Grundbegriffe:

Datenabstraktion /-unabhängigkeit

- Persistente (dauerhafte) Speicherung
- Speicherung ohne Detailkenntnis
- Wiedergewinnung: Auffinden und Aushändigen von Daten

Schichtenmodell

- Zur Strukturierung von Software-Systeme
- Schicht: realisiert einen Dienst und stellt diesen per Schnittstelle zur Verfügung. Verwendet werden darunterliegende (verborgene) Schichten

Vorteile:

- Einfachere Benutzung durch höhere Schichten (Abstraktion durch Hierarchie, Änderungen höherer Schichten beeinflussen tiefere nicht)
- Solange die Schnittstellen gleich bleiben, können Schichten einzeln verändert werden

Wann eine Datenbank?

- Großes Datenvolumen
- Wiederverwendbare Daten
- Gleichzeitig, einfach und schnell Nutzbar
- Strukturiert, Redundanzfrei, Sicher

Bsp. für Schichten ohne Datenbank

- Linux Kernel / Firefox / große Softwaresysteme

Dateiverwaltung

Physische Speichergeräte

Magnetplattenspeicher:

- HDD, wahlfreier Zugriff

Magnetbandspeicher:

- Sequentieller Zugriff, zur Archivierung

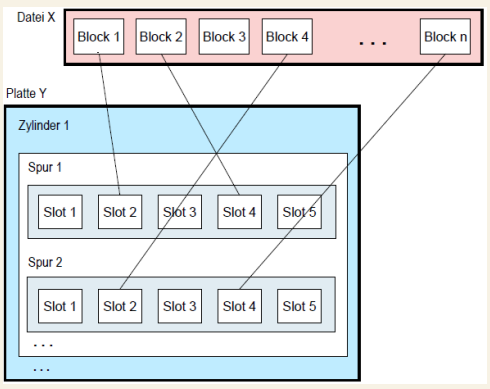
Alternativen:

- Optische Speicher (DVD, SSD)
- Hauptspeicher (z.b. DDR3-RAM)

Logische Speichergeräte

- Erhöhung der Störsicherheit
- Vereinfachung durch Abstraktion
- Speicher zusammenfassen (Raid)

Dateikonzept



Dateikatalog

- Verwaltungsdatenstruktur
- Verzeichnis aller Dateien
- Liegt auf Platte an fester Position
- Liefert unbenutzte Blöcke auf der Platte

Geräteunabhängigkeit

Blockorientierter Dateizugriff ist unabhängig vom Speichergerät. Somit langsamer! Aber sicherer und Programm stabil (keine Programmänderung bei Speicheränderung)!

HDD Berechnungen

- 104830 Zylinder je 4 Spuren
- 1306 Blöcke je 512 Byte pro Spur
- Umderhungszeit UT: 4ms
- Positionierung (Seek-Zeit): 3,4ms
- Berechnungen nur für Mittelwerte

Mittlere Zugriffszeit von random Block:

$$ZZ = SeekZeit + 0,5 \cdot UZ = 5,4ms$$

1000 aufeinanderfolgende Blöcke lesen:
Verteilt auf eine Spur:

$$ZZ + 1000/1304 \cdot UZ = 5,4ms + 3ms = 8,4ms$$

Verteilt auf zwei Spuren:

$$ZZ + \frac{500}{1304} \cdot UZ + 0.5 \cdot UZ + \frac{500}{1304} \cdot UZ = 10,4ms$$

1000 random Blöcke auf random Spuren:

$$1000 \cdot ZZ = 5,4s$$

Größere Blockgrößen:

- Vorteil: Weniger I/O-Operationen
- Nachteil: Mehr Platzverschwendung

SSD Berechnungen

- 80gb = 160.000.000 Blöcke
- Seek-Zeit für random Block: 15 µg
- Lesezeit pro Block: 15µg

1000 aufeinanderfolgende Blöcke:

$$Seek + 1000 \cdot Lesen = 15,015ms$$

1000 random Blöcke:

$$1000 \cdot (Seek + Lesen) = 30ms$$

Zugriff beschleunigen

- Defragmentieren (Daten / Blöcke ordnen)
- RAID-System verwenden
- Prefetching / Kompremieren

Sätze

Satz

- Gegenstandsdaten einer Anwendung
- Satzgröße/Länge wird von Anwendung bestimmt
- Sätze werden in Blöcken gespeichert
- Variable Anzahl von Sätzen pro Block
- keine blockübergreifenden Sätze

⇒ Sätze als Blockabstraktion:
Anwendungsspezifischer und variable Länge

Satzstrukturierte Dateien:

- Dateien auf mehrer Blöcke verteilen
- Platzsparender

Sequenzielle Satzdatei

Folge von Sätzen fester oder variabler Länge. Abspeicherungs- und Lesereihenfolge ist benutzerdefiniert.
Untersützt wird:

- Lesen / Überschreiben ganzer Dateien
- Anhängen / Letztes Element löschen

Nicht Unterstützt wird:

- Wahlfreies Lesen /Einfügen /Ändern von Sätzen

Blockunabhängigkeit
Anwendungsprogramm mit sequentiellen Dateien. Dadurch Speicherunabhängig

Blockung von Sätzen
Blockungsfaktor:

$$bfr = \left\lfloor \frac{Blockgroesse}{feste\ Satzlaenge} \right\rfloor$$

Ungenutzter Speicherplatz:

$$s = B - (bfs \cdot R)$$

Wechselpuffertechnik

- Optimierung verborgen vorm Programmierer
- Sätze und Blöcke werden gepuffert
- Es gibt 2 Puffer

Funktionsweise beim Schreiben:
Puffer 1 voll → Hintergrundspeicher schreiben und mit Puffer 2 weiterarbeiten. Puffer 2 voll → Hintergrundspeicher schreiben und mit Puffer 1 weiterarbeiten.
Beim Lesen funktioniert dies Äquivalent.

Direkte Satzdatei

Unterstützt wird:
Einfügen, Löschen, Ändern einzelner Sätze
⇒ Wahlfreier Zugriff!

Satzadresse (TID):

- Wird beim erstellen automatisch zugewiesen
- Verwendet für den Zugriff auf einen Satz
- Eindeutig und Unveränderlich
- Stabil: Satzadresse bleibt beim Verschieben des Satzes gleich

TID-Konzept:

- TID = Tuple IDentifier
- Satzadresse: Blocknummer + Index
- Satzzugriff benötigt nur Blockzugriff

Einfügen eines Satzes:

- Genügend freien Block suchen
- Speicherplatz belegen
- Anfangsadresse bei Index eintragen

Löschen eines Satzes:

- Belegte Blöcke als ungültig makieren
- ggf. Sätze umsortieren
- Index überschreiben, Satzadressen nicht!

Satz verkleiner:

- Sätze werden verschoben
- Indizes werden angepasst

Satz vergrößtern:

- Analog zu Satz verkleiner
- Möglich: Überlaufbehandlung!

Überlaufbehandlung:

- Satz im alten Block ⇒ Reverenz auf neuen Block
- Satz mit neuer Größe einfügen
- Bei weiterem Überlauf, wird keine zweite Reverenz erstellt, sondern die erste verändert

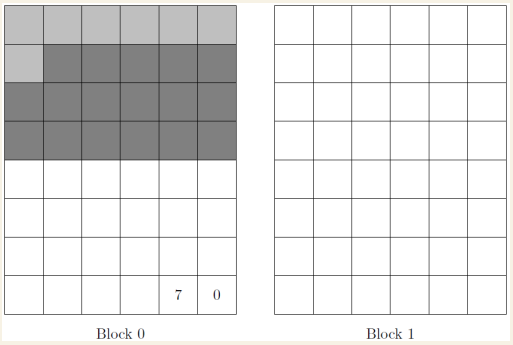
Fragmentierung:

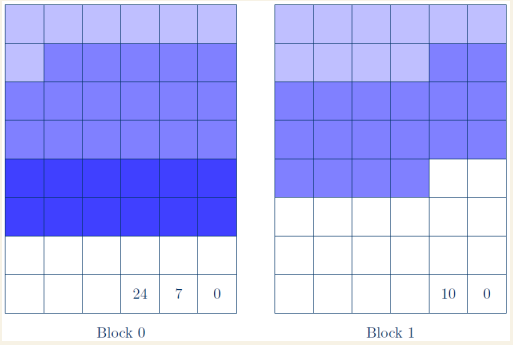
- Verwendung bei zu großen Sätzen
- Fragmente: Blockgröße- Indexgröße- Header
- Somit Satzgröße in Fragmente aufteilen und mit einem Index, einem Header im Block speichern
- Der Header ist ein TID
- Das letzte Fragment hat keinen Header
- TID ist erster zur Speicherung genutzter Block

