kunden erfordert 500.000 einzelne Zugriffe ⇒ Table-Scan ist um Längen schneller
- Folgerung
  - B-Bäume (und auch Hashing) sind sinnvoll für Prädikate mit hoher Selektivität (geringer Anteil von zu erwartenden Tupeln gegenüber allen Tupeln einer Relation)
- Daumenregel
  - Grenztrefferrate liegt bei ca. 3%.
  - Höhere Trefferraten lohnen bereits den Aufwand für einen Indexzugriff nicht mehr

### Grundlegende Idee der Bitmap-Indexstruktur

- Lege für jede Attributausprägung eine Bitmap/Bitliste an
- Jedem Tupel der Tabelle ist ein Bit in der Bitmap zugeordnet
- Bitwert 1 heißt der Attributwert wird angenommen, 0 heißt Attributwert wird nicht angenommen
- Notwendig: Fortlaufende Nummerierung der Tupel (TIDs)




### Größe von Bitmap-Indexstrukturen

- Indexgröße: (Anzahl der Tupel) * (Anzahl der Ausprägungen) Bits
  - Beispiel: Geschlecht mit 2 Ausprägungen in Relation mit 10k Tupeln, 4 Byte TID; Bitmap: 2 * 10k Bits = 20k Bits = 2.5k Bytes
- Eigenschaften von Bitmap-Indexstrukturen
  - wachsen mit der Anzahl der Ausprägungen
  - sind besonders interessant bei Wertigkeiten bis ca. 500
  - sind bei kleinen Wertigkeiten (z.B. Geschlecht) nur sinnvoll, wenn entsprechendes Attribut oft in Konjunktionen mit anderen indizierten Attributen auftritt (z.B. Geschlecht und Wohnort)
  - Indexgröße nicht so problematisch, da gerade bei höherwertigen Attributen die Bitmaps sehr dünn besetzt sind und Kompressionsverfahren (z.B. RLE) sehr gut einsetzbar sind

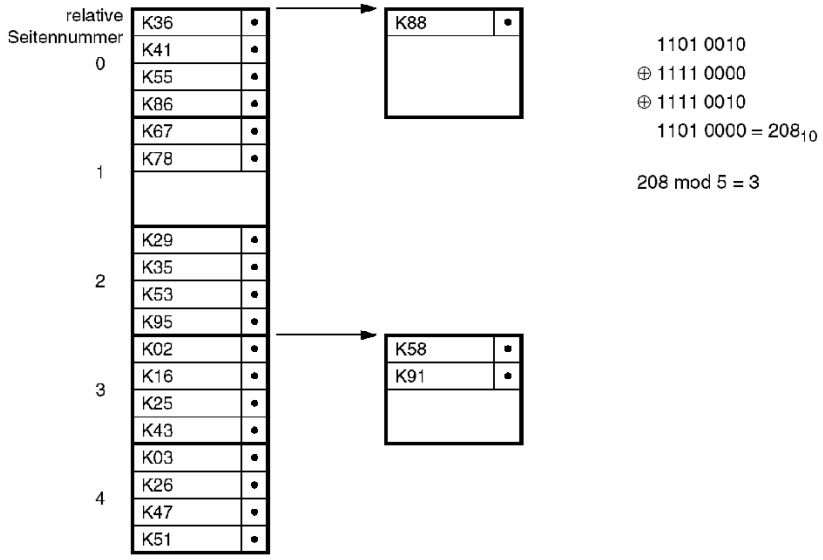
## 7.12 Hash-Verfahren

### Gestreute Speicherungsstrukturen (Hash-Verfahren)

- Idee: direkte Berechnung der Satzadresse über Schlüssel (Schlüsseltransformation)
- Hash-Funktion $h : S \rightarrow \{1, 2, \ldots, n\}$, S = Schlüsselraum, n = Größe des statischen Hash-Bereiches in Seiten (Buckets)
- Idealfall: h ist injektiv (keine Kollisionen)
  - nur in Ausnahmefällen möglich (dichte Schlüsselmenge)
  - jeder Satz kann mit einem einzigen Seitenzugriff referenziert werden
- Statische Hash-Bereiche mit Kollisionsbehandlung
  - vorhandene Schlüsselmenge $K$ ($K \subseteq S$) soll möglichst gleichmäßig auf die n Buckets verteilt werden
  - Behandlung von Synonymen: Aufnahme im selben Bucket, wenn möglich; ggf. Anlegen und Verketten von Überlaufseiten
  - typischer Zugriffsfaktor: 1.1 bis 1.4
  - Vielzahl von Hash-Funktionen anwendbar z. B. Divisionsrestverfahren (Primzahl bestimmt Modul), Faltung, Codierungsmethode, ...

### Statisches Hash-Verfahren mit Überlaufbereichen: Beispiel

- Schlüsselberechnung für K02



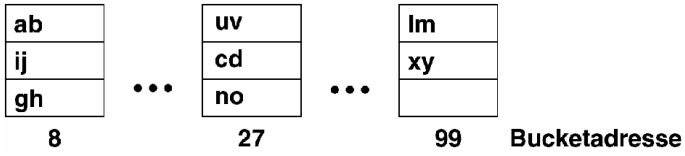

## 7.13 Externes Hashing

### Externes Hashing ohne Überlaufbereiche 1/2

- Ziel: Jeder Satz kann mit genau einem E/A-Zugriff gefunden werden ⇒ gekettete Überlaufbereiche können nicht benutzt werden
- Statisches Hashing
  - $N$ Sätze, $n$ Buckets mit Kapazität $b$
  - Belegungsfaktor $\beta = \frac{N}{n \cdot b}$
- Überlaufbehandlung
  - Open Adressing (ohne Kette oder Zeiger)
  - Bekannteste Schemata: Lineares Sondieren und Double Hashing
  - Sondierungsfolge für einen Satz mit Schlüssel $k$: $H(k) = (h_1(k), h_2(k), ..., h_n(k))$
  - bestimmt Überprüfungsreihenfolge der Buckets (Seiten) beim Einfügen und Suchen
  - wird durch $k$ festgelegt und ist eine Permutation der Menge der Bucketadressen $\{0, 1, \ldots, n-1\}$

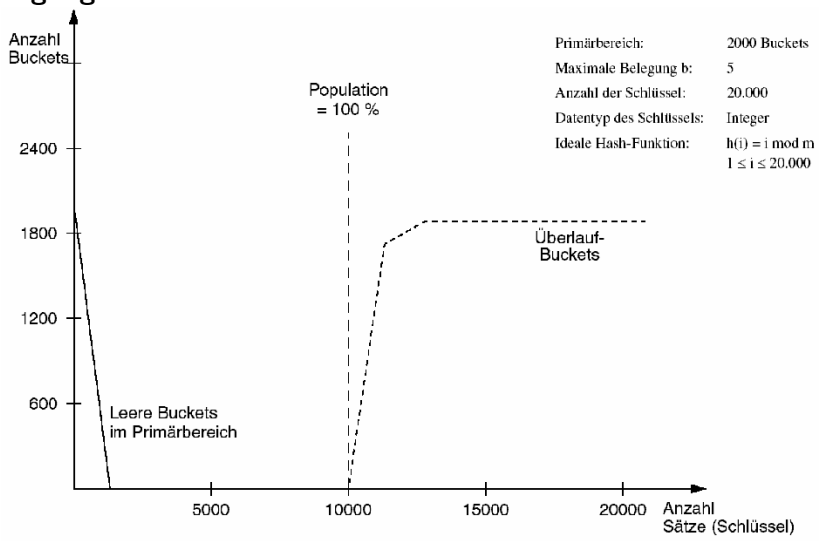
### Externes Hashing ohne Überlaufbereiche 2/2

- Erster Versuch

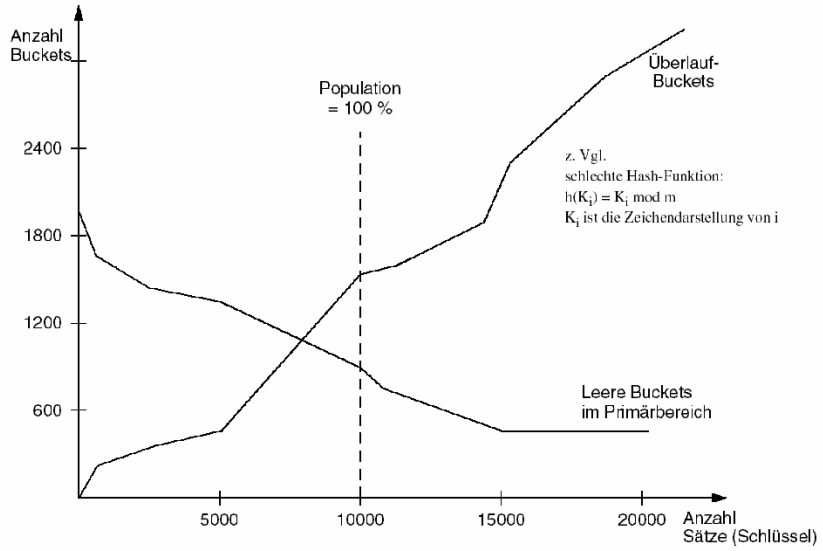



  - Aufsuchen oder Einfügen von k = xy
  - Sondierungsfolge sei H(xy) = (8, 27, 99, ...)
  - Viele E/A-Zugriffe
  - Wie funktioniert das Suchen und Löschen?

## Belegung von Hash-Bereichen Wunschszenario




## Belegung von Hash-Bereichen Messung




## Externes Hashing mit Separatoren 1/4

- Einsatz von Signaturen
  - Jede Signatur $s_i(k)$ ist ein t-Bit Integer
  - Für jeden Satz mit Schlüssel k wird eine Signaturfolge benötigt: $S(k) = (s_1(k), s_2(k), \ldots, s_n(k))$
  - Die Signaturfolge wird eindeutig durch den Schlüsselwert k bestimmt
  - Die Berechnung von $S(k)$ kann durch einen Pseudozufallszahlengenerator mit $k$ als Saat erfolgen (Gleichverteilung der t Bits wichtig)
- Nutzung der Signaturfolge zusammen mit der Sondierungsfolge
  - Bei Sondierung $h_i(k)$ wird $s_i(k)$ benutzt, $i = 1, 2, ..., n$
  - Für jede Sondierung wird eine neue Signatur berechnet!
- Einsatz von Separatoren
  - Ein Separator besteht aus t Bits (entspricht also einer Signatur)

- Separator j, j = 0, 1, 2, ..., n-1, gehört zu Bucket j
- Eine Separatortabelle SEP enthält n Separatoren und wird im Hauptspeicher gehalten.

### Externes Hashing mit Separatoren 2/4

- Nutzung der Separatoren
  - Wenn Bucket $B_i$ r-mal ($r > b$) sondiert wurde, müssen mindestens (r - b) Sätze abgewiesen werden; sie müssen das nächste Bucket in ihrer Sondierungsfolge aufsuchen.
  - Für die Entscheidung, welche Sätze im Bucket gespeichert werden, sind die r Sätze nach ihren momentanen Signaturen zu sortieren.
  - Sätze mit niedrigen Signaturen werden in $B_i$ gespeichert, die mit hohen Signaturen müssen weitersuchen.
  - Eine Signatur, die die Gruppe der niedrigen Signaturen eindeutig von der Gruppe der höheren Signaturen trennt, wird als Separator j für $B_j$ in SEP aufgenommen. Separator j enthält den niedrigsten Signaturwert der Sätze, die weitersuchen müssen.
  - Ein Separator partitioniert also die r Sätze von $B_j$. Wenn die ideale Partitionierung (b, r-b) nicht gewählt werden kann, wird eine der folgenden Partitionierungen versucht:
    - $(b-1, r-b+1), (b-2, r-b+2), \ldots, (o, r)$
    - Ein Bucket mit Überlaufsätzen kann weniger als b Sätze gespeichert haben.

### Externes Hashing mit Separatoren 3/4

- Beispiel: Parameter r = 5, t= 4 Signaturen 0001, 0011, 0100, 0100, 1000 für Bucket $B_i$
  - b = 4: Separator = 1000, Aufteilung (4, 1) ⇒ SEP [j] = 1000
  - b = 3: Separator = 0100, Aufteilung (2, 3) ⇒ SEP [j] = 0100
- Initialisierung der Separatoren mit $2^t - 1$
  - Separator eines Buckets, der noch nicht übergelaufen ist, muss höher als alle tatsächlich auftretenden Signaturen sein → $2^t - 1$
  - Bereich der Signaturen: $0, 1, \ldots, 2^t - 2$
- Aufsuchen
  - In der Sondierungsfolge S(k) werden die $s_i(k)$ mit $SEP[h_i(k)]$, i=1,2,...n, im Hauptspeicher verglichen.
  - Sobald ein $SEP[h_i(k)] > s_i(k)$ gefunden wird, ist die richtige Bucketadresse $h_i(k)$ lokalisiert.
  - Das Bucket wird eingelesen und durchsucht. Wenn der Satz nicht gefunden wird, existiert er nicht. ⇒ genau ein E/A-Zugriff erforderlich

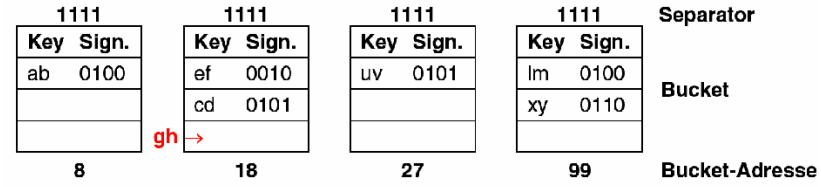
### Externes Hashing mit Separatoren 4/4

- Einfügen
  - Kann Verschieben von Sätzen und Ändern von Separatoren erfordern.
  - Wenn für einen Satz $s_i(k) < SEP[j]$ mit $j = h_i(k)$ gilt, muss er in Bucket $B_j$ eingefugt werden.
  - Falls $B_j$ schon voll ist, müssen ein oder mehrere Sätze verschoben und $SEP[j]$ entsprechend aktualisiert werden.
  - Alle verschobenen Sätze müssen dann in Buckets ihrer Sondierungsfolgen wieder eingefügt werden

- dieser Prozess kann kaskadieren
- $\beta$ nahe bei 1 ist unsinnig, da die Einfügekosten explodieren; Empfehlung: $\beta < 0.8$

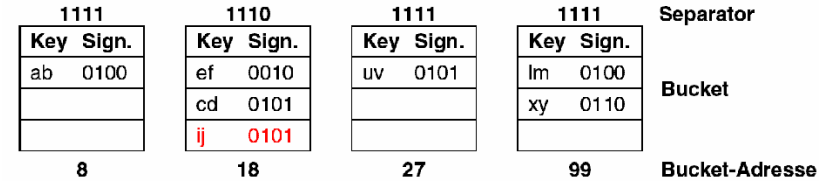
### Externes Hashing mit Separatoren: Beispiel 1

- Beispiel 1: Startsituation



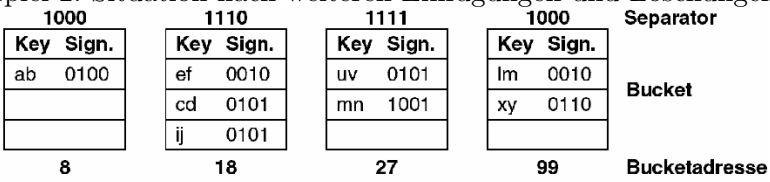

  - Einfügen von k = gh mit $h_1(gh)$=18, $s_1(gh)$ = 1110
  - Einfügen von k = ij mit $h_1(ij)$ = 18, $s_1(ij)$ = 0101

- Erster Bucketüberlauf
  - k = gh muss weiter sondieren: z.B.: $h_2(gh)$ = 99, $s_2(gh)$ = 1010

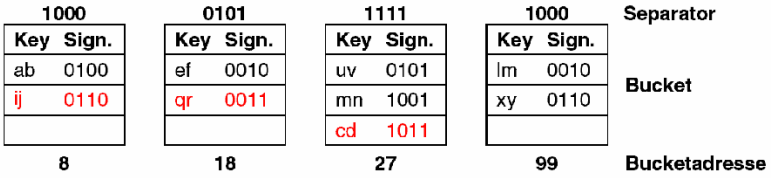



### Externes Hashing mit Separatoren: Beispiel 2

- Beispiel 2: Situation nach weiteren Einfügungen und Löschungen




  - Einfügung von H(qr) = (8, 18, ...) und S(qr) = (1011, 0011, ...)




  - Sondierungs- und Signaturfolgen von cd und ij seien
    - H (cd) = (18, 27, ...) und S (cd) = (0101, 1011, ...)
    - H (ij) = (18, 99, 8, ...) und S (ij) = (0101, 1110, 0110, ...)

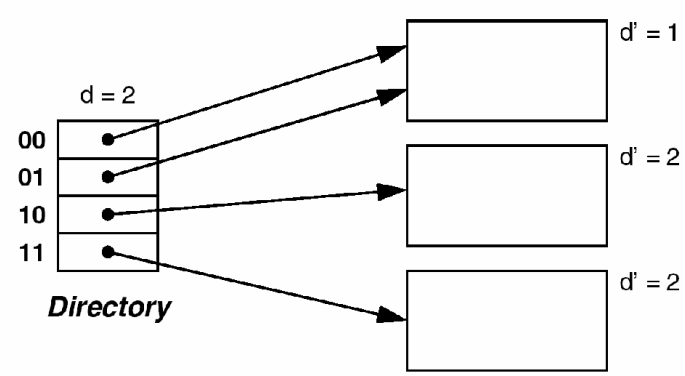
## 7.14 Erweiterbares Hashing

### Erweiterbares Hash-Verfahren (Extensible Hashing) 1/4

- Dynamisches Wachsen und Schrumpfen des Hash-Bereiches
  - Buckets werden erst bei Bedarf bereitgestellt
  - hohe Speicherplatzbelegung möglich
- Keine Überlauf-Bereiche, jedoch Zugriff über Directory

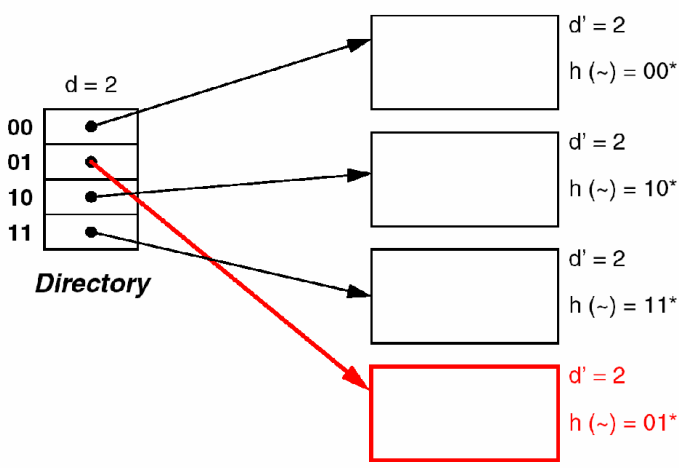
- max. 2 Seitenzugriffe
- Hash-Funktion generiert Pseudoschlüssel zu einem Satz
- d Bits des Pseudoschlüssels werden zur Adressierung verwendet (d = globale Tiefe)
- Directory enthält $2^d$ Einträge; Eintrag verweist auf Bucket, in dem alle zugehörigen Sätze gespeichert sind
- In einem Bucket werden nur Sätze gespeichert, deren Pseudoschlüssel in den ersten d' Bits übereinstimmen (d' = lokale Tiefe)
- d = MAX (d')

## Erweiterbares Hash-Verfahren (Extensible Hashing) 2/4



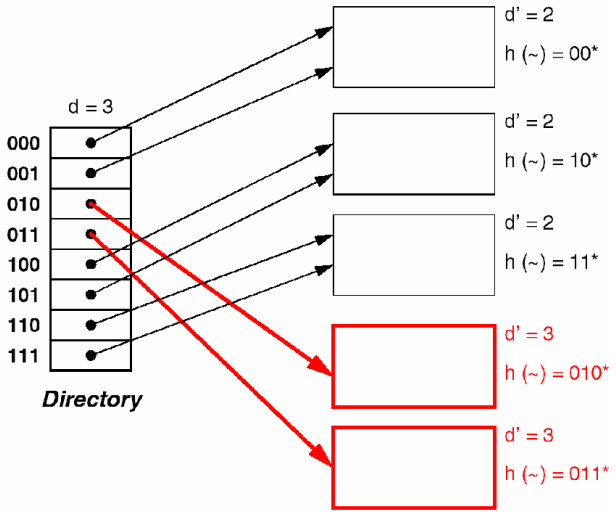

## Erweiterbares Hash-Verfahren (Extensible Hashing) 3/4

- Splitting von Buckets, Fall 1
  - Überlauf eines Buckets, dessen lokale Tiefe kleiner als die globale Tiefe d ist ⇒ lokale Neuverteilung der Daten
  - Erhöhung der lokalen Tiefe und lokale Korrektur der Pointer im Directory




## Erweiterbares Hash-Verfahren (Extensible Hashing) 4/4

- Fall 2: Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist ⇒ lokale Neuverteilung der Daten (Erhöhung der lokalen Tiefe)
  - Verdopplung des Directories (Erhöhung der globalen Tiefe)
  - globale Korrektur/Neuverteilung der Pointer im Directory




## 7.15 Lineares Hashing

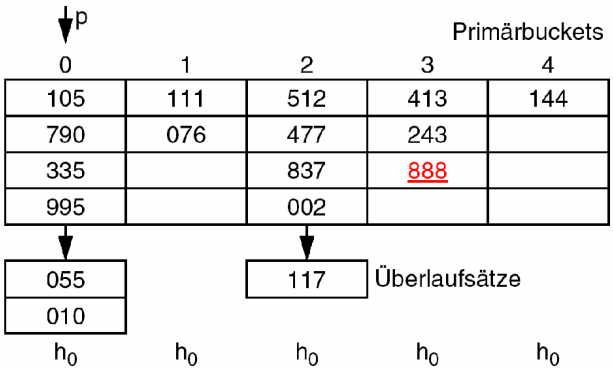
### Lineares Hashing 1/6

- Dynamisches Wachsen und Schrumpfen des (primären) Hash-Bereichs
  - minimale Verwaltungsdaten, keine großen Directories für die Hash-Datei
  - Aber: es gibt keine Möglichkeit, Überlaufsätze vollständig zu vermeiden!
  - eine hohe Rate von Überlaufsätzen wird als Indikator dafür genommen, dass die Datei eine zu hohe Belegung aufweist und deshalb erweitert werden muss
  - Buckets werden in einer fest vorgegebenen Reihenfolge gesplittet ⇒ einzige Information: nächstes zu splittendes Bucket
- Prinzipieller Ansatz
  - n: Größe der Ausgangsdatei in Buckets
  - Folge von Hash-Funktionen $h_0, h_1, ...$
  - wobei $h_0(k) \in \{0, 1, ..., n-1\}$ und $h_{j+1}(k) = h_j(k)$ oder $h_{j+1}(k) = h_j(k) + n \cdot 2^j$ für alle $j \geq 0$ und alle Schlüssel k gilt
  - gleiche Wahrscheinlichkeit für beide Fälle von $h_{j+1}$ erwünscht
  - Beispiel: $h_j(k) = k(mod n \cdot 2^j), j = 0, 1, ...$

### Lineares Hashing 2/6

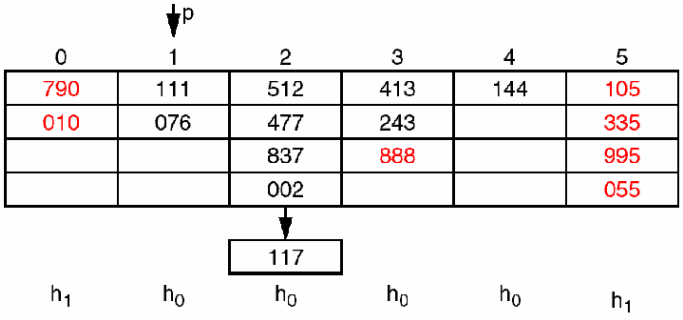
- Beschreibung des Dateizustandes
  - L: Anzahl der bereits ausgeführten Verdopplungen
  - N: Anzahl der gespeicherten Sätze
  - b: Kapazität eines Buckets
  - p: zeigt auf nächstes zu splittendes Bucket $(0 \leq p < n \cdot 2^L)$
  - $\beta$: Belegungsfaktor $= \frac{N}{(n \cdot 2^L + p) \cdot b}$
- Beispiel: Prinzip des linearen Hashing
  - $h_0(k) = k \mod 5$, $h_1(k) = k \mod 10, ...$
  - $b = 4, L = 0, n = 5$

– Splitting, sobald $\beta > \beta_S = 0.8$



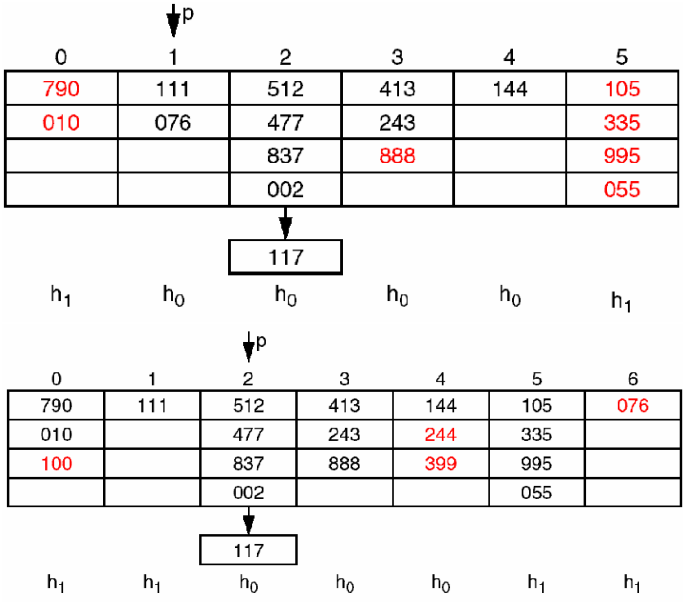

## Lineares Hashing 3/6

- Splitting
  - Einfügen von 888 erhöht Belegung auf $\beta = 17/20 = 0.85$
  - Einfügen von 244, 399 und 100 erhöht Belegung auf $\beta = 20/24 = 0.83$
  - Auslösen eines Splitting-Vorgangs



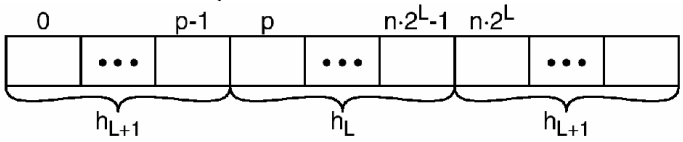

## Lineares Hashing 4/6

- Splitting-Vorgang



### Lineares Hashing 5/6

- Splitting
  - Auslöser: $\beta$, Position: p, Datei wird um 1 vergrößert
  - p wird inkrementiert: $p = (p + 1) \mod (n \cdot 2^L)$
  - Wenn p wieder auf Null gesetzt wird (Verdopplung der Datei beendet), wird L wiederum inkrementiert
- Adressberechnung
  - Wenn $h_0(k) \geq p$, dann ist $h_0$ die gewünschte Adresse
  - Wenn $h_0(k) < p$, dann war das Bucket bereits gesplittet. $h_1(k)$ liefert die gewünschte Adresse
  - Allgemein: $h = \begin{cases} H_L(k) & h \geq p \\ H_{L+1}(k) & \text{sonst} \end{cases}$




### Lineares Hashing 6/6

- Split-Strategien
  - Unkontrolliertes Splitting: Splitting, sobald ein Satz in den Überlaufbereich kommt; $\beta \approx 0.6$, schnelleres Aufsuchen
  - Kontrolliertes Splitting: Splitting, wenn ein Satz in den Überlaufbereich kommt und $\beta > \beta_S$; $\beta \approx \beta_S$, längere Überlaufketten möglich

## 7.16 Zusammenfassung Eindimensionale Zugriffspfade

### Zusammenfassung

- B-Baum / B*-Baum
  - selbstorganisierend, dynamische Reorganisation
  - garantierte Speicherplatzausnutzung
  - jeder Knoten (bis auf die Wurzel) immer mindestens halb voll, d.h. Speicherausnutzung garantiert >= 50%
  - bei zufälliger und gleichverteilter Einfügung Speicherausnutzung ln(2), also rund 70%
  - Effizientes Suchen einfach zu realisieren
  - Aufwendige Einfüge- und Löschoperationen
- Bit-Index
  - keine Hierachie, optimal für Attribute mit geringer Ausprägung und logischen Verknüpfungsoperationen
- Hashing
  - direkte Berechnung der Satzadresse
  - Problem: Dynamisches Wachstum der Datenbereiche

## Vergleich der wichtigsten Zugriffsverfahren

| Zugriffsverfahren | Speicherungsstruktur | Direkter Zugriff | Sequentielle Verarbeitung | Änderungsdienst (Ändern ohne Aufsuchen) |
|---|---|---|---|---|
| fortlaufender Schlüsselvergleich | sequentielle Liste<br>gekettete Liste | O (N) ≈ $10^4$<br>O (N) ≈ $5 \cdot 10^5$ | O (N) ≈ $2 \cdot 10^4$<br>O (N) ≈ $10^6$ | O (1) ≤ 2<br>O (1) ≤ 3 |
| Baumstrukturierter Schlüsselvergleich | Balancierte Binärbäume<br>Mehrwegbäume | O ($\log_2$N) ≈ 20<br>O ($\log_k$N) ≈ 3 - 4 | O (N) ≈ $10^6$<br>O (N) ≈ $10^{6a}$ | O (1) = 2<br>O (1) = 2 |
| Konstante Schlüssel-transformationsverfahren | Externes Hashing mit separatem Überlaufbereich<br>Externes Hashing mit Separatoren | O (1) ≈ 1.1 - 1.4<br>O (1) = 1 | O (N$\log_2$N)$^b$<br>O (N$\log_2$N)$^b$ | O (1) ≈ 1.1<br>O (1) = 1 (+D) |
| Variable Schlüsseltrans-formationsverfahren | Erweiterbares Hashing<br>Lineares Hashing | O (1) = 2<br>O (1) = 1 | O (N$\log_2$N)$^b$<br>O (N$\log_2$N)$^b$ | O (1) ≈ 1.1 (+R)<br>O (1) < 2 |

a. Bei Clusterbildung bis zu Faktor 50 geringer

b. Physisch sequentielles Lesen, Sortieren und sequentielles Verarbeiten der gesamten Sätze, Beispielangaben für N = $10^6$

# 8 Mehrdimensionale Zugriffspfade

## 8.1 Anforderungen und Probleme in mehrdimensionalen Räumen

### Anforderungen an mehrdimensionale Zugriffspfade

- Motivation: Selektion der Sätze durch Angabe von Schlüsselwerten für mehrere Felder (z.B. Name und Wohnort)
  - Wichtigstes Beispiel: räumliche Koordinaten $(x, y)$ oder $(x, y, z)$
- Anforderungen
  - Organisation räumlicher Daten
  - Erhaltung der Topologie (Cluster-Bildung)
  - raumbezogener Zugriff
- Definitionen
  - Zu einem Satztyp $R = (A_1, \ldots, A_n)$ gebe es $N$ Sätze, von denen jeder ein $n$-Tupel $t = (a_1, \ldots, a_n)$ von Werten ist. Die Attribute $A_1, \ldots, A_k$ $(k \leq n)$ seien Schlüssel
  - Eine Anfrage $Q$ spezifiziert die Bedingungen, die von den Schlüsselwerten der Sätze in der Treffermenge erfüllt sein müssen

### Mehrattribut-Anfragen

- exact-match-Anfrage: alle Schlüsselattribute sind in der Anfrage spezifiziert $Q = (A_1 = a_1) \wedge (A_2 = a_2) \wedge \ldots \wedge (A_k = a_k)$
- partial-match-Anfrage: Nur einige der Schlüsselattribute sind spezifiziert $(s < k)$, z.B. Abteilung = K55 $\wedge$ Ort = Dresden
- range-Anfrage: für einige Attribute ist ein Werteintervall angegeben, z.B. Abteilung $\geq$ K10$\wedge$ Abteilung $\leq$ K60
- Merke: das Problem ist mit herkömmlichen Zugriffspfadmethoden, wie z.B. B/B*-Baum, Hashing nicht effizient zu lösen.