TID angewandt
Gegeben:

- Indexeintrag = 1 Byte, TID-Reverenz = 2 Byte

12, 10, 18 einfügen und TID angeben:

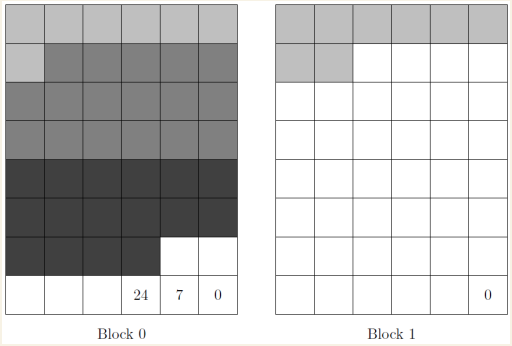


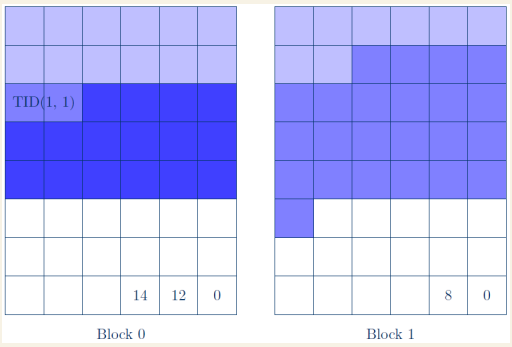


Autor: Julian Kotzur

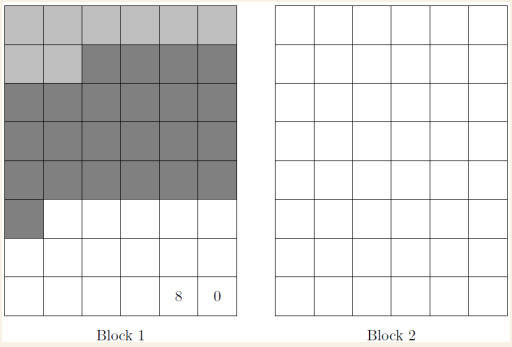
⇒ 12: TID(0,2), 10: TID(1,0), 18: TID(1,1)

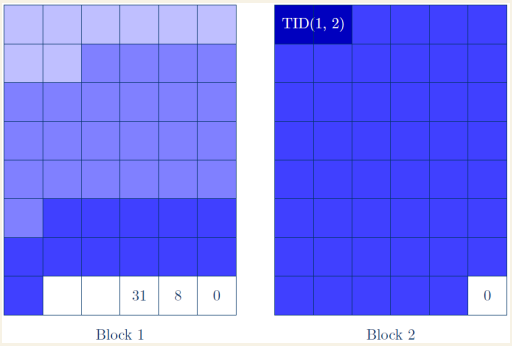
TID(0,0) auf 12 vergrößern, dann TID(0,1) auf 23 vergrößern:





Fragmentierung: Füge 57 ein:





TID Blöcke zusammenfassen:
- Sehr aufwändig, teilweise nicht möglich
- Ansatz: Satz verschieben, Zeiger behalten

Hashing

Hashing

Problem: Satzadressen haben nichts mit den Anwendungsdaten zu tun.
Ziel: Hilfsstruktur um Sätze sinnvoll sortiert abspeichert, damit man Inhalte schneller findet.
Vorgehen: Speicherposition (Bucket) wird aus Schlüsselwert berechnet. Voraussetzung: Satzanzahl ist abschätzbar, um Speicher zu reservieren
Vorteile:
- Schneller Zugriff über Schlüssel auf Inhalte

Nachteile:
- Speicherplatz muss vorher belegt werden
- Suche nur nach einem Kriterium möglich

Hinweis:
Vorbelegung ist bei Hashingvarianten mit Reorganisation nicht notwendig

Hash Funktion

Rechnet Schlüsselwerte in Bucketnummern um.
Ziel: Gleichverteilung der Sätze auf Buckets
Kollision:
Gleiche Bucketnummer bei verschiedenen Schlüsselwerten. (Möglich, teils gewollt)
Geeignete Funktion:
Ist anwendungsabhängig. Oftmals:

$$h(k) = k \ mod \ p$$

k = Schlüsselwert, p = Bucketanzahl als Primzahl
Worst Case: Alle Sätze in einem Bucket ⇒ Liste

Überlaufbehandlung

Durch Ungleichverteilung: Bucket zu klein
Open Adressing:
Ausweichen auf Nachbarbuckets
Vorteil:
- kein zusätzlicher Speicherplatz

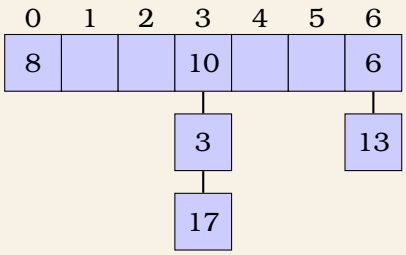
Nachteile:
- Beim Suchen oft mehrere Zugriffe
- Mehrere Überläufe

Overflow-Buckets:
Anlegen spezieller Überlauf-Buckets als verkettete Liste. Dies geschieht für jeden Überlauf Bucket.
Vorteil:
- keine Mischung von Sätzen
- Keine anderen Buckets sind betroffen

Nachteil:
- mehr Speicherbedarf

Vermeidung:
Durch regelmäßiges Reorganisieren.

Hashing mit Overflow-Buckets



- h(k) = k mod 7,
- Bucketgröße: 1
- Füge 10, 6, 3, 8, 13, 17 ein

Virtuelles Hashing

Andauernde Reorganisation der Bucket-Folge während der Einfügungen und Löschungen.

- Mit q Buckets und b Sätzen beginnen
- Kapazität: $q \cdot b$ Sätze. (ohne OverflowBuckets)
- Für den Belegungsfaktor β gilt:

$$\beta = \frac{Anzahl\ gespeicherter\ Saetze}{Kapazitaet}$$

Bei der Anzahl der Sätze zählen auch die in den Overflowbuckets. Zudem gibt es einen Schwellenwert $\alpha \in [0, 1]$, mit 0.8 als Default-Wert. Wenn $\beta > \alpha$: Menge an Buckets vergrößern.

VH1 wenn $\beta > \alpha$

- Buckets verdoppeln und am Ende anhängen
- Neue Hash-Funktion erstellen
- Bitmaske um Verwendung der neuen Hashfunktion zu verwalten
- bei Einfügen eines Satzes in einen alten Bucket: Neuverteilung dieses Buckets mittels h2, Bit setzen (Verzögert Umspeichern aller Sätze)

Lineares Hashing

Gegeben:

- Hashtabelle mit q Buckets der Größe b
- Positionszeiger p und Schwellenwert α
- Folge von Hashfunktionen h_j
- Einzufügenden Werte

Vorgehen:

- Ausgang: Alle Buckets arbeiten mit der Hashfunktion h_0, p = 0
- Einfügen, bis $\beta > \alpha$, dann splitten

Splitt:

- Bucket wird hinzugefügt
- neuer Bucket und Bucket mit Positionszeiger bekommen nächste Hashfunktion h_1
- Bucket mit Positionszeiger wird mit h_1 neu aufgeteilt
- Positionszeiger++

Weiteres Vorgehen:

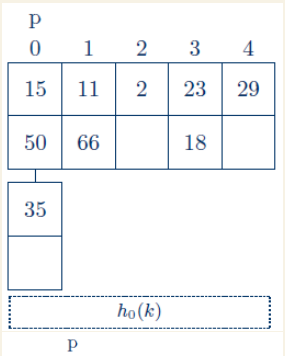
- Weiterhin mit h_0 neue Werte eintragen
- Wenn $h_0(k) < p$ dann mit h_1 hashen

Bemerkung:

- wenn p = q gilt (p steht auf letztem Ausgangsbucket) setzt man p zurück auf 0
- Man beginnt dann von vorne
- Überlaufbuckets sollte man verwenden!

Lineares Hashing

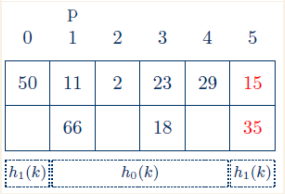
- kontrolliertes Splitting mit $\alpha = 0.8$
- Einfügen:18,50,66,2,35, 51,17,22,38,85,32
- $h_0(k) = k\ mod\ 5$, $h_1(k) = k\ mod\ 10$



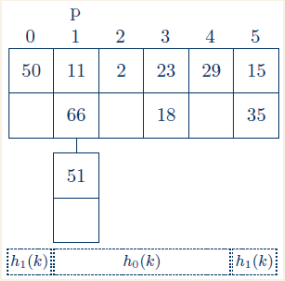
Ausgangszustand, in den 18,50,66,2 und 35 eingefügt wurden. β berechnen:

$$\beta = \frac{9}{10} = 0.9$$

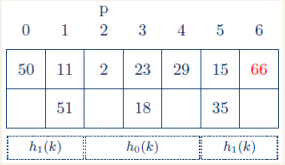
Damit gilt $\beta = 0.9 > \alpha$



Split von Bucket 0. Damit ist $\beta = 0.75$

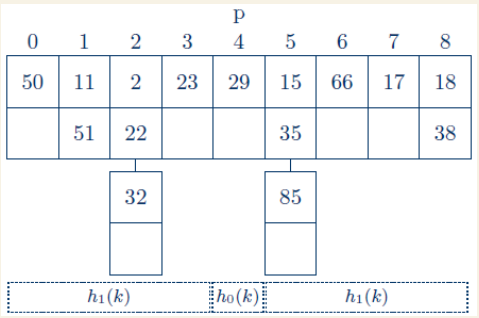


Nun wird 51 eingefügt, damit ist $\beta = 0.83 > \alpha$

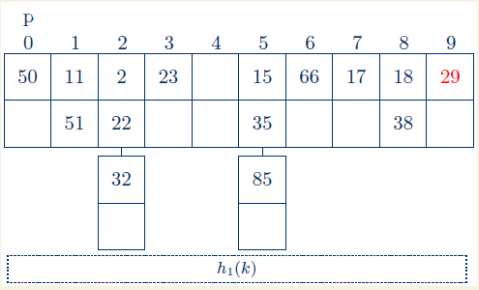


Split von Bucket 1. Damit ist $\beta = 0.71$

Nun wird noch Bucket 2 und 3 gesplittet. Damit steht der Positionszeiger auf 4. Wenn man nun 85 und 32 einfügt, erhält man folgenden Zustand:



Durch Splitten von Bucket 4 gelangt man zu:



Autor: Julian Kotzur

Bäume

B-Baum

Aufbau und Eigenschaften:

- Ein Knoten entspricht einem Block
- k: Blockgröße: Jeder Knoten hat min. k und max. 2k Einträge (Gilt nicht für Wurzelknoten)
- Höhe h: Untere Schranke: $h(n) = log_{2k+1}(n+1)$
  $\Rightarrow$ Obere Schranke: $h(n) = log_{k+1}((n+1)/2)+1$
- Perfekt Balanciert: Alle Pfade von Wurzel zu Blatt sind gleich lang ist
- Innere Knoten mit t Einträgen haben t+1 Kinder
- Alle Knoten sind sortiert

Suche nach Wert K:

- Ähnlich zu binärer Suche

Einfügen:

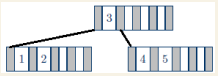
- Eingefügt wird nur in Blattknoten, dieser wird anhand von Suche gefunden
- Sonderfall: Blattknoten ist voll $\Rightarrow$ Splitt
- Neuen Blattknoten auf der Ebene erzeugen
  $\Rightarrow$ Die ersten k Sätze bleiben wo sie sind
  $\Rightarrow$ Die letzten k Sätze kommen ins neue Blatt
  $\Rightarrow$ Die Mitte(k+1) kommt in den Eltern Knoten
- Sonderfall: Innerer Knoten läuft über:
- Zwei Kindknoten erstellen. Links die ersten k Einträge, Rechts die letzten k Einträge. Der Mittlere Eintrag bleibt im Wurzelknoten
- Linke Kindknoten des Wurzelknotens kommen an den neuen Linken Knoten, Rechts analog

Löschen:

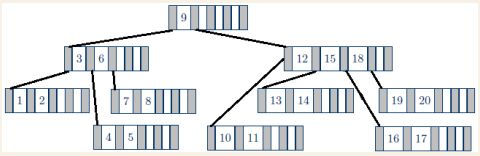
- Knoten Suchen, Eintrag durch Eintrag aus Blattknoten ersetzen: Größter Links, Kleinster Rechts, je nachdem was voller ist
- Bei entstandenem Unterlauf: Rückwärts Spitting anwenden. Kann bis zum Wurzelknoten hoch gehen. Leere Knoten erst am Ende löschen! Dazu Mit Linkem oder rechten Nachbarn und dem einen dazugehörigen Parenteintrag mischen und Leer hochschreiben (ggf. Überlauf behandeln).
- Dabei kann wieder ein Überlauf entstehen (meist bei der Wurzel): Wieder Splitting!
- Baumhöhe kann um 1 schrumpfen

B-Baum Beispiel:

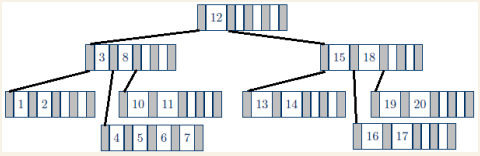
- Sei k = 2 und der Baum leer
- Füge Zahlen von 1 bis 20 ein



Am Ende hat man das Ergebnis:



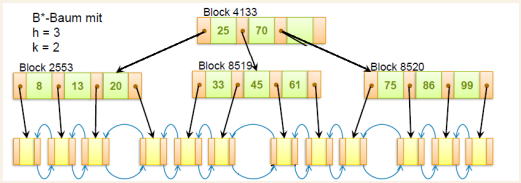
Nun löscht man die 9 und erhält:



B*-Baum

Aufbau:

- Alle Sätze sind in Blattknoten
- Blattknoten verweisen auf Blattknoten
- Innere Knoten haben nur MetaDaten
- Blattknoten sind immer aufsteigend sortiert



Einfügen von Tupel (x,y):

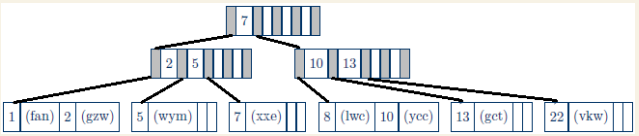
- Tipp: Wurzel anfangs auch ein Blattknoten!
- Wie B-Baum: Blattknoten suchen und einfügen
- Sonderfall: Blattknoten ist voll $\Rightarrow$ Splitt
  $\Rightarrow$ Teile das Blatt in zwei Blätter, links/rechts
  $\Rightarrow$ Links: Tupel kleiner gleich x
  $\Rightarrow$ Rechts: Tupel größer als x
  $\Rightarrow$ x kommt in den Parent-Knoten (sortiert)
- Sonderfall Wurzelknoten hat Überlauf:
  $\Rightarrow$ Neuer Wurzelknoten mit mittlerem Element
  $\Rightarrow$ Aus Wurzelknoten linkes Kind mit Linken Einträgen, dessen Blätter die Linken Kinder des ursprünglichen Wurzelknotens sind.
  $\Rightarrow$ Rechts analog

Löschen:

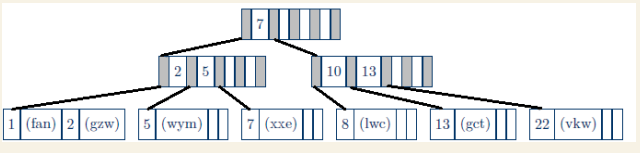
- Tupel zuerst nur im Blattknoten entfernen
- Sonderfall Unterlauf:
  $\Rightarrow$ Leere Blöcke beim Löschen nicht entfernen!
  $\Rightarrow$ Spitting Rückwärts durchführen
  $\Rightarrow$ Entferne MetaDaten des gelöschten Wertes
  $\Rightarrow$ Weitere Unterläufe ebenso behandeln
  $\Rightarrow$ Unterlauf bei inneren Knoten: Parentknoten mit Unterlaufknoten und Geschwisterknoten sortiert zusammenfassen. Alle Kinder mit zusammengefassten Knoten verbinden

B*-Baum Beispiel:

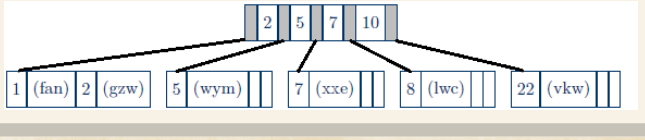
- Maximale Einträge innen: 4 (k = 2)
- Maximale Einträge Blatt: 2 (k = 1)
- Einfügen: (1,fan), (13,gct), (7,xxe), (8,lwc), (22,vkw), (5,wym), (2,gzw), (10,ycc)



Lösche die 10:



Lösche die 13:



Vergleich von B-Baum und B*-Baum

B-Baum	B*-Baum
Redundanzfrei	Redundant
Unsortiert	Sortiert
Geringe Verzweigung	Viel Verzweigung
Eher Höher	Eher weniger Höhe
Differente Suchzeiten	Gleiche Suchzeiten

B-Baum vs. binärer Suchbaum

Beim B-Baum ist ein Knoten ein Block. Da Blockzugriff teuer ist, ist B-Baum besser. Dieser hat einen größeren Verzweigungsgrad und damit weniger Blockzugriffe.