### Grundprobleme

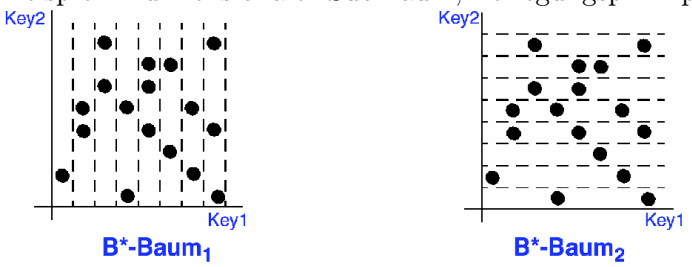
1. Erhaltung der topologischen Struktur
2. Stark variierende Dichte der Objekte
3. Objektdarstellung
   - Punktobjekte
   - Objekte mit Ausdehnung
4. Dynamische Reorganisation
5. Balancierte Zugriffsstruktur
   - beliebige Belegungen und Einfüge-/Löschreihenfolgen
   - Garantie eines gleichförmigen Zugriffs

$\Rightarrow$ 2 oder 3 Externspeicherzugriffe

## 8.2 Nutzung eindimensionaler Indexstrukturen für mehrdimensionale Daten

### Nutzung Eindimensionaler Indexstrukturen

- bisher: Nutzung eindimensionaler Indexe (non-composite index)
  - Indexierung einer Dimension (z.B. über B*-Baum)
  - Beispiel: 2-dimensionaler Suchraum, Zerlegungsprinzip über Key1 oder Key2




  - zwei B*-Bäume notwendig
  - Zugriff nach
    - Key1 → B*-Baum1
    - ...nach Key2 → B*-Baum2
  - ...aber: Zugriff nach (Key1 AND Key2) ???

### Mehrattributsuche 1/2

- Zugriff nach (Key1 AND Key2) o. (Key1 OR Key2)
  - Zeigerliste für Werte von Key1 und
  - Zeigerliste für Werte von Key2
  - Mischen und Zugriff auf Ergebnistupel
- Simulation des mehrdimensionalen Zugriffs mit einem B*-Baum
  - konkatenierte Schlüsselwerte (composite index)
  - Unterstützung für Suchoperationen:
    - (Key1 AND Key2) funktioniert
    - (Key1) funktioniert bei B1
    - (Key2) funktioniert bei B2
    - OR-Verknüpfung ???
  - Skalierung bei $n$ Dimensionen

### Mehrattributsuche 2/2

**B*-Baum $B_1$**

| $Key_1$ | $Key_2$ |
|---|---|
| $A_1$ | $B_1$ |
| $A_1$ | $B_2$ |
| $A_1$ | $B_m$ |
| $A_2$ | $B_1$ |
| $A_2$ | $B_m$ |
| ... | ... |
| $A_n$ | B1 |
| $A_n$ | $B_m$ |

**B*-Baum $B_2$**

| $Key_2$ | $Key_1$ |
|---|---|
| $B_1$ | $A_1$ |
| $B_1$ | $A_2$ |
| $B_1$ | $A_n$ |
| $B_2$ | $A_1$ |
| $B_2$ | $A_n$ |
| ... | ... |
| $B_m$ | A1 |
| $B_m$ | $A_n$ |

### Mischen eindimensionaler Indexe in Oracle

- Beispiel
  - TPC-H Relation `Customer` mit Indizes

```
CREATE INDEX cust_nationkey ON customer(c_nationkey);
CREATE INDEX cust_mktsegment ON customer(c_mktsegment);
```

  - SELECT-Anweisung (würde auch ohne Hint funktionieren!)

```
SELECT /*+AND_EQUAL(t cust_mktsegment cust_nationkey)*/ *
  FROM CUSTOMER t
 WHERE C_NATIONKEY = 7
   AND C_MKTSEGMENT = 'AUTOMOBILE';
```

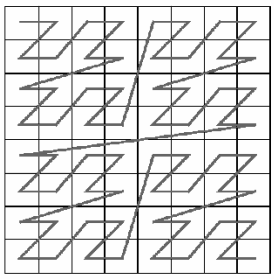
  - Query-Plan

```
SELECT STATEMENT Optimizer Mode=CHOOSE
  TABLE ACCESS BY INDEX ROWID TPCH.CUSTOMER
    AND-EQUAL
      INDEX RANGE SCAN TPCH.CUST_MKTSEGMENT
      INDEX RANGE SCAN TPCH.CUST_NATIONKEY
```

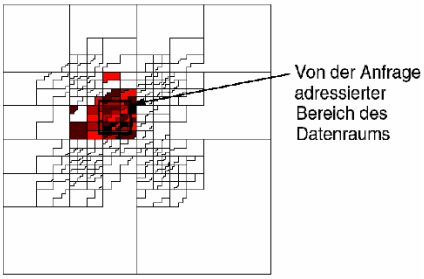
- AND_EQUAL
  - Mischen von ROWIDs gleichartiger Indexe → Indexkonvertierung

### Linearisierung multidimensionaler Räume

- Idee: Linearisierung eines multidimensionalen Raumes und Nutzung eindimensionaler Indexstrukturen
- UB-Baum: Linearisierung durch raumfüllende Z-Kurve; möglichst nachbarschaftserhaltend



a) Linearisierung über Z-Kurve




b) Zugriffe bei Bereichsanfragen

  - beim Zugriff wird mehr als nötig gelesen!
  - entartet in hochdimensionalen Räumen

## 8.3 Raum-organisierende mehrdimensionale Indexstrukturen

### Organisation des umgebenden Datenraums – Divide and Conquer

- Zerlegungsprinzip
  - Datenraum wird dynamisch in Zellen aufgeteilt
  - Objekte einer Zelle als Sätze in Buckets ablegen
  - bei Bucket-Überlauf: lokale Zellverfeinerung → Divide and Conquer
  - abschnittsweise Erhaltung der Topologie
  - Baum als Zugriffsstruktur für die Buckets hat nur Wegweiserfunktion
- Beispiel: Heterogener k-d-Baum mit k=2

- Eigenschaften des k-d-Baumes
  - Clusterbildung durch Buckets ist Voraussetzung für praktischen Einsatz
  - Eingebauter Balancierungsmech. nicht vorhanden
  - Wie werden Aktualisierungsop. (Löschen!) durchgeführt?
  - Wie werden die verschiedenen Anfragetypen unterstützt?

## 8.4 k-d-B-Baum

### k-d-B-Baum 1/2

- Lösungsidee: Kombination von k-d- und B*-Baum
  - k-d-B-Baum paginiert k-d-Bäume und ordnet ihren Teilbäumen Buckets (Seiten) zu, wie das bei B*-Bäumen der Fall ist
  - auf jeder Baumebene wird der k-dimensionale Datenraum in schachtelförmige Zellen oder Regionen (bei k=2 in Rechtecke) partitioniert, wobei eine Region jeweils die Zellen/Regionen eines Knotens der darunterliegenden Ebene zusammenfaßt und repräsentiert
  - alle Datensätze in Blättern (Buckets) gespeichert
  - die inneren Knoten (Index- oder Directory-Seiten) besitzen nur Wegweiser

### k-d-B-Baum 2/2

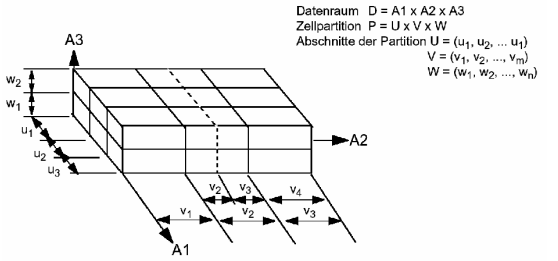
- Lösungsidee: Kombination von k-d- und B*-Baum
  - operationales Verhalten wie beim B*-Baum

    sehr komplexe Regeln für dynamische Reorganisation
  - Bucket-Überlauf erzwingt Split der Zelle – optimierte Aufteilung der Zelle möglich (z. B. Median-Split)
  - Fortpflanzung des Split zur Wurzel hin möglich
  - bei Split von Indexseiten ist manchmal eine Neuaufteilung von Regionen erforderlich (sonst können sich sehr ungünstige Strukturen ergeben), was einen sich zu den Blattknoten hin fortpflanzenden Split (forced split) auslöst!

## 8.5 Grid-File

### Organisation des umgebenden Datenraums – Dimensionsverfeinerung 1/2

- Prinzip: Datenraum D wird dynamisch durch ein orthogonales Raster (grid) partitioniert, so daß k-dimensionale Zellen (Grid-Blöcke) entstehen
  - die in den Zellen enthaltenen Objekte werden Buckets zugeordnet
  - es muß eine eindeutige Abbildung der Zellen zu den Buckets gefunden werden
- Beispiel:



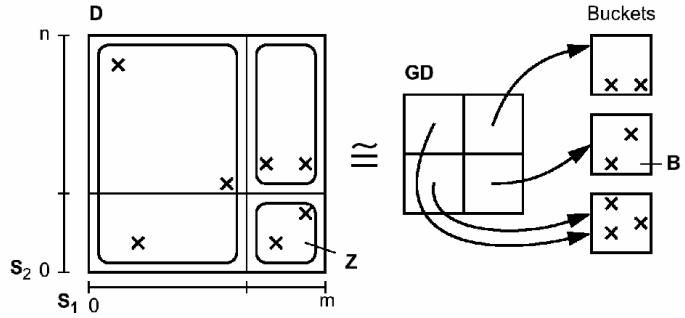

Dreidimensionaler Datenraum D mit Zellpartition P; Veranschaulichung eines Split-Vorganges im Intervall $v_2$

### Organisation des umgebenden Datenraums – Dimensionsverfeinerung 2/2

- Probleme der Dimensionsverfeinerung
  - Wieviele neue Zellen enstehen jedesmal?
  - Was folgt für die Bucketzuordnung?
  - Welche Abbildungsverfahren können gewählt werden?
  - Gibt es Einschränkungen bei der Festlegung der Dimensionsverfeinerung?

### Grid-File: Idee 1/2

- Zerlegungsprinzip von D: Dimensionsverfeinerung




- Komponenten
  - k Skalierungsvektoren (Scales) definieren die Zellen (Grid) auf k-dim. Datenraum D
  - Zell- oder Grid-Directory GD: dynamische k-dim. Matrix zur Abbildung von D auf die Menge der Buckets
  - Bucket: Speicherung der Objekte einer oder mehrerer Zellen (Bucketbereich BB)

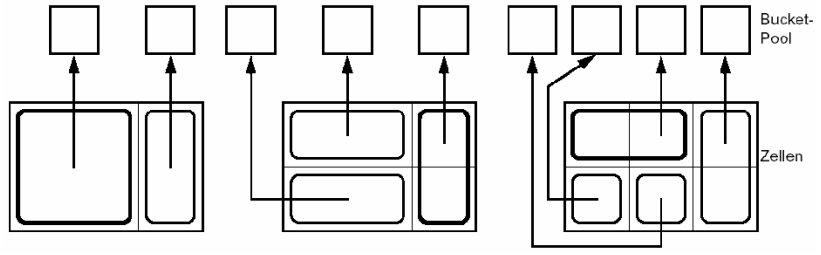
### Grid-File: Idee 2/2

- Eigenschaften
  - 1:1-Beziehung zwischen Zelle $Z_i$ und Element von GD
  - Element von GD = Ptr. zu Bucket B
  - n:1-Beziehung zwischen $Z_i$ und B
- Ziele
  - Erhaltung der Topologie
  - effiziente Unterstützung aller Fragetypen
  - vernünftige Speicherplatzbelegung

### Zentrale Datenstruktur: Grid-Directory

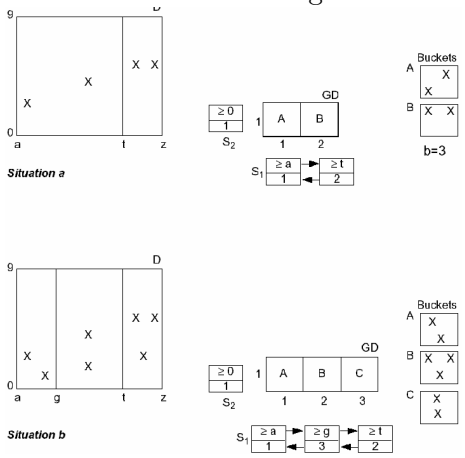
- Anforderungen
  - Prinzip der zwei Plattenzugriffe unabhängig von Werteverteilungen, Operationshäufigkeiten und Anzahl der gespeicherten Sätze
  - Split- und Mischoperationen jeweils nur auf zwei Buckets
  - Speicherplatzbelegung
    - durchschnittliche Belegung der Buckets nicht beliebig klein
    - schiefe Verteilungen vergrößern nur GD
- Entwurf einer Directory-Struktur
  - dynamische k-dim. Matrix GD (auf Externspeicher)
  - $k$ eindim. Vektoren $S_i$ (im Hauptspeicher)
- Operationen auf GD
  - direkter Zugriff auf einen GD-Eintrag
  - relativer Zugriff (Nextabove, Nextbelow)
  - Mischen zweier benachbarter Einträge einer Dimension (mit Umbenennung der betroffenen Einträge)
  - Splitten Eintrages einer Dimension (mit Umbenennung)

### Schachtelförmige Zuweisung von Zellen zu Buckets

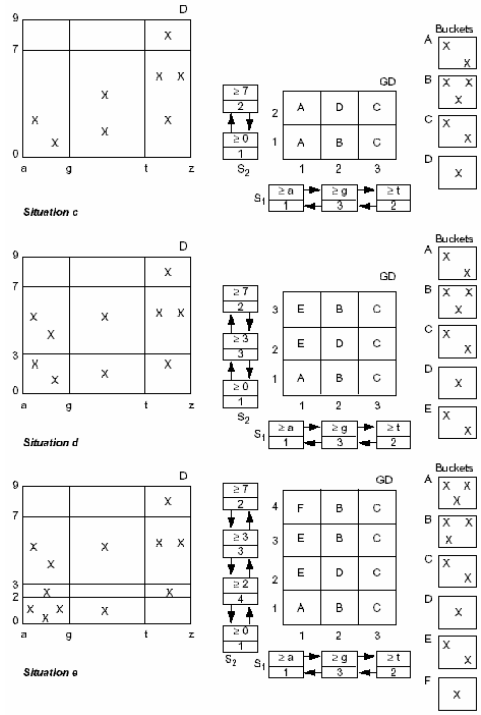


## Grid-File - Beispiel

- Skalierungsvektoren als zweifach gekettete Listen
  - Indirektion erlaubt, GD an Rändern wachsen zu lassen.
  - Minimierung des Änderungsdienstes von GD
  - Stabilität der GD-Einträge
- Schrittweise Entwicklung eines GF



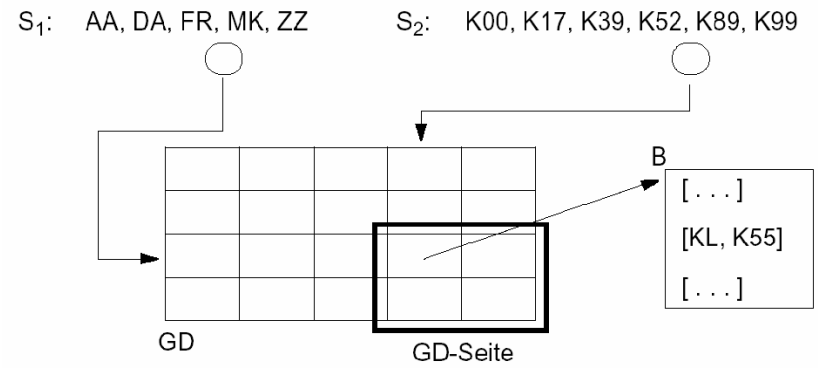

- Verfeinerung des Datenraums, zugehörige Entwicklung des GF




## Grid-File – Suchfragen 1/2

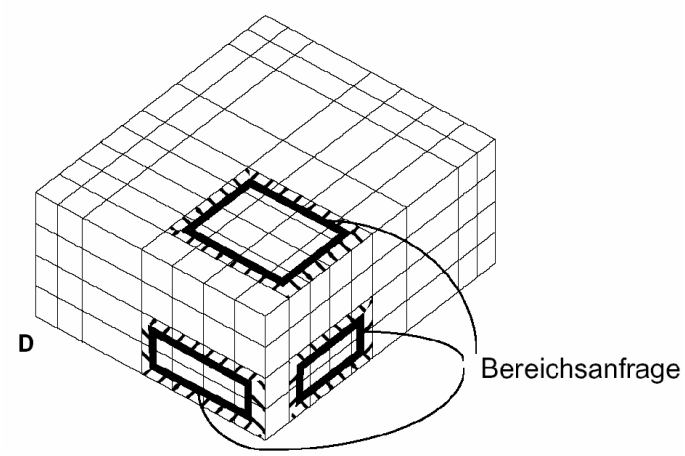
- Exakte Anfrage (exact match)

```
SELECT *
    FROM        PERS
    WHERE       ORT = 'KL' AND ANR = 'K55'
```




### Grid-File – Suchfragen 2/2

- Bereichsanfrage
  - Bestimmung der Skalierungswerte in jeder Dimension
  - Berechnung der qualifizierten GD-Einträge
  - Zugriff auf die GD-Seite(n) und Holen der referenzierten Buckets




## 8.6 Verallgemeinerung auf ausgedehnte räumliche Objekte: R- und R*-Baum

### Zugriffspfade für ausgedehnte räumliche Objekte

- Ausgedehnte räumliche Objekte besitzen
  - allgemeine Merkmale wie Name, Beschaffenheit, ...
  - Ort und Geometrie (Kurve, Polygon, ...)
- Indexierung des räumlichen Objektes
  - genaue Darstellung?
  - Objektapproximation durch schachtelförmige Umhüllung - effektiv!

  $\Rightarrow$ dadurch werden Fehltreffer möglich
- Probleme

- neben Objektdichte muß Objektausdehnung bei der Abbildung und Verfeinerung berücksichtigt werden
- Objekte können andere enthalten oder sich gegenseitig überlappen

### R-Baum 1/4

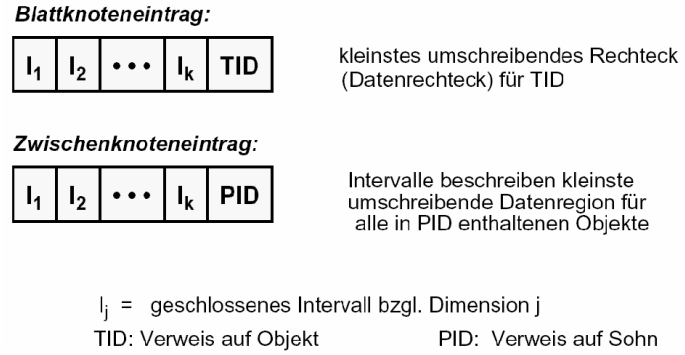
- Ziel: Effiziente Verwaltung räumlicher Objekte (Punkte, Polygone, Quader, ...)
- Anwendungen
  - Kartographie: Speicherung von Landkarten, effiziente Beantwortung geometrischer Fragen
  - CAD: Handhabung von Flächen, Volumina und Körpern (z.B. Rechtecke beim VLSI-Entwurf)
  - Computer-Vision und Robotics
- Hauptoperationen
  - Punktanfragen (point queries): Finde alle Objekte, die einen gegebenen Punkt enthalten
  - Gebietsanfragen (region queries): Finde alle Objekte, die mit einem gegebenen Suchfenster überlappen (in ... vollständig enthalten sind)

### R-Baum 2/4

- Ansatz: Speicherung und Suche von achsenparallelen Rechtecken
  - Objekte werden durch Datenrechtecke repräsentiert und müssen durch kartesische Koordinaten beschrieben werden
  - Repräsentation im R-Baum erfolgt durch minimale begrenzende (k-dimensionale) Rechtecke/Regionen
  - Suchanfragen beziehen sich ebenfalls auf Rechtecke/Regionen

### R-Baum 3/4

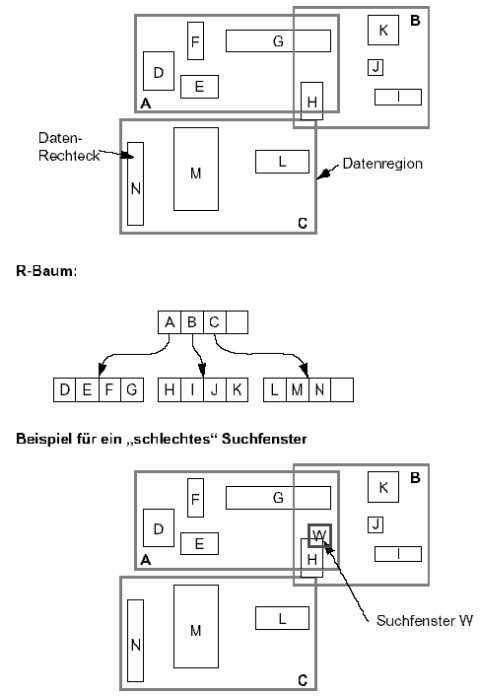
- R-Baum ist höhenbalancierter Mehrwegbaum
  - jeder Knoten entspricht einer Seite
  - pro Knoten maximal M, minimal $m$ ($\geq M/2$) Einträge
  - alle Blätter sind auf einer Ebene




### R-Baum 4/4

- Eigenschaften
  - starke Überlappung der umschreibenden Rechtecke/Regionen auf allen Baumebenen möglich
  - bei Suche nach Rechtecken/Regionen sind ggf. mehrere Teilbäume zu durchlaufen
  - Änderungsoperationen ähnlich wie bei B-Bäumen

### Abbildung beim R-Baum




### R-Baum Aufbau

- Aufbau ist entscheidend für Performanz: Wie wird ein Knoten geteilt, wenn er $M + 1$ Einträge hat
- Aufgabe: Teile M+1 MBRs in zwei Teilmengen
- Kriterien:
  1. Fläche sollte klein sein
  2. Überlappung zwischen Einträgen sollte klein sein
  3. Umfang eines Eintrages sollte klein sein
  4. Platzausnutzung sollte hoch sein
- R-Baum nutzt nur Fläche (1) als Kriterium
- besser wäere Balance zw. 1-3 und 4
- Algorithmen: linear, quadratisch, exponentiell

### R-Baum Aufbau

- Einfügen eines Objektes: wähle Blatt mittels ChooseSubtree

**Algorithm ChooseSubtree**
**CS1** Set N to be the root
**CS2** If N is a leaf,
return N
else
Choose the entry in N whose rectangle needs least area enlargement to include the new data Resolve ties by choosing the entry with the rectangle of smallest area
end
**CS3** Set N to be the childnode pointed to by the childpointer of the chosen entry an repeat from CS2

## R-Baum Algorithmen

**Algorithm QuadraticSplit**
[Divide a set of M+1 entries into two groups]
QS1 Invoke PickSeeds to choose two entries to be the first entries of the groups
QS2 Repeat
DistributeEntry
until
all entries are distributed or
one of the two groups has M-m+1 entries
QS3 If entries remain, assign them to the other group such that it has the minimum number m

**Algorithm PickSeeds**
PS1 For each pair of entries E1 and E2, compose a rectangle R including E1 rectangle and E2 rectangle
Calculate d = area(R) - area(E1 rectangle) - area(E2 rectangle)
PS2 Choose the pair with the largest d

## R-Baum Algorithmen

**Algorithm DistributeEntry**
DE1 Invoke PickNext to choose the next entry to be assigned
DE2 Add it to the group whose covering rectangle will have to be enlarged least to accommodate it Resolve ties by adding the entry to the group with the smallest area, then to the one with the fewer entries, then to either

**Algorithm PickNext**
PN1 For each entry E not yet in a group, calculate $d_1$ = the area increase required in the covering rectangle of Group 1 to include E Rectangle
Calculate $d_2$ analogously for Group 2
PN2 Choose the entry with the maximum difference between $d_1$ and $d_2$

## Beobachtung – Nachteile

- PickSeeds
  - wählt entfernte Einträge
  - tendiert zu kleinen Einträgen
- PickNext
  - kleine entfernte Einträge mit d-1 gleichen Koordinaten $\Rightarrow$ lange schmale MBRs
  - bevorzugt zuerst erweiterten Kandidaten

## R*-Baum 1/2

- Kombiniert alle Kriterien, Überlappung, Rand und Fläche, Kombination wurde durch Ingenieur-Ansatz gefunden (2D-Daten), kein mathematischen Garantien
- Überlappung eines Eintrags

– Seien $E_1, \ldots, E_p$ die Einträge eines Knotens, dann ist die Überlappung

$$overlap(E_k) = \sum_{i=1, i \neq k}^{p} area(E_k.mbr \cap E_l.mbr), \quad 1 \leq k \leq p$$

.

## R*-Baum 2/2

**Algorithm ChooseSubtree**
CS1 Set N to be the root
CS2 If N is a leaf,
return N
else
if the childpointers in N point to leaves [determine
the minimum overlap cost],
choose the entry in N whose rectangle needs least
overlap enlargement to include the new data
rectangle Resolve ties by choosing the entry
whose rectangle needs least area enlargement,
then
the entry with the rectangle of smallest area
if the childpointers in N do not point to leaves
[determine the minimum area cost],
choose the entry in N whose rectangle needs least
area enlargement to include the new data
rectangle Resolve ties by choosing the entry
with the rectangle of smallest area
end
CS3 Set N to be the childnode pointed to by the
childpointer of the chosen entry and repeat from CS2

## R*-Baum, Aufteilungs-Algorithmen

**Algorithm Split**
S1 Invoke ChooseSplitAxis to determine the axis,
perpendicular to which the split is performed
S2 Invoke ChooseSplitIndex to determine the best
distribution into two groups along that axis
S3 Distribute the entries into two groups

**Algorithm ChooseSplitAxis**
CSA1 For each axis
Sort the entries by the lower then by the upper
value of their rectangles and determine all
distributions as described above Compute S, the
sum of all margin-values of the different
distributions
end
CSA2 Choose the axis with the minimum S as split axis

**Algorithm ChooseSplitIndex**
CSI1 Along the chosen split axis, choose the
distribution with the minimum overlap-value
Resolve ties by choosing the distribution with
minimum area-value

## R*-Baum, ReInsert

- R*-Baum ist abhängig von Einfüge-Sequenz

- Erneutes Einfügen alter Einträge ⇒ bessere Raumorganisation

**Algorithm InsertData**

ID1 Invoke Insert starting with the leaf level as a parameter, to insert a new data rectangle

**Algorithm Insert**

I1 Invoke ChooseSubtree, with the level as a parameter, to find an appropriate node N, in which to place the new entry E

I2 If N has less than M entries, accommodate E in N
If N has M entries, invoke OverflowTreatment with the level of N as a parameter [for reinsertion or split]

I3 If OverflowTreatment was called and a split was performed, propagate OverflowTreatment upwards if necessary
If OverflowTreatment caused a split of the root, create a new root

I4 Adjust all covering rectangles in the insertion path such that they are minimum bounding boxes enclosing their children rectangles

## R*-Baum, ReInsert

**Algorithm OverflowTreatment**

OT1 If the level is not the root level and this is the first call of OverflowTreatment in the given level during the insertion of one data rectangle, then
invoke ReInsert
else
invoke Split
end

**Algorithm ReInsert**

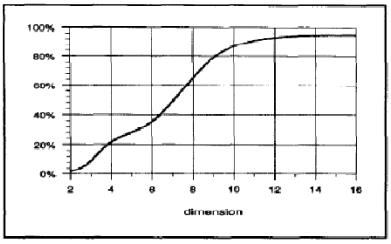
RI1 For all M+1 entries of a node N, compute the distance between the centers of their rectangles and the center of the bounding rectangle of N

RI2 Sort the entries in decreasing order of their distances computed in RI1

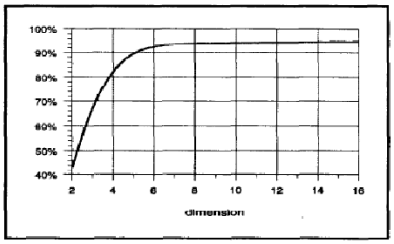
RI3 Remove the first p entries from N and adjust the bounding rectangle of N

RI4 In the sort, defined in RI2, starting with the maximum distance (= far reinsert) or minimum distance (= close reinsert), invoke Insert to reinsert the entries

## R*-Baum, Überlappung




a. Overlap (Uniformly Distributed Data)




b. Weighted Overlap (Real Data)

$$Overlap = \frac{\left\| \bigcup_{i,j \in \{1\ldots n\}, i \neq j} (R_i \cap R_j) \right\|}{\left\| \bigcup_{i \in \{1\ldots n\}} R_i \right\|} \qquad WeightedOverlap = \frac{\left| \left\{ p \mid p \in \bigcup_{i,j \in \{1\ldots n\}, i \neq j} (R_i \cap R_j) \right\} \right|}{\left| \left\{ p \mid p \in \bigcup_{i \in \{1\ldots n\}} R_i \right\} \right|}$$

- Viele Suchpfade mit Random Disk Access

## X-Baum

- Idee: wenn keine überlappungsfreie Aufteilung gefunden wird, dann nicht teilen sondern Superknoten bilden
- Superknoten können seq. gelesen werden

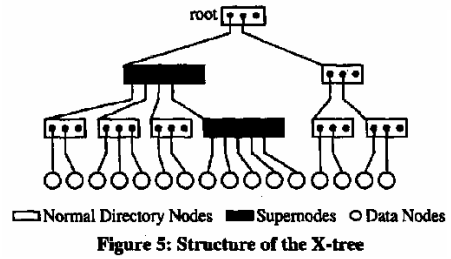


Figure 5: Structure of the X-tree

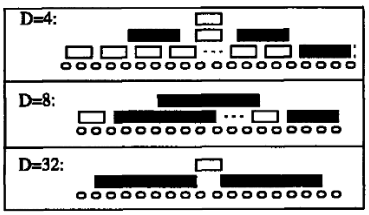


Figure 6: Various Shapes of the X-tree in different dimensions

- Kompromiss aus Baum und sequentieller Scan

# 9 Textsuche mit invertierten Listen

## 9.1 Motivation für Indexstrukturen

### Textsuche mit Indexstrukturen

- Wann lohnt es sich zu einer Anfrage alle Dokumente sequentiell durchsuchen?
  - Kleine Textsammlungen (ein paar MB)
  - Sehr viele Änderungen
- Indexstrukturen
  - Mehraufwand lohnt sich erst für große Textsammlungen
  - Texte sollten mindestens semi-statisch sein
- Typen
  - Inverted Files
  - Suffix Bäume
  - Signature Files

### Beispiel für Anfragen

- Welches Stück von Shakespeare enthält die Wörter Brutus, Caesar aber NICHT Calpurnia?
- Ansatz: finde alle Stücke mit Brutus und Caesar und entferne alle die Calpurnia enthalten
  - Langsam für große Sammlungen
  - NICHT Calpurnia ist nicht trivial

### Term-Dokument Inzidenz Matrix

Beispiel für eine Boolsche Term-Dokument-Matrix


### Inzidenz Vektoren

- 0/1 Vektoren für jeden Term
- Anfrage beantworten: bit weises AND der Vektoren für Brutus, Caesar und Calpurnia (komplementiert)

$$110100 \wedge 110111 \wedge 101111 = 100100$$

- Antworten
  - Antony and Cleopatra
  - Hamlet

### Größere Sammlungen

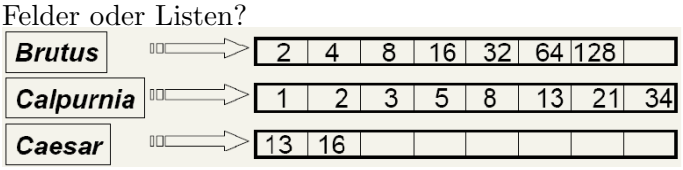
- Annahmen
  - $n = 10^6$ Dokumente, jedes mit etwa $10^3$ Termen
  - Durchschnittlich 6 Bytes/Term mit Leerzeichen/Punktuation
    - 6GB Textdaten in den Dokumenten
  - $m = 500 \cdot 10^3$ verschiedene Terme
- Matrix kann nicht materialisiert werden
  - $500 \cdot 10^3 \cdot 10^6 = 1/2 \cdot 10^12$ Einträge
  - Nicht mehr als $10^9$ Eins-Einträge, Warum?
- Besserer Ansatz: Repräsentiere nur die Einsen

## 9.2 Invertierte Listen: Darstellung

### Invertierte Listen (Inverted Files)

Zu jedem Term $t$ speichere alle Dokumente, die $t$ enthalten.