B*-Baum und Bereichsanfragen

Bereichsanfragen können hier effizient durchgeführt werden, da Blattknoten verkettet sind.

Überlauf zu Unterlauf?

Bei einer Überlaufbehandlung kann kein Unterlauf entstehen. Andersherum schon!

Primär und Sekundärorganisation

Primärorganisation:
Organisiert die direkte Speicherung der Sätze, entscheidet selbst in welchem Block ein Satz liegt. Es gibt nur eine Primärorganisation. Hierbei werden die TIDs verwaltet und z.b. mittels Hashing erzeugt (Blockzuteilung). Der Primärschlüssel muss UNIQUE sein!

Sekundärorganisation:
Organisiert nur Verweise auf die Sätze, wenn die Primärorganisation Direktzugriff unterstützt. Es kann mehrere Sekundärorganisationen geben.

B-Baum meist Sekundärorganisation, da ansonsten die Speicherplatzausnutzung nicht effizient ist.

R-Bäume
Grundlage:

1. B-Bäume mit mehrdimensionalen Knoten
2. Maximale Knotenkapazität: M = 4
3. Minimale Kapazität: m = $\lfloor \frac{M}{2} \rfloor$

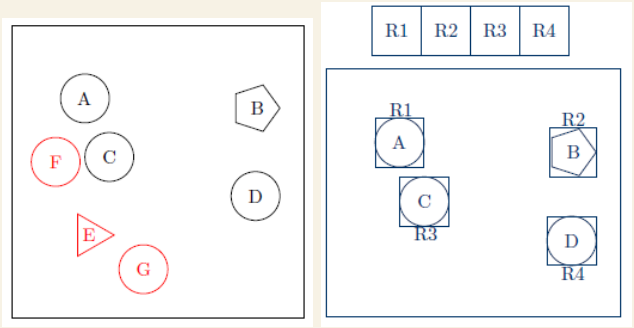
Einfügen:

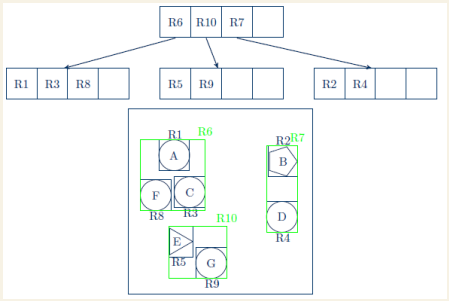
- Zuerst: Quadrat / Rechteck neu einzufügenden Eintrag
- Wurzelknoten hat noch keine Meta Rechtecke
- Eingefügt wird der neue Eintrag in das Rechteck, welches am wenigsten erweitert werden muss
- Überlauf: Zwei neue minimale Rechtecke, sodass die Einträge immernoch alle überdeckt werden. Ein neues Rechteck kommt als Eintrag in den Parentknoten, dass andere erhält den Namen des alten Rechteckes und ist somit schon im Parentknoten.
- Splitt im inneren Knoten (z.b. Wurzelknoten): Zusätzliches zwei minimale Meta Rechteck, welches die Rechtecke beinhaltet. Die beiden Meta Rechtecke sind Einträge des gesplitteten Knotens

Löschen:

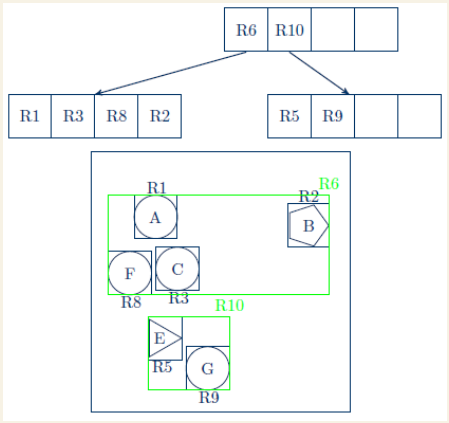
- Eintrag löschen
- Unterlauf: Meta Rechteck löschen und vorhandenes Rechteck so erweitern, dass die Erweiterung minimal ist und den freien Eintrag wieder enthält

Beispiel:
Füge A,B,C,D, E, F, G ein:





Lösche D:



Bitmap-Indexstruktur

- Pro Schlüsselwert eine Bitliste
- Größe: Anzahl Werte mal Anzahl Sätze in Bit
- Vorteil: Einfache und effiziente Operationen

Puffer

Wozu dient der Datenbankpuffer?

- Daten im Hauptspeicher zwischenspeichern
- Höhere Geschwindigkeit (RAM > HDD)

Warum nicht Pufferverwaltung des BS?

- Wird weiterhin verwendet
- Eigener Puffer ist anwendungsspezifischer
- BS bietet nicht die gewünschten Schnittstellen

Seiten:

- Seite ist ein Block im Puffer (gleiche größe)
- Mehrere Blöcke pro Seite möglich (Alter und neuer Block)

Ersetzungsstrategien

First in First out (FiFo):

- Ersetzt ältesten Block

Least frequently used (LFU):

- Ersetzt seltensten benutzten Block

Least recently used (LRU):

- Ersetzt am längsten nicht benutzten Block
- Beispiel Stack: Ersetzt untersten Block. Benutzte werden immer oben drauf gelegt

Second Chance (CLOCK):

- Approx. LRU mit einfacher Implementierung
- Jeder Block besitzt ein Benutzt Bit
- Es gibt einen Zeiger auf einen Block
- Bei Verdrängung Fallunterscheidung:
  ⇒ BlockBit = 1: Setzte auf 0, gehe Weiter
  ⇒ BlockBit = 0: Ersetze Block, gehe Weiter
- Wichtig: Zeigerposition merken!

CLOCK Beispiel

⇒ Kacheln sind zu Beginn leer
⇒ Einfügen: 1, 2, 3, 5, 1, 1, 4, 1, 2, 1, 3, 1.

Zeitpunkt		1	2	3	4	5	6	7	8	9	10	11	12
Referenzfolge		1	2	3	5	1	1	4	1	2	1	3	1
Hauptspeicher	Kachel 1	1	1	1	5	5	5	5	5	2	2	2	2
	Kachel 2		2	2	2	1	1	1	1	1	1	1	1
	Kachel 3			3	3	3	3	4	4	4	4	3	3
Kontrollzustände (Benutzt-Bit)	Kachel 1	1	1	1	1	1	1	1	1	1	1	1	1
	Kachel 2	0	1	1	0	1	1	1	1	0	1	0	1
	Kachel 3	0	0	1	0	0	0	1	1	0	0	1	1
Auswahlzeiger auf Kachel-Nr:		2	3	1	2	3	3	1	1	2	2	1	1

⇒ Beispiel: Fenstergröße w = 8
mit 1, 2, 1, 3, 1, 2, 1, 2, 3, 4, 5, 6, 7
⇒ $|W(t,w)|$: Anzahl unterschiedlich Referenzierter Seiten bei w Zugriffen
⇒ Aktuelle Lokalität:

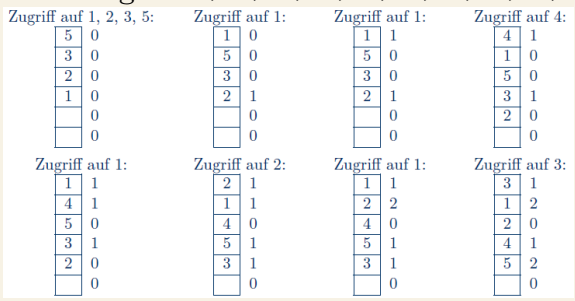
$$AL(t,w) = \frac{|W(t,w)|}{w} \quad t=8: \frac{3}{8} \quad t=13: \frac{7}{8}$$

⇒ Durchschnittliche Lokalität:

$$L(w) = \frac{\sum_{t=w}^{n} AL(t,w)}{n-(w-1)} = \frac{1}{6} \cdot \left(\frac{3}{8}+\frac{3}{8}+\frac{4}{8}+\frac{5}{8}+\frac{6}{8}+\frac{7}{8}\right)$$

LRU Stack Beispiel

⇒ Zählerwerte = Wiederbenutzungshäufigkeit
⇒ Einfügen: 1, 2, 3, 5, 1, 1, 4, 1, 2, 1, 3.



Was bringt Stacktiefenverteilung?
Für eine bestimmte Puffergröße können die Treffer- und Fehlseitenrate bestimmt werden.

Warum LRU in BS schwierig ist?
Ohne Hardwareuntersützung hat man viele Verwaltungsoperationen, dadurch schlechte Performance. Mit Hardwareunterstützung benötigt man viel mehr Speicher.

Fehlersituation

- BS-Absturz, Hardwarefehler, Stromausfall
- Vorbeugung durch Einbringstrategien
  ⇒ Schattenspeicher / Twin Slots

Seitenzuordnung

Direkt:

- Aufeinanderfolgende Seiten sind innerhalb der Datei auf aufeinanderfolgenden Blöcken
- Man muss nur erste Blocknummer kennen
- Seiten können im Puffer verstreut sein
- Schnell und Platzsparend

Indirekt:

- Tabelle legt fest, welche Seite in welchem Block einer Datei liegt (Abbildungstabelle)
- Flexibler als direkte Seitenzuordnung

Seiteneinbringung

Direkt:

- Beim Verdrängen wird der Block überschrieben, aus dem die Seite eingelesen wurde
- Vorteil: Es ist Supereinfach
- Nachteil: Nicht fehlertolerant

Indirekt:

- Beim Verdrängen werden Seiten in freie Blöcke geschrieben. Ursprüngliche Blöcke bleiben erhalten und werden später überschrieben
- Vorteil: fehlertolerant
- Nachteil: Mehr Platzverbrauch

Alte Blöcke wieder löschen:

- Schattenspeicher: Autosave in Spielen, man hat einen aktuellen Version und eine Sicherung. Die Sicherung wird periodisch überschrieben
- Twin Slots: Jede Seite hat zwei Blöcke. Beide werden gelesen und der ältere ersetzt.

Programmzugriff

Eingebettetes SQL (Precompiler)

Arbeitsweise:
- SQL Anfragen im Code werden vom Precompiler zu richtigem Code
- Syntax für SQL: EXEC SQL < Anweisung >;
- Speichervariablen: Mit INTO speichern und mit ':<Name>' Variable auswählen

Vorteile:
- Kompakt zu programmieren, Effizienter
- Fehlererkennung zur Übersetzerzeit
- Keine SQL Injection möglich!

Nachteile:
- Teilweise kein Standart SQL
- Spezielle Vorübersetzer notwendig

Cursor:
- Tupelzeiger um mit Mengen zu arbeiten
- Mit FETCH Tupel sequentiell holen

Beispiel:

```
// Variablen fuer SQL bereitlegen
EXEC SQL BEGIN DECLARE SECTION;
 String name;
EXEC SQL END DECLARE SECTION;
// SQL Anfrage
EXEC SQL SELECT Nachname
 // Speichervariablen waehlen mit ':'
 INTO :name FROM Persons WHERE PNR = 42;
// Cursor:
EXEC SQL DECLARE c1 CURSOR FOR
 SELECT name FROM Persons WHERE Gh > 42;
EXEC SQL OPEN c1;
while(1) { EXEC SQL FETCH c1 INTO :name; }
EXEC SQL CLOSE c1;
```

Unterprogramm (API, Call-Level Interface)

Vorteile:
- Einfacherer für den Hersteller

Nachteile:
- Programmierer hat viel Arbeit
- SQL Injections möglich

Beispiel:

```
// Verbindung zur Datenbank
Connection con = getConnect ion ( ) ;
// Anfrage erstellen:
String sql = "SELECT ..."
Statement stat = con.CreateStatement();
ResultSet res = stat.executeQuery(sql);
// Ergebnis mit Schleife bearbeiten:
while(res.next()){
 // Die Zahl am Ende ist Attributindex
 String name = res.getString (1);
 print(name); }
res.close(); stat.close();
```

Optimierung:
- Vorübersetzte Anfragen
- Gespeicherte Prozeduren

Was muss eine API bieten?
- Methode um Abfrage auszuführen
- Methode zum überprüfen ob es Ergebnisse gibt
- Methode um Ergebnis zu erhalten

Ablauf SELECT ... FROM x WHERE y
- Syntaxprüfung
- Gibt es Relation und Attribute überhaupt?
- Dateiname der Relation finden
- Direkt oder Schlüsselbasierte Datei?
- Gibt es einen Index für y?
- Wie heißt Datei zu y?
- Wie lang sind Sätze von x?

SQL Injection verhindern:

Prepared Statements verwenden

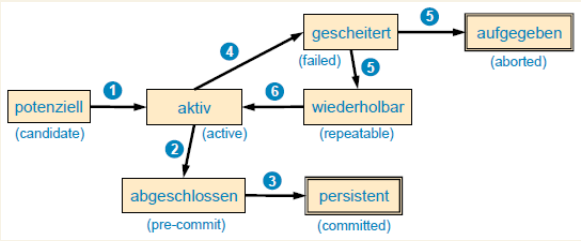
O/R-Mapping

Bildet Objekte der Programmiersprache auf Tupel der relationalen Datenbank ab

Transaktion

Warum Transaktionen:
- Durch Fehler können Probleme entstehen, die manuell behoben werden müssen
- Daher ist eine Systemuntersützung fürs Recovery wichtig
- Zudem sinnvoll bei nebenläufigem Zugriff

Zustände einer Transaktion



Konsistenz
- Erwüschte Zustände für Daten auf der Platte
- **Physische Konsistenz:** Korrekte Speicherstrukturen, alle Verweise und Adressen stimmen. Alle Indexe sind vollständig und stimmen mit den Primärdaten überein
- **Logische Konsistenz:** Korrektheit der Dateninhalte. Alle Bedingungen des Datenmodells und alle Integritätsbedingungen sind erfüllt
- Nach Fehler herrscht meist weder logische noch physische Konsistenz

Die Transaktion
- Anwender definiert die Transaktion
- **commit**: Mitteilung, dass logisch konsistenter Zustand hergestellt ist

Autor: Julian Kotzur

- **abort**: Rückkehr zum Anfangszustand
- Transaktion stellt logische Arbeitseinheit dar, welche mehrere DB-Operationen zusammenfässt
- Fehler vor Ende der Transaktion: Änderungen werden rückgängig gemacht

ACID-Eigenschaften (nach Hofmann)

- **Atomarität:** Unteilbar: Alle Operationen der Transaktion bilden eine Einheit. Ununterbrechbar: Transaktion wird entweder vollständig oder garnicht ausgeführt
- **Konsistenzerhaltung:** Transaktion überführt die DB von einem konsistenten Zustand in einen anderen konsistenten Zustand
- **Isolation:** Eine Transaktion wird nicht von anderen Transaktionen beeinflusst. Daten sind somit blockiert
- **Dauerhaftigkeit:** Eine erfolgreich abgeschlossene Transaktion ist dauerhaft. Ihre Änderungen gehen nicht durch spätere Fehler verloren

Fragen:

Geeignetes Transaktionsende:

- Geeignet: Schleifenende, Programmende
- Wichtig: Die ACID-Eigenschaften dürfen nicht verletzt werden
- Vorgehen: Mögliche Transaktionsenden finden und prüfen ob ACID gilt

Probleme bei Transaktionen:

- Benutzereingaben innerhalb der Transaktion kann beliebig lange dauern und die Daten genauso lange sperren. Lösung: Benutzereingaben vor Transaktionsstart

Synchronisation

Warum Synchronisation

Erhaltung der Konsistenz im Mehrbenutzerbetrieb

Anomalien ohne Synchronisation

- Dirty Read: Transaktion liest noch nicht freigegebene Daten ($w_1(x), r_2(x), c1/a1, c2/a2$)
- Dirty Write: Transaktion schreibt noch nicht freigegebene Daten ($w_1(x), w_2(x), c1/a1, c2/a2$)
- Non Repeatable Read: Transaktion liest ein Tupel mehrmals und erhält unterschiedliche Ergebnisse($r_1(x), w_2(x), r_1(x), c1/a1, c2/a2$)
- Phantom, Lost Update

Lösung: Serialisierung

Serialisierung: Transaktionen hintereinander ausführen. Ein Serialisierbarer Ablauf ist eine Abarbeitung von Transaktionen, bei der das Ergebnis äquivalent zur Serialisierung ist.
Dabei muss folgendes gelten:

$$r_i[A] <_H w_j[A] \Leftrightarrow r_i[A] <_G w_j[A]$$

$$w_i[A] <_H r_j[A] \Leftrightarrow w_i[A] <_G r_j[A]$$

$$w_i[A] <_H w_j[A] \Leftrightarrow w_i[A] <_G w_j[A]$$

Man erkennt, dass 2 mal lesen (r) keine Rolle spielt

Abhängigkeitsgraph

Erstellen:

- Zeiche Transaktionen als Knoten
- Gehe Ablauf sequentiell durch: Wenn Daten der aktuellen Transaktion nochmal verwendet werden, mache Pfeil von aktueller Transaktion zu anderer Transaktion. Schreibe die Daten an die Kante
- Dies gilt nicht, wenn zweimal read hintereinander auf die selbe Datei ausgeführt wird.