- Felder oder Listen?



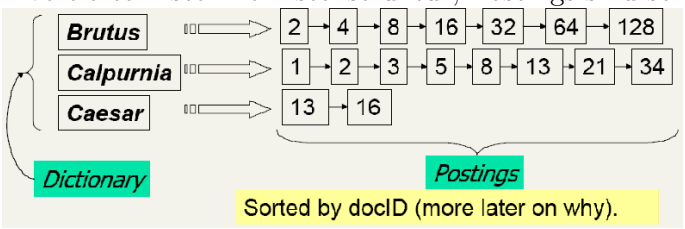

Schematische Darstellung eines Beispiels für invertierte Listen

- Was passiert, wenn das Wort Caesar zu Dokument 14 hinzugefügt wird?

### Invertierte Listen mit Listenstruktur

Listen werden meist Feldern vorgezogen

- Dynamischer Platz
- Einfügen von Termen in Dokumenten ist leicht
- Preis: Mehrverbrauch an Platz für Zeiger
- Invertierte Liste mit Listenstruktur, Postings sind sortiert.



## 9.3 Invertierte Listen: Aufbau

### Index Schritte

Liste von Paaren (Termen, Dokument ID)

**Dokument 1** I did enact Julius Caesar. I was killed i' the Capitol; Brutus killed me.

**Dokument 2** So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.

⇒ Sortierte Liste von (Term, Dokument ID)-Paaren.

| Term | Doc # |
|---|---|
| I | 1 |
| did | 1 |
| enact | 1 |
| julius | 1 |
| caesar | 1 |
| I | 1 |
| was | 1 |
| killed | 1 |
| i' | 1 |
| the | 1 |
| capitol | 1 |
| brutus | 1 |
| killed | 1 |
| me | 1 |
| so | 2 |
| let | 2 |
| it | 2 |
| be | 2 |
| with | 2 |
| caesar | 2 |
| the | 2 |
| noble | 2 |
| brutus | 2 |
| hath | 2 |
| told | 2 |
| you | 2 |
| caesar | 2 |
| was | 2 |
| ambitious | 2 |

### Sortieren der Terme

- Hauptschritt für Indexing

| Term | Doc # |
|---|---|
| I | 1 |
| did | 1 |
| enact | 1 |
| julius | 1 |
| caesar | 1 |
| I | 1 |
| was | 1 |
| killed | 1 |
| i' | 1 |
| the | 1 |
| capitol | 1 |
| brutus | 1 |
| killed | 1 |
| me | 1 |
| so | 2 |
| let | 2 |
| it | 2 |
| be | 2 |
| with | 2 |
| caesar | 2 |
| the | 2 |
| noble | 2 |
| brutus | 2 |
| hath | 2 |
| told | 2 |
| you | 2 |
| caesar | 2 |
| was | 2 |
| ambitious | 2 |

| Term | Doc # |
|---|---|
| ambitious | 2 |
| be | 2 |
| brutus | 1 |
| brutus | 2 |
| capitol | 1 |
| caesar | 1 |
| caesar | 2 |
| caesar | 2 |
| did | 1 |
| enact | 1 |
| hath | 1 |
| I | 1 |
| I | 1 |
| i' | 1 |
| it | 2 |
| julius | 1 |
| killed | 1 |
| killed | 1 |
| let | 2 |
| me | 1 |
| noble | 2 |
| so | 2 |
| the | 1 |
| the | 2 |
| told | 2 |
| you | 2 |
| was | 1 |
| was | 2 |
| with | 2 |

### Mehrfache Einträge

- Mehrfache Einträge in einzelnen Dokumenten zusammenfassen

- Häufigkeiten hinzufügen

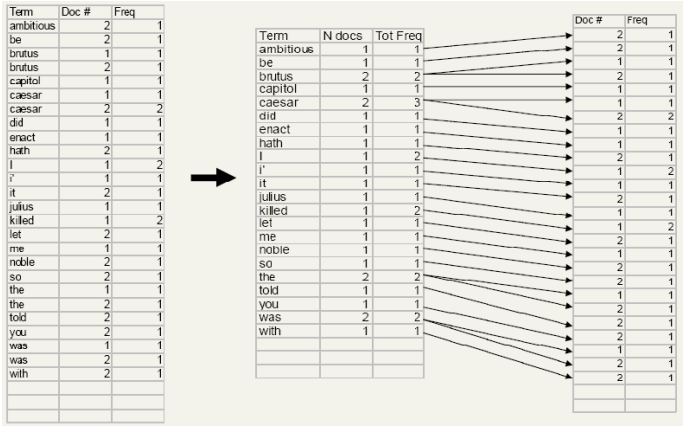
| Term | Doc # |
|---|---|
| ambitious | 2 |
| be | 2 |
| brutus | 1 |
| brutus | 2 |
| capitol | 1 |
| caesar | 1 |
| caesar | 2 |
| caesar | 2 |
| did | 1 |
| enact | 1 |
| hath | 1 |
| I | 1 |
| I | 1 |
| i' | 1 |
| it | 2 |
| julius | 1 |
| killed | 1 |
| killed | 1 |
| let | 2 |
| me | 1 |
| noble | 2 |
| so | 2 |
| the | 1 |
| the | 2 |
| told | 2 |
| you | 2 |
| was | 1 |
| was | 2 |
| with | 2 |



| Term | Doc # | Freq |
|---|---|---|
| ambitious | 2 | 1 |
| be | 2 | 1 |
| brutus | 1 | 1 |
| brutus | 2 | 1 |
| capitol | 1 | 1 |
| caesar | 1 | 1 |
| caesar | 2 | 2 |
| did | 1 | 1 |
| enact | 1 | 1 |
| hath | 2 | 1 |
| I | 1 | 2 |
| i' | 1 | 1 |
| it | 2 | 1 |
| julius | 1 | 1 |
| killed | 1 | 2 |
| let | 2 | 1 |
| me | 1 | 1 |
| noble | 2 | 1 |
| so | 2 | 1 |
| the | 1 | 1 |
| the | 2 | 1 |
| told | 2 | 1 |
| you | 2 | 1 |
| was | 1 | 1 |
| was | 2 | 1 |
| with | 2 | 1 |

### Teile Ergebnis in Dictionary und Postings

- Aufteilung der Sortierten Liste in Dictionary und Postings

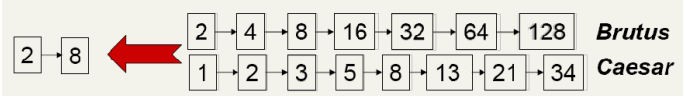



## 9.4 Invertierte Listen: Anfragebearbeitung

### Anfrageverarbeitung

Anfrage: Brutus AND Caesar

- Finde Postings von Brutus und Caesar
- Fasse beide sortierte Listen zusammen, durchlaufe beide Listen simultan
- Zusammenfassung der sortierten Listen, Sort-Merge



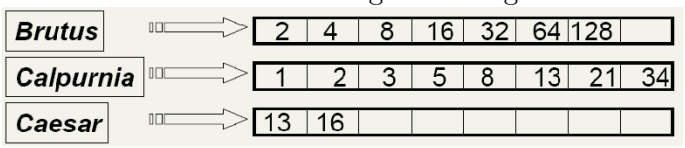

- Wenn Listenlängen $x$ und $y$ sind, Mengenschnitt in $O(x + y)$

### Allgemeinere Anfragen

- Wie muß das Zusammenfassen geändert werden für
  - Brutus AND NOT Caesar
  - Brutus OR Caesar
  - Brutus OR NOT Caesar
- Bleibt die Laufzeit bei $O(x + y)$
- Was passiert bei allg. Boolschen Anfragen?
  - (Brutus or Caesar) and not (Antony or Cleopatra)
  - Kann man immer in linearer Zeit zusammenfassen
  - Linear in was?

### Anfrage Optimierung

- Was ist die beste Bearbeitungsreihenfolge?




- Bearbeite in aufsteigender Häufigkeit (AND-Verknüpfung)
- Ausführung: (Caesar and Brutus) and Calpurnia
- Schätze Größe eines OR als Summe der Einzelhäufigkeiten (Konservativ)

### Beispiel Bearbeitungsreihenfolge

- Empfehle eine Reihenfolge

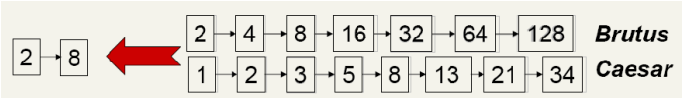
*(tangerine OR trees) AND*
*(marmalade OR skies) AND*
*(kaleidoscope OR eyes)*

| Term | Freq |
|---|---|
| eyes | 213312 |
| kaleidoscope | 87009 |
| marmalade | 107913 |
| skies | 271658 |
| tangerine | 46653 |
| trees | 316812 |

## 9.5 Skippointer

### Wiederholung, Invertierte Liste

- Anfrage mit zwei Termen (logisches UND)
  - Merge-Operation

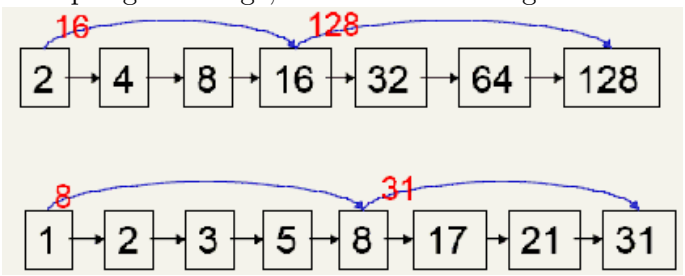



    * Merge-Operation durchläuft beide Listen simultan, lineare Laufzeit in Größe der Listen

  - Wenn die Listen die Größen $x$ und $y$ haben, ist die Laufzeit $O(x + y)$.
- Kann die Laufzeit verbessert werden?
  - Asymptotisch nicht
  - Reale Laufzeit ja, wenn der Index sich nicht oft ändert

### Schnelleres Zusammenfassen durch Skip Pointer

- überspringe Einträge, die nicht in das Ergebnis kommen

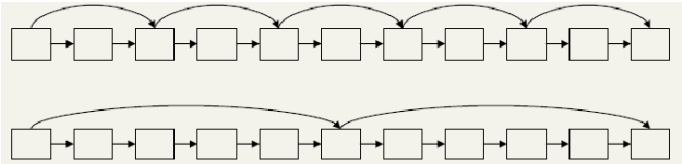



  - Beispiel für Skip-Pointer
- Warum ist das sinnvoll?
- Um Dokument IDs zu überspringen, die nicht im Ergebnis auftauchen werden.
- Wie?
- Wo kommen die Skip Pointer am besten hin?

### Wohin Skip Pointer plazieren? (1/2)

Zielkonflikt

- Mehr Skips ⇒ kürzere Sprünge ⇒ wahrscheinlicher zu springen. Aber viele Zusatzvergleiche.
- Weniger Skips ⇒ weniger Zusatzvergleiche, aber weitere Sprünge ⇒ weniger erfolgreiche Skips



- Zielkonflikt zwischen Anzahl der erfolgreichen Sprünge über Skippointer und Anzahl der Zusatzvergleiche.

### Wohin Skip Pointer plazieren? (2/2)

- Einfache Heuristik
  - Für Postings der Länge $L$, plaziere $\sqrt{L}$ gleichverteilte Skip Pointer
  - Diese Heuristik ignoriert die Verteilung der Anfrageterme.
  - Einfach zu implementieren, falls es wenig Änderungen im Index gibt
- Wie könnte die Verteilung der Anfrageterme auf die Dokumente genutzt werden?

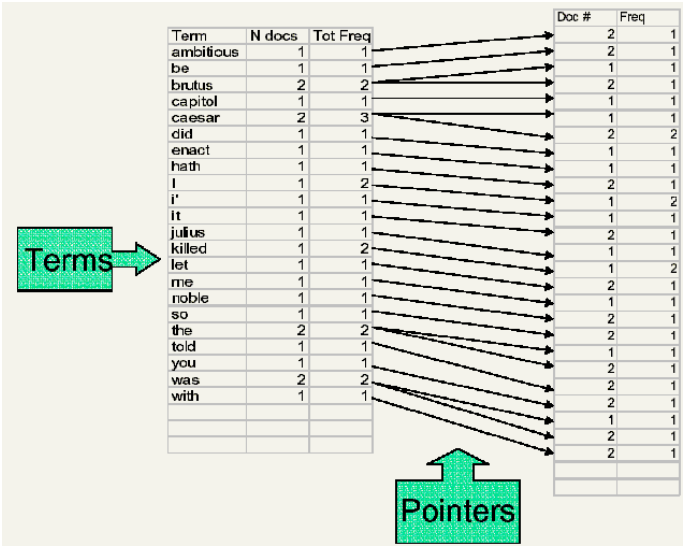
## 9.6 Komprimierung der Posting-Listen

### Komprimierung der Posting-Listen, Rückblick

- Angenommen, wir haben
  - $n = 10^6$ Dokumente, jedes mit etwa $10^3$ Termen.
  - Durchschnittlich werden 6 Bytes pro Term benötigt
  - Dies macht 6GB Daten
- Angenommen, es gibt, $m = 500 \cdot 10^3$ unterschiedliche Terme
- Term-Dokument Matrix kann nicht voll repräsentiert werden
  - $0.5 \cdot 10^{12}$ 0/1-Einträge, aber nur $10^9$ Einsen
  - Deshalb werden invertierte Listen genutzt
- Wo investieren wir in Speicherplatz?

### Wo investieren wir in Speicherplatz?

- Wo wird der meiste Speicherplatz ausgegeben?




| Term | N docs | Tot Freq |
|---|---|---|
| ambitious | 1 | 1 |
| be | 1 | 1 |
| brutus | 2 | 2 |
| capitol | 1 | 1 |
| caesar | 2 | 3 |
| did | 1 | 1 |
| enact | 1 | 1 |
| hath | 1 | 1 |
| I | 1 | 2 |
| i' | 1 | 1 |
| it | 1 | 1 |
| julius | 1 | 1 |
| killed | 1 | 2 |
| let | 1 | 1 |
| me | 1 | 1 |
| noble | 1 | 1 |
| so | 1 | 1 |
| the | 2 | 2 |
| told | 1 | 1 |
| you | 1 | 1 |
| was | 2 | 2 |
| with | 1 | 1 |

| Doc # | Freq |
|---|---|
| 2 | 1 |
| 2 | 1 |
| 1 | 1 |
| 2 | 1 |
| 1 | 1 |
| 1 | 1 |
| 2 | 2 |
| 1 | 1 |
| 1 | 1 |
| 2 | 1 |
| 1 | 2 |
| 1 | 1 |
| 2 | 1 |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 1 | 1 |
| 2 | 1 |
| 2 | 1 |
| 1 | 1 |
| 2 | 1 |
| 2 | 1 |
| 2 | 1 |
| 1 | 1 |
| 2 | 1 |
| 2 | 1 |

### Zeiger: zwei widersprüchliche Ziele

- Term Calpurnia kommt möglicherweise nur einem Dokument aus einer Million Dokumenten vor
  - Platz für Zeiger $\log_2 10^6 \approx 20$ Bits
- Term "the" kommt in nahezu jedem Dokument vor
  - 20 Bit sind sehr teuer,
  - 0/1 Vektor wäre in dem Fall besser

### Komprimierung der Postings

- Dokumentlisten werden sortiert gespeichert *Brutus*: 33,47,154,159,202
- Deshalb: ausreichend die Lückengröße zu speichern *Brutus*: 33,14,107,5,43
- Hoffnung: Lücken brauchen weniger als 20 Bit

### Variable Kodierung der Lücken

- Ziel
  - Für Terme wie Calpurnia $\approx 20$ Bits pro Lücke ausgeben
  - Für Terme wie the"$\approx 1$ Bit pro Lücke ausgeben
  - Wenn die durchschnittliche Lückegröße $\log_2 G$ ist, dann soll $\approx \log_2 G$ Bits pro Lücke ausgeben werden
- Aufgabe
  - Kodiere jeden Integer (Lücke) im Durchschnitt mit so wenig Bits wie möglich

### Gamma Kodes

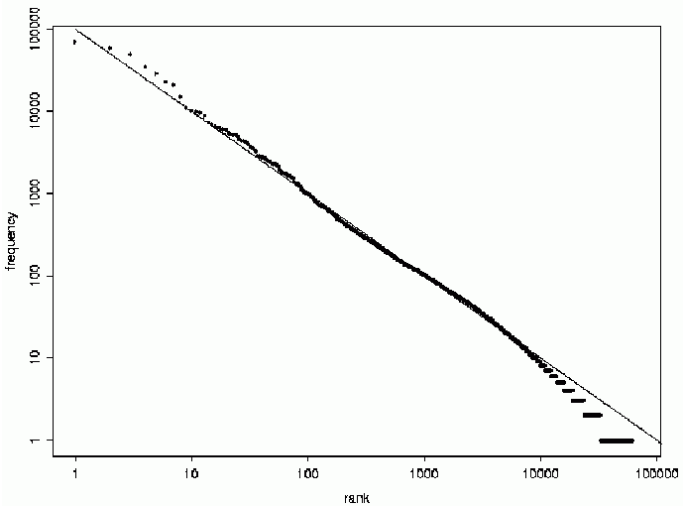
- repräsentiere Lücke $G$ als Paar $< L, O >$, (Länge — Offset)
- Länge $L$ wird unär kodiert und braucht $\lfloor \log_2 G \rfloor + 1$ Bits
- Der Offset $O = G - 2^{\lfloor \log_2 G \rfloor}$ wird binär dargestellt.
- Kode für $G$ braucht $2\lfloor \log_2 G \rfloor + 1$ Bits
- z.B. 9 wird repräsentiert als $< 0001, 001 >$
- Gamma Kode: Platz für Lücken, Doppelte Länge der Minimal-Repräsentation
- Verteilung der Lücken: Zipf-Verteilung angenommen

### Gamma Kode, Beispiele

$1 = 2^0 + 0 = 1$
$2 = 2^1 + 0 = 010$
$3 = 2^1 + 1 = 011$
$4 = 2^2 + 0 = 00100$
$5 = 2^2 + 1 = 00101$
$6 = 2^2 + 2 = 00110$
$7 = 2^2 + 3 = 00111$
$8 = 2^3 + 0 = 0001000$
$9 = 2^3 + 1 = 0001001$
$10 = 2^3 + 2 = 0001010$
$11 = 2^3 + 3 = 0001011$
$12 = 2^3 + 4 = 0001100$
$13 = 2^3 + 5 = 0001101$
$14 = 2^3 + 6 = 0001110$
$15 = 2^3 + 7 = 0001111$
$16 = 2^4 + 0 = 000010000$
$17 = 2^4 + 1 = 000010001$

### Zipf Verteilung

- Beispiel für eine Zipf-Verteilung




### Platz Analyse der Postings für Zipf-verteilte Terme (1/2)

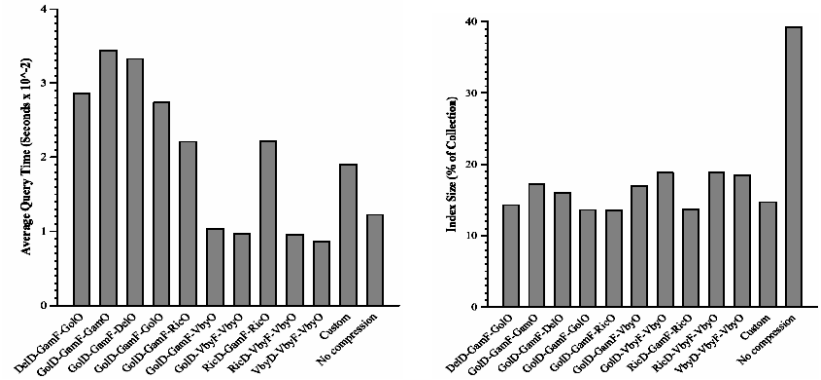
- Zipf's Gesetz
  - $k$-häufigste Term hat eine normalisierte Häufigkeit von $\approx 1/k$.
- Grobe Analyse des Platz für Postings
  - häufigster Term kommt in $n$ Dokumente vor
    - $n$ Lücken der Länge 1
  - Zweithäufigster Term kommt in $n/2$ Dokumenten vor
    - $n/2$ Lücken der Länge 2 ...
  - k-häufigste Term kommt in $n/k$ Dok. vor
    - $n/k$ Lücken der Länge $k$,
    - Gamma Kodes: $2 \log k + 1$ Bits für jede Lücke
    - Gesamtplatzbedarf: $\approx 2n/k \log k$ Bits für den k-häufigsten Term

### Platz Analyse (2/2)

- Summiere über $k$ von 1 bis $m = 500 \cdot 10^3$.
- Teile die Werte für $k$ in Gruppen
  - Gruppe $i$ besteht aus $2^{i-1} \leq k < 2^i$
  - Gruppe $i$ hat $2^{i-1}$ Summanden, jeder trägt maximal $ni/2^{i-1}$ bei ($n = 10^6$).
- Summiert man $i$ von 1 bis 19, bekommt man 340 MBits $\approx$ 45MB für die Postings
- Gamma Kodes sind nicht praktikabel, da Computer mit 16, 32 oder 64 Bits auf einmal rechnen $\Rightarrow$ langsam
- Word-aligned Kompression ist besser

## Experimente (Scholer, Zobel, SIGIR 2002)

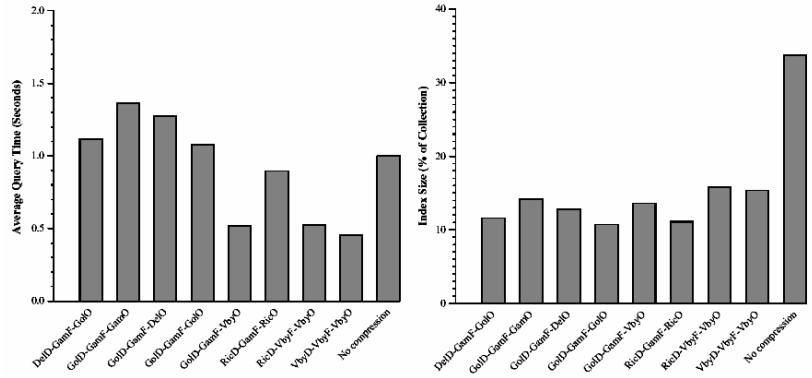
- 500 MB Daten, 10.000 Anfragen




- Anfragezeiten und Indexgrößen für verschiedene Kodierungen, 500 MB Daten, 10.000 Anfragen

## Experimente (Scholer, Zobel, SIGIR 2002)

- 20 GB Daten, 25.000 Anfragen




- Anfragezeiten und Indexgrößen für verschiedene Kodierungen, 20 GB Daten, 25.000 Anfragen

# 10 Satzorientierte DB-Schnittstelle

## 10.1 Logische Zugriffspfade

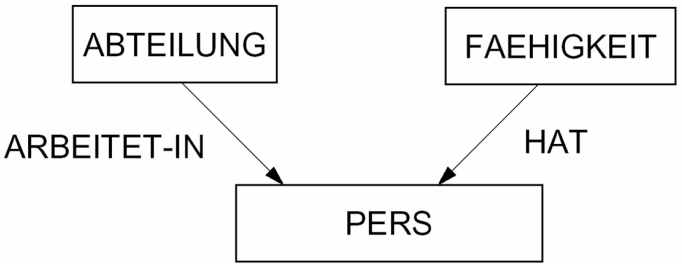
### Logische Zugriffspfade 1/2

- Charakterisierung der Abbildung
- Beispiel
  - `STORE <record>`
    `FETCH <record> USING <attr. 1> = 400 AND <attr. 2> >=7`
    `CONNECT <record> TO <set>`
- Abbildungsfunktionen
  - Physischer Satz ↔ Externer Satz
  - Externer Satz ↔ zugehörige Zugriffspfade
  - Suchausdruck → unterstützende Zugriffspfade
- Beispiel
  - `insert <record> at ...`
    `add <entry> to...`
    `retrieve <address-list> from...`
    `retrieve <record> with`

### Logische Zugriffspfade 2/2

- Eigenschaften der oberen Schnittstelle
  - Logische Sätze (dynamische Format-Umsetzung)
  - Logische Zugriffspfade (inhaltsadressierbarer Speicher, hierarchische Beziehungen zwischen Satztypen)
  - Zugriff in Einheiten von einem Satz pro Aufruf
  - Anwendungsprogrammierschnittstelle (API = application programming interface) mit navigierendem Zugriff
    - z. B. entsprechend dem Netzwerk-Datenmodell oder
    - objektorientierten Datenmodellen

### Das Netzwerk-Datenmodell - Ein Beispiel




1. Einfache Inhaltsadressierung `FETCH PERS USING PNR=12345`
2. ... kombiniert mit hierarchischem Zugriff

```
FETCH ABTEILUNG USING ABT-NAME ='VERTRIEB'
FETCH NEXT PERS WITHIN ARBEITET-IN
```

3. ... mit zwei hierarchischen Beziehungen

```
FETCH FAEHIGKEIT USING BERUF = 'PROGRAMMIERER'
FETCH NEXT PERS WITHIN HAT
FETCH OWNER WITHIN ARBEITET-IN
```

### Satzorientierte DB-Schnittstelle 1/2

- Bereitstellung der Objekte und Operationen
  - als externe Schnittstelle bei satzorientierten DBS
  - als interne Schnittstelle bei mengenorientierten DBS
- Typische Objekte
  - Segment (Area), Satztyp (Relation) und Satz (Tupel), Index (Search Key) und Set (Link)
- Operatoren auf Segmenten
  - Öffnen und Schließen von Segmenten (OPEN/CLOSE)
  - Erwerb und Freigabe von Segmenten (ACQUIRE / RELEASE)
  - Sichern und Zurücksetzen von Segmenten (SAVE / RESTORE)

### Satzorientierte DB-Schnittstelle 2/2

- Operatoren auf Sätzen und Zugriffspfaden
  - Direktes Auffinden von Sätzen über Attributwerte (FIND RECORD USING ...)
  - Navigierendes Auffinden von Sätzen über einen Zugriffspfad (FIND NEXT RECORD WITHIN ...)
  - Hinzufügen eines Satzes (INSERT)
  - Löschen eines Satzes (DELETE)
  - Aktual. von Attributwerten eines Satzes (UPDATE)
- Manipulation von benutzerkontrollierten Zugriffspfaden
  - Einbringen eines Satzes (CONNECT)
  - Aufheben dieser Verknüpfung (DISCONNECT)
- Weitere typische Operationen
  - BEGIN / COMMIT / ABORT_TRANSACTION
  - LOCK, UNLOCK

### Abbildung von externen Sätzen 1/2

- Beschreibung der externen Sätze durch Subschema-Konzept
- Aufgaben des Subschema-Konzeptes
  - Anpassung der Datentypen
  - Selektive Auswahl von Attributen
  - Abbildung eines externen Satzes auf interne Sätze eines oder mehrerer Satztypen
- Partitionierte Speicherung eines (großen) Satztyps (Relation)

- Zuordnung disjunkter Satzmengen zu separaten Speicherungseinheiten
  - Leistungsgründe: E/A-Parallelität
  - Verfügbarkeit: Erstellung von Kopien, Migration, ...
  - Partition ist Einheit der Reorganisation, des Backup, der Archivierung, des Ladens, von Zugriffsverfahren etc. in DB2
  - Spezifikation der Partitionierung über Werte (Schlüsselbereiche, Hashing) oder über Prozeduren (user exit)

### Abbildung von externen Sätzen 2/2

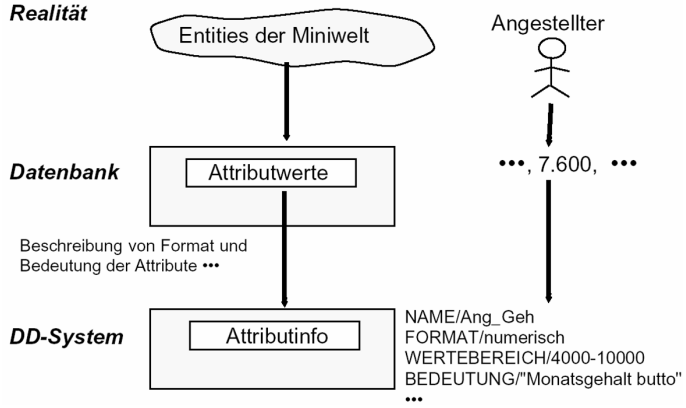
- Optionen der Speicherung bei internen Sätzen
  - Aufteilung und Zuordnung der Felder nach Zugriffshäufigkeiten
  - redundante Speicherng
  - Verdichtung von Feldern und Sätzen

  ⇒ Möglichkeiten der Abstimmung/Verbesserung des Leistungsverhaltens (Tuning)

## 10.2 Data Dictionary

### Integriertes Data Dictionary 1/4

- Aufgabe
  - Ein D/D-System dient zur Verwaltung von Metadaten, die Informationen über Inhalt, Nutzung, Integritätsbedingungen usw. eines Datenverwaltungs- bzw. Datenbanksystems geben.
- Charakterisierung der erforderlichen Abbild.



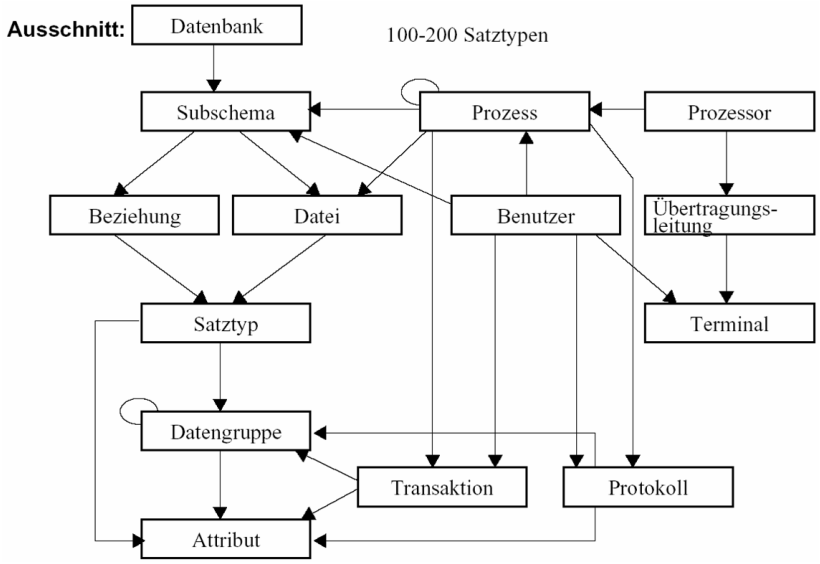

### Integriertes Data Dictionary 2/4

- SQL-Norm
  - einheitliche Sprachschnittstelle (SQL-Sprache)
  - gespeicherte Daten und ihre Eigenschaften (Metadaten) müssen nach einheitlichen und verbindlichen Richtlinien beschrieben werden

  ⇒ Portabilität, Integration heterogener DBS usw.
- Datenbankkataloge
  - normkonforme Implementierungen bieten einheitliche Sichten auf Metadaten
  - Sichten werden durch sog. Informationsschema definiert
  - Informationsschema ist für den Benutzer definierte Schnittstelle zum DB-Katalog

## Integriertes Data Dictionary 3/4

- Konzeptionelles Schema eines Data Dictionary




## Integriertes Data Dictionary 4/4

- DB-Katalog der SQL-Norm1: Kern eines D/D
  - Beschreibungen für Tabellen, Sichten, Domänen und Zusicherungen
  - Für autonome Tabellen (Basisrelationen) sind 3 Arten von Integritätsbedingungen (table constraints) spezifizierbar: Schlüssel (unique constraints), Fremdschlüssel (referential constraints) und allgem. Bedingungen (check constraints)
  - Spezifikation eigenständiger Bedingungen (Zusicherungen, assertions) unabhängig von einzelnen Tabellen
  - Definition von Nutzungsrechten (priviledges) für Domänen, Zeichensätze, Vergleichs- / Transformationsregeln für Tabellen und Spalten [1]

## Aufgaben & Funktionen des Data Dictionary

- Speicherung der Beschreibungs- oder Metainformation
  - Angaben über Definition, Struktur und Benutzungsvorschriften
  - Angaben über Speicherung, Codierung und Zugriffspfade
  - direkte Zugriffsmöglichkeit auf Schema-Information durch Anwendungsprogramme (AP) vorteilhaft
- Definition und Kontrolle durch DBA
  - Ableitung von Subschemata durch DDL-Übersetzer
- Erweiterung des D/D
  - Info über Herkunft, aktuelle Benutzung und Änderung der Daten
  - Namen und Charakteristika von AP (Datenverwendung)

[1] Jeder SQL-Katalog enthält ein INFORMATION_SCHEMA genanntes SQLSchema, für das drei Domänen, eine Basistabelle und 23 Sichten vorgeschrieben sind. Die Sichten kann man in vier Kategorien aufteilen: Struktur, Datentypen, Integritätsbedingungen und Rechte)

### Spezielle Funktionen zur Auswertung

- Dokumentation der Daten
- Analyse der vorhandenen Datenobjekte
- Unterstützung des Entwurfs von Anwendungen
- Information über Gültigkeit & Verfügbarkeit der Daten
- Analyse der Auswirkungen von Änderungen der Datendefinition
- Optimierung der Speicherungsstrukturen aufgrund von Benutzungshäufigkeiten und Werteverteilungen

## 10.3 Satzorientierte Verarbeitung und Scans

### Satzorientierte Verarbeitung 1/3

- Wie wird die satzorientierte Verarbeitung durch die Schicht der Speicherungsstrukturen unterstützt?
- Verarbeitungskonzepte
  - Kontextfreie Operationen
  - Satzweise Navigation über vorhandene Zugriffspfade
  - Umordnung einer Satzmenge, falls passende Reihenfolge nicht vorhanden
- Verarbeitungsprimitive
  1. Direkter Zugriff (Hashing, Bäume, ...)
  2. Navigierender Zugriff über Scan (auf welchen Objekten?)
  3. Sortierung einer Satzmenge und Scan darauf (Welche Satzmengen?)
  4. Satzweise Aktualisierung (Insert, Delete, Update)
- Einführung eines Navigationskonzeptes
  - Bereitstellen und Warten von transaktionsbezogenen Verarbeitungspositionen zur Navigation
  - Verschiedene Cursor-Konzepte

### Satzorientierte Verarbeitung 2/3

- Implizites Cursor-Konzept (Currency-Konzept bei CODASYL)
  - Änderung mehrerer Cursor durch Ausführung einer DB-Anweisung
  - Benutzerkontrolle durch RETAINING-Klausel
  - Hochgradige Komplexität, Fehleranfälligkeit
- Explizites Cursor-Konzept
  - Cursor werden durch AP definiert, verhalten sich wie normale Programmvariablen und werden unter expliziter Kontrolle des AP verändert
  - Definition von n Cursor auf einer Relation
  - Eindeutige Änderungssemantik

### Satzorientierte Verarbeitung 3/3

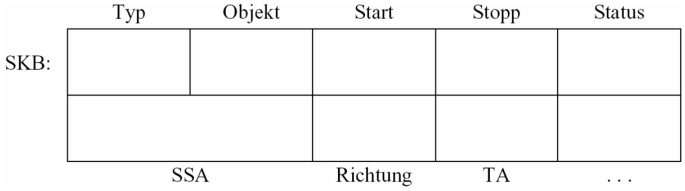
- Was ist ein Scan?
  - Ein Scan erlaubt das satzweise Durchmustern und Verarbeiten einer physischen Satzmenge
  - Er kann an einen Zugriffspfad (Index, Link, ...) gebunden werden
  - Er definiert eine Verarbeitungsposition, die als Kontextinformation für die Navigation dient
- Wie unterstützt das Scan-Konzept die mengenorientierte Verarbeitung (SQL)?

### Spektrum von Scan-Typen 1/2

- Scan-Typen
  - Satztyp-Scan (Relationen-Scan) zum Aufsuchen aller Sätze eines Satztyps (Relation)
  - Index-Scan zum Aufsuchen von Sätzen in einer wertabhängigen Reihenfolge
  - Link-Scan zum Aufsuchen von Sätzen in benutzerkontrollierter Einfügereihenfolge
  - k-d-Scan zum Aufsuchen von Sätzen über einen k-dimensionalen Index
- Implementierung von Scans
  - explizite Definition/Freigabe: OPEN/CLOSE SCAN
  - Navigation: NEXT TUPLE
  - Scans werden auf Zugriffspfaden definiert
  - Optionen: Start-, Stopp- und Suchbedingung, Suchrichtung: NEXT/PRIOR, FIRST/LAST, n-th

### Spektrum von Scan-Typen 2/2

- Scan-Kontrollblock (SKB): Angaben über Typ, Status, Position etc.




### Relationen-Scan 1/2

- Anfragebeispiel:

```
SELECT * FROM PERS
WHERE ANR BETWEEN K28 AND K67
AND BERUF = 'Programmierer'
```

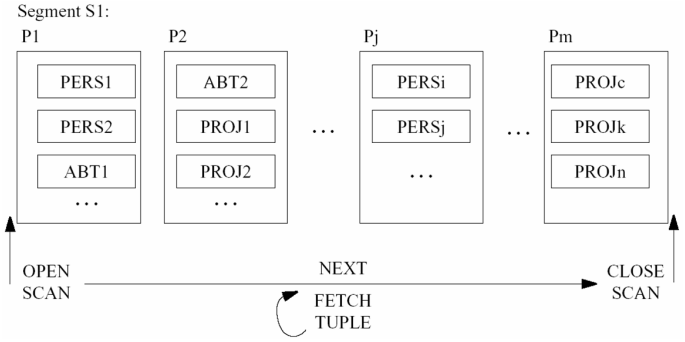
- Scan-Optionen
  - Startbedingung (SB): BOS (Beginn von S1)
  - Stoppbedingung (STB): EOS (Ende von S1)
  - Suchrichtung: NEXT
  - Suchbedingung (SSA): ANR $\geq$ K28 AND ANR $\leq$ K67 AND BERUF = Programmierer

## Relationen-Scan 2/2

- Beispiel für Relationen-Scan

```
OPEN SCAN (PERS, BOS, EOS)                                    /* SCB1 */
WHILE (NOT FINISHED)
DO
    FETCH TUPLE (SCB1, NEXT,
                 ANR ≥ 'K28' AND ANR ≤ 'K67' AND BERUF = 'Programmierer')
    ...
END
CLOSE SCAN (SCB1)
```

- Ablauf beim Relationen-Scan




## Index-Scan 1/2

- Anfragebeispiel:

```
SELECT * FROM PERS
WHERE ANR BETWEEN K28 AND K67
AND BERUF = 'Programmierer'
```

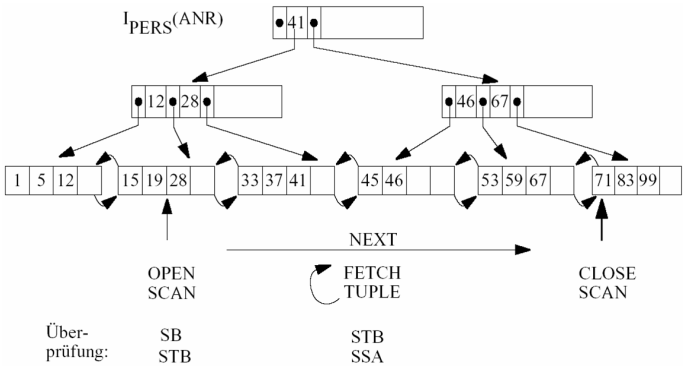
  Scan-Optionen
  - Startbedingung: ANR $\geq$ K28
  - Stoppbedingung: ANR $>$ K67
  - Suchrichtung: NEXT
  - Suchbedingung: BERUF = Programmierer

- Beispiel für Index-Scan

```
OPEN SCAN (I_PERS(ANR), ANR ≥ 'K28', ANR > 'K67')              /* SCB1 */
WHILE (NOT FINISHED)
DO
    FETCH TUPLE (SCB1, NEXT, BERUF = 'Programmierer')
    ...
END
CLOSE SCAN (SCB1)
```

## Index-Scan 2/2

- Ablauf beim Index-Scan




- Die Verweise (TIDs) auf die PERS-Tupel sind in IPERS(ANR) weggelassen.
- Die Suchbedingung (SSA = simple search argument) darf nur Wertvergleiche Attribut $\Theta$ Wert (mit $\Theta \in \{<, =, >, \leq, \neq, \geq\}$) enthalten und wird auf jedem Tupel überprüft, das über den Index-Scan erreicht wird.
- Der Operator FETCH TUPLE liefert also nur Tupel zurück, die die WHERE-Bedingung erfüllen.

## Link-Scan 1/4

- Anfragebeispiel:

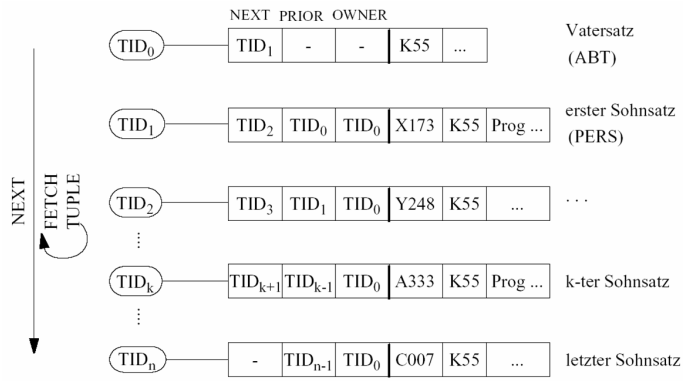
```
SELECT * FROM PERS
WHERE ANR BETWEEN K28 AND K67
AND BERUF = 'Programmierer'
```

- Auffinden des Vaters
  - Startposition bereits gefunden (ANR gegeben)
  - Fortschalten in ABT erforderlich (ANR BETWEEN K28 AND K67)
- Einzelner Link-Scan

```
OPEN SCAN (L_ABT-PERS(ANR), NONE, EOL)                    /* SCB1 */
WHILE (NOT FINISHED)
DO
    FETCH TUPLE (SCB1, NEXT, BERUF = 'Programmierer')
    ...
END
CLOSE SCAN (SCB1)
```

## Link-Scan 2/4

- Ablauf beim Link-Scan



## Link-Scan 3/4

- Auffinden des Vaters
  - Nutzung einer Indexstruktur
  - Schachtelung von Index-Scan (ANR BETWEEN K28 AND K67) und Link-Scan
- Scan-Optionen

| | Index-Scan | Link-Scan |
|---|---|---|
| Startbedingung: | ANR $\geq$ K28 | - |
| Stoppbedingung: | ANR $>$ K67 | EOL |
| Suchrichtung: | NEXT | NEXT |
| Suchbedingung: | - | BERUF = Programmierer |

## Link-Scan 4/4

- Index- und Link-Scan

```
OPEN SCAN (I_ABT(ANR), ANR ≥ 'K28', ANR > 'K67')                    /* SCB1 */
...
WHILE (NOT FINISHED)
DO
    FETCH TUPLE (SCB1, NEXT, NONE)
    ...
    OPEN SCAN (L_ABT-PERS(ANR), NONE, EOL)                          /* SCB2 */
    ...
    WHILE (NOT FINISHED)
    DO
        FETCH TUPLE (SCB2, NEXT, BERUF = 'Programmierer')
        ...
    END
    CLOSE SCAN (SCB2)
END
CLOSE SCAN (SCB1)
```

## k-d-Scan 1/2

- Anfragebeispiel
  - ```
    SELECT * FROM PERS
    WHERE ANR BETWEEN K28 AND K67
    AND ALTER BETWEEN 20 AND 30
    AND BERUF = 'Programmierer'
    ```
- Scan-Optionen

| | Dimension 1 | Dimension 2 |
|---|---|---|
| Startbedingung: | ANR $\geq$ K28 | ALTER $\geq$ 20 |
| Stoppbedingung: | ANR $>$ K67 | ALTER $>$ 30 |
| Suchrichtung: | NEXT | NEXT |
| Suchbedingung: | BERUF = 'Programmierer' (wird auf den PERS-Sätzen ausgewertet) | |

## k-d-Scan 2/2

- 2-d-Scan
    ```

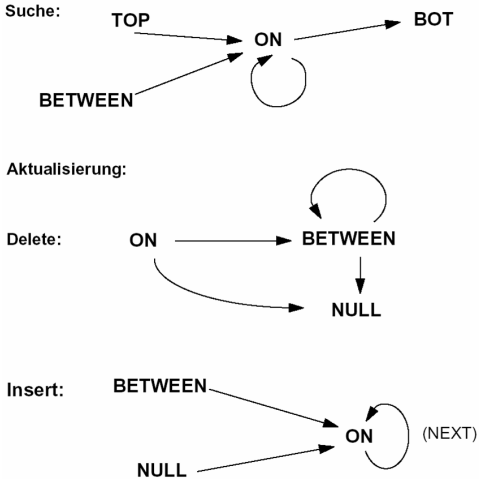
```
OPEN SCAN (I_PERS(ANR, ALTER), ANR ≥ 'K28' AND ALTER ≥ 20,
        ANR > 'K67' AND ALTER > 30)                          /* SCB1 */
...
WHILE (NOT (ALTER > 30))
DO
    /* Zwischenspeichern der SCB1-Position in SCANPOS                  */
    WHILE (NOT (ANR > 'K67'))
    DO
        FETCH TUPLE (SCB1, NEXT(ANR), BERUF = 'Programmierer')
        ...
    END
    /* Zurücksetzen der SCB1-Position auf SCANPOS                      */
    ...
    MOVE SCB1 TO NEXT(ALTER)
END
CLOSE SCAN (SCB1)
```

## Anwendung von Scans

- Scan-Zustände in Abhängigkeit von der Position in der Satzmenge
  - vor dem ersten Satz (TOP)
  - auf einem Satz (ON)
  - in einer Lücke zwischen zwei Sätzen (BETWEEN)
  - hinter dem letzten Satz (BOT)
  - in einer leeren Menge (NULL)

## Anwendung von Scans

- Zustandsübergänge beim Scan



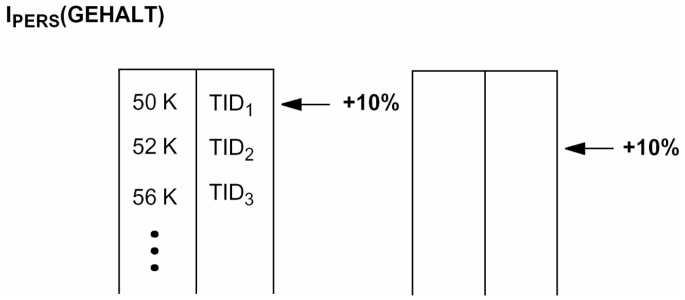

⇒ Scan-Semantik festlegen!

## Anwendung von Scans 1/2

- Problem
  Mengenorientierte Spezifikation ⇔ satzorientierte Auswertung
  - Die navigierende Auswertung einer deklarativen SQL-Anweisung über einen Scan kann Verarbeitungsprobleme verursachen, insbesondere wenn das zu aktualisierende Objekt (Relation, Zugriffspfad) vom Scan benutzt wird (Verbot?)

### Anwendung von Scans 2/2

- Beispiel, Anweisung:
  `UPDATE PERS SET GEHALT = 1.1 * GEHALT`
  Ausführung: Gehaltsverbesserung um 10% werde mit Hilfe eines Scans über Index GEHALT durchgeführt




  ⇒ Halloween-Problem

## 10.4 Sortieroperator und Anwendungen

### Einsatz eines Sortieroperators 1/2

- Explizite Umordnung der Sätze gemäß vorgegebenem Suchschlüssel (ORDER-Klausel)
- Beispiel für Umordnung mit einer Restriktion

```
SELECT * FROM PERS
WHERE ANR > 'K50'
ORDER BY GEHALT DESC
```

- Partitionierung von Satzmengen

```
SELECT ANR, AVG (GEHALT)
FROM PERS
GROUP BY ANR
```

- Duplikateliminierung in einer Satzmenge

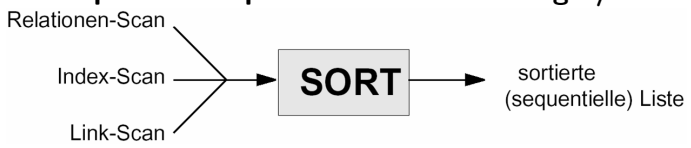
```
SELECT DISTINCT BERUF
FROM PERS
WHERE ANR > 'K50' AND ANR < 'K56'
```

### Einsatz eines Sortieroperators 2/2

- Unterstützung von Mengen- und Verbundoperationen
- Umordnen von Zeigern zur Optimierung der Auswertung
- oder Zugriffsreihenfolge
- Dynamische Erzeugung von Indexstrukturen (bottom-up-Aufbau von B*-Bäumen)
- Erzeugen einer Clusterbildung beim Laden und während der Reorganisation

⇒ Reduktion der Komplexität von $O(N^2)$ nach $O(N \log N)$ bei Mengen- und Verbundoperationen

### SORT-Operator - Optionen und Anwendung 1/2




- Scans können mit Suchbedingungen eingeschränkt sein
  (SSA = Simple Search Arguments)
- SORT-Optionen zur Duplikateliminierung:
  - N = keine Eliminierung
  - K = Duplikateliminierung bezüglich Sortierkriterium
  - S = STOPP, sobald Duplikat entdeckt wird
- SORT dient als Basisoperator für Operationen auf höherer Ebene

### SORT-Operator - Optionen und Anwendung 2/2

- Bsp.: Einsatz von Scan- und Sortier-Operator

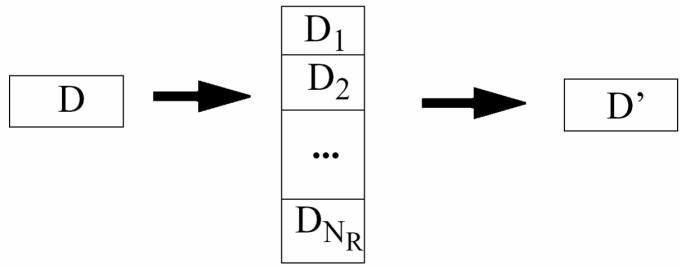
```
OPEN SCAN (R1, SB1, STB1)                    /* SCB1 */
SORT   R1  INTO  S1  USING  SCAN (SCB1)
CLOSE  SCAN (SCB1)
OPEN SCAN (R2, SB2, STB2)                    /* SCB2 */
SORT   R2  INTO  S2  USING  SCAN (SCB2)
CLOSE  SCAN (SCB2)
OPEN SCAN (S1, BOS, EOS)                     /* SCB3 */
OPEN SCAN (S2, BOS, EOS)                     /* SCB4 */
WHILE (NOT FINSHED)
DO
   FETCH TUPLE (SCB3, NEXT, NONE)
   FETCH TUPLE (SCB4, NEXT, NONE)
   ...
END
```

## 10.5 Externes Sortieren

### Externes Sortieren 1/7

- Wie wird sortiert?
  - zu sortierende Datenmenge (n Sätze) ist i.a. viel größer als der zum Sortieren verfügbare Hauptspeicher (HSP) (q Sätze), (DB-Pufferbereich oder spezieller Arbeitsspeicher)
  - Anwendung eines internen Sortierverfahrens erzeugt einen sogenannten Run
  - Welche Sortierverfahren sind geeignet?
- Mehrmalige Durchführung der Sortierung

– Lesen der Eingabedaten aus Datei $D$ und Schreiben der Runs nach $D_i$



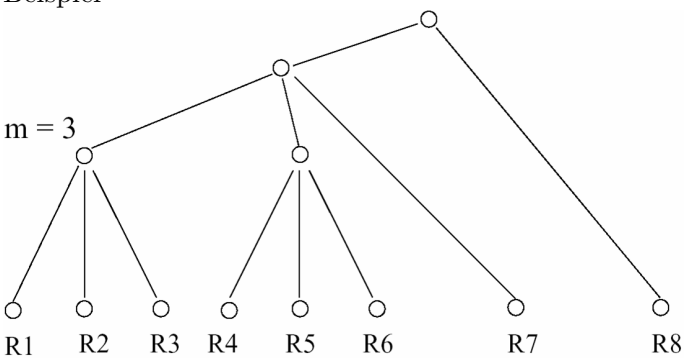

– Mischen aller $N_R$ anfänglichen Runs erforderlich

### Externes Sortieren 2/7

- m-Wege-Mischen bei Magnetplatten
  - Es stehen $m_{max} + 1$ Plätze (Seiten) im HSP zur Verfügung. $m_{max}$ bestimmt die maximale Mischordnung.
  - Die $N_R$ anfänglichen Runs sind i.a. über mehrere Platten verteilt
  - Jeder Run wird nach Möglichkeit sequentiell geschrieben. Er kann jederzeit wahlfrei gelesen werden.
  - Typischerweise sind die anfänglichen Runs gleich lang.

### Externes Sortieren 3/7

- Optimale Mischbäume
  - Optimierung der Zugriffsbewegungen auf den Magnetplatten ist sehr schwierig: wird gewöhnlich nicht berücksichtigt
  - Ziel: Minimierung der E/A und Vergleiche
  - einfach bei idealen Anzahlen $N_R$, z.B. $N_R = m_{max}^p$ Es ergibt sich ein vollständiger Mischbaum der Höhe $p$ vom Grad $m_{max}$.
- Wann sind unvollständige Mischbäume optimal?
  - Beispiel

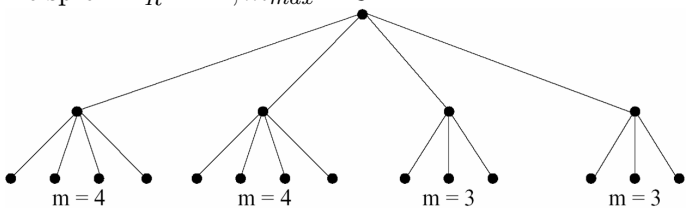



### Externes Sortieren 4/7

- Allgemein gilt: $N_R \leq (m_{max})^{p_{min}}$
  - Annahmen: nur 2 Dateien für Ein-/Ausgabe, ungewichtete Runs
  - Heuristik 1: Harmonisiere Mischbaum
    1. Bestimme $p_{min}$

2. Finde kleinstes $m$, so daß gilt $N_R \leq m^{p_{min}}$
3. Wende nur Mischordnungen von $m$ und $m-1$ an.

- Beispiel: $N_R = 14, m_{max} = 8$




## Externes Sortieren 5/7

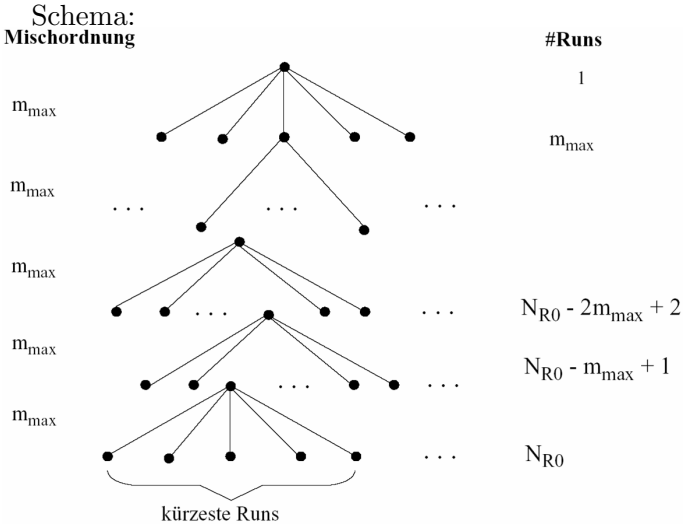
- Annahmen
  - keine Beschränkung für Ein-/Ausgabe
  - gewichtete Runs
- Heuristik 2: Minimiere Anzahl der Ein-/Ausgaben und Vergleiche
  1. $N_R \leq (m_{max} - 1) \cdot p + 1$; Bestimme minimales $p$
  2. Erzeuge zusätzliche leere Runs, so daß gilt:

$$N_{R0} = (m_{max} - 1) \cdot p + 1$$

  3. Wähle bei jedem Mischdurchlauf jeweils die $m_{max}$ kürzesten Runs und bilde daraus einen neuen Run, bis ein Run übrig bleibt.

## Externes Sortieren 6/7

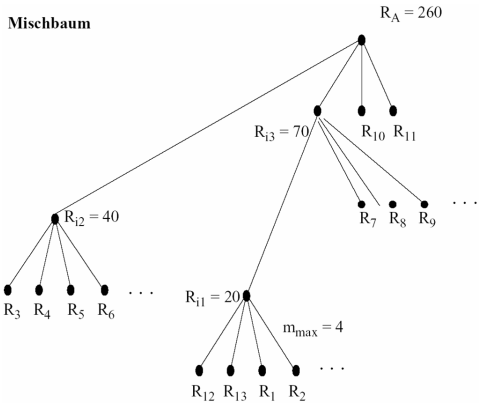
Schema:




## Externes Sortieren 7/7

- Annahmen
  - keine Beschränkung für Ein-/Ausgabe
  - gewichtete Runs
- Beispiel für Heuristik 2:
  - $N_R = 11, m_{max} = 4 \Rightarrow p = 4, N_{R0} = 13$

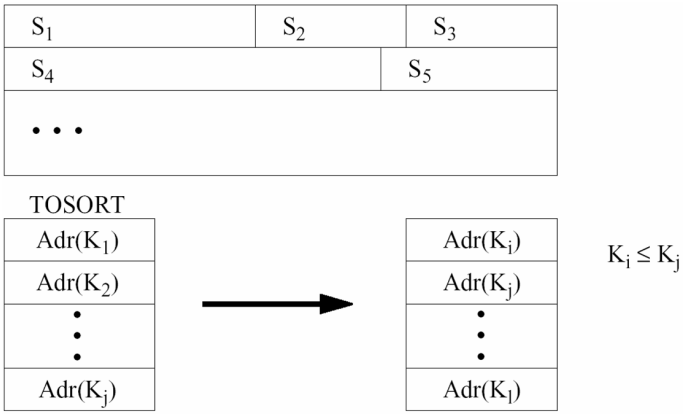
- Längen von: $R_1, \ldots, R_7 = 10$; $R_8, R_9 = 20$; $R_{10} = 50, R_{11} = 100$; $R_{12}, R_{13} = 0$




## 10.6 Sortieren von variablen Sätzen

### Sortieren von variablen Sätzen

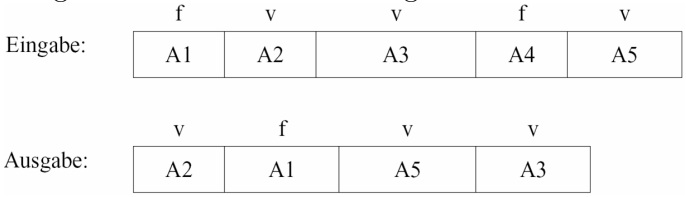
- Datensätze in Dateien sind oft variabel lang
  - bestehend aus festen (f) und variabel langen (v) Feldern
  - entsprechend sind Sortierschlüssel variabel lang
- Wie organisiert man Sortierbereich im HSP ?




- Wann werden die Datensätze in die Sortierreihenfolge gebracht ?

### Sortieren von variablen Sätzen

- Weitere Probleme
  - Ausgabeformat weicht vom Eingabeformat ab




  - Sortierschlüssel besteht aus mehreren Feldern, nach denen aufsteigend (+) und/oder absteigend (-) zu sortieren ist
    - z. B. Sortierschlüssel: A5 +, A3 +, A1 -

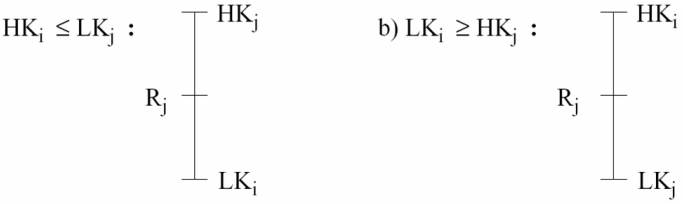
### Sortieren von variablen Sätzen 1/2

- Sortierfolge bei variabel langen Schlüsseln
  - aufsteigend: 0, A, AA, AAA ; absteigend: AAA, AA, A, 0
- Wie vergleicht man zwei zusammengesetzte Schlüssel variabler Länge?
  - K sei definiert durch A5+|A3+:
  - Was ist das Ergebnis des Vergleichs von K1 und K2 ?
  - K1 = 12345|67, K2 = 123|6789

### Sortieren von variablen Sätzen 2/2

- Wie kann man eine Vorsortierung ausnutzen ?
- Run $R_j$ besitzt $HK_j$ (Highkey) und $LK_j$ (Lowkey)
  1. Vergleich von $R_j$ mit dem neu erzeugten Run $R_i$


  2. Wenn möglich, Verlängerung von $R_j$ durch $R_i$




  3. Sonst: separate Speicherung von $R_i$

### Zusammenfassung 1/2

- Trennung von internen und externen Sätzen und flexible Abbildungskonzepte erforderlich
- Transaktionsbezogene Kontroll- und Überwachungsaufgaben
  - Diese verlangen einen schichtenübergreifenden Informationsfluß
  - Lastkontrolle und -balancierung ist komplexes Forschungsthema
- Data Dictionary-/Directory-System verwaltet Metadaten zur Beschreibung aller Aspekte der Benutzerdaten
  - Inhalt, Nutzung, Integritätsbedingungen, Zugriffskontrollbedingungen
- Weitergehende Einsatzmöglichkeiten von D/D
  - Unterstützung des Programm- und Anwendungssystementwurfs
  - Revisionsunterstützung (Sammlung und Auswertung von zeitbezogenen Informationen)
  - integriertes D/D erforderlich

### Zusammenfassung 2/2

- Cursorkonzept
  - Scan-Technik zur satzweisen Navigation auf Zugriffspfaden
  - flexibler Einsatz durch Start-, Stopp- und Suchbedingung sowie Suchrichtung
- Sortierkomponente
  - wichtig zur Implementierung relationaler Operationen
  - große Relationen (Dateien) erfordern Sortier-/Mischverfahren

# 11 Relationale Operatoren

### Relationenoperationen – Implementierung

- Operationen der Relationenalgebra
  - unäre Operationen: $\pi, \sigma$
  - binäre Operationen: $join, \times, +, \cap, \cup, -$

  $\Rightarrow$ SQL-Anfragen enthalten logische Ausdrücke, die auf die Operationen der Relationenalgebra zurückgeführt werden können. Sie werden in Zugriffspläne umgesetzt. Planoperatoren implementieren diese logischen Operationen.

## 11.1 Planoperatoren auf einer Relation

### Planoperatoren auf einer Relation 1/2

- Planoperatoren zur Selektion, Allgemeine Auswertungsmöglichkeiten:
  - direkter Zugriff über ein gegebenes TID, über ein Hash-Verfahren oder eine ein- bzw. mehrdimensionale Indexstruktur
  - sequentielle Suche in einer Relation
  - Suche über eine Indexstruktur (Indextabelle, Bitliste)
  - Auswahl mit Hilfe mehrerer Verweislisten, wobei mehr als eine Indexstruktur ausgenutzt werden kann
  - Suche über eine mehrdimens. Indexstruktur
- Projektion wird typischerweise in Kombination mit Sortierung, Selektion oder Verbund durchgeführt

### Planoperatoren auf einer Relation 2/2

- Planoperatoren zur Modifikation
  - Änderungen sind in SQL mengenorientiert, aber auf eine Relation beschränkt
  - INSERT, DELETE und UPDATE werden direkt auf die entsprechenden Operationen der Speicherungsstrukturen abgebildet
  - automatische Abwicklung von Wartungsoperationen zur Aktualisierung von Zugriffspfaden, zur Gewährleistung von Cluster-Bildung und Reorganisation usw.
  - Durchführung von Logging- und Recovery-Maßnahmen usw.

### Planoperatoren für die Selektion 1/2

- Nutzung des Scan-Operators
  - Definition von Start- und Stop-Bedingung
  - Definition von einfachen Suchargumenten
- Planoperatoren
  1. Relationen-Scan: immer möglich
     - SCAN-Operator impliziert Selektionsoperation
  2. Index-Scan
     - Auswahl des kostengünstigsten Index

- Spezifikation des Suchbereichs (Start, Stop)

3. k-d-Scan
   - Auswertung mehrdimensionaler Suchkriterien
   - Nutzung verschiedener Auswertungsrichtungen durch Navigation
4. TID-Algorithmus
   - Auswertung aller brauchbaren Indexstrukturen
   - Auffinden von variabel langen TID-Listen
   - Boolesche Verknüpfung der einzelnen Listen
   - Zugriff auf Tupel entsprechend Trefferliste

#### Planoperatoren für die Selektion 2/2

- Weitere Planoperatoren in Kombination mit der Selektion
  - Sortierung
  - Gruppenbildung (siehe Sortieroperator)
  - spezielle Operatoren z. B. in Data-Warehouse-Anwendungen zur Gruppen- und Aggregatbildung (CUBE-Operator)

### 11.2 Planoperatoren über mehrere Relationen

#### Operatoren über mehrere Relationen 1/4

- SQL erlaubt komplexe Anfragen über $k$ Relationen
  - Ein-Variablen-Ausdrücke: beschreiben Bedingungen für die Auswahl von Elementen aus einer Relation.
  - Zwei-Variablen-Ausdrücke: beschreiben Bedingungen für die Kombination von Elementen aus zwei Relationen.
  - $k$-Variablen-Ausdrücke werden typischerweise in Ein- und Zwei-Variablen- Ausdrücke zerlegt und durch entsprechende Planoperatoren ausgewertet
- Planoperatoren über mehrere Relationen Allgemeine Auswertungsmöglichkeiten:
  - Schleifeniteration (nested iteration) für jedes Element der äußeren Relation $R_a$ Durchlauf der inneren Relation $R_i$: $O(N_a \cdot N_i + N_a)$.
  - wichtigste Anwendung: nested loops join

#### Operatoren über mehrere Relationen 2/4

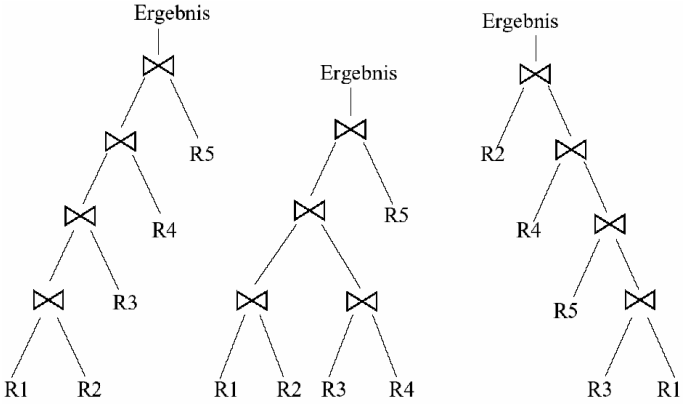
- Planoperatoren über mehrere Relationen Allgemeine Auswertungsmöglichkeiten (Fortsetzung):
  - Mischmethode (merge method) sequentieller, schritthaltender Durchlauf beider Relationen $R_1, R_2$: $O(N_1 + N_2)$.
    - ggf. zusätzliche Sortierkosten
    - wichtigste Anwendung: merging join
  - Hash-Methode (hashing) Partitionierung der inneren Relation $R_i$. Laden der $p$ Partitionen in eine Hash-Tabelle $HT$ im HSP. Probing der äußeren Relation $R_a$ oder ihrer entsprechenden Partitionen mit $HT$: $\Rightarrow O(p \cdot N_a + N_i)$.

### Operatoren über mehrere Relationen 3/4

- $n$-Wege-Verbunde
  - Zerlegung in $n-1$ Zwei-Wege-Verbunde
  - Anzahl der Verbundreihenfolgen ist abhängig von den gewählten Verbundattribute
  - $n!$ verschiedene Reihenfolgen möglich
  - Einsatz von Pipelining-Techniken
  - Optimale Auswertungsreihenfolge abhängig von:
    1. Planoperatoren
    2. passende Sortierordnungen für Verbundattribute und
    3. Größe der Operanden usw.

### Operatoren über mehrere Relationen 4/4

- Verschiedene Verbundreihenfolgen mit Zwei-Wege-Verbunden ($n = 5$)




- Analoge Vorgehensweise bei Mengenoperationen

### Planoperatoren für den Verbund 1/2

- Verbund
  - satztypübergreifende Operation: gewöhnlich sehr teuer
  - häufige Nutzung: wichtiger Optimierungskandidat
  - typische Anwendung: Gleichverbund
  - allgemeiner Θ-Verbund selten
- Implementierung der Verbundoperation kann gleichzeitig Selektionen auf den beteiligten Relationen R und S ausführen