Der Graph ist serialisierbar, wenn keine Zyklen vorkommen! Ansonsten ist eine Ausführreihenfolge, den Graphen so abzusteigen, dass man jeden Knoten einmal besucht.

Implementierung durch Sperren

Durch Sperren kann Serialisierbarkeit gewährleistet werden.

Wann Sperren:

- Statisch Sperren: Zu Beginn der Transaktion. Nachteil: Es wird zu viel gesperrt
- Dynamisch Sperren: Während der Transaktion. Nachteil: Deadlock möglich
- Sperrenauflösung am Ende der Transaktion

Arten von Sperren:

- S: Geteilte Lesesperre
- X: Exklusive Schreibsperre
- IS Sperre: Bevor ein Knoten mit S gesperrt wird, werden alle Vorgänger mit IS gesperrt
- IX Sperre: Bevor ein Knoten mit X gesperrt wird, werden alle Vorgänger mit IX gesperrt
- Die Sperre wird dabei von Oben nach Unten (Top-Down) erzeugt und von Unten nach Oben (Bottom-Up) wieder aufgelöst
- Zur Optimierung gibt es noch die SIX Sperre. Alles wird lesend gesperrt, aber nur ein paar Nachfolger schreibend gesperrt. Gut, wenn sich wenig verändert und viel gelesen wird

Probleme des Sperrens:

- Sperren muss sehr schnell gehen, da Anforderungen sehr hoch
- halten von Sperren bis Transaktionsende führt zu langen Wartezeiten
- Explizites Sperren führt zu umfangreichen Sperrtabellen
- Datensperrung ist ungleich verteilt: Manche Daten werden öfters gesperrt ⇒ Warteschlange

Lösungen für Deadlocks:

- Timeout: Transaktion nach t Zeit zurücksetzen
- Verhütung: Preclaiming aller Daten
- Deadlocks erkennen und vermeiden
- Erkennung: Wartegraphen verwenden und vorkommende Zyklen auflösen

Recovery

Ziel

Erhaltung der physischen und logischen Konsistenz der Daten. Dabei sind Sicherungen und Protokolierungen (Logging) immer notwendig.

Fehlerarten

Transaktionsfehler, Systemfehler, Gerätefehler, Katastrophe: z.b. Serverraum brennt

Recovery Klassen

- **Undo:** Zurücksetzen aller (global) oder einzelner (partial) Transaktionen
- **Redo:** Wiederholen aller (global) oder einzelner (partial) Transaktionen nach einen sicheren, wiederhergestellten Zustand
- **Forward Recovery:** Nur Redo Recovery, hoher Speicherbedarf, da TA's bis zum Ende im Puffer stehen
- **Backward Recovery:** Nur Undo Recovery, hohe I/O-Aufwand, da TA's Zwischenergebnisse auslagern
- Daher Kombination von Forwad und Backward, da nur ein größerer Log benötigt wird

Einbringstrategien

Wann dürfen Seiten auf die Festplatte?

- STEAL: Vor Transaktionsende bei Verdrängung aus dem Puffer
- NOSTEAL: Nur bei Transaktionsende

Wann müssen Seiten auf die Festplatte?

- NOFORCE: Bei Verdrängung aus dem Puffer
- FORCE: Am Ende erfolgreicher Transaktionen

Wie kommen Seiten auf die Festplatte?

- NOTATOMIC: unterbrechbares direktes Einbringen: in-place
- ATOMIC: ununterbrechbares indirektes Einbringen: neuer Speicherort

Protokollverfahren

- **Zustandsprotokollierung:** Before-Image für Undo, After-Image für Redo
- **Seitenprotokollierung:** Für jede geänderte Seiten before- und afterimage sichern
- **Eintragsprotokollierung:** Aufwandreduzierung, da nur die geänderten Teile der seite protokolliert werden

Checkpoints

- **transaction oriented:** Einbringung direkt am Ende der Transaktion. Kein Redo notwenig
- **transaction consistent:** Anmeldung eines checkpoints Verzögert neue Transaktionen, bis alle aktuellen fertig sind und checkpoint erzeugt wurde. Undo und Redo werden begrenzt
- **action consistent:** Sicherung, wenn keine Änderungen aktiv sind. Geringere Qualität des Recoverys. Redo wird begrenzt

Wiederherstellungsprozedur

- Analyse vom letzten Checkpoint bis Log-Ende
- erfolgreiche Transaktionen wiederholen (Redo)
- fehlgeschlagene Transaktionen rückgängig machen (Undo)

Speicherung:

Speicherung von Tupeln in Sätzen

- Sätze bestehe aus einzelnen Felder
- Beispiel Felder: Name, Wert
- Satztyp: Satzmenge mit gleicher Struktur
- Anforderung an Tupelspeicherung:
  ⇒ Speichereffizienz
  ⇒ Direkter Zugriff auf Felder
  ⇒ Flexibilität (Add, Delete)

Systemkatalog

Enthält Informationen über die Felder und deren Reihenfolge (Metadaten). Beispiele für solche Metadaten: Name eines Feldes, Charakteristik (feste, variable Länge), Länge.

Speicherstrukturen in Sätzen

Aufbau:

- GL Gesamtlänge
- FL Länge des festen Strukturteils (mit PTR)
- Z Zeiger (mit PTR)
- F Inhalt fester Länge
- L Längenangabe
- V Inhalt variabler Länge

Hinweis: Zeiger nur für Inhalte variabler Länge

Speicherstrukturen in Sätzen:

Einfügen:

- 1, Müller, 91058, Erlangen, 20100427

Gegeben:

Kundennr	NUMBER(6)
Name	CHAR(120)
PLZ	NUMBER(5)
Einkauf	DATE

Mit fester Feldlänge:

Feldname	Datentyp	Pos.	Inhalt	Länge
Gesamtlänge	Länge	0	142	4
Kundennr.	NUMBER(6)	1	1	5
Name	CHAR(120)	2	Müller	120
PLZ	NUMBER(5)	3	91058	4
Einkauf	DATE	4	20100427	7

Mit variabler Feldlänge:

Feldname	Datentyp	Pos.	Inhalt	Länge
Gesamtlänge	Länge	0	41	4
Länge KN.	Länge	1	2	4
Kundennr.	NUMBER(6)	2	1	2
Länge Name	Länge	3	12	4
Name	CHAR(120)	4	Müller	12
Länge PLZ	Länge	5	4	4
PLZ	NUMBER(5)	6	91058	4
Einkauf	DATE	4	20100427	7

Autor: Julian Kotzur

Mit Zeigern der Länge 4:

Feldname	Datentyp	Pos.	Inhalt	Länge
Gesamtlänge	Länge	0	57	4
Zeigerlänge	Länge	1	31	4
Z KN	Zeiger	2	Pos. 6	4
Z Name	Zeiger	3	Pos. 8	4
Z PLZ	Zeiger	4	Pos. 10	4
Einkauf	DATE	5	20100427	7
Länge KN.	Länge	6	2	4
Kundennr.	NUMBER(6)	7	1	2
Länge Name	Länge	8	12	4
Name	CHAR(120)	9	Müller	12
Länge PLZ	Länge	10	4	4
PLZ	NUMBER(5)	11	91058	4