```
SELECT *
FROM R, S
WHERE R.VA THETA S.VA
      AND PR
      AND PS
```

  - VA: Verbundattribute
  - $P_R$ und $P_S$: Prädikate definiert auf Selektionsattributen (SA) von R und S

### Planoperatoren für den Verbund 2/2

- Mögliche Zugriffspfade
  - Scans über $R$ und $S$ (geht immer)
  - Scans über $I_R(VA)$, $I_S(VA)$ falls vorhanden ⇒ liefern Sortierreihenfolge nach VA
  - Scans über $I_R(SA)$, $I_S(SA)$ (wenn vorhanden) ⇒ ggf. schnelle Selektion für $P_R$ und $P_S$
  - Scans über andere Indexstrukturen wenn vorhanden ⇒ ggf. schnelleres Auffinden aller Sätze

## 11.3 Nested-Loop-Verbund

### Nested-Loop-Verbund 1/2

- Annahmen:
  - Sätze in $R$ und $S$ sind nicht nach den Verbundattributen geordnet
  - es sind keine Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden
- Algorithmus für $\Theta$-Verbund:
  Scan über $S$,
  Für jeden Satz s, falls $P_S$:
  Scan über $R$,
  Für jeden Satz $r$, falls $P_R$ AND $(r.VA\ \Theta\ s.VA)$:
  führe Verbund aus,
  übernehme kombi. Satz $(r, s)$ in Ergebnismenge.
- Komplexität: $O(N^2)$ (Kardinalität $N$ für $R$ und $S$)

### Nested-Loop-Verbund 2/2

- Nested-Loop-Verbund mit Indexzugriff
  Scan über $S$,
  Für jeden Satz $s$, falls $P_S$:
  Ermittle mittels Zugriff auf $I_R(VA)$ alle TIDs
  für Sätze mit $r.VA = s.VA$,
  für jedes TID:
  hole Satz $r$, falls $P_R$:
  übernehme kombinierten Satz $(r, s)$ in Ergebnis.
- Nested-Block-Verbund
  Scan über $S$,
  für jede Seite (bzw. Menge aufeinanderfolgender Seiten) von $S$:
  Scan über $R$,
  Für jede Seite (bzw. Menge aufeinanderfolgender
  Seiten) von $R$:
  Für jeden Satz $s$ der $S$-Seite, falls $P_S$:
  Für jeden Satz $r$ der $R$-Seite,
  Falls $P_R$ AND $(r.VA\ \Theta\ s.VA)$:
  Übernehme komb. Satz $(r, s)$ in Ergebnis.

## 11.4 Sort-Merge-Verbund

### Sort-Merge-Verbund 1/2

- Algorithmus besteht aus 2 Phasen
- Phase 1
  - Sortierung von $R$ und $S$ nach $R(VA)$ und $S(VA)$ (falls nicht bereits vorhanden)
  - dabei frühzeitige Eliminierung nicht benötigter Sätze ($\Rightarrow P_R, P_S$)
- Phase 2
  - Schritthaltende Scans über sortierte $R$- und $S$-Sätze mit Durchführung des Verbundes bei $r.VA = s.VA$
- Komplexität: $O(N \log N)$

### Sort-Merge-Verbund 2/2

- Spezialfall
  - Falls $I_R(VA)$ und $I_S(VA)$ oder
  - verallgemeinerte Zugriffspfadstruktur über $R(VA)$ und $S(VA)$ (Join-Index) vorhanden

$\Rightarrow$ Ausnutzung von Indexstrukturen auf Verbundattributen:
Schritthaltende Scans über $I_R(VA)$ und $I_S(VA)$:
Für jeweils zwei Schlüssel aus $I_R(VA)$ und $I_S(VA)$,
Falls $r.VA = s.VA$:
Hole mit den zugehörigen TIDs die Tupel,
Falls $P_R$ und $P_S$:
Übernehme kombinierten Satz $(r, s)$ in die Ergebnismenge.

## 11.5 Hash-Verbund

### Hash-Verbund 1/3

- Einfachster Fall (classic hashing):

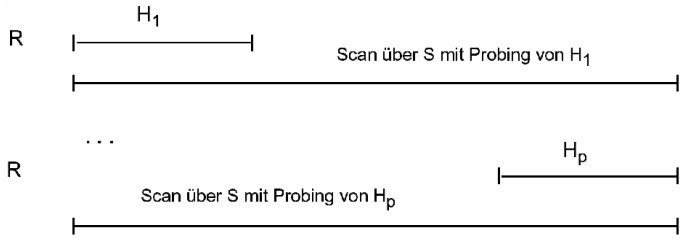
**Schritt 1** Abschnittsweises Lesen der (kleineren) Relation $R$ und Aufbau einer Hash-Tabelle mit $h_A(r(VA))$ nach Werten von $R(VA)$ entsprechend den Abschnitten $R_i,\ (1 \leq i \leq p)$, so daß jeder der $p$ Abschnitte in den verfügbaren Hauptspeicher paßt und jeder Satz $P_R$ erfüllt

**Schritt 2** Überprüfung (Probing) für jeden Satz von $S$ mit $P_S$; im Erfolgsfall Durchführung des Verbundes

**Schritt 3** Wiederhole Schritt 1 und 2 solange, bis $R$ erschöpft ist.

### Hash-Verbund 2/3

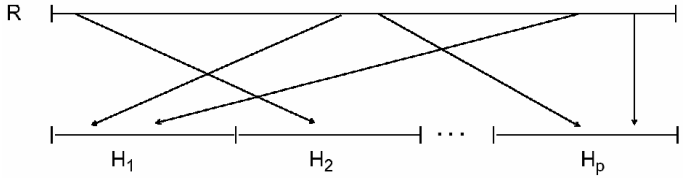
- Aufbau der Hash-Tabelle und Probing
  - Es erfolgt ein Scan über $R$
  - Dabei wird die Hash-Tabelle $H_i,\ (1 \leq i \leq p)$ der Reihe nach im HSP aufgebaut




- Komplexität: $O(p \cdot N)$
  - Spezialfall: $R$ paßt in den Hauptspeicher: eine Partition ($p = 1$) $\Rightarrow$ ein Scan über $S$ genügt

## Hash-Verbund 3/3

- Partitionierung von $R$ mit Hash-Funktion $h_P$




  - Partitionierung von $R$ in Teilmengen $R_1, R_2, \ldots, R_p$: Ein Satz $r$ von $R$ ist in $R_i$, wenn $h(r)$ in $H_i$ ist. $\Rightarrow$ Warum ist diese Partitionierung eine kritische Operation? Welche Hilfsoperationen können erforderlich sein? Ist für die Partitionierung der Einsatz einer Hash-Funktion notwendig?
  - Relation $S$ wird mit derselben Funktion $hP$ unter Auswertung von $P_S$ partitioniert
- Varianten des Hash-Verbundes: Sie unterscheiden sich vor allem durch die Art der Partitionsbildung

## Simple-Hash-Join 1/3

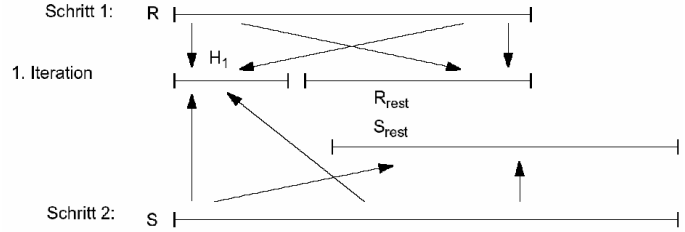
**Schritt 1** Führe Scan auf kleinerer Relation $R$ aus, überprüfe $P_R$ und wende auf jedes qualifizierte Tupel $r$ die Hash-Funktion $h_P$ an. Fällt $h_P(r(VA))$ in den gewählten Bereich, trage es in $H_i$ ein. Anderenfalls schreibe $r$ in einen Puffer für die Ausgabe in eine Datei für übergangene $r$-Tupel.

**Schritt 2** Führe Scan auf $S$ aus, überprüfe $P_S$ und wende auf jedes qualifizierte Tupel $s$ die Hash-Funktion $h_P$ an. Fällt $h_P(s(VA))$ in den gewählten Bereich, suche in $H_i$ einen Verbundpartner (Probing). Falls erfolgreich, bilde ein Verbundtupel und ordne es dem Ergebnis zu. Anderenfalls schreibe $s$ in einen Puffer für die Ausgabe in eine Datei für übergangene $s$-Tupel.

**Schritt 3** Wiederhole Schritt 1 und 2 mit den bisher übergangenen Tupeln solange, bis $R$ erschöpft ist. Dabei ist die Überprüfung von $P_R$ und $P_S$ nicht mehr erforderlich.

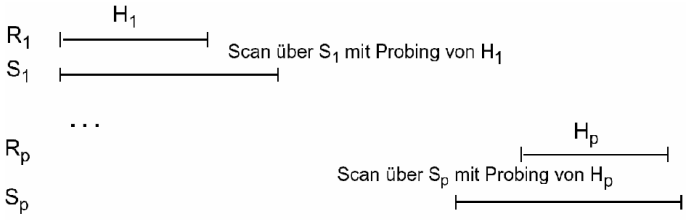
## Simple-Hash-Join 2/3

- Partitionierungstechnik beim Simple-Hash-Join: gezeigt an Aufbau und Probing von $H_1$



### Simple-Hash-Join 3/3

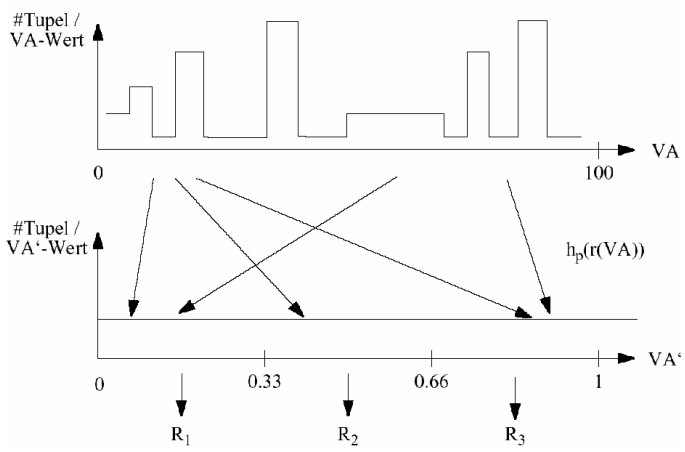
- Verbesserung der Partitionsbildung (Bei Grace-Join und Hybrid-Hash-Join)
  - Partitionsbildung findet vor dem Verbund statt
  - Partitionen $R_i, S_i$ sind in Dateien zwischengespeichert
  - Aufbau von $H_i$ im HSP mit $R_i$ und Probing mit $S_i$




- Hybrid-Hash-Join optimiert das Verfahren dadurch, daß während der Partitionsbildung Aufbau und Probing von $H_1$ erfolgt

### Hash-Verbund – Beispiel 1/2

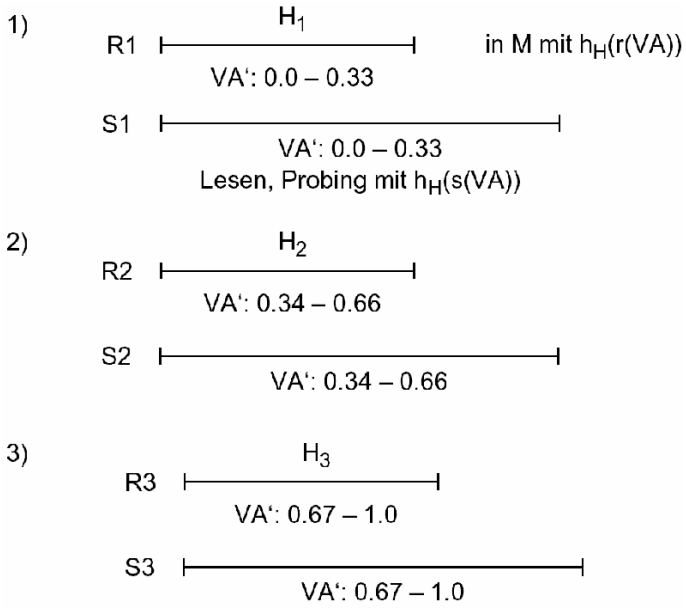
1. Partitionieren
   a) Partitionieren von $R$ mit $h_p(r(VA))$




   b) Partitionieren von $S$ mit $h_p(s(VA))$

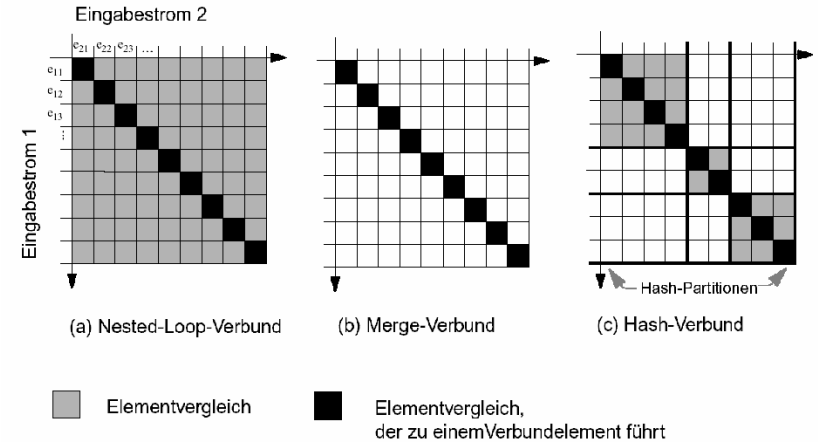
### Hash-Verbund – Beispiel 2/2

1. Verbund




## 11.6 Verbundalgorithmen – Vergleich

### Verbundalgorithmen – Vergleich 1/3

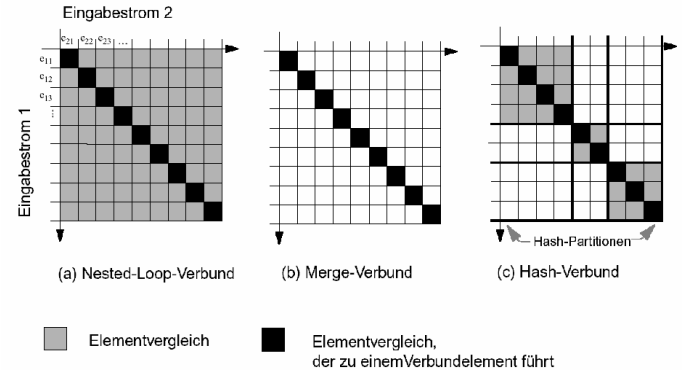



- Nested-Loop-Verbund ist immer anwendbar, jedoch ist dabei stets das vollständige Durchsuchen des gesamten Suchraums in Kauf zu nehmen

### Verbundalgorithmen – Vergleich 2/3

- Merge-Verbund benötigt die geringsten Suchkosten, verlangt aber, daß die Eingabeströme bereits sortiert sind. Indexstrukturen auf beiden Verbundattributen erfüllen diese Voraussetzung. Sonst reduziert das Sortieren beider Relationen nach den Verbundattributen den Kostenvorteil in erheblichem Maße. Ein Sort-Merge-Verbund kann dennoch zusätzliche Vorteile besitzen, falls das Ergebnis in sortierter Folge verlangt wird und das Sortieren des großen Ergebnisses aufwendiger ist als das Sortieren zweier kleiner Ergebnismengen.

### Verbundalgorithmen – Vergleich 3/3




- Beim Hash-Verbund wird der Suchraum partitioniert. In Bild (c) ist unterstellt, daß die gleiche Hash-Funktion $h$ auf die Relationen $R$ und $S$ angewendet worden ist. Die Partitionsgröße (bei der kleineren) Relation richtet sich nach der verfügbaren Puffergröße im Hauptspeicher. Eine Verkleinerung der Partitionsgröße, um den Fall (b) anzunähern, verursacht höhere Vorbereitungskosten und ist deshalb nicht zu empfehlen

## 11.7 Verbundalgorithmen in verteilten DBS

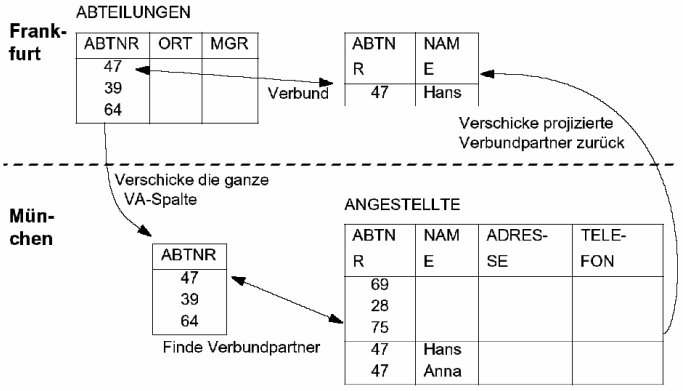
### Verbundalgorithmen in verteilten DBS 1/2

- Problemstellung: Anfrage in Knoten $K$, die einen Verbund zwischen (Teil-) Relationen $R$ am Knoten $K_R$ und (Teil-) Relation $S$ am Knoten $K_S$ erfordert
  - Festlegung des Ausführungsknotens: $K$, $K_R$ oder $K_S$
- Auswertestrategien
  - Sende beteiligte Relationen vollständig an einen Knoten und führe lokale Verbundberechnung durch (Ship Whole)
    - minimale Nachrichtenanzahl
    - sehr hohes Übertragungsvolumen
  - Fordere für jeden Verbundwert der ersten Relation zugehörige Tupel der zweiten an (Fetch as Needed)
    - hohe Nachrichtenanzahl
    - nur relevante Tupel werden berücksichtigt
  - Kompromißlösungen
    - Semi-Verbund bzw. Erweiterungen wie Bit-Vektor-Verbund (Hash-Filter-Join)

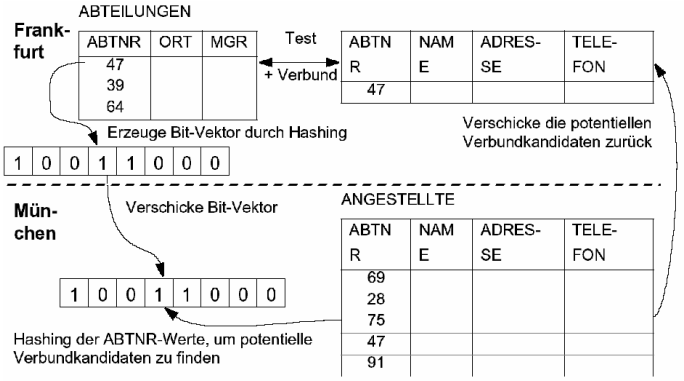
### Verbundalgorithmen in verteilten DBS 2/2

- Semi-Verbund
  - Versenden einer Liste der $VA$ von $R$ zum Knoten $S$
  - Ermitteln der Verbundpartner in $S$ und Zurückschicken zum Knoten von $R$
  - Durchführung des Verbundes
- Bit-Vektor-Verbund
  - ähnlich wie Semi-Verbund, nur Versenden eines durch Hashfunktion erstellten Bitvektors (Bloom-Filter)
  - Rücksenden einer Obermenge der Verbundpartner in $S$

### Semi-Verbund

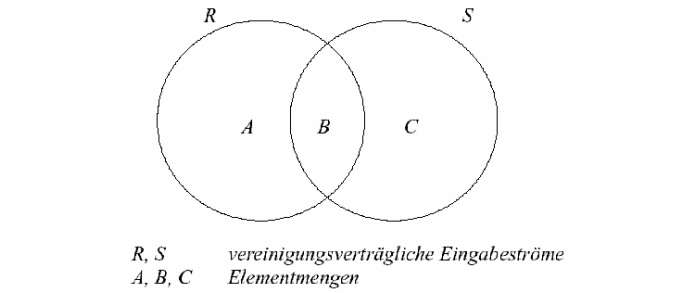



### Bit-Vektor-Verbund




## 11.8 Mengenoperationen

### Mengenoperationen




*R, S* *vereinigungsverträgliche Eingabeströme*
*A, B, C* *Elementmengen*

| Operations-ergebnis | Übereinstimmung in allen Attributen | Übereinstimmung in einem oder mehreren Attributen |
|---|---|---|
| A | Differenz (R–S) | Anti-Semiverbund (S, R) |
| B | Durchschnitt | Verbund, Semiverbund (S, R) |
| C | Differenz (S–R) | Anti-Semiverbund (R, S) |
| A, B | | linksseitiger Äußerer Verbund |
| A, C | Anti-Differenz | Anti-Verbund |
| B, C | | rechtsseitiger Äußerer Verbund |
| A, B, C | Vereinigung | symmetrischer Äußerer Verbund |

### Zusammenfassung

- Selektionsoperationen
  - vorhandene Zugriffspfadtypen erfordern zugeschnittene Operationen und effiziente Abbildung

  - Kombination verschiedener Zugriffspfade möglich (TID-Algorithmus)

- Allgemeine Klassen von Auswertungsverfahren für binäre Operationen
  - Schleifeniteration (nested iteration)
  - Mischmethode (merge method)
  - Hash-Methode (hashing)

- Viele Optionen zur Durchführung von Verbünden
  - Nested-Loop-Verbund
  - Sort-Merge-Verbund
  - Hash-Verbund
  - und Variationen

- Mengenoperationen
  - prinzipiell Nutzung der gleichen Verfahrensklassen
  - Variation der Vergleichsdurchführung

# 12 Mengenorientierte DB-Schnittstelle

## 12.1 Übersetzung deskriptiver Anfragen

### Logische Datenstrukturen

- Charakterisierung der Abbildung

```
SELECT PNR, ABT-NAME
FROM ABTEILUNG, PERS, FAEHIGKEIT
WHERE BERUF = "PROGRAMMIERER" &
      FAEHIGKEIT.FA-NR = PERS.FA-NR &
      PERS.ABT-NR = ABTEILUNG.ABT-NR
```

- Abbildungsfunktionen
  - Sichten ↔ Basisrelationen
  - Relation. Ausdrücke ↔ Logische Zugriffspfade
  - Satzmengen ↔ Einzelne Sätze, Pos.-anzeiger

```
FETCH FAEHIGKEIT USING ...
FETCH NEXT PERS ...
FETCH OWNER WITHIN ...
```

- Eigenschaften der oberen Schnittstelle
  - Zugriffspfad-unabhängiges (relationales) Datenmodell
  - Alle Sachverhalte und Beziehungen werden durch Werte dargestellt
  - Nicht-prozedurale (deskriptive) Anfragesprachen
  - Zugriff auf Satzmengen

### Aspekte der Benutzung deskriptiver DB-Sprachen 1/3

- Prozedurale Sprachen
  - Sie erlauben leichte Abbildung der DML-Befehle auf interne Satzoperationen des DBVS (∼ 1:1)
  - Verantwortung für die Zugriffspfadauswahl liegt beim Programmierer; er bestimmt die Art und Reihenfolge der Zugriffe durch Navigation
  - Bei der Übersetzung sind lediglich Namensauflösung und Formatkonversionen erforderlich
- Typische Beispiele (Netzwerkmodell)
  - FIND NEXT PERS WITHIN BESCHÄFTIGT SET bezieht sich auf eine relative Position (Currency) in einer Set-Struktur
  - FIND OWNER WITHIN BESCHÄFTIGT SET stellt den OWNER-Satz der Current-Set-Ausprägung zur Verfügung

  ⇒ Lediglich bei allg. Suchanfragen fallen gewisse Optimierungsaufgaben an; sie sind jedoch auf einen Satztyp beschränkt

### Aspekte der Benutzung deskriptiver DB-Sprachen 2/3

- Deskriptive Anfragen erfordern zusätzlich
  - Überprüfung syntaktischer Korrektheit (komplexere Syntax)
  - Überprüfung von Zugriffsberechtigung und Integritätsbedingungen
  - Anfrageoptimierung zur Erzeugung einer effizient ausführbaren Folge interner DBS-Operationen
- Zentrales Problem
  - Umsetzung deskriptiver Anfragen in eine zeitoptimale Folge interner DBS-Operationen
  - Anfrageübersetzer/-optimierer des DBS ist im wesentlichen für eine effiziente Abarbeitung verantwortlich, nicht der Programmierer
- Hohe Komplexität der Übersetzung

### Aspekte der Benutzung deskriptiver DB-Sprachen 3/3

- Zusätzliche Anforderungen
  - auch die Manipulationsoperationen sind mengenorientiert
  - referentielle Integrität ist aktiv mit Hilfe referentieller Aktionen zu wahren
  - Operationen können sich auf Sichten von Relationen beziehen
  - vielfältige Optionen der Datenkontrolle sind zu berücksichtigen
- Anfrageformulierung
  - Formulierung von nicht angemessenen Anfragen (ohne Zugriffspfadunterstützung) erfordert in navigierenden Anfragesprachen einen erheblichen Programmieraufwand
  - in deskriptiven Anfragesprachen dagegen sind sie genauso leicht zu formulieren wie günstige Anfragen

  $\Rightarrow$ Die Ausführung ist in beiden Fällen gleich langsam

### Übersetzung von DB-Anweisungen 1/3

1. Lexikalische und syntaktische Analyse
   - Erstellung eines Anfragegraphs (AG) als Bezugsstruktur für die nachfolgenden Übersetzungsschritte
   - Überprüfung auf korrekte Syntax (Parsing)
2. Semantische Analyse
   - Feststellung der Existenz und Gültigkeit der referenzierten Relationen, Sichten und Attribute
   - Einsetzen der Sichtdefinitionen in den AG
   - Ersetzen der externen durch interne Namen (Namensauflösung)
   - Konversion vom externen Format in interne Darstellung
3. Zugriffs- und Integritätskontrolle sollen aus Leistungsgründen, soweit möglich, schon zur Übersetzungszeit erfolgen
   - Zugriffskontrolle erfordert bei Wertabhängigkeit Generierung von Laufzeitaktionen
   - Durchführung einfacher Integritätskontrollen (Kontrolle von Formaten und Konversion von Datentypen)
   - Generierung von Laufzeitaktionen für komplexere Kontrollen

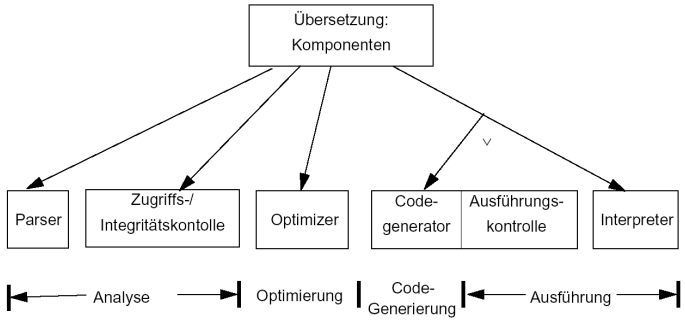
### Übersetzung von DB-Anweisungen 2/3

4. Standardisierung und Vereinfachung
   - Dient der effektiveren Übersetzung und frühzeitigen Fehlererkennung
   - Überführung des AG in eine Normalform
   - Elimination von Redundanzen
5. Restrukturierung und Transformation
   - Anwendung von heuristischen Regeln (algebraische Optimierung) zur Restrukturierung des AG
   - Transformation führt Ersetzung und ggf. Zusammenfassen der logischen Operatoren durch Planoperatoren (nicht-algebraische Optimierung): Meist sind mehrere Planoperatoren als Implementierung eines logischen Operators verfügbar
   - Bestimmung alternativer Ausführungspläne (nicht-algebraische Optimierung): Meist sind viele Ausführungsreihenfolgen oder Zugriffspfade auswählbar
   - Bewertung der Kosten und Auswahl des günstigsten Ausführungsplanes

   ⇒ Schritte 4 + 5 werden als Anfrageoptimierung zusammengefaßt

### Übersetzung von DB-Anweisungen 3/3

6. Code-Generierung
   - Generierung eines zugeschnittenen Programms für die vorgegebene (SQL-) Anfrage
   - Erzeugung eines ausführbaren Zugriffsmoduls
   - Verwaltung der Zugriffsmodule in einer DBVS-Bibliothek




## 12.2 Anfrageoptimierung

### Anfrageoptimierung 1/3

- Von der Anfrage (Was?) zur Auswertung (Wie?) ⇒ Ziel: kostengünstiger Auswertungsweg
- Einsatz einer großen Anzahl von Techniken und Strategien
  - logische Transformation von Anfragen
  - Auswahl von Zugriffspfaden
  - optimierte Speicherung von Daten auf Externspeichern
- Schlüsselproblem
  - genaue Optimierung ist im allgemeinen nicht berechenbar
  - Fehlen von genauer statistischer Information

  - breiter Einsatz von Heuristiken (Daumenregeln)

- Optimierungsziele
  - Maximierung des Outputs bei gegebenen Ressourcen oder
  - Minimierung der Ressourcennutzung für gegebenen Output
  - Durchsatzmaximierung ?
  - Antwortzeitminimierung für eine gegebene Anfragesprache, einem Mix von Anfragen verschiedenen Typs und einer gegebenen Systemumgebung!

### Anfrageoptimierung 2/3

- Welche Kosten sind zu berücksichtigen?
  - Kommunikationskosten in verteilten DBS
    - # der Nachrichten
    - Menge der zu übertragenden Daten
  - Berechnungskosten (CPU-Kosten, Pfadlängen)
  - E/A-Kosten (# der physischen Referenzen)
  - Speicherungskosten
    - temporäre Speicherbelegung im DB-Puffer und auf Externspeichern

  $\Rightarrow$ Kostenarten sind nicht unabhängig $\Rightarrow$ in zentralisierten DBS oft gewichtete Funktion von Berechnungs- und E/A-Kosten

### Anfrageoptimierung 3/3

- Vorgehensweise

  **Schritt 1** Finde nach Übersetzung geeignete Interndarstellung für Anfrage (Anfragegraph)

  **Schritt 2:** Wende logische Restrukturierung auf Anfragegraph an

  **Schritt 3:** Bilde restrukturierte Anfrage auf alternative Folgen von Planoperatoren (Transformation) ab $\Rightarrow$ Menge von Ausführungsplänen

  **Schritt 4:** Berechne Kostenvoranschläge für jeden Ausführungsplan, wähle billigsten aus

### Standardisierung einer Anfrage 1/2

- Standardisierung
  - Wahl einer Normalform, z.B. konjunktive Normalform
    $(A_{11} \vee \cdots \vee A_{1n}) \wedge \cdots \wedge (A_{m1} \vee \cdots \vee A_{mn})$
  - Verschiebung von Quantoren

- Umformungsregeln für Boole'sche Ausdrücke

Kommutativregeln

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| A | **OR** | B | ⇔ | B | **OR** | A | | |
| A | **AND** | B | ⇔ | B | **AND** | A | | |

Assoziativregeln

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| (A | **OR** | B) | **OR** | C | ⇔ | A | **OR** | (B | **OR** | C) |
| (A | **AND** | B) | **AND** | C | ⇔ | A | **AND** | (B | **AND** | C) |

Distributivregeln

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | **OR** | (B | **AND** | C) | ⇔ | (A | **OR** | B) | **AND** | (A | **OR** | C) |
| A | **AND** | (B | **OR** | C) | ⇔ | (A | **AND** | B) | **OR** | (A | **AND** | C) |

De Morgan'sche Regeln

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **NOT** (A | **AND** | B) | ⇔ | **NOT** (A) | **OR** | **NOT** (B) | | |
| **NOT** (A | **OR** | B) | ⇔ | **NOT** (A) | **AND** | **NOT** (B) | | |

Doppelnegationsregel

| | | |
|---|---|---|
| **NOT** (**NOT** (A)) | ⇔ | A |

## Standardisierung einer Anfrage 2/2

- Idempotenzregeln für Boole'sche Ausdrücke

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| A | **OR** | A | | | ⇔ | A |
| A | **AND** | A | | | ⇔ | A |
| A | **OR** | **NOT** (A) | | | ⇔ | TRUE |
| A | **AND** | **NOT** (A) | | | ⇔ | FALSE |
| A | **AND** | (A | **OR** | B) | ⇔ | A |
| A | **OR** | (A | **AND** | B) | ⇔ | A |
| A | **OR** | **FALSE** | | | ⇔ | A |
| A | **OR** | **TRUE** | | | ⇔ | TRUE |
| A | **AND** | **FALSE** | | | ⇔ | FALSE |

## Vereinfachung einer Anfrage 1/2

- Äquivalente Ausdrücke können einen unterschiedlichen Grad an Redundanz besitzen
  - Behandlung/Eliminierung gemeinsamer Teilausdrücke (A1 = a11 or A1 = a12) and (A1 = a12 or A1 = a11)
  - Vereinfachung von Ausdrücken, die an leere Relationen gebunden sind
  - Konstanten-Propagierung A op B and B = const. ⇒ A op const.
  - nicht-erfüllbare Ausdrücke $(A \geq B) \wedge (B > C) \wedge (C \geq A) \Rightarrow A > A \rightarrow false$
- Nutzung von Integritätsbedingungen (IB) IB sind wahr für alle Tupel der betreffenden Relation
  - A ist Primärschlüssel: $\pi_A \rightarrow$ keine Duplikateliminierung erforderlich
  - Regel: FAM-STAND = 'verh.' AND STEUERKLASSE $\geq 3$ ⇒ Ausdruck: (FAM-STAND = 'verh.' AND STEUERKLASSE = 1) → false

## Vereinfachung einer Anfrage 1/2

- Verbesserung der Auswertbarkeit
  - Hinzufügen einer IB zur WHERE-Bedingung verändert den Wahrheitswert eines Auswahlausdrucks nicht ⇒ Einsatz zur verbesserten Auswertung (knowledge-based query processing)
  - einfachere Auswertungsstruktur, jedoch effiziente Heuristiken benötigt

### Anfragerestrukturierung 1/2

- Wichtigste Regeln für Restrukturierung und Transformation
  - Selektionen ($\sigma$) und Projektionen ($\pi$) ohne Duplikateliminierung sollen möglichst frühzeitig ausgeführt werden.
  - Folgen von unären Operatoren (wie $\sigma$ und $\pi$) auf einer Relation sind zu einer Operation mit komplexerem Prädikat zusammenzufassen.
  - Selektionen und Projektionen, die eine Relation betreffen, sollen so zusammengefaßt werden, daß jedes Tupel nur einmal verarbeitet werden muß.
  - Bei Folgen von binären Operatoren (wie $\cup, \cap, -, \times, \bowtie$) ist eine Minimierung der Größe der Zwischenergebnisse anzustreben.
  - Gleiche Teile im AG sind nur einmal auszuwerten.
- Zusammenfassung von Operationsfolgen
  - R1: $\pi_{An}(\ldots \pi_{A2}(\pi_{A1}(Rel))\ldots) \Leftrightarrow \pi_{An}(Rel)$
  - R2: $\sigma_{pn}(\ldots \sigma_{p2}(\sigma_{p1}(Rel))\ldots)\ldots) \Leftrightarrow \sigma_{p1 \wedge p2 \ldots \wedge pn}(Rel)$

### Anfragerestrukturierung 2/2

- Minimierung der Größe von Zwischenergebnissen
  - selektive Op. ($\sigma, \pi$) vor konstruktiven Op. ($\times, \bowtie$)
- Restrukturierungsalgorithmus
  1. Zerlege komplexe Verbundprädikate in binäre Verbunde (Bilden von binären Verbunden).
  2. Teile Selektionen mit mehreren Prädikatstermen in separate Selektionen mit jeweils einem Prädikatsterm auf.
  3. Führe Selektionen so früh wie möglich aus, schiebe Selektionen hinunter zu den Blättern des AG.
  4. Fasse einfache aufeinanderfolgende Selektionen (derselben Relation) zu einer zusammen.
  5. Führe Projektionen ohne Duplikateliminierung so früh wie möglich aus, d. h., schiebe sie soweit wie möglich zu den Blättern des AG hinunter (projection push-down).
  6. Fasse einfache Projektionen (derselben Relation) zu einer Operation zusammen.

### Anfragetransformation 1/2

- Zusammenfassung von logischen Operatoren (Ein- und Zwei-Variablen-Ausdrücke) und ihre Ersetzung durch Planoperatoren
- Typische Planoperatoren in relationalen Systemen
  - auf einer Relation
    - Selektion
    - Projektion
    - Sortierung
    - Aggregation
    - Änderungsoperationen (Einfügen, Löschen, Modifizieren)
    - ACCESS zum Zugriff auf Basisrelationen

  - auf zwei Relationen
    * Verbund- und Mengen-Operationen
    * Kartesisches Produkt

### Anfragetransformation 2/2

- Anpassungen im AG zum effektiven Einsatz von Planoperatoren
  1. Gruppierung von direkt benachbarten Operatoren zur Auswertung durch einen Planoperator;
     - z. B. lassen sich durch einen speziellen Planoperator ersetzen: Verbund (oder Kartesisches Produkt) mit Selektionen und/oder Projektionen auf den beteiligten Relationen.
  2. Bestimmung der Verknüpfungsreihenfolge bei Mengen- und Verbundoperationen
     - Dabei sollen die minimalen Kosten für die Operationsfolge erzielt werden.
     - Als Heuristik ist dazu die Größe der Zwischenergebnisse zu minimieren, d. h., die kleinsten (Zwischen-)Relationen sind immer zuerst zu verknüpfen.
  3. Erkennung gemeinsamer Teilbäume
     - die dann nur jeweils einmal zu berechnen sind.
     - Dieser Einsparung steht die Zwischenspeichung der Ergebnisrelation gegenüber.

### Bewertung von Ausführungsplänen – Grundsätzliche Probleme

- Anfrageoptimierung beruht i.a. auf zwei Annahmen
  1. Alle Datenelemente und alle Attributwerte sind gleichverteilt
  2. Suchprädikate in Anfragen sind unabhängig

  $\Rightarrow$ beide Annahmen sind im allgemeinen falsch
- Beispiel
  - (GEHALT $\geq$ 100K) AND (ALTER BETWEEN 20 AND 30)
  - Bereiche: Gehalt$\in$ [10K,1M], Alter$\in [20, 65]$

  $\Rightarrow$ lineare Interpolation, Multiplikation von Wahrscheinlichkeiten
- Lösung
  - Verbesserung der Statistiken/Heuristiken
  - Berechnung/Bewertung von noch mehr Ausführungsplänen
  - Obwohl die Kostenabschätzungen meist falsch sind ...

### Erstellung und Auswahl von Ausführungsplänen 1/7

- Eingabe:
  - optimierter Anfragegraph (AG)
  - existierende Speicherungsstrukturen und Zugriffspfade
  - Kostenmodell
- Ausgabe: optimaler Ausführungsplan (oder wenigstens gut)
- Vorgehensweise:
  1. Generiere alle vernünftigen logischen Ausführungspläne zur Auswertung der Anfrage

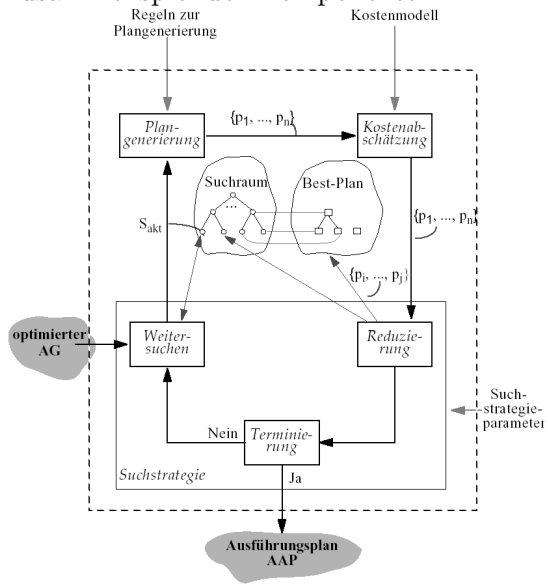
2. Vervollständige Ausführungspläne durch Einzelheiten der physischen Datenrepräsentation (Sortierreihenfolge, Zugriffspfadmerkmale, statistische Information)
3. Wähle den billigsten Ausführungsplan gemäß dem vorgegebenen Kostenmodell aus

### Erstellung und Auswahl von Ausführungsplänen 2/7

- Alternative Ausführungspläne für einen AG entstehen durch
  - verschiedene Methoden (Implementierungen) für Planoperatoren
  - Operationsreihenfolgen (z. B. bei Mehrfachverbunden) können variiert werden
  - komplexe Anfragen haben sehr große Suchräume mit Alternativen (z. B. $10^{70}$ mögliche Ausführungspläne bei einer Anfrage mit 15 Verbunden).
- Generierung durch Optimierer
  - $\Rightarrow$ kleine Menge der Pläne, die den optimalen Plan enthält
  - $\Rightarrow$ Einschränkung durch Heuristiken
  - hierarchische Generierung basierend auf dem Schachtelungskonzept von SQL
  - Zerlegung in eine Menge von Teilanfragen mit höchstens Zwei-Variablen-Ausdrücken

### Erstellung und Auswahl von Ausführungsplänen 3/7

- Zusammenspiel der Komponenten




### Erstellung und Auswahl von Ausführungsplänen 4/7

- Plangenerierung soll
  - immer und möglichst schnell den optimalen Plan finden
  - mit einer möglichst kleinen Anzahl generierter Pläne auskommen
- Suchstrategien
  - voll-enumerativ
  - beschränkt-enumerativ
  - zufallsgesteuert $\Rightarrow$ Reduzierung: Bestimmte Suchpfade zur Erstellung von AAPs werden nicht weiter verfolgt

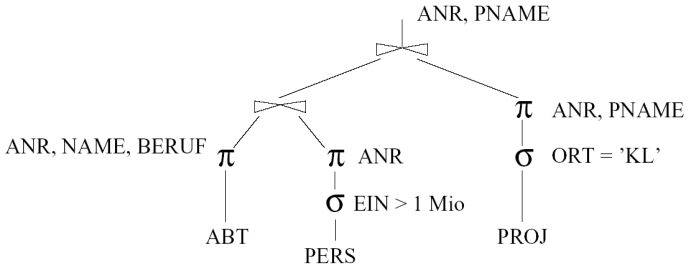
- Kostenabschätzung
  - verlangt hinreichend genaues Kostenmodell
  - wird bei allen Suchverfahren inkrementell durchgeführt

### Erstellung und Auswahl von Ausführungsplänen 5/7

- Beispielproblem
- SQL-Anfrage:

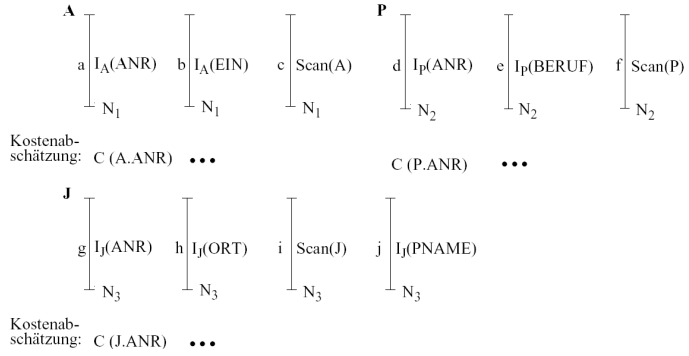
```
SELECT P.NAME, P.BERUF, J.PNAME
FROM PERS P, ABT A, PROJ J
WHERE A.EIN > 1000000 AND J.ORT = 'KL'
AND A.ANR = P.ANR AND A.ANR = J.ANR;
```

- Zugehöriger Anfragegraph

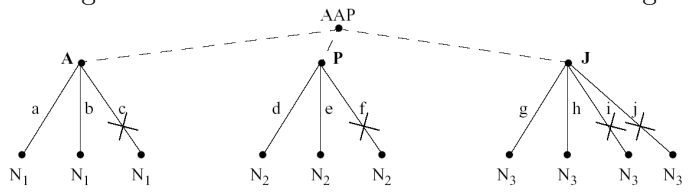



### Erstellung und Auswahl von Ausführungsplänen 6/7

1. mögliche Zugriffspfade für die einzelnen Relationen



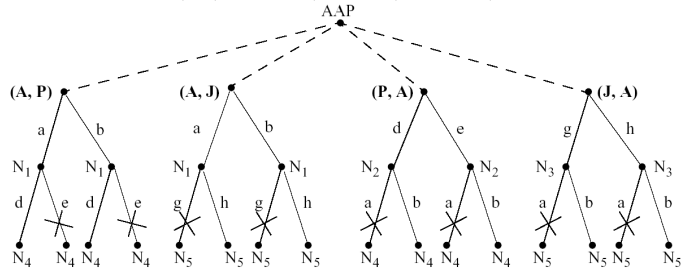

2. Lösungsbaum für einzelne Relationen: Reduzierung durch Abschneiden von Teilbäumen




### Erstellung und Auswahl von Ausführungsplänen 7/7

3. Erweiterter Lösungsbaum für den Nested-Loop-Verbund mit der zweiten Relation Kostenabschätzung pro Pfad:

z. B. durch C(C(A.ANR) + C(P.ANR) + Verbundkosten)




## Ausführungsplan – Beispiel

- SQL-Anfrage-Beispiel

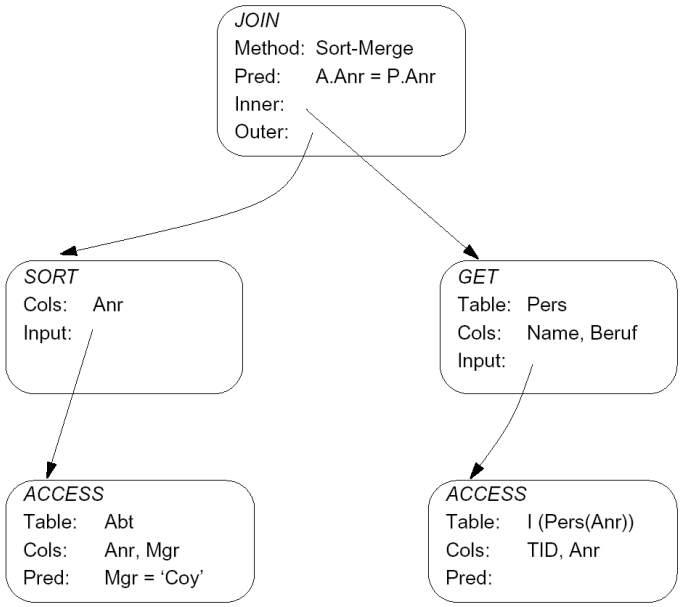
```
SELECT Name, Beruf
FROM Pers P, Abt A
WHERE P.Anr = A.Anr
AND A.Mgr = 'Coy'
```

- Dazugehöriges Programm

```
JOIN ( Sort-Merge, A.Anr = P.Anr,
  SORT (ACCESS (Abt, {Anr, Mgr}, {Mgr = 'Coy'}), Anr),
  GET (ACCESS (I (Pers(Anr)), {TID, Anr}, emptyset),
    Pers, {Name, Beruf} , emptyset)).
```

## Ausführungsplan – Beispiel

- Ein möglicher Operatorbaum




## Berechnung der Zugriffskosten 1/2

- Optimierer erstellt Kostenvoranschlag für jeden Ausführungsplan
- Gewichtete Kostenformel

$C =$ #physischer Seitenzugriffe+

$W *$ (#Aufrufe des Zugriffssystems)

- gewichtetes Maß für E/A- und CPU-Auslastung
- W ist das Verhältnis des Aufwandes für einen ZS-Aufruf zu einem Seitenzugriff

### Berechnung der Zugriffskosten 2/2

- Ziel der Gewichtung: Minimierung der Kosten in Abhängigkeit des Systemzustandes
  - System I/O-bound: ⇒ kleines W

$$W_{I/O} = \frac{\#\text{Instr. pro ZS Aufruf}}{\#\text{Instr.pro E/A} + \text{Zugriffszeit} \cdot \text{MIPS-Rate}}$$

    Bsp.

$$W_{I/O} = \frac{1000 \text{ Instr.}}{2500 \text{ Instr.} + 12 \text{ msec} \cdot 10^7 \text{Instr./sec}} = 0.008$$

  - System CPU-bound: ⇒ relativ großes W

$$W_{CPU} = \frac{\#\text{Instruktionen pro ZS-Aufruf}}{\#\text{Instruktionen pro E/A}}$$

    Bsp.

$$W_{CPU} = \frac{1000}{2500} = 0.4$$

### Kostenmodell – statistische Werte

- statistische Größen für Segmente
  - $M_S$ Anzahl der Datenseiten des Segmentes $S$
  - $L_S$ Anzahl der leeren Seiten in $S$
- statistische Größen für Relationen
  - $N_R$ Anzahl der Tupeln der Relation $R$ ($Card(R)$)
  - $T_{R,S}$ Anzahl der Seiten in $S$ mit Tupeln von $R$
  - $C_R$ Clusterfaktor (Anzahl Tupel pro Seite)
- statistische Größen pro Index $I$ auf Attributen $A$ einer Rel. $R$
  - $j_I$ Anzahl der Attributwerte / Schlüsselwerte im Index ($= Card(\pi_A(R))$
  - $B_I$ Anzahl der Blattseiten (B*-Baum)
- Statistiken müssen im DB-Katalog gewartet werden
  - Aktualisierung bei jeder Änderung sehr aufwendig
  - zusätzliche Schreib- und Log-Operationen
  - DB-Katalog wird zum Sperr-Engpaß
- Alternative
  - Initialisierung der statistischen Werte zum Lade- oder Generierungszeitpunkt von Relationen und Indexstrukturen
  - periodische Neubestimmung der Statistiken durch eigenes Kommando/Dienstprogramm

### Kostenmodell – Berechnungsgrundlagen 1/2

- Mit Statistiken kann Optimierer jedem Verbundterm im Qualifikationsprädikat einen Selektivitätsfaktor ($0 \leq SF \leq 1$) zuordnen
  - erwarteter Anteil an Tupeln, die das Prädikat erfüllen
  - $Card(\sigma_p(R)) = SF(p) \cdot Card(R)$
- Selektivitätsfaktor SF bei:

| | | | |
|---|---|---|---|
| $A_i = a_i$ | SF = | $1/j_i$ | wenn Index auf $A_i$ |
| | | 1/10 | sonst |
| $A_i = A_k$ | SF = | $1 / Max(j_i, j_k)$ | wenn Index auf $A_i$, $A_k$ |
| | | $1 / j_i$ | wenn Index auf $A_i$ |
| | | 1/10 | sonst |
| $A_i \geq a_i$ | SF = | (high-key - $a_i$) / (high-key - low-key) | bei linearer Interpolation |
| | | 1/3 | sonst |
| $A_i \geq a_i \wedge A_i \leq a_k$ | SF = | ($a_k$-$a_i$) / (high-key - low-key) | Index auf $A_i$ |
| | | 1/4 | sonst |
| $A_i$ IN (Liste von Werten) | SF = | $r / j_i$ | bei r Werten auf Index |
| | | 1/2 | sonst |

### Kostenmodell – Berechnungsgrundlagen 1/2

- Berechnung von Ausdrücken
  - $SF(p(A) \wedge p(B)) = SF(p(A)) \cdot SF(p(B))$
  - $SF(p(A) \vee p(B)) = SF(p(A)) + SF(p(B)) - SF(p(A)) \cdot SF(p(B))$
  - $SF(\neg p(A)) = 1 - SF(p(A))$
- Join-Selektivitätsfaktor (JSF)
  - $Card(R \bowtie S) = JSF \cdot Card(R) \cdot Card(S)$
  - bei (N:1)-Joins (verlustfrei)
    $Card(RS) = Max(Card(R), Card(S))$

### Beispiel: Einfache Anfrage

- SQL-Anfrage