C-Store

- Tupel werden spaltenweise über mehrere Sätze abgespeichert (Column Store)
- Sortiert nach einem Attribut
- Schreibspeicher: Zum schnellen Einfügen
- Leseoptimierter Speicher zum schnellen Lesen (macht C-Store aus)
- Flexibilität durch Löschen / Einfügen
- Vorteil: analytische Auswertungen
- Nachteil: Änderungen und Joins

Projektion:

- Tabelle aus einem oder mehreren Attributen einer Relation, sortiert nach einem Attribut
- Die Menge der Projektionen enthält alle Daten der Relation (kein Datenverlust)

Aufbau:

Name	Age	Dept	Salary
Bob	25	Math	10K
Bill	27	EECS	50K
Jill	24	Biology	80K

Table 1: Sample EMP data

```
EMP1(name, age| age)
EMP2(dept, age, DEPT.floor| DEPT.floor)
EMP3(name, salary| salary)
DEPT1(dname, floor| floor)
```

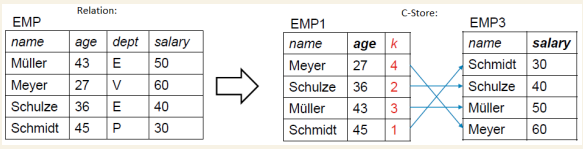
Example: Projections in Example 1 with sort orders

Speicherschlüssel:

- Jedes Attribut (Spalteneintrag) hat einen Schlüssel
- Schlüssel ist aus Position berechenbar
- Beim Lesespeicher ableitbar
- Beim Schreibspeicher fest gespeichert

Verbund-Index:

- Um die Ausgangsrelation wieder zu konstruieren
- Hilfsstrukturen die mitabgespeichert sind
- Zusammenhängende Tupel innerhalb zweier Pojektionen verweisen aufeinander. T1 hat somit einen Zeiger gespeichert, sodass das Tupel aus T1 auf das zugehörige Tupel aus T2 zeigt!



Komprimierungen:

- **Sortiert weniger Werte:** Tripel(Wert, Position, Anzahl) in B-Baum
- **Unsortiert wenige Werte:** Bitmaps pro Wert. Bitmaps werden mit B-Baum verwaltet
- **Sortiert viele Werte:** Delta-Kodiert mit B-Baum
- **Unsortiert viele Werte:** unkomprimiert

ANFRAGEVERARBEITUNG

Aufgabe der Anfrageverarbeitung

Abbildung von mengenorientierten Operationen auf effiziente satzorientierte Operationen!

Phasen der Anfrageverarbeitung

- Syntaxanalyse: Erstelle Anfragebaum
- Semantikanalyse: Prüfe Relationen / Attribute
- Zugriffs und Integritätskontrolle
- Standardisierung und Optimierung
- Code Generierung
- Ausführungskontrolle

Relationale Algebra

Operationen:

- PROJ(R,L): Auswahl der L Attribute aus R
- SEL(R, pred(... AND ...)): in pred steht hierbei eine Gleichung. Dabei wird eine Teilmenge von Tupeln aus R ausgewählt
- Mengenorientierte Operationen:
  - CROSS(R1,R2,...,Rn)
  - JOIN(R1,R2,...,Rn, pred(...))
  - UNION(R1,...,Rn), Mengenvereinigung
  - INTERSECT(R1,R2) Mengenschnitt
- RENAME(R,name)
- DUP-ELIM, Duplikatelimination
- Aggretaton z.b. SUM(R, name)

Anfragebaum Struktur

- Gegeben ist SQL Anfrage
- Pfeile zeigen immer auf das leere zw. Kommas
- **Proj**(, (< Alles nach SELECT >))
- **SEL**(, (< Alles nach HAVING (mit neuen Namen und Gleichung) /or/ Alles nach WHERE >)). Bei AND: Mehrere Gleichungen in der Klammer durch AND voneinander getrennt aufschreiben.
- **GROUP**(, (< Das nach GROUP BY >) , < Rechenoperationen in HAVING UND SELECT, < neuerName > >))
- **CROSS**(„,) mit n-1 Kommas, mit n als Anzahl der Relationen aus FROM /or/ **JOIN**(„, (< Alles Nach ON >)) äquivalent zu CROSS nur mit ON
- **Relationenname** ⇒ **Rename**(, < neuer Name >) ⇒ **CROSS/JOIN** für alle Relationen von FROM und alle Relationen nach JOIN
- Union funktioniert wie ein CROSS, nur dass man dabei Unterabfragen hat, welche mit einem PROJ enden
- Berechnungen wie AVG(a) in SELECT innerhalb des Baums in GROUP durchführen und nur dessen Namen in PROJ schreiben

Restrukturierung (Optimierung)

Regeln:

- **Mache Joins Binär:** JOIN(R1,R2,Rn) = JOIN(R1, JOIN(R2, RN, pred(R2, RN)) , pred(R1, Rest)
- **Selektionen zusammenfassen:** SEL(SEL(R, pred1), pred2) = SEL(R,(pred1 AND pred2))
- **Projektion zusammenfassen:** PROJ(PROJ(R,L1),L2) = PROJ(R, L2)

- **Projektionen erweitert vorziehen:** PROJ(SEL(R,pred (M)),L) = PROJ(SEL(PROJ(R,(L ∪ M)), pred (M)),L)
- **SEL und JOIN tauschen:** SEL(JOIN(R,S,pred1), pred2(R)) = JOIN(SEL(R,pred2), S, pred1)
- **SEL und UNION tauschen:** SEL(UNION(R,S), pred) = UNION(SEL(R,pred), SEL(S,pred))
- **SEL und CROSS zusammenfassen:** SEL(CROSS(R,S), pred) = JOIN(R,S,pred)

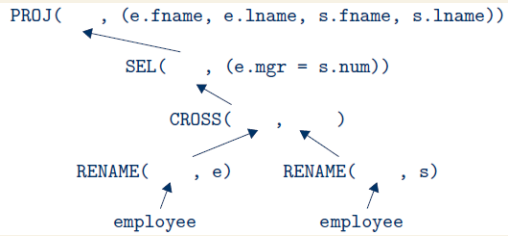
Algorithmisches Vorgehen:

- Komplexe Verbünde binär machen
- Selektionen separieren und nach unten schieben
- SEL und CROSS zusammenfassen
- SEL wieder zusammenfassen
- Projektionen so früh wie möglich

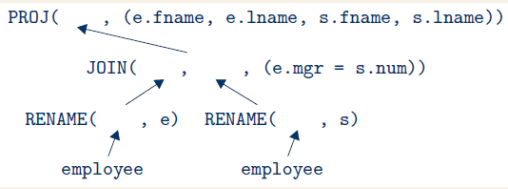
Beispiel:
Gegeben:

- SELECT e.fname , e.lname , s.fname , s. lname
- FROM employee e, employee s
- WHERE e.mgr = s.num

Nicht optimiert:



Rekonstruiert:



Ausführungsplan mit Sichten

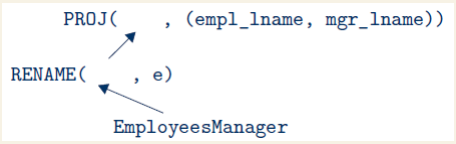
Virtuelle Sichten:
Zwischenergebnisse einer SQL-Anfrage. Werden erzeugt und dann wieder verworfen.

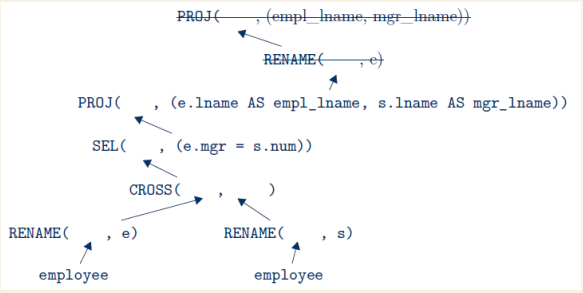
Materielle Sichten:
Relationen die durch eine SQL Anfrage entstehen, aber fest gespeichert sind! Beim Aufruf muss die SQL Anfrage nicht nochmal abgearbeitet werden.

Optimierung:

- Aufgabe: Ausführungsplan erstellen
- Nicht optimierten Ausführungsplan A erstellen
- Ersetze in A Grundrelation durch Ausführungsplan der virtuellen Sicht
- Wenn Möglich: Ersetze einen Teilbaum durch die materielle Sicht
- Nun kann man noch unnötige Anfrage löschen (z.b. wenn 2 mal gleiches PROJ vorkommt, löscht man das obere)

Beispiel:





Proj und Schnittmenge vertauschen

Nicht möglich, da in der Projection Daten verworfen werden, die relevant für die Schnittmenge sind!