```
SELECT NAME, GEHALT
FROM PERS
WHERE BERUF = PROGRAMMIERER
      AND GEHALT >= 100.000
```

- Vorhandene Zugriffspfade
  - $I_{PERS}(BERUF)$
  - $I_{PERS}(GEHALT)$
- Statistische Kennwerte (aus DB-Katalog):
  - $N = \#$ der Tupel in Relation $PERS$ ($N = 5000$)
  - $C =$ durchschn. $\#$ von $PERS$-Tupeln pro Seite ($C = 4$)

  - $j_i$ = Index-Kardinalität (#Attributwerte für $A_i$)
    - es gibt 25 unterschiedliche Berufe
    - die Gehälter liegen zwischen 30.000 und 120.000
  - Information über Clusterbildung

**Methode 1: Scan über $I_{PERS}(BERUF)$**

**Methode 2: Scan über $I_{PERS}(GEHALT)$**

**Zusammenfassung**

- Anfrageoptimierung: Kernproblem der Übersetzung mengenorientierter DB-Sprachen
  - fatale Annahmen:
    1. Gleichverteilung aller Attributwerte
    2. Unabhängigkeit aller Attribute
  - Kostenvoranschläge für Ausführungspläne:
    1. CPU-Zeit und E/A-Aufwand
    2. Anzahl der Nachrichten und zu übertragende Datenvolumina (im verteilten Fall)
  - gute Heuristiken zur Erstellung und Auswahl von Ausführungsplänen sehr wichtig

# 13 Synchronisation und Transaktionen

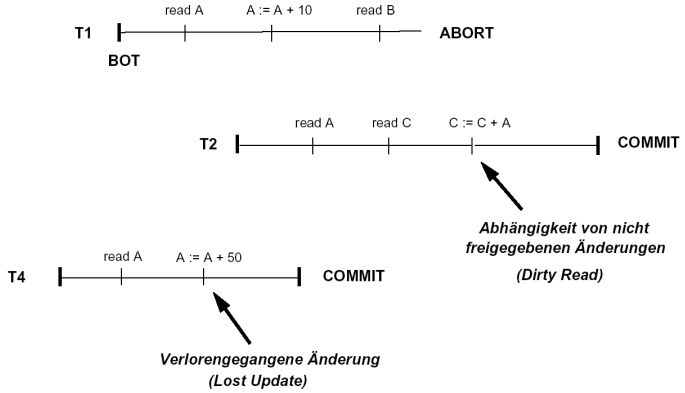
## 13.1 Anomalien im Mehrbenutzerbetrieb

### Anomalien im Mehrbenutzerbetrieb ohne Synchronisation

1. Verlorengegangene Änderungen (lost updates)
2. Abhängigkeiten von nicht freigegebenen Änderungen (dirty read, dirty overwrite)
3. Inkonsistente Analyse (non-repeatable read)
4. Phantom-Problem

$\Rightarrow$ nur durch Änderungs-TA verursacht

### Beispiele für Anomalien




### Inkonsistente Analyse (Non-repeatable Read)

| Lesetransaktion (Gehaltssumme berechnen) | Änderungstransaktion | DB-Inhalt (PNR, GEHALT) | |
|---|---|---|---|
| SELECT GEHALT INTO gehalt | | 2345 | 39.000 |
| FROM PERS | | 3456 | 48.000 |
| WHERE PNR=2345 | | | |
| summe := summe + gehalt | | | |
| | UPDATE PERS | | |
| | SET GEHALT = GEHALT + 1000 | | |
| | WHERE PNR = 2345 | 2345 | 40.000 |
| | UPDATE PERS | | |
| | SET GEHALT = GEHALT + 2000 | | |
| | WHERE PNR = 3456 | 3456 | 50.000 |
| SELECT GEHALT INTO gehalt | | | |
| FROM PERS | | | |
| WHERE PNR = 3456 | | | |
| summe := summe + gehalt | | | |

### Phantom-Problem

| Lesetransaktion (Gehaltssumme überprüfen) | Änderungstransaktion (Einfügen eines neuen Angestellten) |
|---|---|
| SELECT SUM (GEHALT) INTO summe<br>FROM PERS<br>WHERE ANR=17 | |
| | INSERT INTO PERS (PNR, ANR, GEHALT)<br>VALUES (4567, 17, 55.000) |
| | UPDATE ABT<br>SET GEHALTSSUMME = GEHALTSSUMME + 55.000<br>WHERE ANR=17 |
| SELECT GEHALTSSUMME INTO gsumme<br>FROM ABT<br>WHERE ANR= 17 | |
| IF gsumme <> summe THEN<br><Fehlerbehandlung> | |

## 13.2 Serialisierbarkeit

### Synchronisation von Transaktionen: Modellannahmen 1/2

- TRANSAKTION: Ein Programm T mit DML-Anweisungen, das folgende Eigenschaft erfüllt
  - Wenn T allein auf einer konsistenten DB ausgeführt wird, dann terminiert T (irgendwann) und hinterläßt die DB in einem konsistenten Zustand.
  - Während der TA-Verarbeitung werden keine Konsistenzgarantien eingehalten
- Wenn Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der DB erhalten.
- Modellbildung für die Synchronisation
  - DML-Anweisungen durch Lese- und Schreiboperationen auf Seiten nachbilden
    - READ (O)
    - WRITE (O:=4)
  - DBS sieht TA als: BOT, Folge von READ- und WRITE-Anweisungen, EOT

### Synchronisation von Transaktionen: Modellannahmen 2/2

- Ablauffolge von TA mit Operationen durch Plan (Schedule) beschrieben
- Beispiel
  - $r_1(x), r_2(x), r_3(y), w_1(x), w_3(y), r_1(y), c_1, r_3(x), w_2(x),\ a_2, w_3(x), c_3, \ldots$
- Beispiel eines seriellen Plans
  - $r_1(x), w_1(x), r_1(y), c_1, r_3(y), w_3(y), r_3(x), c_3, r_2(x),\ w_2(x), c_2, \ldots$
- BOT ist implizit, EOT wird durch $c_i$ (commit) oder $a_i$ (abort) dargestellt

### Korrektheitskriterium der Synchronisation 1/2

- Ziel der Synchronisation
  - logischer Einbenutzerbetrieb, d. h. Vermeidung aller Mehrbenutzeranomalien
- Gleichbedeutend mit dem formalen Korrektheitskriterium der Serialisierbarkeit
  - Die parallele Ausführung einer Menge von Transaktionen ist serialisierbar, wenn es eine serielle Ausführung derselben TA-Menge gibt, die den gleichen DB-Zustand und die gleichen Ausgabewerte wie die ursprüngliche Ausführung erzielt.

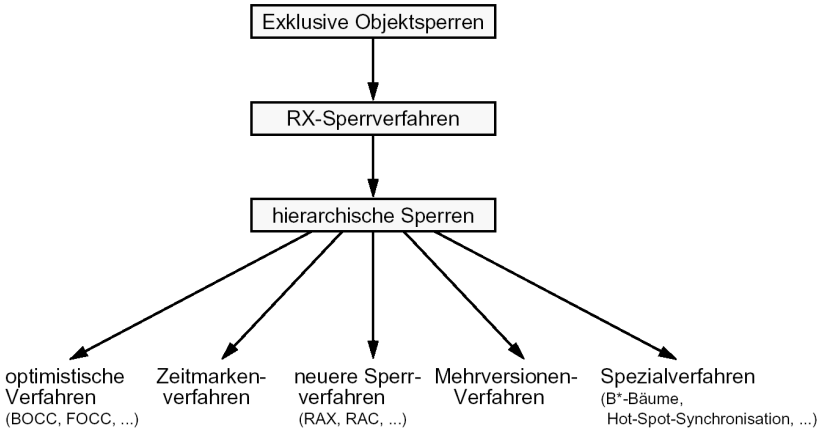
- Hintergrund:
  - serielle Ablaufpläne sind korrekt
  - jeder Ablaufplan, der denselben Effekt wie ein serieller erzielt, ist akzeptierbar

### Korrektheitskriterium der Synchronisation 2/2

- Nachweis der Serialisierbarkeit
  - Führen von zeitlichen Abhängigkeiten zwischen TA in einem Abhängigkeitsgraphen
  - Abhängigkeit (Konflikt) besteht, wenn zwei TA auf dasselbe Objekt mit nicht reihenfolgeunabhängigen Operationen zugreifen
    - z. B. Schreib-/Lese-, Lese-/Schreib-, Schreib-/Schreib-Konflikte
  - Serialisierbarkeit liegt vor, wenn der Abhängigkeitsgraph keine Zyklen enthält
    - Abhängigkeitsgraph beschreibt partielle Ordnung zwischen TA, die sich zu einer vollständigen erweitern läßt (Serialisierungsreihenfolge)

## 13.3 Sperrverfahren

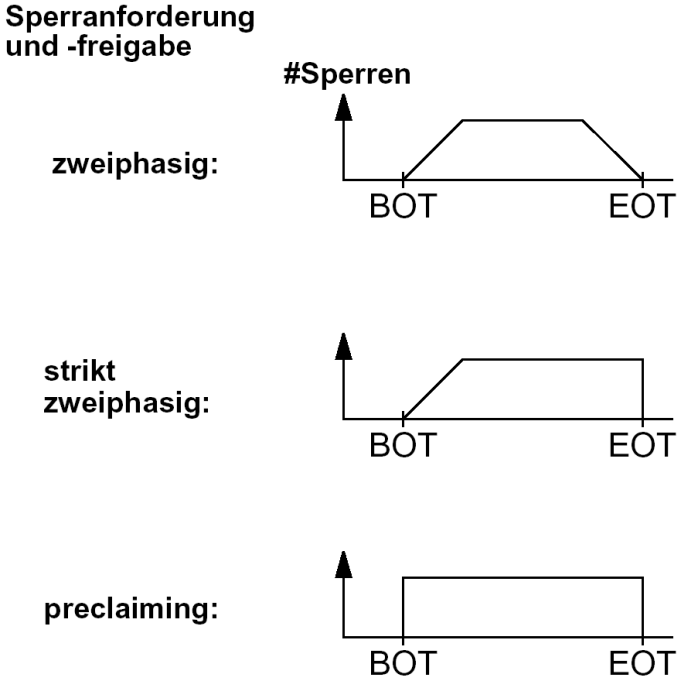
### Historische Entwicklung von Synchronisationsverfahren




### Zweiphasen-Sperrprotokolle (2PL)

- Einhaltung folgender Regeln gewährleistet Serialisierbarkeit
  1. Vor jedem Objektzugriff muß Sperre mit ausreichendem Modus angefordert werden
  2. Gesetzte Sperren anderer TA sind zu beachten
  3. Eine TA darf nicht mehrere Sperren für ein Objekt anfordern
  4. Zweiphasigkeit
     - Anfordern von Sperren erfolgt in einer Wachstumsphase
     - Freigabe der Sperren in Schrumpfungsphase
     - Sperrfreigabe kann erst beginnen, wenn alle Sperren gehalten werden

5. Spätestens bei EOT sind alle Sperren freizugeben



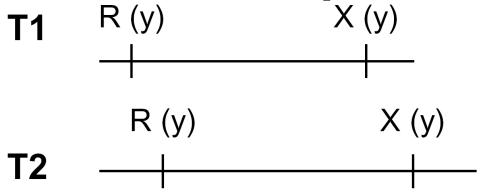

### RX - Sperrverfahren

- Gewährter Sperrmodus des Obj.: NL, R, X
- Sperranforderung einer Transaktion: R, X
- Kompatibilitätsmatrix

| angeforderter Modus \ aktueller Modus | NL | R | X |
|---|---|---|---|
| R | + | + | - |
| X | + | - | - |

(NL (no lock) wird meist weggelassen)

- Deadlock-Gefahr durch Sperrkonversionen




### RUX - Sperrverfahren

- Erweitertes Sperrverfahren:
  - Ziel: Verhinderung von Konversions-Deadlocks
  - U-Sperre für Lesen mit Änderungsabsicht

– bei Änderung Konversion $U \rightarrow X$, andernfalls $U \rightarrow R$ (downgrading)

| | R | U | X |
|---|---|---|---|
| R | + | - | - |
| U | + | - | - |
| X | - | - | - |

– das Verfahren ist unsymmetrisch – was würde eine Symmetrie bei U bewirken?

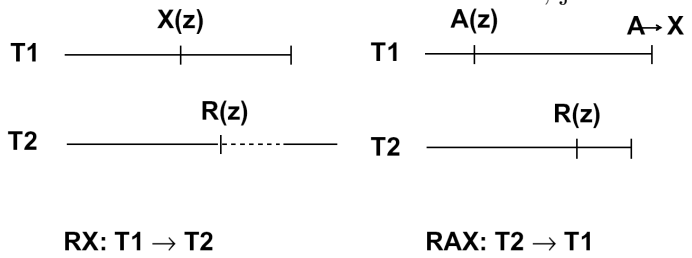
**RAX-Sperrverfahren 1/2**

| | R | A | X |
|---|---|---|---|
| R | + | ⊕ | - |
| A | ⊕ | - | - |
| X | - | - | - |

- Änderungen erfolgen in temporärer Objektkopie, paralleles Lesen der gültigen Version wird zugelassen
- Schreiben wird nach wie vor sequentialisiert (A-Sperre)
- bei EOT Konversion der A- und X-Sperren, ggf. auf Freigabe von Lesesperren warten (Deadlock-Gefahr)

**RAX-Sperrverfahren 2/2**

- höhere Parallelität als beim RX-Verfahren, jedoch i. a. andere Serialisierungsreihenfolge:




- starke Behinderungen von Update-TA durch (lange) Leser möglich

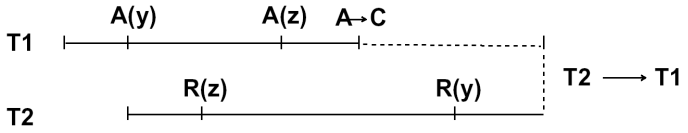
**RAC-Sperrverfahren 1/2**

| | R | A | C |
|---|---|---|---|
| R | + | + | ⊕ |
| A | + | - | - |
| C | ⊕ | - | - |

- Änderungen erfolgen ebenfalls in temporärer Objektkopie, A-Sperre erforderlich
- bei EOT Konversion von $A \rightarrow C$-Sperre

- $C$-Sperre zeigt Existenz zweier gültiger Objektversionen an ⇒ kein Warten auf Freigabe von Lesesperren auf alter Version (R- und C-Modus sind verträglich)
- maximal 2 Versionen, da C-Sperren mit sich selbst und mit A-Sperren unverträglich sind

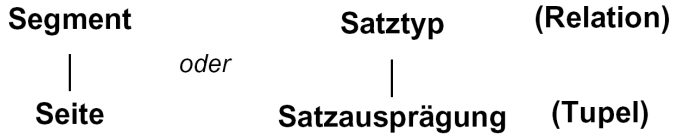
### RAC-Sperrverfahren 2/2




- Leseanforderungen bewirken nie Blockierung/Rücksetzung, jedoch: Auswahl der richtigen Version erforderlich (z. B. über Abhängigkeitsgraphen)
- Änderungs-TA, die auf C-Sperre laufen, müssen warten, bis alle Leser der alten Version beendet, weil nur 2 Versionen ⇒ Abhilfe: allgemeines Mehrversionen-Konzept

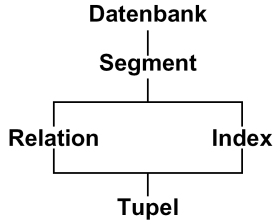
### Hierarchische Sperrverfahren

- Sperrgranulat bestimmt Parallelität/Aufwand
  - feines Granulat reduziert Sperrkonflikte,
  - jedoch sind viele Sperren anzufordern und zu verwalten
- Hierarchische Verfahren erlauben Flexibilität bei Wahl des Granulates (multigranularity locking), z. B.
  - lange TA auf Relationenebene
  - kurze TA auf Tupelebene synchronisieren
- kommerzielle DBS unterstützen zumeist mindestens 2-stufige Objekthierarchie, z. B.



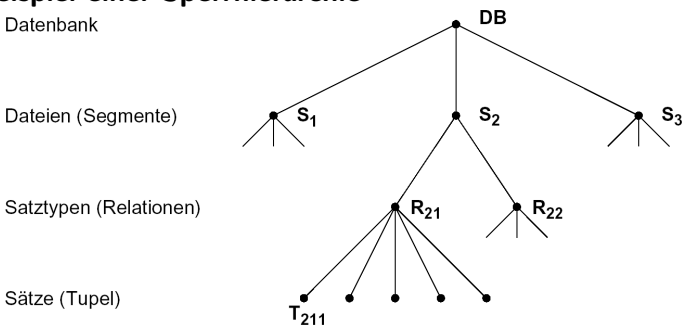

### Hierarchische Sperrverfahren

- Verfahren nicht auf reine Hierarchien beschränkt, sondern auch auf halbgeordnete Objektmengen erweiterbar.




- Verfahren erheblich komplexer als einfache Sperrverfahren (mehr Sperrmodi, Konversionen, Deadlock Behandlung, . . . )

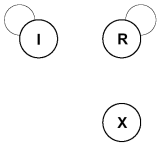
### Beispiel einer Sperrhierarchie

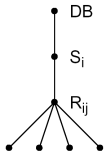



### Hierarchische Sperrverfahren: Anwartschaftssperren

- Mit R- und X-Sperre werden alle Nachfolgerknoten implizit mitgesperrt ⇒ Einsparungen möglich
- Alle Vorgängerknoten sind ebenfalls zu sperren, um Unverträglichkeiten zu vermeiden
  - Verwendung von Anwartschaftssperren (intention locks) Allgemeine Anwartschaftssperre (I-Sperre)

| | I | R | X |
|---|---|---|---|
| I | + | - | - |
| R | - | + | - |
| X | - | - | - |



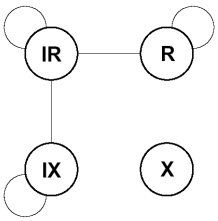



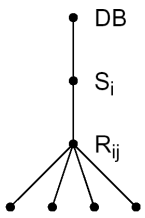

- Unverträglichkeit von I- u. R-Sperren zu restriktiv ⇒ zwei Arten von Anwartschaftssperren (IR und IX)

### Anwartschaftssperren 1/4

- Anwartschaftssperren für Leser und Schreiber

| | IR | IX | R | X |
|---|---|---|---|---|
| IR | + | + | + | - |
| IX | + | + | - | - |
| R | + | - | + | - |
| X | - | - | - | - |




- IR-Sperre (intent read), falls auf untergeordneten Objekten nur lesend zugegriffen wird, sonst IX-Sperre

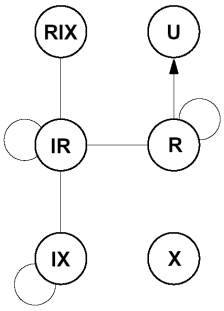
### Anwartschaftssperren 2/4

- Weitere Verfeinerung sinnvoll, um den Fall zu unterstützen, wo alle Tupel eines Satztyps gelesen und nur einige davon geändert werden sollen
- X-Sperre auf Satztyp sehr restriktiv
- IX-Sperre auf Satztyp verlangt Sperren jedes Tupels
  - neuer Typ von Anwartschaftssperre: RIX = R + IX
  - sperrt das Objekt in R-Modus und verlangt
  - X-Sperren auf tieferer Hierarchieebene nur für zu ändernde Objekte

### Anwartschaftssperren 3/4

- Vollständiges Protokoll der Anwartschaftssperren
  - RIX gibt ein Leserecht auf den Knoten und seine Nachfolger. Weiterhin ist damit das Recht verbunden, auf Nachfolger-Knoten IX, U und X-Sperren anzufordern.
  - U gewährt ein Leserecht auf den Knoten und seine Nachfolger. Dieser Modus repräsentiert die Absicht, den Knoten in der Zukunft zu verändern. Bei Änderung Konversion $U \rightarrow X$, sonst $U \rightarrow R$.

| | IR | IX | R | RIX | U | X |
|---|---|---|---|---|---|---|
| IR | + | + | + | + | - | - |
| IX | + | + | - | - | - | - |
| R | + | - | + | - | - | - |
| RIX | + | - | - | - | - | - |
| U | - | - | + | - | - | - |
| X | - | - | - | - | - | - |




### Anwartschaftssperren 4/4

- Sperrdisziplin erforderlich
  - Sperranforderungen von der Wurzel zu den Blättern
  - Bevor T eine R- oder IR-Sperre für einen Knoten anfordert, muß sie für alle Vorgängerknoten IX- oder IR-Sperren besitzen
  - Bei einer X-, U-, RIX- oder IX-Anforderung müssen alle Vorgängerknoten in RIX oder IX gehalten werden
  - Sperrfreigaben von den Blättern zu der Wurzel
  - Bei EOT sind alle Sperren freizugeben

### Deadlock-Behandlung

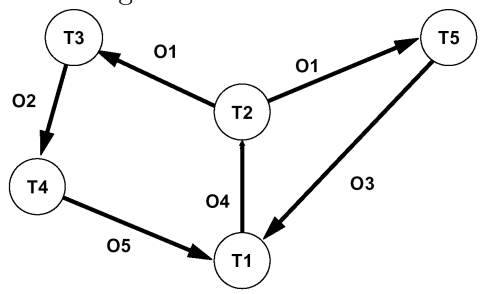
- Voraussetzungen für Deadlock:
  - paralleler Zugriff
  - exklusive Zugriffsanforderungen
  - anfordernde TA besitzt bereits Objekte/Sperren
  - keine vorzeitige Freigabe von Objekten/Sperren (non-preemption)

- zyklische Wartebeziehung zwischen zwei oder mehr TA

- Lösungsmöglichkeiten:
  1. Timeout-Verfahren
     - TA wird nach festgelegter Wartezeit auf Sperre zurückgesetzt
     - problematische Bestimmung des Timeout-Wertes
  2. Deadlock-Verhütung (Prevention)
     - keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
     - Bsp.: Preclaiming (in DBS i. a. nicht praktikabel)
  3. Deadlock-Vermeidung (Avoidance)
     - potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden ⇒ Laufzeitunterstützung nötig
  4. Deadlock-Erkennung (Detection)

### Deadlock-Erkennung

- Explizites Führen eines Wartegraphen (wait-for graph) und Zyklensuche zur Erkennung von Verklemmungen




- Deadlock-Auflösung durch Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA (z. B. Verursacher oder billigste TA zurücksetzen)

- Zyklensuche entweder
  - bei jedem Sperrkonflikt bzw.
  - verzögert (z. B. über Timeout gesteuert)

### Sperrverfahren in Datenbanksystemen 1/2

- Aufgabe von Sperrverfahren: Vermeidung von Anomalien
  - zu ändernde Objekte dem Zugriff aller anderen Transaktionen entzogen werden
  - zu lesende Objekte vor Änderungen geschützt werden

- Standardverfahren: Hierarchisches Zweiphasen-Sperrprotokoll
  - mehrere Sperrgranulate
  - Verringerung der Anzahl der Sperranforderungen

- Probleme bei der Implementierung von Sperren
  - kleine Sperreinheiten (wünschenswert) erfordern hohen Aufwand
  - Sperranforderung und -freigabe sollten sehr schnell erfolgen, da sie sehr häufig benötigt werden

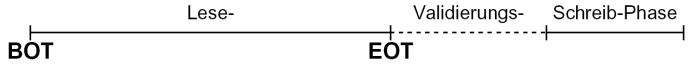
### Sperrverfahren in Datenbanksystemen 2/2

- Probleme bei der Implementierung von Sperren
  - explizite, satzweise Sperren führen u. U. zu umfangreichen Sperrtabellen und großem Zusatzaufwand
  - Zweiphasigkeit der Sperren führt häufig zu langen Wartezeiten (starke Serialisierung)
  - häufig berührte Zugriffspfade können zu Engpässen werden
  - Eigenschaften des Schemas können hot spots erzeugen
- Optimierungen
  - Änderungen auf privaten Objektkopien (verkürzte Dauer exklusiver Sperren)
  - Nutzung mehrerer Objektversionen
  - spezialisierte Sperren (Nutzung der Semantik von Änderungsoperationen)

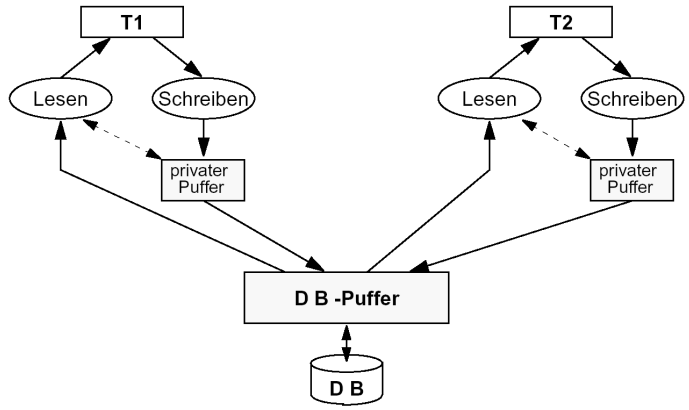
## 13.4 Optimistische Synchronisation

### Optimistische Synchronisation

- 3-phasige Verarbeitung:




- Lesephase
  - eigentliche TA-Verarbeitung
  - Änderungen einer Transaktion werden in privatem Puffer durchgeführt




### Optimistische Synchronisation

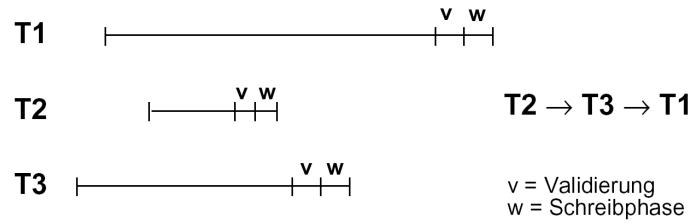
- Validierungsphase
  - Überprüfung, ob ein Lese-/Schreib- oder Schreib-/Schreib-Konflikt mit einer der parallel ablaufenden Transaktionen passiert ist
  - Konfliktauflösung durch Zurücksetzen von Transaktionen
- Schreibphase
  - nur bei positiver Validierung
  - Lese-Transaktion ist ohne Zusatzaufwand beendet
  - Schreib-Transaktion schreibt hinreichende Log-Information und propagiert ihre Änderungen

### Optimistische Synchronisation (OCC)

- Grundannahme: geringe Konfliktwahrscheinlichkeit
- Allgemeine Eigenschaften von OCC
  - \+ einfache TA-Rücksetzung
  - \+ keine Deadlocks
  - \+ potentiell höhere Parallelität als bei Sperrverfahren
  - – mehr Rücksetzungen als bei Sperrverfahren
  - – Gefahr des Verhungerns von TA
- Pro Transaktion wird zur Validierung
  - – Read-Set (RS) und
  - – Write-Set (WS) geführt
- Forderung
  - – TA kann nur erfolgreich validieren, wenn sie alle Änderungen von zuvor validierten TA gesehen hat ⇒ Validierungsreihenfolge bestimmt Serialisierungsreihenfolge
- Validierungsstrategien:
  - – Backward Oriented (BOCC): Validierung gegenüber bereits beendeten (Änderungs-) TA
  - – Forward Oriented (FOCC): Validierung gegenüber laufenden TA

### BOCC 1/2

- Validierung von Transaktion T
  - – BOCC-Test gegenüber allen Änderungs-TA $T_j$, die seit BOT von T erfolgreich validiert haben
  - – IF $RS(T) \cap WS(T_j) \neq \emptyset$ THEN ABORT T; ELSE SCHREIBPHASE

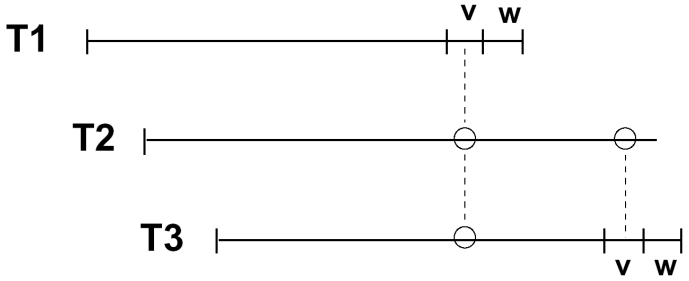



### BOCC 2/2

- Nachteile/Probleme:
  - – unnötige Rücksetzungen wegen ungenauer Konfliktanalyse
  - – Aufbewahren der Write-Sets beendeter TA erforderlich
  - – hohe Anzahl von Vergleichen bei Validierung
  - – Rücksetzung erst bei EOT ⇒ viel unnötige Arbeit
  - – es kann nur die validierende TA zurückgesetzt werden ⇒ Gefahr von starvation
  - – hohes Rücksetzrisiko für lange TA und bei Hot-Spots

## FOCC 1/2

- nur Änderungs-TA validieren gegenüber laufenden TA $T_i$
- Validierungstest: $WS(T) \cap RS(T_j) \neq \emptyset$



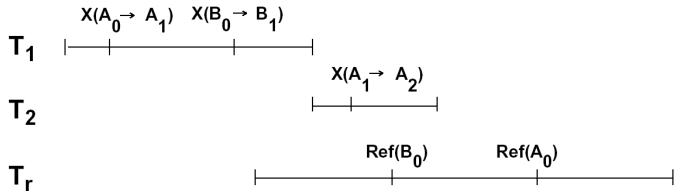

- Vorteile:
  - Wahlmöglichkeit des Opfers (Kill, Abort, Prioritäten, ...)
  - keine unnötigen Rücksetzungen
  - frühzeitige Rücksetzung möglich ⇒ Einsparen unnötiger Arbeit
  - keine Aufbewahrung von Write-Sets, geringerer Validierungsaufwand als bei BOCC

## FOCC 2/2

- Probleme:
  - Während Validierungs- und Schreibphase muß $WS(T)$ gesperrt sein, damit sich die $RS(T_i)$ nicht ändern (keine Deadlocks damit möglich)
  - immer noch hohe Rücksetzrate möglich
  - es kann immer nur einer TA Durchkommen definitiv zugesichert werden

### Mehrversionen-Konzept

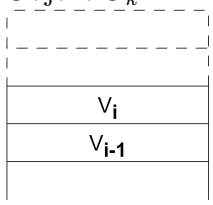
- jede Änderung erzeugt neue Objektversion
- Lese-TA sehen den bei ihrem BOT gültigen DB-Zustand ⇒ werden bei Synchronisation nicht mehr berücksichtigt
- keine Blockierungen und Rücksetzungen für Lese-TA, dafür ggf. Zugriff auf veraltete Objektversionen




- Änderungs-TA werden untereinander über ein allgemeines Verfahren (Sperren, OCC, ...) synchronisiert
  - ⇒ weniger Synchronisationskonflikte
- zusätzlicher Speicher- und Wartungsaufwand
  - Versionenpoolverwaltung
  - Auffinden von Versionen
  - Garbage Collection

### Zugriff auf Objektversionen 1/2

- Objekt $O_k$

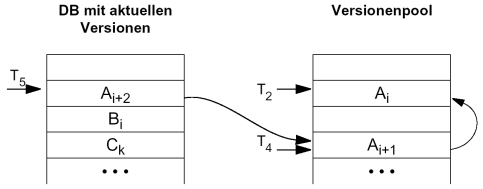



- zeitliche Reihenfolge der Zugriffe auf $O_k$

| | | |
|---|---|---|
| $T_j$ (BOT) | ➥ | $V_i$ (aktuelle Version) |
| $T_m$(X) | ➥ | Erzeugen $V_{i+1}$ |
| $T_n$(X) | ➥ | Verzögern bis $T_m$(EOT) |
| $T_m$(EOT) | ➥ | Freigeben $V_{i+1}$ |
| $T_n$(X) | ➥ | Erzeugen $V_{i+2}$ |
| $T_j$ (Ref) | ➥ | $V_i$ |
| $T_n$(EOT) | ➥ | Freigeben $V_{i+2}$ |

### Zugriff auf Objektversionen 2/2

- Speicherungsschema für Versionen




- Versionenpool: Teil des DBS-Puffers
- Speicherplatzoptimierung: Versionen auf Satzebene, Einsatz von Komprimierungstechniken

### Zeitstempel-Verfahren 1/4

- Grundsätzliche Idee
  - TA bekommt bei BOT systemweit eindeutigen Zeitstempel
  - TA hinterläßt Zeitstempels (als Lese- oder Schreibstempel RTS bzw. WTS) bei jedem Objekt $O_i$, auf das sie zugreift
  - Prüfung der Serialisierbarkeit ist sehr einfach (Zeitstempelvergleich)
- Prinzipielle Arbeitsweise
  - Vergabe von eindeutigen TA-IDs (Zeitstempel ts der TA) in aufsteigender Reihenfolge
  - Zeitstempel des Objektes $O$: $TS(O)$
  - Zugriffe von $T_i$ auf $O$ (r/w): $TS(O) := ts(T_i)$
  - Konfliktprüfung: if $TS(T_i) < TS(O)$ then ABORT; else verarbeite;
  - Zugriffsfolge auf Objekt O:


$\Rightarrow$ kein Konflikt bei $r_9$!

## Zeitstempel-Verfahren 2/4

- Verfeinerung des Zeitstempelverfahrens
  - 2 Zeitstempel pro Objekt
    1. Erhöhung beim Schreiben: WTS
    2. Erhöhung beim Lesen: RTS
  - Regeln für $T_i$ und $O$: (Abk. $ts(T_i) = i$)

R1: $r_i \wedge (i \geq WTS(O)) \Rightarrow$ if RTS(O) < i then RTS(O) := i; Lesen

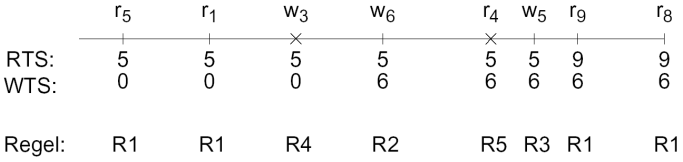
R2: $w_i \wedge (i \geq RTS(O)) \wedge (i \geq WTS(O)) \Rightarrow$ WTS(O) := i; Ändern

R3: $w_i \wedge (i \geq RTS(O)) \wedge (i < WTS(O)) \Rightarrow$ kein Konflikt *(blind update)* – keine Schreibaktion – weiter

R4: $w_i \wedge (i < RTS(O)) \Rightarrow$ Zurücksetzen

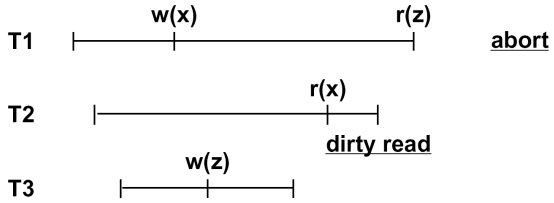
R5: $r_i \wedge (i < WTS(O)) \Rightarrow$ Zurücksetzen

  - Zugriffsfolge auf Objekt O:




## Zeitstempel-Verfahren 3/4

- TA $T$ wird zurückgesetzt, falls $ts(T) < WTS$
  - bei Lesezugriff
  - und bei Schreibzugriff gilt (ohne Berücksichtigung von blind updates)




- Vorkehrungen für den ABORT-Fall
  - sofortige Zulassung aller Schreiboperationen erzeugt inkonsistente DB
  - Einfrieren der Zeitstempel bis COMMIT der ändernden TA
  - Erwerb von Anwartschaften: Prewrites
  - Prewrite $i$ verzögert $r_j, w_j$ mit $j > i$
  - Einführung von Read-queues, Prewrite-queues und Write-queues

## Zeitstempel-Verfahren 4/4

- Eigenschaften
  - Serialisierungsreihenfolge einer Transaktion wird bei BOT festgelegt
  - Deadlocks sind ausgeschlossen

  - aber: (viel) höhere Rücksetzraten als pessimistische Verfahren
  - ungelöste Probleme, z. B. wiederholter ABORT einer Transaktion
- Hauptsächlicher Einsatz
  - Synchronisation in Verteilten DBS
  - lokale Prüfung der Serialisierbarkeit direkt am Objekt $O_i$ (geringer Kommunikationsaufwand)

## 13.5 Zusammenfassung: Synchronisation

### Zusammenfassung

- Korrektheitskriterium der Synchronisation: Serialisierbarkeit
- Sperrverfahren sind universell einsetzbar
  - Zweiphasen-Sperrprotokolle
  - reine OCC- und Zeitstempelverfahren erzeugen zuviele Rücksetzungen
- Hierarchische Synchronisationsverfahren
  - erlauben Begrenzung des Verwaltungsaufwands
- generelle Optimierungen:
  - reduzierte Konsistenzebene
  - Mehrversionen-Ansatz
- Harte Synchronisationsprobleme:
  1. Hot Spots / High Traffic-Elemente
  2. lange (Änderung-) TA

# 14 Logging und Recovery

## 14.1 Einführung Logging und Recovery

### DB-Recovery

- „A recoverable action is 30% harder and requires 20% more code than a non-recoverable action“ (J. Gray)
- Aufgabe des DBVS: Automatische Behandlung aller erwarteten Fehler
- Fehlermodell von zentralisierten DBVS
  - Transaktionsfehler
  - Systemfehler
  - Gerätefehler
- Probleme
  - Fehlererkennung
  - Fehlereingrenzung
  - Abschätzung des Schadens
  - Durchführung der Recovery
- Annahmen: (Unter welchen Voraussetzungen funktioniert die Wiederherstellung der Daten?)
  - quasi-stabiler Speicher
  - fehlerfreier DBS-Code
  - fehlerfreie Log-Daten
  - Durchführung der Wiederherstellung

### DB-Recovery

- Voraussetzung: Sammeln redundanter Informationen während des normalen Betriebes (Logging)
- Transaktionsparadigma verlangt:
  - Alles-oder-Nichts-Eigenschaft von Transaktionen
  - Dauerhaftigkeit erfolgreicher Änderungen
- Zielzustand nach erfolgreicher Recovery:
  - Durch die Recovery-Aktionen ist der jüngste Zustand vor Erkennen des Fehlers wiederherzustellen, der allen semantischen Integritätsbedingungen entspricht, der also ein möglichst aktuelles, exaktes Bild der Miniwelt darstellt
  - $\Rightarrow$ jüngster transaktionskonsistenter DB-Zustand
- Forward-Recovery i.a. nicht anwendbar
  - Fehlerursache häufig falsche Programme, Eingabefehler u. ä.
  - durch Fehler unterbrochene Transaktionen sind zurückzusetzen (Backward Recovery)
- Backward-Recovery
  - setzt voraus, daß auf allen Abstraktionsebenen genau definiert ist, auf welchen Zustand die DB im Fehlerfall zurückzusetzen ist.

## Fehlerarten

**Fehlerarten**

| Auswirkung eines Fehlers auf | Fehlertyp | Fehlerklassifikation |
|---|---|---|
| eine Transaktion | - Verletzung von Systemrestriktionen<br>• Verstoß gegen Sicherheitsbestimmungen<br>• übermäßige Betriebsmittelanforderungen<br>- anwendungsbedingte Fehler<br>• z. B. falsche Operationen und Werte | Transaktionsfehler |
| mehrere Transaktionen | - geplante Systemschließung<br>- Schwierigkeiten bei der Betriebsmittelvergabe<br>• Überlast des Systems<br>• Verklemmung mehrerer Transaktionen | |
| alle Transaktionen (das gesamte Systemverhalten) | - Systemzusammenbruch mit Verlust der Hauptspeicherinhalte<br>• Hardware-Fehler<br>• falsche Werte in kritischen Tabellen | Systemfehler |
| | - Zerstörung von Sekundärspeichern | Gerätefehler |
| | - Zerstörung des Rechenzentrums | Katastrophen |

## Recovery-Arten 1/2

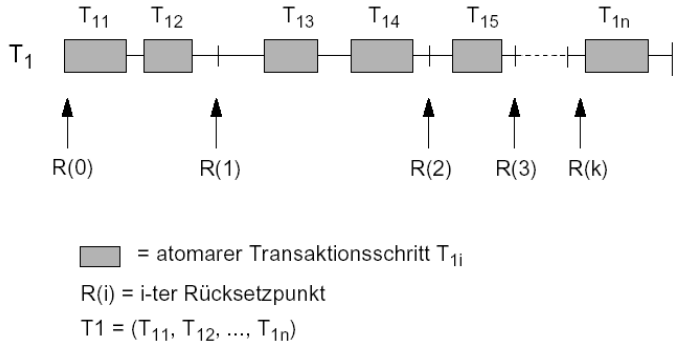
1. Zurücksetzen einzelner Transaktionen im laufenden Betrieb (Transaktionsfehler, Deadlock, etc.)
   - vollständiges Zurücksetzen auf Transaktionsbeginn (TA-UNDO) bzw.
   - partielles Zurücksetzen auf Rücksetzpunkt (Savepoint) innerhalb der Transaktion
2. Crash-Recovery nach Systemfehler Wiederherstellen des jüngsten transaktionskonsistenten DB-Zustandes:
   - (partielles) REDO für erfolgreiche Transaktionen (Wiederholung verlorengegangener Änderungen)
   - UNDO aller durch Ausfall unterbrochenen Transaktionen (Entfernen der Änderungen aus der materialisierten DB)
3. Platten-Recovery nach Gerätefehler
   - Spiegelplatten bzw.
   - vollständiges Wiederholen (REDO) aller Änderungen auf einer Archivkopie

## Recovery-Arten 2/2

4. Katastrophen-Recovery
   - Betrieb eines entfernten zweiten Rechenzentrums
   - redundante Sammlung der DB-Daten und Log-Daten

## Partielles Zurücksetzen von Transaktionen 1/2

- Voraussetzung: transaktionsinterne Rücksetzpunkte (Savepoints)

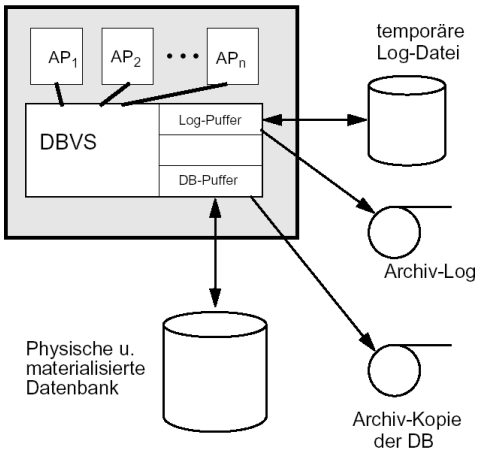



### Partielles Zurücksetzen von Transaktionen 2/2

- Zusätzliche Operationen: SAVE R(i) und RESTORE R(j)
  - Protokollierung aller Änderungen, Sperren, Cursor-Positionen etc. notwendig
  - Partielle UNDO-Operation bis R(j) in LIFO-Reihenfolge
  - Rücksetzpunkte müssen vom DBS sowie vom Laufzeitsystem der Programmiersprache unterstützt werden
  - Derzeitige Implementierungen bieten keine Unterstützung von persistenten Savepoints !

  ⇒ Nach Systemfehler wird Transaktion vollständig zurückgesetzt

### DB-Recovery – Systemkomponenten 1/2




- Daten werden in die physische DB geschrieben und in die materialisierte DB eingebracht
- Pufferung von Log-Daten im Hauptspeicher (Log-Puffer)

### DB-Recovery – Systemkomponenten 2/2

1. Temporäre Log-Datei für Kurzzeit-Recovery:
   - Behandlung von Transaktionsfehlern
   - Behandlung von Systemfehlern

   DB + temp. Log ⇒ DB
2. Behandlung von Gerätefehlern (Langzeit-Recovery):
   - Archiv-Kopie + Archiv-Log ⇒ DB

## 14.2 Logging-Strategien

### Logging-Techniken 1/2

- Logging: Sammlung redundanter Daten bei Änderungen im Normalbetrieb ⇒ Voraussetzung für Recovery
- Physisches Logging
  - Die alten Zustände (Before-Images) und neuen Zustände (After-Images) geänderter Objekte werden auf die Protokolldatei geschrieben
  - Log-Granulat: Seite vs. Eintrag/Satz
  - Zustands- und Übergangs-Logging anwendbar
  - Physisches Logging ist bei direkten und verzögerten Einbringstrategien anwendbar
  - Die meisten DBS verwenden physisches Logging

### Logging-Techniken 2/2

- Logisches Logging
  - Protokollierung der ändernden DML-Befehle mit ihren Parametern
  - Voraussetzung: nach einem Systemausfall müssen auf der materialisierten DB DML-Operationen ausführbar sein, d. h., sie muß wenigstens operationskonsistent sein (bzgl. der verwendeten Operationen !)
    ⇒ verzögertes Einbringen von Änderungen erforderlich
  - UNDO-Probleme v.a. bei nicht-relationalen Systemen
    - z. B. Löschen einer Hierarchie von Set-Ausprägungen (ERASE ALL)
- Generelles Problem
  - mengenorientierte Aktualisierungsoperationen
    - z. B. DELETE <relation>

### Logging: Anwendungsbeispiel

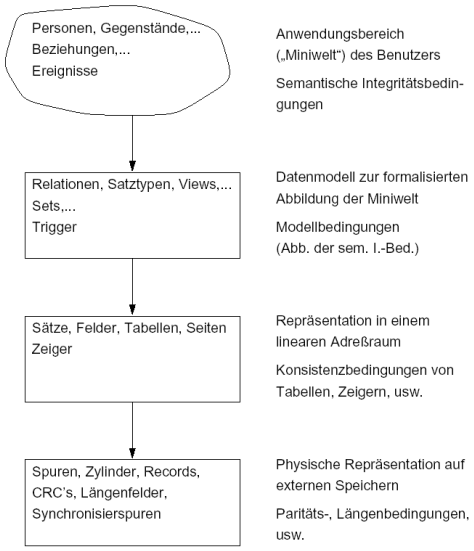
- Änderungen bezüglich einer Seite A:
  1. Ein Objekt a wird in Seite A eingefügt
  2. In A wird ein bestehendes Objekt $b_{alt}$ nach $b_{neu}$ geändert
- Zustandsübergänge von A: $A_1 \xrightarrow{1.} A_2 \xrightarrow{2.} A_3$

| | *logisch* | *physisch* |
|---|---|---|
| *Zustände* | | Protokollierung der Before- und After-Images<br>1. $A_1$ und $A_2$<br>2. $A_2$ und $A_3$ |
| *Übergänge* | Protokollierung der Operationen mit Parameter<br>1. Insert (a)<br>2. Update ($b_{alt}$,$b_{neu}$) | Differenzen-Logging<br>1. $A_1 \oplus A_2$<br>2. $A_2 \oplus A_3$ |

Rekonstruktion von Seiten beim Differenzen-Logging: $A_1$ als Anfangs- oder $A_3$ als Endzustand seien verfügbar

| | |
|---|---|
| $A_1 \oplus (A_1 \oplus A_2) = A_2$ | $A_3 \oplus (A_2 \oplus A_3) = A_2$ |
| $A_2 \oplus (A_2 \oplus A_3) = A_3$ | $A_2 \oplus (A_1 \oplus A_2) = A_1$ |
| REDO-Recovery | UNDO-Recovery |

## Abstraktionsebenen und Logging




- Logging kann auf jeder Ebene erfolgen:
- Wie kann ebenenspezifische Konsistenz (Operationskonsistenz) im Fehlerfall garantiert werden?

## Bewertung der Logging-Strategien

| | Logging-Aufwand | Restart-Aufwand |
|---|---|---|
| Seitenzustands-Logging | – – | + |
| Seitenübergangs-Logging | – | + |
| Eintrags-Logging | + | + |
| logisches Logging | ++ | – |

– – sehr hoch
– hoch
+ gering
++ sehr gering

- Vorteile von Eintrags-Logging gegenüber Seiten-Logging:
  - geringerer Platzbedarf
  - weniger Log-E/As
  - erlaubt bessere Pufferung von Log-Daten (Gruppen-Commit)
  - unterstützt feine Synchronisationsgranulate (Seiten-Logging ==> Synchronisation auf Seitenebene)

  ⇒ jedoch: Recovery komplexer als mit Seiten-Logging

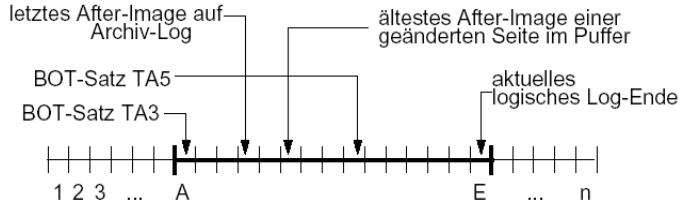
## Aufbau der (temporären) Log-Datei 1/2

- i. a. sequentielle Datei, Schreiben neuer Protokolldaten an das aktuelle Dateiende
- übliche Satzarten
  - Begin-of-Transaction (BOT
  - UNDO-Informationen (z. B. Before-Images)
  - REDO-Informationen (z .B. After-Images)

  - Commit-Satz
  - Abort-Satz
  - Checkpoint-Sätze
- Log-Sätze einer Transaktion werden rückwärts verkettet (für Transaktions-UNDO)

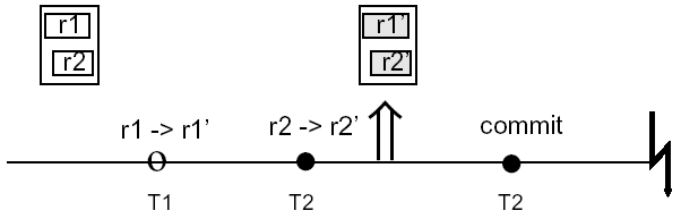
### Aufbau der (temporären) Log-Datei 2/2

- Log-Daten sind für Crash-Recovery nur begrenzte Zeit relevant:
  - UNDO-Sätze für erfolgreich beendete Transaktionen werden nicht mehr benötigt
  - nach Einbringen der Seite in die DB wird REDO-Information nicht mehr benötigt
- Ringpufferorganisation der Log-Datei




### Abhängigkeiten zu anderen Systemkomponenten 1/2

1. Sperrverwaltung
   - Log-Granulat muß kleiner oder gleich dem Sperrgranulat sein !
   - Beispiel:
     - Sperren auf Satzebene,
     - Before- bzw. After-Images auf Seitenebene
     - UNDO (REDO) einer Änderung kann parallel durchgeführte Änderungen derselben Seite überschreiben (lost update)




### Abhängigkeiten zu anderen Systemkomponenten 2/2

1. Einbringstrategie für Änderungen
   - direkt (¬ ATOMIC, Update-in-Place)
   - verzögert (ATOMIC, Bsp.: Schattenspeicherkonzept)
2. DB-Pufferverwaltung
   - Verdrängen schmutziger Seiten
     - STEAL vs. ¬STEAL
   - Ausschreibstrategie für geänderte Seiten
     - FORCE vs. ¬FORCE

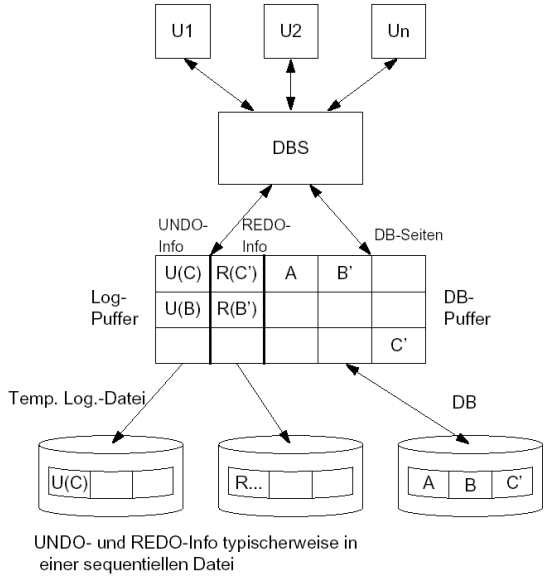
## 14.3 Recovery Verfahren

### Einbringstrategien

- Direkt (Update-in-Place)
  - geänderte Seite wird immer in denselben Block auf Platte zurückgeschrieben
  - atomares Zurückschreiben mehrerer geänderter Seiten nicht möglich (¬ATOMIC)
- Verzögert (ATOMIC)
  - z. B. Schattenspeicherkonzept (System R, SQL/DS)
  - geänderte Seite wird in separaten Block auf Platte geschrieben
  - Seitentabelle gibt aktuelle Adresse einer Seite an
  - verzögertes, atomares Einbringen mehrerer Änderungen durch Umschalten von Seitentabellen möglich
  - ⇒ operations- oder transaktionskonsistente DB auf Platte (logisches Logging anwendbar)
- Schwerwiegende Nachteile:
  - aufwendiges Einbringen
  - Seitentabelle kann für große DB nicht mehr im Hauptspeicher gehalten werden
  - Clustereigenschaften werden zerstört
  - Speicherplatzbedarf

### Direktes Einbringen 1/2

- ¬ATOMIC (Update-in-Place)

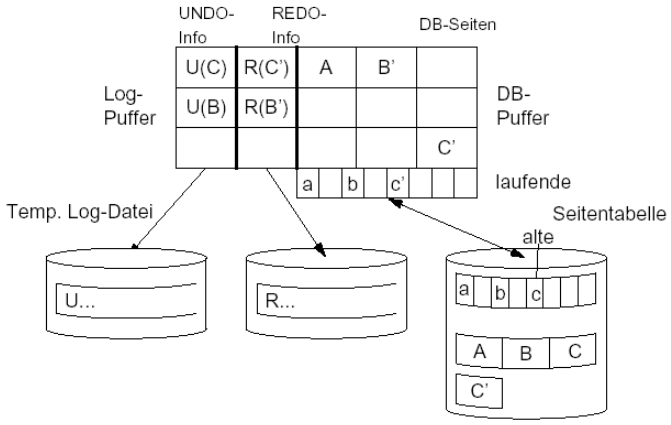



### Direktes Einbringen 2/2

- Es sind 2 Prinzipien einzuhalten (Minimalforderung)
  1. WAL-Prinzip: Write Ahead Log für UNDO-Info
     - U(B) vor B'
  2. Ausschreiben der REDO-Info spätestens bei COMMIT
     - R(C') + R(B') vor COMMIT

### Verzögertes Einbringen

- ATOMIC




1. WAL-Prinzip bei verzögertem Einbringen
   - Transaktionsbezogene UNDO-Info ist vor Sicherungspunkt zu schreiben
   - U(C) + U(B) vor Sicherungspunkt
2. Ausschreiben der REDO-Info spätestens bei COMMIT
   - R(C') + R(B') vor COMMIT

### Abhängigkeiten zur Ersetzungsstrategie

- Problem: Ersetzung schmutziger Seiten
- STEAL:
  - geänderte Seiten können jederzeit, insbesondere vor EOT der ändernden Transaktion, ersetzt und in die materialisierte DB eingebracht werden
    - \+ große Flexibilität zur Seitenersetzung
    - – UNDO-Recovery vorzusehen (Transaktions-Abbruch, Systemfehler)

  ⇒ STEAL erfordert Einhaltung des Write-Ahead-Log (WAL)-Prinzip: vor dem Einbringen einer schmutzigen Änderung müssen zugehörige UNDO-Informationen (z. B. Before-Images) in die Log-Datei geschrieben werden
- NOSTEAL (¬STEAL):
  - Seiten mit schmutzigen Änderungen dürfen nicht ersetzt werden
  - keine UNDO-Recovery auf der materialisierten DB vorzusehen
  - Probleme bei langen Änderungstransaktionen

### Abhängigkeiten zur Ausschreibstrategie (EOT-Behandlung)

- FORCE:
  alle geänderten Seiten werden spätestens beim EOT (bei Commit) in die materialisierte DB eingebracht (Durchschreiben)
  - \+ keine REDO-Recovery nach Rechnerausfall
  - – hoher Schreibaufwand
  - – große DB-Puffer werden schlecht genutzt

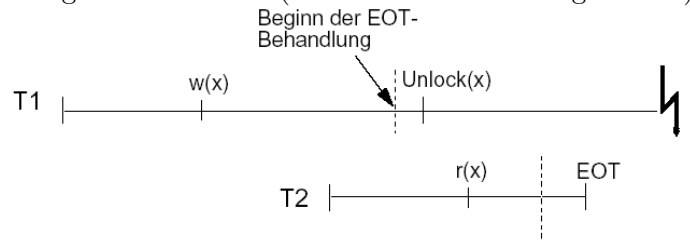
  - Antwortzeitverlängerung für Änderungstransaktionen

- NOFORCE (¬FORCE):
  + kein Durchschreiben der Änderungen bei EOT
  + beim Commit werden lediglich REDO-Informationen in die Log-Datei geschrieben
  - REDO-Recovery nach Rechnerausfall

- Commit-Regel
  - bevor das Commit einer Transaktion ausgeführt werden kann, sind für ihre Änderungen ausreichende REDO-Informationen (z. B. After-Images) zu sichern

### Commit-Behandlung 1/2

- Sichern der Änderungen einer Transaktion bei Commit
- andere Transaktionen dürfen Änderungen erst sehen, wenn Durchkommen der ändernden Transaktion gewährleistet ist (Problem der Cascading Aborts)




### Commit-Behandlung 2/2

- Zweiphasige Commit-Bearbeitung
  1. Wiederholbarkeit der Transaktion sichern
     - ggf. Änderungen noch sichern
     - Commit-Satz auf Log schreiben
  2. Änderungen sichtbarmachen (Freigabe der Sperren)

  Benutzer kann nach Phase 1 vom erfolgreichen Ende der Transaktion informiert werden (Ausgabenachricht)

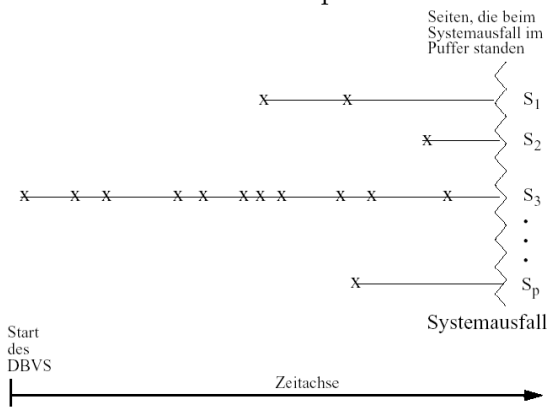
- Bsp.: Commit-Behandlung bei FORCE, STEAL:
  1. Before-Images auf Log schreiben
  2. FORCE der geänderten DB-Seiten
  3. After-Images und Commit-Satz schreiben

  bei NOFORCE lediglich 3.) für erste Commit-Phase notwendig

### Sicherungspunkte (Checkpoints) 1/2

- Sicherungspunkt
  - Maßnahme zur Begrenzung des REDO-Aufwandes nach Systemfehlern (NOFORCE)
- ohne Sicherungspunkte müßten potentiell alle Änderungen seit Start des DBVS wiederholt werden

- besonders kritisch: Hot-Spot-Seiten




### Sicherungspunkte (Checkpoints) 2/2

- Log-Datei
  - BEGIN_CHKPT-Satz
  - Checkpoint-Informationen
  - END_CHKPT-Satz
- Log-Adresse des letzten Checkpoint-Satzes wird in spezieller Restart-Datei geführt (DB2: Bootstrap Data Set)

### Arten von Sicherungspunkten 1/2

- Direkte Sicherungspunkte
  - alle geänderten Seiten im DB-Puffer werden in die materialisierte DB eingebracht
  - REDO-Recovery beginnt bei letztem Checkpoint
  - Nachteil: lange Totzeit des Systems, da während des Sicherungspunktes keine Änderungen durchgeführt werden können
  - Problem wird durch große Hauptspeicher verstärkt
  - Transaktionskonsistente oder operationskonsistente Sicherungspunkte

  ⇒ FORCE kann als spezieller Checkpoint-Typ aufgefaßt werden (nur Seiten einer Transaktion werden ausgeschrieben ⇒ transaktionsorientiert)

### Arten von Sicherungspunkten 2/2

- Indirekte/Unscharfe Sicherungspunkte (Fuzzy Checkpoints)
  - kein Hinauszwingen geänderter Seiten
  - nur Statusinformationen (Pufferbelegung, Menge aktiver Transaktionen, offene Dateien etc.) werden in Log schreiben
  - sehr geringer Checkpoint-Aufwand
  - i. a. REDO-Informationen vor letztem Sicherungspunkt noch zu berücksichtigen

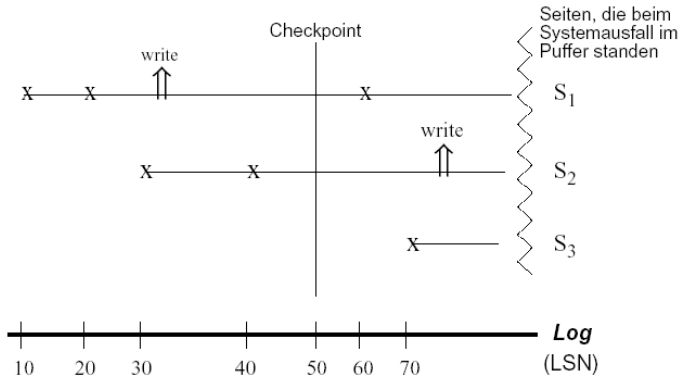
  ⇒ Sonderbehandlung von Hot-Spot-Seiten

## Fuzzy Checkpoints 1/2

- DB auf Platte bleibt fuzzy, nicht operationskonsistent
  - nur bei Update-in-Place (¬ATOMIC) relevant
- Problem: Bestimmung der Log-Position, an der REDO-Recovery beginnen muß
  - bei Änderung einer Seite im Puffer wird ein Log-Satz erzeugt
  - Pufferverwalter vermerkt sich zu jeder geänderten Seite START-LSN, d. h. Adresse des Log-Satzes (LSN: Log Sequence Number) der ersten Änderung seit Einlesen von Platte
  - REDO-Recovery nach Rechnerausfall beginnt bei MIN (START-LSN)

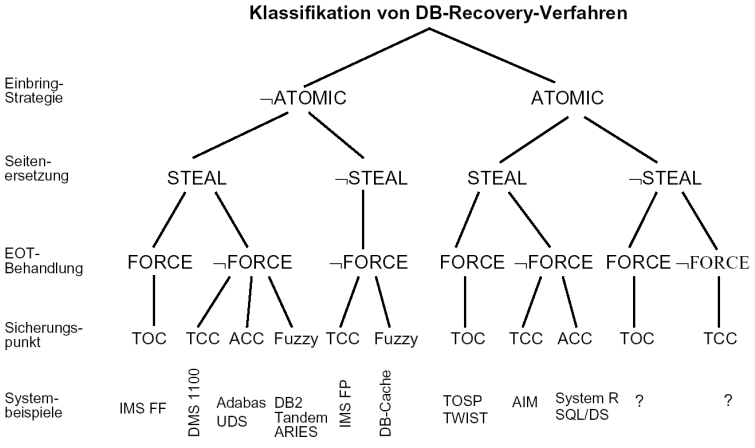
## Fuzzy Checkpoints 2/2

- Startposition wird in Checkpoint-Information vermerkt (daneben laufende Transaktionen, geänderte Seiten, ...)




- geänderte Seiten werden asynchron ausgeschrieben
  - ggf. Kopie der Seite anlegen (für Hot-Spot-Seiten)
  - Seite ausschreiben
  - START-LSN anpassen / zurücksetzen

## Klassifikation von DB-Recovery-Verfahren




## Zwischentest zur Fehlerbehandlung

**Zwischentest zur Fehlerbehandlung**

| Situation im Fehlerfall (Crash) | Datenseite bereits in die Datenbank eingebracht | Log-Satz bereits in die Log-Datei geschrieben | Transaktion | |
|---|---|---|---|---|
| | | | nicht beendet ggf. Zurücksetzung | abgeschlossen ggf. Wiederholung |
| 1. | Nein | Nein | | |
| 2. | Nein | Ja | | |
| 3. | Ja | Nein | | |
| 4. | Ja | Ja | | |

- Mögliche Antworten:
  1. Tue überhaupt nichts
  2. Benutze die UNDO-Information und setze zurück
  3. Benutze die REDO-Information und wiederhole
  4. Situation ist logisch unmöglich
  5. WAL-Prinzip verhindert diese Situation
  6. Zwei-Phasen-Commit-Protokoll verhindert diese Situation

## 14.4 Crash-Recovery

### Crash-Recovery

- Ziel: jüngster transaktionskonsistenter DB-Zustand aus
  - materialisierter DB
  - temporärer Log-Datei
- Update-in-Place (¬ATOMIC):
  - Zustand der materialisierten DB nach Crash unvorhersehbar ⇒ nur physische Logging-Verfahren anwendbar
  - ein Block der materialisierten DB ist entweder
    1. aktuell oder
    2. veraltet (NOFORCE) ⇒ REDO oder
    3. schmutzig (STEAL) ⇒ UNDO
- bei ATOMIC:
  - materialisierte DB entspricht Zustand des letzten Einbringens
  - operationskonsistent ⇒ DML-Befehle ausführbar
  - FORCE: kein REDO
  - NOFORCE:
    1. transaktionskonsistentes Einbringen ⇒ REDO, jedoch kein UNDO
    2. operationskonsistentes Einbringen ⇒ UNDO + REDO

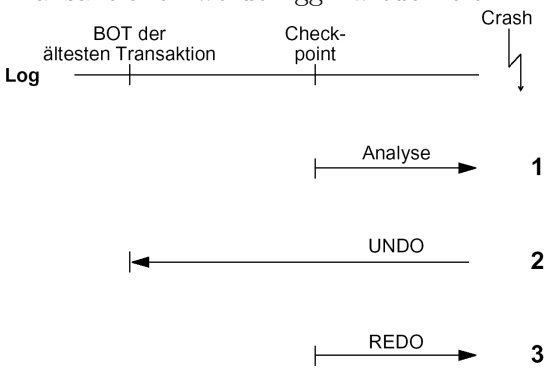
### Allgemeine Restart-Prozedur 1/2

- (¬ATOMIC, STEAL,¬FORCE, CHECKPOINT)
  Temporäre Log-Datei wird 3-mal gelesen
  1. Analyse-Lauf (vom letzten Checkpoint bis zum Log-Ende)
     - Bestimmung von Gewinner- und Verlierer-Transaktionen sowie der Seiten, die von ihnen geändert wurden

2. UNDO-Lauf
   - Rücksetzen der Verlierer-Transaktionen durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-Transaktion

### Allgemeine Restart-Prozedur 2/2

3. REDO-Lauf
   - Vorwärtslesen des Logs (Startpunkt abhängig vom Checkpoint-Typ); Änderungen der Gewinner-Transaktionen werden ggf. wiederholt




     * für Schritt 2 und 3 sind betroffene DB-Seiten einzulesen
     * LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
     * am Ende sind alle geänderten Seiten wieder in die DB einzubringen

### Crash-Recovery

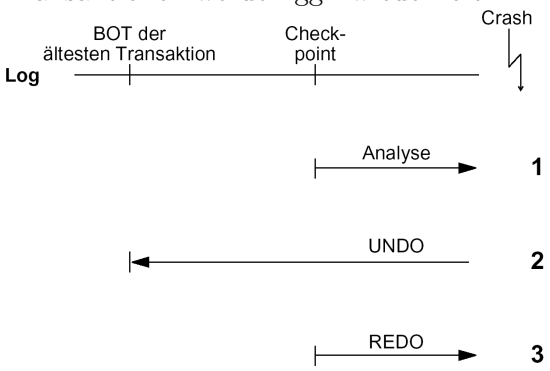
- Ziel: jüngster transaktionskonsistenter DB-Zustand aus
  - materialisierter DB
  - temporärer Log-Datei
- Update-in-Place (¬ATOMIC):
  - Zustand der materialisierten DB nach Crash unvorhersehbar ⇒ nur physische Logging-Verfahren anwendbar
  - ein Block der materialisierten DB ist entweder
    1. aktuell oder
    2. veraltet (NOFORCE) ⇒ REDO oder
    3. schmutzig (STEAL) ⇒ UNDO
- bei ATOMIC:
  - materialisierte DB entspricht Zustand des letzten Einbringens
  - operationskonsistent ⇒ DML-Befehle ausführbar
  - FORCE: kein REDO
  - NOFORCE:
    1. transaktionskonsistentes Einbringen ⇒ REDO, jedoch kein UNDO
    2. operationskonsistentes Einbringen ⇒ UNDO + REDO

### Allgemeine Restart-Prozedur 1/2

- (¬ATOMIC, STEAL,¬FORCE, CHECKPOINT)
  Temporäre Log-Datei wird 3-mal gelesen
  1. Analyse-Lauf (vom letzten Checkpoint bis zum Log-Ende)
     - Bestimmung von Gewinner- und Verlierer-Transaktionen sowie der Seiten, die von ihnen geändert wurden
  2. UNDO-Lauf
     - Rücksetzen der Verlierer-Transaktionen durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-Transaktion

### Allgemeine Restart-Prozedur 2/2

3. REDO-Lauf
   - Vorwärtslesen des Logs (Startpunkt abhängig vom Checkpoint-Typ); Änderungen der Gewinner-Transaktionen werden ggf. wiederholt




   * für Schritt 2 und 3 sind betroffene DB-Seiten einzulesen
   * LSN der Seiten zeigen, ob Log-Informationen anzuwenden sind
   * am Ende sind alle geänderten Seiten wieder in die DB einzubringen

## 14.5 Zusammenfassung Logging und Recovery

### Zusammenfassung 1/2

- Fehlerarten
  - Transaktions-, System-, Gerätefehler und Katastrophen
- breites Spektrum von Logging- und Recovery-Verfahren
  - Logging kann auf verschiedenen Systemebenen angesiedelt werden
  - erfordert ebenenspezifische Konsistenz im Fehlerfall
- Atomic-Verfahren
  - erhalten den DB-Zustand des letzten Checkpoint
  - gewährleisten demnach die gewählte Operationskonsistenz auch bei der Recovery von einem Crash und
  - erlauben folglich logisches Logging
- Update-in-Place-Verfahren

  - sind i. allg. ATOMIC-Strategien vorzuziehen, weil sie im Normalbetrieb wesentlich billiger sind und
  - nur eine geringe Crash-Wahrscheinlichkeit zu unterstellen ist
  - Sie erfordern jedoch physisches Logging

- Grundprinzipien bei Update-in-Place
  1. WAL-Prinzip: Write Ahead Log für UNDO-Info
  2. REDO-Info ist spätestens bei COMMIT zu schreiben

### Zusammenfassung 2/2

- Grundprinzipien bei ATOMIC
  1. WAL-Prinzip bei verzögertem Einbringen
     - Transaktionsbezogene UNDO-Info ist vor Checkpoint zu schreiben
  2. REDO-Info ist spätestens bei COMMIT auf die Log-Datei zu schreiben
- Eintrags-Logging ist Seiten-Logging überlegen
  - geringerer Platzbedarf, weniger E/As
- NOFORCE-Strategien
  - sind FORCE-Verfahren vorzuziehen
  - erfordern den Einsatz von Checkpoint-Maßnahmen zur Begrenzung des Redo-Aufwandes:

  $\Rightarrow$ Fuzzy Checkpoints erzeugen den geringsten Overhead im Normalbetrieb
- STEAL-Methoden
  - verlangen die Einhaltung des WAL-Prinzips
  - erfordern Undo-Aktionen nach einem Rechnerausfall
- Synchronisationsgranulat muß größer oder gleich dem Log-Granulat sein