Optimierung

Logische Operatoren vs. Planoperatoren

Logische Operatoren (SEL, PROJ, JOIN, ...) repräsentieren die Semantik eines Operators. Planoperatoren repräsentieren die Implementierung eines logischen Operators

Zu lösende Teilprobleme:

- Gruppierung von direkt benachbarten Operatoren zur Auswertung durch einen einzigen Planoperator
- Bestimmung der Reihenfolge bei Verbundoperationen
- Erkennen gemeinsamer Teilbäume

Nested-Loop-Verbund:

Es werden nacheinander alle Tupel aus der Relation R ausgewählt und mit jedem Tupel aus der Relation S verglichen. Erkennbar an Unterabfrage im WHERE.

Unnesting: Auflösen solcher Zustände. Hierbei wird eine Unteranfrage aus dem WHERE zu einer Unteranfrage aus dem FROM.

Beispiel:

```
SELECT s.name, n.pruefungsnr, n.note
FROM   Studierende s, Noten n
WHERE  s.matrikelnr = n.matrikelnr AND
       n.note = (SELECT min(n2.note)
                 FROM Noten n2
                 WHERE s.matrikelnr = n2.matrikelnr);
```

```
SELECT s.name, n.pruefungsnr, n.note
FROM   Studierende s, Noten n,
       (SELECT  n2.matrikelnr as matrikelnr,
                min(n2.note) as beste
        FROM    Noten n2
        GROUP BY n2.matrikelnr) m
WHERE  s.matrikelnr = n.matrikelnr AND
       s.matrikelnr = m.matrikelnr AND
       n.note = m.beste;
```

Komplexität: $O(n^2)$

Sort-Merge-Verbund

Vorgehensweise:

- Sortiere beide Tabellen nach einem Attribut
- Nehme Element aus erster Tabelle und Vergleiche es sequentiell mit den Elementen der zweiten Tabelle
- Wenn die Attributwerte gleich sind, Join die Tupel und speichere diese in extra Tabelle
- Wiederhole dies für alle anderen Elemente
- R1 noch R2 immer von vorne durchiterieren. Optimierung: Man startet bei den zuletzt gejointen Tupeln.

Vorgehen mit M Kacheln:

- Große Relation R in M-2 Listen sortierte Listen umwandeln M_n
- Sortiere die kleinere Relation S
- Lies kleinstes Element von M_n in die ersten M-2 Kacheln, dann das kleinste aus S. Die letzte freie Kachel ist für das Ergebnis
- Wähle kleinstes Element aus, aktualisiere ggf. in der Ergebniskachel (Vergleiche Element mit Element in S-Kachel) und ersetze es durch nächstes Element der Liste (Das kleinste ausgewählte Element kann auch von S sein, dann wird die S-Kachel aktualisiert)
- Arbeite damit alle Listen ab

Durch die Sortierung der beiden Tabellen, sinkt die Laufzeit von $O(n^2)$ zu $O(n\ log\ n)$.

Hash-Verbund

Warum: Großen Hauptspeicher ausnutzen

Ablauf:

- Gegeben: Zwei Relationen R und S
- Teile die kleinere Relation, hier R, in p Abschnitte, welche in den Hauptspeicher passen
- Äußere Schleife iteriert über Abschnitte p:
- Erzeuge eine Hashtabelle mit den Werten aus p
- Interne Schleife iteriert über S:
  - Hashe jeden Eintrag in S
  - Vergleiche die Einträge mit allen Einträgen im selben Bucket
  - Speichere Verbund in Ergebnisrelation

Komplexität:

- $O(p \cdot N)$ mit p Hash-Abschnitten
- Passt R ganz in den Hauptspeicher ist p = 1

Kosten

Kostenfaktoren:

- Blockzugriffe, CPU-Zeit, Speicherbedarf

Kosten ohne Pipelining:

- B(S) ist Anzahl der Blöcke der Relation S
- Selektion(S): B(S)
- Projektion(S): B(S)
- NestedLoopJoin(S,T): B(S) + B(S) * B(T)
- SortMergeJoin(S,T): 3*(B(S) + B(T))
- HashJoin(S,T): B(S) + B(T)

Kosten mit Pipelining:

- Für Nested-Loop und Hash-Verbund irrelevant!
- C(S) sind die Kosten zur Erzeugung der Zwischenergebnisrelation
- Selektion(S): C(S)
- Projektion(S): C(S)
- TableScan(S): B(S)
- IndexScan(S): a*B(S) * Selektivitätsfaktor
- NestedLoopJoin(S,T): C(S) + B(S) * C(T)
- SortMergeJoin(S,T): C(S) + C(T) + 2(B(S) + B(T))
- HashJoin(S,T): C(S) + C(T)

Bestimmung der Ausführungskosten:

- Benötigt: Kostenformeln der Operationen/ Scans
- Alle möglichen Formeln berechnen und billigste nehmen
- Bsp: Relation A und B mit B(S) + B(T)*B(S)
- 1. B(A) + B(B) * B(A)
- 2. B(B) + B(A) * B(B)

Schwierigkeiten der Optimierung:

Es gibt sehr viele mögliche Optionen, nicht alle können betrachtet werden. Zudem sind genaue Kosten oft nicht bekannt, weshalb geschätzt werden muss.

Ziel der Plangenerierung:

Die perfekte Lösung findet man zwar oftmals nicht, dennoch sollten extrem schlechte Lösungen vermieden werden. Die Optimierung soll schnell ablaufen.

Unterschiedliche Strategien: voll-enumerativ, beschränkt enumerativ, zufallsgesteuert.

Selektivität

- Statistiken müssen gesammelt und im DB Katalog gewartet werden
- Da man den DB Katalog nicht bei jeder Änderung aktualisieren kann (zu aufwändig), werden Statistiken bei der Erstellung einer Relation generiert und periodisch aktualisiert
- Der Selektivitätsfaktor ist ein Beispiel für eine solche Statistik, wird aus Abschätzungen errechnet und gibt an, ob sich Indexscans lohnen oder nicht

Grenztrefferrate und Planoperatoren

Indexstrukturen können die Laufzeit reduzieren, lohnen sich aber nur bei geringen Trefferrate.

Wenn die Grenztrefferrate über 5 % liegt, lohnt sich Indexscan nicht. Kleine Trefferrate = hohe Selektivität!

Grundlagen

Grundbegriffe

Datenbanksystem DBS:
System zur Beschreibung, Speicherung u. Wiedergewinnung von umfangreichen Datenmengen, die von mehreren Anwendungsprogrammen benutzt werden.

Datenbank DB:
Komponente des DBS, in der Daten abgelegt werden

Datenbank-Verwaltungssystem DBVS:
Softwäre zur Verwaltung der Daten einer Datenbank

SQL

Grundlagen:
- **SELECT**: Projektion, ein Teil einer Relation (z.b. eine Spalte)
- **WHERE, HAVING**: Selektion, nur bestimmte Tupel (z.b. einzelne Spalten)
- **FROM:** Wählt eine Tabelle aus, oder bildet das Kreuzprodukt mehrerer Tabellen
- **JOIN:** Fässt zwei Tabellen nach einem Attribut zusammen
- **GROUP / ORDER BY**: Sortiert eine Relation nach Attributwerten
- **Weitere Operationen:** MIN, MAX, COUNT, AVG, SUM

Abarbeitunsreihenfolge SQL Statement:
FROM ⇒ WHERE ⇒ GROUP BY ⇒
HAVING ⇒ SELECT ⇒ ORDER BY

Beispiele:
songs(title, artist, length, album[albums])
producers(name, birthdate)
albums(title, producer[producers], release, genre)

Titel aller Songs von Ray kürzer als 180 sec:
SELECT title FROM songs WHERE artist = 'Ray' AND length <= 180 ;

Gebe Songtitel mit Geburtstag des Produzenten des Albums aus, aus dem der Song ist:
SELECT s.title , p.birthdate
FROM songs s, producers p, albums a
WHERE s. album = a. title AND a. producer = p. name ;

Gebe eine aufsteigend nach Künstlername sortierte Liste der Künstler und der Anzahl an Songs zurückgibt. Berücksichtige nur Songs, die auf einem Album erschienen sind und nur Künstler mit mindestens 5 solcher Songs.
SELECT artist , count (album) AS anzahl
FROM songs
GROUP BY artist
HAVING count (album) >= 5
ORDER BY artist ASC ;

Schicksenmodell